www.AllAbtEngg.com

UNIT I

MINIMIZATION TECHNIQUES AND LOGIC GATES

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 hexa decimalnumber 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 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. There all world, however, is analogue. Most physical quantities – position, velocity, acceleration, force, pressure, temperature and flow rate, 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

AllAbtEngg Android Application for Anna University, Polytechnic & School

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 byr^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, are100, 101, 102 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

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

AllAbtEngg Android Application for Anna University, Polytechnic & School

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) and2^{-1}, 2^{-2}, 2^{-3} and so on (for the fractional part).

Example

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

AllAbtEngg Android Application for Anna University, Polytechnic & School

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 digit sare 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 hexa decimal 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 15respectively, 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 536different 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

AllAbtEngg Android Application for Anna University, Polytechnic & School

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'scomplement* 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'scomplement* 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 $+(2^{n-1} -$

AllAbtEngg Android Application for Anna University, Polytechnic & School

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'scomplement 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 = $0 \times 2^{-1} + 1 \times 2^{-2} + 0 \times 2^{-3} + 1 \times 2{-4} = 0 + 0.25 + 0$

$+ 0.0625 = 0.3125$

AllAbtEngg Android Application for Anna University, Polytechnic & School

• 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

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

$= 0 + 2 + 4 + 8 + 0 + 0 + 0 = 14$

Therefore, 00001110 represents +14

AllAbtEngg Android Application for Anna University, Polytechnic & School

(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 = $0 \times 2^0 + 1 \times 2^1 + 0 \times 2^2 + 0 \times 2^3 + 1 \times 2^4 + 1 \times 2^5 + 1 \times 2^6$

$= 0 + 2 + 0 + 0 + 16 + 32 + 64 = 114$

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 2and 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

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

AllAbtEngg Android Application for Anna University, Polytechnic & School

• 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

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 × 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

AllAbtEngg Android Application for Anna University, Polytechnic & School

www.AllAbtEngg.com

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 × 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

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 = $(1110100.0100111)_2$

• $(1110100.0100111)_2$ = $(1\ 110\ 100.010\ 011\ 1)_2$

= $(001\ 110\ 100.010\ 011\ 100)_2$ = $(164.234)_8$

AllAbtEngg Android Application for Anna University, Polytechnic & School

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

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.

AllAbtEngg Android Application for Anna University, Polytechnic & School

Example

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

AllAbtEngg Android Application for Anna University, Polytechnic & School

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

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

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

Solution

• The 7's complement = 5264.

• Therefore, the octal number = $(2513)_8$.

• The binary equivalent = $(010\ 101\ 001\ 011)_2 = (10101001011)_2$.

AllAbtEngg Android Application for Anna University, Polytechnic & School

• 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

$$N = m \times b^e \quad (1.1)$$

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 (±d. *dddd…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

$$N = m \times 10^e (1.2)$$

Hexadecimal system

$$N = m \times 16^e (1.3)$$

Binary system

$$N = m \times 2^e (1.4)$$

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$2.57ABF \times 16^2$. 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

AllAbtEngg Android Application for Anna University, Polytechnic & School

written as −.101 × 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 .11010110and .11100000.

1.18Binary 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 use done 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.

1.19 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 straight forward. 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

Table 1.1 BCD codes.

Decimal	8421 BCD code	4221 BCD code	5421 BCD code
0	0000	0000	0000
1	0001	0001	0001
2	0010	0010	0010

AllAbtEngg Android Application for Anna University, Polytechnic & School

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 and1 in the 4221 BCD code and 5, 4, 2 and 1 in the 5421 BCD code represent weights of the relevant bits. Table 1.1 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.

1.19.1 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 these cond 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, $(0010\ 1001.0111\ 0101)_{BCD}= (11101.11)_2$.

1.19.2 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.

AllAbtEngg Android Application for Anna University, Polytechnic & School

• The BCD equivalent can then be written as 0001 0111 0001.0110 0010 0101.

1.19.3 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 10bits 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.

1.19.4 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.

1.20 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 1.2 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

AllAbtEngg Android Application for Anna University, Polytechnic & School

Table 1.2Excess-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 be0011001100111000 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 subtracting0011 from each four-bit group. The new number is the 8421 BCD equivalent of the givenexcess-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'scomplement 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

AllAbtEngg Android Application for Anna University, Polytechnic & School

Find (a) the excess-3 equivalent of $(237.75)_{10}$and (b) the decimal equivalent of the excess-3 number110010100011.01110101.

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, theexcess-3code 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 0000.01000010.

Therefore, the decimal equivalent = $(970.42)_{10}$.

1.21 Gray Code

The Gray code was designed by Frank Gray at Bell Labs and patented in 1953. It is an weighted 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 1.3 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 1.3, 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 soon. 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 1.4. 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.

AllAbtEngg Android Application for Anna University, Polytechnic & School

Table 1.3Gray 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 1.4Generation 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

AllAbtEngg Android Application for Anna University, Polytechnic & School

111	1111
110	1110
010	1010
011	1011
001	1001
000	1000

1.21.1 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 code1- - -

Binary1011

Gray code11- -

Binary1011

Gray code111-

AllAbtEngg Android Application for Anna University, Polytechnic & School

Binary1011

Gray code1110

1.21.2 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 code1110

Binary1- - -

Gray code1110

Binary10 - -

Gray code1110

Binary101-

Gray code1110

Binary1011

1.22 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

AllAbtEngg Android Application for Anna University, Polytechnic & School

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 interms 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.

1.23Digital 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.

1.24 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 1.5 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

AllAbtEngg Android Application for Anna University, Polytechnic & School

Table 1.5Binary 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 1.6 summarizes the binary subtraction operation. The entries in Table 1.6 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 1.6Binary subtraction.

Inputs			Outputs	
Minuend	Subtrahend	Borrow-in	Difference	Borrow-out
0	0	0	0	0
0	0	1	1	1

AllAbtEngg Android Application for Anna University, Polytechnic & School

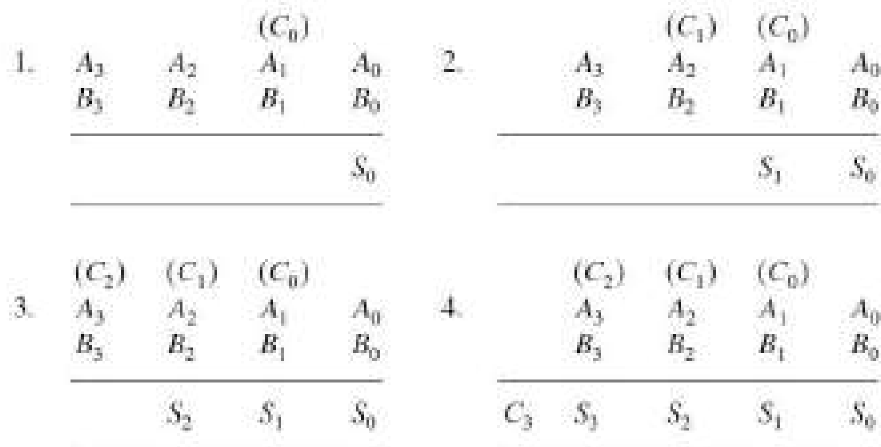
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

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 3.3respectively.

1.25 Addition of Larger-Bit Binary Numbers

The addition of larger binary integers, fractions or mixed binary numbers is performed column wise 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_0representing 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:



AllAbtEngg Android Application for Anna University, Polytechnic & School

1.25.1 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:

AllAbtEngg Android Application for Anna University, Polytechnic & School

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

• 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.

AllAbtEngg Android Application for Anna University, Polytechnic & School

www.AllAbtEngg.com

• 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

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:

```
 100010011.110
 000100101.111
 -------------
 100111001.101
```

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

2. $(AF1.B3)_{16} = (101011110001.10110011)_2$ and $(FFF.E)_{16} = (111111111111.1110)_2$. $(111111111111.1110)_2$ can also be written as $(111111111111.11100000)_2$ to have the same

AllAbtEngg Android Application for Anna University, Polytechnic & School

number of bits in the integer and fractional parts. The two numbers can now be added as follows:

```
 0101011110001.10110011
 0111111111111.11100000
 1101011110001.10010011
```

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

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

Add − 118 and − 32 firstly using eight-bit 2's complement arithmetic and then using 16-bit 2'scomplement 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'scomplement 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.

AllAbtEngg Android Application for Anna University, Polytechnic & School

1.26 Subtraction of Larger-Bit Binary Numbers

Subtraction is also done column wise 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
    -------            -------
```

1.26.1 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.

AllAbtEngg Android Application for Anna University, Polytechnic & School

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

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

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.

AllAbtEngg Android Application for Anna University, Polytechnic & School

• 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.

AllAbtEngg Android Application for Anna University, Polytechnic & School

• 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. In2’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:

AllAbtEngg Android Application for Anna University, Polytechnic & School

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 lines in the range from -2^{n-1} to $+(2^{n-1} - 1)$, n being the number of bits used to represent the numbers.

Example

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

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:

AllAbtEngg Android Application for Anna University, Polytechnic & School

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

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.

1.27 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. $1 \times 1 = 1$.

AllAbtEngg Android Application for Anna University, Polytechnic & School

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.

1.27.1 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.

```
                      1 0 1 1 1
Multiplicand :        × 1 1 0  .............. (23)10
Multiplier :   ______________  .............. (6)10
                      0 0 0 0 0
                    1 0 1 1 1
                  1 0 1 1 1
               ______________
               1 0 0 0 1 0 1 0
```

The decimal equivalent of (10001010) is (138) , which is the correct result.

1.27.2 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

AllAbtEngg Android Application for Anna University, Polytechnic & School

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 1.7.

• Therefore, $(10111)_2 \times (110)_2 = (10001010)_2$.

Table 1.7Multiplication 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

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:

AllAbtEngg Android Application for Anna University, Polytechnic & School

```
      1 0 0 0 1
        × 1 0 1
  -------------
      1 0 0 0 1
    0 0 0 0 0
  1 0 0 0 1
  -------------
  1 0 1 0 1 0 1
  -------------
```

The multiplication result is then given by placing the binary point three bits after the LSB, which gives (1010.101) as the final result. Also, $(100.01)_2$ = $(4.25)_{10}$and $(10.1)_2$ = $(2.5)_{10}$. Moreover, $(4.25)_{10}$ × $(2.5)_{10}$= $(10.625)_{10}$ 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, $(2B)_{16}$ × $(3)_{16}$= $(129)_{10}$. Moreover, $(10000001)_2$ = $(129)_{10}$, which verifies the result.

1.28 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 1.8Example.

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

AllAbtEngg Android Application for Anna University, Polytechnic & School

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

1.28.1 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 1.9. The quotient = 011 and the remainder = 10.

Table 1.9Binary 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 1.10Binary division using the repeate subtract and left-shift algorithm.

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

AllAbtEngg Android Application for Anna University, Polytechnic & School

www.AllAbtEngg.com

1.28.2 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 1.10. The quotient = 011 and the remainder = 10.

Example

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 1.11Example

AllAbtEngg Android Application for Anna University, Polytechnic & School

	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

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 Table1.12. 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

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.

AllAbtEngg Android Application for Anna University, Polytechnic & School

www.AllAbtEngg.com

Solution

• The dividend = $(AF)_{16}$.

• As it is a negative hexadecimal number, the magnitude of this number is determined by its 2'scomplement (or more precisely by its 16's complement in hexadecimal number language).

Table 1.12Example.

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.

AllAbtEngg Android Application for Anna University, Polytechnic & School

• 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 $(F7)_{16}$.

1.29 Boolean Algebra and Simplification Techniques

Boolean algebra is mathematics of logic. It is one of the most basic tools available to the logic designer and thus can be effectively used for simplification of complex logic expressions. Other useful and widely used techniques based on Boolean theorems include the use of Karnaugh maps in what is known as the mapping method of logic simplification and the tabular method given by Quine–McCluskey. In this chapter, we will have a closer look at the different postulates and theorems of Boolean algebra and their applications in minimizing Boolean expressions. We will also discuss at length the mapping and tabular methods of minimizing fairly complex and large logic expressions.

1.30 Introduction to Boolean Algebra

Boolean algebra, quite interestingly, is simpler than ordinary algebra. It is also composed of a set of symbols and a set of rules to manipulate these symbols. However, this is the only similarity between the two. The differences are many. These include the following:

1. In ordinary algebra, the letter symbols can take on any number of values including infinity. In Boolean algebra, they can take on either of two values, that is, 0 and 1.

2. The values assigned to a variable have a numerical significance in ordinary algebra, whereas in its Boolean counterpart they have a logical significance.

3. While '.' and '+' are respectively the signs of multiplication and addition in ordinary algebra, in Boolean algebra '.' means an AND operation and '+' means an OR operation. For instance, A + Bin ordinary algebra is read as A plus B, while the same in Boolean algebra is read as A OR B. Basic logic operations such as AND, OR and NOT have already been discussed at length in Chapter 4.

4. More specifically, Boolean algebra captures the essential properties of both logic operations such as AND, OR and NOT and set operations such as intersection, union and complement. As an illustration, the logical assertion that both a statement and its negation cannot be true has a counterpart in set theory, which says that the intersection of a subset and its complement is a null(or empty) set.

AllAbtEngg Android Application for Anna University, Polytechnic & School

5. Boolean algebra may also be defined to be a set A supplied with two binary operations of logical AND (Λ), logical OR (V), a unary operation of logical NOT (¬) and two elements, namely logical FALSE (0) and logical TRUE (1). This set is such that, for all elements of this set, the postulates or axioms relating to the associative, commutative, distributive, absorption and complementation properties of these elements hold good. These postulates are described in the following pages.

1.30.1 Variables, Literals and Terms in Boolean Expressions

Variables are the different symbols in a Boolean expression. They may take on the value '0' or '1'. For instance, in expression (1.1), A, B and C are the three variables. In expression (1.2), P, Q, R and S are the variables:

$\overline{A} + A.B + A.\overline{C} + \overline{A}.B.C$(1.1)

$(\overline{P} + Q)(R + \overline{S})\ (P + \overline{Q} + R)$(1.2)

The complement of a variable is not considered as a separate variable. Each occurrence of a variable or its complement is called a *literal*. In expressions (1.1) and (1.2) there are eight and seven literals respectively. A term is the expression formed by literals and operations at one level. Expression (1.1)has five terms including four AND terms and the OR term that combines the first-level AND terms.

1.30.2 Equivalent and Complement of Boolean Expressions

Two given Boolean expressions are said to be *equivalent* if one of them equals '1' only when the other equals '1' and also one equals '0' only when the other equals '0'. They are said to be the *complement* of each other if one expression equals '1' only when the other equals '0', and vice versa. The complement of a given Boolean expression is obtained by complementing each literal, changing all '.' to '+' and all '+' to '.', all 0s to 1s and all 1s to 0s. The examples below give some Boolean expressions and their complements:

Given Boolean expression

$\overline{A}.B + A.\overline{B}$(1.3)

Corresponding complement

$(A + \overline{B}).(\overline{A} + B)$(1.4)

Given Boolean expression

$(A + B).(\overline{A} + \overline{B})$(1.5)

Corresponding complement

$(\overline{A}.\overline{B}) + (A.B)$(1.6)

AllAbtEngg Android Application for Anna University, Polytechnic & School

When OR ed with its complement the Boolean expression yields a ‘1’, and when ANDed with its complement it yields a ‘0’. The ‘.’ sign is usually omitted in writing Boolean expressions and is implied merely by writing the literals in juxtaposition. For instance, A.B would normally be written as AB.

1.30.3 Dual of a Boolean Expression

The dual of a Boolean expression is obtained by replacing all ‘.’ operations with ‘+’ operations, all‘+’ operations with ‘.’ operations, all 0s with 1s and all 1s with 0s and leaving all literals unchanged.

The examples below give some Boolean expressions and the corresponding dual expressions:

Given Boolean expression

$\overline{A}.B + A.\overline{B}$(1.7)

Corresponding dual

$(\overline{A} + B).(A + \overline{B})$(1.8)

Given Boolean expression

$(A + B).(\overline{A} + \overline{B})$(1.9)

Corresponding dual

$A.B + \overline{A}.\overline{B}$(1.10)

Duals of Boolean expressions are mainly of interest in the study of Boolean postulates and theorems. Otherwise, there is no general relationship between the values of dual expressions. That is, both of them may equal ‘1’ or ‘0’. One may even equal ‘1’ while the other equals ‘0’. The fact that the dual of a given logic equation is also a valid logic equation leads to many more useful laws of Boolean algebra. The principle of duality has been put to ample use during the discussion on postulates and theorems of Boolean algebra. The postulates and theorems, to be discussed in the paragraphs to follow, have been presented in pairs, with one being the dual of the other.

Example

Find (a) the dual of $A.\overline{B} + B.\overline{C} + C.\overline{D}$*and (b) the complement of*$[(A.\overline{B} + \overline{C}).D + \overline{E}].F.$

Solution

(a) The dual of $A.\overline{B} + B.\overline{C} + C.\overline{D}$is given by $(A + \overline{B}).(B + \overline{C}).(C + \overline{D})$

(b) The complement of$[(A.\overline{B} + \overline{C}).D + \overline{E}].F$ is given by $[(\overline{A} + B).C + \overline{D}].E + \overline{F}.$

AllAbtEngg Android Application for Anna University, Polytechnic & School

Example

Simplify $(A.B + C.D)[(\overline{A} + \overline{B}).(\overline{C} + \overline{D})]$.

Solution

• Let $(A.B + C.D) = X$.

• Then the given expression reduces to $X.\overline{X}$.

• Therefore, $(A.B + C.D)[(\overline{A} + \overline{B}).(\overline{C} + \overline{D})] = 0$.

1.31 Postulates of Boolean Algebra

The following are the important postulates of Boolean algebra:

1. $1.1 = 1.0 + 0 = 0$.

2. $1.0 = 0.1 = 0.0 + 1 = 1 + 0 = 1$.

3. $0.0 = 0.1 + 1 = 1$.

4. $\overline{1} = 0$ and $\overline{0} = 1$.

Many theorems of Boolean algebra are based on these postulates, which can be used to simplify Boolean expressions. These theorems are discussed in the next section.

1.32 Theorems of Boolean Algebra

The theorems of Boolean algebra can be used to simplify many a complex Boolean expression and also to transform the given expression into a more useful and meaningful equivalent expression. The theorems are presented as pairs, with the two theorems in a given pair being the dual of each other. These theorems can be very easily verified by the method of 'perfect induction'. According to this method, the validity of the expression is tested for all possible combinations of values of the variables involved. Also, since the validity of the theorem is based on its being true for all possible combinations of values of variables, there is no reason why a variable cannot be replaced with its complement, or vice versa, without disturbing the validity. Another important point is that, if a given expression is valid, its dual will also be valid. Therefore, in all the discussion to follow in this section, only one of the theorems in a given pair will be illustrated with a proof. Proof of the other being its dual is implied.

AllAbtEngg Android Application for Anna University, Polytechnic & School

1.32.1 Theorem 1 (Operations with '0' and '1')

(a) 0.X = 0 and (b) 1 + X = 1 (1.11)

where X is not necessarily a single variable – it could be a term or even a large expression.

Theorem 1(a) can be proved by substituting all possible values of X, that is, 0 and 1, into the given expression and checking whether the LHS equals the RHS:

• For X = 0, LHS = 0.X = 0.0 = 0 = RHS.

• For X = 1, LHS = 0.1 = 0 = RHS.

Thus, 0.X = 0 irrespective of the value of X, and hence the proof.

Theorem 1(b) can be proved in a similar manner. In general, according to theorem 1, 0.(Boolean expression) = 0 and 1 + (Boolean expression) = 1. For example, $0.(A.B + B.C + C.D) = 0$ and $1 + (A.B + B.C + C.D) = 1$, where A, B and C are Boolean variables.

1.32.2 Theorem 2 (Operations with '0' and '1')

(a) 1.X = X and (b) 0 + X = X (1.12)

where X could be a variable, a term or even a large expression. According to this theorem, ANDing a Boolean expression to '1' or ORing '0' to it makes no difference to the expression:

• For X = 0, LHS = 1.0 = 0 = RHS.

• For X = 1, LHS = 1.1 = 1 = RHS.

Also, 1.(Boolean expression) = Boolean expression and 0 + (Boolean expression) = Boolean expression. For example,

$$1.(A + B.C + C.D) = 0 + (A + B.C + C.D) = A + B.C + C.D$$

1.32.3 Theorem 3 (Idempotent or Identity Laws)

(a) X.X.X.....X = Xand (b) X + X + X + · · · + X = X (1.13)

Theorems 3(a) and (b) are known by the name of *idempotent laws*, also known as *identity laws*. Theorem 3(a) is a direct outcome of an AND gate operation, whereas theorem 3(b) represents an ORgate operation when all the inputs of the gate have been tied together. The scope of idempotent laws can be expanded further by considering X to be a term or an expression. For example, let us apply idempotent laws to simplify the following Boolean expression:

$$(A.\overline{B}.\overline{B} + C.C)(A.\overline{B}.\overline{B} + A.\overline{B} + C.C) = (A.\overline{B} + C)(A.\overline{B} + A.\overline{B} + C)$$

AllAbtEngg Android Application for Anna University, Polytechnic & School

$$= (A.\overline{B} + C)\,(A.\overline{B} + C) = (A.\overline{B} + C)$$

1.32.4 Theorem 4 (Complementation Law)

(a) $X.\overline{X} = 0$ and (b) $X + \overline{X} = 1$(1.14)

According to this theorem, in general, any Boolean expression when ANDed to its complement yield as '0' and when ORed to its complement yields a '1', irrespective of the complexity of the expression:

- For $X = 0, \overline{X} = 1$. Therefore, $X.\overline{X} = 0.1 = 0$.
- For $X = 1, \overline{X} = 0$. Therefore, $X.\overline{X} = 1.0 = 0$.

Hence, theorem 4(a) is proved. Since theorem 4(b) is the dual of theorem 4(a), its proof is implied.

The example below further illustrates the application of complementation laws:

$$(A + B.C)\,(\overline{A + B.C}) = 0 \text{and} (A + B.C) + (\overline{A + B.C}) = 1$$

Example

Simplify the following:

$$[1 + L.M + L.\overline{M} + \overline{L}.M].[(L + \overline{M})(\overline{L}.M) + \overline{L}.\overline{M}.(L + M)].$$

Solution

- We know that (1 + Boolean expression) = 1.
- Also, $(\overline{L}.M)$ is the complement of $(L + \overline{M})$ and $(\overline{L}.\overline{M})$ is the complement of $(L + M)$.
- Therefore, the given expression reduces to $1.(0 + 0) = 1.0 = 0$.

1.32.5 Theorem 5 (Commutative Laws)

(a) $X + Y = Y + X$ and (b) $X.Y = Y.X$(1.15)

Theorem 5(a) implies that the order in which variables are added or ORed is immaterial. That is, the result of A OR B is the same as that of B OR A. Theorem 5(b) implies that the order in which variables are ANDed is also immaterial. The result of A AND B is same as that of B AND A.

1.32.6 Theorem 6 (Associative Laws)

(a) X + (Y + Z) = Y + (Z + X) = Z + (X + Y)

AllAbtEngg Android Application for Anna University, Polytechnic & School

and

(b) X. (Y.Z) = Y. (Z.X) = Z. (X.Y) (1.16)

Theorem 6(a) says that, when three variables are being ORed, it is immaterial whether we do this by ORing the result of the first and second variables with the third variable or by ORing the first variable with the result of ORing of the second and third variables or even by ORing the second variable with the result of ORing of the first and third variables. According to theorem 6(b), when three variables are being ANDed, it is immaterial whether you do this by ANDing the result of ANDing of the first and second variables with the third variable or by ANDing the result of ANDing of the second and third variables with the first variable or even by ANDing the result of ANDing of the third and first variables with the second variable.

For example,

$$\overline{A}.B + (C.\overline{D} + \overline{E}.\overline{F}) = C.\overline{D} + (\overline{A}.B + \overline{E}.\overline{F}) = \overline{E}.\overline{F} + (\overline{A}.B + C.\overline{D})$$

Also

$$\overline{A}.B.(C.\overline{D}.\overline{E}.\overline{F}) = C.\overline{D}.(\overline{A}.B.\overline{E}.\overline{F}) = \overline{E}.\overline{F}.(\overline{A}.B.C.\overline{D})$$

Theorems 6(a) and (b) are further illustrated by the logic diagrams in Figs (a) and (b).

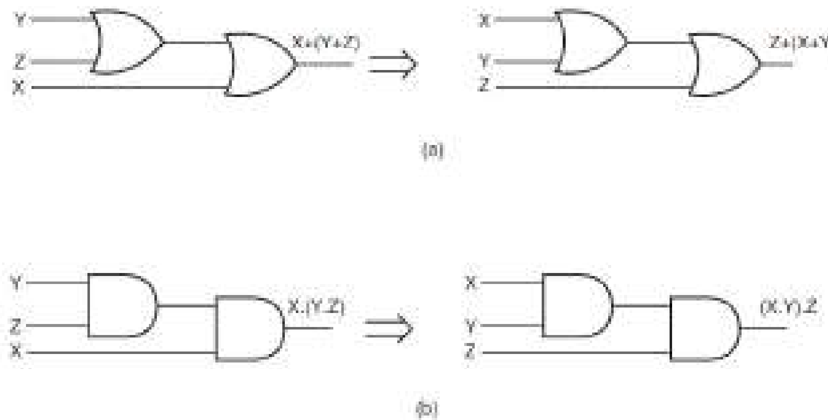


Figure Associative laws.

1.32.7 Theorem 7 (Distributive Laws)

$$(a)\, X.(Y + Z) = X.Y + X.Z \text{ and } (b)\, X + Y.Z = (X + Y).(X + Z) \quad (1.17)$$

Theorem 7(b) is the dual of theorem 7(a). The distribution law implies that a Boolean expression can always be expanded term by term. Also, in the case of the expression being the sum of two or more than two terms having a common variable, the common variable can be

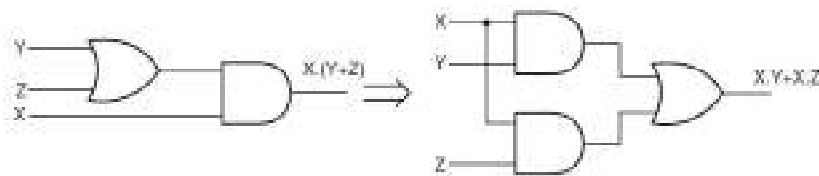
AllAbtEngg Android Application for Anna University, Polytechnic & School

taken as common as in the case of ordinary algebra. Table gives the proof of theorem 7(a) using the method of perfect induction. Theorem 7(b) is the dual of theorem 7(a) and therefore its proof is implied. Theorems 7(a) and (b) are further illustrated by the logic diagrams in Figs 6.2(a) and (b). As an illustration, theorem 7(a) can be used to simplify $\overline{A}.\overline{B} + \overline{A}.B + A.\overline{B} + A.B$ as follows:

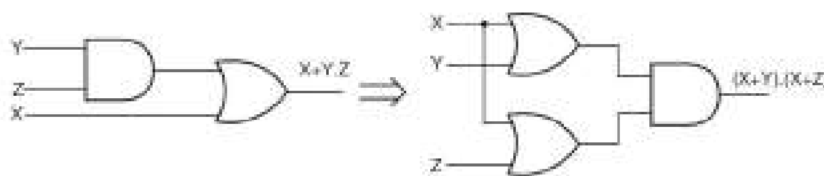
$$\overline{A}.\overline{B} + \overline{A}.B + A.\overline{B} + A.B = \overline{A}.(\overline{B} + B) + A.(\overline{B} + B) = \overline{A}.1 + A.1 = \overline{A} + A = 1$$

Table Proof of distributive law.

X	Y	Z	Y+Z	XY	XZ	X(Y+Z)	XY+XZ
0	0	0	0	0	0	0	0
0	0	1	1	0	0	0	0
0	1	0	1	0	0	0	0
0	1	1	1	0	0	0	0
1	0	0	0	0	0	0	0
1	0	1	1	0	1	1	1
1	1	0	1	1	0	1	1
1	1	1	1	1	1	1	1



(a)



(b)

Figure Distributive laws.

Theorem 7(b) can be used to simplify$(\overline{A} + \overline{B}).(\overline{A} + B).(A + \overline{B}).(A + B)$ as follows:

$(\overline{A} + \overline{B}).(\overline{A} + B).(A + \overline{B}).(A + B) = (\overline{A} + \overline{B}.B).(A + \overline{B}.B) = (\overline{A} + 0).(A + 0) = \overline{A}.A = 0$

AllAbtEngg Android Application for Anna University, Polytechnic & School

1.32.8 Theorem 8

$$(a)\ X.Y + X.\overline{Y} = X \quad \text{and} \quad (b)\ (X+Y).(X+\overline{Y}) = X$$

This is a special case of theorem 7 as

$$X.Y + X.\overline{Y} = X.(Y+\overline{Y}) = X.1 = X \quad \text{and} \quad (X+Y).(X+\overline{Y}) = X + Y.\overline{Y} = X + 0 = X$$

This theorem, however, has another very interesting interpretation. Referring to theorem 8(a), there are two two-variable terms in the LHS expression. One of the variables, Y , is present in all possible combinations in this expression, while the other variable, X, is a common factor. The expression then reduces to this common factor. This interpretation can be usefully employed to simplify many a complex Boolean expression.

As an illustration, let us consider the following Boolean expression:

$$A.\overline{B}.\overline{C}.\overline{D} + A.\overline{B}.\overline{C}.D + A.\overline{B}.C.\overline{D} + A.\overline{B}.C.D + A.B.\overline{C}.\overline{D} + A.B.\overline{C}.D + A.B.C.\overline{D} + A.B.C.D$$

In the above expression, variables B, C and D are present in all eight possible combinations, and variable A is the common factor in all eight product terms. With the application of theorem 8(a), this expression reduces to A. Similarly, with the application of theorem 8(b), $(A + \overline{B} + \overline{C}).(A + \overline{B} + C).(A + B + \overline{C}).(A + B + C)$ also reduces to A as the variables B and C are present in all four possiblecombinations in sum terms and variable A is the common factor in all the terms.

1.32.9 Theorem 9

$$(a)(X + \overline{Y}).Y = X.Y \ and \ (b)X.\overline{Y} + Y = X + Y \qquad (1.18)$$

$$(X + \overline{Y}).Y = X.Y + \overline{Y}.Y = X.Y$$

Theorem 9(b) is the dual of theorem 9(a) and hence stands proved.

1.32.10 Theorem 10 (Absorption Law or Redundancy Law)

$$(a)X + X.Y = X \ and \ (b)X.(X + Y) = X \qquad (1.19)$$

The proof of absorption law is straightforward:

$$X + X.Y = X.(1 + Y) = X.1 = X$$

Theorem 10(b) is the dual of theorem 10(a) and hence stands proved.

The crux of this simplification theorem is that, if a smaller term appears in a larger term, then the larger term is redundant. The following examples further illustrate the underlying concept:

AllAbtEngg Android Application for Anna University, Polytechnic & School

$$A + A.\overline{B} + A.\overline{B}.\overline{C} + A.\overline{B}.C + \overline{C}.B.A = A$$

and

$$(\overline{A} + B + \overline{C}).(\overline{A} + B).(C + B + \overline{A}) = \overline{A} + B$$

1.32.11 Theorem 11

$$(a)Z.X + Z.\overline{X}.Y = Z.X + Z.Y$$

and

$$(b)\ (Z + X)(Z + X + Y) = (Z + X)(Z + Y)$$

(1.20)

Table gives the proof of theorem 11(a) using the method of perfect induction. Theorem 11(b) is the dual of theorem 11(a) and hence stands proved. A useful interpretation of this theorem is that, when

Table Proof of theorem 11(a).

X	Y	Z	ZX	ZY	$Z\overline{X}$	$Z\overline{X}Y$	$ZX+Z\overline{X}Y$	ZX+ZY
0	0	0	0	0	0	0	0	0
0	0	1	0	0	1	0	0	0
0	1	0	0	0	0	0	0	0
0	1	1	0	1	1	1	1	1
1	0	0	0	0	0	0	0	0
1	0	1	1	0	0	0	1	1
1	1	0	0	0	0	0	0	0
1	1	1	1	1	0	0	1	1

a smaller term appears in a larger term except for one of the variables appearing as a complement in the larger term, the complemented variable is redundant.

As an example, $(A + \overline{B}).(\overline{A} + \overline{B} + C).(\overline{A} + \overline{B} + D)$ can be simplified as follows:

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

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

1.32.12 Theorem 12 (Consensus Theorem)

$$(a)\ X.Y + \overline{X}.Z + Y.Z = X.Y + \overline{X}.Z$$

AllAbtEngg Android Application for Anna University, Polytechnic & School

and

$$(b)(X+Y).(\overline{X}+Z).(Y+Z) = (X+Y).(\overline{X}+Z) \quad (1.21)$$

Table shows the proof of theorem 12(a) using the method of perfect induction. Theorem 12(b) is the dual of theorem 12(a) and hence stands proved.

A useful interpretation of theorem 12 is as follows. If in a given Boolean expression we can identify two terms with one having a variable and the other having its complement, then the term that is formed by the product of the remaining variables in the two terms in the case of a sum-of-products expression

Table Proof of theorem 12(a).

X	Y	Z	XY	$\overline{X}Z$	YZ	$XY+\overline{X}Z+YZ$	$XY+\overline{X}Z$
0	0	0	0	0	0	0	0
0	0	1	0	1	0	1	1
0	1	0	0	0	0	0	0
0	1	1	0	1	1	1	1
1	0	0	0	0	0	0	0
1	0	1	0	0	0	0	0
1	1	0	1	0	0	1	1
1	1	1	1	0	1	1	1

or by the sum of the remaining variables in the case of a product-of-sums expression will be redundant. The following example further illustrates the point:

$$A.B.C+\overline{A}.C.D+\overline{B}.C.D+B.C.D+A.C.D = A.B.C+\overline{A}.C.D+\overline{B}.C.D$$

If we consider the first two terms of the Boolean expression, B C D becomes redundant. If we consider the first and third terms of the given Boolean expression, A C D becomes redundant.

Example

AllAbtEngg Android Application for Anna University, Polytechnic & School

Prove that $A.B.C.D+A.B.\overline{C}.\overline{D}+A.B.C.\overline{D}+A.B.\overline{C}.D+A.B.C.D.E+A.B.\overline{C}.\overline{D}.\overline{E}+A.B.\overline{C}.D.E$ can be simplified to $A.B$.

Solution

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

- $A.B.C.D$ appears in $A.B.C.D.E$, $A.B.\overline{C}.\overline{D}$ appears in $A.B.\overline{C}.\overline{D}.\overline{E}$ and $A.B.\overline{C}.D$ appears in $A.B.\overline{C}.D.E$.
- As a result, all three five-variable terms are redundant.
- Also, variables C and D appear in all possible combinations and are therefore redundant.

1.32.13 Theorem 13 (DeMorgan's Theorem)

$$\text{(a) } [\overline{X_1+X_2+X_3+\ldots+X_n}]=\overline{X_1}.\overline{X_2}.\overline{X_3}.\ldots.\overline{X_n}$$
$$\text{(b) } [\overline{X_1.X_2.X_3.\ldots.X_n}]=[\overline{X_1}+\overline{X_2}+\overline{X_3}+\ldots+\overline{X_n}]$$

According to the first theorem the complement of a sum equals the product of complements, while according to the second theorem the complement of a product equals the sum of complements. Figures(a) and (b) show logic diagram representations of De Morgan's theorems. While the first theorem can be interpreted to say that a multi-input NOR gate can be implemented as a multi-input bubbled AND gate, the second theorem, which is the dual of the first, can be interpreted to say that a multi-input NAND gate can be implemented as a multi-input bubbled OR gate.

DeMorgan's theorem can be proved as follows. Let us assume that all variables are in a logic '0'state. In that case

$$\text{LHS}=[\overline{X_1+X_2+X_3+\cdots+X_n}]=[\overline{0+0+0+\cdots+0}]=\overline{0}=1$$
$$\text{RHS}=\overline{X_1}.\overline{X_2}.\overline{X_3}.\ldots.\overline{X_n}=\overline{0}.\overline{0}.\overline{0}.\ldots.\overline{0}=1.1.1.\ldots.1=1$$

Therefore, LHS = RHS.

Now, let us assume that any one of the n variables, say X_1, is in a logic HIGH state:

AllAbtEngg Android Application for Anna University, Polytechnic & School

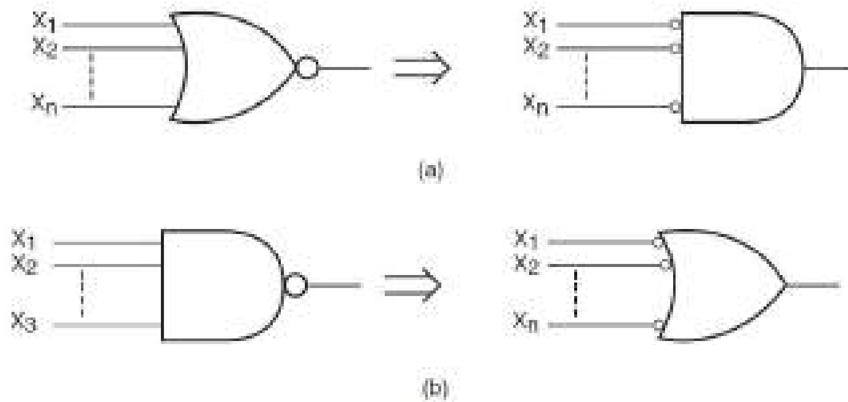


Figure DeMorgan's theorem.

$$\text{LHS} = \overline{[X_1 + X_2 + X_3 + \cdots X_n]} = \overline{[1+0+0+\cdots+0]} = \overline{1} = 0$$

$$\text{RHS} = \overline{X_1}.\overline{X_2}.\overline{X_3}.\ \ldots\ .\overline{X_n} = \overline{1}.\overline{0}.\overline{0}.\ \ldots\ .\overline{0} = 0.1.1.\ \ldots\ .1 = 0$$

Therefore, again LHS = RHS.

1.32.14 Theorem 14 (Transposition Theorem)

$$\text{(a) } X.Y + \overline{X}.Z = (X+Z).(\overline{X}+Y)$$

and

$$\text{(b) } (X+Y).(\overline{X}+Z) = X.Z + \overline{X}.Y$$

This theorem can be applied to any sum-of-products or product-of-sums expression having two terms, provided that a given variable in one term has its complement in the other. Table gives the proof of theorem 14(a) using the method of perfect induction. Theorem 14(b) is the dual of theorem 14(a)and hence stands proved.

As an example,

$$\overline{A}.B + A.\overline{B} = (A+B).(\overline{A}+\overline{B}) \quad \text{and} \quad A.B + \overline{A}.\overline{B} = (A+\overline{B}).(\overline{A}+B)$$

Incidentally, the first expression is the representation of a two-input EX-OR gate, while the second expression gives two forms of representation of a two-input EX-NOR gate.

www.AllAbtEngg.com

Table Proof of theorem 13(a).

X	Y	Z	XY	$\overline{X}Z$	X+Z	$\overline{X}+Y$	$XY+\overline{X}Z$	$(X+Z)(\overline{X}+Y)$
0	0	0	0	0	0	1	0	0
0	0	1	0	1	1	1	1	1
0	1	0	0	0	0	1	0	0
0	1	1	0	1	1	1	1	1
1	0	0	0	0	1	0	0	0
1	0	1	0	0	1	0	0	0
1	1	0	1	0	1	1	1	1
1	1	1	1	0	1	1	1	1

1.32.15 Theorem 15

(a) $X.f(X, \overline{X}, Y, Z, \ldots) = X.f(1, 0, Y, Z, \ldots)$

(b) $X + f(X, \overline{X}, Y, Z, \ldots) = X + f(0, 1, Y, Z, \ldots)$

According to theorem 15(a), if a variable X is multiplied by an expression containing X and $\overline{X}$ in addition to other variables, then all Xs and $\overline{X}$s can be replaced with 1s and 0s respectively. This would be valid as $X.X = X$ and $X.1 = X$. Also, $X.\overline{X} = 0$ and $X.0 = 0$. According to theorem 15(b), if a variable X is added to an expression containing terms having X and $\overline{X}$ in addition to other variables, then all Xs can be replaced with 0s and all $\overline{X}$s can be replaced with 1s. This is again permissible as $X + X$ as well as $X + 0$ equals X. Also, $X + \overline{X}$ and $\overline{X} + 1$ both equal 1.

This pair of theorems is very useful in eliminating redundancy in a given expression. An important corollary of this pair of theorems is that, if the multiplying variable is $\overline{X}$ in theorem 15(a), then all Xs will be replaced by 0s and all $\overline{X}$s will be replaced by 1s. Similarly, if the variable being added in theorem 15(b) is $\overline{X}$, then Xs and $\overline{X}$s in the expression are replaced by 1s and 0s respectively. In that case the two theorems can be written as follows:

AllAbtEngg Android Application for Anna University, Polytechnic & School

$$\text{(a) } \overline{X}.f(X, \overline{X}, Y, Z, \ldots) = \overline{X}.f(0, 1, Y, Z, \ldots)$$

$$\text{(b) } \overline{X} + f(X, \overline{X}, Y, Z, \ldots) = \overline{X} + f(1, 0, Y, Z, \ldots)$$

The theorems are further illustrated with the help of the following examples:

1. $A.[\overline{A}.B + A.\overline{C} + (\overline{A} + D).(A + \overline{E})] = A.[0.B + 1.\overline{C} + (0 + D).(1 + \overline{E})] = A.(\overline{C} + D).$
2. $\overline{A} + [\overline{A}.B + A.\overline{C} + (\overline{A} + B).(A + \overline{E})] = \overline{A} + [0.B + 1.\overline{C} + (0 + B).(1 + \overline{E})] = \overline{A} + \overline{C} + B.$

1.32.16 Theorem 16

$$\text{(a) } f(X, \overline{X}, Y, \ldots, Z) = X.f(1, 0, Y, \ldots, Z) + \overline{X}.f(0, 1, Y, \ldots, Z)$$

$$\text{(b) } f(X, \overline{X}, Y, \ldots, Z) = [X + f(0, 1, Y, \ldots, Z)][\overline{X} + f(1, 0, Y, \ldots, Z)]$$

The proof of theorem 16(a) is straightforward and is given as follows:

$$\begin{aligned} f(X, \overline{X}, Y, \ldots, Z) &= X.f(X, \overline{X}, Y, \ldots, Z) + \overline{X}.f(X, \overline{X}, Y, \ldots, Z) \\ &= X.f(1, 0, Y, \ldots, Z) + \overline{X}.f(0, 1, Y, \ldots, Z) \text{[Theorem 15(a)]} \end{aligned}$$

Also

$$\begin{aligned} f(X, \overline{X}, Y, \ldots, Z) &= [X + f(X, \overline{X}, Y, \ldots, Z)][\overline{X} + f(X, \overline{X}, Y, \ldots, Z)] \\ &= [X + f(0, 1, Y, \ldots, Z)][\overline{X} + f(1, 0, Y, \ldots, Z)] \text{[Theorem 15(b)]} \end{aligned}$$

1.32.17 Theorem 17 (Involution Law)

$$\overline{\overline{X}} = X$$

Involution law says that the complement of the complement of an expression leaves the expression unchanged. Also, the dual of the dual of an expression is the original expression. This theorem forms the basis of finding the equivalent product-of-sums expression for a given sum-of-products expression, and vice versa.

Example

AllAbtEngg Android Application for Anna University, Polytechnic & School

Prove the following:

1. $L.(M+\overline{N})+\overline{L}.\overline{P}.Q=(L+\overline{P}.Q).(\overline{L}+M+\overline{N})$.
2. $[A.\overline{B}+\overline{C}+\overline{D}].[D+(E+\overline{F}).G]=D.(A.\overline{B}+\overline{C})+\overline{D}.G.(E+\overline{F})$.

Solution

1. Let us assume that $L=X$, $(M+\overline{N})=Y$ and $\overline{P}.Q=Z$.
   The LHS of the given Boolean equation then reduces to $X.Y+\overline{X}.Z$.
   Applying the transposition theorem,

$$X.Y+\overline{X}.Z=(X+Z).(\overline{X}+Y)=(L+\overline{P}.Q)(\overline{L}+M+\overline{N})=\text{RHS}$$

2. Let us assume $\overline{D}=X$, $A.\overline{B}+\overline{C}=Y$ and $(E+\overline{F}).G=Z$.
   The LHS of given the Boolean equation then reduces to $(X+Y).(\overline{X}+Z)$.
   Applying the transposition theorem,

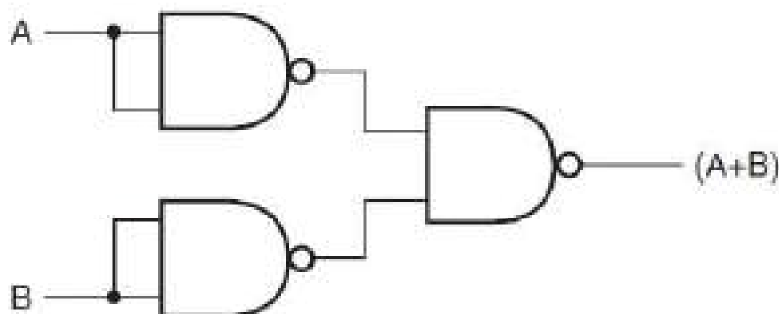
$$(X+Y).(\overline{X}+Z)=X.Z+\overline{X}.Y=\overline{D}.G.(E+\overline{F})+D.(A.\overline{B}+\overline{C})=\text{RHS}$$

Example

Starting with the Boolean expression for a two-input OR gate, apply Boolean laws and theorems to modify it in such a way as to facilitate the implementation of a two-input OR gate by using two-input NAND gates only.

Solution

- A two-input OR gate is represented by the Boolean equation $Y=(A+B)$, where A and B are the input logic variables and Y is the output.
- Now, $(A+B)=\overline{(\overline{\overline{A}+\overline{B}})}$ Involution law
  $=\overline{(\overline{A}.\overline{B})}$ DeMorgan's theorem
  $=\overline{[(\overline{A.A}).(\overline{B.B})]}$ Idempotent law
- Figure shows the NAND gate implementation of a two-input OR gate.



AllAbtEngg Android Application for Anna University, Polytechnic & School

Example

Apply suitable Boolean laws and theorems to modify the expression for a two-input EX-OR gate in such a way as to implement a two-input EX-OR gate by using the minimum number of two-input NAND gates only.

Solution

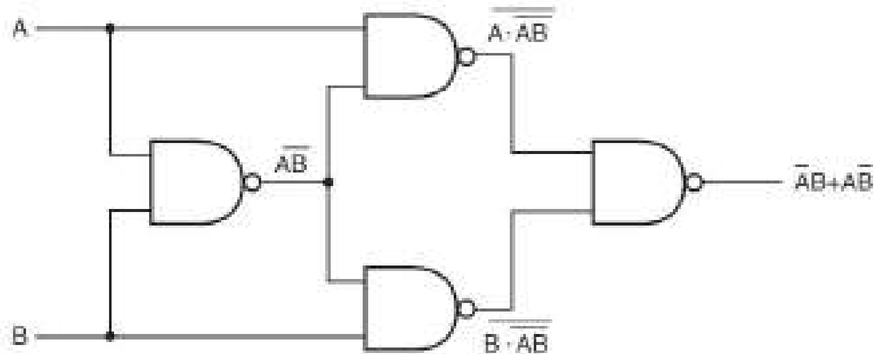
- A two-input EX-OR gate is represented by the Boolean expression $Y = \overline{A}.B + A.\overline{B}$.
- Now, $\overline{A}.B + A.\overline{B} = \overline{\overline{\overline{A}.B}} + \overline{\overline{A.\overline{B}}}$ Involution law

$$= \overline{\overline{\overline{A}.B}.\overline{A.\overline{B}}}$$ DeMorgan's law

$$= \overline{[\overline{B.(\overline{A}+\overline{B})}].[\overline{A.(\overline{A}+\overline{B})}]}$$

$$= \overline{(\overline{B.\overline{A.B}}).(\overline{A.\overline{A.B}})}$$

- Equation is in a form that can be implemented with NAND gates only.
- Figure shows the logic diagram.



1.33 Simplification Techniques

In this section, we will discuss techniques other than the application of laws and theorems of Boolean algebra discussed in the preceding paragraphs of this chapter for simplifying or more precisely minimizing a given complex Boolean expression. The primary objective of all simplification procedures is to obtain an expression that has the minimum number of terms. Obtaining an expression with the minimum number of literals is usually the secondary objective. If there is more than one possible solution with the same number of terms, the one having the minimum number of literals is the choice.

The techniques to be discussed include:

(a) the Quine–McCluskey tabular method;

(b) the Karnaugh map method.

Before we move on to discuss these techniques in detail, it would be relevant briefly to describe sum-of-products and product-of-sums Boolean expressions. The given Boolean expression will

AllAbtEngg Android Application for Anna University, Polytechnic & School

be in either of the two forms, and the objective will be to find a minimized expression in the same or the other form.

1.33.1 Sum-of-Products Boolean Expressions

A sum-of-products expression contains the sum of different terms, with each term being either a single literal or a product of more than one literal. It can be obtained from the truth table directly by considering those input combinations that produce a logic '1' at the output. Each such input combination produces a term. Different terms are given by the product of the corresponding literals. The sum of all terms gives the expression. For example, the truth table in Table can be represented by the Boolean expression

$$Y = \overline{A}\,.\overline{B}.\overline{C} + \overline{A}.B.C + A.B.\overline{C} + A.\overline{B}.C$$

Considering the first term, the output is '1' when A = 0 B = 0 and C = 0. This is possible only when $\overline{A}$, $\overline{B}$ and $\overline{C}$ are ANDed. Also, for the second term, the output is '1' only when B, C and $\overline{A}$ are ANDed. Other terms can be explained similarly. A sum-of-products expression is also known as a *minterm expression*.

Table truth table of boolean expression of equation

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

1.33.2 Product-of-Sums Expressions

A product-of-sums expression contains the product of different terms, with each term being either a single literal or a sum of more than one literal. It can be obtained from the truth table by considering those input combinations that produce a logic '0' at the output. Each such input combination gives a term, and the product of all such terms gives the expression. Different terms are obtained by taking the sum of the corresponding literals. Here, '0' and '1' respectively mean the uncomplemented and complemented variables, unlike sum-of-products expressions where '0' and '1' respectively mean complemented and uncomplemented variables.

AllAbtEngg Android Application for Anna University, Polytechnic & School

www.AllAbtEngg.com

To illustrate this further, consider once again the truth table in Table 6.5. Since each term in the case of the product-of-sums expression is going to be the sum of literals, this implies that it is going to be implemented using an OR operation. Now, an OR gate produces a logic '0' only when all its inputs are in the logic '0' state, which means that the first term corresponding to the second row of the truth table will be $A+B+\overline{C}$. The product-of-sums Boolean expression for this truth table is given by $(A+B+\overline{C}).(A+\overline{B}+C).(\overline{A}+B+C).(\overline{A}+\overline{B}+\overline{C})$.

Transforming the given product-of-sums expression into an equivalent sum-of-products expression is a straightforward process. Multiplying out the given expression and carrying out the obvious simplification provides the equivalent sum-of-products expression:

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

A given sum-of-products expression can be transformed into an equivalent product-of-sums expression by (a) taking the dual of the given expression, (b) multiplying out different terms to get the sum-of-products form, (c) removing redundancy and (d) taking a dual to get the equivalent product-of-sums expression. As an illustration, let us find the equivalent product-of-sums expression of the sum-of-products expression

$$A.B+\overline{A}.\overline{B}$$

The dual of the given expression $=(A+B).(\overline{A}+\overline{B})$:

$$(A+B).(\overline{A}+\overline{B})=A.\overline{A}+A.\overline{B}+B.\overline{A}+B.\overline{B}=0+A.\overline{B}+B.\overline{A}+0=A.\overline{B}+\overline{A}.B$$

The dual of $(A.\overline{B}+\overline{A}.B)=(A+\overline{B}).(\overline{A}+B)$. Therefore

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

1.33.3 Expanded Forms of Boolean Expressions

Expanded sum-of-products and product-of-sums forms of Boolean expressions are useful not only in analysing these expressions but also in the application of minimization techniques such as the Quine–McCluskey tabular method and the Karnaugh mapping method for simplifying given Boolean expressions. The expanded form, sum-of-products or product-of-sums, is obtained by including all possible combinations of missing variables.

As an illustration, consider the following sum-of-products expression:

$$A.\overline{B}+B.\overline{C}+A.B.\overline{C}+\overline{A}.C$$

AllAbtEngg Android Application for Anna University, Polytechnic & School

It is a three-variable expression. Expanded versions of different minterms can he written as follows:

- $A.\overline{B} = A.\overline{B}.(C+\overline{C}) = A.\overline{B}.C + A.\overline{B}.\overline{C}$.
- $B.\overline{C} = B.\overline{C}.(A+\overline{A}) = B.\overline{C}.A + B.\overline{C}.\overline{A}$.
- $A.B.\overline{C}$ is a complete term and has no missing variable.
- $\overline{A}.C = \overline{A}.C.(B+\overline{B}) = \overline{A}.C.B + \overline{A}.C.\overline{B}$.

The expanded sum-of-products expression is therefore given by

$$A.\overline{B}.C + A.\overline{B}.\overline{C} + A.B.\overline{C} + \overline{A}.B.\overline{C} + A.B.\overline{C} + \overline{A}.B.C + \overline{A}.\overline{B}.C = A.\overline{B}.C + A.\overline{B}.\overline{C}$$
$$+ A.B.\overline{C} + \overline{A}.B.\overline{C} + \overline{A}.B.C + \overline{A}.\overline{B}.C$$

As another illustration, consider the product-of-sums expression

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

It is four-variable expression with A, B, C and D being the four variables. $\overline{A}+B$ in this case expands to $(\overline{A}+B+C+D).(\overline{A}+B+C+\overline{D}).(\overline{A}+B+\overline{C}+D).(\overline{A}+B+\overline{C}+\overline{D})$.

The expanded product-of-sums expression is therefore given by

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

1.33.4 Canonical Form of Boolean Expressions

An expanded form of Boolean expression, where each term contains all Boolean variables in their true or complemented form, is also known as the *canonical form* of the expression.

As an illustration, $f(A, B, C) = \overline{A}.\overline{B}.\overline{C} + \overline{A}.\overline{B}.C + A.B.C$ is a Boolean function of three variables expressed in canonical form. This function after simplification reduces to $\overline{A}.\overline{B} + A.B.C$ and loses its canonical form.

1.33.5 *Σ and Π and Nomenclature*

Σ and Π notations are respectively used to represent sum-of-products and product-of-sums Boolean expressions. We will illustrate these notations with the help of examples. Let us consider the following Boolean function:

$$f(A, B, C, D) = A.\overline{B}.\overline{C} + A.B.C.D + \overline{A}.B.\overline{C}.D + \overline{A}.\overline{B}.\overline{C}.D$$

We will represent this function using Σ notation. The first step is to write the expanded sum-of-products given by

AllAbtEngg Android Application for Anna University, Polytechnic & School

$$f(A, B, C, D) = A.\overline{B}.\overline{C}.(D+\overline{D}) + A.B.C.D + \overline{A}.B.\overline{C}.D + \overline{A}.\overline{B}.\overline{C}.D$$
$$= A.\overline{B}.\overline{C}.D + A.\overline{B}.\overline{C}.\overline{D} + A.B.C.D + \overline{A}.B.\overline{C}.D + \overline{A}.\overline{B}.\overline{C}.D$$

Different terms are then arranged in ascending order of the binary numbers represented by various terms, with true variables representing a '1' and a complemented variable representing a '0'. The expression becomes

$$f(A, B, C, D) = \overline{A}.\overline{B}.\overline{C}.D + \overline{A}.B.\overline{C}.D + A.\overline{B}.\overline{C}.\overline{D} + A.\overline{B}.\overline{C}.D + A.B.C.D$$

The different terms represent 0001, 0101, 1000, 1001 and 1111. The decimal equivalent of these terms enclosed in the Σ then gives the Σ notation for the given Boolean function. That is, $f(A, B, C, D) = \sum 1, 5, 8, 9, 15$.

The complement of $f(A, B, C, D)$, that is, $f'(A, B, C, D)$, can be directly determined from Σ notation by including the left-out entries from the list of all possible numbers for a four-variable function. That is,

$$f'(A, B, C, D) = \sum 0, 2, 3, 4, 6, 7, 10, 11, 12, 13, 14$$

Let us now take the case of a product-of-sums Boolean function and its representation in Π nomenclature. Let us consider the Boolean function

$$f(A, B, C, D) = (B + \overline{C} + \overline{D}).(\overline{A} + \overline{B} + C + D).(A + \overline{B} + \overline{C} + \overline{D})$$

The expanded product-of-sums form is given by

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

The binary numbers represented by the different sum terms are 0011, 1011, 1100 and 0111 (true and complemented variables here represent 0 and 1 respectively). When arranged in ascending order, these numbers are 0011, 0111, 1011 and 1100. Therefore,

$$f(A, B, C, D) = \prod 3, 7, 11, 12 \quad \text{and} \quad f'(A, B, C, D) = \prod 0, 1, 2, 4, 5, 6, 8, 9, 10, 13, 14, 15$$

An interesting corollary of what we have discussed above is that, if a given Boolean function $f(A,B,C)$ is given by $f(A, B, C) = \sum 0, 1, 4, 7$, then

$$f(A, B, C) = \prod 2, 3, 5, 6 \quad \text{and} \quad f'(A, B, C) = \sum 2, 3, 5, 6 = \prod 0, 1, 4, 7$$

Optional combinations can also be incorporated into Σ and Π nomenclature using suitable identifiers; ϕ or d are used as identifiers. For example, if $f(A, B, C) = \overline{A}.\overline{B}.\overline{C} + A.\overline{B}.\overline{C} + A.\overline{B}.C$ and $\overline{A}.B.C$, $A.B.C$ are optional combinations, then

$$f(A, B, C) = \sum 0, 4, 5 + \sum_{\phi} 3, 7 = \sum 0, 4, 5 + \sum_{d} 3, 7$$
$$f(A, B, C) = \prod 1, 2, 6 + \prod_{\phi} 3, 7 = \prod 1, 2, 6 + \prod_{d} 3, 7$$

Example

AllAbtEngg Android Application for Anna University, Polytechnic & School

For a Boolean function $f(A, B) = \sum 0, 2$, prove that $f(A, B) = \prod 1, 3$ and $f'(A, B) = \sum 1, 3 = \prod 0, 2$.

Solution

- $f(A, B) = \sum 0, 2 = \overline{A}.\overline{B} + A.\overline{B} = \overline{B}.(A + \overline{A}) = \overline{B}$.
- Now, $\prod 1, 3 = (A + \overline{B}).(\overline{A} + \overline{B}) = A.\overline{A} + A.\overline{B} + \overline{B}.\overline{A} + \overline{B}.\overline{B} = A.\overline{B} + \overline{A}.\overline{B} + \overline{B} = \overline{B}$.
- Now, $\sum 1, 3 = \overline{A}.B + A.B = B.(\overline{A} + A) = B$.
  and $\prod 0, 2 = (A + B).(\overline{A} + B) = A.\overline{A} + A.B + B.\overline{A} + B.B = A.B + \overline{A}.B + B = B$.
- Therefore, $\sum 1, 3 = \prod 0, 2$.
- Also, $f(A, B) = \overline{B}$.
- Therefore, $f'(A, B) = B$ or $f'(A, B) = \sum 1, 3 = \prod 0, 2$.

1.34 Quine–McCluskey Tabular Method

The Quine–McCluskey tabular method of simplification is based on the complementation theorem, which says that

$$X.Y + X.\overline{Y} = X$$

where X represents either a variable or a term or an expression and Y is a variable. This theorem implies that, if a Boolean expression contains two terms that differ only in one variable, then they can be combined together and replaced with a term that is smaller by one literal. The same procedure is applied for the other pairs of terms wherever such a reduction is possible. All these terms reduced by one literal are further examined to see if they can be reduced further. The process continues until the terms become irreducible. The irreducible terms are called *prime implicants*. An optimum set of prime implicants that can account for all the original terms then constitutes the minimized expression. The technique can be applied equally well for minimizing sum-of-products and product-of-sums expressions and is particularly useful for Boolean functions having more than six variables as it can be mechanized and run on a computer. On the other hand, the Karnaugh mapping method, to be discussed later, is a graphical method and becomes very cumbersome when the number of variables exceeds six.

The step-by-step procedure for application of the tabular method for minimizing Boolean expressions, both sum-of-products and product-of-sums, is outlined as follows:

1. The Boolean expression to be simplified is expanded if it is not in expanded form.
2. Different terms in the expression are divided into groups depending upon the number of 1s they have. True and complemented variables in a sum-of-products expression mean '1' and '0' respectively.

AllAbtEngg Android Application for Anna University, Polytechnic & School

The reverse is true in the case of a product-of-sums expression. The groups are then arranged, beginning with the group having the least number of 1s in its included terms. Terms within the same group are arranged in ascending order of the decimal numbers represented by these terms.

As an illustration, consider the expression

$$A.B.C+\overline{A}.B.C+A.\overline{B}.\overline{C}+A.\overline{B}.C+\overline{A}.\overline{B}.\overline{C}$$

The grouping of different terms and the arrangement of different terms within the group are shown below:

Term		Binary	Group
$\overline{A}.\overline{B}.\overline{C}$		000	First group
$\overline{A.\overline{B}.\overline{C}}$		100	Second group
$\overline{\overline{A}.B.C}$	→	011	Third group
$A.\overline{B}.C$		101	
$\overline{ABC}$		111	Fourth group

As another illustration, consider a product-of-sums expression given by

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

The formation of groups and the arrangement of terms within different groups for the product-of-sums expression are as follows:

Term		Binary
$A.B.C.D$		0000
$A.B.\overline{C}.\overline{D}$		0011
$A.\overline{B}.C.\overline{D}$		0101
$\overline{A}.B.\overline{C}.D$	→	1010
$A.\overline{B}.\overline{C}.\overline{D}$		0111
$\overline{A}.\overline{B}.\overline{C}.D$		1110
$\overline{\overline{A}.\overline{B}.\overline{C}.\overline{D}}$		1111

It may be mentioned here that the Boolean expressions that we have considered above did not contain any optional terms. If there are any, they are also considered while forming groups. This completes the first table.

3. The terms of the first group are successively matched with those in the next adjacent higher-order group to look for any possible matching and consequent reduction. The terms are considered matched when all literals except for one match. The pairs of matched terms are replaced with a

AllAbtEngg Android Application for Anna University, Polytechnic & School

www.AllAbtEngg.com

single term where the position of the unmatched literals is replaced with a dash (—). These new terms formed as a result of the matching process find a place in the second table. The terms in the first table that do not find a match are called the prime implicants and are marked with an asterisk (*). The matched terms are ticked (✓).

4. Terms in the second group are compared with those in the third group to look for a possible match. Again, terms in the second group that do not find a match become the prime implicants.
5. The process continues until we reach the last group. This completes the first round of matching. The terms resulting from the matching in the first round are recorded in the second table.
6. The next step is to perform matching operations in the second table. While comparing the terms for a match, it is important that a dash (—) is also treated like any other literal, that is, the dash signs also need to match. The process continues on to the third table, the fourth table and so on until the terms become irreducible any further.
7. An optimum selection of prime implicants to account for all the original terms constitutes the terms for the minimized expression. Although optional (also called 'don't care') terms are considered for matching, they do not have to be accounted for once prime implicants have been identified.

Let us consider an example. Consider the following sum-of-products expression:

$$\overline{A}.B.C+\overline{A}.\overline{B}.D+A.\overline{C}.D+B.\overline{C}.\overline{D}+\overline{A}.B.\overline{C}.D$$

In the first step, we write the expanded version of the given expression. It can be written as follows:

$$\overline{A}.B.C.D+\overline{A}.B.C.\overline{D}+\overline{A}.\overline{B}.C.D+\overline{A}.\overline{B}.\overline{C}.D+A.B.\overline{C}.D+A.\overline{B}.\overline{C}.D+A.B.\overline{C}.\overline{D}$$
$$+\overline{A}.B.\overline{C}.\overline{D}+\overline{A}.B.\overline{C}.D$$

The formation of groups, the placement of terms in different groups and the first-round matching are shown as follows:

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

A	B	C	D	
0	0	0	1	✓
0	1	0	0	✓
0	0	1	1	✓
0	1	0	1	✓
0	1	1	0	✓
1	0	0	1	✓
1	1	0	0	✓
0	1	1	1	✓
1	1	0	1	✓

A	B	C	D	
0	0	—	1	✓
0	—	0	1	✓
—	0	0	1	✓
0	1	0	—	✓
0	1	—	0	✓
—	1	0	0	✓
0	—	1	1	✓
0	1	—	1	✓
—	1	0	1	✓
0	1	1	—	✓
1	—	0	1	✓
1	1	0	—	✓

The second round of matching begins with the table shown on the previous page. Each term in the first group is compared with every term in the second group. For instance, the first term in the first group 00−1 matches with the second term in the second group 01−1 to yield 0−−1, which is recorded in the table shown below. The process continues until all terms have been compared for a possible match. Since this new table has only one group, the terms contained therein are all prime implicants. In the present example, the terms in the first and second tables have all found a match. But that is not always the case.

AllAbtEngg Android Application for Anna University, Polytechnic & School

A	B	C	D	
0	–	–	1	*
–	–	0	1	*
0	1	–	–	*
–	1	0	–	*

The next table is what is known as the prime implicant table. The prime implicant table contains all the original terms in different columns and all the prime implicants recorded in different rows as shown below:

0001	0011	0100	0101	0110	0111	1001	1100	1101		
✓	✓		✓		✓				0– –1	$P \rightarrow \overline{A}.D$
✓			✓			✓		✓	– –01	$Q \rightarrow \overline{C}.D$
		✓	✓	✓	✓				01– –	$R \rightarrow \overline{A}.B$
		✓	✓				✓	✓	–10–	$S \rightarrow B.\overline{C}$

Each prime implicant is identified by a letter. Each prime implicant is then examined one by one and the terms it can account for are ticked as shown. The next step is to write a product-of-sums expression using the prime implicants to account for all the terms. In the present illustration, it is given as follows.

$$(P+Q).(P).(R+S).(P+Q+R+S).(R).(P+R).(Q).(S).(Q+S)$$

Obvious simplification reduces this expression to $PQRS$ which can be interpreted to mean that all prime implicants, that is, P, Q, R and S, are needed to account for all the original terms.

Therefore, the minimized expression $= \overline{A}.D + \overline{C}.D + \overline{A}.B + B.\overline{C}$.

What has been described above is the formal method of determining the optimum set of prime implicants. In most of the cases where the prime implicant table is not too complex, the exercise can be done even intuitively. The exercise begins with identification of those terms that can be accounted for by only a single prime implicant. In the present example, 0011, 0110, 1001 and 1100 are such terms. As a result, P, Q, R and S become the essential prime implicants. The next step is to find out if any terms have not been covered by the essential prime implicants. In the present case, all terms have been covered by essential prime implicants. In fact, all prime implicants are essential prime implicants in the present example.

As another illustration, let us consider a product-of-sums expression given by

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

The procedure is similar to that described for the case of simplification of sum-of-products expressions. The resulting tables leading to identification of prime implicants are as follows:

AllAbtEngg Android Application for Anna University, Polytechnic & School

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

A	B	C	D	
0	1	0	1	✓
0	1	1	1	✓
1	1	0	1	✓
1	1	1	0	✓
1	1	1	1	✓

A	B	C	D	
0	1	–	1	✓
–	1	0	1	✓
–	1	1	1	✓
1	1	–	1	✓
1	1	1	–	*

A	B	C	D	
–	1	–	1	*

The prime implicant table is constructed after all prime implicants have been identified to look for the optimum set of prime implicants needed to account for all the original terms. The prime implicant table shows that both the prime implicants are the essential ones:

0101	0111	1101	1110	1111	Prime implicants
			✓	✓	111–
✓	✓	✓		✓	–1–1

The minimized expression = $(\overline{A}+\overline{B}+\overline{C}).(\overline{B}+\overline{D})$.

Example

Using the Quine–McCluskey tabular method, find the minimum sum of products for $f(A, B, C, D) = \sum(1, 2, 3, 9, 12, 13, 14) + \sum_{\phi}(0, 7, 10, 15)$.

Solution

The different steps to finding the solution to the given problem are tabulated below. As we can see, eight prime implicants have been identified. These prime implicants along with the inputs constitute the prime implicant table. Remember that optional inputs are not considered while constructing the prime implicant table:

A	B	C	D	
0	0	0	0	✓
0	0	0	1	✓
0	0	1	0	✓
0	0	1	1	✓
1	0	0	1	✓
1	0	1	0	✓
1	1	0	0	✓
0	1	1	1	✓
1	1	0	1	✓
1	1	1	0	✓
1	1	1	1	✓

A	B	C	D	
0	0	0	–	✓
0	0	–	0	✓
0	0	–	1	✓
–	0	0	1	*
0	0	1	–	✓
–	0	1	0	*
0	–	1	1	*
1	–	0	1	*
1	–	1	0	*
1	1	0	–	✓
1	1	–	0	✓
–	1	1	1	*
1	1	–	1	✓
1	1	1	–	✓

A	B	C	D	
0	0	–	–	*
1	1	–	–	*

AllAbtEngg Android Application for Anna University, Polytechnic & School

The product-of-sums expression that tells about the combination of prime implicants required to account for all the terms is given by the expression

$$(L+S).(M+S).(N+S).(L+P).(T).(P+T).(Q+T)$$

After obvious simplification, this reduces to the expression

$$\begin{aligned} &T.(L+S).(M+S).(N+S).(L+P) \\ &= T.(LM+LS+MS+S).(LN+PN+LS+PS) \\ &= T.(LM+S).(LN+PN+LS+PS) \\ &= T.(LMN+LMPN+LMS+LMPS+LNS+PNS+LS+PS) \\ &= T.(LMN+LMPN+LS+PS) \\ &= TLMN+TLMPN+TLS+TPS \end{aligned}$$

0001	0010	0011	1001	1100	1101	1110	Prime implicants	
✓			✓				–001	L
	✓						–010	M
		✓					0–11	N
			✓		✓		1–01	P
						✓	1–10	Q
							–111	R
✓	✓	✓					00––	S
				✓	✓	✓	11––	T

The sum-of-products Boolean expression (6.39) states that all the input combinations can be accounted for by the prime implicants (T, L, M, N) or (T, L, M, P, N) or (T, L, S) or (T, P, S). The most optimum expression would result from either TLS or TPS. Therefore, the minimized Boolean function is given by

$$f(A,B,C,D) = A.B+\overline{B}.\overline{C}.D+\overline{A}.\overline{B}$$

or by

$$f(A,B,C,D) = A.B+\overline{A}.\overline{B}+A.\overline{C}.D$$

1.35 Karnaugh Map Method

A Karnaugh map is a graphical representation of the logic system. It can be drawn directly from either minterm (sum-of-products) or maxterm (product-of-sums) Boolean expressions. Drawing a Karnaugh map from the truth table involves an additional step of writing the minterm or maxterm expression depending upon whether it is desired to have a minimized sum-of-products or a minimized product-of-sums expression.

1.35.1 Construction of a Karnaugh Map

An n-variable Karnaugh map has 2^n squares, and each possible input is allotted a square. In the case of a minterm Karnaugh map, '1' is placed in all those squares for which the output is '1', and '0'

AllAbtEngg Android Application for Anna University, Polytechnic & School

is placed in all those squares for which the output is '0'. 0s are omitted for simplicity. An 'X' is placed in squares corresponding to 'don't care' conditions. In the case of a maxterm Karnaugh map, a '1' is placed in all those squares for which the output is '0', and a '0' is placed for input entries corresponding to a '1' output. Again, 0s are omitted for simplicity, and an 'X' is placed in squares corresponding to 'don't care' conditions.

The choice of terms identifying different rows and columns of a Karnaugh map is not unique for a given number of variables. The only condition to be satisfied is that the designation of adjacent rows and adjacent columns should be the same except for one of the literals being complemented. Also, the extreme rows and extreme columns are considered adjacent. Some of the possible designation styles for two-, three- and four-variable minterm Karnaugh maps are given in Figs

The style of row identification need not be the same as that of column identification as long as it meets the basic requirement with respect to adjacent terms. It is, however, accepted practice to adopt a uniform style of row and column identification. Also, the style shown in Figs (a), (a) and (a) is more commonly used. Some more styles are shown in Fig. . A similar discussion applies for maxterm Karnaugh maps.

Having drawn the Karnaugh map, the next step is to form groups of 1s as per the following guidelines:

1. Each square containing a '1' must be considered at least once, although it can be considered as often as desired.
2. The objective should be to account for all the marked squares in the minimum number of groups.
3. The number of squares in a group must always be a power of 2, i.e. groups can have 1, 2, 4, 8, 16, . . . squares.
4. Each group should be as large as possible, which means that a square should not be accounted for by itself if it can be accounted for by a group of two squares; a group of two squares should not be made if the involved squares can be included in a group of four squares and so on.
5. 'Don't care' entries can be used in accounting for all of 1-squares to make optimum groups. They are marked 'X' in the corresponding squares. It is, however, not necessary to account for all 'don't care' entries. Only such entries that can be used to advantage should be used.

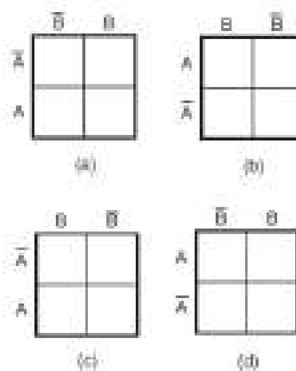


Figure Two-variable Karnaugh map.

AllAbtEngg Android Application for Anna University, Polytechnic & School

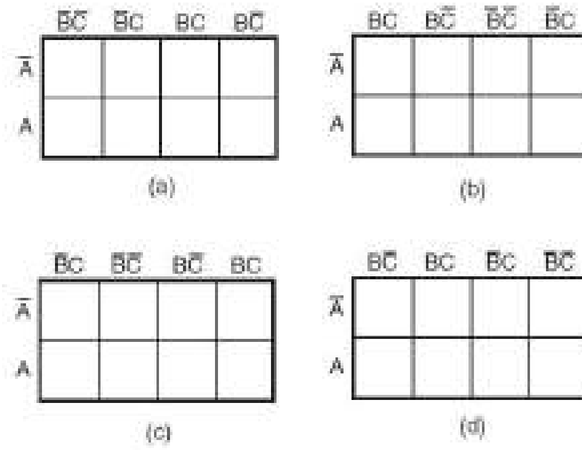


Figure Three-variable Karnaugh map.

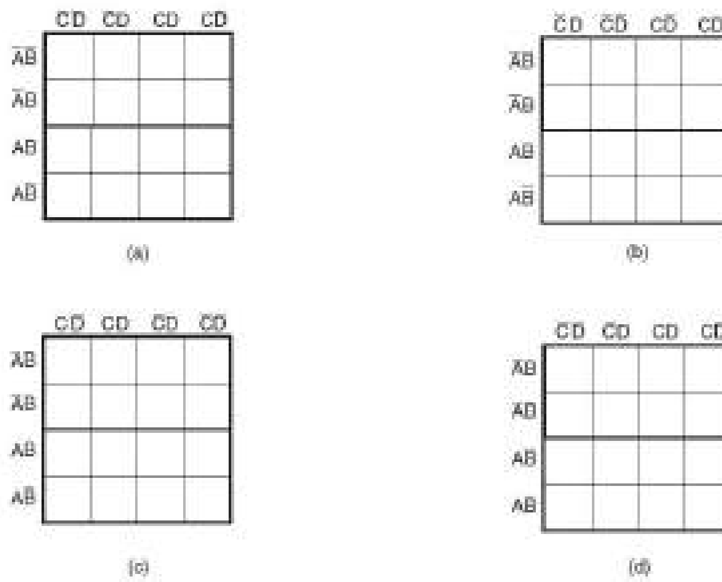


Figure Four-variable Karnaugh map.

AllAbtEngg Android Application for Anna University, Polytechnic & School

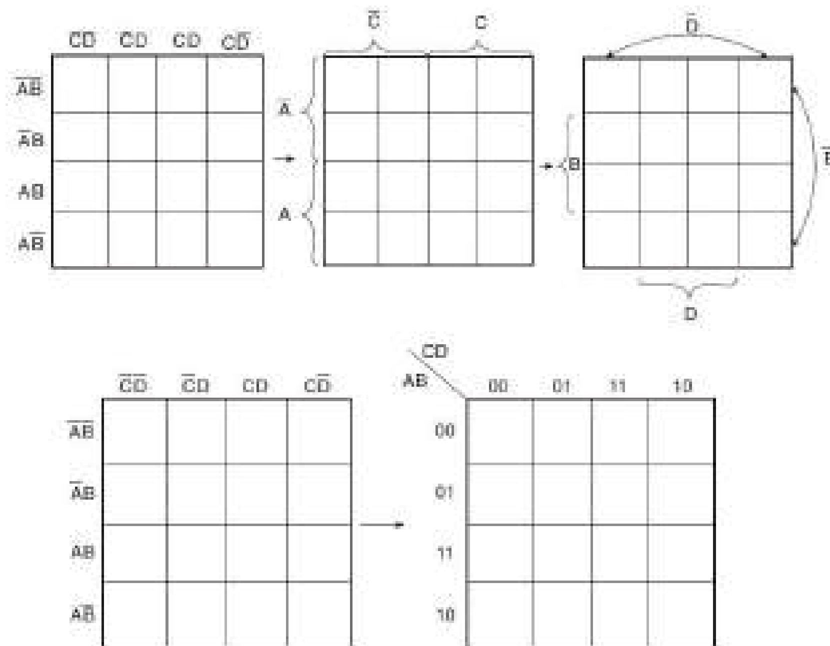


Figure Different styles of row and column identification.

Having accounted for groups with all 1s, the minimum 'sum-of-products' or 'product-of-sums' expressions can be written directly from the Karnaugh map.

Figure 6.10 shows the truth table, minterm Karnaugh map and maxterm Karnaugh map of the Boolean function of a two-input OR gate. The minterm and maxterm Boolean expressions for the two-input OR gate are as follows:

$$Y = A + B \text{ (maxterm or product-of-sums)}$$

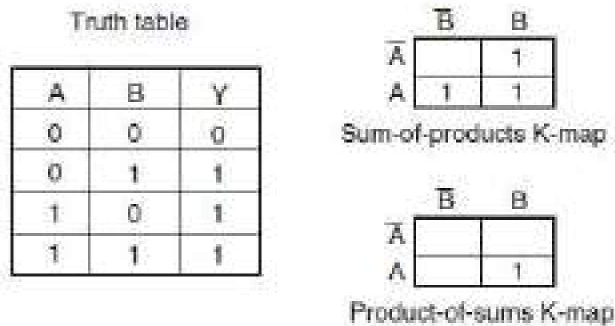
$$Y = \overline{A}.B + A.\overline{B} + A.B \text{ (minterm or sum-of-products)}$$

Figure 6.11 shows the truth table, minterm Karnaugh map and maxterm Karnaugh map of the three-variable Boolean function

$$Y = \overline{A}.\overline{B}.\overline{C} + \overline{A}.B.\overline{C} + A.\overline{B}.\overline{C} + A.B.\overline{C}$$

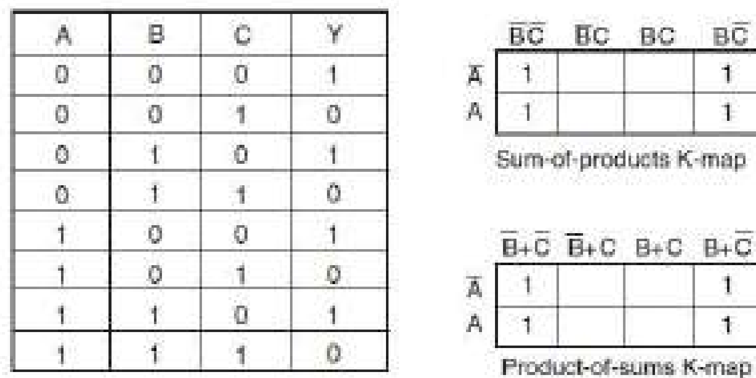
$$Y = (\overline{A} + \overline{B} + \overline{C}).(\overline{A} + B + \overline{C}).(A + \overline{B} + \overline{C}).(A + B + \overline{C})$$

AllAbtEngg Android Application for Anna University, Polytechnic & School


Truth table

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

Figure Two-variable Karnaugh maps.



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

Figure Three-variable Karnaugh maps.

$$Y=\overline{A}.\overline{B}.\overline{C}.\overline{D}+\overline{A}.\overline{B}.\overline{C}.D+\overline{A}.B.\overline{C}.\overline{D}+\overline{A}.B.\overline{C}.D+A.\overline{B}.\overline{C}.\overline{D}+A.\overline{B}.\overline{C}.D+A.B.\overline{C}.\overline{D}+A.B.\overline{C}.D$$

$$Y=(A+B+\overline{C}+D).(A+B+\overline{C}+\overline{D}).(A+\overline{B}+\overline{C}+D).(A+\overline{B}+\overline{C}+\overline{D})$$
$$.(\overline{A}+B+\overline{C}+D).(\overline{A}+B+\overline{C}+\overline{D}).(\overline{A}+\overline{B}+\overline{C}+D).(\overline{A}+\overline{B}+\overline{C}+\overline{D})$$

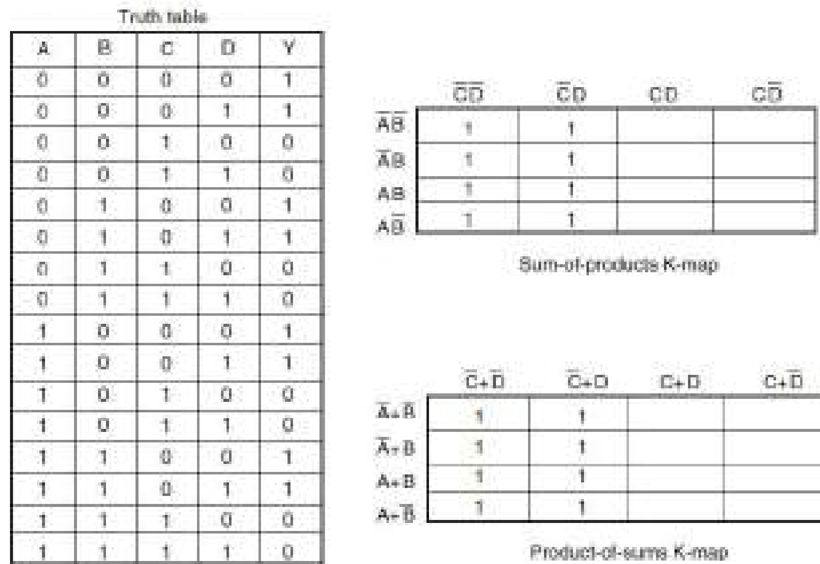
$$Y=\overline{A}.\overline{B}.\overline{C}.\overline{D}+\overline{A}.\overline{B}.C.\overline{D}+\overline{A}.B.\overline{C}.D+\overline{A}.B.C.D+A.\overline{B}.\overline{C}.\overline{D}+A.\overline{B}.C.\overline{D}+A.B.\overline{C}.D+A.B.C.D$$

$$Y=(A+B+C+\overline{D}).(A+B+\overline{C}+\overline{D}).(A+\overline{B}+C+D).(A+\overline{B}+C+\overline{D}).(A+\overline{B}+\overline{C}+\overline{D})$$
$$.(A+\overline{B}+\overline{C}+D).(\overline{A}+\overline{B}+C+\overline{D}).(\overline{A}+\overline{B}+\overline{C}+\overline{D}).(\overline{A}+B+C+\overline{D}).(\overline{A}+B+\overline{C}+\overline{D})$$

$$Y=\overline{B}.\overline{D}+B.D$$

$$Y=\overline{D}.(A+\overline{B})$$

AllAbtEngg Android Application for Anna University, Polytechnic & School



Truth table

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

	$\bar{C}\bar{D}$	$\bar{C}D$	CD	$C\bar{D}$
$\bar{A}\bar{B}$	1	1		
$\bar{A}B$	1	1		
AB	1	1		
$A\bar{B}$	1	1		

Sum-of-products K-map

	$\bar{C}+\bar{D}$	$\bar{C}+D$	$C+D$	$C+\bar{D}$
$\bar{A}+\bar{B}$	1	1		
$\bar{A}+B$	1	1		
$A+B$	1	1		
$A+\bar{B}$	1	1		

Product-of-sums K-map

Figure Four-variable Karnaugh maps.

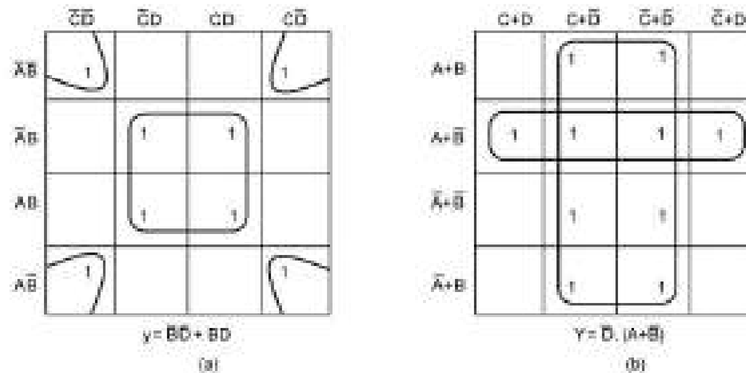


Figure Group formation in minterm and maxterm Karnaugh maps.

1.36 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

AllAbtEngg Android Application for Anna University, Polytechnic & School

www.AllAbtEngg.com

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. Different logic families used to hardware-implement different logic functions in the form of digital integrated circuits are discussed in the following chapter.

1.37 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.

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.

1.38 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, 10and 11. Figure (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 2n possible input combinations, or in other words 2n rows. Figure 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. is

AllAbtEngg Android Application for Anna University, Polytechnic & School

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 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 Truth table of a three-input logic system

1.39 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.

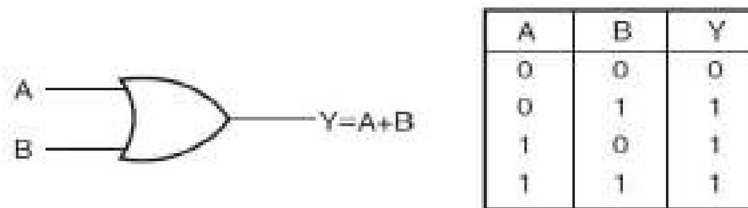
1.39.1 OR Gate

AllAbtEngg Android Application for Anna University, Polytechnic & School

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 Band 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 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

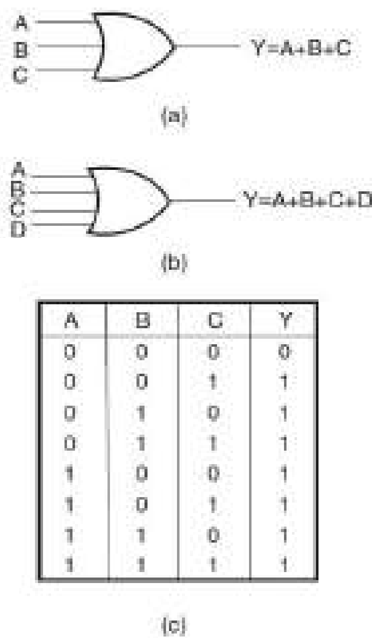
$$Y = A + B$$

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 Two-input OR gate.



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

AllAbtEngg Android Application for Anna University, Polytechnic & School

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

Figures (a) and (b) show the circuit symbol of three-input and four-input OR gates. Figure(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

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

Solution

Figure(a) shows one possible arrangement of two-input OR gates that simulates a four-input ORgate. A, B, C and D are logic inputs and Y 3 is the output. Figure(b) shows another possible arrangement. In the case of Fig.(a), the output of OR gate 1 is Y 1 = (A + B). The second

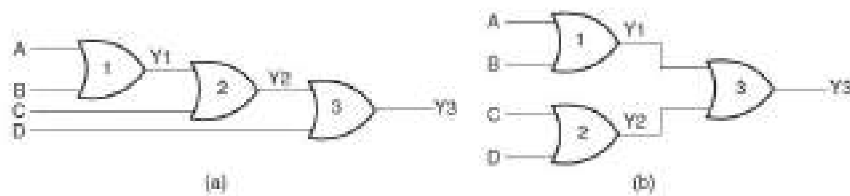


Figure Example.

OR gate produces the output Y 2 = (Y 1 + C) = (A + B + C). Similarly, the output of OR gate 3 isY 3 = (Y 2 + D) = (A + B + C + D). In the case of Fig.(b), the output of OR gate 1 is Y 1 = (A + B). The second OR gate produces the output Y 2 = (C + D). Output Y 3 of the third OR gate is given by (Y 1 + Y 2) = (A + B + C + D).

Example

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

Solution

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

AllAbtEngg Android Application for Anna University, Polytechnic & School

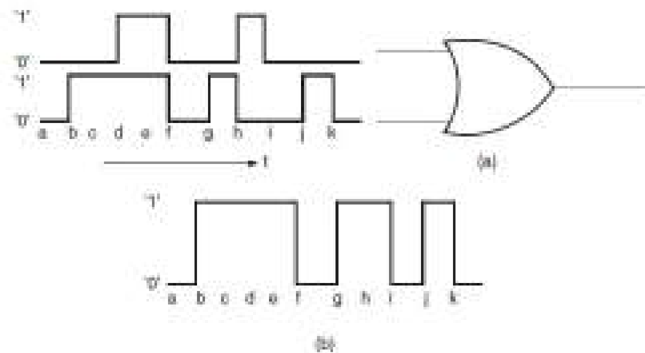
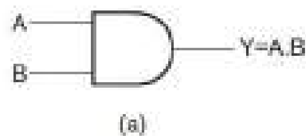


Figure Example.

1.39.2 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 (a) and (b) respectively. Figures (a)and (b) show the logic symbols of three-input and four-input AND gates respectively. Figure(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:

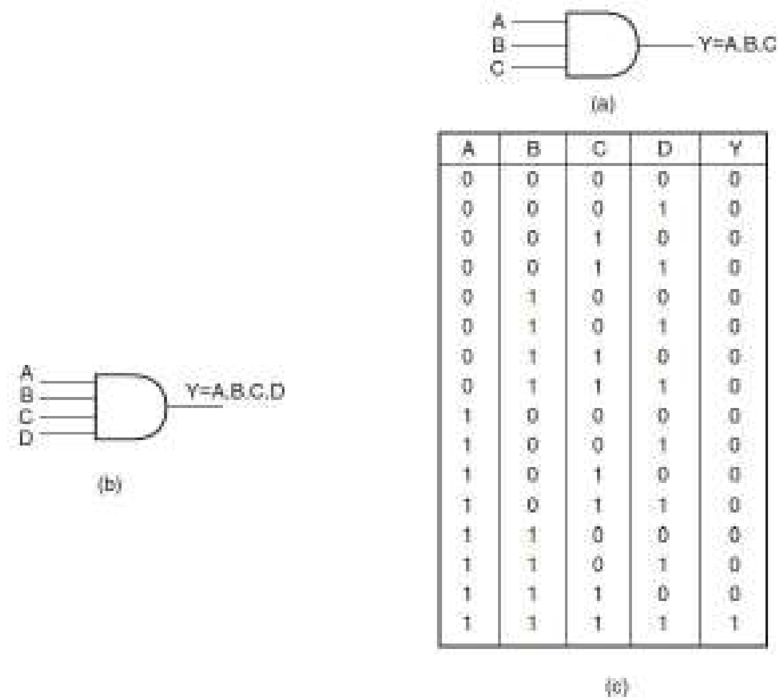


A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

(b)

Figure Two-input AND gate.

AllAbtEngg Android Application for Anna University, Polytechnic & School



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

Figure (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

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

Solution

Figure shows the hardware implementation of a four-input AND gate using two-input AND gates. The output of AND gate 1 is Y 1 = A B. The second AND gate produces an output Y 2 given byY 2 = Y 1 C = A B C. Similarly, the output of AND gate 3 is Y = Y 2.D = A B C D and hence the result.

AllAbtEngg Android Application for Anna University, Polytechnic & School

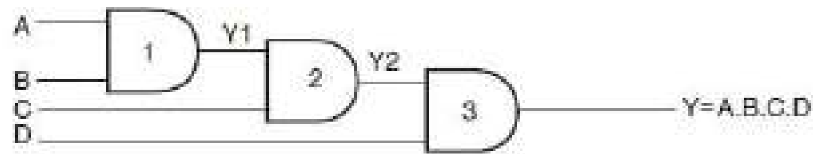
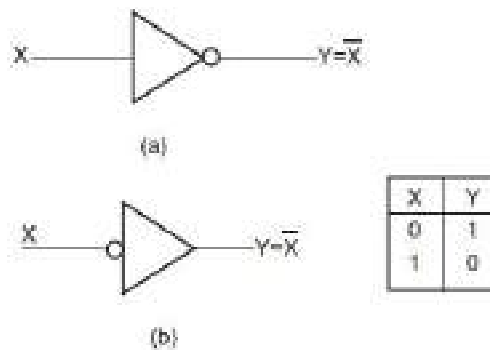


Figure Implementation of a four-input AND gate using two-input AND gates.

1.39.3 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 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.



X	Y
0	1
1	0

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

Example

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

Solution

In the case of the OR gate arrangement of Fig. (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.(b), the output will be permanently in

AllAbtEngg Android Application for Anna University, Polytechnic & School

www.AllAbtEngg.com

logic ‘0’state as the two inputs can never be in logic ‘1’ state together owing to the presence of the inverter.

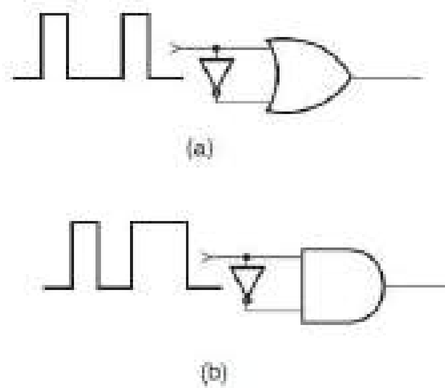


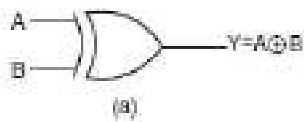
Figure Example.

1.39.4 EXCLUSIVE-OR Gate

The EXCLUSIVE-OR gate, commonly written as EX-OR gate, is a two-input, one-output gate. Figures (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 (c) shows the truth table of a four-input EX-OR function. The output of a two-input EX-OR gate is expressed by

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

AllAbtEngg Android Application for Anna University, Polytechnic & School



(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 (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

1.39.5 NAND Gate

NAND stands for NOT AND. An AND gate followed by a NOT circuit makes it a NAND gate [Fig. (a)]. Figure (b) shows the circuit symbol of a two-input NAND gate. The truth table of a NAND gate is obtained from the truth table of an AND gate by complementing the output entries [Fig. (c)]. The output of a NAND gate is a logic '0' when all its inputs are a logic '1'. For all other input combinations, the output is a logic '1'. NAND gate operation is logically expressed as

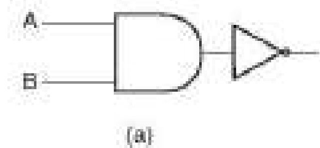
$$Y = \overline{A.B}$$

In general, the Boolean expression for a NAND gate with more than two inputs can be written as

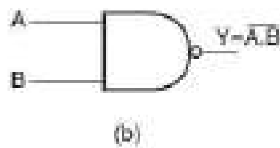
$$Y = \overline{(A.B.C.D...)}$$

AllAbtEngg Android Application for Anna University, Polytechnic & School

www.AllAbtEngg.com



(a)



(b)

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

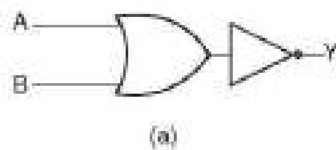
(c)

Figure (a) Two-input NAND implementation using an AND gate and a NOT circuit, (b) the circuit symbol of a two-input NAND gate and (c) the truth table of a two-input NAND gate.

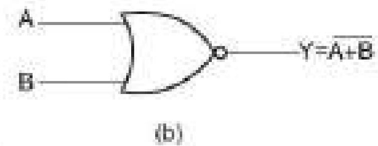
1.39.6 NOR Gate

NOR stands for NOT OR. An OR gate followed by a NOT circuit makes it a NOR gate [Fig. (a)]. The truth table of a NOR gate is obtained from the truth table of an OR gate by complementing the output entries. The output of a NOR gate is a logic '1' when all its inputs are logic '0'. For all other input combinations, the output is a logic '0'. The output of a two-input NOR gate is logically expressed as

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



(a)



(b)

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

(c)

AllAbtEngg Android Application for Anna University, Polytechnic & School

Figure (a) Two-input NOR implementation using an OR gate and a NOT circuit, (b) the circuit symbol of a two-input NOR gate and (c) the truth table of a two-input NOR gate.

In general, the Boolean expression for a NOR gate with more than two inputs can be written as

$$Y = \overline{(A + B + C + D \ldots)}$$

1.39.7 EXCLUSIVE-NOR Gate

EXCLUSIVE-NOR (commonly written as EX-NOR) means NOT of EX-OR, i.e. the logic gate that we get by complementing the output of an EX-OR gate. Figure shows its circuit symbol along with its truth table.

The truth table of an EX-NOR gate is obtained from the truth table of an EX-OR gate by complementing the output entries. Logically,

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

(a)

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

(b)

Figure (a) Circuit symbol of a two-input EXCLUSIVE-NOR gate and (b) the truth table of a two-input EXCLUSIVE-NOR gate.

The output of a two-input EX-NOR gate is a logic '1' when the inputs are like and a logic '0' when they are unlike. In general, the output of a multiple-input EX-NOR logic function is a logic '0' when the number of 1s in the input sequence is odd and a logic '1' when the number of 1s in the input sequence is even including zero. That is, an all 0s input sequence also produces a logic '1' at the output.

AllAbtEngg Android Application for Anna University, Polytechnic & School

1.40 Universal Gates

A universal gate is a gate which can implement any Boolean function without need to use any other gate type.

The NAND and NOR gates are universal gates.

In practice, this is advantageous since NAND and NOR gates are economical and easier to fabricate and are the basic gates used in all IC digital logic families.

In fact, an AND gate is typically implemented as a NAND gate followed by an inverter not the other way around!!

Likewise, an OR gate is typically implemented as a NOR gate followed by an inverter not the other way around!!

1.40.1 NAND Gate is a Universal Gate

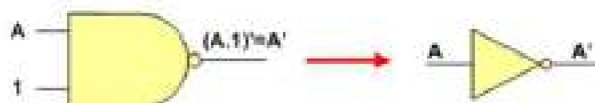
To prove that any Boolean function can be implemented using only NAND gates, we will show that the AND, OR, and NOT operations can be performed using only these gates.

Implementing an Inverter Using only NAND Gate

The figure shows two ways in which a NAND gate can be used as **an inverter (NOT gate)**.

1. All NAND input pins connect to the input signal **A** gives an output **A'**.

2. One NAND input pin is connected to the input signal **A** while all other input pins are connected to logic **1**. The output will be **A'**.



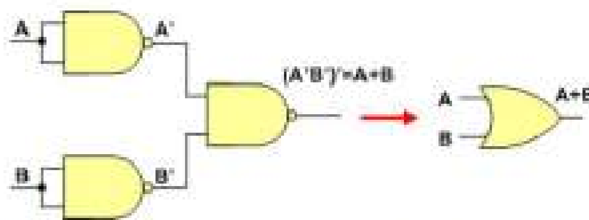
AllAbtEngg Android Application for Anna University, Polytechnic & School

Implementing AND Using only NAND Gates

An AND gate can be replaced by NAND gates as shown in the figure (The AND is replaced by a NAND gate with its output complemented by a NAND gate inverter).

Implementing OR Using only NAND Gates

An OR gate can be replaced by NAND gates as shown in the figure (The OR gate is replaced by a NAND gate with all its inputs complemented by NAND gate inverters).



Thus, the NAND gate is a universal gate since it can implement the AND, OR and NOT functions.

1.40.2 NOR Gate is a Universal Gate

To prove that any Boolean function can be implemented using only NOR gates, we will show that the AND, OR, and NOT operations can be performed using only these gates.

Implementing an Inverter Using only NOR Gate

The figure shows two ways in which a NOR gate can be used as **an inverter (NOT gate)**.

AllAbtEngg Android Application for Anna University, Polytechnic & School

1. All NOR input pins connect to the input signal **A** gives an output **A'**.

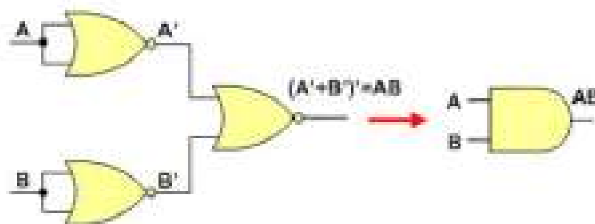
2. One NOR input pin is connected to the input signal **A** while all other input pins are connected to logic **0**. The output will be **A'**.

Implementing OR Using only NOR Gates

An OR gate can be replaced by NOR gates as shown in the figure (The OR is replaced by a NOR gate with its output complemented by a NOR gate inverter)

Implementing AND Using only NOR Gates

An AND gate can be replaced by NOR gates as shown in the figure (The AND gate is replaced by a NOR gate with all its inputs complemented by NOR gate inverters)

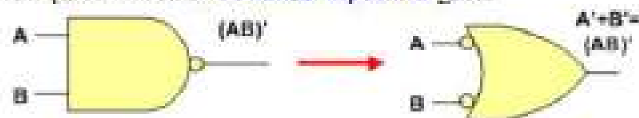


Thus, the NOR gate is a universal gate since it can implement the AND, OR and NOT functions.

1.41 Equivalent Gates

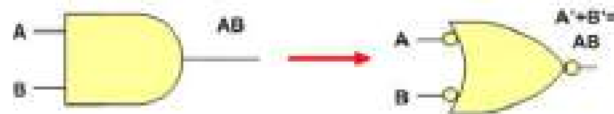
The shown figure summarizes important cases of gate equivalence. Note that bubbles indicate a complement operation (inverter).

A NAND gate is equivalent to an inverted-input OR gate.

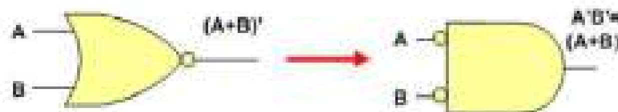


AllAbtEngg Android Application for Anna University, Polytechnic & School

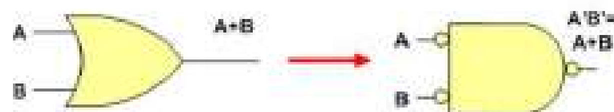
An AND gate is equivalent to an inverted-input NOR gate.



A NOR gate is equivalent to an inverted-input AND gate.



An OR gate is equivalent to an inverted-input NAND gate.



Two NOT gates in series are same as a buffer because they cancel each other as A'' = A.

1.42 Two-Level Implementations

We have seen before that Boolean functions in either SOP or POS forms can be implemented using 2-Level implementations.

For SOP forms AND gates will be in the first level and a single OR gate will be in the second level.

For POS forms OR gates will be in the first level and a single AND gate will be in the second level.

Note that using inverters to complement input variables is not counted as a level.

We will show that SOP forms can be implemented using only NAND gates, while POS forms can be implemented using only NOR gates.

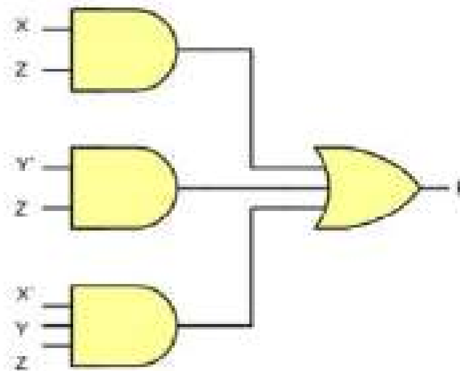
This is best explained through examples.

Example 1: Implement the following SOP function

$$F = XZ + Y'Z + X'YZ$$

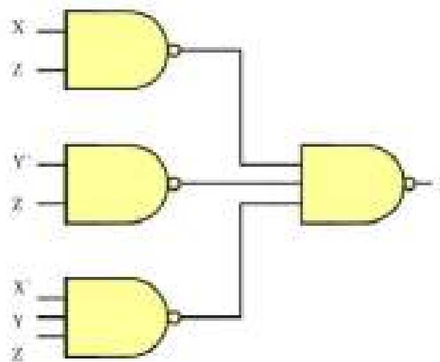
Being an SOP expression, it is implemented in 2-levels as shown in the figure.

AllAbtEngg Android Application for Anna University, Polytechnic & School



Introducing two successive inverters at the inputs of the OR gate results in the shown equivalent implementation. Since two successive inverters on the same line will not have an overall effect on the logic as it is shown before.

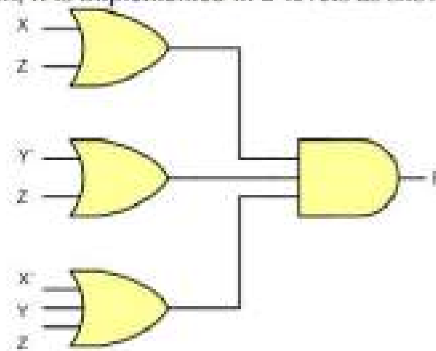
By associating one of the inverters with the output of the first level AND gate and the other with the input of the OR gate, it is clear that this implementation is reducible to 2-level implementation where both levels are NAND gates as shown in Figure.



Example 2: Implement the following POS function

$$F = (X+Z)\,(Y'+Z)\,(X'+Y+Z)$$

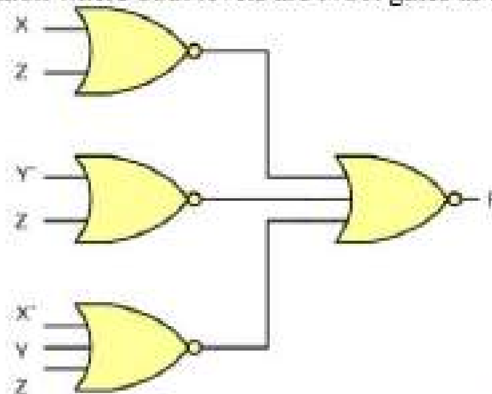
Being a POS expression, it is implemented in 2-levels as shown in the figure.



AllAbtEngg Android Application for Anna University, Polytechnic & School

Introducing two successive inverters at the inputs of the AND gate results in the shown equivalent implementation. Since two successive inverters on the same line will not have an overall effect on the logic as it is shown before.

By associating one of the inverters with the output of the first level OR gates and the other with the input of the AND gate, it is clear that this implementation is reducible to 2-level implementation where both levels are NOR gates as shown in Figure.



There are some other types of 2-level combinational circuits which are

- NAND-AND
- AND-NOR,
- NOR-OR,
- OR-NAND

These are explained by examples.

AllAbtEngg Android Application for Anna University, Polytechnic & School

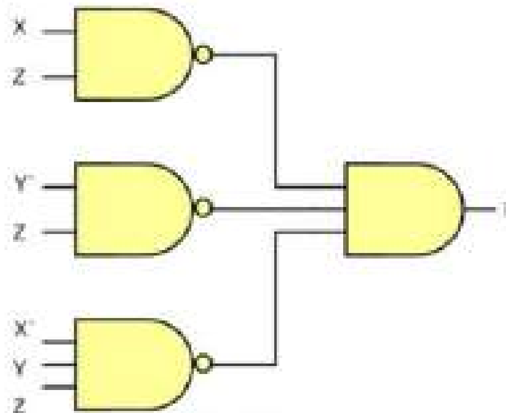
AND-NOR functions:

Example 3: Implement the following function

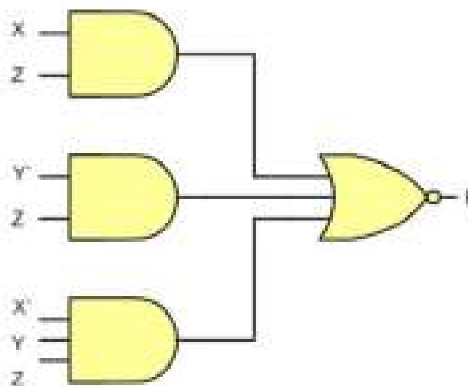
$F = \overline{XZ + \overline{Y}Z + \overline{X}YZ}$ **or**

$\overline{F} = XZ + \overline{Y}Z + \overline{X}YZ$

Since F' is in SOP form, it can be implemented by using NAND-NAND circuit. By complementing the output we can get F, or by using ***NAND-AND*** circuit as shown in the figure.



It can also be implemented using ***AND-NOR*** circuit as it is equivalent to NAND-AND circuit as shown in the figure.



OR-NAND functions:

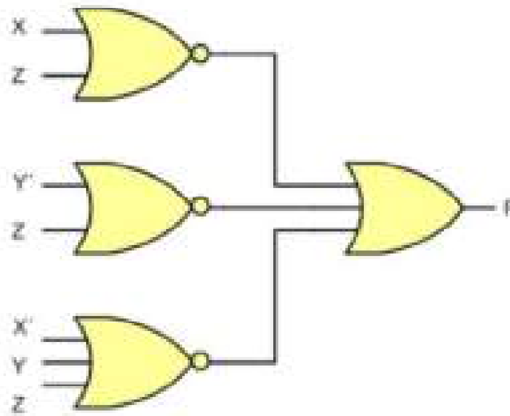
Example 4: Implement the following function

$F = \overline{(X+Z).(\overline{Y}+Z).(\overline{X}+Y+Z)}$ **or**

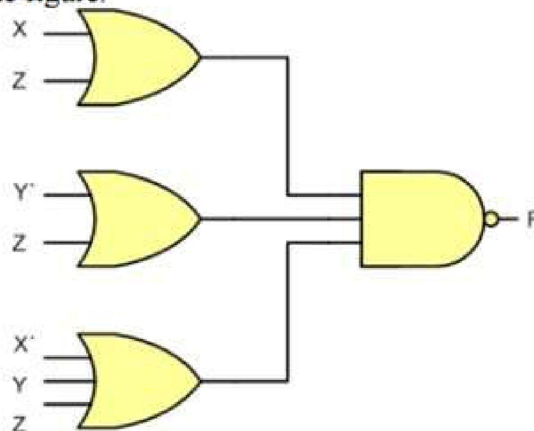
$\overline{F} = (X+Z)(\overline{Y}+Z)(\overline{X}+Y+Z)$

AllAbtEngg Android Application for Anna University, Polytechnic & School

Since F' is in POS form, it can be implemented by using NOR-NOR circuit.
By complementing the output we can get F, or by using ***NOR-OR*** circuit as shown in the figure.



It can also be implemented using ***OR-NAND*** circuit as it is equivalent to NOR-OR circuit as shown in the figure.



1.43 Two marks Questions and Answers

1. **Define Digital Systems.**

   A System which is processing discrete or digital signal is called as Digital System.
2. **What is meant by bit?**

   A Binary digit is called bit.
3. **What is the best example of digital system?**

   Digital computer is the best example of a digital system.
4. **Define Radix.**

   It specifies the number of symbols used for corresponding number system. .
5. **Define Nibble and Byte.**

AllAbtEngg Android Application for Anna University, Polytechnic & School