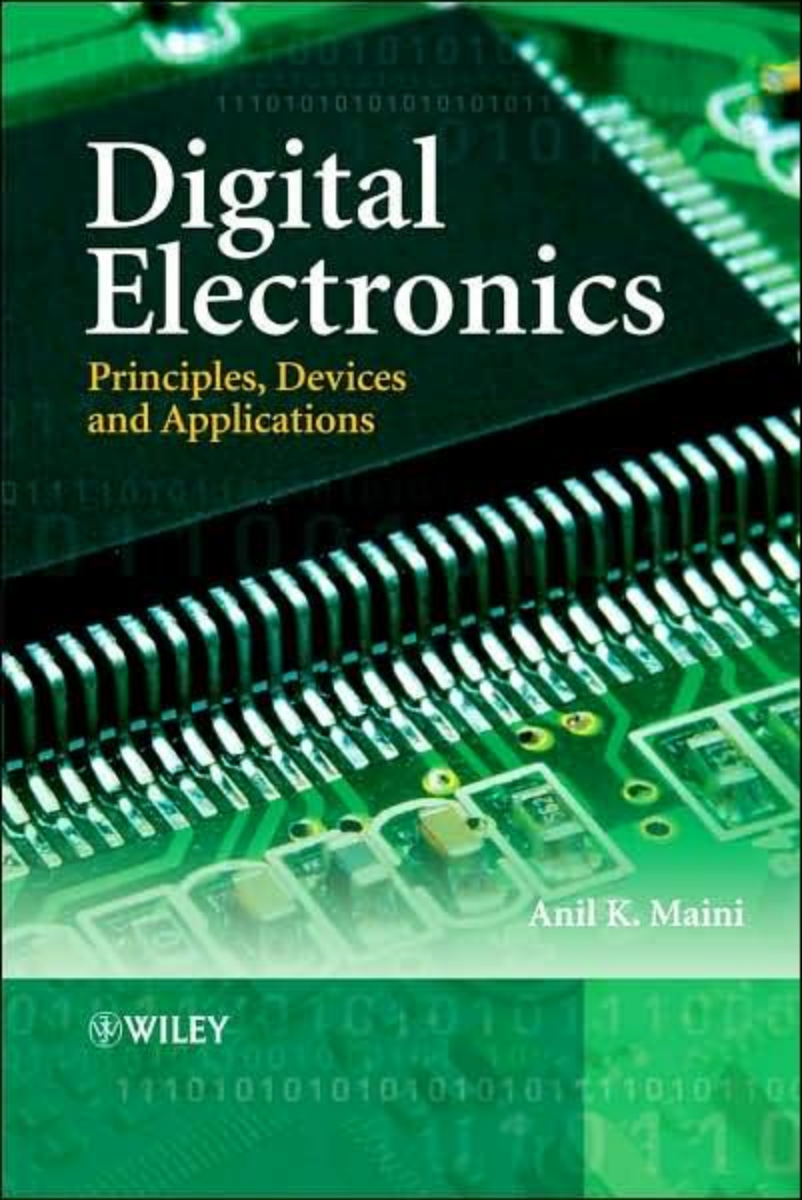

Digital
Electronics
Principles, Devices
and Applications
Anil K. Maini
WILEY

Preface

Digital electronics is essential to understanding the design and working of a wide range of applications, from consumer and industrial electronics to communications; from embedded systems, and computers to security and military equipment. As the devices used in these applications decrease in size and employ more complex technology, it is essential for engineers and students to fully understand both the fundamentals and also the implementation and application principles of digital electronics, devices and integrated circuits, thus enabling them to use the most appropriate and effective technique to suit their technical needs.

Digital Electronics: Principles, Devices and Applications is a comprehensive book covering, in one volume, both the fundamentals of digital electronics and the applications of digital devices and integrated circuits. It is different from similar books on the subject in more than one way. Each chapter in the book, whether it is related to operational fundamentals or applications, is amply illustrated with diagrams and design examples. In addition, the book covers several new topics, which are of relevance to any one having an interest in digital electronics and not covered in the books already in print on the subject. These include digital troubleshooting, digital instrumentation, programmable logic devices, microprocessors and microcontrollers. While the book covers in entirety what is required by undergraduate and graduate level students of engineering in electrical, electronics, computer science and information technology disciplines, it is intended to be a very useful reference book for professionals, R&D scientists and students at post graduate level.

The book is divided into sixteen chapters covering seven major topics. These are: *digital electronics fundamentals* (chapters 1 to 6), *combinational logic circuits* (chapters 7 and 8), *programmable logic devices* (chapter 9), *sequential logic circuits* (chapters 10 and 11), *data conversion devices and circuits* (chapter 12), *microprocessors, microcontrollers and microcomputers* (chapters 13 to 15) and *digital troubleshooting and instrumentation* (chapter 16). The contents of each of the sixteen chapters are briefly described in the following paragraphs.

The first six chapters deal with the fundamental topics of digital electronics. These include different number systems that can be used to represent data and binary codes used for representing numeric and alphanumeric data. Conversion from one number system to another and similarly conversion from one code to another is discussed at length in these chapters. Binary arithmetic, covering different methods of performing arithmetic operations on binary numbers is discussed next. Chapters four and five cover logic gates and logic families. The main topics covered in these two chapters are various logic gates and related devices, different logic families used to hardware implement digital integrated circuits, the interface between digital ICs belonging to different logic families and application information such

as guidelines for using logic devices of different families. Boolean algebra and its various postulates and theorems and minimization techniques, providing exhaustive coverage of both Karnaugh mapping and Quine-McCluskey techniques, are discussed in chapter six. The discussion includes application of these minimization techniques for multi-output Boolean functions and Boolean functions with larger number of variables. The concepts underlying different fundamental topics of digital electronics and discussed in first six chapters have been amply illustrated with solved examples.

As a follow-up to logic gates – the most basic building block of combinational logic – chapters 7 and 8 are devoted to more complex combinational logic circuits. While chapter seven covers arithmetic circuits, including different types of adders and subtractors, such as half and full adder and subtractor, adder-subtractor, larger bit adders and subtractors, multipliers, look ahead carry generator, magnitude comparator, and arithmetic logic unit, chapter eight covers multiplexers, de-multiplexers, encoders and decoders. This is followed by a detailed account of programmable logic devices in chapter nine. Simple programmable logic devices (SPLDs) such as PAL, PLA, GAL and HAL devices, complex programmable logic devices (CPLDs) and field programmable gate arrays (FPGAs) have been exhaustively treated in terms of their architecture, features and applications. Popular devices, from various international manufacturers, in the three above-mentioned categories of programmable logic devices are also covered with regard to their architecture, features and facilities.

The next two chapters, 10 and 11, cover the sequential logic circuits. Discussion begins with the most fundamental building block of sequential logic, that is, *flip flop*. Different types of flip flops are covered in detail with regard to their operational fundamentals, different varieties in each of the categories of flip flops and their applications. Multivibrator circuits, being operationally similar to flip flops, are also covered at length in this chapter. Counters and registers are the other very important building blocks of sequential logic with enormous application potential. These are covered in chapter 11. Particular emphasis is given to timing requirements and design of counters with varying count sequence requirements. The chapter also includes a detailed description of the design principles of counters with arbitrary count sequences. Different types of shift registers and some special counters that have evolved out of shift registers have been covered in detail.

Chapter 12 covers data conversion circuits including digital-to-analogue and analogue-to-digital converters. Topics covered in this chapter include operational basics, characteristic parameters, types and applications. Emphasis is given to definition and interpretation of the terminology and the performance parameters that characterize these devices. Different types of digital-to-analogue and analogue-to-digital converters, together with their merits and drawbacks are also addressed. Particular attention is given to their applications. Towards the end of the chapter, application oriented information in the form of popular type numbers along with their major performance specifications, pin connection diagrams etc. is presented. Another highlight of the chapter is the inclusion of detailed descriptions of newer types of converters, such as quad slope and sigma-delta types of analogue-to-digital converters.

Chapters 13 and 14 discuss microprocessors and microcontrollers – the two versatile devices that have revolutionized the application potential of digital devices and integrated circuits. The entire range of microprocessors and microcontrollers along with their salient features, operational aspects and application guidelines are covered in detail. As a natural follow-up to these, microcomputer fundamentals, with regard to their architecture, input/output devices and memory devices, are discussed in chapter 15.

The last chapter covers digital troubleshooting techniques and digital instrumentation. Troubleshooting guidelines for various categories of digital electronics circuits are discussed. These will particularly benefit practising engineers and electronics enthusiasts. The concepts are illustrated with the help of a large number of troubleshooting case studies pertaining to combinational, sequential and memory devices. A wide range of digital instruments is covered after a discussion on troubleshooting guidelines. The instruments covered include digital multimeters, digital oscilloscopes, logic probes,

logic analysers, frequency synthesizers, and synthesized function generators. Computer-instrument interface standards and the concept of virtual instrumentation are also discussed at length towards the end of the chapter.

As an extra resource, a companion website for my book contains lot of additional application relevant information on digital devices and integrated circuits. The information on this website includes numerical and functional indices of digital integrated circuits belonging to different logic families, pin connection diagrams and functional tables of different categories of general purpose digital integrated circuits and application relevant information on microprocessors, peripheral devices and microcontrollers. Please go to URL http://www.wiley.com/go/maini_digital.

The motivation to write this book and the selection of topics to be covered were driven mainly by the absence a book, which, in one volume, covers all the important aspects of digital technology. A large number of books in print on the subject cover all the routine topics of digital electronics in a conventional way with total disregard to the needs of application engineers and professionals. As the author, I have made an honest attempt to cover the subject in entirety by including comprehensive treatment of newer topics that are either ignored or inadequately covered in the available books on the subject of digital electronics. This is done keeping in view the changed requirements of my intended audience, which includes undergraduate and graduate level students, R&D scientists, professionals and application engineers.

Anil K. Maini

1

Number Systems

The study of *number systems* is important from the viewpoint of understanding how data are represented before they can be processed by any digital system including a digital computer. It is one of the most basic topics in digital electronics. In this chapter we will discuss different number systems commonly used to represent data. We will begin the discussion with the decimal number system. Although it is not important from the viewpoint of digital electronics, a brief outline of this will be given to explain some of the underlying concepts used in other number systems. This will then be followed by the more commonly used number systems such as the binary, octal and hexadecimal number systems.

1.1 Analogue Versus Digital

There are two basic ways of representing the numerical values of the various physical quantities with which we constantly deal in our day-to-day lives. One of the ways, referred to as *analogue*, is to express the numerical value of the quantity as a continuous range of values between the two expected extreme values. For example, the temperature of an oven settable anywhere from 0 to 100 °C may be measured to be 65 °C or 64.96 °C or 64.958 °C or even 64.9579 °C and so on, depending upon the accuracy of the measuring instrument. Similarly, voltage across a certain component in an electronic circuit may be measured as 6.5 V or 6.49 V or 6.487 V or 6.4869 V. The underlying concept in this mode of representation is that variation in the numerical value of the quantity is continuous and could have any of the infinite theoretically possible values between the two extremes.

The other possible way, referred to as *digital*, represents the numerical value of the quantity in steps of discrete values. The numerical values are mostly represented using binary numbers. For example, the temperature of the oven may be represented in steps of 1 °C as 64 °C, 65 °C, 66 °C and so on. To summarize, while an analogue representation gives a continuous output, a digital representation produces a discrete output. Analogue systems contain devices that process or work on various physical quantities represented in analogue form. Digital systems contain devices that process the physical quantities represented in digital form.

Digital Electronics: Principles, Devices and Applications Anil K. Maini
 ISBN: 978-0-470-03214-5

Digital techniques and systems have the advantages of being relatively much easier to design and having higher accuracy, programmability, noise immunity, easier storage of data and ease of fabrication in integrated circuit form, leading to availability of more complex functions in a smaller size. The real world, however, is analogue. Most physical quantities – position, velocity, acceleration, force, pressure, temperature and flowrate, for example – are analogue in nature. That is why analogue variables representing these quantities need to be digitized or discretized at the input if we want to benefit from the features and facilities that come with the use of digital techniques. In a typical system dealing with analogue inputs and outputs, analogue variables are digitized at the input with the help of an analogue-to-digital converter block and reconverted back to analogue form at the output using a digital-to-analogue converter block. Analogue-to-digital and digital-to-analogue converter circuits are discussed at length in the latter part of the book. In the following sections we will discuss various number systems commonly used for digital representation of data.

1.2 Introduction to Number Systems

We will begin our discussion on various number systems by briefly describing the parameters that are common to all number systems. An understanding of these parameters and their relevance to number systems is fundamental to the understanding of how various systems operate. Different characteristics that define a number system include the number of independent digits used in the number system, the place values of the different digits constituting the number and the maximum numbers that can be written with the given number of digits. Among the three characteristic parameters, the most fundamental is the number of independent digits or symbols used in the number system. It is known as the *radix* or *base* of the number system. The decimal number system with which we are all so familiar can be said to have a radix of 10 as it has 10 independent digits, i.e. 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9. Similarly, the binary number system with only two independent digits, 0 and 1, is a radix-2 number system. The octal and hexadecimal number systems have a radix (or base) of 8 and 16 respectively. We will see in the following sections that the radix of the number system also determines the other two characteristics. The place values of different digits in the integer part of the number are given by r^0, r^1, r^2, r^3 and so on, starting with the digit adjacent to the radix point. For the fractional part, these are r^{-1}, r^{-2}, r^{-3} and so on, again starting with the digit next to the radix point. Here, r is the radix of the number system. Also, maximum numbers that can be written with n digits in a given number system are equal to r^n.

1.3 Decimal Number System

The decimal number system is a radix-10 number system and therefore has 10 different digits or symbols. These are 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9. All higher numbers after '9' are represented in terms of these 10 digits only. The process of writing higher-order numbers after '9' consists in writing the second digit (i.e. '1') first, followed by the other digits, one by one, to obtain the next 10 numbers from '10' to '19'. The next 10 numbers from '20' to '29' are obtained by writing the third digit (i.e. '2') first, followed by digits '0' to '9', one by one. The process continues until we have exhausted all possible two-digit combinations and reached '99'. Then we begin with three-digit combinations. The first three-digit number consists of the lowest two-digit number followed by '0' (i.e. 100), and the process goes on endlessly.

The place values of different digits in a mixed decimal number, starting from the decimal point, are 10^0, 10^1, 10^2 and so on (for the integer part) and 10^{-1}, 10^{-2}, 10^{-3} and so on (for the fractional part).

The value or magnitude of a given decimal number can be expressed as the sum of the various digits multiplied by their place values or weights.

As an illustration, in the case of the decimal number 3586.265, the integer part (i.e. 3586) can be expressed as

$$3586 = 6 \times 10^0 + 8 \times 10^1 + 5 \times 10^2 + 3 \times 10^3 = 6 + 80 + 500 + 3000 = 3586$$

and the fractional part can be expressed as

$$265 = 2 \times 10^{-1} + 6 \times 10^{-2} + 5 \times 10^{-3} = 0.2 + 0.06 + 0.005 = 0.265$$

We have seen that the place values are a function of the radix of the concerned number system and the position of the digits. We will also discover in subsequent sections that the concept of each digit having a place value depending upon the position of the digit and the radix of the number system is equally valid for the other more relevant number systems.

1.4 Binary Number System

The binary number system is a radix-2 number system with '0' and '1' as the two independent digits. All larger binary numbers are represented in terms of '0' and '1'. The procedure for writing higher-order binary numbers after '1' is similar to the one explained in the case of the decimal number system. For example, the first 16 numbers in the binary number system would be 0, 1, 10, 11, 100, 101, 110, 111, 1000, 1001, 1010, 1011, 1100, 1101, 1110 and 1111. The next number after 1111 is 10000, which is the lowest binary number with five digits. This also proves the point made earlier that a maximum of only 16 ($= 2^4$) numbers could be written with four digits. Starting from the binary point, the place values of different digits in a mixed binary number are 2^0, 2^1, 2^2 and so on (for the integer part) and 2^{-1}, 2^{-2}, 2^{-3} and so on (for the fractional part).

Example 1.1

Consider an arbitrary number system with the independent digits as 0, 1 and X. What is the radix of this number system? List the first 10 numbers in this number system.

Solution

- The radix of the proposed number system is 3.
- The first 10 numbers in this number system would be 0, 1, X, 10, 11, 1X, X0, X1, XX and 100.

1.4.1 Advantages

Logic operations are the backbone of any digital computer, although solving a problem on computer could involve an arithmetic operation too. The introduction of the mathematics of logic by George Boole laid the foundation for the modern digital computer. He reduced the mathematics of logic to a binary notation of '0' and '1'. As the mathematics of logic was well established and had proved itself to be quite useful in solving all kinds of logical problem, and also as the mathematics of logic (also known as Boolean algebra) had been reduced to a binary notation, the binary number system had a clear edge over other number systems for use in computer systems.

Yet another significant advantage of this number system was that all kinds of data could be conveniently represented in terms of 0s and 1s. Also, basic electronic devices used for hardware implementation could be conveniently and efficiently operated in two distinctly different modes. For example, a bipolar transistor could be operated either in cut-off or in saturation very efficiently.

Lastly, the circuits required for performing arithmetic operations such as addition, subtraction, multiplication, division, etc., become a simple affair when the data involved are represented in the form of 0s and 1s.

1.5 Octal Number System

The octal number system has a radix of 8 and therefore has eight distinct digits. All higher-order numbers are expressed as a combination of these on the same pattern as the one followed in the case of the binary and decimal number systems described in Sections 1.3 and 1.4. The independent digits are 0, 1, 2, 3, 4, 5, 6 and 7. The next 10 numbers that follow '7', for example, would be 10, 11, 12, 13, 14, 15, 16, 17, 20 and 21. In fact, if we omit all the numbers containing the digits 8 or 9, or both, from the decimal number system, we end up with an octal number system. The place values for the different digits in the octal number system are 8^0, 8^1, 8^2 and so on (for the integer part) and 8^{-1}, 8^{-2}, 8^{-3} and so on (for the fractional part).

1.6 Hexadecimal Number System

The hexadecimal number system is a radix-16 number system and its 16 basic digits are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E and F. The place values or weights of different digits in a mixed hexadecimal number are 16^0, 16^1, 16^2 and so on (for the integer part) and 16^{-1}, 16^{-2}, 16^{-3} and so on (for the fractional part). The decimal equivalent of A, B, C, D, E and F are 10, 11, 12, 13, 14 and 15 respectively, for obvious reasons.

The hexadecimal number system provides a condensed way of representing large binary numbers stored and processed inside the computer. One such example is in representing addresses of different memory locations. Let us assume that a machine has 64K of memory. Such a memory has 64K ($= 2^{16}$ $= 65\,536$) memory locations and needs 65 536 different addresses. These addresses can be designated as 0 to 65 535 in the decimal number system and 00000000 00000000 to 11111111 11111111 in the binary number system. The decimal number system is not used in computers and the binary notation here appears too cumbersome and inconvenient to handle. In the hexadecimal number system, 65 536 different addresses can be expressed with four digits from 0000 to FFFF. Similarly, the contents of the memory when represented in hexadecimal form are very convenient to handle.

1.7 Number Systems – Some Common Terms

In this section we will describe some commonly used terms with reference to different number systems.

1.7.1 Binary Number System

Bit is an abbreviation of the term 'binary digit' and is the smallest unit of information. It is either '0' or '1'. A *byte* is a string of eight bits. The byte is the basic unit of data operated upon as a single unit in computers. A *computer word* is again a string of bits whose size, called the 'word length' or 'word size', is fixed for a specified computer, although it may vary from computer to computer. The word length may equal one byte, two bytes, four bytes or be even larger.

The *1's complement* of a binary number is obtained by complementing all its bits, i.e. by replacing 0s with 1s and 1s with 0s. For example, the 1's complement of $(10010110)_2$ is $(01101001)_2$. The *2's complement* of a binary number is obtained by adding '1' to its 1's complement. The 2's complement of $(10010110)_2$ is $(01101010)_2$.

1.7.2 Decimal Number System

Corresponding to the 1's and 2's complements in the binary system, in the decimal number system we have the 9's and 10's complements. The *9's complement* of a given decimal number is obtained by subtracting each digit from 9. For example, the 9's complement of $(2496)_{10}$ would be $(7503)_{10}$. The *10's complement* is obtained by adding '1' to the 9's complement. The 10's complement of $(2496)_{10}$ is $(7504)_{10}$.

1.7.3 Octal Number System

In the octal number system, we have the 7's and 8's complements. The *7's complement* of a given octal number is obtained by subtracting each octal digit from 7. For example, the 7's complement of $(562)_8$ would be $(215)_8$. The *8's complement* is obtained by adding '1' to the 7's complement. The 8's complement of $(562)_8$ would be $(216)_8$.

1.7.4 Hexadecimal Number System

The 15's and 16's complements are defined with respect to the hexadecimal number system. The *15's complement* is obtained by subtracting each hex digit from 15. For example, the 15's complement of $(3BF)_{16}$ would be $(C40)_{16}$. The *16's complement* is obtained by adding '1' to the 15's complement. The 16's complement of $(2AE)_{16}$ would be $(D52)_{16}$.

1.8 Number Representation in Binary

Different formats used for binary representation of both positive and negative decimal numbers include the sign-bit magnitude method, the 1's complement method and the 2's complement method.

1.8.1 Sign-Bit Magnitude

In the sign-bit magnitude representation of positive and negative decimal numbers, the MSB represents the 'sign', with a '0' denoting a plus sign and a '1' denoting a minus sign. The remaining bits represent the magnitude. In eight-bit representation, while MSB represents the sign, the remaining seven bits represent the magnitude. For example, the eight-bit representation of $+9$ would be 00001001, and that for -9 would be 10001001. An $n-$bit binary representation can be used to represent decimal numbers in the range of $-(2^{n-1}-1)$ to $+(2^{n-1}-1)$. That is, eight-bit representation can be used to represent decimal numbers in the range from -127 to $+127$ using the sign-bit magnitude format.

1.8.2 1's Complement

In the 1's complement format, the positive numbers remain unchanged. The negative numbers are obtained by taking the 1's complement of the positive counterparts. For example, +9 will be represented as 00001001 in eight-bit notation, and −9 will be represented as 11110110, which is the 1's complement of 00001001. Again, n-bit notation can be used to represent numbers in the range from $-(2^{n-1}-1)$ to $+(2^{n-1}-1)$ using the 1's complement format. The eight-bit representation of the 1's complement format can be used to represent decimal numbers in the range from −127 to +127.

1.8.3 2's Complement

In the 2's complement representation of binary numbers, the MSB represents the sign, with a '0' used for a plus sign and a '1' used for a minus sign. The remaining bits are used for representing magnitude. Positive magnitudes are represented in the same way as in the case of sign-bit or 1's complement representation. Negative magnitudes are represented by the 2's complement of their positive counterparts. For example, +9 would be represented as 00001001, and −9 would be written as 11110111. Please note that, if the 2's complement of the magnitude of +9 gives a magnitude of −9, then the reverse process will also be true, i.e. the 2's complement of the magnitude of −9 will give a magnitude of +9. The n-bit notation of the 2's complement format can be used to represent all decimal numbers in the range from $+(2^{n-1}-1)$ to $-(2^{n-1})$. The 2's complement format is very popular as it is very easy to generate the 2's complement of a binary number and also because arithmetic operations are relatively easier to perform when the numbers are represented in the 2's complement format.

1.9 Finding the Decimal Equivalent

The decimal equivalent of a given number in another number system is given by the sum of all the digits multiplied by their respective place values. The integer and fractional parts of the given number should be treated separately. Binary-to-decimal, octal-to-decimal and hexadecimal-to-decimal conversions are illustrated below with the help of examples.

1.9.1 Binary-to-Decimal Conversion

The decimal equivalent of the binary number $(1001.0101)_2$ is determined as follows:

- The integer part = 1001
- The decimal equivalent $= 1 \times 2^0 + 0 \times 2^1 + 0 \times 2^2 + 1 \times 2^3 = 1 + 0 + 0 + 8 = 9$
- The fractional part = .0101
- Therefore, the decimal equivalent $= 0 \times 2^{-1} + 1 \times 2^{-2} + 0 \times 2^{-3} + 1 \times 2^{-4} = 0 + 0.25 + 0 + 0.0625 = 0.3125$
- Therefore, the decimal equivalent of $(1001.0101)_2 = 9.3125$

1.9.2 Octal-to-Decimal Conversion

The decimal equivalent of the octal number $(137.21)_8$ is determined as follows:

- The integer part = 137
- The decimal equivalent $= 7 \times 8^0 + 3 \times 8^1 + 1 \times 8^2 = 7 + 24 + 64 = 95$

- The fractional part = .21
- The decimal equivalent = $2 \times 8^{-1} + 1 \times 8^{-2} = 0.265$
- Therefore, the decimal equivalent of $(137.21)_8 = (95.265)_{10}$

1.9.3 Hexadecimal-to-Decimal Conversion

The decimal equivalent of the hexadecimal number $(1E0.2A)_{16}$ is determined as follows:

- The integer part = 1E0
- The decimal equivalent = $0 \times 16^0 + 14 \times 16^1 + 1 \times 16^2 = 0 + 224 + 256 = 480$
- The fractional part = 2A
- The decimal equivalent = $2 \times 16^{-1} + 10 \times 16^{-2} = 0.164$
- Therefore, the decimal equivalent of $(1E0.2A)_{16} = (480.164)_{10}$

Example 1.2

Find the decimal equivalent of the following binary numbers expressed in the 2's complement format:

(a) 00001110;
(b) 10001110.

Solution

(a) The MSB bit is '0', which indicates a plus sign.
The magnitude bits are 0001110.
The decimal equivalent $= 0\times2^0+1\times2^1+1\times2^2+1\times2^3+0\times2^4+0\times2^5+0\times2^6$
$= 0+2+4+8+0+0+0 = 14$

Therefore, 00001110 represents +14

(b) The MSB bit is '1', which indicates a minus sign
The magnitude bits are therefore given by the 2's complement of 0001110, i.e. 1110010
The decimal equivalent $= 0\times2^0+1\times2^1+0\times2^2+0\times2^3+1\times2^4+1\times2^5$
$+1\times2^6$
$= 0+2+0+0+16+32+64 = 114$

Therefore, 10001110 represents −114

1.10 Decimal-to-Binary Conversion

As outlined earlier, the integer and fractional parts are worked on separately. For the integer part, the binary equivalent can be found by successively dividing the integer part of the number by 2 and recording the remainders until the quotient becomes '0'. The remainders written in reverse order constitute the binary equivalent. For the fractional part, it is found by successively multiplying the fractional part of the decimal number by 2 and recording the carry until the result of multiplication is '0'. The carry sequence written in forward order constitutes the binary equivalent of the fractional

part of the decimal number. If the result of multiplication does not seem to be heading towards zero in the case of the fractional part, the process may be continued only until the requisite number of equivalent bits has been obtained. This method of decimal–binary conversion is popularly known as the double-dabble method. The process can be best illustrated with the help of an example.

Example 1.3

We will find the binary equivalent of $(13.375)_{10}$.

Solution

- The integer part = 13

Divisor	Dividend	Remainder
2	13	—
2	6	1
2	3	0
2	1	1
—	0	1

- The binary equivalent of $(13)_{10}$ is therefore $(1101)_2$
- The fractional part = .375
- $0.375 \times 2 = 0.75$ with a carry of 0
- $0.75 \times 2 = 0.5$ with a carry of 1
- $0.5 \times 2 = 0$ with a carry of 1
- The binary equivalent of $(0.375)_{10} = (.011)_2$
- Therefore, the binary equivalent of $(13.375)_{10} = (1101.011)_2$

1.11 Decimal-to-Octal Conversion

The process of decimal-to-octal conversion is similar to that of decimal-to-binary conversion. The progressive division in the case of the integer part and the progressive multiplication while working on the fractional part here are by '8' which is the radix of the octal number system. Again, the integer and fractional parts of the decimal number are treated separately. The process can be best illustrated with the help of an example.

Example 1.4

We will find the octal equivalent of $(73.75)_{10}$.

Solution

- The integer part = 73

Divisor	Dividend	Remainder
8	73	—
8	9	1
8	1	1
—	0	1

- The octal equivalent of $(73)_{10} = (111)_8$
- The fractional part = 0.75
- $0.75 \times 8 = 0$ with a carry of 6
- The octal equivalent of $(0.75)_{10} = (.6)_8$
- Therefore, the octal equivalent of $(73.75)_{10} = (111.6)_8$

1.12 Decimal-to-Hexadecimal Conversion

The process of decimal-to-hexadecimal conversion is also similar. Since the hexadecimal number system has a base of 16, the progressive division and multiplication factor in this case is 16. The process is illustrated further with the help of an example.

Example 1.5

Let us determine the hexadecimal equivalent of $(82.25)_{10}$.

Solution

- The integer part = 82

Divisor	Dividend	Remainder
16	82	—
16	5	2
—	0	5

- The hexadecimal equivalent of $(82)_{10} = (52)_{16}$
- The fractional part = 0.25
- $0.25 \times 16 = 0$ with a carry of 4
- Therefore, the hexadecimal equivalent of $(82.25)_{10} = (52.4)_{16}$

1.13 Binary–Octal and Octal–Binary Conversions

An octal number can be converted into its binary equivalent by replacing each octal digit with its three-bit binary equivalent. We take the three-bit equivalent because the base of the octal number system is 8 and it is the third power of the base of the binary number system, i.e. 2. All we have then to remember is the three-bit binary equivalents of the basic digits of the octal number system. A binary number can be converted into an equivalent octal number by splitting the integer and fractional parts into groups of three bits, starting from the binary point on both sides. The 0s can be added to complete the outside groups if needed.

Example 1.6

Let us find the binary equivalent of $(374.26)_8$ and the octal equivalent of $(1110100.0100111)_2$.

Solution

- The given octal number = $(374.26)_8$
- The binary equivalent = $(011\ 111\ 100.010\ 110)_2 = (011111100.010110)_2$

- Any 0s on the extreme left of the integer part and extreme right of the fractional part of the equivalent binary number should be omitted. Therefore, $(011111100.010110)_2 = (11111100.01011)_2$
- The given binary number $= (1110100.0100111)_2$
- $(1110100.0100111)_2 = (1\ 110\ 100.010\ 011\ 1)_2$
  $= (001\ 110\ 100.010\ 011\ 100)_2 = (164.234)_8$

1.14 Hex–Binary and Binary–Hex Conversions

A hexadecimal number can be converted into its binary equivalent by replacing each hex digit with its four-bit binary equivalent. We take the four-bit equivalent because the base of the hexadecimal number system is 16 and it is the fourth power of the base of the binary number system. All we have then to remember is the four-bit binary equivalents of the basic digits of the hexadecimal number system. A given binary number can be converted into an equivalent hexadecimal number by splitting the integer and fractional parts into groups of four bits, starting from the binary point on both sides. The 0s can be added to complete the outside groups if needed.

Example 1.7

Let us find the binary equivalent of $(17E.F6)_{16}$ and the hex equivalent of $(1011001110.011011101)_2$.

Solution

- The given hex number $= (17E.F6)_{16}$
- The binary equivalent $= (0001\ 0111\ 1110.1111\ 0110)_2$
  $= (000101111110.11110110)_2$
  $= (101111110.1111011)_2$
- The 0s on the extreme left of the integer part and on the extreme right of the fractional part have been omitted.
- The given binary number $= (1011001110.011011101)_2$
  $= (10\ 1100\ 1110.0110\ 1110\ 1)_2$
- The hex equivalent $= (0010\ 1100\ 1110.0110\ 1110\ 1000)_2 = (2CE.6E8)_{16}$

1.15 Hex–Octal and Octal–Hex Conversions

For hexadecimal–octal conversion, the given hex number is firstly converted into its binary equivalent which is further converted into its octal equivalent. An alternative approach is firstly to convert the given hexadecimal number into its decimal equivalent and then convert the decimal number into an equivalent octal number. The former method is definitely more convenient and straightforward. For octal–hexadecimal conversion, the octal number may first be converted into an equivalent binary number and then the binary number transformed into its hex equivalent. The other option is firstly to convert the given octal number into its decimal equivalent and then convert the decimal number into its hex equivalent. The former approach is definitely the preferred one. Two types of conversion are illustrated in the following example.

Example 1.8

Let us find the octal equivalent of $(2F.C4)_{16}$ and the hex equivalent of $(762.013)_8$.

Solution

- The given hex number $= (2F.C4)_{16}$.
- The binary equivalent $= (0010\ 1111.1100\ 0100)_2 = (00101111.11000100)_2$
  $= (101111.110001)_2 = (101\ 111.110\ 001)_2 = (57.61)_8$.
- The given octal number $= (762.013)_8$.
- The octal number $= (762.013)_8 = (111\ 110\ 010.000\ 001\ 011)_2$
  $= (111110010.000001011)_2$
  $= (0001\ 1111\ 0010.0000\ 0101\ 1000)_2 = (1F2.058)_{16}$.

1.16 The Four Axioms

Conversion of a given number in one number system to its equivalent in another system has been discussed at length in the preceding sections. The methodology has been illustrated with solved examples. The complete methodology can be summarized as four axioms or principles, which, if understood properly, would make it possible to solve any problem related to conversion of a given number in one number system to its equivalent in another number system. These principles are as follows:

1. Whenever it is desired to find the decimal equivalent of a given number in another number system, it is given by the sum of all the digits multiplied by their weights or place values. The integer and fractional parts should be handled separately. Starting from the radix point, the weights of different digits are r^0, r^1, r^2 for the integer part and r^{-1}, r^{-2}, r^{-3} for the fractional part, where r is the radix of the number system whose decimal equivalent needs to be determined.
2. To convert a given mixed decimal number into an equivalent in another number system, the integer part is progressively divided by r and the remainders noted until the result of division yields a zero quotient. The remainders written in reverse order constitute the equivalent. r is the radix of the transformed number system. The fractional part is progressively multiplied by r and the carry recorded until the result of multiplication yields a zero or when the desired number of bits has been obtained. The carrys written in forward order constitute the equivalent of the fractional part.
3. The octal–binary conversion and the reverse process are straightforward. For octal–binary conversion, replace each digit in the octal number with its three-bit binary equivalent. For hexadecimal–binary conversion, replace each hex digit with its four-bit binary equivalent. For binary–octal conversion, split the binary number into groups of three bits, starting from the binary point, and, if needed, complete the outside groups by adding 0s, and then write the octal equivalent of these three-bit groups. For binary–hex conversion, split the binary number into groups of four bits, starting from the binary point, and, if needed, complete the outside groups by adding 0s, and then write the hex equivalent of the four-bit groups.
4. For octal–hexadecimal conversion, we can go from the given octal number to its binary equivalent and then from the binary equivalent to its hex counterpart. For hexadecimal–octal conversion, we can go from the hex to its binary equivalent and then from the binary number to its octal equivalent.

Example 1.9

Assume an arbitrary number system having a radix of 5 and 0, 1, 2, L and M as its independent digits. Determine:

(a) the decimal equivalent of (12LM.L1);
(b) the total number of possible four-digit combinations in this arbitrary number system.

Solution

(a) The decimal equivalent of (12LM) is given by

$$M \times 5^0 + L \times 5^1 + 2 \times 5^2 + 1 \times 5^3 = 4 \times 5^0 + 3 \times 5^1 + 2 \times 5^2 + 1 \times 5^3 (L = 3, M = 4)$$
$$= 4 + 15 + 50 + 125 = 194$$

The decimal equivalent of (L1) is given by

$$L \times 5^{-1} + 1 \times 5^{-2} = 3 \times 5^{-1} + 5^{-2} = 0.64$$

Combining the results, $(12LM.L1)_5 = (194.64)_{10}$.

(b) The total number of possible four-digit combinations $= 5^4 = 625$.

Example 1.10

The 7's complement of a certain octal number is 5264. Determine the binary and hexadecimal equivalents of that octal number.

Solution

- The 7's complement = 5264.
- Therefore, the octal number $= (2513)_8$.
- The binary equivalent $= (010\ 101\ 001\ 011)_2 = (10101001011)_2$.
- Also, $(10101001011)_2 = (101\ 0100\ 1011)_2 = (0101\ 0100\ 1011)_2 = (54B)_{16}$.
- Therefore, the hex equivalent of $(2513)_8 = (54B)_{16}$ and the binary equivalent of $(2513)_8 = (10101001011)_2$.

1.17 Floating-Point Numbers

Floating-point notation can be used conveniently to represent both large as well as small fractional or mixed numbers. This makes the process of arithmetic operations on these numbers relatively much easier. Floating-point representation greatly increases the range of numbers, from the smallest to the largest, that can be represented using a given number of digits. Floating-point numbers are in general expressed in the form

$$N = m \times b^e \tag{1.1}$$

where m is the fractional part, called the *significand* or *mantissa*, e is the integer part, called the *exponent*, and b is the *base* of the number system or numeration. Fractional part m is a p-digit number of the form $(\pm d.ddd \ldots dd)$, with each digit d being an integer between 0 and $b - 1$ inclusive. If the leading digit of m is nonzero, then the number is said to be normalized.

Equation (1.1) in the case of decimal, hexadecimal and binary number systems will be written as follows:

Decimal system

$$N = m \times 10^e \tag{1.2}$$

Hexadecimal system

$$N = m \times 16^e \tag{1.3}$$

Binary system

$$N = m \times 2^e \tag{1.4}$$

For example, decimal numbers 0.0003754 and 3754 will be represented in floating-point notation as 3.754×10^{-4} and 3.754×10^3 respectively. A hex number 257.ABF will be represented as $2.57\text{ABF} \times 16^2$. In the case of normalized binary numbers, the leading digit, which is the most significant bit, is always '1' and thus does not need to be stored explicitly.

Also, while expressing a given mixed binary number as a floating-point number, the radix point is so shifted as to have the most significant bit immediately to the right of the radix point as a '1'. Both the mantissa and the exponent can have a positive or a negative value.

The mixed binary number $(110.1011)_2$ will be represented in floating-point notation as $.1101011 \times 2^3 = .1101011e + 0011$. Here, .1101011 is the mantissa and $e + 0011$ implies that the exponent is +3. As another example, $(0.000111)_2$ will be written as $.111e - 0011$, with .111 being the mantissa and $e - 0011$ implying an exponent of −3. Also, $(-0.00000101)_2$ may be written as $-.101 \times 2^{-5} = -.101e - 0101$, where −.101 is the mantissa and $e - 0101$ indicates an exponent of −5. If we wanted to represent the mantissas using eight bits, then .1101011 and .111 would be represented as .11010110 and .11100000.

1.17.1 Range of Numbers and Precision

The range of numbers that can be represented in any machine depends upon the number of bits in the exponent, while the fractional accuracy or precision is ultimately determined by the number of bits in the mantissa. The higher the number of bits in the exponent, the larger is the range of numbers that can be represented. For example, the range of numbers possible in a floating-point binary number format using six bits to represent the magnitude of the exponent would be from 2^{-64} to 2^{+64}, which is equivalent to a range of $10^{-19}$to 10^{+19}. The precision is determined by the number of bits used to represent the mantissa. It is usually represented as decimal digits of precision. The concept of precision as defined with respect to floating-point notation can be explained in simple terms as follows. If the mantissa is stored in n number of bits, it can represent a decimal number between 0 and $2^n - 1$ as the mantissa is stored as an unsigned integer. If M is the largest number such that $10^M - 1$ is less than or equal to $2^n - 1$, then M is the precision expressed as decimal digits of precision. For example, if the mantissa is expressed in 20 bits, then decimal digits of precision can be found to be about 6, as $2^{20} - 1$ equals 1 048 575, which is a little over $10^6 - 1$. We will briefly describe the commonly used formats for binary floating-point number representation.

1.17.2 Floating-Point Number Formats

The most commonly used format for representing floating-point numbers is the IEEE-754 standard. The full title of the standard is IEEE Standard for Binary Floating-point Arithmetic (ANSI/IEEE STD 754-1985). It is also known as Binary Floating-point Arithmetic for Microprocessor Systems, IEC

60559:1989. An ongoing revision to IEEE-754 is IEEE-754r. Another related standard IEEE 854-1987 generalizes IEEE-754 to cover both binary and decimal arithmetic. A brief description of salient features of the IEEE-754 standard, along with an introduction to other related standards, is given below.

ANSI/IEEE-754 Format

The IEEE-754 floating point is the most commonly used representation for real numbers on computers including Intel-based personal computers, Macintoshes and most of the UNIX platforms. It specifies four formats for representing floating-point numbers. These include single-precision, double-precision, single-extended precision and double-extended precision formats. Table 1.1 lists characteristic parameters of the four formats contained in the IEEE-754 standard. Of the four formats mentioned, the single-precision and double-precision formats are the most commonly used ones. The single-extended and double-extended precision formats are not common.

Figure 1.1 shows the basic constituent parts of the single- and double-precision formats. As shown in the figure, the floating-point numbers, as represented using these formats, have three basic components including the sign, the exponent and the mantissa. A '0' denotes a positive number and a '1' denotes a negative number. The n-bit exponent field needs to represent both positive and negative exponent values. To achieve this, a bias equal to $2^{n-1} - 1$ is added to the actual exponent in order to obtain the stored exponent. This equals 127 for an eight-bit exponent of the single-precision format and 1023 for an 11-bit exponent of the double-precision format. The addition of bias allows the use of an exponent in the range from -127 to $+128$, corresponding to a range of 0–255 in the first case, and in the range from -1023 to $+1024$, corresponding to a range of 0–2047 in the second case. A negative exponent is always represented in 2's complement form. The single-precision format offers a range from 2^{-127} to 2^{+127}, which is equivalent to 10^{-38} to 10^{+38}. The figures are 2^{-1023} to 2^{+1023}, which is equivalent to 10^{-308} to 10^{+308} in the case of the double-precision format.

The extreme exponent values are reserved for representing special values. For example, in the case of the single-precision format, for an exponent value of -127, the biased exponent value is zero, represented by an all 0s exponent field. In the case of a biased exponent of zero, if the mantissa is zero as well, the value of the floating-point number is exactly zero. If the mantissa is nonzero, it represents a denormalized number that does not have an assumed leading bit of '1'. A biased exponent of $+255$, corresponding to an actual exponent of $+128$, is represented by an all 1s exponent field. If the mantissa is zero, the number represents infinity. The sign bit is used to distinguish between positive and negative infinity. If the mantissa is nonzero, the number represents a 'NaN' (Not a Number). The value NaN is used to represent a value that does not represent a real number. This means that an eight-bit exponent can represent exponent values between -126 and $+127$. Referring to Fig. 1.1(a), the MSB of byte 1 indicates the sign of the mantissa. The remaining seven bits of byte 1 and the MSB of byte 2 represent an eight-bit exponent. The remaining seven bits of byte 2 and the 16 bits of byte 3 and byte 4 give a 23-bit mantissa. The mantissa m is normalized. The left-hand bit of the normalized mantissa is always

Table 1.1 Characteristic parameters of IEEE-754 formats.

Precision	Sign (bits)	Exponent (bits)	Mantissa (bits)	Total length (bits)	Decimal digits of precision
Single	1	8	23	32	> 6
Single-extended	1	≥ 11	≥ 32	≥ 44	> 9
Double	1	11	52	64	> 15
Double-extended	1	≥ 15	≥ 64	≥ 80	> 19

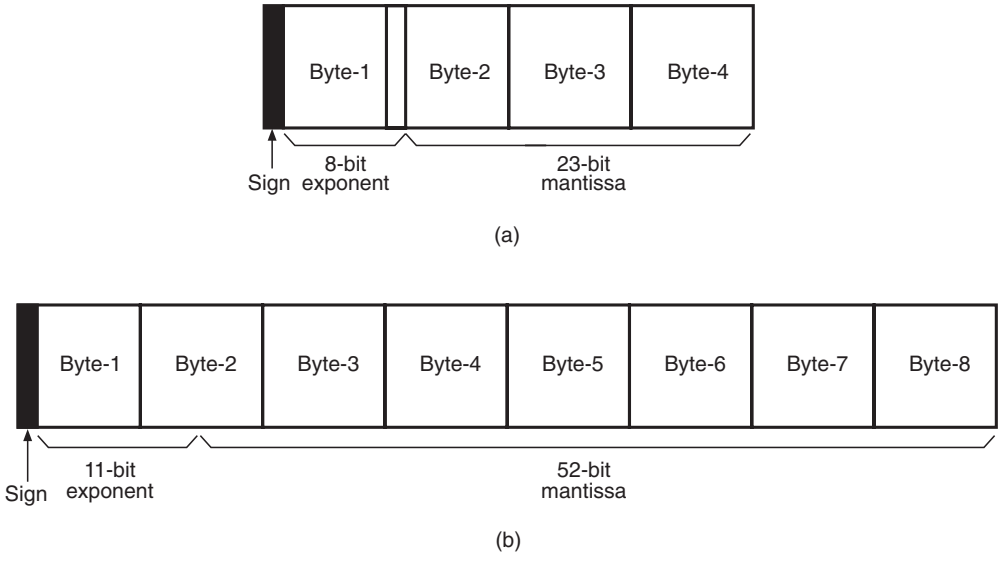


Figure 1.1 Single-precision and double-precision formats.

'1'. This '1' is not included but is always implied. A similar explanation can be given in the case of the double-precision format shown in Fig. 1.1(b).

Step-by-step transformation of $(23)_{10}$ into an equivalent floating-point number in single-precision IEEE format is as follows:

- $(23)_{10} = (10111)_2 = 1.0111e + 0100$.
- The mantissa = 0111000 00000000 00000000.
- The exponent = 00000100.
- The biased exponent = 00000100 + 01111111 = 10000011.
- The sign of the mantissa = 0.
- $(+23)_{10}$ = 01000001 10111000 00000000 00000000.
- Also, $(-23)_{10}$= 11000001 10111000 00000000 00000000.

IEEE-754r Format

As mentioned earlier, IEEE-754r is an ongoing revision to the IEEE-754 standard. The main objective of the revision is to extend the standard wherever it has become necessary, the most obvious enhancement to the standard being the addition of the 128-bit format and decimal format. Extension of the standard to include decimal floating-point representation has become necessary as most commercial data are held in decimal form and the binary floating point cannot represent decimal fractions exactly. If the binary floating point is used to represent decimal data, it is likely that the results will not be the same as those obtained by using decimal arithmetic.

In the revision process, many of the definitions have been rewritten for clarification and consistency. In terms of the addition of new formats, a new addition to the existing binary formats is the 128-bit 'quad-precision' format. Also, three new decimal formats, matching the lengths of binary formats,

have been described. These include decimal formats with a seven-, 16- and 34-digit mantissa, which may be normalized or denormalized. In order to achieve maximum range (decided by the number of exponent bits) and precision (decided by the number of mantissa bits), the formats merge part of the exponent and mantissa into a combination field and compress the remainder of the mantissa using densely packed decimal encoding. Detailed description of the revision, however, is beyond the scope of this book.

IEEE-854 Standard

The main objective of the IEEE-854 standard was to define a standard for floating-point arithmetic without the radix and word length dependencies of the better-known IEEE-754 standard. That is why IEEE-854 is called the IEEE standard for radix-independent floating-point arithmetic. Although the standard specifies only the binary and decimal floating-point arithmetic, it provides sufficient guidelines for those contemplating the implementation of the floating point using any other radix value such as 16 of the hexadecimal number system. This standard, too, specifies four formats including single, single-extended, double and double-extended precision formats.

Example 1.11

Determine the floating-point representation of $(-142)_{10}$ using the IEEE single-precision format.

Solution

- As a first step, we will determine the binary equivalent of $(142)_{10}$. Following the procedure outlined in an earlier part of the chapter, the binary equivalent can be written as $(142)_{10} = (10001110)_2$.
- $(10001110)_2 = 1.000\ 1110 \times 2^7 = 1.0001110e + 0111$.
- The mantissa = 0001110 00000000 00000000.
- The exponent = 00000111.
- The biased exponent = 00000111 + 01111111 = 10000110.
- The sign of the mantissa = 1.
- Therefore, $(-142)_{10}$ = 11000011 00001110 00000000 00000000.

Example 1.12

Determine the equivalent decimal numbers for the following floating-point numbers:

(a) 00111111 01000000 00000000 00000000 (IEEE-754 single-precision format);
(b) 11000000 00101001 01100 . . . 45 0s (IEEE-754 double-precision format).

Solution

(a) From an examination of the given number:
The sign of the mantissa is positive, as indicated by the '0' bit in the designated position.
The biased exponent = 01111110.
The unbiased exponent = 01111110 − 01111111 = 11111111.
It is clear from the eight bits of unbiased exponent that the exponent is negative, as the 2's complement representation of a number gives '1' in place of MSB.
The magnitude of the exponent is given by the 2's complement of $(11111111)_2$, which is $(00000001)_2 = 1$.

Therefore, the exponent $= -1$.
The mantissa bits = 11000000 00000000 00000000 ('1' in MSB is implied).
The normalized mantissa = 1.1000000 00000000 00000000.
The magnitude of the mantissa can be determined by shifting the mantissa bits one position to the left.
That is, the mantissa $= (.11)_2 = (0.75)_{10}$.

(b) The sign of the mantissa is negative, indicated by the '1' bit in the designated position.
The biased exponent = 10000000010.
The unbiased exponent = 10000000010 − 01111111111 = 00000000011.
It is clear from the 11 bits of unbiased exponent that the exponent is positive owing to the '0' in place of MSB. The magnitude of the exponent is 3. Therefore, the exponent $= +3$.
The mantissa bits = 1100101100 . . . 45 0s ('1' in MSB is implied).
The normalized mantissa = 1.100101100 . . . 45 0s.
The magnitude of the mantissa can be determined by shifting the mantissa bits three positions to the right.
That is, the mantissa $= (1100.101)_2 = (12.625)_{10}$.
Therefore, the equivalent decimal number $= -12.625$.

Review Questions

1. What is meant by the radix or base of a number system? Briefly describe why hex representation is used for the addresses and the contents of the memory locations in the main memory of a computer.
2. What do you understand by the l's and 2's complements of a binary number? What will be the range of decimal numbers that can be represented using a 16-bit 2's complement format?
3. Briefly describe the salient features of the IEEE-754 standard for representing floating-point numbers.
4. Why was it considered necessary to carry out a revision of the IEEE-754 standard? What are the main features of IEEE-754r (the notation for IEEE-754 under revision)?
5. In a number system, what decides (a) the place value or weight of a given digit and (b) the maximum numbers representable with a given number of digits?
6. In a floating-point representation, what represents (a) the range of representable numbers and (b) the precision with which a given number can be represented?
7. Why is there a need to have floating-point standards that can take care of decimal data and decimal arithmetic in addition to binary data and arithmetic?

Problems

1. Do the following conversions:

   (a) eight-bit 2's complement representation of $(-23)_{10}$;
   (b) The decimal equivalent of $(00010111)_2$ represented in 2's complement form.

   (a) 11101001; (b) +23

2. Two possible binary representations of $(-1)_{10}$ are $(10000001)_2$ and $(11111111)_2$. One of them belongs to the sign-bit magnitude format and the other to the 2's complement format. Identify.

   $(10000001)_2$ = sign-bit magnitude and $(11111111)_2$ = 2's complement form

3. Represent the following in the IEEE-754 floating-point standard using the single-precision format:

   (a) 32-bit binary number 11110000 11001100 10101010 00001111;
   (b) $(-118.625)_{10}$.

(a) 01001111 01110000 11001100 10101010;
(b) 11000010 11101101 01000000 00000000

4. Give the next three numbers in each of the following hex sequences:

   (a) 4A5, 4A6, 4A7, 4A8, . . . ;
   (b) B998, B999, . . .

   (a) 4A9, 4AA, 4AB; (b) B99A, B99B, B99C

5. Show that:

   (a) $(13A7)_{16} = (5031)_{10}$;
   (b) $(3F2)_{16} = (1111110010)_2$.

6. Assume a radix-32 arbitrary number system with 0–9 and A–V as its basic digits. Express the mixed binary number $(110101.001)_2$ in this arbitrary number system.

   1L.4

Further Reading

1. Tokheim, R. L. (1994) *Schaum's Outline Series of Digital Principles*, McGraw-Hill Companies Inc., USA.
2. Atiyah, S. K. (2005) *A Survey of Arithmetic*, Trafford Publishing, Victoria, BC, Canada.
3. Langholz, G., Mott, J. L. and Kandel, A. (1998) *Foundations of Digital Logic Design*, World Scientific Publ. Co. Inc., Singapore.
4. Cook, N. P. (2003) *Practical Digital Electronics*, Prentice-Hall, NJ, USA.
5. Lu, M. (2004) *Arithmetic and Logic in Computer Systems*, John Wiley & Sons, Inc., NJ, USA.

2

Binary Codes

The present chapter is an extension of the previous chapter on *number systems*. In the previous chapter, beginning with some of the basic concepts common to all number systems and an outline on the familiar decimal number system, we went on to discuss the binary, the hexadecimal and the octal number systems. While the binary system of representation is the most extensively used one in digital systems, including computers, octal and hexadecimal number systems are commonly used for representing groups of binary digits. The binary coding system, called the straight binary code and discussed in the previous chapter, becomes very cumbersome to handle when used to represent larger decimal numbers. To overcome this shortcoming, and also to perform many other special functions, several binary codes have evolved over the years. Some of the better-known binary codes, including those used efficiently to represent numeric and alphanumeric data, and the codes used to perform special functions, such as detection and correction of errors, will be detailed in this chapter.

2.1 Binary Coded Decimal

The binary coded decimal (BCD) is a type of binary code used to represent a given decimal number in an equivalent binary form. BCD-to-decimal and decimal-to-BCD conversions are very easy and straightforward. It is also far less cumbersome an exercise to represent a given decimal number in an equivalent BCD code than to represent it in the equivalent straight binary form discussed in the previous chapter.

The BCD equivalent of a decimal number is written by replacing each decimal digit in the integer and fractional parts with its four-bit binary equivalent. As an example, the BCD equivalent of $(23.15)_{10}$ is written as $(0010\ 0011.0001\ 0101)_{BCD}$. The BCD code described above is more precisely known as the 8421 BCD code, with 8, 4, 2 and 1 representing the weights of different bits in the four-bit groups, starting from MSB and proceeding towards LSB. This feature makes it a weighted code, which means that each bit in the four-bit group representing a given decimal digit has an assigned

Digital Electronics: Principles, Devices and Applications Anil K. Maini
 ISBN: 978-0-470-03214-5

Table 2.1 BCD codes.

Decimal	8421 BCD code	4221 BCD code	5421 BCD code
0	0000	0000	0000
1	0001	0001	0001
2	0010	0010	0010
3	0011	0011	0011
4	0100	1000	0100
5	0101	0111	1000
6	0110	1100	1001
7	0111	1101	1010
8	1000	1110	1011
9	1001	1111	1100

weight. Other weighted BCD codes include the 4221 BCD and 5421 BCD codes. Again, 4, 2, 2 and 1 in the 4221 BCD code and 5, 4, 2 and 1 in the 5421 BCD code represent weights of the relevant bits. Table 2.1 shows a comparison of 8421, 4221 and 5421 BCD codes. As an example, $(98.16)_{10}$ will be written as 1111 1110.0001 1100 in 4221 BCD code and 1100 1011.0001 1001 in 5421 BCD code. Since the 8421 code is the most popular of all the BCD codes, it is simply referred to as the BCD code.

2.1.1 BCD-to-Binary Conversion

A given BCD number can be converted into an equivalent binary number by first writing its decimal equivalent and then converting it into its binary equivalent. The first step is straightforward, and the second step was explained in the previous chapter. As an example, we will find the binary equivalent of the BCD number 0010 1001.0111 0101:

- BCD number: 0010 1001.0111 0101.
- Corresponding decimal number: 29.75.
- The binary equivalent of 29.75 can be determined to be 11101 for the integer part and .11 for the fractional part.
- Therefore, $(0010\ 1001.0111\ 0101)_{BCD} = (11101.11)_2$.

2.1.2 Binary-to-BCD Conversion

The process of binary-to-BCD conversion is the same as the process of BCD-to-binary conversion executed in reverse order. A given binary number can be converted into an equivalent BCD number by first determining its decimal equivalent and then writing the corresponding BCD equivalent. As an example, we will find the BCD equivalent of the binary number 10101011.101:

- The decimal equivalent of this binary number can be determined to be 171.625.
- The BCD equivalent can then be written as 0001 0111 0001.0110 0010 0101.

2.1.3 Higher-Density BCD Encoding

In the regular BCD encoding of decimal numbers, the number of bits needed to represent a given decimal number is always greater than the number of bits required for straight binary encoding of the same. For example, a three-digit decimal number requires 12 bits for representation in conventional BCD format. However, since $2^{10} > 10^3$, if these three decimal digits are encoded together, only 10 bits would be needed to do that. Two such encoding schemes are *Chen-Ho encoding* and the *densely packed decimal*. The latter has the advantage that subsets of the encoding encode two digits in the optimal seven bits and one digit in four bits like regular BCD.

2.1.4 Packed and Unpacked BCD Numbers

In the case of unpacked BCD numbers, each four-bit BCD group corresponding to a decimal digit is stored in a separate register inside the machine. In such a case, if the registers are eight bits or wider, the register space is wasted.

In the case of packed BCD numbers, two BCD digits are stored in a single eight-bit register. The process of combining two BCD digits so that they are stored in one eight-bit register involves shifting the number in the upper register to the left 4 times and then adding the numbers in the upper and lower registers. The process is illustrated by showing the storage of decimal digits '5' and '7':

- Decimal digit 5 is initially stored in the eight-bit register as: 0000 0101.
- Decimal digit 7 is initially stored in the eight-bit register as: 0000 0111.
- After shifting to the left 4 times, the digit 5 register reads: 0101 0000.
- The addition of the contents of the digit 5 and digit 7 registers now reads: 0101 0111.

Example 2.1

How many bits would be required to encode decimal numbers 0 to 9999 in straight binary and BCD codes? What would be the BCD equivalent of decimal 27 in 16-bit representation?

Solution

- Total number of decimals to be represented $= 10\,000 = 10^4 = 2^{13.29}$.
- Therefore, the number of bits required for straight binary encoding $= 14$.
- The number of bits required for BCD encoding $= 16$.
- The BCD equivalent of 27 in 16-bit representation $= 0000000000100111$.

2.2 Excess-3 Code

The excess-3 code is another important BCD code. It is particularly significant for arithmetic operations as it overcomes the shortcomings encountered while using the 8421 BCD code to add two decimal digits whose sum exceeds 9. The excess-3 code has no such limitation, and it considerably simplifies arithmetic operations. Table 2.2 lists the excess-3 code for the decimal numbers 0–9.

The excess-3 code for a given decimal number is determined by adding '3' to each decimal digit in the given number and then replacing each digit of the newly found decimal number by

Table 2.2 Excess-3 code equivalent of decimal numbers.

Decimal number	Excess-3 code	Decimal number	Excess-3 code
0	0011	5	1000
1	0100	6	1001
2	0101	7	1010
3	0110	8	1011
4	0111	9	1100

its four-bit binary equivalent. It may be mentioned here that, if the addition of '3' to a digit produces a carry, as is the case with the digits 7, 8 and 9, that carry should not be taken forward. The result of addition should be taken as a single entity and subsequently replaced with its excess-3 code equivalent. As an example, let us find the excess-3 code for the decimal number 597:

- The addition of '3' to each digit yields the three new digits/numbers '8', '12' and '10'.
- The corresponding four-bit binary equivalents are 1000, 1100 and 1010 respectively.
- The excess-3 code for 597 is therefore given by: 1000 1100 1010 = 100011001010.

Also, it is normal practice to represent a given decimal digit or number using the maximum number of digits that the digital system is capable of handling. For example, in four-digit decimal arithmetic, 5 and 37 would be written as 0005 and 0037 respectively. The corresponding 8421 BCD equivalents would be 0000000000000101 and 0000000000110111 and the excess-3 code equivalents would be 0011001100111000 and 0011001101101010.

Corresponding to a given excess-3 code, the equivalent decimal number can be determined by first splitting the number into four-bit groups, starting from the radix point, and then subtracting 0011 from each four-bit group. The new number is the 8421 BCD equivalent of the given excess-3 code, which can subsequently be converted into the equivalent decimal number. As an example, following these steps, the decimal equivalent of excess-3 number 01010110.10001010 would be 23.57.

Another significant feature that makes this code attractive for performing arithmetic operations is that the complement of the excess-3 code of a given decimal number yields the excess-3 code for 9's complement of the decimal number. As adding 9's complement of a decimal number B to a decimal number A achieves A – B, the excess-3 code can be used effectively for both addition and subtraction of decimal numbers.

Example 2.3

Find (a) the excess-3 equivalent of $(237.75)_{10}$ and (b) the decimal equivalent of the excess-3 number 110010100011.01110101.

Solution

(a) Integer part = 237. The excess-3 code for $(237)_{10}$ is obtained by replacing 2, 3 and 7 with the four-bit binary equivalents of 5, 6 and 10 respectively. This gives the excess-3 code for $(237)_{10}$ as: 0101 0110 1010 = 010101101010.

Fractional part = .75. The excess-3 code for $(.75)_{10}$ is obtained by replacing 7 and 5 with the four-bit binary equivalents of 10 and 8 respectively. That is, the excess-3 code for $(.75)_{10} = .10101000$.
Combining the results of the integral and fractional parts, the excess-3 code for $(237.75)_{10} = 010101101010.10101000$.

(b) The excess-3 code = 110010100011.01110101 = 1100 1010 0011.0111 0101.
Subtracting 0011 from each four-bit group, we obtain the new number as: 1001 0111 0000.0100 0010.
Therefore, the decimal equivalent = $(970.42)_{10}$.

2.3 Gray Code

The Gray code was designed by Frank Gray at Bell Labs and patented in 1953. It is an unweighted binary code in which two successive values differ only by 1 bit. Owing to this feature, the maximum error that can creep into a system using the binary Gray code to encode data is much less than the worst-case error encountered in the case of straight binary encoding. Table 2.3 lists the binary and Gray code equivalents of decimal numbers 0–15. An examination of the four-bit Gray code numbers, as listed in Table 2.3, shows that the last entry rolls over to the first entry. That is, the last and the first entry also differ by only 1 bit. This is known as the *cyclic property* of the Gray code. Although there can be more than one Gray code for a given word length, the term was first applied to a specific binary code for non-negative integers and called the *binary-reflected Gray code* or simply the Gray code.

There are various ways by which Gray codes with a given number of bits can be remembered. One such way is to remember that the least significant bit follows a repetitive pattern of '2' (11, 00, 11, . . .), the next higher adjacent bit follows a pattern of '4' (1111, 0000, 1111, . . .) and so on. We can also generate the n-bit Gray code recursively by prefixing a '0' to the Gray code for $n-1$ bits to obtain the first 2^{n-1} numbers, and then prefixing '1' to the reflected Gray code for $n-1$ bits to obtain the remaining 2^{n-1} numbers. The reflected Gray code is nothing but the code written in reverse order. The process of generation of higher-bit Gray codes using the reflect-and-prefix method is illustrated in Table 2.4. The columns of bits between those representing the Gray codes give the intermediate step of writing the code followed by the same written in reverse order.

Table 2.3 Gray code.

Decimal	Binary	Gray	Decimal	Binary	Gray
0	0000	0000	8	1000	1100
1	0001	0001	9	1001	1101
2	0010	0011	10	1010	1111
3	0011	0010	11	1011	1110
4	0100	0110	12	1100	1010
5	0101	0111	13	1101	1011
6	0110	0101	14	1110	1001
7	0111	0100	15	1111	1000

Table 2.4 Generation of higher-bit Gray code numbers.

One-bit Gray code	Two-bit Gray code		Three-bit Gray code		Four-bit Gray code	
0	0	00	00	000	000	0000
1	1	01	01	001	001	0001
	1	11	11	011	011	0011
	0	10	10	010	010	0010
			10	110	110	0110
			11	111	111	0111
			01	101	101	0101
			00	100	100	0100
					100	1100
					101	1101
					111	1111
					110	1110
					010	1010
					011	1011
					001	1001
					000	1000

2.3.1 Binary–Gray Code Conversion

A given binary number can be converted into its Gray code equivalent by going through the following steps:

1. Begin with the most significant bit (MSB) of the binary number. The MSB of the Gray code equivalent is the same as the MSB of the given binary number.
2. The second most significant bit, adjacent to the MSB, in the Gray code number is obtained by adding the MSB and the second MSB of the binary number and ignoring the carry, if any. That is, if the MSB and the bit adjacent to it are both '1', then the corresponding Gray code bit would be a '0'.
3. The third most significant bit, adjacent to the second MSB, in the Gray code number is obtained by adding the second MSB and the third MSB in the binary number and ignoring the carry, if any.
4. The process continues until we obtain the LSB of the Gray code number by the addition of the LSB and the next higher adjacent bit of the binary number.

The conversion process is further illustrated with the help of an example showing step-by-step conversion of $(1011)_2$ into its Gray code equivalent:

Binary 1011
Gray code 1- - -
Binary 1011
Gray code 11- -
Binary 1011
Gray code 111-
Binary 1011
Gray code 1110

2.3.2 Gray Code–Binary Conversion

A given Gray code number can be converted into its binary equivalent by going through the following steps:

1. Begin with the most significant bit (MSB). The MSB of the binary number is the same as the MSB of the Gray code number.
2. The bit next to the MSB (the second MSB) in the binary number is obtained by adding the MSB in the binary number to the second MSB in the Gray code number and disregarding the carry, if any.
3. The third MSB in the binary number is obtained by adding the second MSB in the binary number to the third MSB in the Gray code number. Again, carry, if any, is to be ignored.
4. The process continues until we obtain the LSB of the binary number.

The conversion process is further illustrated with the help of an example showing step-by-step conversion of the Gray code number 1110 into its binary equivalent:

Gray code 1110
Binary 1- - -
Gray code 1110
Binary 10 - -
Gray code 1110
Binary 101
Gray code 1110
Binary 1011

2.3.3 n-ary Gray Code

The binary-reflected Gray code described above is invariably referred to as the 'Gray code'. However, over the years, mathematicians have discovered other types of Gray code. One such code is the n-ary Gray code, also called the non-Boolean Gray code owing to the use of non-Boolean symbols for encoding. The generalized representation of the code is the (n, k)-Gray code, where n is the number of independent digits used and k is the word length. A ternary Gray code ($n = 3$) uses the values 0, 1 and 2, and the sequence of numbers in the two-digit word length would be (00, 01, 02, 12, 11, 10, 20, 21, 22). In the quaternary ($n = 4$) code, using 0, 1, 2 and 3 as independent digits and a two-digit word length, the sequence of numbers would be (00, 01, 02, 03, 13, 12, 11, 10, 20, 21, 22, 23, 33, 32, 31, 30). It is important to note here that an (n, k)-Gray code with an odd n does not exhibit the cyclic property of the binary Gray code, while in case of an even n it does have the cyclic property.

The (n, k)-Gray code may be constructed recursively, like the binary-reflected Gray code, or may be constructed iteratively. The process of generating larger word-length ternary Gray codes is illustrated in Table 2.5. The columns between those representing the ternary Gray codes give the intermediate steps.

2.3.4 Applications

1. The Gray code is used in the transmission of digital signals as it minimizes the occurrence of errors.
2. The Gray code is preferred over the straight binary code in angle-measuring devices. Use of the Gray code almost eliminates the possibility of an angle misread, which is likely if the

Table 2.5 Generation of a larger word-length ternary Gray code.

One-digit ternary code	Two-digit ternary code		Three-digit ternary code
0	0	00	000
1	1	01	001
2	2	02	002
	2	12	012
	1	11	011
	0	10	010
	0	20	020
	1	21	021
	2	22	022
		22	122
		21	121
		20	120
		10	110
		11	111
		12	112
		02	102
		01	101
		00	100
		00	200
		01	201
		02	202
		12	212
		11	211
		10	210
		20	220
		21	221
		22	222

angle is represented in straight binary. The cyclic property of the Gray code is a plus in this application.

3. The Gray code is used for labelling the axes of Karnaugh maps, a graphical technique used for minimization of Boolean expressions.
4. The use of Gray codes to address program memory in computers minimizes power consumption. This is due to fewer address lines changing state with advances in the program counter.
5. Gray codes are also very useful in genetic algorithms since mutations in the code allow for mostly incremental changes. However, occasionally a one-bit change can result in a big leap, thus leading to new properties.

Example 2.4

Find (a) the Gray code equivalent of decimal 13 and (b) the binary equivalent of Gray code number 1111.

Solution

(a) The binary equivalent of decimal 13 is 1101.
Binary–Gray conversion

Binary 1101
Gray 1- - -
Binary 1101
Gray 10 - -
Binary 1101
Gray 101 –
Binary 1101
Gray 1011

(b) *Gray–binary conversion*

Gray 1111
Binary 1- - -
Gray 1111
Binary 10- -
Gray 1111
Binary 101-
Gray 1111
Binary 1010

Example 2.5

Given the sequence of three-bit Gray code as (000, 001, 011, 010, 110, 111, 101, 100), write the next three numbers in the four-bit Gray code sequence after 0101.

Solution

The first eight of the 16 Gray code numbers of the four-bit Gray code can be written by appending '0' to the eight three-bit Gray code numbers. The remaining eight can be determined by appending '1' to the eight three-bit numbers written in reverse order. Following this procedure, we can write the next three numbers after 0101 as 0100, 1100 and 1101.

2.4 Alphanumeric Codes

Alphanumeric codes, also called character codes, are binary codes used to represent alphanumeric data. The codes write alphanumeric data, including letters of the alphabet, numbers, mathematical symbols and punctuation marks, in a form that is understandable and processable by a computer. These codes enable us to interface input–output devices such as keyboards, printers, VDUs, etc., with the computer. One of the better-known alphanumeric codes in the early days of evolution of computers, when punched cards used to be the medium of inputting and outputting data, is the 12-bit Hollerith code. The Hollerith code was used in those days to encode alphanumeric data on punched cards. The code has, however, been rendered obsolete, with the punched card medium having completely vanished from the scene. Two widely used alphanumeric codes include the ASCII and the EBCDIC codes. While the former is popular with microcomputers and is used on nearly all personal computers and workstations, the latter is mainly used with larger systems.

Traditional character encodings such as ASCII, EBCDIC and their variants have a limitation in terms of the number of characters they can encode. In fact, no single encoding contains enough characters so as to cover all the languages of the European Union. As a result, these encodings do not permit multilingual computer processing. Unicode, developed jointly by the Unicode Consortium and the International Standards Organization (ISO), is the most complete character encoding scheme that allows text of all forms and languages to be encoded for use by computers. Different codes are described in the following.

2.4.1 ASCII code

The ASCII (American Standard Code for Information Interchange), pronounced 'ask-ee', is strictly a seven-bit code based on the English alphabet. ASCII codes are used to represent alphanumeric data in computers, communications equipment and other related devices. The code was first published as a standard in 1967. It was subsequently updated and published as ANSI X3.4-1968, then as ANSI X3.4-1977 and finally as ANSI X3.4-1986. Since it is a seven-bit code, it can at the most represent 128 characters. It currently defines 95 printable characters including 26 upper-case letters (A to Z), 26 lower-case letters (a to z), 10 numerals (0 to 9) and 33 special characters including mathematical symbols, punctuation marks and space character. In addition, it defines codes for 33 nonprinting, mostly obsolete control characters that affect how text is processed. With the exception of 'carriage return' and/or 'line feed', all other characters have been rendered obsolete by modern mark-up languages and communication protocols, the shift from text-based devices to graphical devices and the elimination of teleprinters, punch cards and paper tapes. An eight-bit version of the ASCII code, known as US ASCII-8 or ASCII-8, has also been developed. The eight-bit version can represent a maximum of 256 characters.

Table 2.6 lists the ASCII codes for all 128 characters. When the ASCII code was introduced, many computers dealt with eight-bit groups (or bytes) as the smallest unit of information. The eighth bit was commonly used as a parity bit for error detection on communication lines and other device-specific functions. Machines that did not use the parity bit typically set the eighth bit to '0'.

Table 2.6 ASCII code.

Decimal	Hex	Binary	Code	Code description
0	00	0000 0000	NUL	Null character
1	01	0000 0001	SOH	Start of header
2	02	0000 0010	STX	Start of text
3	03	0000 0011	ETX	End of text
4	04	0000 0100	EOT	End of transmission
5	05	0000 0101	ENQ	Enquiry
6	06	0000 0110	ACK	Acknowledgement
7	07	0000 0111	BEL	Bell
8	08	0000 1000	BS	Backspace
9	09	0000 1001	HT	Horizontal tab
10	0A	0000 1010	LF	Line feed
11	0B	0000 1011	VT	Vertical tab
12	0C	0000 1100	FF	Form feed
13	0D	0000 1101	CR	Carriage return
14	0E	0000 1110	SO	Shift out
15	0F	0000 1111	SI	Shift in
16	10	0001 0000	DLE	Data link escape
17	11	0001 0001	DC1	Device control 1 (XON)

Table 2.6 (*continued*).

Decimal	Hex	Binary	Code	Code description
18	12	0001 0010	DC2	Device control 2
19	13	0001 0011	DC3	Device control 3 (XOFF)
20	14	0001 0100	DC4	Device control 4
21	15	0001 0101	NAK	Negative acknowledgement
22	16	0001 0110	SYN	Synchronous idle
23	17	0001 0111	ETB	End of transmission block
24	18	0001 1000	CAN	Cancel
25	19	0001 1001	EM	End of medium
26	1A	0001 1010	SUB	Substitute
27	1B	0001 1011	ESC	Escape
28	1C	0001 1100	FS	File separator
29	1D	0001 1101	GS	Group separator
30	1E	0001 1110	RS	Record separator
31	1F	0001 1111	US	Unit separator
32	20	0010 0000	SP	Space
33	21	0010 0001	!	Exclamation point
34	22	0010 0010	"	Quotation mark
35	23	0010 0011	#	Number sign, octothorp, pound
36	24	0010 0100	$	Dollar sign
37	25	0010 0101	%	Percent
38	26	0010 0110	&	Ampersand
39	27	0010 0111	’	Apostrophe, prime
40	28	0010 1000	(	Left parenthesis
41	29	0010 1001	)	Right parenthesis
42	2A	0010 1010	*	Asterisk, ‘star’
43	2B	0010 1011	+	Plus sign
44	2C	0010 1100	,	Comma
45	2D	0010 1101	-	Hyphen, minus sign
46	2E	0010 1110	.	Period, decimal Point, ‘dot’
47	2F	0010 1111	/	Slash, virgule
48	30	0011 0000	0	0
49	31	0011 0001	1	1
50	32	0011 0010	2	2
51	33	0011 0011	3	3
52	34	0011 0100	4	4
53	35	0011 0101	5	5
54	36	0011 0110	6	6
55	37	0011 0111	7	7
56	38	0011 1000	8	8
57	39	0011 1001	9	9
58	3A	0011 1010	:	Colon
59	3B	0011 1011	;	Semicolon
60	3C	0011 1100	<	Less-than sign
61	3D	0011 1101	=	Equals sign
62	3E	0011 1110	>	Greater-than sign
63	3F	0011 1111	?	Question mark
64	40	0100 0000	@	At sign
65	41	0100 0001	A	A

(*continued overleaf*)

Table 2.6 (*continued*).

Decimal	Hex	Binary	Code	Code description
66	42	0100 0010	B	B
67	43	0100 0011	C	C
68	44	0100 0100	D	D
69	45	0100 0101	E	E
70	46	0100 0110	F	F
71	47	0100 0111	G	G
72	48	0100 1000	H	H
73	49	0100 1001	I	I
74	4A	0100 1010	J	J
75	4B	0100 1011	K	K
76	4C	0100 1100	L	L
77	4D	0100 1101	M	M
78	4E	0100 1110	N	N
79	4F	0100 1111	O	O
80	50	0101 0000	P	P
81	51	0101 0001	Q	Q
82	52	0101 0010	R	R
83	53	0101 0011	S	S
84	54	0101 0100	T	T
85	55	0101 0101	U	U
86	56	0101 0110	V	V
87	57	0101 0111	W	W
88	58	0101 1000	X	X
89	59	0101 1001	Y	Y
90	5A	0101 1010	Z	Z
91	5B	0101 1011	[	Opening bracket
92	5C	0101 1100	\	Reverse slash
93	5D	0101 1101	]	Closing bracket
94	5E	0101 1110	^	Circumflex, caret
95	5F	0101 1111	_	Underline, underscore
96	60	0110 0000	`	Grave accent
97	61	0110 0001	a	a
98	62	0110 0010	b	b
99	63	0110 0011	c	c
100	64	0110 0100	d	d
101	65	0110 0101	e	e
102	66	0110 0110	f	f
103	67	0110 0111	g	g
104	68	0110 1000	h	h
105	69	0110 1001	i	i
106	6A	0110 1010	j	j
107	6B	0110 1011	k	k
108	6C	0110 1100	l	l
109	6D	0110 1101	m	m
110	6E	0110 1110	n	n
111	6F	0110 1111	o	o
112	70	0111 0000	p	p
113	71	0111 0001	q	q
114	72	0111 0010	r	r

Table 2.6 (*continued*).

Decimal	Hex	Binary	Code	Code description
115	73	0111 0011	s	s
116	74	0111 0100	t	t
117	75	0111 0101	u	u
118	76	0111 0110	v	v
119	77	0111 0111	w	w
120	78	0111 1000	x	x
121	79	0111 1001	y	y
122	7A	0111 1010	z	z
123	7B	0111 1011	{	Opening brace
124	7C	0111 1100	|	Vertical line
125	7D	0111 1101	}	Closing brace
126	7E	0111 1110	~	Tilde
127	7F	0111 1111	DEL	Delete

Looking at the structural features of the code as reflected in Table 2.6, we can see that the digits 0 to 9 are represented with their binary values prefixed with 0011. That is, numerals 0 to 9 are represented by binary sequences from 0011 0000 to 0011 1001 respectively. Also, lower-case and upper-case letters differ in bit pattern by a single bit. While upper-case letters 'A' to 'O' are represented by 0100 0001 to 0100 1111, lower-case letters 'a' to 'o' are represented by 0110 0001 to 0110 1111. Similarly, while upper-case letters 'P' to 'Z' are represented by 0101 0000 to 0101 1010, lower-case letters 'p' to 'z' are represented by 0111 0000 to 0111 1010.

With widespread use of computer technology, many variants of the ASCII code have evolved over the years to facilitate the expression of non-English languages that use a Roman-based alphabet. In some of these variants, all ASCII printable characters are identical to their seven-bit ASCII code representations. For example, the eight-bit standard ISO/IEC 8859 was developed as a true extension of ASCII, leaving the original character mapping intact in the process of inclusion of additional values. This made possible representation of a broader range of languages. In spite of the standard suffering from incompatibilities and limitations, ISO-8859-1, its variant Windows-1252 and the original seven-bit ASCII continue to be the most common character encodings in use today.

2.4.2 EBCDIC code

The EBCDIC (Extended Binary Coded Decimal Interchange Code), pronounced 'eb-si-dik', is another widely used alphanumeric code, mainly popular with larger systems. The code was created by IBM to extend the binary coded decimal that existed at that time. All IBM mainframe computer peripherals and operating systems use EBCDIC code, and their operating systems provide ASCII and Unicode modes to allow translation between different encodings. It may be mentioned here that EBCDIC offers no technical advantage over the ASCII code and its variant ISO-8859 or Unicode. Its importance in the earlier days lay in the fact that it made it relatively easier to enter data into larger machines with punch cards. Since, punch cards are not used on mainframes any more, the code is used in contemporary mainframe machines solely for backwards compatibility.

It is an eight-bit code and thus can accommodate up to 256 characters. Table 2.7 gives the listing of characters in binary as well as hex form in EBCDIC. The arrangement is similar to the one adopted for Table 2.6 for the ASCII code. A single byte in EBCDIC is divided into two four-bit groups called

Table 2.7 EBCDIC code.

Decimal	Hex	Binary	Code	Code description
0	00	0000 0000	NUL	Null character
1	01	0000 0001	SOH	Start of header
2	02	0000 0010	STX	Start of text
3	03	0000 0011	ETX	End of text
4	04	0000 0100	PF	Punch off
5	05	0000 0101	HT	Horizontal tab
6	06	0000 0110	LC	Lower case
7	07	0000 0111	DEL	Delete
8	08	0000 1000		
9	09	0000 1001		
10	0A	0000 1010	SMM	Start of manual message
11	0B	0000 1011	VT	Vertical tab
12	0C	0000 1100	FF	Form feed
13	0D	0000 1101	CR	Carriage return
14	0E	0000 1110	SO	Shift out
15	0F	0000 1111	SI	Shift in
16	10	0001 0000	DLE	Data link escape
17	11	0001 0001	DC1	Device control 1
18	12	0001 0010	DC2	Device control 2
19	13	0001 0011	TM	Tape mark
20	14	0001 0100	RES	Restore
21	15	0001 0101	NL	New line
22	16	0001 0110	BS	Backspace
23	17	0001 0111	IL	Idle
24	18	0001 1000	CAN	Cancel
25	19	0001 1001	EM	End of medium
26	1A	0001 1010	CC	Cursor control
27	1B	0001 1011	CU1	Customer use 1
28	1C	0001 1100	IFS	Interchange file separator
29	1D	0001 1101	IGS	Interchange group separator
30	1E	0001 1110	IRS	Interchange record separator
31	1F	0001 1111	IUS	Interchange unit separator
32	20	0010 0000	DS	Digit select
33	21	0010 0001	SOS	Start of significance
34	22	0010 0010	FS	Field separator
35	23	0010 0011		
36	24	0010 0100	BYP	Bypass
37	25	0010 0101	LF	Line feed
38	26	0010 0110	ETB	End of transmission block
39	27	0010 0111	ESC	Escape
40	28	0010 1000		
41	29	0010 1001		
42	2A	0010 1010	SM	Set mode
43	2B	0010 1011	CU2	Customer use 2
44	2C	0010 1100		
45	2D	0010 1101	ENQ	Enquiry
46	2E	0010 1110	ACK	Acknowledge
47	2F	0010 1111	BEL	Bell
48	30	0011 0000		

Table 2.7 (*continued*).

Decimal	Hex	Binary	Code	Code description
49	31	0011 0001		
50	32	0011 0010	SYN	Synchronous idle
51	33	0011 0011		
52	34	0011 0100	PN	Punch on
53	35	0011 0101	RS	Reader stop
54	36	0011 0110	UC	Upper case
55	37	0011 0111	EOT	End of transmission
56	38	0011 1000		
57	39	0011 1001		
58	3A	0011 1010		
59	3B	0011 1011	CU3	Customer use 3
60	3C	0011 1100	DC4	Device control 4
61	3D	0011 1101	NAK	Negative acknowledge
62	3E	0011 1110		
63	3F	0011 1111	SUB	Substitute
64	40	0100 0000	SP	Space
65	41	0100 0001		
66	42	0100 0010		
67	43	0100 0011		
68	44	0100 0100		
69	45	0100 0101		
70	46	0100 0110		
71	47	0100 0111		
72	48	0100 1000		
73	49	0100 1001		
74	4A	0100 1010	¢	Cent sign
75	4B	0100 1011	.	Period, decimal point
76	4C	0100 1100	<	Less-than sign
77	4D	0100 1101	(	Left parenthesis
78	4E	0100 1110	+	Plus sign
79	4F	0100 1111	|	Logical OR
80	50	0101 0000	&	Ampersand
81	51	0101 0001		
82	52	0101 0010		
83	53	0101 0011		
84	54	0101 0100		
85	55	0101 0101		
86	56	0101 0110		
87	57	0101 0111		
88	58	0101 1000		
89	59	0101 1001		
90	5A	0101 1010	!	Exclamation point
91	5B	0101 1011	$	Dollar sign
92	5C	0101 1100	*	Asterisk
93	5D	0101 1101	)	Right parenthesis
94	5E	0101 1110	;	Semicolon
95	5F	0101 1111	∧	Logical NOT
96	60	0110 0000	-	Hyphen, minus sign

(*continued overleaf*)

Table 2.7 (*continued*).

Decimal	Hex	Binary	Code	Code description
97	61	0110 0001	/	Slash, virgule
98	62	0110 0010		
99	63	0110 0011		
100	64	0110 0100		
101	65	0110 0101		
102	66	0110 0110		
103	67	0110 0111		
104	68	0110 1000		
105	69	0110 1001		
106	6A	0110 1010		
107	6B	0110 1011	,	Comma
108	6C	0110 1100	%	Percent
109	6D	0110 1101	_	Underline, underscore
110	6E	0110 1110	>	Greater-than sign
111	6F	0110 1111	?	Question mark
112	70	0111 0000		
113	71	0111 0001		
114	72	0111 0010		
115	73	0111 0011		
116	74	0111 0100		
117	75	0111 0101		
118	76	0111 0110		
119	77	0111 0111		
120	78	0111 1000		
121	79	0111 1001	‘	Grave accent
122	7A	0111 1010	:	Colon
123	7B	0111 1011	#	Number sign, octothorp, pound
124	7C	0111 1100	@	At sign
125	7D	0111 1101	’	Apostrophe, prime
126	7E	0111 1110	=	Equals sign
127	7F	0111 1111	“	Quotation mark
128	80	1000 0000		
129	81	1000 1001	a	a
130	82	1000 1010	b	b
131	83	1000 1011	c	c
132	84	1000 1100	d	d
133	85	1000 0101	e	e
134	86	1000 0110	f	f
135	87	1000 0111	g	g
136	88	1000 1000	h	h
137	89	1000 1001	i	i
138	8A	1000 1010		
139	8B	1000 1011		
140	8C	1000 1100		
141	8D	1000 1101		
142	8E	1000 1110		
143	8F	1000 1111		
144	90	1001 0000		
145	91	1001 0001	j	j

Table 2.7 (*continued*).

Decimal	Hex	Binary	Code	Code description
146	92	1001 0010	k	k
147	93	1001 0011	l	l
148	94	1001 0100	m	m
149	95	1001 0101	n	n
150	96	1001 0110	o	o
151	97	1001 0111	p	p
152	98	1001 1000	q	q
153	99	1001 1001	r	r
154	9A	1001 1010		
155	9B	1001 1011		
156	9C	1001 1100		
157	9D	1001 1101		
158	9E	1001 1110		
159	9F	1001 1111		
160	A0	1010 0000		
161	A1	1010 0001	~	Tilde
162	A2	1010 0010	s	s
163	A3	1010 0011	t	t
164	A4	1010 0100	u	u
165	A5	1010 0101	v	v
166	A6	1010 0110	w	w
167	A7	1010 0111	x	x
168	A8	1010 1000	y	y
169	A9	1010 1001	z	z
170	AA	1010 1010		
171	AB	1010 1011		
172	AC	1010 1100		
173	AD	1010 1101		
174	AE	1010 1110		
175	AF	1010 1111		
176	B0	1011 0000		
177	B1	1011 0001		
178	B2	1011 0010		
179	B3	1011 0011		
180	B4	1011 0100		
181	B5	1011 0101		
182	B6	1011 0110		
183	B7	1011 0111		
184	B8	1011 1000		
185	B9	1011 1001		
186	BA	1011 1010		
187	BB	1011 1011		
188	BC	1011 1100		
189	BD	1011 1101		
190	BE	1011 1110		
191	BF	1011 1111		
192	C0	1100 0000	{	Opening brace
193	C1	1100 0001	A	A

(*continued overleaf*)

Table 2.7 (*continued*).

Decimal	Hex	Binary	Code	Code description
194	C2	1100 0010	B	B
195	C3	1100 0011	C	C
196	C4	1100 0100	D	D
197	C5	1100 0101	E	E
198	C6	1100 0110	F	F
199	C7	1100 0111	G	G
200	C8	1100 1000	H	H
201	C9	1100 1001	I	I
202	CA	1100 1010		
203	CB	1100 1011		
204	CC	1100 1100		
205	CD	1100 1101		
206	CE	1100 1110		
207	CF	1100 1111		
208	D0	1101 0000	}	Closing brace
209	D1	1101 0001	J	J
210	D2	1101 0010	K	K
211	D3	1101 0011	L	L
212	D4	1101 0100	M	M
213	D5	1101 0101	N	N
214	D6	1101 0110	O	O
215	D7	1101 0111	P	P
216	D8	1101 1000	Q	Q
217	D9	1101 1001	R	R
218	DA	1101 1010		
219	DB	1101 1011		
220	DC	1101 1100		
221	DD	1101 1101		
222	DE	1101 1110		
223	DF	1101 1111		
224	E0	1110 0000	\	Reverse slant
225	E1	1110 0001		
226	E2	1110 0010	S	S
227	E3	1110 0011	T	T
228	E4	1110 0100	U	U
229	E5	1110 0101	V	V
230	E6	1110 0110	W	W
231	E7	1110 0111	X	X
232	E8	1110 1000	Y	Y
233	E9	1110 1001	Z	Z
234	EA	1110 1010		
235	EB	1110 1011		
236	EC	1110 1100		
237	ED	1110 1101		
238	EE	1110 1110		
239	EF	1110 1111		
240	F0	1111 0000	0	0
241	F1	1111 0001	1	1

Table 2.7 (*continued*).

Decimal	Hex	Binary	Code	Code description
242	F2	1111 0010	2	2
243	F3	1111 0011	3	3
244	F4	1111 0100	4	4
245	F5	1111 0101	5	5
246	F6	1111 0110	6	6
247	F7	1111 0111	7	7
248	F8	1111 1000	8	8
249	F9	1111 1001	9	9
250	FA	1111 1010	|	
251	FB	1111 1011		
252	FC	1111 1100		
253	FD	1111 1101		
254	FE	1111 1110		
255	FF	1111 1111	eo	

nibbles. The first four-bit group, called the 'zone', represents the category of the character, while the second group, called the 'digit', identifies the specific character.

2.4.3 Unicode

As briefly mentioned in the earlier sections, encodings such as ASCII, EBCDIC and their variants do not have a sufficient number of characters to be able to encode alphanumeric data of all forms, scripts and languages. As a result, these encodings do not permit multilingual computer processing. In addition, these encodings suffer from incompatibility. Two different encodings may use the same number for two different characters or different numbers for the same characters. For example, code 4E (in hex) represents the upper-case letter 'N' in ASCII code and the plus sign '+' in the EBCDIC code. Unicode, developed jointly by the Unicode Consortium and the International Organization for Standardization (ISO), is the most complete character encoding scheme that allows text of all forms and languages to be encoded for use by computers. It not only enables the users to handle practically any language and script but also supports a comprehensive set of mathematical and technical symbols, greatly simplifying any scientific information exchange. The Unicode standard has been adopted by such industry leaders as HP, IBM, Microsoft, Apple, Oracle, Unisys, Sun, Sybase, SAP and many more.

Unicode and ISO-10646 Standards

Before we get on to describe salient features of Unicode, it may be mentioned that another standard similar in intent and implementation to Unicode is the ISO-10646. While Unicode is the brainchild of the Unicode Consortium, a consortium of manufacturers (initially mostly US based) of multilingual software, ISO-10646 is the project of the International Organization for Standardization. Although both organizations publish their respective standards independently, they have agreed to maintain compatibility between the code tables of Unicode and ISO-10646 and closely coordinate any further extensions.

The Code Table

The code table defined by both Unicode and ISO-10646 provides a unique number for every character, irrespective of the platform, program and language used. The table contains characters required to represent practically all known languages and scripts. The list includes not only the Greek, Latin, Cyrillic, Arabic, Arabian and Georgian scripts but also Japanese, Chinese and Korean scripts. In addition, the list also includes scripts such as Devanagari, Bengali, Gurmukhi, Gujarati, Oriya, Telugu, Tamil, Kannada, Thai, Tibetan, Ethiopic, Sinhala, Canadian Syllabics, Mongolian, Myanmar and others. Scripts not yet covered will eventually be added. The code table also covers a large number of graphical, typographical, mathematical and scientific symbols.

In the 32-bit version, which is the most recent version, the code table is divided into 2^{16} subsets, with each subset having 2^{16} characters. In the 32-bit representation, elements of different subsets therefore differ only in the 16 least significant bits. Each of these subsets is known as a plane. Plane 0, called the basic multilingual plane (BMP), defined by 00000000 to 0000FFFF, contains all most commonly used characters including all those found in major older encoding standards. Another subset of 2^{16} characters could be defined by 00010000 to 0001FFFF. Further, there are different slots allocated within the BMP to different scripts. For example, the basic Latin character set is encoded in the range 0000 to 007F. Characters added to the code table outside the 16-bit BMP are mostly for specialist applications such as historic scripts and scientific notation. There are indications that there may never be characters assigned outside the code space defined by 00000000 to 0010FFFF, which provides space for a little over 1 million additional characters.

Different characters in Unicode are represented by a hexadecimal number preceded by 'U+'. For example, 'A' and 'e' in basic Latin are respectively represented by U+0041 and U+0065. The first 256 code numbers in Unicode are compatible with the seven-bit ASCII-code and its eight-bit variant ISO-8859-1. Unicode characters U+0000 to U+007F (128 characters) are identical to those in the ASCII code, and the Unicode characters in the range U+0000 to U+00FF (256 characters) are identical to ISO-8859-1.

Use of Combining Characters

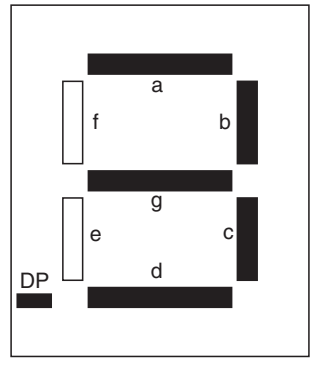
Unicode assigns code numbers to combining characters, which are not full characters by themselves but accents or other diacritical marks added to the previous character. This makes it possible to place any accent on any character. Although Unicode allows the use of combining characters, it also assigns separate codes to commonly used accented characters known as precomposed characters. This is done to ensure backwards compatibility with older encodings. As an example, the character 'ä' can be represented as the precomposed character U+00E4. It can also be represented in Unicode as U+0061 (Latin lower-case letter 'a') followed by U+00A8 (combining character '..').

Unicode and ISO-10646 Comparison

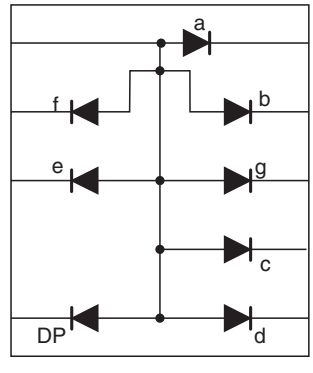
Although Unicode and ISO-10646 have identical code tables, Unicode offers many more features not available with ISO-10646. While the ISO-10646 standard is not much more than a comprehensive character set, the Unicode standard includes a number of other related features such as character properties and algorithms for text normalization and handling of bidirectional text to ensure correct display of mixed texts containing both right-to-left and left-to-right scripts.

2.5 Seven-segment Display Code

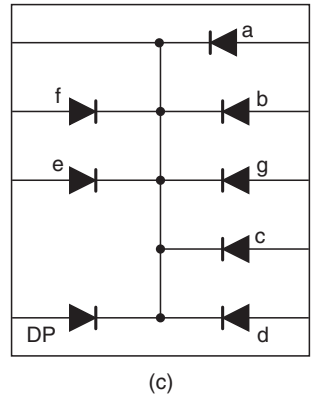
Seven-segment displays [Fig. 2.1(a)] are very common and are found almost everywhere, from pocket calculators, digital clocks and electronic test equipment to petrol pumps. A single seven-segment display or a stack of such displays invariably meets our display requirement. There are both LED and



(a)



(b)



(c)

Figure 2.1 Seven-segment displays.

Table 2.8 Seven-segment display code.

Common cathode type '1' means ON									Common anode type '0' means ON								
	a	b	c	d	e	f	g	DP		a	b	c	d	e	f	g	DP
0	1	1	1	1	1	1	0		0	0	0	0	0	0	0	1	
1	0	1	1	0	0	0	0		1	1	0	0	1	1	1	1	
2	1	1	0	1	1	0	1		2	0	0	1	0	0	1	0	
3	1	1	1	1	0	0	1		3	0	0	0	0	1	1	0	
4	0	1	1	0	0	1	1		4	1	0	0	1	1	0	0	
5	1	0	1	1	0	1	1		5	0	1	0	0	1	0	0	
6	0	0	1	1	1	1	1		6	1	1	0	0	0	0	0	
7	1	1	1	0	0	0	0		7	0	0	0	1	1	1	1	
8	1	1	1	1	1	1	1		8	0	0	0	0	0	0	0	
9	1	1	1	0	0	1	1		9	0	0	0	1	1	0	0	
a	1	1	1	1	1	0	1		a	0	0	0	0	0	1	0	
b	0	0	1	1	1	1	1		b	1	1	0	0	0	0	0	
c	0	0	0	1	1	0	1		c	1	1	1	0	0	1	0	
d	0	1	1	1	1	0	1		d	1	0	0	0	0	1	0	
e	1	1	0	1	1	1	1		e	0	0	1	0	0	0	0	
f	1	0	0	0	1	1	1		f	0	1	1	1	0	0	0	

LCD types of seven-segment display. Furthermore, there are common anode-type LED displays where the arrangement of different diodes, designated *a*, *b, c, d, e, f* and *g*, is as shown in Fig. 2.1(b), and common cathode-type displays where the individual diodes are interconnected as shown in Fig. 2.1(c). Each display unit usually has a dot point (DP).

The DP could be located either towards the left (as shown) or towards the right of the figure '8' display pattern. This type of display can be used to display numerals from 0 to 9 and letters from A to F. Table 2.8 gives the binary code for displaying different numeric and alphabetic characters for both the common cathode and the common anode type displays. A '1' lights a segment in the common cathode type display, and a '0' lights a segment in the common anode type display.

2.6 Error Detection and Correction Codes

When we talk about digital systems, be it a digital computer or a digital communication set-up, the issue of error detection and correction is of great practical significance. Errors creep into the bit stream owing to noise or other impairments during the course of its transmission from the transmitter to the receiver. Any such error, if not detected and subsequently corrected, can be disastrous, as digital systems are sensitive to errors and tend to malfunction if the bit error rate is more than a certain threshold level. Error detection and correction, as we will see below, involves the addition of extra bits, called check bits, to the information-carrying bit stream to give the resulting bit sequence a unique characteristic that helps in detection and localization of errors. These additional bits are also called redundant bits as they do not carry any information. While the addition of redundant bits helps in achieving the goal of making transmission of information from one place to another error free or reliable, it also makes it inefficient. In this section, we will examine some common error detection and correction codes.

2.6.1 Parity Code

A parity bit is an extra bit added to a string of data bits in order to detect any error that might have crept into it while it was being stored or processed and moved from one place to another in a digital system.

We have an *even parity*, where the added bit is such that the total number of ls in the data bit string becomes even, and an *odd parity*, where the added bit makes the total number of ls in the data bit string odd. This added bit could be a '0' or a '1'. As an example, if we have to add an even parity bit to 01000001 (the eight-bit ASCII code for 'A'), it will be a '0' and the number will become 001000001. If we have to add an odd parity bit to the same number, it will be a 'l' and the number will become 101000001. The odd parity bit is a complement of the even parity bit. The most common convention is to use even parity, that is, the total number of 1s in the bit stream, including the parity bit, is even.

The parity check can be made at different points to look for any possible single-bit error, as it would disturb the parity. This simple parity code suffers from two limitations. Firstly, it cannot detect the error if the number of bits having undergone a change is even. Although the number of bits in error being equal to or greater than 4 is a very rare occurrence, the addition of a single parity cannot be used to detect two-bit errors, which is a distinct possibility in data storage media such as magnetic tapes. Secondly, the single-bit parity code cannot be used to localize or identify the error bit even if one bit is in error. There are several codes that provide self-single-bit error detection and correction mechanisms, and these are discussed below.

2.6.2 Repetition Code

The repetition code makes use of repetitive transmission of each data bit in the bit stream. In the case of threefold repetition, '1' and '0' would be transmitted as '111' and '000' respectively. If, in the received data bit stream, bits are examined in groups of three bits, the occurrence of an error can be detected. In the case of single-bit errors, '1' would be received as 011 or 101 or 110 instead of 111, and a '0' would be received as 100 or 010 or 001 instead of 000. In both cases, the code becomes self-correcting if the bit in the majority is taken as the correct bit. There are various forms in which the data are sent using the repetition code. Usually, the data bit stream is broken into blocks of bits, and then each block of data is sent some predetermined number of times. For example, if we want to send eight-bit data given by 11011001, it may be broken into two blocks of four bits each. In the case of threefold repetition, the transmitted data bit stream would be 111111011101100110011001. However, such a repetition code where the bit or block of bits is repeated 3 times is not capable of correcting two-bit errors, although it can detect the occurrence of error. For this, we have to increase the number of times each bit in the bit stream needs to be repeated. For example, by repeating each data bit 5 times, we can detect and correct all two-bit errors. The repetition code is highly inefficient and the information throughput drops rapidly as we increase the number of times each data bit needs to be repeated to build error detection and correction capability.

2.6.3 Cyclic Redundancy Check Code

Cyclic redundancy check (CRC) codes provide a reasonably high level of protection at low redundancy level. The cycle code for a given data word is generated as follows. The data word is first appended by a number of 0s equal to the number of check bits to be added. This new data bit sequence is then divided by a special binary word whose length equals $n+1$, n being the number of check bits to be added. The remainder obtained as a result of modulo-2 division is then added to the dividend bit

sequence to get the cyclic code. The code word so generated is completely divisible by the divisor used in the generation of the code. Thus, when the received code word is again divided by the same divisor, an error-free reception should lead to an all '0' remainder. A nonzero remainder is indicative of the presence of errors.

The probability of error detection depends upon the number of check bits, n, used to construct the cyclic code. It is 100 % for single-bit and two-bit errors. It is also 100 % when an odd number of bits are in error and the error bursts have a length less than $n + 1$. The probability of detection reduces to $1 - (1/2)^{n-1}$ for an error burst length equal to $n + 1$, and to $1 - (1/2)^n$ for an error burst length greater than $n + 1$.

2.6.4 Hamming Code

We have seen, in the case of the error detection and correction codes described above, how an increase in the number of redundant bits added to message bits can enhance the capability of the code to detect and correct errors. If we have a sufficient number of redundant bits, and if these bits can be arranged such that different error bits produce different error results, then it should be possible not only to detect the error bit but also to identify its location. In fact, the addition of redundant bits alters the 'distance' code parameter, which has come to be known as the Hamming distance. The Hamming distance is nothing but the number of bit disagreements between two code words. For example, the addition of single-bit parity results in a code with a Hamming distance of at least 2. The smallest Hamming distance in the case of a threefold repetition code would be 3. Hamming noticed that an increase in distance enhanced the code's ability to detect and correct errors. Hamming's code was therefore an attempt at increasing the Hamming distance and at the same time having as high an information throughput rate as possible.

The algorithm for writing the generalized Hamming code is as follows:

1. The generalized form of code is $P_1P_2D_1P_3D_2D_3D_4P_4D_5D_6D_7D_8D_9D_{10}D_{11}P_5\ldots$, where P and D respectively represent parity and data bits.
2. We can see from the generalized form of the code that all bit positions that are powers of 2 (positions 1, 2, 4, 8, 16, . . .) are used as parity bits.
3. All other bit positions (positions 3, 5, 6, 7, 9, 10, 11, . . .) are used to encode data.
4. Each parity bit is allotted a group of bits from the data bits in the code word, and the value of the parity bit (0 or 1) is used to give it certain parity.
5. Groups are formed by first checking $N-1$ bits and then alternately skipping and checking N bits following the parity bit. Here, N is the position of the parity bit; 1 for P_1, 2 for P_2, 4 for P_3, 8 for P_4 and so on. For example, for the generalized form of code given above, various groups of bits formed with different parity bits would be $P_1D_1D_2D_4D_5\ldots$, $P_2D_1D_3D_4D_6D_7\ldots$, $P_3D_2D_3D_4D_8D_9\ldots$, $P_4D_5D_6D_7D_8D_9D_{10}D_{11}\ldots$ and so on. To illustrate the formation of groups further, let us examine the group corresponding to parity bit P_3. Now, the position of P_3 is at number 4. In order to form the group, we check the first three bits ($N-1=3$) and then follow it up by alternately skipping and checking four bits ($N=4$).

The Hamming code is capable of correcting single-bit errors on messages of any length. Although the Hamming code can detect two-bit errors, it cannot give the error locations. The number of parity bits required to be transmitted along with the message, however, depends upon the message length, as shown above. The number of parity bits n required to encode m message bits is the smallest integer that satisfies the condition $(2^n - n) > m$.

Table 2.9 Generation of Hamming code.

	P_1	P_2	D_1	P_3	D_2	D_3	D_4
Data bits (without parity)			0		1	1	0
Data bits with parity bit P_1	1		0		1		0
Data bits with parity bit P_2		1	0			1	0
Data bits with parity bit P_3				0	1	1	0
Data bits with parity	1	1	0	0	1	1	0

The most commonly used Hamming code is the one that has a code word length of seven bits with four message bits and three parity bits. It is also referred to as the Hamming (7, 4) code. The code word sequence for this code is written as $P_1P_2D_1P_3D_2D_3D_4$, with P_1, P_2 and P_3 being the parity bits and D_1, D_2, D_3 and D_4 being the data bits. We will illustrate step by step the process of writing the Hamming code for a certain group of message bits and then the process of detection and identification of error bits with the help of an example. We will write the Hamming code for the four-bit message 0110 representing numeral '6'. The process of writing the code is illustrated in Table 2.9, with even parity.

Thus, the Hamming code for 0110 is 1100110. Let us assume that the data bit D_1 gets corrupted in the transmission channel. The received code in that case is 1110110. In order to detect the error, the parity is checked for the three parity relations mentioned above. During the parity check operation at the receiving end, three additional bits X, Y and Z are generated by checking the parity status of $P_1D_1D_2D_4$, $P_2D_1D_3D_4$ and $P_3D_2D_3D_4$ respectively. These bits are a '0' if the parity status is okay, and a '1' if it is disturbed. In that case, ZYX gives the position of the bit that needs correction. The process can be best explained with the help of an example.

Examination of the first parity relation gives $X = 1$ as the even parity is disturbed. The second parity relation yields $Y = 1$ as the even parity is disturbed here too. Examination of the third relation gives $Z = 0$ as the even parity is maintained. Thus, the bit that is in error is positioned at 011 which is the binary equivalent of '3'. This implies that the third bit from the MSB needs to be corrected. After correcting the third bit, the received message becomes 1100110 which is the correct code.

Example 2.6

By writing the parity code (even) and threefold repetition code for all possible four-bit straight binary numbers, prove that the Hamming distance in the two cases is at least 2 in the case of the parity code and 3 in the case of the repetition code.

Solution

The generation of codes is shown in Table 2.10. An examination of the parity code numbers reveals that the number of bit disagreements between any pair of code words is not less than 2. It is either 2 or 4. It is 4, for example, between 00000 and 10111, 00000 and 11011, 00000 and 11101, 00000 and 11110 and 00000 and 01111. In the case of the threefold repetition code, it is either 3, 6, 9 or 12 and therefore not less than 3 under any circumstances.

Example 2.7

It is required to transmit letter 'A' expressed in the seven-bit ASCII code with the help of the Hamming (11, 7) code. Given that the seven-bit ASCII notation for 'A' is 1000001 and that the data word gets

Table 2.10 Example 2.6.

Binary number	Parity code	Three-time repetition Code	Binary number	Parity code	Three-time repetition code
0000	00000	000000000000	1000	11000	100010001000
0001	10001	000100010001	1001	01001	100110011001
0010	10010	001000100010	1010	01010	101010101010
0011	00011	001100110011	1011	11011	101110111011
0100	10100	010001000100	1100	01100	110011001100
0101	00101	010101010101	1101	11101	110111011101
0110	00110	011001100110	1110	11110	111011101110
0111	10111	011101110111	1111	01111	111111111111

corrupted to 1010001 in the transmission channel, show how the Hamming code can be used to identify the error. Use even parity.

Solution

- The generalized form of the Hamming code in this case is $P_1P_2D_1P_3D_2D_3D_4P_4D_5D_6D_7 = P_1P_21P_3000P_4001$.
- The four groups of bits using different parity bits are $P_1D_1D_2D_4D_5D_7$, $P_2D_1D_3D_4D_6D_7$, $P_3D_2D_3D_4$ and $P_4D_5D_6D_7$.
- This gives $P_1 = 0$, $P_2 = 0$, $P_3 = 0$ and $P_4 = 1$.
- Therefore, the transmitted Hamming code for 'A' is 00100001001.
- The received Hamming code is 00100101001.
- Checking the parity for the P_1 group gives '0' as it passes the test.
- Checking the parity for the P_2 group gives '1' as it fails the test.
- Checking the parity for the P_3 group gives '1' as it fails the test.
- Checking the parity for the P_4 group gives '0' as it passes the test.
- The bits resulting from the parity check, written in reverse order, constitute 0110, which is the binary equivalent of '6'. This shows that the bit in error is the sixth from the MSB.
- Therefore, the corrected Hamming code is 00100001001, which is the same as the transmitted code.
- The received data word is 1000001.

Review Questions

1. Distinguish between weighted and unweighted codes. Give two examples each of both types of code.
2. What is an excess-3 BCD code? Which shortcoming of the 8421 BCD code is overcome in the excess-3 BCD code? Illustrate with the help of an example.
3. What is the Gray code? Why is it also known as the binary-reflected Gray code? Briefly outline some of the important applications of the Gray code.
4. Briefly describe salient features of the ASCII and EBCDIC codes in terms of their capability to represent characters and suitability for their use in different platforms.
5. What is the Unicode? Why is it called the most complete character code?

6. What is a parity bit? Define even and odd parity. What is the limitation of the parity code when it comes to detection and correction of bit errors?
7. What is the Hamming distance? What is the role of the Hamming distance in deciding the error detection and correction capability of a code meant for the purpose? How does it influence the information throughput rate?
8. With the help of the generalized form of the Hamming code, explain how the number of parity bits required to transmit a given number of data bits is decided upon.

Problems

1. Write the excess-3 equivalent codes of $(6)_{10}$, $(78)_{10}$ and $(357)_{10}$, all in 16-bit format.
0011001100111001, 0011001110101011, 0011011010001010
2. Determine the Gray code equivalent of $(10011)_2$ and the binary equivalent of the Gray code number 110011.
11010, $(100010)_2$
3. A 16-bit data word given by 1001100001110110 is to be transmitted by using a fourfold repetition code. If the data word is broken into four blocks of four bits each, then write the transmitted bit stream.
1001100110011001100010001000100001110111011101110111001100110011
4. Write (a) the Hamming (7, 4) code for 0000 using even parity and (b) the Hamming (11, 7) code for 1111111 using odd parity.
(a) 0000000; (b) 00101110111
5. Write the last four of the 16 possible numbers in the two-bit quaternary Gray code with 0, 1, 2 and 3 as its independent digits, beginning with the thirteenth number.
33, 32, 31, 30

Further Reading

1. Tokheim, R. L. (1994) *Schaum's Outline Series of Digital Principles*, McGraw-Hill Book Companies Inc., USA.
2. Gillam, R. (2002) *Unicode Demystified: A Practical Programmer's Guide to the Encoding Standard*, 1st edition, Addison-Wesley Professional, Boston, MA, USA.
3. MacWilliams, F. J. and Sloane, N. J. A. (2006) *The Theory of Error-Correcting Codes*, North-Holland Mathematical Library, Elsevier Ltd, Oxford, UK.
4. Huffman, W. C. and Pless, V. (2003) *Fundamentals of Error-Correcting Codes*, Cambridge University Press, Cambridge, UK.

3

Digital Arithmetic

Having discussed different methods of numeric and alphanumeric data representation in the first two chapters, the next obvious step is to study the rules of data manipulation. Two types of operation that are performed on binary data include arithmetic and logic operations. Basic arithmetic operations include addition, subtraction, multiplication and division. AND, OR and NOT are the basic logic functions. While the rules of arithmetic operations are covered in the present chapter, those related to logic operations will be discussed in the next chapter.

3.1 Basic Rules of Binary Addition and Subtraction

The basic principles of binary addition and subtraction are similar to what we all know so well in the case of the decimal number system. In the case of addition, adding '0' to a certain digit produces the same digit as the sum, and, when we add '1' to a certain digit or number in the decimal number system, the result is the next higher digit or number, as the case may be. For example, 6 + 1 in decimal equals '7' because '7' immediately follows '6' in the decimal number system. Also, 7 + 1 in octal equals '10' as, in the octal number system, the next adjacent higher number after '7' is '10'. Similarly, 9 + 1 in the hexadecimal number system is 'A'. With this background, we can write the basic rules of binary addition as follows:

1. $0 + 0 = 0$.
2. $0 + 1 = 1$.
3. $1 + 0 = 1$.
4. $1 + 1 = 0$ with a carry of '1' to the next more significant bit.
5. $1 + 1 + 1 = 1$ with a carry of '1' to the next more significant bit.

Table 3.1 summarizes the sum and carry outputs of all possible three-bit combinations. We have taken three-bit combinations as, in all practical situations involving the addition of two larger bit

Digital Electronics: Principles, Devices and Applications Anil K. Maini
 ISBN: 978-0-470-03214-5

Table 3.1 Binary addition of three bits.

A	B	Carry-in (C_{in})	Sum	Carry-out (C_o)	*A*	*B*	Carry-in (C_{in})	Sum	Carry-out (C_o)
0	0	0	0	0	1	0	0	1	0
0	0	1	1	0	1	0	1	0	1
0	1	0	1	0	1	1	0	0	1
0	1	1	0	1	1	1	1	1	1

numbers, we need to add three bits at a time. Two of the three bits are the bits that are part of the two binary numbers to be added, and the third bit is the carry-in from the next less significant bit column.

The basic principles of binary subtraction include the following:

1. $0 - 0 = 0$.
2. $1 - 0 = 1$.
3. $1 - 1 = 0$.
4. $0 - 1 = 1$ with a borrow of 1 from the next more significant bit.

The above-mentioned rules can also be explained by recalling rules for subtracting decimal numbers. Subtracting '0' from any digit or number leaves the digit or number unchanged. This explains the first two rules. Subtracting '1' from any digit or number in decimal produces the immediately preceding digit or number as the answer. In general, the subtraction operation of larger-bit binary numbers also involves three bits, including the two bits involved in the subtraction, called the minuend (the upper bit) and the subtrahend (the lower bit), and the borrow-in. The subtraction operation produces the difference output and borrow-out, if any. Table 3.2 summarizes the binary subtraction operation. The entries in Table 3.2 can be explained by recalling the basic rules of binary subtraction mentioned above, and that the subtraction operation involving three bits, that is, the minuend (A), the subtrahend (B) and the borrow-in (B_{in}), produces a difference output equal to ($A - B - B_{in}$). It may be mentioned here that, in the case of subtraction of larger-bit binary numbers, the least significant bit column always involves two bits to produce a difference output bit and the borrow-out

Table 3.2 Binary subtraction.

Inputs			Outputs	
Minuend (A)	Subtrahend (B)	Borrow-in (B_{in})	Difference (D)	Borrow-out (B_o)
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

bit. The borrow-out bit produced here becomes the borrow-in bit for the next more significant bit column, and the process continues until we reach the most significant bit column. The addition and subtraction of larger-bit binary numbers is illustrated with the help of examples in sections 3.2 and 3.3 respectively.

3.2 Addition of Larger-Bit Binary Numbers

The addition of larger binary integers, fractions or mixed binary numbers is performed columnwise in just the same way as in the case of decimal numbers. In the case of binary numbers, however, we follow the basic rules of addition of two or three binary digits, as outlined earlier. The process of adding two larger-bit binary numbers can be best illustrated with the help of an example.

Consider two generalized four-bit binary numbers $(A_3\,A_2\,A_1\,A_0)$ and $(B_3\,B_2\,B_1\,B_0)$, with A_0 and B_0 representing the LSB and A_3 and B_3 representing the MSB of the two numbers. The addition of these two numbers is performed as follows. We begin with the LSB position. We add the LSB bits and record the sum S_0 below these bits in the same column and take the carry C_0, if any, to the next column of bits. For instance, if $A_0 = 1$ and $B_0 = 0$, then $S_0 = 1$ and $C_0 = 0$. Next we add the bits A_1 and B_1 and the carry C_0 from the previous addition. The process continues until we reach the MSB bits. The four steps are shown ahead. C_0, C_1, C_2 and C_3 are carrys, if any, produced as a result of adding first, second, third and fourth column bits respectively, starting from LSB and proceeding towards MSB. A similar procedure is followed when the given numbers have both integer as well as fractional parts:

1.

			(C_0)	
A_3	A_2	A_1	A_0	
B_3	B_2	B_1	B_0	
			S_0	

2.

		(C_1)	(C_0)	
A_3	A_2	A_1	A_0	
B_3	B_2	B_1	B_0	
		S_1	S_0	

3.

(C_2)	(C_1)	(C_0)		
A_3	A_2	A_1	A_0	
B_3	B_2	B_1	B_0	
	S_2	S_1	S_0	

4.

	(C_2)	(C_1)	(C_0)	
A_3	A_2	A_1	A_0	
B_3	B_2	B_1	B_0	
C_3	S_3	S_2	S_1	S_0

3.2.1 Addition Using the 2's Complement Method

The 2's complement is the most commonly used code for processing positive and negative binary numbers. It forms the basis of arithmetic circuits in modern computers. When the decimal numbers to be added are expressed in 2's complement form, the addition of these numbers, following the basic laws of binary addition, gives correct results. Final carry obtained, if any, while adding MSBs should be disregarded. To illustrate this, we will consider the following four different cases:

1. Both the numbers are positive.
2. Larger of the two numbers is positive.
3. The larger of the two numbers is negative.
4. Both the numbers are negative.

Case 1

- Consider the decimal numbers +37 and +18.
- The 2's complement of +37 in eight-bit representation = 00100101.
- The 2's complement of +18 in eight-bit representation = 00010010.
- The addition of the two numbers, that is, +37 and +18, is performed as follows

$$\begin{array}{r} 00100101 \\ +\ 00010010 \\ \hline 00110111 \end{array}$$

- The decimal equivalent of $(00110111)_2$ is (+55), which is the correct answer.

Case 2

- Consider the two decimal numbers +37 and -18.
- The 2's complement representation of +37 in eight-bit representation = 00100101.
- The 2's complement representation of −18 in eight-bit representation = 11101110.
- The addition of the two numbers, that is, +37 and −18, is performed as follows:

$$\begin{array}{r} 00100101 \\ +\ 11101110 \\ \hline 00010011 \end{array}$$

- The final carry has been disregarded.
- The decimal equivalent of $(00010011)_2$ is +19, which is the correct answer.

Case 3

- Consider the two decimal numbers +18 and −37.
- −37 in 2's complement form in eight−bit representation = 11011011.
- +18 in 2's complement form in eight−bit representation = 00010010.
- The addition of the two numbers, that is, −37 and +18, is performed as follows:

$$\begin{array}{r} 11011011 \\ +\ 00010010 \\ \hline 11101101 \end{array}$$

- The decimal equivalent of $(11101101)_2$, which is in 2's complement form, is −19, which is the correct answer. 2's complement representation was discussed in detail in Chapter 1 on number systems.

Case 4

- Consider the two decimal numbers −18 and −37.
- −18 in 2's complement form is 11101110.
- −37 in 2's complement form is 11011011.
- The addition of the two numbers, that is, −37 and −18, is performed as follows:

$$\begin{array}{r} 11011011 \\ +\ 11101110 \\ \hline 11001001 \end{array}$$

- The final carry in the ninth bit position is disregarded.
- The decimal equivalent of $(11001001)_2$, which is in 2's complement form, is -55, which is the correct answer.

It may also be mentioned here that, in general, 2's complement notation can be used to perform addition when the expected result of addition lies in the range from -2^{n-1} to $+(2^{n-1}-1)$, n being the number of bits used to represent the numbers. As an example, eight-bit 2's complement arithmetic cannot be used to perform addition if the result of addition lies outside the range from -128 to $+127$. Different steps to be followed to do addition in 2's complement arithmetic are summarized as follows:

1. Represent the two numbers to be added in 2's complement form.
2. Do the addition using basic rules of binary addition.
3. Disregard the final carry, if any.
4. The result of addition is in 2's complement form.

Example 3.1

Perform the following addition operations:

1. $(275.75)_{10} + (37.875)_{10}$.
2. $(AF1.B3)_{16} + (FFF.E)_{16}$.

Solution

1. As a first step, the two given decimal numbers will be converted into their equivalent binary numbers (decimal-to-binary conversion has been covered at length in Chapter 1, and therefore the decimal-to-binary conversion details will not be given here):

$$(275.75)_{10} = (100010011.11)_2 \text{ and } (37.875)_{10} = (100101.111)_2$$

The two binary numbers can be rewritten as $(100010011.110)_2$ and $(000100101.111)_2$ to have the same number of bits in their integer and fractional parts. The addition of two numbers is performed as follows:

$$\begin{array}{r} 100010011.110 \\ 000100101.111 \\ \hline 100111001.101 \end{array}$$

The decimal equivalent of $(100111001.101)_2$ is $(313.625)_{10}$.

2. $(AF1.B3)_{16} = (1010111100001.10110011)_2$ and $(FFF.E)_{16} = (1111111111111.1110)_2$. $(1111111111.1110)_2$ can also be written as $(1111111111111.11100000)_2$ to have the same number of bits in the integer and fractional parts. The two numbers can now be added as follows:

$$\begin{array}{r} 0101011110001.10110011 \\ 0111111111111.11100000 \\ \hline 1101011110001.10010011 \end{array}$$

The hexadecimal equivalent of $(1101011110001.10010011)_2$ is $(1AF1.93)_{16}$, which is equal to the hex addition of $(AF1.B3)_{16}$ and $(FFF.E)_{16}$.

Example 3.2

Find out whether 16-bit 2's complement arithmetic can be used to add 14 276 and 18 490.

Solution

The addition of decimal numbers 14 276 and 18 490 would yield 32 766. 16-bit 2's complement arithmetic has a range of -2^{15} to $+(2^{15} - 1)$, i.e. $-32\,768$ to $+32\,767$. The expected result is inside the allowable range. Therefore, 16-bit arithmetic can be used to add the given numbers.

Example 3.3

Add -118 and -32 firstly using eight-bit 2's complement arithmetic and then using 16-bit 2's complement arithmetic. Comment on the results.

Solution

- -118 in eight-bit 2's complement representation = 10001010.
- -32 in eight-bit 2's complement representation = 11100000.
- The addition of the two numbers, after disregarding the final carry in the ninth bit position, is 01101010. Now, the decimal equivalent of $(01101010)_2$, which is in 2's complement form, is $+106$. The reason for the wrong result is that the expected result, i.e. -150, lies outside the range of eight-bit 2's complement arithmetic. Eight-bit 2's complement arithmetic can be used when the expected result lies in the range from -2^7 to $+ (2^7 - 1)$, i.e. -128 to $+127$. -118 in 16-bit 2's complement representation = 1111111110001010.
- -32 in 16-bit 2's complement representation = 1111111111100000.
- The addition of the two numbers, after disregarding the final carry in the 17th position, produces 1111111101101010. The decimal equivalent of $(1111111101101010)_2$, which is in 2's complement form, is -150, which is the correct answer. 16-bit 2's complement arithmetic has produced the correct result, as the expected result lies within the range of 16-bit 2's complement notation.

3.3 Subtraction of Larger-Bit Binary Numbers

Subtraction is also done columnwise in the same way as in the case of the decimal number system. In the first step, we subtract the LSBs and subsequently proceed towards the MSB. Wherever the subtrahend (the bit to be subtracted) is larger than the minuend, we borrow from the next adjacent

higher bit position having a '1'. As an example, let us go through different steps of subtracting $(1001)_2$ from $(1100)_2$.

In this case, '1' is borrowed from the second MSB position, leaving a '0' in that position. The borrow is first brought to the third MSB position to make it '10'. Out of '10' in this position, '1' is taken to the LSB position to make '10' there, leaving a '1' in the third MSB position. $10-1$ in the LSB column gives '1', $1-0$ in the third MSB column gives '1', $0-0$ in the second MSB column gives '0' and $1-1$ in the MSB also gives '0' to complete subtraction. Subtraction of mixed numbers is also done in the same manner. The above-mentioned steps are summarized as follows:

1.	**1**	**1**	**0**	**0**	**2.**	**1**	**1**	**0**	**0**
	1	**0**	**0**	**1**		**1**	**0**	**0**	**1**
				1				**1**	**1**
3.	**1**	**1**	**0**	**0**	**4.**	**1**	**1**	**0**	**0**
	1	**0**	**0**	**1**		**1**	**0**	**0**	**1**
		0	**1**	**1**		**0**	**0**	**1**	**1**

3.3.1 Subtraction Using 2's Complement Arithmetic

Subtraction is similar to addition. Adding 2's complement of the subtrahend to the minuend and disregarding the carry, if any, achieves subtraction. The process is illustrated by considering six different cases:

1. Both minuend and subtrahend are positive. The subtrahend is the smaller of the two.
2. Both minuend and subtrahend are positive. The subtrahend is the larger of the two.
3. The minuend is positive. The subtrahend is negative and smaller in magnitude.
4. The minuend is positive. The subtrahend is negative and greater in magnitude.
5. Both minuend and subtrahend are negative. The minuend is the smaller of the two.
6. Both minuend and subtrahend are negative. The minuend is the larger of the two.

Case 1

- Let us subtract $+14$ from $+24$.
- The 2's complement representation of $+24 = 00011000$.
- The 2's complement representation of $+14 = 00001110$.
- Now, the 2's complement of the subtrahend (i.e. $+14$) is 11110010.
- Therefore, $+24 - (+14)$ is given by

$$\begin{array}{r} 00011000 \\ +\ 11110010 \\ \hline \underline{00001010} \end{array}$$

  with the final carry disregarded.
- The decimal equivalent of $(00001010)_2$ is $+10$, which is the correct answer.

Case 2

- Let us subtract +24 from +14.
- The 2's complement representation of +14 = 00001110.
- The 2's complement representation of +24 = 00011000.
- The 2's complement of the subtrahend (i.e. +24) = 11101000.
- Therefore, +14 − (+24) is given by

$$\begin{array}{r} 00001110 \\ +\ 11101000 \\ \hline 11110110 \end{array}$$

- The decimal equivalent of $(11110110)_2$, which is of course in 2's complement form, is −10 which is the correct answer.

Case 3

- Let us subtract −14 from +24.
- The 2's complement representation of +24 = 00011000 = minuend.
- The 2's complement representation of −14 = 11110010 = subtrahend.
- The 2's complement of the subtrahend (i.e. −14) = 00001110.
- Therefore, +24 − (−14) is performed as follows:

$$\begin{array}{r} 00011000 \\ +\ 00001110 \\ \hline 00100110 \end{array}$$

- The decimal equivalent of $(00100110)_2$ is +38, which is the correct answer.

Case 4

- Let us subtract −24 from +14.
- The 2's complement representation of +14 = 00001110 = minuend.
- The 2's complement representation of −24 = 11101000 = subtrahend.
- The 2's complement of the subtrahend (i.e. −24) = 00011000.
- Therefore, +14 − (−24) is performed as follows:

$$\begin{array}{r} 00001110 \\ +\ 00011000 \\ \hline 00100110 \end{array}$$

- The decimal equivalent of $(00100110)_2$ is +38, which is the correct answer.

Case 5

- Let us subtract −14 from −24.
- The 2's complement representation of −24 = 11101000 = minuend.

- The 2's complement representation of -14=11110010 = subtrahend.
- The 2's complement of the subtrahend = 00001110.
- Therefore, $-24 - (-14)$ is given as follows:

$$\begin{array}{r} 11101000 \\ +\,00001110 \\ \hline 11110110 \end{array}$$

- The decimal equivalent of $(11110110)_2$, which is in 2's complement form, is -10, which is the correct answer.

Case 6

- Let us subtract -24 from -14.
- The 2's complement representation of -14 = 11110010 = minuend.
- The 2's complement representation of -24=11101000 = subtrahend.
- The 2's complement of the subtrahend = 00011000.
- Therefore, $-14 - (-24)$ is given as follows:

$$\begin{array}{r} 11110010 \\ +\,00011000 \\ \hline 00001010 \end{array}$$

with the final carry disregarded.

- The decimal equivalent of $(00001010)_2$, which is in 2's complement form, is $+10$, which is the correct answer.

It may be mentioned that, in 2's complement arithmetic, the answer is also in 2's complement notation, only with the MSB indicating the sign and the remaining bits indicating the magnitude. In 2's complement notation, positive magnitudes are represented in the same way as the straight binary numbers, while the negative magnitudes are represented as the 2's complement of their straight binary counterparts. A '0' in the MSB position indicates a positive sign, while a '1' in the MSB position indicates a negative sign.

The different steps to be followed to do subtraction in 2's complement arithmetic are summarized as follows:

1. Represent the minuend and subtrahend in 2's complement form.
2. Find the 2's complement of the subtrahend.
3. Add the 2's complement of the subtrahend to the minuend.
4. Disregard the final carry, if any.
5. The result is in 2's complement form.
6. 2's complement notation can be used to perform subtraction when the expected result of subtraction lies in the range from -2^{n-1} to $+(2^{n-1} - 1)$, n being the number of bits used to represent the numbers.

Example 3.4

Subtract $(1110.011)_2$ from $(11011.11)_2$ using basic rules of binary subtraction and verify the result by showing equivalent decimal subtraction.

Solution

The minuend and subtrahend are first modified to have the same number of bits in the integer and fractional parts. The modified minuend and subtrahend are $(11011.110)_2$ and $(01110.011)_2$ respectively:

$$\begin{array}{r} 11011.110 \\ -\ 01110.011 \\ \hline 01101.011 \end{array}$$

The decimal equivalents of $(11011.110)_2$ and $(01110.011)_2$ are 27.75 and 14.375 respectively. Their difference is 13.375, which is the decimal equivalent of $(01101.011)_2$.

Example 3.5

Subtract (a) $(-64)_{10}$ from $(+32)_{10}$ and (b) $(29.A)_{16}$ from $(4F.B)_{16}$. Use 2's complement arithmetic.

Solution:

(a) $(+32)_{10}$ in 2's complement notation $= (00100000)_2$.
$(-64)_{10}$ in 2's complement notation $= (11000000)_2$.
The 2's complement of $(-64)_{10} = (01000000)_2$.
$(+32)_{10} - (-64)_{10}$ is determined by adding the 2's complement of $(-64)_{10}$ to $(+32)_{10}$.
Therefore, the addition of $(00100000)_2$ to $(01000000)_2$ should give the result. The operation is shown as follows:

$$\begin{array}{r} 00100000 \\ +\ 01000000 \\ \hline 01100000 \end{array}$$

The decimal equivalent of $(01100000)_2$ is $+96$, which is the correct answer as $+32 - (-64) = +96$.

(b) The minuend $= (4F.B)_{16} = (01001111.1011)_2$.
The minuend in 2's complement notation $= (01001111.1011)_2$.
The subtrahend $= (29.A)_{16} = (00101001.1010)_2$.
The subtrahend in 2's complement notation $= (00101001.1010)_2$.
The 2's complement of the subtrahend $= (11010110.0110)_2$.
$(4F.B)_{16} - (29.A)_{16}$ is given by the addition of the 2's complement of the subtrahend to the minuend.

$$\begin{array}{r} 01001111.1011 \\ +\ 11010110.0110 \\ \hline 00100110.0001 \end{array}$$

with the final carry disregarded. The result is also in 2's complement form. Since the result is a positive number, 2's complement notation is the same as it would be in the case of the straight binary code.
The hex equivalent of the resulting binary number $= (26.1)_{16}$, which is the correct answer.

3.4 BCD Addition and Subtraction in Excess-3 Code

Below, we will see how the excess-3 code can be used to perform addition and subtraction operations on BCD numbers.

3.4.1 Addition

The excess-3 code can be very effectively used to perform the addition of BCD numbers. The steps to be followed for excess-3 addition of BCD numbers are as follows:

1. The given BCD numbers are written in excess-3 form by adding '0011' to each of the four-bit groups.
2. The two numbers are then added using the basic laws of binary addition.
3. Add '0011' to all those four-bit groups that produce a carry, and subtract '0011' from all those four-bit groups that do not produce a carry during addition.
4. The result thus obtained is in excess-3 form.

3.4.2 Subtraction

Subtraction of BCD numbers using the excess-3 code is similar to the addition process discussed above. The steps to be followed for excess-3 substraction of BCD numbers are as follows:

1. Express both minuend and subtrahend in excess-3 code.
2. Perform subtraction following the basic laws of binary subtraction.
3. Subtract '0011' from each invalid BCD four-bit group in the answer.
4. Subtract '0011' from each BCD four-bit group in the answer if the subtraction operation of the relevant four-bit groups required a borrow from the next higher adjacent four-bit group.
5. Add '0011' to the remaining four-bit groups, if any, in the result.
6. This gives the result in excess-3 code.

The process of addition and subtraction can be best illustrated with the help of following examples.

Example 3.6

Add $(0011\ 0101\ 0110)_{BCD}$ and $(0101\ 0111\ 1001)_{BCD}$ using the excess-3 addition method and verify the result using equivalent decimal addition.

Solution

The excess-3 equivalents of 0011 0101 0110 and 0101 0111 1001 are 0110 1000 1001 and 1000 1010 1100 respectively. The addition of the two excess-3 numbers is given as follows:

$$\begin{array}{r} 0110\ 1000\ 1001 \\ 1000\ 1010\ 1100 \\ \hline 1111\ 0011\ 0101 \end{array}$$

After adding 0011 to the groups that produced a carry and subtracting 0011 from the groups that did not produce a carry, we obtain the result of the above addition as 1100 0110 1000. Therefore, 1100

0110 1000 represents the excess-3 code for the true result. The result in BCD code is 1001 0011 0101, which is the BCD equivalent of 935. This is the correct answer as the addition of the given BCD numbers 0011 0101 0110 = $(356)_{10}$ and 0101 0111 1001 = $(579)_{10}$ yields $(935)_{10}$ only.

Example 3.7

Perform $(185)_{10} - (8)_{10}$ *using the excess-3 code.*

Solution

- $(185)_{10}$ = $(0001\ 1000\ 0101)_{BCD}$.The excess-3 equivalent of $(0001\ 1000\ 0101)_{BCD}$ = 0100 1011 1000.
- $(8)_{10} = (008)_{10} = (0000\ 0000\ 1000)_{BCD}$. The excess-3 equivalent of $(0000\ 0000\ 1000)_{BCD}$ = 0011 0011 1011.
- Subtraction is performed as follows:

$$\begin{array}{r} 0100\ 1011\ 1000 \\ -\ 0011\ 0011\ 1011 \\ \hline 0001\ 0111\ 1101 \\ \hline \end{array}$$

- In the subtraction operation, the least significant column of four-bit groups needed a borrow, while the other two columns did not need any borrow. Also, the least significant column has produced an invalid BCD code group. Subtracting 0011 from the result of this column and adding 0011 to the results of other two columns, we get 0100 1010 1010. This now constitutes the result of subtraction expressed in excess-3 code.
- The result in BCD code is therefore 0001 0111 0111.
- The decimal equivalent of 0001 0111 0111 is 177, which is the correct result.

3.5 Binary Multiplication

The basic rules of binary multiplication are governed by the way an AND gate functions when the two bits to be multiplied are fed as inputs to the gate. Logic gates are discussed in detail in the next chapter. As of now, it would suffice to say that the result of multiplying two bits is the same as the output of the AND gate with the two bits applied as inputs to the gate. The basic rules of multiplication are listed as follows:

1. $0 \times 0 = 0$.
2. $0 \times 1 = 0$.
3. $1 \times 0 = 0$.
4. $1 \times 1 = 1$.

One of the methods for multiplication of larger-bit binary numbers is similar to what we are familiar with in the case of decimal numbers. This is called the 'repeated left-shift and add' algorithm. Microprocessors and microcomputers, however, use what is known as the 'repeated add and right-shift' algorithm to do binary multiplication as it is comparatively much more convenient to implement than the 'repeated left-shift and add' algorithm. The two algorithms are briefly described below. Also, binary multiplication of mixed binary numbers is done by performing multiplication without considering the

binary point. Starting from the LSB, the binary point is then placed after n bits, where n is equal to the sum of the number of bits in the fractional parts of the multiplicand and multiplier.

3.5.1 Repeated Left-Shift and Add Algorithm

In the 'repeated left-shift and add' method of binary multiplication, the end-product is the sum of several partial products, with the number of partial products being equal to the number of bits in the multiplier binary number. This is similar to the case of decimal multiplication. Each successive partial product after the first is shifted one digit to the left with respect to the immediately preceding partial product. In the case of binary multiplication too, the first partial product is obtained by multiplying the multiplicand binary number by the LSB of the multiplier binary number. The second partial product is obtained by multiplying the multiplicand binary number by the next adjacent higher bit in the multiplier binary number and so on. We begin with the LSB of the multiplier to obtain the first partial product. If the LSB is a '1', a copy of the multiplicand forms the partial product, and it is an all '0' sequence if the LSB is a '0'. We proceed towards the MSB of the multiplier and obtain various partial products. The second partial product is shifted one bit position to the left relative to the first partial product; the third partial product is shifted one bit position to the left relative to the second partial product and so on. The addition of all partial products gives the final answer. If the multiplicand and multiplier have different signs, the end result has a negative sign, otherwise it is positive. The procedure is further illustrated by showing $(23)_{10} \times (6)_{10}$ multiplication.

$$
\begin{array}{rl}
 & 1\ 0\ 1\ 1\ 1 \\
\textit{Multiplicand}: & \times\ 1\ 1\ 0 \quad \ldots\ldots\ldots\ldots\ (23)_{10} \\
\textit{Multiplier}: & \hline \quad \ldots\ldots\ldots\ldots\ (6)_{10} \\
 & 0\ 0\ 0\ 0\ 0 \\
 & 1\ 0\ 1\ 1\ 1 \\
 & 1\ 0\ 1\ 1\ 1 \\
 & \hline \\
 & 1\ 0\ 0\ 0\ 1\ 0\ 1\ 0
\end{array}
$$

The decimal equivalent of $(10001010)_2$ is $(138)_{10}$, which is the correct result.

3.5.2 Repeated Add and Right-Shift Algorithm

The multiplication process starts with writing an all '0' bit sequence, with the number of bits equal to the number of bits in the multiplicand. This bit sequence (all '0' sequence) is added to another same-sized bit sequence, which is the same as the multiplicand if the LSB of the multiplier is a '1', and an all '0' sequence if it is a '0'. The result of the first addition is shifted one bit position to the right, and the bit shifted out is recorded. The vacant MSB position is replaced by a '0'. This new sequence is added to another sequence, which is an all '0' sequence if the next adjacent higher bit in the multiplier is a '0', and the same as the multiplicand if it is a '1'. The result of the second addition is also shifted one bit position to the right, and a new sequence is obtained. The process continues until all multiplier bits are exhausted. The result of the last addition together with the recorded bits constitutes the result of multiplication. We will illustrate the procedure by doing $(23)_{10} \times (6)_{10}$ multiplication again, this time by using the 'repeated add and right-shift' algorithm:

- The multiplicand $= (23)_{10} = (10111)_2$ and the multiplier $= (6)_{10} = (110)_2$. The multiplication process is shown in Table 3.3.
- Therefore, $(10111)_2 \times (110)_2 = (10001010)_2$.

Table 3.3 Multiplication using the repeated add and right-shift algorithm.

1 0 1 1 1 1 1 0	Multiplicand Multiplier
0 0 0 0 0 + 0 0 0 0 0	Start
0 0 0 0 0	Result of first addition
0 0 0 0 0 + 1 0 1 1 1	0 (Result of addition shifted one bit to right)
1 0 1 1 1	Result of second addition
0 1 0 1 1 + 1 0 1 1 1	10 (Result of addition shifted one bit to right)
1 0 0 0 1 0	Result of third addition
0 1 0 0 0 1	010 (Result of addition shifted one bit to right)

Example 3.8

Multiply (a) $(100.01)_2 \times (10.1)_2$ *by using the 'repeated add and left-shift' algorithm and (b)* $(2B)_{16} \times (3)_{16}$ *by using the 'add and right-shift' algorithm. Verify the results by showing equivalent decimal multiplication.*

Solution

(a) As a first step, we will multiply $(10001)_2$ by $(101)_2$. The process is shown as follows:

$$\begin{array}{r} 1\ 0\ 0\ 0\ 1 \\ \times\ 1\ 0\ 1 \\ \hline 1\ 0\ 0\ 0\ 1 \\ 0\ 0\ 0\ 0\ 0 \\ 1\ 0\ 0\ 0\ 1 \\ \hline 1\ 0\ 1\ 0\ 1\ 0\ 1 \\ \hline \end{array}$$

The multiplication result is then given by placing the binary point three bits after the LSB, which gives $(1010.101)_2$ as the final result. Also, $(100.01)_2 = (4.25)_{10}$ and $(10.1)_2 = (2.5)_{10}$. Moreover, $(4.25)_{10} \times (2.5)_{10} = (10.625)_{10}$ and $(1010.101)_2$ equals $(10.625)_{10}$, which verifies the result.

(b) $(2B)_{16} = 00101011 = 101011$ and $(3)_{16} = 0011 = 11$.
Different steps involved in the multiplication process are shown in Table 3.4.
The result of multiplication is therefore $(10000001)_2$. Also, $(2B)_{16} = (43)_{10}$ and $(3)_{16} = (3)_{10}$. Therefore, $(2B)_{16} \times (3)_{16} = (129)_{10}$. Moreover, $(10000001)_2 = (129)_{10}$, which verifies the result.

3.6 Binary Division

While binary multiplication is the process of repeated addition, binary division is the process of repeated subtraction. Binary division can be performed by using either the 'repeated right-shift and

Table 3.4 Example 3.8.

1 0 1 0 1 1	Multiplicand
1 1	Multiplier
0 0 0 0 0 0	Start
+ 1 0 1 0 1 1	
1 0 1 0 1 1	Result of first addition
0 1 0 1 0 1	1 (Result of addition shifted one bit to right)
+ 1 0 1 0 1 1	
1 0 0 0 0 0 0	Result of second addition
0 1 0 0 0 0 0	01 (Result of addition shifted one bit to right)

subtract' or the 'repeated subtract and left-shift' algorithm. These are briefly described and suitably illustrated in the following sections.

3.6.1 Repeated Right-Shift and Subtract Algorithm

The algorithm is similar to the case of conventional division with decimal numbers. At the outset, starting from MSB, we begin with the number of bits in the dividend equal to the number of bits in the divisor and check whether the divisor is smaller or greater than the selected number of bits in the dividend. If it happens to be greater, we record a '0' in the quotient column. If it is smaller, we subtract the divisor from the dividend bits and record a '1' in the quotient column. If it is greater and we have already recorded a '0', then, as a second step, we include the next adjacent bit in the dividend bits, shift the divisor to the right by one bit position and again make a similar check like the one made in the first step. If it is smaller and we have made the subtraction, then in the second step we append the next MSB of the dividend to the remainder, shift the divisor one bit to the right and again make a similar check. The options are again the same. The process continues until we have exhausted all the bits in the dividend. We will illustrate the algorithm with the help of an example. Let us consider the division of $(100110)_2$ by $(1100)_2$. The sequence of operations needed to carry out the above division is shown in Table 3.5. The quotient = 011 and the remainder = 10.

Table 3.5 Binary division using the repeated right-shift and subtract algorithm.

	Quotient		
First step	0	1 0 0 1 1 0	Dividend
		−1 1 0 0	Divisor
Second step	1	1 0 0 1 1	First five MSBs of dividend
		−1 1 0 0	Divisor shifted to right
		0 1 1 1	First subtraction remainder
Third step	1	0 1 1 1 0	Next MSB appended
		−1 1 0 0	Divisor right shifted
		0 0 1 0	Second subtraction remainder

Table 3.6 Binary division using the repeate subtract and left-shift algorithm.

Quotient	1 0 0 1 −1 1 0 0	1 0
0	1 1 0 1 +1 1 0 0	Borrow exists
	1 0 0 1	Final carry ignored
	1 0 0 1 1 −1 1 0 0	Next MSB appended
1	0 1 1 1	No borrow
	0 1 1 1 0 −1 1 0 0	Next MSB appended
1	0 0 0 1 0	No borrow

3.6.2 Repeated Subtract and Left-Shift Algorithm

The procedure can again be best illustrated with the help of an example. Let us consider solving the above problem using this algorithm. The steps needed to perform the division are as follows. We begin with the first four MSBs of the dividend, four because the divisor is four bits long. In the first step, we subtract the divisor from the dividend. If the subtraction requires borrow in the MSB position, enter a '0' in the quotient column; otherwise, enter a '1'. In the present case there exists a borrow in the MSB position, and so there is a '0' in the quotient column. If there is a borrow, the divisor is added to the result of subtraction. In doing so, the final carry, if any, is ignored. The next MSB is appended to the result of the first subtraction if there is no borrow, or to the result of subtraction, restored by adding the divisor, if there is a borrow. By appending the next MSB, the remaining bits of the dividend are one bit position shifted to the left. It is again compared with the divisor, and the process is repeated. It goes on until we have exhausted all the bits of the dividend. The final remainder can be further processed by successively appending 0s and trying subtraction to get fractional part bits of the quotient. The different steps are summarized in Table 3.6. The quotient = 011 and the remainder = 10.

Example 3.9

Use the 'repeated right-shift and subtract' algorithm to divide $(110101)_2$ by $(1011)_2$. Determine both the integer and the fractional parts of the quotient. The fractional part may be determined up to three bit places.

Solution

The sequence of operations is given in Table 3.7. The operations are self-explanatory.

- The quotient = 100.110.
- Now, $(110101)_2 = (53)_{10}$ and $(1011)_2 = (11)_{10}$.
- $(53)_{10}$ divided by $(11)_{10}$ gives $(4.82)_{10}$.
- $(100.110)_2 = (4.75)_{10}$, which matches with the expected result to a good approximation.

Table 3.7 Example 3.9.

	Quotient		
First step	1	1 1 0 1 0 1 −1 0 1 1	Dividend Divisor
		0 0 1 0	First subtraction
Second step	0	0 0 1 0 0 −1 0 1 1	Next MSB appended Divisor right shifted
Third step	0	0 0 1 0 0 1 −1 0 1 1	Next MSB appended Divisor right shifted
		0 0 1 0 0 1	All bits exhausted
	1	0 0 1 0 0 1 0 −1 0 1 1	'0' appended Divisor right shifted
		0 1 1 1	Second subtraction
Fourth step	1	0 1 1 1 0 −1 0 1 1	'0' appended Divisor right shifted
		0 0 0 1 1	Third subtraction
Fifth step	0	0 0 0 1 1 0 −1 0 1 1	'0' appended Divisor right shifted
		0 0 1 1	Fourth subtraction

Example 3.10

Use the 'repeated subtract and left-shift' algorithm to divide $(100011)_2$ by $(100)_2$ to determine both the integer and fractional parts of the quotient. Verify the result by showing equivalent decimal division. Determine the fractional part to two bit places.

Solution

The sequence of operations is given in Table 3.8. The operations are self-explanatory.

- The quotient $= (1000.11)_2 = (8.75)_{10}$.
- Now, $(100011)_2 = (35)_{10}$ and $(100)_2 = (4)_{10}$.
- $(35)_{10}$ divided by $(4)_{10}$ gives $(8.75)_{10}$ and hence is verified.

Example 3.11

Divide $(AF)_{16}$ by $(09)_{16}$ using the method of 'repeated right shift and subtract', bearing in mind the signs of the given numbers, assuming that we are working in eight-bit 2's complement arithmetic.

Solution

- The dividend $= (AF)_{16}$.
- As it is a negative hexadecimal number, the magnitude of this number is determined by its 2's complement (or more precisely by its 16's complement in hexadecimal number language).

Table 3.8 Example 3.10.

Quotient		
	1 0 0 −1 0 0	0 1 1 Dividend Divisor
1	0 0 0	No borrow
	0 0 0 0 −1 0 0	Next MSB appended
0	1 0 0 +1 0 0	Borrow exists
	0 0 0 0 0 0 1 −1 0 0	Final carry ignored Next MSB appended
0	1 0 1 + 1 0 0	Borrow exists
	0 0 1 0 0 1 1 − 1 0 0	Final carry ignored Next MSB appended
0	1 1 1 +1 0 0	Borrow exists
	0 1 1 0 1 1 0 − 1 0 0	Final carry ignored '0' appended
1	0 1 0	No borrow
	0 1 0 0 −1 0 0	'0' appended
1	0 0 0	No borrow

- The 16's complement of $(AF)_{16} = (51)_{16}$.
- The binary equivalent of $(51)_{16} = 01010001 = 1010001$.
- The divisor $= (09)_{16}$.
- It is a positive number.
- The binary equivalent of $(09)_{16} = 00001001$.
- As the dividend is a negative number and the divisor a positive number, the quotient will be a negative number. The division process using the 'repeated right-shift and subtract' algorithm is given in Table 3.9.
- The quotient $= 1001 = (09)_{16}$.
- As the quotient should be a negative number, its magnitude is given by the 16's complement of $(09)_{16}$, i.e. $(F7)_{16}$.
- Therefore, $(AF)_{16}$ divided by $(09)_{16}$ gives $(F7)_{16}$.

3.7 Floating-Point Arithmetic

Before performing arithmetic operations on floating-point numbers, it is necessary to make a few checks, such as finding the signs of the two mantissas, checking any possible misalignment of exponents, etc.

Table 3.9 Example 3.11

1	1 0 1 0 0 0 1 −1 0 0 1	Divisor less than dividend
	0 0 0 1	
0	0 0 0 1 0 −1 0 0 1	Divisor greater than dividend
0	0 0 0 1 0 0 −1 0 0 1	Divisor still greater
1	0 0 0 1 0 0 1 −1 0 0 1	Divisor less than dividend
	0 0 0 0 0 0 0	

For example, if the exponents of the two numbers are not equal, the addition and subtraction operations necessitate that they be made equal. In that case, the mantissa of the smaller of the two numbers is shifted right, and the exponent is incremented for each shift until the two exponents are equal. Once the binary points are aligned and the exponents made equal, addition and subtraction operations become straightforward. While doing subtraction, of course, a magnitude check is also required to determine the smaller of the two numbers.

3.7.1 Addition and Subtraction

If N_1 and N_2 are two floating-point numbers given by

$$N_1 = m_1 \times 2^e$$

$$N_2 = m_2 \times 2^e$$

then

$$N_1 + N_2 = m_1 \times 2^e + m_2 \times 2^e = (m_1 + m_2) \times 2^e$$

and

$$N_1 - N_2 = m_1 \times 2^e - m_2 \times 2^e = (m_1 - m_2) \times 2^e$$

The subtraction operation assumes that $N_1 > N_2$. Post-normalization of the result may be required after the addition or subtraction operation.

3.7.2 Multiplication and Division

In the case of multiplication of two floating-point numbers, the mantissas of the two numbers are multiplied and their exponents are added. In the case of a division operation, the mantissa of the

quotient is given by the division of the two mantissas (i.e. dividend mantissa divided by divisor mantissa) and the exponent of the quotient is given by subtraction of the two exponents (i.e. dividend exponent minus divisor exponent).

If

$$N_1 = m_1 \times 2^{e1} and\ N_2 = m_2 \times 2^{e2}$$

then

$$N_1 \times N_2 = (m_1 \times m_2) \times 2^{(e1+e2)}$$

and

$$N_1/N_2 = (m_1/m_2) \times 2^{(e1-e2)}$$

Again, post-normalization may be required after multiplication or division, as in the case of addition and subtraction operations.

Example 3.12

Add (a) $(39)_{10}$ and $(19)_{10}$ and (b) $(1E)_{16}$ and $(F3)_{16}$ using floating-point numbers. Verify the answers by performing equivalent decimal addition.

Solution

(a) $(39)_{10} = 100111 = 0.100111 \times 2^6$.
$(19)_{10} = 10011 = 0.10011 \times 2^5 = 0.010011 \times 2^6$.
Therefore, $(39)_{10} + (19)_{10} = 0.100111 \times 2^6 + 0.010011 \times 2^6$
$= (0.100111 + 0.010011) \times 2^6 = 0.111010 \times 2^6$
$= 111010 = (58)_{10}$
and hence is verified.

(b) $(1E)_{16} = (00011110)_2 = 0.00011110 \times 2^8$.
$(F3)_{16} = (11110011)_2 = 0.11110011 \times 2^8$.
$(1E)_{16} + (F3)_{16} = (0.00011110 + 0.11110011) \times 2^8 = 100010001$
$= 000100010001$
$= (111)_{16}$.
Also, $(1E)_{16} + (\mathrm{F3})_{16} = (111)_{16}$ and hence is proved.

Example 3.13

Subtract $(17)_8$ from $(21)_8$ using floating-point numbers and verify the answer.

Solution

- $(21)_8 = (010001)_2 = 0.010001 \times 2^6$.
- $(17)_8 = (001111)_2 = 0.001111 \times 2^6$.
- Therefore, $(21)_8 - (17)_8 = (0.010001 - 0.001111) \times 2^6$
$= 0.000010 \times 2^6 = 000010 = (02)_8$.
- Also, $(21)_8 - (17)_8 = (02)_8$ and hence is verified.

Example 3.14

Multiply $(37)_{10}$ by $(10)_{10}$ using floating-point numbers. Verify by showing equivalent decimal multiplication.

Solution

- The multiplicand $= (37)_{10} = (100101)_2 = 0.100101 \times 2^6$.
- The multiplier $= (10)_{10} = (1010)_2 = 0.1010 \times 2^4$.
- $(37)_{10} \times (10)_{10} = (0.100101 \times 0.1010) \times 2^{10} = 0.0101110010 \times 2^{10} = 101110010$
  $= (370)_{10}$ and hence is verified.

Example 3.15

Perform $(E3B)_{16} \div (1A)_{16}$ using binary floating-point numbers. Verify by showing equivalent decimal division.

Solution

- Dividend $= (E3B)_{16} = (111000111011)_2 = 0.111000111011 \times 2^{12}$.
- Divisor $= (1A)_{16} = (00011010)_2 = (11010)_2 = 0.11010 \times 2^5$.
- Therefore, $(E3B)_{16} \div (1A)_{16} = (0.111000111011 \div 0.11010) \times 2^7$.
- By performing division of the mantissas using either of the two division algorithms described earlier, we obtain the result of division as $(10001100.00011)_2$.
- $(10001100.00011)_2 = (140.093)_{10}$.
- Also, $(E3B)_{16} = (3643)_{10}$ and $(1A)_{16} = (26)_{10}$.
- $(E3B)_{16} \div (1A)_{16} = (3643)_{10} \div (26)_{10} = (140.1)_{10}$, which is the same as the result obtained with binary floating-point arithmetic to a good approximation.

Review Questions

1. Outline the different steps involved in the addition of larger-bit binary numbers for the following two cases:

   (a) The larger of the two numbers is positive and the other number is negative.
   (b) The larger of the two numbers is negative and the other number is positive.

2. Outline the different steps involved in the subtraction of larger-bit binary numbers for the following two cases:

   (a) The minuend is positive. The subtrahend is negative and smaller in magnitude.
   (b) The minuend is positive. The subtrahend is negative and larger in magnitude.

3. What decides whether a particular binary addition or subtraction operation would be possible with 2's complement arithmetic?
4. Why in microprocessors and microcomputers is the 'repeated add and right-shift' algorithm preferred over the 'repeated left-shift and add' algorithm for binary multiplication? Briefly outline the procedure for multiplication in the case of the former.

5. Prove that the largest six-digit hexadecimal number when subtracted from the largest eight-digit octal number yields zero in decimal.

Problems

1. Perform the following operations using 2's complement arithmetic. The numbers are represented using 2's or 10's or 16's complement notation as the case may be. Express the result both in 2's complement binary as well as in decimal.

   (a) $(7F)_{16} + (A1)_{16}$.
   (b) $(110)_{10} + (0111)_2$.

   (a) $(00100000)_2, (32)_{10}$; (b) $(01110101)_2, (117)_{10}$

2. Evaluate the following to two binary places:

   (a) $(100.0001)_2 \div (10.1)_2$.
   (b) $(111001)_2 \div (1001)_2$.
   (c) $(111.001)_2 \times (1.11)_2$.

   (a) 1.10; (b) 110.01; (c) 1100.01

3. Prove that 16-bit 2's complement arithmetic cannot be used to add +18 150 and +14 618, while it can be used to add −18 150 and −14 618.
4. Add the maximum positive integer to the minimum negative integer, both represented in 16-bit 2's complement binary notation. Express the answer in 2's complement binary.

   1111111111111111

5. The result of adding two BCD numbers represented in excess-3 code is 0111 1011 when the two numbers are added using simple binary addition. If one of the numbers is $(12)_{10}$, find the other.

   $(03)_{10}$

6. Perform the following operations using 2's complement arithmetic:

   (a) $(+43)_{10} - (-53)_{10}$.
   (b) $(1ABC)_{16} + (1DEF)_{16}$.
   (c) $(3E91)_{16} - (1F93)_{16}$.

   (a) 01100000; (b) $(38AB)_{16}$; (c) $(1EFE)_{16}$

Further Reading

1. Ercegovac, M. D. and Lang, T. (2003) *Digital Arithmetic*, Morgan Kaufmann Publishers, CA, USA.
2. Tocci, R. J. (2006) *Digital Systems – Principles and Applications*, Prentice-Hall Inc., NJ, USA.
3. Ashmila, E. M., Dlay, S. S. and Hinton, O. R. (2005) 'Adder methodology and design using probabilistic multiple carry estimates'. *IET Computers and Digital Techniques*, **152**(6), pp. 697–703.
4. Lu, M. (2005) *Arithmetic and Logic in Computer Systems,* John Wiley & Sons, Inc., NJ, USA.

4

Logic Gates and Related Devices

Logic gates are electronic circuits that can be used to implement the most elementary logic expressions, also known as Boolean expressions. The logic gate is the most basic building block of combinational logic. There are three basic logic gates, namely the OR gate, the AND gate and the NOT gate. Other logic gates that are derived from these basic gates are the NAND gate, the NOR gate, the EXCLUSIVE-OR gate and the EXCLUSIVE-NOR gate. This chapter deals with logic gates and some related devices such as buffers, drivers, etc., as regards their basic functions. The treatment of the subject matter is mainly with the help of respective truth tables and Boolean expressions. The chapter is adequately illustrated with the help of solved examples. Towards the end, the chapter contains application-relevant information in terms of popular type numbers of logic gates from different logic families and their functional description to help application engineers in choosing the right device for their application. Pin connection diagrams are given on the companion website at http://www.wiley.com/go/maini_digital. Different logic families used to hardware-implement different logic functions in the form of digital integrated circuits are discussed in the following chapter.

4.1 Positive and Negative Logic

The binary variables, as we know, can have either of the two states, i.e. the logic '0' state or the logic '1' state. These logic states in digital systems such as computers, for instance, are represented by two different voltage levels or two different current levels. If the more positive of the two voltage or current levels represents a logic '1' and the less positive of the two levels represents a logic '0', then the logic system is referred to as a *positive logic system*. If the more positive of the two voltage or current levels represents a logic '0' and the less positive of the two levels represents a logic '1', then the logic system is referred to as a *negative logic system*. The following examples further illustrate this concept.

Digital Electronics: Principles, Devices and Applications Anil K. Maini
 ISBN: 978-0-470-03214-5

If the two voltage levels are 0 V and +5 V, then in the positive logic system the 0 V represents a logic '0' and the +5 V represents a logic '1'. In the negative logic system, 0 V represents a logic '1' and +5 V represents a logic '0'.

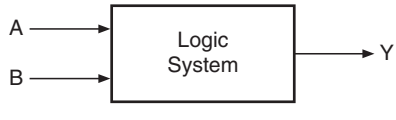
If the two voltage levels are 0 V and −5 V, then in the positive logic system the 0 V represents a logic '1' and the −5 V represents a logic '0'. In the negative logic system, 0 V represents a logic '0' and −5 V represents a logic '1'.

It is interesting to note, as we will discover in the latter part of the chapter, that a positive OR is a negative AND. That is, OR gate hardware in the positive logic system behaves like an AND gate in the negative logic system. The reverse is also true. Similarly, a positive NOR is a negative NAND, and vice versa.

4.2 Truth Table

A truth table lists all possible combinations of input binary variables and the corresponding outputs of a logic system. The logic system output can be found from the logic expression, often referred to as the Boolean expression, that relates the output with the inputs of that very logic system.

When the number of input binary variables is only one, then there are only two possible inputs, i.e. '0' and '1'. If the number of inputs is two, there can be four possible input combinations, i.e. 00, 01, 10 and 11. Figure 4.1(b) shows the truth table of the two-input logic system represented by Fig. 4.1(a). The logic system of Fig. 4.1(a) is such that $Y = 0$ only when both $A = 0$ and $B = 0$. For all other possible input combinations, output $Y = 1$. Similarly, for three input binary variables, the number of possible input combinations becomes eight, i.e. 000, 001, 010, 011, 100, 101, 110 and 111. This statement can be generalized to say that, if a logic circuit has n binary inputs, its truth table will have 2^n possible input combinations, or in other words 2^n rows. Figure 4.2 shows the truth table of a three-input logic circuit, and it has 8 $(= 2^3)$ rows. Incidentally, as we will see later in the chapter, this is the truth table of a three-input AND gate. It may be mentioned here that the truth table of a three-input AND gate as given in Fig. 4.2 is drawn following the positive logic system, and also that, in all further discussion throughout the book, we will use a positive logic system unless otherwise specified.



(a)

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

(b)

Figure 4.1 Two-input logic system.

A	B	C	Y
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

Figure 4.2 Truth table of a three-input logic system

4.3 Logic Gates

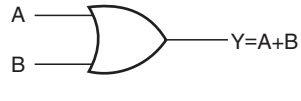
The logic gate is the most basic building block of any digital system, including computers. Each one of the basic logic gates is a piece of hardware or an electronic circuit that can be used to implement some basic logic expression. While laws of Boolean algebra could be used to do manipulation with binary variables and simplify logic expressions, these are actually implemented in a digital system with the help of electronic circuits called logic gates. The three basic logic gates are the OR gate, the AND gate and the NOT gate.

4.3.1 OR Gate

An OR gate performs an ORing operation on two or more than two logic variables. The OR operation on two independent logic variables A and B is written as $Y = A + B$ and reads as Y equals A OR B and not as A plus B. An OR gate is a logic circuit with two or more inputs and one output. The output of an OR gate is LOW only when all of its inputs are LOW. For all other possible input combinations, the output is HIGH. This statement when interpreted for a positive logic system means the following. The output of an OR gate is a logic '0' only when all of its inputs are at logic '0'. For all other possible input combinations, the output is a logic '1'. Figure 4.3 shows the circuit symbol and the truth table of a two-input OR gate. The operation of a two-input OR gate is explained by the logic expression

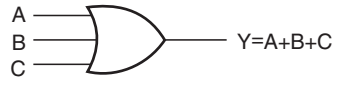
$$Y = A + B \tag{4.1}$$

As an illustration, if we have four logic variables and we want to know the logical output of $(A + B + C + D)$, then it would be the output of a four-input OR gate with A, B, C and D as its inputs.

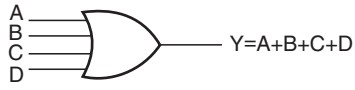


A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

Figure 4.3 Two-input OR gate.



(a)



(b)

A	B	C	Y
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

(c)

Figure 4.4 (a) Three-input OR gate, (b) four-input OR gate and (c) the truth table of a three-input OR gate.

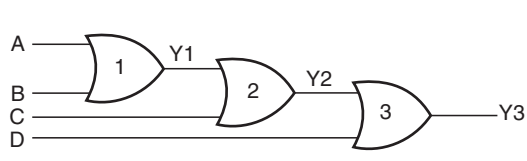
Figures 4.4(a) and (b) show the circuit symbol of three-input and four-input OR gates. Figure 4.4(c) shows the truth table of a three-input OR gate. Logic expressions explaining the functioning of three-input and four-input OR gates are $Y = A + B + C$ and $Y = A + B + C + D$.

Example 4.1

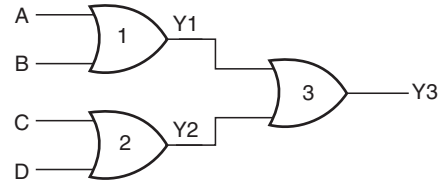
How would you hardware-implement a four-input OR gate using two-input OR gates only?

Solution

Figure 4.5(a) shows one possible arrangement of two-input OR gates that simulates a four-input OR gate. A, B, C and D are logic inputs and $Y3$ is the output. Figure 4.5(b) shows another possible arrangement. In the case of Fig. 4.5(a), the output of OR gate 1 is $Y1 = (A + B)$. The second



(a)



(b)

Figure 4.5 Example 4.1.

OR gate produces the output $Y2 = (Y1 + C) = (A + B + C)$. Similarly, the output of OR gate 3 is $Y3 = (Y2 + D) = (A + B + C + D)$. In the case of Fig. 4.5(b), the output of OR gate 1 is $Y1 = (A + B)$. The second OR gate produces the output $Y2 = (C + D)$. Output $Y3$ of the third OR gate is given by $(Y1 + Y2) = (A + B + C + D)$.

Example 4.2

Draw the output waveform for the OR gate and the given pulsed input waveforms of Fig. 4.6(a).

Solution

Figure 4.6(b) shows the output waveform. It can be drawn by following the truth table of the OR gate.

4.3.2 AND Gate

An AND gate is a logic circuit having two or more inputs and one output. The output of an AND gate is HIGH only when all of its inputs are in the HIGH state. In all other cases, the output is LOW. When interpreted for a positive logic system, this means that the output of the AND gate is a logic '1' only when all of its inputs are in logic '1' state. In all other cases, the output is logic '0'. The logic symbol and truth table of a two-input AND gate are shown in Figs 4.7(a) and (b) respectively. Figures 4.8(a) and (b) show the logic symbols of three-input and four-input AND gates respectively. Figure 4.8(c) gives the truth table of a four-input AND gate.

The AND operation on two independent logic variables A and B is written as $Y = A.B$ and reads as Y equals A AND B and not as A multiplied by B. Here, A and B are input logic variables and Y is the output. An AND gate performs an ANDing operation:

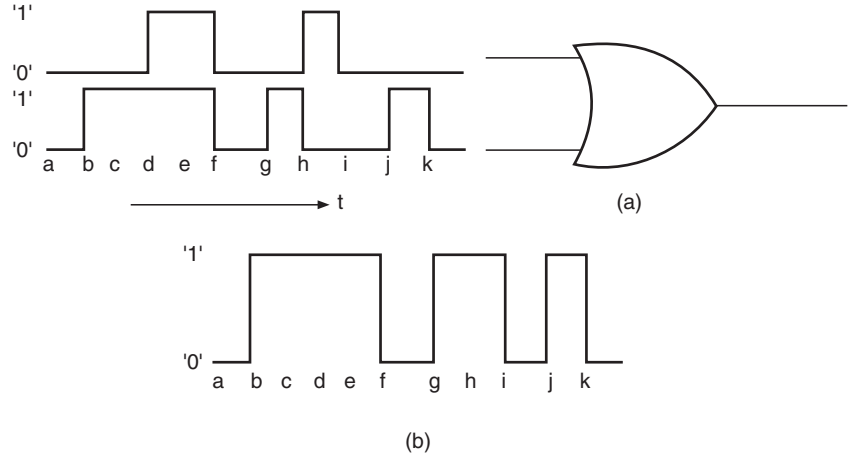
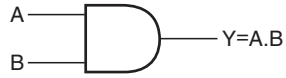


Figure 4.6 Example 4.2.

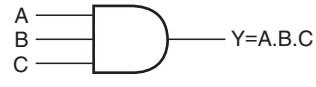


(a)

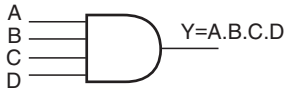
A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

(b)

Figure 4.7 Two-input AND gate.



(a)



(b)

A	B	C	D	Y
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

(c)

Figure 4.8 (a) Three-input AND gate, (b) four-input AND gate and (c) the truth table of a four-input AND gate.

- for a two-input AND gate, $Y = A.B$;
- for a three-input AND gate, $Y = A.B.C$;
- for a four-input AND gate, $Y = A.B.C.D$.

If we interpret the basic definition of OR and AND gates for a negative logic system, we have an interesting observation. We find that an OR gate in a positive logic system is an AND gate in a negative logic system. Also, a positive AND is a negative OR.

Example 4.3

Show the logic arrangement for implementing a four-input AND gate using two-input AND gates only.

Solution

Figure 4.9 shows the hardware implementation of a four-input AND gate using two-input AND gates. The output of AND gate 1 is $Y1 = A.B$. The second AND gate produces an output $Y2$ given by $Y2 = Y1.C = A.B.C$. Similarly, the output of AND gate 3 is $Y = Y2.D = A.B.C.D$ and hence the result.

4.3.3 NOT Gate

A NOT gate is a one-input, one-output logic circuit whose output is always the complement of the input. That is, a LOW input produces a HIGH output, and vice versa. When interpreted for a positive logic system, a logic '0' at the input produces a logic '1' at the output, and vice versa. It is also known as a 'complementing circuit' or an 'inverting circuit'. Figure 4.10 shows the circuit symbol and the truth table.

The NOT operation on a logic variable X is denoted as $\overline{X}$ or X'. That is, if X is the input to a NOT circuit, then its output Y is given by $Y = \overline{X}$ or X' and reads as Y equals NOT X. Thus, if $X = 0$, $Y = 1$ and if $X = 1$, $Y = 0$.

Example 4.4

For the logic circuit arrangements of Figs 4.11(a) and (b), draw the output waveform.

Solution

In the case of the OR gate arrangement of Fig. 4.11(a), the output will be permanently in logic '1' state as the two inputs can never be in logic '0' state together owing to the presence of the inverter. In the case of the AND gate arrangement of Fig. 4.11(b), the output will be permanently in logic '0' state as the two inputs can never be in logic '1' state together owing to the presence of the inverter.

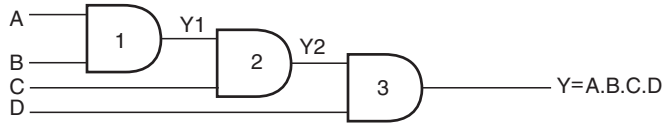


Figure 4.9 Implementation of a four-input AND gate using two-input AND gates.

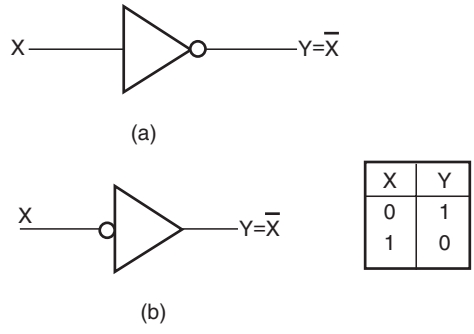


Figure 4.10 (a) Circuit symbol of a NOT circuit and (b) the truth table of a NOT circuit.

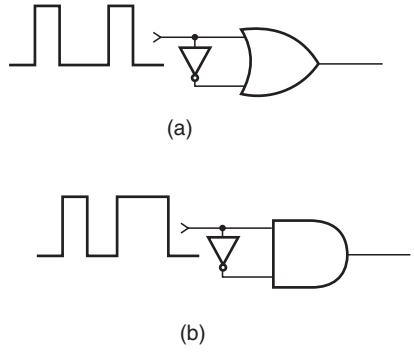


Figure 4.11 Example 4.4.

4.3.4 EXCLUSIVE-OR Gate

The EXCLUSIVE-OR gate, commonly written as EX-OR gate, is a two-input, one-output gate. Figures 4.12(a) and (b) respectively show the logic symbol and truth table of a two-input EX-OR gate. As can be seen from the truth table, the output of an EX-OR gate is a logic '1' when the inputs are unlike and a logic '0' when the inputs are like. Although EX-OR gates are available in integrated circuit form only as two-input gates, unlike other gates which are available in multiple inputs also, multiple-input EX-OR logic functions can be implemented using more than one two-input gates. The truth table of a multiple-input EX-OR function can be expressed as follows. The output of a multiple-input EX-OR logic function is a logic '1' when the number of 1s in the input sequence is odd and a logic '0' when the number of 1s in the input sequence is even, including zero. That is, an all 0s input sequence also produces a logic '0' at the output. Figure 4.12(c) shows the truth table of a four-input EX-OR function. The output of a two-input EX-OR gate is expressed by

$$Y = (A \oplus B) = \overline{A}B + A\overline{B} \tag{4.2}$$

(a)

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

(b)

A	B	C	D	Y
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	0
0	1	0	0	1
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	1
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	1
1	1	1	1	0

(c)

Figure 4.12 (a) Circuit symbol of a two-input EXCLUSIVE-OR gate, (b) the truth table of a two-input EXCLUSIVE-OR gate and (c) the truth table of a four-input EXCLUSIVE-OR gate

Example 4.5

How do you implement three-input and four-input EX-OR logic functions with the help of two-input EX-OR gates?

Solution

Figures 4.13(a) and (b) show the implementation of a three-input EX-OR logic function and a four-input EX-OR logic function using two-input logic gates:

- For Fig. 4.13(a), the output $Y1$ is given by $A \oplus B$. The final output Y is given by $Y = (Y1 \oplus C) = (A \oplus B) \oplus C = A \oplus B \oplus C$.
- Figure 4.13(b) can be explained on similar lines.

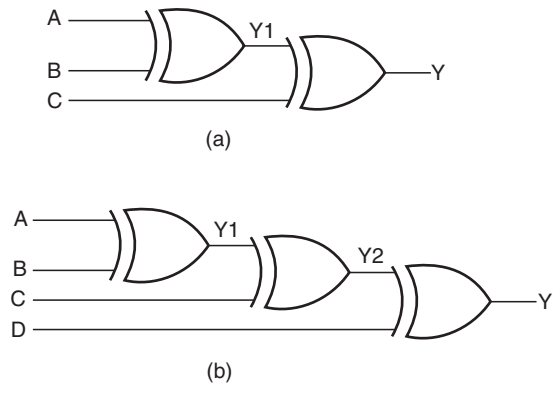


Figure 4.13 (a) Three-input EX-OR gate and (b) a four-input EX-OR gate.

Example 4.6

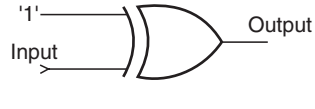
How can you implement a NOT circuit using a two-input EX-OR gate?

Solution

Refer to the truth table of a two-input EX-OR gate reproduced in Fig. 4.14(a). It is clear from the truth table that, if one of the inputs of the gate is permanently tied to logic '1' level, then the other input and output perform the function of a NOT circuit. Figure 4.14(b) shows the implementation.

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

(a)



(b)

Figure 4.14 Implementation of a NOT circuit using an EX-OR gate.