

Digital Systems

By

SUDHEER K

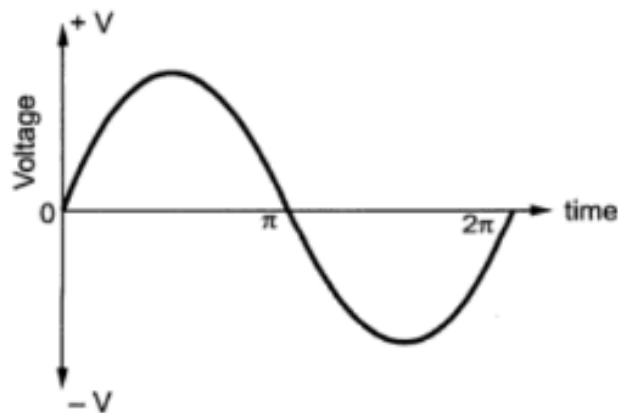
Digital system

Every day digital concepts are being applied to problems that could only be solved by analog methods several years ago. Fast and reliable solutions using digital techniques proved the tremendous power and usefulness of digital electronics. Now a days digital circuits are used in wide variety of industrial and consumer products such as automated industrial machinery, pocket calculators, microprocessors, computers, digital watches, TV games, signal processing and so on.

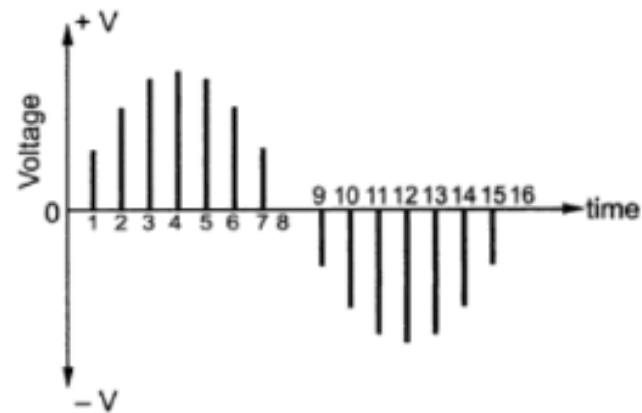
Digital vs Analog systems

A digital/discrete signal is a signal that can have one of a finite set of possible values at any time.

An analog/continuous signal is a signal that can have one of an infinite number of possible values.



(a) Continuous signal



(b) Discrete signal

Advantages of Digital System

Ease of Design : Digital systems are easy to design than analog systems, because digital design involves logic design, and logic design does not require special maths skill and its behaviour can be visualized part by part.

To visualize the behaviour of analog systems is somewhat difficult. It needs the special insights about the analog components such as capacitors, inductors and transistors used in the analog systems.

Reproducibility of Result : The output of analog systems vary with temperature, power-supply voltage, component aging, and other factors. So it is difficult to produce the same result every time with same set of inputs and circuit components. This is not the case with digital systems. They always produce exactly same results with same set of inputs and circuit components.

Flexibility : Digital systems are more flexible to design as its design involves set of logical steps and have various discrete and integrated options.

Functionality : It is easy to provide added functionality in digital systems than analog systems. For example, in digital systems it is easy to add password functionality. However, in analog systems it is a tough job.

Programmability : Nowadays, digital design is carried out by writing programs, in **hardware description language (HDL)**. These languages allow to simulate and test the performance of digital circuits. This feature is very useful in designing critical digital systems.

Speed : Today's digital devices are very fast and they can produce 500 million or more results as there operating speed is less than 2 nanoseconds.

Economy : Digital circuits can easily be integrated into single chip and can be produced in large quantity to have low cost.

Upgrading Technology : As digital technology is becoming more and more popular, more research is going on in this field. Therefore, more technological upgrade is expected in the digital world, and we know that after about every six months we hear about the launching of a new processor, new memory technology.

Number Systems

Number system is a basis for counting various items. On hearing the word 'number', all of us immediately think of the familiar decimal number system with its 10 digits : 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9.

Modern computers communicate and operate with binary numbers which use only the digits 0 and 1. Let us consider decimal number 18. This number is represented in binary as 10010.

In this chapter, we discuss binary, octal, hexadecimal, and BCD number systems, and we will see how to convert from decimal to binary, octal and hexadecimal, and vice versa. In the later section of this chapter we are going to see binary, hexadecimal, Excess-3 and BCD arithmetic.

Decimal Number System

Before considering any number system, let us consider familiar decimal number system. In decimal number system we can express any decimal number in units, tens, hundreds, thousands and so on. When we write a decimal number say, 5678.9, we know it can be represented as

$$5000 + 600 + 70 + 8 + 0.9 = 5678.9$$

The decimal number 5678.9 can also be written as 5678.9_{10} , where the 10 subscript indicates the radix or base.

In power of 10, we can write as

The diagram illustrates the expansion of the decimal number 5678.9 into its place value components using powers of 10. The number 5678.9 is shown at the bottom, with lines connecting each digit to its corresponding term in the sum above:

- The digit 5 is connected to 5×10^3
- The digit 6 is connected to 6×10^2
- The digit 7 is connected to 7×10^1
- The digit 8 is connected to 8×10^0
- The digit 9 is connected to 9×10^{-1}

$$\boxed{5 \times 10^3} + \boxed{6 \times 10^2} + \boxed{7 \times 10^1} + \boxed{8 \times 10^0} + \boxed{9 \times 10^{-1}}$$

5 6 7 8 . 9

This says that, the position of a digit with reference to the decimal point determines its value/weight. The sum of all the digits multiplied by their weights gives the total number being represented. The leftmost digit, which has the greatest weight is called the **most significant digit** and the rightmost digit, which has the least weight, is called the **least significant digit**. Fig. 1.2 shows decimal digit and its weights expressed as a power of 10.

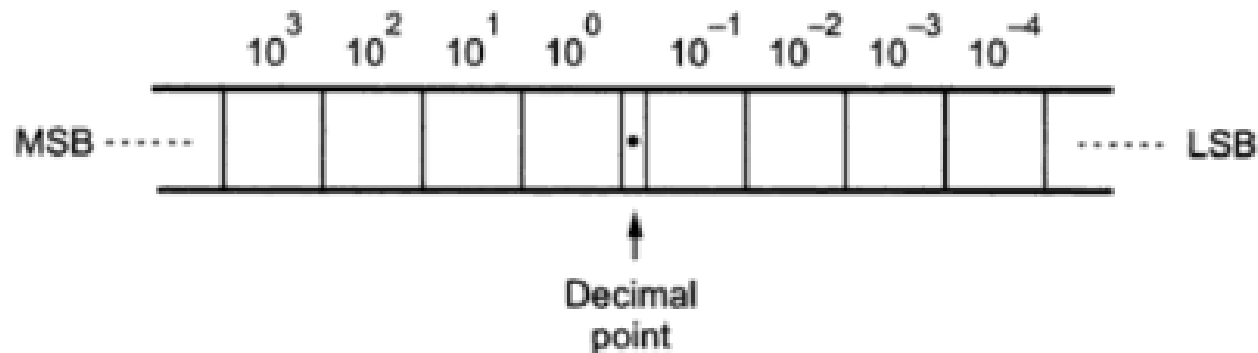


Fig. 1.2 Decimal position values as powers of 10

➡ **Example 1.1** : Represent decimal number 98.72 in power of 10.

Solution : $N = 9 \times 10^1 + 8 \times 10^0 + 7 \times 10^{-1} + 2 \times 10^{-2}$

The digit 9 has a weight of 10, the digit 8 has a weight of 1, the digit 7 has a weight of $1/10$ and the digit 2 has a weight of $1/100$.

Binary Number System

We know that decimal system with its ten digits is a base-ten system. Similarly, binary system with its two digits is a base-two system. The two binary digits (bits) are 1 and 0. Like digital system, in binary system each binary digit commonly known as bit, has its own value or weight. However in binary system weight is expressed as a power of 2, as shown in Fig. 1.3.

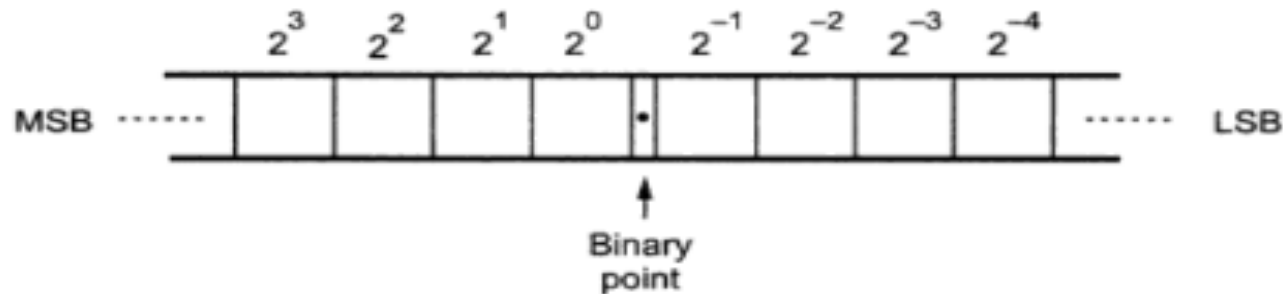


Fig. 1.3 Binary position values as a power of 2

➡ **Example 1.2 :** Represent binary number 1101.101 in power of 2 and find its decimal equivalent.

Solution : Representing given binary number in power of 2 we have,

$$\begin{aligned} N &= 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} \\ &\quad + 0 \times 2^{-2} + 1 \times 2^{-3} \\ &= 8 + 4 + 0 + 1 + 0.5 + 0 + 0.125 \\ &= 13.625_{10} \end{aligned}$$

Octal Number System

We know that the base of the decimal number system is 10 because it uses the digits 0 to 9, and the base of binary number system is 2 because it uses digits 0 and 1. The octal number system uses first eight digits of decimal number system : 0, 1, 2, 3, 4, 5, 6, and 7. As it uses 8 digits, its base is 8.

►►► **Example 1.3** : Represent octal number 567 in power of 8 and find its decimal equivalent.

Solution : The given octal number 567 can be represented in power of 8 as

$$\begin{array}{c} \boxed{5 \times 8^2} + \boxed{6 \times 8^1} + \boxed{7 \times 8^0} \\ \swarrow \quad \downarrow \quad \searrow \\ \quad 5 \quad 6 \quad 7 \\ = 5 \times 64 + 6 \times 8 + 7 \times 1 \\ = 320 + 48 + 7 \\ = 375_{10} \end{array}$$

HexaDecimal Number System

The hexadecimal number system has a base of 16 having 16 digits : 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E and F. It is another number system that is particularly useful for human communications with a computer. Although it is somewhat more difficult to interpret than the octal number system, it has become the most popular. Since its base is a power of 2 (2^4), it is easy to convert hexadecimal numbers to binary and vice versa.

Table 1.1 shows the relationship between decimal, binary and hexadecimal. Note that each hexadecimal digit represents a group of four binary digits, called nibbles, that are fundamental parts of larger binary words.

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

Table 1.1 Relation between decimal, binary and hexadecimal numbers

➡ **Example 1.4** : Represent hexadecimal number 3FD in power of 16 and find its decimal equivalent.

Solution : The given hexadecimal number $3FD_{16}$ can be represented in power of 16.

$$\begin{array}{c} \boxed{3 \times 16^2} + \boxed{F \times 16^1} + \boxed{D \times 16^0} \\ \diagdown \quad | \quad \diagup \\ \quad 3 \quad F \quad D \end{array}$$

$$= 3 \times 256 + F(15) \times 16 + D(13) \times 1$$

$$= 768 + 240 + 13$$

$$= 1021_{10}$$

Counting in Radix(r)

In previous sections we have seen number systems with radix (base) r equal to 10, 2, 8 and 16. Each number system has r set of characters. For example, in decimal number system r equals to 10 has 10 characters from 0 to 9, in binary number system r equals to 2 has 2 characters 0 and 1 and so on. In general we can say that, a number represented in radix r , has r characters in its set and r can be any value. This is illustrated in Table. 1.2.

Radix (Base) r	Characters in set
2 (Binary)	0, 1
3	0, 1, 2
4	0, 1, 2, 3
:	
:	
7	0, 1, 2, 3, 4, 5, 6
8 (Octal)	0, 1, 2, 3, 4, 5, 6, 7
:	
:	
10 (Decimal)	0, 1, 2, 3, 4, 5, 6, 7, 8, 9
:	
:	
16 (Hexadecimal)	0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

Table 1.2 Radix and character set

⇒ **Example 1.5** : Find the decimal equivalent of $(231.23)_4$

Solution :

$$\begin{aligned} N &= 2 \times 4^2 + 3 \times 4^1 + 1 \times 4^0 + 2 \times 4^{-1} + 3 \times 4^{-2} \\ &= 32 + 12 + 1 + 0.5 + 0.1875 \\ &= 45.6875_{10} \end{aligned}$$

⇒ **Example 1.6** : Count from 0 to 9 in radix 5.

Solution : The Table 1.2 indicates that radix 5 has 5 characters. A count sequence from 0 decimal to 9 decimal is

00, 01, 02, 03, 04, 10, 11, 12, 13, 14.

Number Conversion

The human beings use decimal number system while computer uses binary number system. Therefore, it is necessary to convert decimal number into its equivalent binary while feeding number into the computer and to convert binary number into its decimal equivalent while displaying result of operation to the human beings. However, dealing with a large quantity of binary numbers of many bits is inconvenient for human beings. Therefore, octal and hexadecimal numbers are used as a shorthand means of expressing large binary numbers. But it is necessary to keep in mind that the digital circuits and systems work strictly in binary; we are using octal and hexadecimal only as a convenience for the operators of the system.

Before going to see conversions between binary, octal and hexadecimal numbers we see the number of digits in several number systems. Table 1.3 shows the decimal, binary, octal and hexadecimal numbers.

Decimal	Binary	Octal	Hexadecimal
0	0 0 0 0	0	0
1	0 0 0 1	1	1
2	0 0 1 0	2	2
3	0 0 1 1	3	3
4	0 1 0 0	4	4
5	0 1 0 1	5	5
6	0 1 1 0	6	6
7	0 1 1 1	7	7
8	1 0 0 0	10	8
9	1 0 0 1	11	9
10	1 0 1 0	12	A
11	1 0 1 1	13	B
12	1 1 0 0	14	C
13	1 1 0 1	15	D
14	1 1 1 0	16	E
15	1 1 1 1	17	F

Table 1.3 Decimal, binary, octal and hexadecimal numbers

Binary to Octal conversion

We know that base for octal numbers is 8 and the base for binary numbers is 2. The base for octal number is the third power of the base for binary numbers. Therefore, by grouping 3 digits of binary numbers and then converting each group digit to its octal equivalent we can convert binary number to its octal equivalent.

➡ **Example 1.7** : Convert $(1\ 1\ 1\ 1\ 0\ 1\ 1\ 0\ 0)_2$ to octal equivalent.

Solution :

111	101	100
7	5	4

∴

Octal number = $(754)_8$

Octal to Binary conversion

Conversion from octal to binary is a reversal of the process explained in the previous section. Each digit of the octal number is individually converted to its binary equivalent to get octal to binary conversion of the number.

⇒ **Example 1.8** : Convert $(634)_8$ to binary.

Solution :

6	3	4
110	011	100

∴ Binary number = 110 011 100

⇒ **Example 1.9** : Convert $(725.63)_8$ to binary.

Solution :

7	2	5	.	6	3
111	010	101	.	110	011

∴ Binary number = 111010101 . 110 011

Binary to Hexadecimal conversion

We know that base for hexadecimal numbers is 16 and the base for binary numbers is 2. The base for hexadecimal number is the fourth power of the base for binary numbers. Therefore, by grouping 4 digits of binary numbers and then converting each group digit to its hexadecimal equivalent we can convert binary number to its hexadecimal equivalent.

➡ **Example 1.10** : Convert $(11011000\ 1001\ 1011)_2$ to hexadecimal equivalent.

Solution :

1101	1000	1001	1011
D	8	9	B

∴

Hexadecimal number = $(D89B)_H$

Hexadecimal to Binary conversion

Conversion from hexadecimal to binary is a reversal of the process explained in the previous section. Each digit of the hexadecimal number is individually converted to its binary equivalent to get hexadecimal to binary conversion of the number.

⇒ **Example 1.11** : Convert $(3FD)_H$ to binary.

Solution :

3	F	D
0011	1111	1101

∴ Binary number = **0011 1111 1101**

⇒ **Example 1.12** : Convert $(5A9.B4)_H$ to binary.

Solution :

5	A	9	.	B	4
0101	1010	1001	.	1011	0100

∴ Binary number = **0101 1010 1001 . 1011 0100**

The easiest way to convert octal number to hexadecimal number is given below.

1. Convert octal number to its binary equivalent.
2. Convert binary number to its hexadecimal equivalent.

⇒ **Example 1.13** : Convert $(615)_8$ to its hexadecimal equivalent.

Solution :

Step 1 : Octal to binary

6	1	5
110	001	101

∴ Binary number = 110001101

Step 2 : Binary to hexadecimal

0001	1000	1101
1	8	D

∴ Hexadecimal number = $18D_H$

Hexadecimal to Octal conversion

The easiest way to convert hexadecimal number to octal number is given below.

1. Convert hexadecimal number to its binary equivalent.
2. Convert binary number to its octal equivalent.

⇒ **Example 1.14** : Convert $(25B)_H$ to its octal equivalent.

Solution :

Step 1 : Hexadecimal to binary

2	5	B
0010	0101	1011

∴ Binary number = 0010 0101 1011

Step 2 : Binary to octal

001	001	011	011
1	1	3	3

∴ Octal number = 1133_8

Radix to decimal number

➡ **Example 1.15** : Convert binary number 1101.1 to its decimal equivalent.

Solution :

$$\begin{aligned} N &= 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} \\ &= 8 + 4 + 0 + 1 + 0.5 \\ &= 13.5_{10} \end{aligned}$$

OR

Step 1 :	1	1	0	1	1
Step 2 :	8	4	2	1	0.5
Step 3 :	8	+ 4	+ 0	+ 1	+ 0.5 = 13.5 ₁₀

➡ **Example 1.16** : Convert $(3102.12)_4$ to its decimal equivalent.

Solution :

$$\begin{aligned} N &= 3 \times 4^3 + 1 \times 4^2 + 0 \times 4^1 + 2 \times 4^0 + 1 \times 4^{-1} + 2 \times 4^{-2} \\ &= 192 + 16 + 0 + 2 + 0.25 + 0.125 = 210.375_{10} \end{aligned}$$

Decimal to any radix conversion

➡ **Example 1.17** : Convert decimal number 37 to its binary equivalent.

Solution : Here r is 2

Divide 37 by 2	$\begin{array}{r} 18 \\ 2 \overline{) 37} \\ - 36 \\ \hline 1 \end{array}$	R	1		LSD
Divide 18 by 2	$\begin{array}{r} 9 \\ 2 \overline{) 18} \\ - 18 \\ \hline 0 \end{array}$	R	0		
Divide 9 by 2	$\begin{array}{r} 4 \\ 2 \overline{) 9} \\ - 8 \\ \hline 1 \end{array}$	R	1		
Divide 4 by 2	$\begin{array}{r} 2 \\ 2 \overline{) 4} \\ - 4 \\ \hline 0 \end{array}$	R	0	≡	
Divide 2 by 2	$\begin{array}{r} 1 \\ 2 \overline{) 2} \\ - 2 \\ \hline 0 \end{array}$	R	0		
Divide 1 by 2	$\begin{array}{r} 0 \\ 2 \overline{) 1} \\ - 0 \\ \hline 1 \end{array}$	R	1		MSD

	Q	R			
2	37	1		↑	LSD
2	18	0			
2	9	1			
2	4	0			
2	2	0			
2	1	1			
	0				

Note : Q : Quotient
R : Remainder

Binary equivalent = **100101₂**

➡ **Example 1.18** : Convert decimal number 214 to its octal equivalent.

Solution : Here r is 8

Divide 214 by 8	$\begin{array}{r} 26 \\ 8 \overline{) 214} \\ \underline{- 208} \\ 6 \end{array}$	R			
		6	→	6	LSD
Divide 26 by 8	$\begin{array}{r} 3 \\ 8 \overline{) 26} \\ \underline{- 24} \\ 2 \end{array}$			2	
		2	→	2	
Divide 3 by 8	$\begin{array}{r} 0 \\ 8 \overline{) 3} \\ \underline{- 0} \\ 3 \end{array}$			3	MSD

≡

		Q	R	
8		214		6

8		26		2

8		3		3

		0		

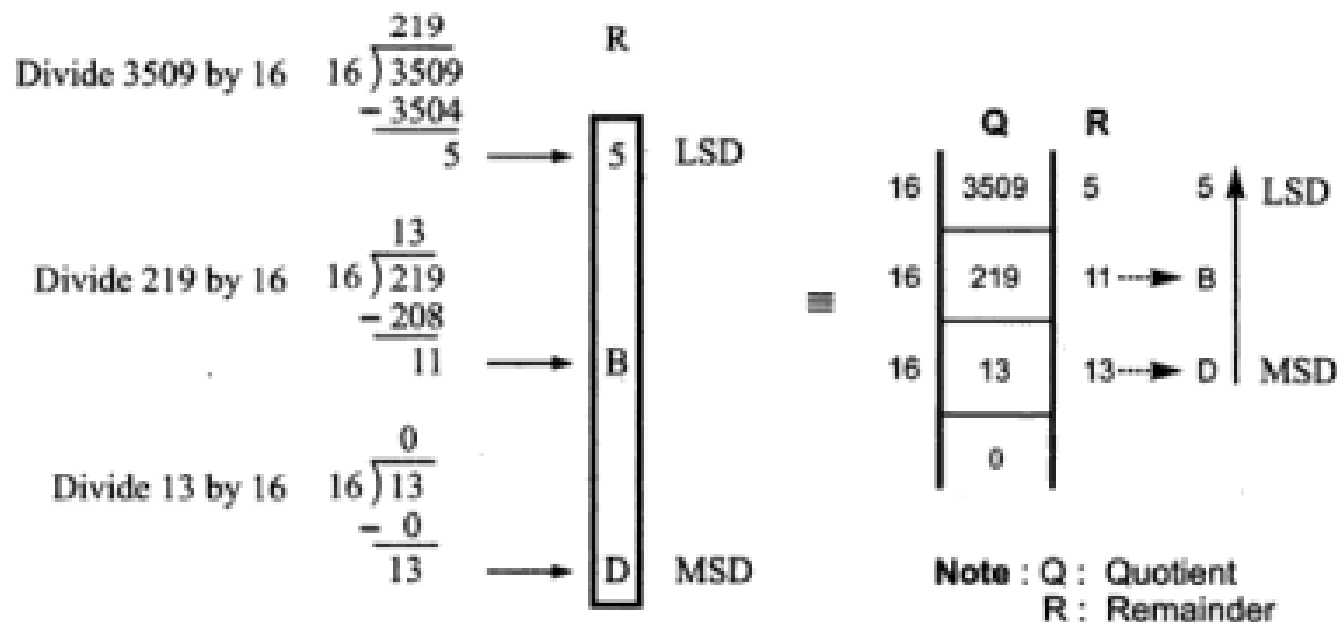
↑ LSD
MSD

Note : Q : Quotient
R : Remainder

The conversion is over when quotient is 0, and we get 326_8 as octal equivalent to decimal number 214.

➡ **Example 1.19** : Convert decimal number 3509 to its hexadecimal equivalent.

Solution : Here r is 16



The conversion is over when quotient is 0, and we get $DB5_{16}$ as hexadecimal equivalent to decimal number 3509.

➡ **Example 1.20** : Convert 54_{10} to radix 4.

Solution : Here r is 4

Divide 54 by 4	$\begin{array}{r} 13 \\ 4 \overline{) 54} \\ \underline{- 52} \\ 2 \end{array}$	R			
	2	→	2	LSD	
Divide 13 by 4	$\begin{array}{r} 3 \\ 4 \overline{) 13} \\ \underline{- 12} \\ 1 \end{array}$		1	≡	
Divide 3 by 4	$\begin{array}{r} 0 \\ 4 \overline{) 3} \\ \underline{- 0} \\ 3 \end{array}$		3	MSD	

	Q	R	
4	54	2	↑ LSD
4	13	1	
4	3	3	↑ MSD
	0		

Note : Q : Quotient
 R : Remainder

The conversion is over when quotient is 0, and we get $(312)_4$ as radix 4 equivalent to decimal number 54.

Decimal fraction to binary conversion

➡ **Example 1.21 :** *Convert 0.8125 decimal number to its binary equivalent.*

Solution :

Fraction	Radix	Result	Recorded carries	
0.8125	$\times 2$	$= 1.625 = 0.625$	with a carry of 1	MSD
0.625	$\times 2$	$= 1.25 = 0.25$	with a carry of 1	↓
0.25	$\times 2$	$= 0.5 = 0.5$	with a carry of 0	↓
0.5	$\times 2$	$= 1.0 = 0.0$	with a carry of 1	LSD

Reading carries downward we get,

Binary fraction = 0.1101, which is equivalent to 0.8125 decimal.

➡ **Example 1.22** : Convert 0.95 decimal number to its binary equivalent.

Solution :

Fraction	Radix	Result	Recorded carries	
0.95	× 2	= 1.9 = 0.9	with a carry of 1	MSD ↓ LSD
0.9	× 2	= 1.8 = 0.8	with a carry of 1	
0.8	× 2	= 1.6 = 0.6	with a carry of 1	
0.6	× 2	= 1.2 = 0.2	with a carry of 1	
0.2	× 2	= 0.4 = 0.4	with a carry of 0	
0.4	× 2	= 0.8 = 0.8	with a carry of 0	
0.8	× 2	= 1.6 = 0.6	with a carry of 1	

In this case, 0.8 is repeated and if we multiply further, we will get repeated sequence. If we stop here, we get 7 binary digits, 0.1111001. This answer is an approximate answer. To get more accurate answer we have to continue multiplying by 2 until we have as many digits as necessary for our application.

➡ **Example 1.23 :** Convert 0.640625 decimal number to its octal equivalent.

Solution :

Fraction	Radix	Result	Recorded carries
0.640625	× 8	= 5.125	= 0.125 with a carry of 5
0.125	× 8	= 1.0	= 0 with a carry of 1

MSD
↓
LSD

Reading carries downward we get octal fraction = 0.51, which is equivalent to 0.640625 decimal.

➡ **Example 1.24 :** Convert 0.1289062 decimal number to its hex equivalent.

Solution :

Fraction	Radix	Result	Recorded carries
0.1289062	× 16	= 2.0625	= 0.0625 with carry of 2
0.0625	× 16	= 1.0	= 0 with carry of 1

MSD
↓
LSD

Reading carries downward we get hexadecimal fraction = 0.21₁₆, which is equivalent to 0.1289062 decimal.

Complements

In this section, we see the complement numbers used in different number systems. These complement numbers are used in case of subtraction of two numbers in corresponding number systems.

1's Complement Representation

The 1's complement of a binary number is the number that results when we change all 1's to zeros and the zeros to ones.

➡ **Example 1.25 :** Find 1's complement of $(1\ 1\ 0\ 1)_2$.

Solution :

1	1	0	1	← number
0	0	1	0	← 1's complement

➡ **Example 1.26 :** Find 1's complement of $1\ 0\ 1\ 1\ 1\ 0\ 0\ 1$.

Solution :

1	0	1	1	1	0	0	1	number
0	1	0	0	0	1	1	0	1's complement

2's Complement Representation

The 2's complement is the binary number that results when we add 1 to the 1's complement. It is given as

$$\text{2's complement} = \text{1's complement} + 1$$

The 2's complement form is used to represent negative numbers.

➡ **Example 1.27 :** Find 2's complement of $(1\ 0\ 0\ 1)_2$.

Solution :	1 0 0 1	number
	0 1 1 0	1's complement
	+ 1	
	<hr/>	
	0 1 1 1	2's complement

➡ **Example 1.28 :** Find 2's complement of $(1\ 0\ 1\ 0\ 0\ 0\ 1\ 1)_2$.

Solution :	1 0 1 0	0 0 1 1	number
	0 1 0 1	1 1 0 0	1's complement
	+ 1		
	<hr/>		
	0 1 0 1	1 1 0 1	2's complement

Boolean Algebra and Logic Gates

2.1 Basic Definitions

In 1854, George Boole introduced a systematic treatment of logic and developed for this purpose an algebraic system now called Boolean Algebra. In 1938 C.E. Shannon introduced a two-valued Boolean algebra called switching algebra. Boolean algebra is a system of mathematical logic. It differs from both ordinary algebra and the binary number system. As an illustration, in Boolean, $1 + 1 = 1$, in binary arithmetic the result is 10. Thus, although there are similarities, Boolean algebra is a unique system.

The Boolean algebra is defined with a set of elements, a set of operators and number of rules, laws, theorems and postulates. The postulates of a mathematical system from the basic assumption from which it is possible to deduce the rules, law, theorems, and properties of the system. Boolean algebra is formulated by a defined set of elements, together with two binary operators, + and \cdot . Let us see the definition of postulates used to formulate the Boolean algebra.

1. Set : A set of element is a group of objects having a common property. If S is a set and X and Y are certain objects, then $X \in S$ denotes that X is an element of set S and $Y \notin S$ denotes that Y is not an element of set S.

2. Closure : A particular set is closed with respect to a binary operator if, a every pair of elements of a set obtains a unique element of the same set after being operated by the operator.

3. Associative law : A binary operator * on a set S is said to be associative whenever $(A * B) * C = A * (B * C)$ for all A, B, C \in S .

4. Commutative law : A binary operator * on a set S is said to be commutative whenever.

$$a * b = b * a \text{ for all } a, b \in S$$

5. Identity element : A set S is said to have an identity element with respect to binary operator * on S if there exists an element $e \in S$ with the property.

$$a * e = e * a = a \text{ for every } a \in S.$$

For example : The element 0 is an identity with respect to binary operator + since

$$a + 0 = 0 + a = a.$$

6. **Distributive law** : If Δ and $*$ are two operators on a set S , Δ is said to be distributive over $*$ whenever,

$$A \Delta (B * C) = (A \Delta B) * (A \Delta C)$$

2.2 Axiomatic Definition of Boolean Algebra

Boolean algebra is formulated by a defined set of elements, together with two binary operators, $+$ and \cdot ; provided that the following axioms or postulates are satisfied.

- **Closure (a)** : Closure with respect to the operator $+$

When two binary elements are operated by operator $+$ the result is a unique binary element.

- **Closure (b)** : Closure with respect to the operator \cdot (dot).

When two binary elements are operated by operator \cdot (dot), the result is a unique binary element.

- An identity element with respect to $+$, designated by

$$0 : A + 0 = 0 + A = A$$

- An identity element with respect to \cdot , designated by 1 : $A \cdot 1 = 1 \cdot A = A$

- Commutative with respect to $+$: $A + B = B + A$

- Commutative with respect to \cdot : $A \cdot B = B \cdot A$

- Distributive property of \cdot over $+$:

$$A \cdot (B + C) = (A \cdot B) + (A \cdot C)$$

- Distributive property of $+$ over \cdot :

$$A + (B \cdot C) = (A + B) \cdot (A + C)$$

- For every binary element, there exists complement element. For example, if A is an element, we have \bar{A} is a complement of A . i.e. if $A = 0$, $\bar{A} = 1$ and if

$$A = 1, \bar{A} = 0.$$

- There exists at least two elements, say A and B in the set of binary elements such that $A \neq B$.

2.3 Basic Rules, Laws, Theorems and Properties of Boolean Algebra

2.3.1 Rules in Boolean Algebra

1. The symbol which represent an arbitrary elements of an Boolean algebra is known as **variable**. Any single variable or a function of several variables can have either a 1 or 0 value. For example, in expression $Y = A + BC$, variables A , B and C can

- have either a 1 or 0 value, and function Y also can have either a 1 or 0 value; however its value depends on the value of Boolean expression.
- A complement of a variable is represented by a "bar" over the letter. For example, the complement of a variable A will be denoted by \bar{A} . So if $A = 1$, $\bar{A} = 0$ and if $A = 0$, $\bar{A} = 1$. Sometimes a prime symbol ($'$) is used to denote the complement. For example, the complement of A can be written as A' .
 - The logical AND operator of two variables is represented either by writing a dot (\cdot) between two variables, such as $A \cdot B$ or by simply writing two variables, such as AB . Similarly, AND operation between three variables can be represented as $A \cdot B \cdot C$ or ABC .
 - The logical OR operator of two variables is represented by writing a '+' sign between the two variables such as $A + B$. Similarly, OR operation between three variables can be represented as $A + B + C$.
 - The logical OR operator in the Boolean algebra with variables having value either a 0 or a 1 gives following results.

$$0 + 0 = 0$$

$$1 + 0 = 1$$

$$0 + 1 = 1$$

$$1 + 1 = 1$$

From the above results following rules are defined in the Boolean algebra.

Rule 1 :

$$\begin{array}{l} 0 + 0 = 0 \\ 0 + 1 = 1 \end{array} \Rightarrow 0 + A = A \text{ or } A + 0 = A$$

Rule 2 :

$$\begin{array}{l} 1 + 0 = 1 \\ 1 + 1 = 1 \end{array} \Rightarrow 1 + A = 1 \text{ or } A + 1 = 1$$

Rule 3 :

$$\begin{array}{l} 0 + 0 = 0 \\ 1 + 1 = 1 \end{array} \Rightarrow A + A = A$$

Rule 4 :

$$\begin{array}{|c|} \hline 0 \\ \hline 1 \\ \hline \end{array} + \begin{array}{|c|} \hline 1 \\ \hline 0 \\ \hline \end{array} = 1$$

$$\Rightarrow A + \bar{A} = 1 \text{ or } \bar{A} + A = 1$$

6. The logical AND operator in the Boolean algebra with variables having value either a 0 or a 1 gives following results.

$$\begin{array}{l} 0 \cdot 0 = 0 \quad 1 \cdot 0 = 0 \\ 0 \cdot 1 = 0 \quad 1 \cdot 1 = 1 \end{array}$$

From the above result following rules are defined in the Boolean algebra.

Rule 5 :

$$\begin{array}{|c|} \hline 0 \\ \hline 0 \\ \hline \end{array} = 0$$

$$\Rightarrow 0 \cdot A = 0 \text{ or } A \cdot 0 = 0$$

$$\begin{array}{|c|} \hline 0 \\ \hline 1 \\ \hline \end{array} = 0$$

Rule 6 :

$$\begin{array}{|c|} \hline 0 \\ \hline 0 \\ \hline \end{array} = \begin{array}{|c|} \hline 0 \\ \hline 0 \\ \hline \end{array}$$

$$\Rightarrow 1 \cdot A = A \text{ or } A \cdot 1 = A$$

$$\begin{array}{|c|} \hline 1 \\ \hline 1 \\ \hline \end{array} = \begin{array}{|c|} \hline 1 \\ \hline 1 \\ \hline \end{array}$$

Rule 7 :

$$\begin{array}{|c|} \hline 0 \\ \hline 0 \\ \hline \end{array} = \begin{array}{|c|} \hline 0 \\ \hline 0 \\ \hline \end{array}$$

$$\Rightarrow A \cdot A = A$$

$$\begin{array}{|c|} \hline 1 \\ \hline 1 \\ \hline \end{array} = \begin{array}{|c|} \hline 1 \\ \hline 1 \\ \hline \end{array}$$

Rule 8 :

$$\begin{array}{|c|} \hline 0 \\ \hline 1 \\ \hline \end{array} \cdot \begin{array}{|c|} \hline 1 \\ \hline 0 \\ \hline \end{array} = 0$$

$$\Rightarrow A \cdot \bar{A} = 0 \text{ or } \bar{A} \cdot A = 0$$

7. The NOT operator in the Boolean algebra with variable having value either a 0 or a 1 gives following results.

$$\begin{array}{l} \bar{0} = 1 \quad \bar{0} = 0 \\ \bar{1} = 0 \quad \bar{1} = 1 \end{array}$$

From the previous result following rule is defined in Boolean algebra

Rule 9 :

$$\overline{\overline{0}} = 0$$

$$\overline{\overline{1}} = 1$$

$$\overline{\overline{A}} = A$$

2.3.2 Laws of Boolean Algebra

Three of the basic laws of Boolean algebra are the same as in ordinary algebra: the commutative laws, associative laws, and the distributive law.

Commutative Laws

LAW 1 : $A + B = B + A$: This states that the order in which the variables are ORed makes no difference in the output. The truth tables are identical. Therefore, A OR B is same as B OR A.

A	B	A + B
0	0	0
0	1	1
1	0	1
1	1	1

=

B	A	B + A
0	0	0
0	1	1
1	0	1
1	1	1

Table 2.1 Truth table for commutative law for OR gates

LAW 2 : $AB = BA$: The commutative law of multiplication states that the order in which the variables are ANDed makes no difference in the output. The truth tables are identical. Therefore, A AND B is same as B AND A.

A	B	AB
0	0	0
0	1	0
1	0	0
1	1	1

=

B	A	BA
0	0	0
0	1	0
1	0	0
1	1	1

Table 2.2 Truth table for commutative law for AND gates

It is important to note that the commutative laws can be extended to any number of variables. For example, since $A + B = B + A$, it follows that $A + B + C = B + A + C$, and since $A + C = C + A$, it is true that $B + A + C = B + C + A$. Similarly, $ABCD = BACD = BADC = ABDC$, and so on.

Associative Laws

LAW 1 : $A + (B + C) = (A + B) + C$: This law states that in the ORing of several variables, the result is the same regardless of the grouping of the variables. For three variables, A OR B ORed with C is the same as A ORed with B OR C.

A	B	C	A + B	(A + B) + C	A	B	C	B + C	A + (B + C)
0	0	0	0	0	0	0	0	0	0
0	0	1	0	1	0	0	1	1	1
0	1	0	1	1	0	1	0	1	1
0	1	1	1	1	0	1	1	1	1
1	0	0	1	1	1	0	0	0	1
1	0	1	1	1	1	0	1	1	1
1	1	0	1	1	1	1	0	1	1
1	1	1	1	1	1	1	1	1	1

Table 2.3 Truth tables for associative law for OR gates

LAW 2 : $(AB) C = A (BC)$: The associative law of multiplication states that it makes no difference in what order the variables are grouped when ANDing several variables. For three variables, A AND B ANDed with C is the same as A ANDed with B and C.

A	B	C	AB	(AB) C	A	B	C	BC	A (BC)
0	0	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	1	0	0
0	1	0	0	0	0	1	0	0	0
0	1	1	0	0	0	1	1	1	0
1	0	0	0	0	1	0	0	0	0
1	0	1	0	0	1	0	1	0	0
1	1	0	1	0	1	1	0	0	0
1	1	1	1	1	1	1	1	1	1

Table 2.4 Truth table for associative law for AND gates

Distributive Law

LAW : $A (B + C) = AB + AC$: The distributive law states that ORing several variables and ANDing the result with a single variable is equivalent to ANDing the result with a single variable with each of the several variables and then ORing the products.

A	B	C	(B + C)	A(B + C)	=	A	B	C	AB	AC	AB + AC
0	0	0	0	0		0	0	0	0	0	0
0	0	1	1	0		0	0	1	0	0	0
0	1	0	1	0		0	1	0	0	0	0
0	1	1	1	0		0	1	1	0	0	0
1	0	0	0	0		1	0	0	0	0	0
1	0	1	1	1		1	0	1	0	1	1
1	1	0	1	1		1	1	0	1	0	1
1	1	1	1	1		1	1	1	1	1	1

Table 2.5 Truth table for distributive law

It is important to note that the distributive property is often used in reverse; i.e., given $AB + AC$, we replace it by its equivalent, $A(B + C)$. As in ordinary algebra, this process is called **factoring**. We factored A out of the expression $AB + AC$.

2.3.3 Additional Rules in Boolean Algebra

For manipulation and simplification of Boolean algebra three more rules are defined in Boolean algebra. These are :

- $A + AB = A$
- $A + \bar{A}B = A + B$
- $(A + B)(A + C) = A + BC$

These rules are derived from the basic rules and laws of the Boolean algebra.

Rule 10 : $A + AB = A$
Proof : $A + AB = A(1 + B)$ Distributive law
 $= A(1)$ Rule 2 : $(1 + B) = 1$
 $= A$ Rule 6 : $(A \cdot 1) = A$

Rule 11 : $A + \bar{A}B = A + B$
Proof : $A + \bar{A}B = A + AB + \bar{A}B$ Rule 10 : $A + AB = A$
 $= A + B(A + \bar{A})$ Distributive law
 $= A + B(1)$ Rule 4 : $A + \bar{A} = 1$
 $= A + B$ Rule 6 : $(B \cdot 1) = B$

Rule 12 : $(A + B)(A + C) = A + BC$
Proof : $(A + B)(A + C) = AA + AC + AB + BC$ Distributive law
 $= A + AC + AB + BC$ Rule 7 : $A \cdot A = A$
 $= A(1 + C + B) + BC$ Distributive law
 $= A + BC$ Rule 2 : $1 + A = 1$

2.3.4 Theorems in Boolean Algebra

DeMorgan's Theorems

DeMorgan suggested two theorems that form an important part of Boolean algebra. In the equation form, they are :

$$1) \overline{AB} = \overline{A} + \overline{B}$$

The complement of a product is equal to the sum of the complements. This is illustrated by truth Table 2.6.

Truth Table :

A	B	\overline{AB}	$\overline{A} + \overline{B}$
0	0	1	1
0	1	1	1
1	0	1	1
1	1	0	0

Table 2.6

$$2) \overline{\overline{A} + \overline{B}} = \overline{\overline{A}} \overline{\overline{B}}$$

The complement of a sum is equal to the product of the complements. The truth Table 2.7 illustrates this law.

Truth Table :

A	B	$\overline{\overline{A} + \overline{B}}$	$\overline{\overline{A}} \overline{\overline{B}}$
0	0	1	1
0	1	0	0
1	0	0	0
1	1	0	0

Table 2.7

Consensus Theorem

In simplification of Boolean expression, an expression of the form $AB + \overline{A}C + BC$ the term BC is redundant and can be eliminated to form the equivalent expression $AB + \overline{A}C$. The theorem used for this simplification is known as consensus theorem and it is stated as

$$AB + \overline{A}C + BC = AB + \overline{A}C$$

The key to recognize the consensus terms is to first find a pair of terms, one of which contains a variable and the other contains its complement. Now we have to find the third term which should contain the remaining variables from pair of terms eliminating selected variable and its complement.

Proof :

$$\begin{aligned} AB + \bar{A}C + BC &= AB + \bar{A}C + (A + \bar{A})BC \\ &= AB + \bar{A}C + AB + \bar{A}C \\ &= AB + \bar{A}C \end{aligned}$$

Example 2.1 : Solve the given expression using consensus theorem.
 $\bar{A}\bar{B} + AC + \bar{B}\bar{C} + \bar{B}C + AB$

Solution :

$$\begin{aligned} \bar{A}\bar{B} + AC + \bar{B}\bar{C} + \bar{B}C + AB &= \bar{A}\bar{B} + AC + \bar{B}\bar{C} + AB \\ \bar{A}\bar{B} + AC + \bar{B}\bar{C} + \bar{B}C + AB &= \bar{A}\bar{B} + AC + \bar{B}\bar{C} \\ \therefore \bar{A}\bar{B} + AC + \bar{B}\bar{C} + \bar{B}C + AB &= \bar{A}\bar{B} + AC + \bar{B}\bar{C} \end{aligned}$$

Note : The brackets indicate how the consensus terms are identified.

Dual of Consensus Theorem

The dual form of consensus theorem is stated as

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

Proof :

$$\begin{aligned} (A\bar{A} + AC + \bar{A}B + BC)(B + C) &= A\bar{A} + AC + \bar{A}B + BC \\ (AC + \bar{A}B + BC)(B + C) &= AC + \bar{A}B + BC \\ ABC + \bar{A}BB + BCC + ACC + \bar{A}BC + BCC &= AC + \bar{A}B + BC \\ (A + \bar{A})BC + \bar{A}B + BC + AC &= AC + \bar{A}B + BC \\ \bar{A}B + BC + AC &= AC + \bar{A}B + BC \end{aligned}$$

... Proved

Example 2.2 : Solved the following Boolean expression using dual of consensus theorem
 $(A + B)(\bar{A} + C)(B + C)(\bar{A} + D)(B + D)$

Solution :

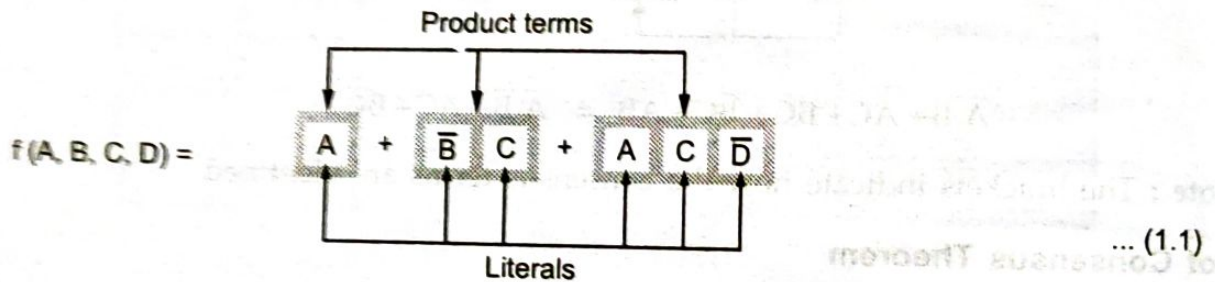
$$\begin{aligned} (A + B)(\bar{A} + C)(B + C)(\bar{A} + D)(B + D) &= (A + B)(\bar{A} + C)(\bar{A} + D)(B + D) \\ &= (A + B)(\bar{A} + C)(\bar{A} + D) \end{aligned}$$

2.4 Boolean Function

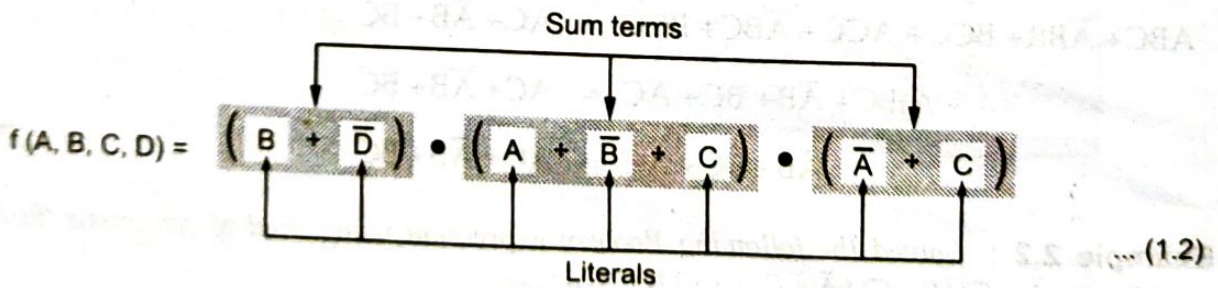
Boolean expressions are constructed by connecting the Boolean constants and variables with the Boolean operations. These Boolean expressions are also known as **Boolean formulas**. We use Boolean expressions to describe **Boolean functions**. For example, if the Boolean expression $(A + \bar{B}) C$ is used to describe the function f , then Boolean function is written as

$$f(A, B, C) = (A + \bar{B}) C \text{ or } f = (A + \bar{B}) C$$

Based on the structure of Boolean expression, it can be categorized in different formulas. One such categorization are the normal formulas. Let us consider the four-variable Boolean function.



In this Boolean function the variables are appeared either in a complemented or an uncomplemented form. Each occurrence of a variable in either a complemented or an uncomplemented form is called a **literal**. Thus, the above Boolean function (1.1) consists of six literals. They appear in the product terms. A **product term** is defined as either a literal or a product (also called conjunction) of literals. Function 1.1 contains three product terms, namely, A , $\bar{B} C$ and $A C \bar{D}$. Let us consider another four variable Boolean function



The above Boolean function consists of seven literals. Here, they appear in the sum terms. A **sum term** is defined as either a literal or a sum (also called disjunction) of literals. Function 1.2 contains three sum terms, namely, $(B + \bar{D})$, $(A + \bar{B} + C)$ and $(\bar{A} + C)$. These literals and terms are arranged in one of the two forms :

- Sum of product form (SOP) and
- Product of sum form (POS).

2.4.1 Sum of Product Form

The words sum and product are derived from the symbolic representations of the OR and AND functions by + and · (addition and multiplication), respectively. But we realize that these are not arithmetic operators in the usual sense. A product term is any group of literals that are ANDed together. For example, ABC, XY and so on. A sum term is any group of literals that are ORed together such as A + B + C, X + Y and so on. A sum of products (SOP) is a group of product terms ORed together. Some examples of this form are :

$$1. \quad f(A, B, C) = \boxed{ABC} + \boxed{A\bar{B}\bar{C}}$$

$$2. \quad f(P, Q, R, S) = \boxed{\bar{P}Q} + \boxed{QR} + \boxed{RS}$$

Each of these sum of products expressions consist of two or more product terms (AND) that are ORed together. Each product term consists of one or more literals appearing in either complemented or uncomplemented form. For example, in the sum of products expression $ABC + A\bar{B}\bar{C}$, the first product term contains literals A, B and C in their uncomplemented form. The second product term contains B and C in their complemented (inverted) form. The sum of product form is also known as **disjunctive normal form** or **disjunctive normal formula**.

2.4.2 Product of Sum Form

A product of sums is any groups of sum terms ANDed together. Some examples of this form are :

$$1. \quad f(A, B, C) = \boxed{(A + B)} \cdot \boxed{(B + C)}$$

$$f(P, Q, R, S) = (P + Q) \cdot (R + \bar{S}) \cdot (P + S)$$

Each of these product of sums expressions consist of two or more sum terms (OR) that are ANDed together. Each sum term consists of one or more literals appearing in either complemented or uncomplemented form. The product of sum form is also known as **conjunctive normal form** or **conjunctive normal formula**.

2.5 Canonical and Standard Forms

The canonical forms are the special cases of SOP and POS forms. These are also known as standard SOP and POS forms.

2.5.1 Standard SOP Form or Minterm Canonical Form

We can realize that in the SOP form, all the individual terms do not involve all literals. For example, in expression $AB + ABC$ the first product term do not contain literal C. If each term in SOP form contains all the literals then the SOP form is known as **standard or canonical SOP form**. Each individual term in the standard SOP form is called **minterm**. Therefore, canonical SOP form is also known as **minterm canonical form**.

In expression $ABC + ABC + \bar{A}BC + A\bar{B}C$ all the literals are present in each product term. In other words we can say that a sum of products is a standard (canonical) sum of products if every product term involves every literal or its complement. One standard sum of products expression is as shown in Fig. 2.1.

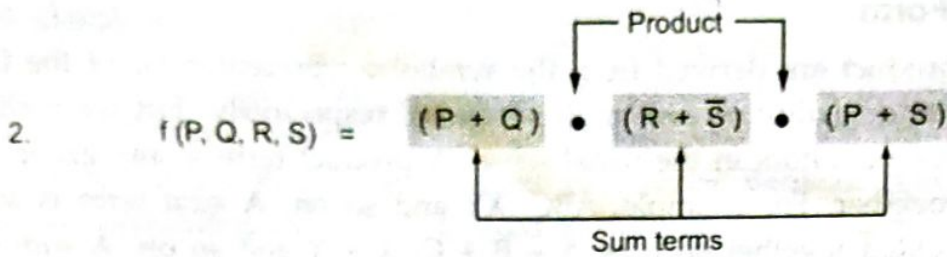
$$f(A, B, C) = \bar{A}BC + ABC + A\bar{B}C$$

Each product term consists of all literals in either complemented form or uncomplemented form

Fig. 2.1 Standard SOP form

2.5.2 Standard POS Form or Maxterm Canonical Form

If each term in POS form contains all the literals then the POS form is known as **standard or canonical POS form**. Each individual term in the standard POS form is called **maxterm**. Therefore, canonical POS form is also known as **maxterm canonical form**. In other words, we can say that a product of sums is a standard or canonical product of sums if every sum term involves every literal or its complement. One standard product of sums expression is as shown in Fig. 2.2.



Each of these product of sums expressions consist of two or more sum terms (OR) that are ANDed together. Each sum term consists of one or more literals appearing in either complemented or uncomplemented form. The product of sum form is also known as **conjunctive normal form** or **conjunctive normal formula**.

2.5 Canonical and Standard Forms

The canonical forms are the special cases of SOP and POS forms. These are also known as standard SOP and POS forms.

2.5.1 Standard SOP Form or Minterm Canonical Form

We can realize that in the SOP form, all the individual terms do not involve all literals. For example, in expression $AB + ABC̄$ the first product term do not contain literal C. If each term in SOP form contains all the literals then the SOP form is known as **standard or canonical SOP form**. Each individual term in the standard SOP form is called **minterm**. Therefore, canonical SOP form is also known as **minterm canonical form**.

In expression $ABC̄ + ABC + \bar{A}BC + A\bar{B}C$ all the literals are present in each product term. In other words we can say that a sum of products is a standard (canonical) sum of products if every product term involves every literal or its complement. One standard sum of products expression is as shown in Fig. 2.1.

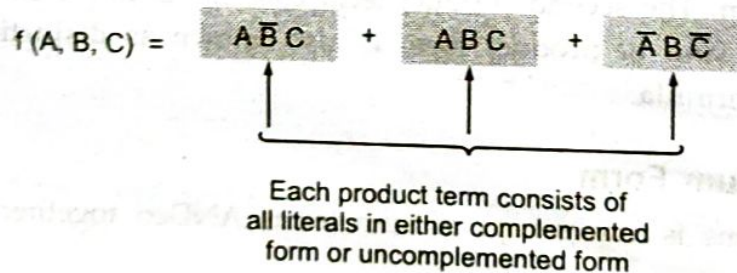


Fig. 2.1 Standard SOP form

2.5.2 Standard POS Form or Maxterm Canonical Form

If each term in POS form contains all the literals then the POS form is known as **standard or canonical POS form**. Each individual term in the standard POS form is called **maxterm**. Therefore, canonical POS form is also known as **maxterm canonical form**. In other words, we can say that a product of sums is a standard or canonical product of sums if every sum term involves every literal or its complement. One standard product of sums expression is as shown in Fig. 2.2.

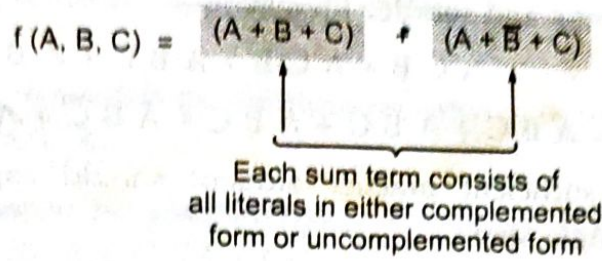


Fig. 2.2 Standard POS form

2.5.3 Converting Expressions in Standard SOP or POS Forms

Sum of products form can be converted to standard sum of products by ANDing the terms in the expression with terms formed by ORing the literal and its complement which are not present in that term. For example for a three literal expression with literals A, B and C, if there is a term AB, where C is missing, then we form term $(C + \bar{C})$ and AND it with AB. Therefore, we get $AB(C + \bar{C}) = ABC + AB\bar{C}$.

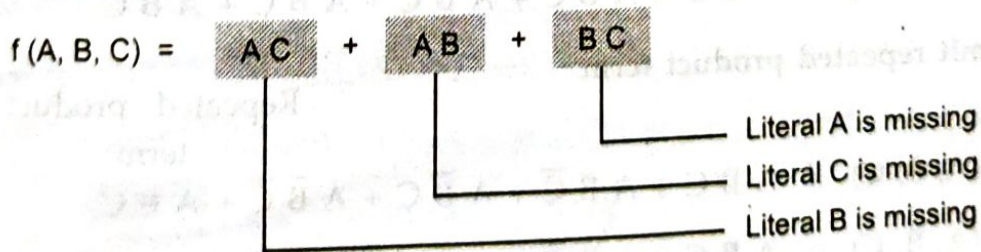
Steps to convert SOP to standard SOP form

- Step 1 : Find the missing literal in each product term if any.
- Step 2 : AND each product term having missing literal/s with term/s form by ORing the literal and its complement.
- Step 3 : Expand the terms by applying distributive law and reorder the literals in the product terms.
- Step 4 : Reduce the expression by omitting repeated product terms if any. Because $A + A = A$.

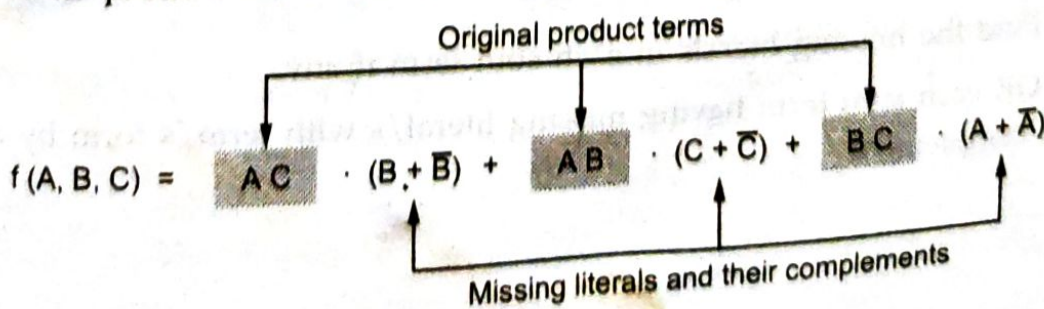
➡ **Example 2.3 :** Convert the given expression in standard SOP form.

$$f(A, B, C) = AC + AB + BC$$

Solution : Step 1 : Find the missing literal/s in each product term



Step 2 : AND product term with (missing literal + it's complement)



Step 3 : Expand the terms and reorder literals.

Expand : $f(A, B, C) = A C B + A C \bar{B} + A B C + A B \bar{C} + B C A + B C \bar{A}$

Recorder : $f(A, B, C) = A B C + A \bar{B} C + A B C + A B \bar{C} + A B C + \bar{A} B C$

Note : After having sufficient practice student should expand product term and reorder literals in it in a single step.

Step 4 : Omit repeated product terms

$$f(A, B, C) = A B C + A \bar{B} C + \boxed{A B C} + A B \bar{C} + \boxed{A B C} + \bar{A} B C$$

$$\therefore f(A, B, C) = A B C + A \bar{B} C + A B \bar{C} + \bar{A} B C$$

Example 2.4 : Convert the given expression in standard SOP form.

Solution : Step 1 : Find the missing literal/s in each product term

$$f(A, B, C) = \boxed{A} + A B C$$

Literals B and C are missing

Step 2 : AND product term with (missing literal + its complement)

Original terms

$$f(A, B, C) = \boxed{A} \cdot (B + \bar{B}) \cdot (C + \bar{C}) + \boxed{A B C}$$

Missing literals and their complements

Step 3 : Expand the terms and reorder literals

$$f(A, B, C) = A B C + A B \bar{C} + A \bar{B} C + A \bar{B} \bar{C} + A B C$$

Step 4 : Omit repeated product term

$$f(A, B, C) = A B C + A B \bar{C} + A \bar{B} C + A \bar{B} \bar{C} + \boxed{A B C}$$

Repeated product term

$$\therefore f(A, B, C) = A B C + A B \bar{C} + A \bar{B} C + A \bar{B} \bar{C}$$

Steps to convert POS to standard POS form

Step 1 : Find the missing literals in each sum term if any

Step 2 : OR each sum term having missing literal/s with term/s form by ANDing the literal and its complement.

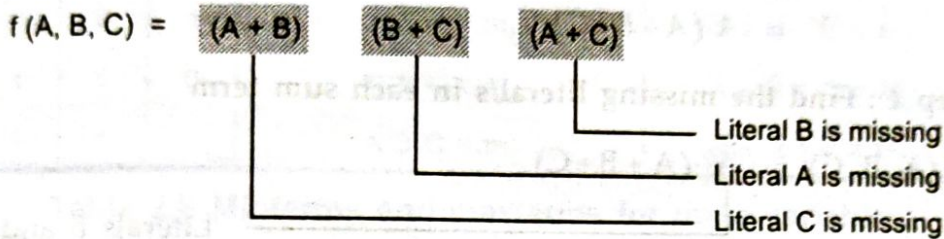
Step 3 : Expand the terms by applying distributive law and reorder the literals in the sum terms.

Step 4 : Reduce the expression by omitting repeated sum terms if any.
Because $A \cdot A = A$.

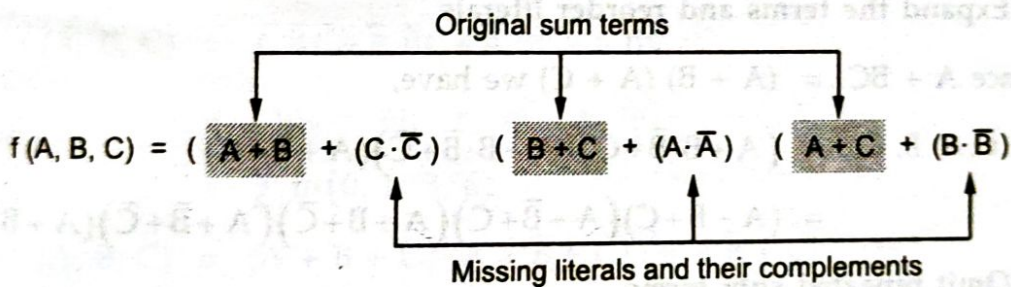
Example 2.5 : Convert the given expression in standard POS form.

$$f(A, B, C) = (A + B)(B + C)(A + C)$$

Solution : Step 1 : Find the missing literal/s in each sum term



Step 2 : OR sum term with (missing literal • its complement)



Step 3 : Expand the terms and reorder literals

Expand :

Since $A + BC = (A + B)(A + C)$ we have,

$$f(A, B, C) = (A + B + C)(A + B + \bar{C})(B + C + A)(B + C + \bar{A})$$

$$(A + C + B)(A + C + \bar{B})$$

Reorder :

$$f(A, B, C) = (A + B + C)(A + B + \bar{C})(A + B + C)(\bar{A} + B + C)$$

$$(A + B + C)(A + \bar{B} + C)$$

Step 4 : Omit repeated sum terms

$$f(A, B, C) = (A + B + C) (A + B + \bar{C}) \overbrace{(A + B + C)}^{\text{Repeated sum terms}} (\bar{A} + B + C) \overbrace{(A + B + C)}^{\text{Repeated sum terms}} (A + \bar{B} + C)$$

$$\therefore f(A, B, C) = (A + B + C) (A + B + \bar{C}) (\bar{A} + B + C) (A + \bar{B} + C)$$

Example 2.6 : Convert the given expression in standard POS form.

$$Y = A \cdot (A + B + C)$$

Solution : Step 1 : Find the missing literal/s in each sum term

$$f(A, B, C) = \overbrace{A}^{\text{Literals B and C are missing}} \cdot (A + B + C)$$

Step 2 : OR sum term with (missing literal • its complement)

$$f(A, B, C) = (A + B \cdot \bar{B} + C \cdot \bar{C})(A + B + C)$$

Step 3 : Expand the terms and reorder literals

Since $A + BC = (A + B)(A + C)$ we have,

$$f(A, B, C) = (A + B \cdot \bar{B} + C)(A + B \cdot \bar{B} + \bar{C})(A + B + C)$$

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

Step 4 : Omit repeated sum terms

$$f(A, B, C) = (A + B + C)(A + \bar{B} + C)(A + B + \bar{C})(A + \bar{B} + \bar{C}) \overbrace{(A + B + C)}^{\text{Repeated sum term}}$$

$$\therefore f(A, B, C) = (A + B + C)(A + \bar{B} + C)(A + B + \bar{C})(A + \bar{B} + \bar{C})$$

2.5.4 M-Notations : Minterms and Maxterms

Each individual term in standard SOP form is called **minterm** and each individual term in standard POS form is called **maxterm**. The concept of minterms and maxterms allows us to introduce a very convenient shorthand notations to express logical functions. Table 2.8 gives the minterms and maxterms for a three literal/variable logical function where the number of minterms as well as maxterms is $2^3 = 8$. In general, for an n-variable logical function there are 2^n minterms and an equal number of maxterms.

Variables			Minterms	Maxterms
A	B	C	m_i	M_i
0	0	0	$\bar{A} \bar{B} \bar{C} = m_0$	$A + B + C = M_0$
0	0	1	$\bar{A} \bar{B} C = m_1$	$A + B + \bar{C} = M_1$
0	1	0	$\bar{A} B \bar{C} = m_2$	$A + \bar{B} + C = M_2$
0	1	1	$\bar{A} B C = m_3$	$A + \bar{B} + \bar{C} = M_3$
1	0	0	$A \bar{B} \bar{C} = m_4$	$\bar{A} + B + C = M_4$
1	0	1	$A \bar{B} C = m_5$	$\bar{A} + B + \bar{C} = M_5$
1	1	0	$A B \bar{C} = m_6$	$\bar{A} + \bar{B} + C = M_6$
1	1	1	$A B C = m_7$	$\bar{A} + \bar{B} + \bar{C} = M_7$

Table 2.8 Minterms and maxterms for three variables

As shown in Table 2.8 each minterm is represented by m_i and each maxterm is represented by M_i , where the subscript i is the decimal number equivalent of the natural binary number. With these shorthand notations logical function can be represented as follows :

$$\begin{aligned}
 1. \quad f(A, B, C) &= \bar{A} \bar{B} \bar{C} + \bar{A} \bar{B} C + \bar{A} B \bar{C} + A B \bar{C} \\
 &= m_0 + m_1 + m_3 + m_6 \\
 &= \sum m(0, 1, 3, 6) \\
 2. \quad f(A, B, C) &= (A + B + \bar{C})(A + \bar{B} + \bar{C})(\bar{A} + \bar{B} + C) \\
 &= M_1 + M_3 + M_6 \\
 &= \pi M(1, 3, 6)
 \end{aligned}$$

where Σ denotes **sum of product** while π denotes **product of sum**.

We know that logical expression can be represented in the truth table form. It is possible to write logical expression in standard SOP or POS form corresponding to a given truth table. The logical expression corresponding to a given truth table can be written in a standard sum of products form by writing one product term for each input combination that produces an output of 1. These product terms are ORed together to create the standard sum of products. The product terms are expressed by writing complement of a variable when it appears as an input 0, and the variable itself when it appears as an input 1. Consider, for example, the following truth table :

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

← $\bar{A}\bar{B}\bar{C}$

← $\bar{A}BC$

← ABC

Table 2.9

The product corresponding to input combination 010 is $\bar{A}\bar{B}\bar{C}$, the product corresponding to input combination 011 is $\bar{A}BC$, and product corresponding to input combination 110 is ABC . Thus the standard sum of products form is

$$f(A, B, C) = \bar{A}\bar{B}\bar{C} + \bar{A}BC + ABC$$

$$= m_2 + m_3 + m_6$$

The logic expression corresponding to a truth table can also be written in a standard product of sums form by writing one sum term for each output 0. The sum terms are expressed by writing complement of a variable when it appears as an input 1, and the variable itself when it appears as an input 0. Consider, for example, the following truth table :

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

← $A + \bar{B} + C$

← $\bar{A} + B + \bar{C}$

Table 2.10

The sum corresponding to input combinations 010 is $A + \bar{B} + C$, and the sum corresponding to input 101 is $\bar{A} + B + \bar{C}$. Thus, the standard product of sums form is

$$f(A, B, C) = (A + \bar{B} + C)(\bar{A} + B + \bar{C})$$

$$= M_2 + M_5$$

2.5.5 Complements of Canonical Formulae

A product of sums form derived from a truth table is logically equivalent to a sum of products form derived from the Truth Table 2.10. Let us write the standard SOP and POS from the previous truth table :

SOP form : $f(A, B, C) = \bar{A}\bar{B}\bar{C} + \bar{A}\bar{B}C + \bar{A}BC + A\bar{B}\bar{C} + A\bar{B}C + ABC$

POS form : $f(A, B, C) = (A + \bar{B} + C)(\bar{A} + B + \bar{C})$

Now by simplifying equation in the POS form we have,

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

Coverting to standard sum of products we have,

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

Rearranging terms we have,

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

Therefore, we can say that POS and SOP derived from the same truth table are logically equivalent. In terms of minterms and maxterms we can then write

$$f(A, B, C) = m_0 + m_1 + m_3 + m_4 + m_6 + m_7 = M_2 + M_5$$

$$\begin{aligned} \therefore f(A, B, C) &= \sum m(0, 1, 3, 4, 6, 7) \\ &= \prod M(2, 5) \end{aligned}$$

From the above expressions we can easily notice that there is a complementary type of relationship between a function expressed in terms of maxterms. Using this complementary relationship we can find logical function in terms of maxterms if function in minterms is known or vice-versa. For example, for a four variables if

$$f(A, B, C, D) = \sum m(0, 2, 4, 6, 8, 10, 12, 14)$$

$$\text{then } f(A, B, C, D) = \prod M(1, 3, 5, 7, 9, 11, 13, 15)$$

2.6 Other Logic Operations

In chapter 1, we have seen the basic logic operators - AND, OR and NOT. Using the combination of these basic operators we can perform additional logic operations such as NAND, NOR, exclusive-OR and exclusive NOR.

The NAND operation is the complement of AND operation and is an abbreviation of NOT-AND. Similarly, the NOR is the complement of OR operation and is an abbreviation of NOT-OR. The exclusive-OR, abbreviated XOR, is similar to OR, but excludes the combination of same two variables. For example, if both binary variables are logic 1 or 0, the XOR operation gives logic 0 output. The exclusive-NOR, abbreviated XNOR, operation gives output 1 when the two binary variables are equal, i.e., When both are 0 or both are 1. Thus, exclusive-NOR operation is also known as equivalence operation. The exclusive OR and equivalence operations are the complements of each other.

2.7 Digital Logic Gates

Logic gates are the basic elements that make up a digital system. The electronic gate is a circuit that is able to operate on a number of binary inputs in order to perform a particular logical function. The types of gates available are the NOT, AND, OR, NAND, NOR, exclusive-OR, and the exclusive-NOR. Except for the exclusive-NOR gate, they are available in monolithic integrated circuit form.

The gate is a digital circuit with one or more input voltages but only one output voltage. By connecting the different gates in different ways, we can build circuits that perform arithmetic and other functions associated with the human brain because they simulate mental processes.

The operation of a logic gate can be easily understood with the help of "truth table". A truth table is a table that shows all the input-output possibilities of a logic circuit; i.e. the truth table indicates the outputs for different possibilities of the inputs.

2.7.1 Inverter : NOT Gate

The inverter (NOT circuit) performs a basic logic function called "inversion" or "complementation". The inverter changes one logic level to its opposite level. In terms of bits, it changes a logic 1 to a logic 0 and a logic 0 to a logic 1. The Fig. 2.3 shows the symbol for the inverter.



Fig. 2.3 Inverter symbol

The bubble [o] appearing on the output is the negation (inversion) indicator.

Inverter Operation :

When a HIGH level is applied to an inverter input, a LOW level will appear on its output. When a LOW level is applied to its input, a HIGH level will appear on its output. This operation is summarized in the truth table 2.11, which indicates the output for each possible input in terms of levels and bits.

Inverter Truth Table :

Input	Output	Input	Output
LOW	HIGH	0	1
HIGH	LOW	1	0

Table 2.11

2.7.2 AND Gate

The AND gate performs logical multiplication, more commonly known as the AND function. The AND gate may have two or more inputs and a single output, as indicated by the standard logic symbols shown in the Fig. 2.4.



Fig. 2.4 (a) Two inputs to AND gate



Fig. 2.4 (b) Four inputs to AND gate

Gates with two and four inputs are shown in Fig. 2.4; however, an AND gate can have any number of inputs greater than one. The operation of the AND gate is such that the output is HIGH only when all of the inputs are HIGH. When any of the inputs are LOW, the output is LOW. Hence, the AND gate determines when certain conditions are simultaneously true, as indicated by HIGH levels on all of its inputs, and to produce a HIGH on its output indicating this conditions. The Fig. 2.5 illustrates a two-input AND gate with all four possibilities of input combinations, and the resulting output for each.

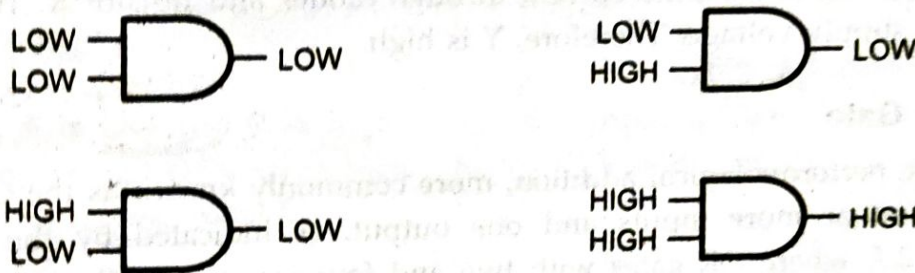


Fig. 2.5 Four possible inputs for two input AND gate and resulting outputs

The truth table for a two-input AND gate is shown in Table 2.12. This table can be expanded for any number of inputs. For any AND gate, regardless of the number of inputs, the output is high only when all inputs are HIGH.

Inputs		Output
A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

Table 2.12 Truth table for 2 input AND gate

Fig. 2.6 shows one way to build a 2-input AND gate. The inputs are labelled A and B, while the output is Y. Let us assume a supply voltage V_{CC} of +5 V. Also we will assume the input voltages are either 0 V (Low) or +5 V (High). With 2 inputs, there are four possible input cases and we will now observe the output for all four input cases.

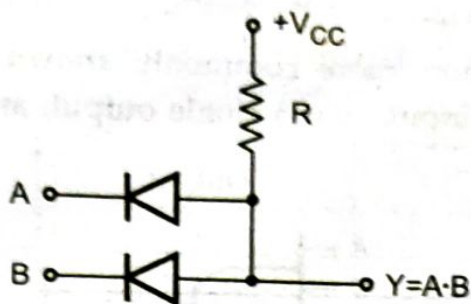


Fig. 2.6 2-input AND gate

Case 1 : A is low and B is low :
 When both input voltages are low, the cathode of each diode is grounded. Therefore, the positive supply forward-biases both diodes in parallel. Because of this, the output voltage is ideally zero (practically 0.7 V for Si). This means Y is low.

Case 2 : A is low and B is high : When A is low, the upper diode is forward-biased (ON), and it pulls the output down to a low voltage, i.e. $Y = 0$. With the B input high, the lower diode goes into reverse bias (OFF).

Case 3 : A is high and B is low : Because of the symmetry of the circuit, the circuit operation is similar to case 2. But in this case, upper diode is reverse biased (OFF), lower diode is forward biased (ON), and Y is low.

Case 4 : A is high and B is high : When both inputs are at +5 V, both diodes are reverse biased and there is no current through diodes and resistor R. This pulls up the output Y to the supply voltage. Therefore, Y is high.

2.7.3 The OR Gate

The OR gate performs logical addition, more commonly known as the OR function. An OR gate has two or more inputs and one output, as indicated by the standard logic symbol in Fig. 2.7, where OR gates with two and four inputs are illustrated. An OR gate



Fig. 2.7 (a) Two inputs to OR gate



Fig. 2.7 (b) Four inputs to OR gate

produces a HIGH on the output when any of the inputs is HIGH. The output is LOW only when all of the inputs are LOW. Hence, the purpose of an OR gate is to determine when one or more of its inputs are HIGH and to produce a HIGH on its output to indicate this condition. The Fig. 2.8 illustrates the logic operation for a two-input OR gate for all four possible input combinations :



Fig. 2.8 Four possible inputs for two input OR gate and resulting outputs

The truth table 2.13 describes the logical operation of the two-input OR gate. The truth table can be expanded for any number of inputs; however, regardless of the number of inputs, the output is HIGH when any of the inputs is HIGH.

Inputs		Output Y
A	B	
0	0	0
0	1	1
1	0	1
1	1	1

Table 2.13 Truth table for 2-input OR gate

Fig. 2.9 shows one way to build a 2-input OR gate. The inputs are labelled A and B, while the output is Y. Let us assume the input voltages are either 0 V (Low) or +5 V (High). With 2-inputs, there are four possible input cases and we will now observe the output for all four input cases.

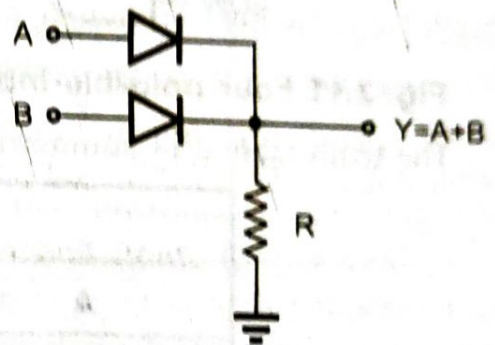


Fig. 2.9 2-Input OR gate

Case 1 : A is low and B is low : When both input voltages are low, the anodes of both the diodes are grounded. Therefore, the diodes are reverse biased and output, Y is low.

Case 2 : A is low and B is high : When B input is high, it forward-biases lower diode, producing an output voltage high (ideally +5 V and practically + 4.3 V). Note that the upper diode is reverse biased.

Case 3 : A is high and B is low : Because of the symmetry of the circuit, the circuit operation is similar to case 2. But in this case, upper diode is forward biased (ON), and lower diode is reverse biased.

Case 4 : A is high and B is high : When both inputs are at +5 V, both diodes are forward biased. Since the input voltages are in parallel, the output voltage is ideally +5 V and practically +4.3 V. Therefore, output Y is high.

2.7.4 The NAND Gate

The term NAND is a contraction of NOT-AND and implies an AND function with a complemented (inverted) output. A standard logic symbol for a two-input NAND gate and its equivalency to an AND gate followed by an inverter are shown in Fig. 2.10.

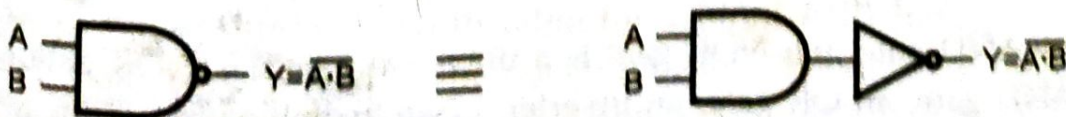


Fig. 2.10 NAND gate symbol and equivalent circuit

The NAND gate is a **universal gate** as it can be used to construct an AND gate, an OR gate an inverter, or any combination of these functions. The logical operation of the NAND gate is such that a LOW output occurs only when **all** inputs are HIGH. When any of the inputs is LOW, the output will be HIGH. Note that this operation is **opposite** to that of the AND as far as output is concerned. Fig. 2.11 illustrates the logical operation of a two-input NAND gate for all four input combinations.

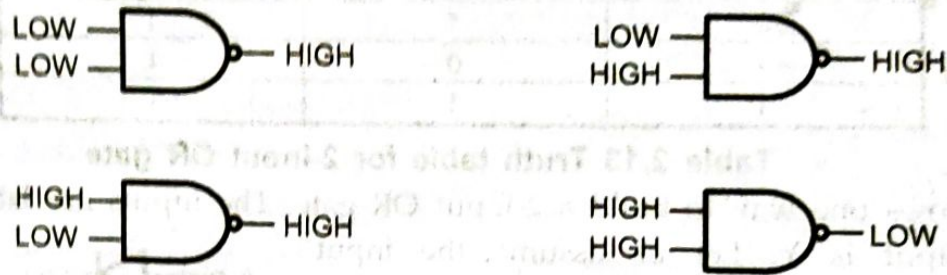


Fig. 2.11 Four possible inputs for two input NAND Gate and resulting outputs

The truth table 2.14 summarizes the logical operation of the two input NAND gate.

Inputs		Output
A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0

Table 2.14 Truth table for 2-input NAND gate

2.7.5 The NOR Gate

The term NOR is a contraction of NOT-OR and implies an OR function with an inverted output. A standard logic symbol for a two-input NOR gate and its equivalent OR gate followed by an inverter is shown in the Fig. 2.12.

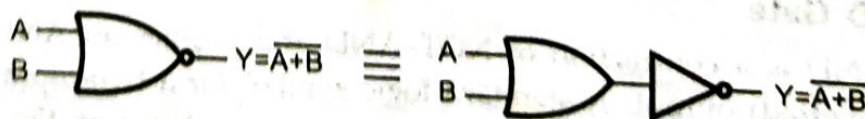


Fig. 2.12 NOR gate symbol and equivalent circuit

Similar to NAND gate, the NOR gate is a universal gate, i.e. NOR gate can be used to construct an AND gate, an OR gate, an inverter, or any combination of these functions.

The logic operation of the NOR gate is that a LOW output occurs when **any** of its inputs is HIGH. Only when all of its inputs are LOW, the output is HIGH.

The Fig. 2.13 illustrates the logical operation of a two-input NOR gate for all four possible input combinations.

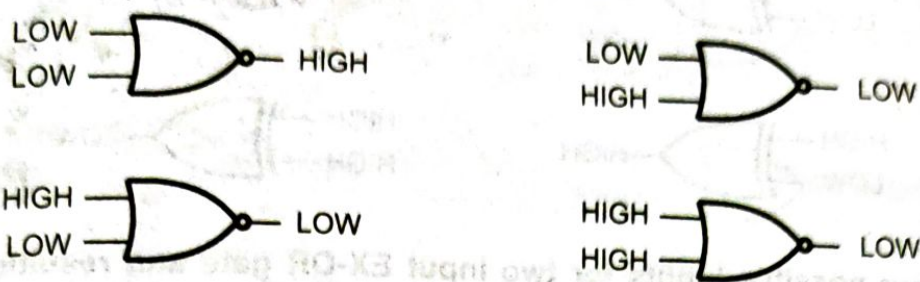


Fig. 2.13 Four possible inputs for two input NOR gate and resulting outputs

The truth table for a two-input NOR gate is shown in Table 2.15 :

Inputs		Output
A	B	Y
0	0	1
0	1	0
1	0	0
1	1	0

Table 2.15 Truth table for 2-input NOR gate

2.7.6 The Exclusive-OR Gate

The EX-OR gate is an abbreviation for Exclusive-OR gate. An EX-OR gate has two or more inputs and one output, as indicated by the standard logic symbol in Fig. 2.14 where EX-OR gates with two and four inputs are shown.



Fig. 2.14 (a) Two input EX-OR

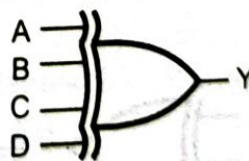


Fig. 2.14 (b) Four input EX-OR

It recognizes only the words that have an odd number of ones. This means that for odd number of ones, output of EX-OR gate is high. The Fig. 2.15 illustrates the logic operation for a two-input EX-OR gate for all four possible input combinations.

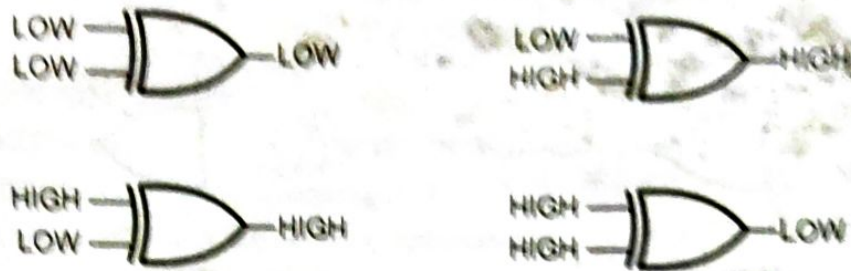


Fig. 2.15 Four possible inputs for two input EX-OR gate and resulting outputs

The truth table 2.16 describes the logical operations of the two-input EX-OR gate.

Inputs		Output
A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

Table 2.16 Truth table for 2-input EX-OR gate

The truth table can be expanded for any number of inputs; however, regardless of the number of inputs, the output is high only when odd number of inputs are HIGH.

2.7.7 The Exclusive-NOR Gate

The term EX-NOR is a contraction of NOT-X-OR, NOT exclusive OR gate. It is logically equivalent to an EX-OR gate followed by an inverter. An EX-NOR gate has two or more inputs and one output, as indicated by the standard logic symbol in Fig. 2.16 where EX-NOR gates with two and four inputs are shown.



Fig. 2.16 (a) Two input EX-NOR

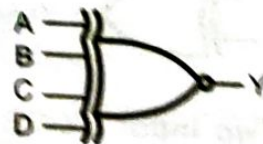


Fig. 2.16 (b) Four input EX-NOR

It recognizes only the words that have an even number of ones and inputs having all zeroes. This means that for even number of ones at the input, or inputs having all zeroes, the output of EX-NOR gate is high. The Fig. 2.17 illustrates the logic operation for a two input EX-NOR gate for all four possible input combinations.

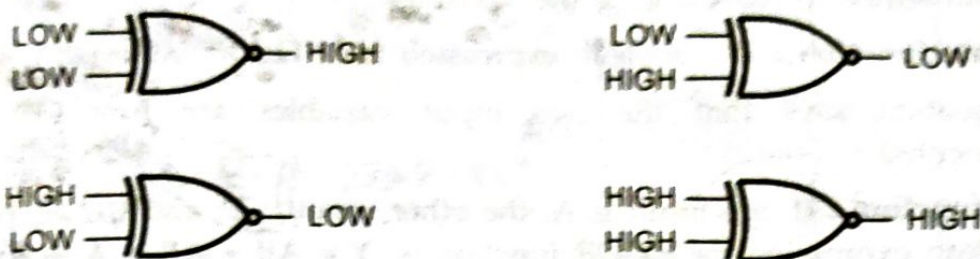


Fig. 2.17 Four possible inputs for two input EX-NOR gate and resulting outputs

The truth table 2.17 describes the logical operations of the two input EX-NOR gate. The truth table can be expanded for any number of input; however, regardless of the number of inputs, the output is high when even number of inputs are high or when all input are zeroes.

Inputs		Output
A	B	Y
0	0	1
0	1	0
1	0	0
1	1	1

Table 2.17 Truth table for 2-input EX-NOR

2.7.8 Boolean Functions for Logic Gates

1. **NOT gate function** : The operation of an inverter (NOT circuit) can be expressed as follows : If the input variable is A and the output variable is Y; then $Y = \bar{A}$. The output is complement of the input.

2. **AND gate function** : Let the two input variables be A and B and the output variable Y; then the Boolean expression is $Y = AB$.

If there are four input variables A, B, C, D, then the output $Y = ABCD$.

The output is 1 (HIGH) only when all the inputs are 1s (HIGH).

3. **OR gate function** : If one input is A, the other input is B, and the output is Y; then the Boolean expression for OR function is $Y = A+B$.

If there are four input variables A, B, C, D; then the output is $Y = A + B + C + D$.

The output is a 1 (HIGH) when any one or more of the inputs are 1 (HIGH).

4. **NAND gate function** : The Boolean expression for NAND function is $Y = \overline{AB}$.

This expression tells that the two input variables, A and B, are first ANDed and then complemented, as indicated by the bar over the AND expression. The NAND

expression can be extended to more than two input variables by including additional letters to represent all the variables.

5. **NOR gate function** : The Boolean expression for NOR function is $Y = \overline{A+B}$.

This equation says that the two input variables are first ORed and then complemented.

6. **EX-OR function** : If one input is A, the other input is B, and the output is Y. Then the Boolean expression for EX-OR function is $Y = A\overline{B} + \overline{A}B = A \oplus B$.

7. **EX-NOR function** : If one input is A, the other input is B, and the output is Y. Then the Boolean expression for EX-NOR function is $Y = AB + \overline{A}\overline{B} = \overline{A \oplus B}$ or $A \odot B$.

This equation says that the two input variables are first EX-ORed and then complemented.

The following Table 2.18 summarizes the logic expressions for all logic gates.


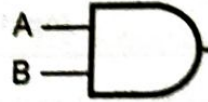

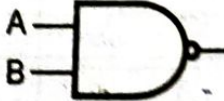



<p>NOT :  $Y = \overline{A}$</p>	<p>AND :  $Y = A \cdot B \text{ or } AB$</p>
<p>OR :  $Y = A + B$</p>	<p>NAND :  $Y = \overline{A \cdot B} \text{ or } \overline{AB}$</p>
<p>NOR :  $Y = \overline{A + B}$</p>	<p>EX-OR :  $Y = A\overline{B} + \overline{A}B = A \oplus B$</p>
<p>EX-NOR :  $Y = AB + \overline{A}\overline{B} = \overline{A \oplus B} \text{ or } A \odot B$</p>	

Table 2.18

2.7.9 Universal Gates

The NAND and NOR gates are known as universal gates, since any logic function can be implemented using NAND or NOR gates. This is illustrated in following sections.

2.7.9.1 NAND Gate

The NAND gate can be used to generate the NOT function, the AND function, the OR function, and the NOR function.

NOT Function :

An inverter can be made from a NAND gate by connecting all of the inputs together and creating, in effect, a single common input, as shown in Fig. 2.18, for a two-input gate.

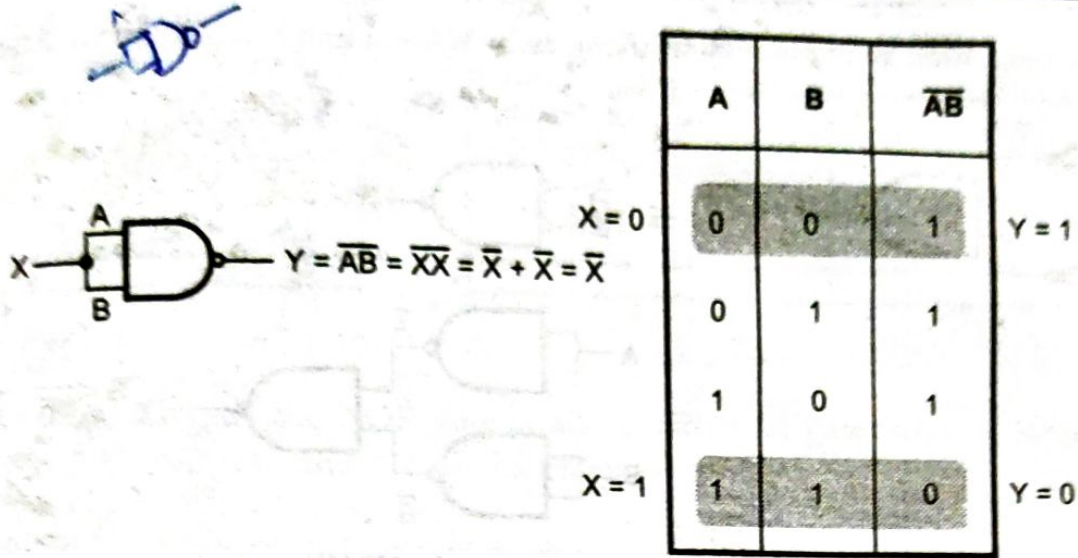


Fig. 2.18 NOT function using NAND gate

AND Function :

An AND function can be generated using only NAND gates. It is generated by simply inverting output of NAND gate; i.e. $\overline{\overline{AB}} = AB$, Fig. 2.19 shows the two input AND gate using NAND gates.

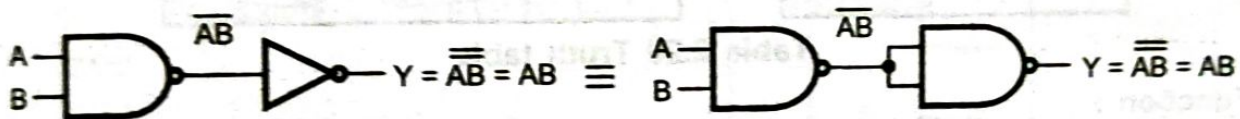


Fig. 2.19 AND function using NAND gates

A	B	AB
0	0	0
0	1	0
1	0	0
1	1	1

=

A	B	\overline{AB}	$\overline{\overline{AB}}$
0	0	1	0
0	1	1	0
1	0	1	0
1	1	0	1

Table 2.19 Truth Table

OR Function :

OR function is generated using only NAND gates as follows : We know that Boolean expression for OR gate is

$$\begin{aligned}
 Y &= A + B \\
 &= \overline{\overline{A}} + \overline{\overline{B}} \\
 &= \overline{\overline{A} \cdot \overline{B}}
 \end{aligned}$$

Rule 9 : $[\overline{\overline{A}} = A]$

DeMorgan's Theorem 1

The above equation is implemented using only NAND gates as shown in the Fig.2.20.

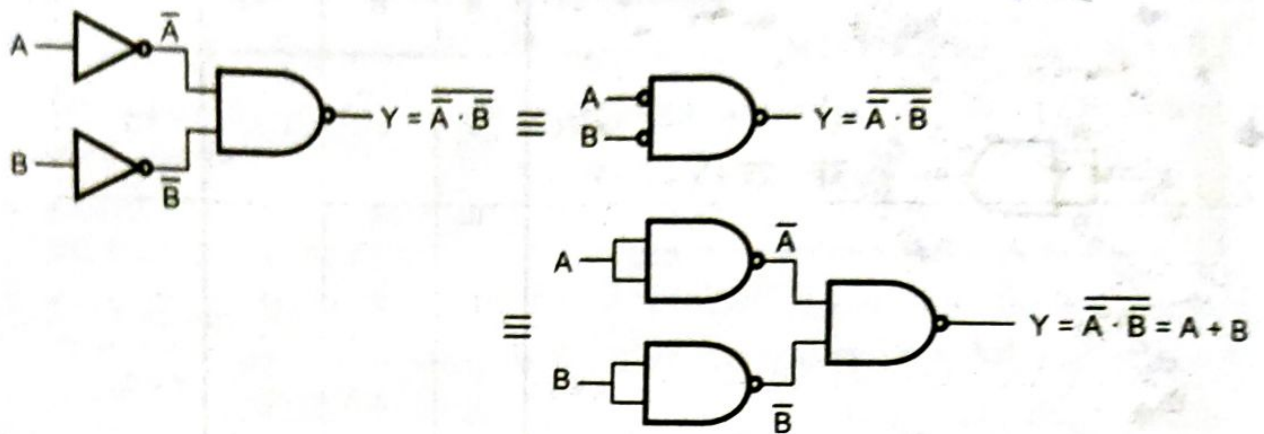


Fig. 2.20 OR function using only NAND gates

Note : Bubble at the input of NAND gate indicates inverted input.

A	B	A + B
0	0	0
0	1	1
1	0	1
1	1	1

A	B	$\bar{A} \cdot \bar{B}$	$\overline{\bar{A} \cdot \bar{B}}$
0	0	1	0
0	1	0	1
1	0	0	1
1	1	0	1

Table 2.20 Truth table

NOR Function :

NOR function is generated using only NAND gates as follows : We know that Boolean expression for NOR gate is

$$\begin{aligned}
 Y &= \overline{A + B} \\
 &= \bar{A} \cdot \bar{B} \\
 &= \overline{\overline{\bar{A} \cdot \bar{B}}}
 \end{aligned}$$

DeMorgan's Theorem 2

Rule 9 : $[\overline{\bar{A}} = A]$

The above equation is implemented using only NAND gates, as shown in the Fig. 2.21.

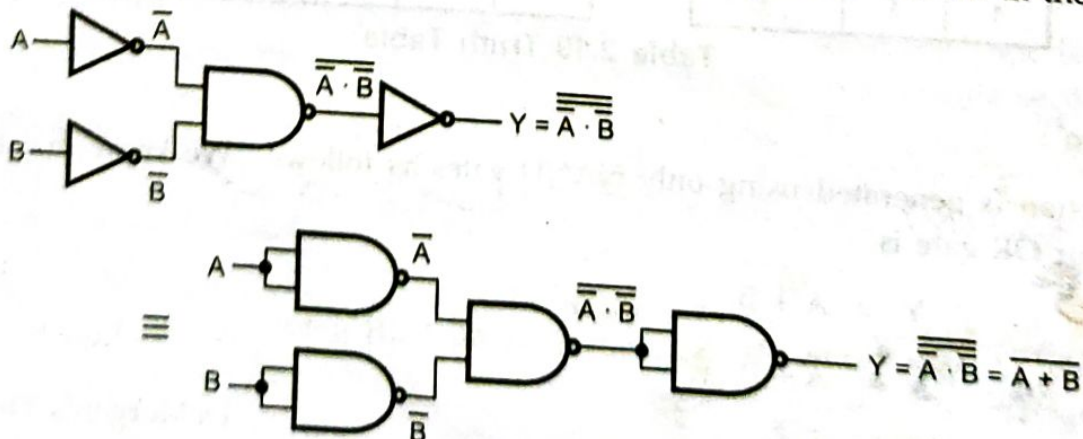


Fig. 2.21 NOR function using only NAND gates

A	B	$\overline{A+B}$
0	0	1
0	1	0
1	0	0
1	1	0

A	B	$\overline{A \cdot B}$	$\overline{\overline{A \cdot B}}$	$\overline{\overline{\overline{A \cdot B}}}$
0	0	1	0	1
0	1	0	1	0
1	0	0	1	0
1	1	0	1	0

2.7.9.2 NOR Gate

Similar to NAND gate, the NOR gate is also a universal gate, since it can be used to generate the NOT, AND, OR and NAND functions.

NOT Function :

An inverter can be made from a NOR gate by connecting all of the inputs together and creating, in effect, a single common input, as shown in Fig. 2.22.

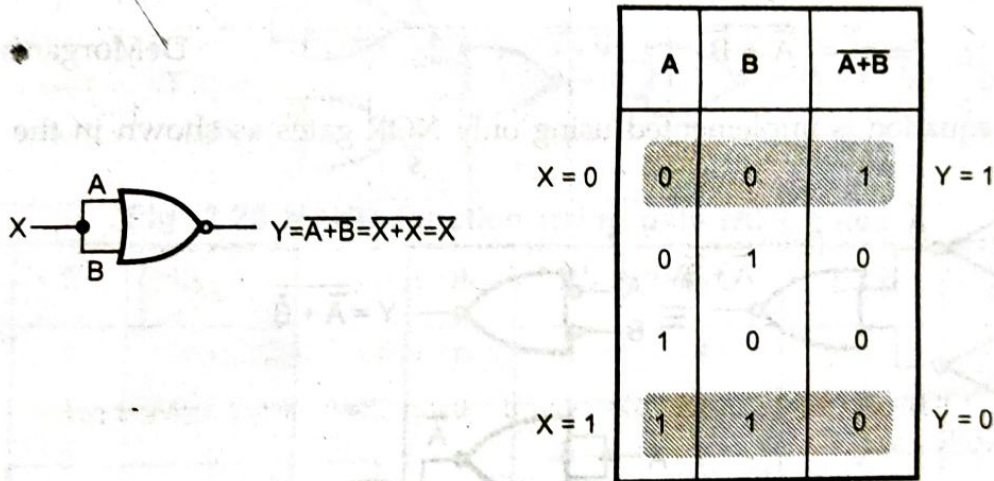


Fig. 2.22 NOT function using NOR gate

OR Function :

An OR function can be generated using only NOR gates. It can be generated by simply inverting output of NOR gate; i.e. $\overline{\overline{A+B}} = A + B$. Fig. 2.23 shows the two input OR gate using NOR gates.

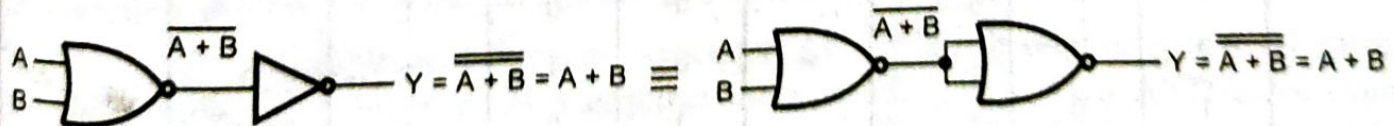


Fig. 2.23 OR function using NOR gates

A	B	$A + B$	$\overline{A + B}$
0	0	0	1
0	1	1	0
1	0	1	0
1	1	1	0

Table 2.21 Truth table

AND Function :

AND function is generated using only NOR gates as follows : We know that Boolean expression for AND gate is

$$\begin{aligned}
 Y &= A \cdot B \\
 &= \overline{\overline{A} \cdot \overline{B}} \\
 &= \overline{\overline{A + B}}
 \end{aligned}$$

Rule 9 : $[\overline{\overline{A}} = A]$

DeMorgan's Theorem 2

The above equation is implemented using only NOR gates as shown in the Fig. 2.24.

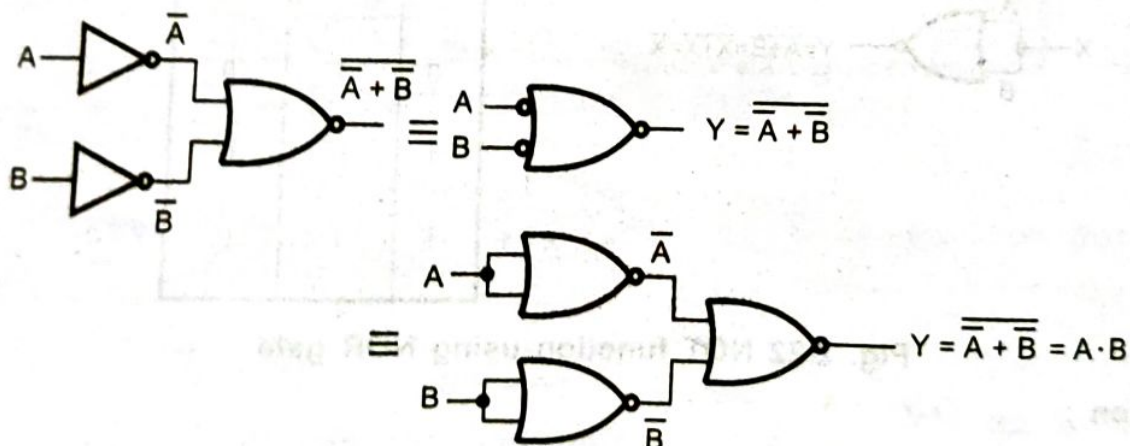


Fig. 2.24 AND function using NOR gates

Note : Bubble at the input of NOR gate indicates inverted input.

A	B	$A \cdot B$	$\overline{A + B}$	$\overline{\overline{A + B}}$
0	0	0	1	0
0	1	0	1	0
1	0	0	1	0
1	1	1	0	1

Table 2.22 Truth table

NAND Function :

NAND function is generated using only NOR gates as follows : We know that Boolean expression for NAND gate is

$$\begin{aligned}
 Y &= \overline{A \cdot B} \\
 &= \overline{A} + \overline{B} \\
 &= \overline{\overline{\overline{A} + \overline{B}}}
 \end{aligned}$$

DeMorgan's Theorem 1

Rule 9 : $\overline{\overline{A}} = A$

The above equation is implemented using only NOR gates, as shown in the Fig. 2.25.

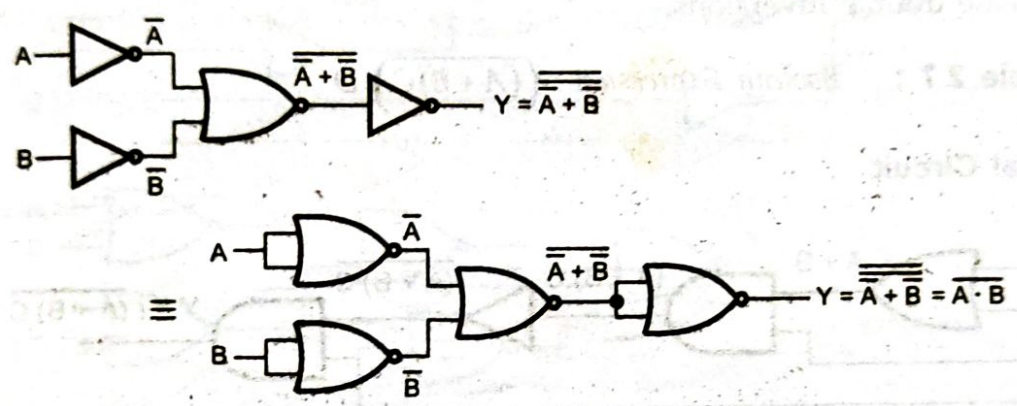


Fig. 2.25 NAND function using only NOR gates

A	B	$\overline{A \cdot B}$
0	0	1
0	1	1
1	0	1
1	1	0

=

A	B	$\overline{A} + \overline{B}$	$\overline{\overline{\overline{A} + \overline{B}}}$	$\overline{\overline{\overline{A} + \overline{B}}}$
0	0	1	0	1
0	1	1	0	1
1	0	1	0	1
1	1	0	1	0

Truth table 2.23

2.7.10 Conversion of AND/OR/NOT Logic to NAND/NOR Logic using Graphical Procedure

When we implement any Boolean expression most of the times we find that it involves various logic gates. We know that logic gates are available in standard IC packages. Therefore to implement given Boolean expression we need various standard ICs. Many times all gates within the standard ICs are not utilized.

For example, to implement Boolean expression $AB + \overline{CD}$, we require two AND gates, one OR gate and one inverter. This requires three standard ICs. Two AND gates from AND IC and only one OR gate and one inverter are utilized from OR and inverter ICs, respectively. Other gates from these three ICs are not utilized. To improve utilization of ICs and to reduce number of ICs required, one can use only NAND/NOR gates to implement Boolean expression. To do this, we have to convert given AND/OR/NOT Boolean expression logic to NAND/NOR logic. This is explained in this section.

Steps for converting to NAND/NOR logic using graphical procedure

1. Draw AND/OR logic.
2. If NAND hardware has been chosen, add bubbles on the output of each AND gate, and bubbles on input side to all OR gates.
3. If NOR hardware has been chosen add bubbles on output of each OR gate and bubble on input of each AND gate.
4. Add or subtract an inverter on each line that received a bubble in step 2 or 3
5. Replace bubbled OR by NAND and bubbled AND by NOR.
6. Eliminate double inversions.

Example 2.7 : Boolean Expression : $((A+B)C)D$

Original Circuit

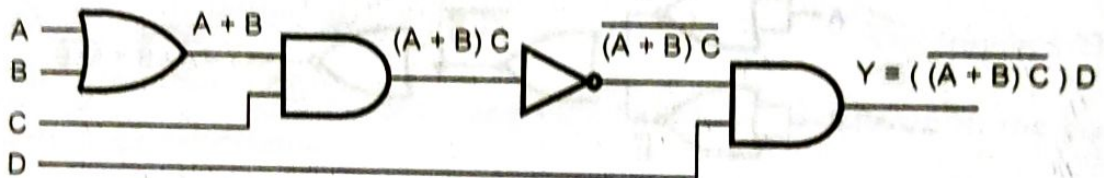


Fig. 2.26

NAND Circuit

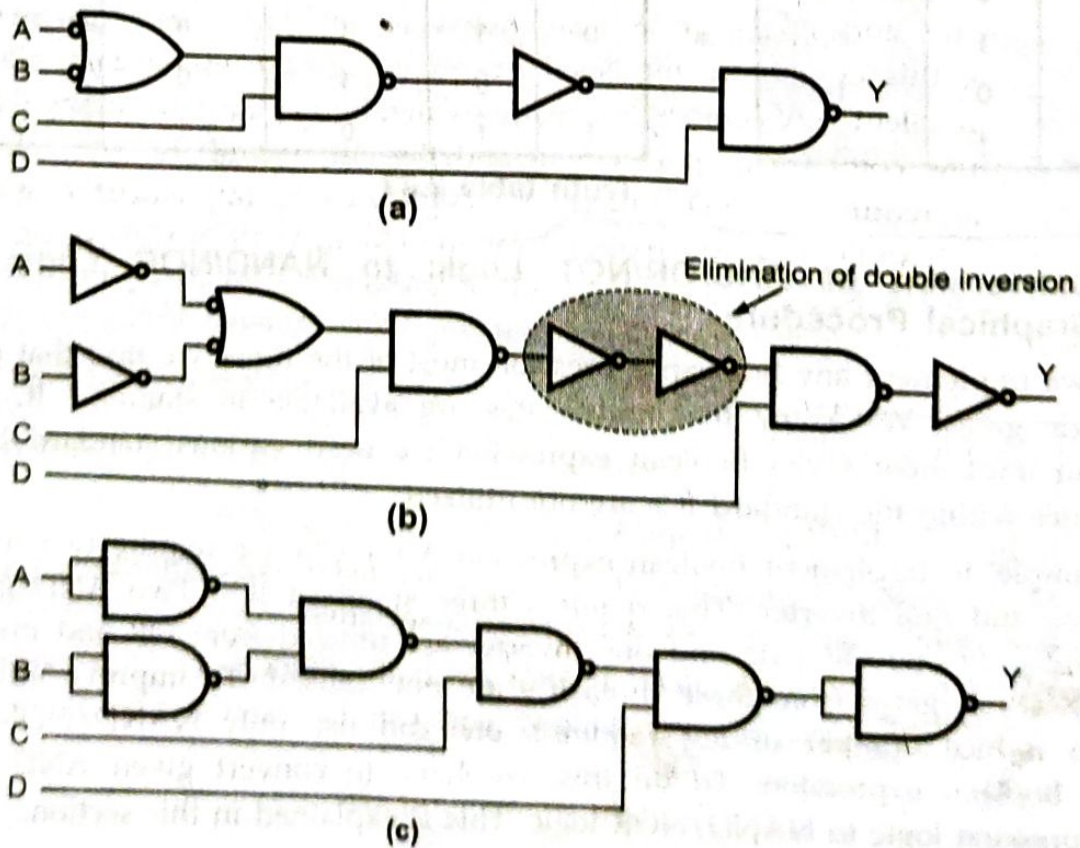


Fig. 2.27

NOR Circuit

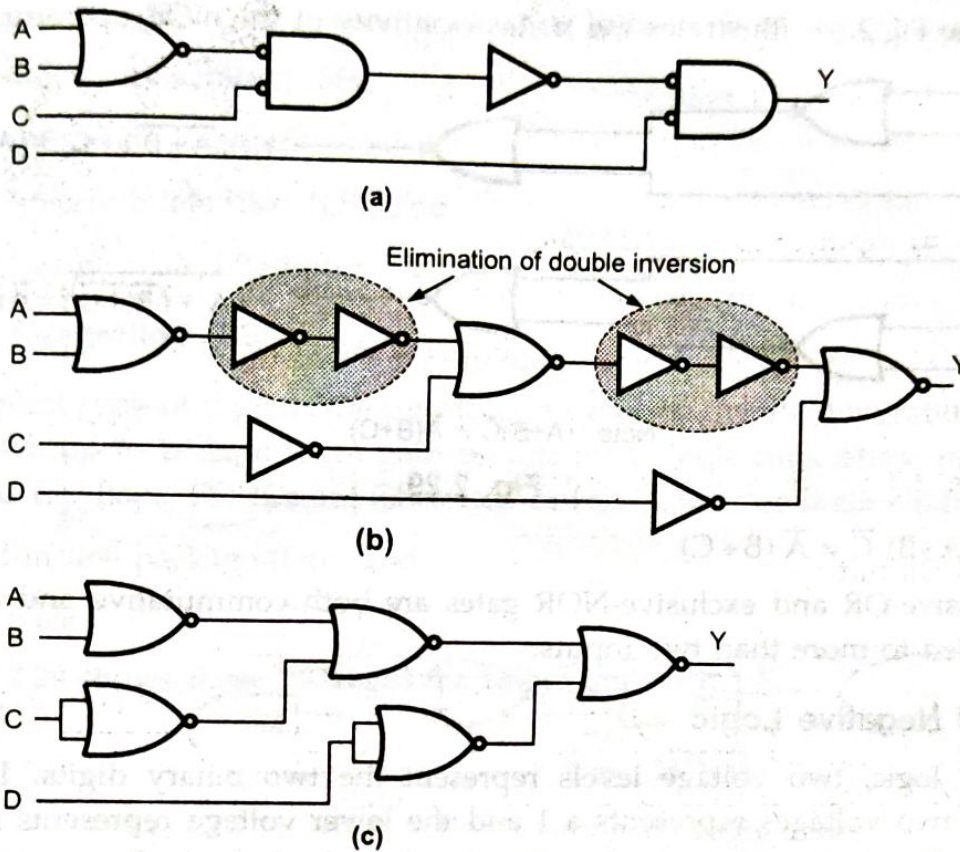


Fig. 2.28

The straight forward method for implementing the given expression uses two AND gates, one OR gate and one inverter, as shown in the Fig. 2.26. Now we will see implementation of this circuit accomplished by replacing each AND gate, OR gate and inverter by its equivalent NAND gate using steps mentioned earlier. This is illustrated in Fig. 2.27. If we now compare original circuit and the circuit with NAND gates, we find that three ICs are required (AND, OR, and INVERTER) to implement original circuit, whereas only two NAND ICs are required for the circuit with NAND gates. (With two NAND ICs we have 8 NAND gates and only 6 NANDs are required to implement the logic circuits).

Applying similar steps for NOR gate we get NOR circuit as shown in Fig. 2.28. Here, we need only two ICs of NOR gate.

2.7.11 Extension to Multiple Inputs

Except inverter and buffer, the gate can be extended to have multiple inputs if the binary operation it represents is commutative and associative.

AND and OR functions in boolean algebra are both commutative and associative and hence their gate inputs can be interchanged and can be extended to more than two variables.

$$= \overline{A}BC + ACC\overline{C} + \overline{A}B\overline{B} + \overline{A}B\overline{C} + B\overline{B}C + BCC\overline{C}$$

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

Rule 8 : $[A\overline{A} = 0]$

Example 2.9 : Simplify each of the following using Demorgan's theorem

a) $\overline{(A+B)}(\overline{A+B})$ b) $\overline{\overline{A}BCD}$

Solution : a) $\overline{(A+B)}(\overline{A+B})$

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

DeMorgan's theorem 1

$$= \overline{A} \cdot \overline{B} + \overline{A} \cdot \overline{B}$$

DeMorgan's theorem 2

$$= \overline{A} \cdot B + A \cdot \overline{B}$$

Rule 9 : $[\overline{\overline{A}} = A]$

$$= A \oplus B$$

b) $\overline{\overline{\overline{A}BCD}}$

$$= \overline{\overline{ABC} + \overline{D}}$$

DeMorgan's theorem 1

$$= \overline{ABC} + \overline{D}$$

Rule 9 : $[\overline{\overline{A}} = A]$

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

DeMorgan's theorem 1

Example 2.10 : Verify the following Boolean algebraic manipulation. Justify each step with a reference to a postulate or theorem :

i) $(X + \overline{Y} + XY)(X + \overline{Y})\overline{X}Y = 0$

ii) $(AB + C + D)(\overline{C} + D)(\overline{C} + D + E) = ABC + D$

Solution : i) $(X + \overline{Y} + XY)(X + \overline{Y})(\overline{X}Y)$

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

$\because A + \overline{A}B = A + B$

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

$\because A + A = A$

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

$\because A \cdot A = A$

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

$\because A \cdot \overline{A} = 0$

$$= 0 \dots \dots \dots \text{Proved}$$

ii) $(AB + C + D)(\overline{C} + D)(\overline{C} + D + E)$

$$= (AB + C + D)(\overline{C}\overline{C} + \overline{C}D + \overline{C}E + \overline{C}D + DD + DE)$$

$$= (AB + C + D)(\overline{C} + \overline{C}D + \overline{C}E + \overline{C}D + D + DE)$$

$\because A \cdot A = A$

$$= (AB + C + D)[\overline{C} + D(\overline{C} + \overline{C} + 1 + E) + \overline{C}E]$$

$$= (AB + C + D)(\overline{C} + D + \overline{C}E)$$

$\because A + 1 = A$

$$\begin{aligned}
 &= (AB+C+D) [\bar{C}+(1+E)+D] \\
 &= (AB+C+D) (\bar{C}+D) && \because A+1 = A \\
 &= AB\bar{C}+ABD+C\bar{C}+CD+\bar{C}D+D \\
 &= AB\bar{C}+ABD+CD+\bar{C}D+D && \because A\bar{A} = 0 \\
 &= AB\bar{C}+D(AB+C+\bar{C}+1) \\
 &= AB\bar{C}+D \quad \dots \text{Proved} && \because A+1 = A
 \end{aligned}$$

► **Example 2.11 :** Simplify the following expression
 $a+a\bar{b}+a\bar{b}\bar{c}+a\bar{b}\bar{c}\bar{d}+\dots$

Solution : $a+a\bar{b}+a\bar{b}\bar{c}+a\bar{b}\bar{c}\bar{d}+\dots$

$$\begin{aligned}
 &= a[1+\bar{b}+\bar{b}\bar{c}+\bar{b}\bar{c}\bar{d}+\dots] \\
 &= a && \because A+1 = 1
 \end{aligned}$$

► **Example 2.12 :** Prove that
 $(a+b)(\bar{a}+c)(b+c) = (a+b)(\bar{a}+c)$

Solution : $(a+b)(\bar{a}+c)(b+c)$

$$\begin{aligned}
 &= (a\bar{a}+ac+\bar{a}b+bc)(b+c) \\
 &= (ac+\bar{a}b+bc)(b+c) && \because A\bar{A} = 0 \\
 &= abc+acc+\bar{a}bb+\bar{a}bc+bbc+bcc \\
 &= abc+ac+\bar{a}b+\bar{a}bc+bc+bc && \because AA = A \\
 &= ac(b+1)+\bar{a}b+bc(\bar{a}+1+1) \\
 &= ac+\bar{a}b+bc && \because A+1 = 1 \\
 &= ac+\bar{a}b+bc+a\bar{a} && \because a\bar{a} = 0 \\
 &= c(a+b)+\bar{a}(a+b) \\
 &= (a+b)(\bar{a}+c) \quad \dots \text{Proved.}
 \end{aligned}$$

► **Example 2.13 :** Prove from fundamentals the following expressions :

- i) $xy + \bar{x} + yz = \bar{x}z + xy$
 ii) $\bar{x}\bar{y}z + \bar{x}yz + x\bar{y} = x\bar{y} = \bar{x}z + x\bar{y}$

Solution : i) $xy + \bar{x}z + yz$

$$\begin{aligned}
 &= xy + \bar{x}z + yz(x+\bar{x}) \\
 &= xy + \bar{x}z + xyz + \bar{x}yz \\
 &= xy(1+z) + \bar{x}z(1+y) \\
 &= xy + \bar{x}z \quad \dots \text{Proved.}
 \end{aligned}$$

Example 1.47. Add $33 \frac{3}{8}$ and $22 \frac{5}{8}$ in binary.

Solution.

$$\begin{aligned} 33 \frac{3}{8} &= 10001.011 \\ 22 \frac{5}{8} &= 10110.101 \\ 111000.000 &= (56)_{10} \\ &= (111000)_2 \end{aligned}$$

Example 1.48. Add 101.11 , 1101.01 and 10000.001 .

Solution.

$$\begin{aligned} 101.11 &= (5.75)_{10} \\ 1101.01 &= (13.25)_{10} \\ 10000.001 &= (16.125)_{10} \\ \hline 100011.001 &= (35.125)_{10} \\ &= (100011.001)_2 \end{aligned}$$

Example 1.49. Add 4.25 , 7.75 and 8.0 in binary.

Solution.

$$\begin{aligned} 4.25 &= 100.01 \\ 7.75 &= 111.11 \\ 8.0 &= 1000.00 \\ \hline (20.00)_{10} &= 10100.00 \\ &= (10100)_2 \end{aligned}$$

1.8.2 Binary Subtraction

Binary subtraction can be carried out in either of the two different ways.

- (1) Direct subtraction.
- (2) Complement subtraction.

1.8.2.1 Direct subtraction

The rules for performing the binary subtraction are given in the Table 1.6.

Table 1.6. Binary subtraction

A (Minuend)	B (Subtrahend)	Difference D	Borrow B
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

Example 1.50. Subtract 10101 from 11011 .

Solution.

$$\begin{aligned} 11011 &= 27 \\ - 10101 &= 21 \\ \hline 00110 &= 6 \\ &= (110)_2 \end{aligned}$$

Example 1.51. Use binary arithmetic to subtract 10 from (a) 16 (b) 22.

Solution.

$$(a) \quad \begin{array}{r} 10000 \\ - 1010 \\ \hline 00110 \end{array} = \frac{16}{6}$$

⇒ $(110)_2$

$$(b) \quad \begin{array}{r} 10110 \\ - 1010 \\ \hline 01100 \end{array} = \frac{22}{12}$$

⇒ $(1100)_2$

Example 1.52. Subtract the following by using binary arithmetic.

(a) 7.25 from 9.50 (b) 6.25 from 10.75

Solution. (a)
$$\begin{array}{r} 1001.10 \\ - 0111.01 \\ \hline 0010.01 \end{array} = \frac{9.50}{2.25}$$

⇒ $(10.01)_2$

(b)
$$\begin{array}{r} 1010.11 \\ - 0110.01 \\ \hline 0100.10 \end{array} = \frac{10.75}{4.50}$$

⇒ $(100.10)_2$

Signed-magnitude representation or signed binary numbers

Until now, we have considered only unsigned numbers, numbers without any positive or negative sign. These unsigned numbers represent only magnitude. To perform the subtraction, representation of negative number is needed. In the decimal system, a + sign is used to denote positive numbers, and - sign is used to denote negative numbers. This type of representation of numbers is known as 'signed numbers'. To indicate this sign we must use a symbol, and in a digital circuit either 0 or 1, generally. 0 is used as Most Significant Bit (MSB) to represent a positive (+) number and 1 is used as Most Significant Bit (MSB) to represent a negative (-) number.

These numbers are represented by the signed magnitude format. Fig. 1.7 shows the signed magnitude format for 8-bit signed number.

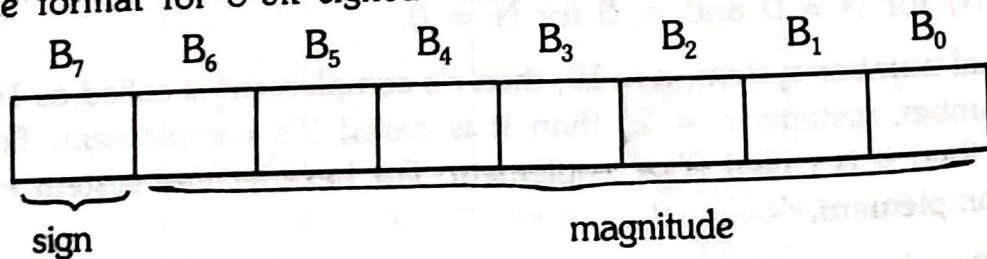


Fig. 1.7 Signed magnitude representation.

Here, the most significant bit (MSB) represents sign of the number. If MSB is 1, number is negative and if MSB is 0, number is positive. The remaining (seven) bits represent magnitude of the number.

Some examples for signed-magnitude numbers.

- Example.**
- (1) 01000100 is a $(+68)_{10}$
 - (2) 11000100 is a $(-68)_{10}$
 - (3) 10001110 is a $(-14)_{10}$
 - (4) 00011001 is a $(+25)_{10}$

In case of unsigned 8-bit binary numbers, the decimal range is 0 to 255. For signed magnitude 8-bit binary numbers, the largest magnitude is reduced from 255 to 127. Sign-magnitude numbers require much hardware for addition and subtraction.

1.8.2.2 Complement subtraction

Complements are used in digital computers for simplifying the subtraction operation and for logical manipulation.

There are two types of complements for each 'base- r ' system.

(1) radix (r) complement (or) r 's complement

(or)

(2) Diminished radix complement ($r-1$)'s complement.

(1) **$(r-1)$'s complement.** Given positive number 'N' in base- r having an integer part of 'n' digits and fractional part of 'm' digits, then the $(r-1)$'s complement of 'N' is defined as

$$(r^n - r^m - N)$$

for only integer part i.e., $m = 0$

The $(r-1)$'s complement of N

$$= [(r^n - 1) - N]$$

For decimal number system, $r = 10$, then $(r-1)$'s i.e., $(10-1)$'s complement is called as 9's complement and for binary number system, $r = 2$, then $(r-1)$'s i.e., $(2-1)$'s complement is called 1's complement. For octal system, $r = 8$, then $(r-1)$'s complement is called 7's complement, for hex system, $r = 16$, then $(r-1)$'s complement i.e., $(16-1)$'s complement is called 15's complement.

(2) **Radix complement (or) r 's complement.** For a given number N, with base 'r', with an integer part of 'n' digits and fractional part of 'm' digits, then the r 's complement of N is $(r^n - N)$ for $N \neq 0$ and $= 0$ for $N = 0$.

For decimal number system, $r = 10$, then r 's complement is called as 10's complement. For binary number systems $r = 2$, then it is called 2's complement. For octal number system $r = 8$ then it is called 8's complement. For hexadecimal system $r = 16$ then it is called 16's complement.

The 1's complement and 2's complement of a binary number represent a negative number and are more useful in binary subtraction.

Note: (1) r 's complement of a number exists for any base 'r' if $r > 1$ and an integer.

(2) r 's complement is obtained by adding 1 to the $(r-1)$'s complement.

$$\text{since } r^n - N = [(r^n - 1) - N] + 1$$

(3) Complement of complement = original number.

$$r^n - N = r^n - (r^n - N) = N$$

or

$$(r^n - 1 - N) = r^n - 1 - (r^n - 1 - N) = N$$

1's complement system (representation)

The 1's complement of any binary number is performed by simply changing all 1's into 0's and 0's into 1's. i.e., each bit is replaced by its complement (complement of 1 is 0 and complement of 0 is 1).

So, 1's complement of a number (N) = $(2^n - 2^{-m} - N)$

i.e., each bit is subtracted from binary '1'.

i.e.,

$$1 - 0 = 1 \text{ and}$$

$$1 - 1 = 0$$

which causes the bit change, from 0 to 1 and from 1 to 0.

Example 1.53. Find the 1's complement of $(1011)_2$.

Solution.

given number = 1011, $n = 4$, $m = 0$

$$\begin{aligned} \text{1's complement of the number} &= (2^4 - 2^0 - 1011) = 100 \\ &= 16 - 1 - 1011 \\ &= 15 - 1011 \\ &= 1111 - 1011 \Rightarrow 0100 \end{aligned}$$

Example 1.54. Find the 1's complement of 1001.1.

Solution.

given number = 1001.1, $n = 4$, $m = 1$

$$\begin{aligned} \text{1's complement of the number} &= (2^4 - 2^{-1} - N) = 10000.1 - 1001.1 \\ &\Rightarrow (0110.0)_2 \end{aligned}$$

Some more examples

<u>binary number</u>	<u>1's complement</u>
010001	101110
11110101	00001010
1010.0101	0101.1010
1100.001	0011.110
10010.000	01101.111
10010.10	01101.01
0000.11	1111.00
0000.00	1111.11

1's complement subtraction method

Subtraction between the binary numbers can be performed by using 1's complement method. Here, instead of subtracting a number, we add the 1's complement of the number

In the 1's complement subtraction, we have two cases.

Case (i) Subtraction of smaller number from larger number.

Procedure :

- (1) Determine 1's complement of the smaller number (subtrahend).
- (2) Add 1's complement to the larger number (minuend) and the result is a positive number.
- (3) Carry will be generated after the addition which is called "End-Around Carry" (EAC) Remove the EAC and add it to the result.

Example 1.55. Subtract 101011 from 111101 using 1's complement method.

Solution.

$$\begin{array}{r}
 111101 \\
 \text{1's complement} \quad 010100 \\
 \text{of 101011 is EAC } \oplus 010001 \\
 \hline
 1 \quad \text{Add end around carry} \\
 010010 \\
 \hline
 \Rightarrow (10010)_2 = (18)_{10}
 \end{array}$$

Example 1.56. Perform the binary subtraction for the following using 1's complement method.

- (a) $28 - 8$ (b) $30 - 25$ (c) $25.5 - 12.25$ (d) $10.625 - 8.75$.

Solution. (a)

$$\begin{array}{l}
 28_{10} = 11100 \\
 8_{10} = 01000 \\
 \text{1's complement of } 8 = -8 \\
 = 10111
 \end{array}$$

$$\begin{array}{r}
 \therefore \quad 11100 \text{ - Minuend} \\
 \quad 10111 \text{ - 1's complement of subtrahend} \\
 \hline
 \text{EAC } \oplus 10011 \\
 \quad + 1 \quad \text{Add EAC} \\
 \hline
 10100
 \end{array}$$

$$= (10100)_2 = (20)_{10}$$

(b)

$$\begin{array}{l}
 30_{10} = 11110 \\
 \text{Minuend} = 11110 \\
 25_{10} = 11001
 \end{array}$$

1's complement of subtrahend = 00110

$$\begin{array}{r}
 \text{EAC } \oplus 00100 \\
 \quad + 1 \quad \text{Add EAC} \\
 \hline
 00101
 \end{array}$$

$$\Rightarrow (101)_2 = (5)_{10}$$

(c)

$$\begin{array}{l}
 25.5 = 11001.1 \\
 \text{Binary of } 12.25 = 01100.011 \\
 \text{1's complement of } 12.25 = 10011.100
 \end{array}$$

\therefore

$$\begin{array}{r}
 11001.10 \\
 10011.10 \\
 \hline
 \text{EAC } \oplus 01101.00 \\
 \quad 1 \quad \text{Add EAC} \\
 \hline
 1101.01
 \end{array}$$

\Rightarrow

$$(1101.01)_2 = (13.25)_{10}$$

(d) Minuend 10.625 = 1010.101

Subtrahend 8.75 = 1000.110

1's complement of 8.75 = 0111.001

∴
$$\begin{array}{r} 1010.101 \\ 0111.001 \\ \hline \end{array}$$

EAC ①0001.110

$$\begin{array}{r} 1 \\ \hline 1.111 \end{array} \quad \text{Add EAC}$$

⇒ $(1.111)_2 = (1.875)_2$

Case (ii) Subtraction of larger number from smaller number.

Procedure:

(1) Determine the 1's complement of the larger number (subtrahend)

(2) Add 1's complement to the smaller number (minuend)

(3) After addition, no carry will be generated but answer is in 1's complement form (negative number). To get the answer in true form, take the 1's complement of it and assign negative sign to the answer.

Example 1.57. Perform the binary subtraction operation using 1's complement method for the following.

(a) 43 - 57

(b) 8 - 10

(c) 8.75 - 10.625

(d) $\frac{3}{4} - \frac{13}{16}$

(e) $\frac{5}{8} - \frac{7}{8}$

(f) 11.125 - 16.875

Solution.

(a) $(57)_{10} = 111001$

$(43)_{10} = 101011$

Minuend 101011

1's complement of subtrahend $\frac{000110}{110001}$

The result has no carry, so the answer is in 1's complement form

1's complement of (110001) is -(001110)

answer in true form = $(-14)_{10}$

(b) $(8)_{10} = 1000$

Minuend = 1000

$(10)_{10} = 1010$ 1's complement of subtrahend = $\frac{0101}{1101}$

No carry, so the answer is in 1's complement form.

∴ True answer = $-(0010)_2$
= $(-2)_{10}$

$$(c) \quad (8.75)_{10} = 1000.110$$

$$(10.625)_{10} = 1010.101$$

$$\text{Minuend} = 1000.110$$

$$1\text{'s complement of subtrahend} = \underline{0101.010}$$

$$1110.000$$

There is no carry, so the answer is in the 1's complement form

$$\therefore \text{True answer} = - (1.111)_2$$

$$= - (1.875)_{10}$$

$$(d) \quad \frac{3}{4} = 0.1100 \quad \text{Minuend} = 0.1100$$

$$\frac{13}{16} = 0.1101 \quad 1\text{'s complement of subtrahend} = \underline{0.0010}$$

$$0.1110$$

There is no carry, so the answer is in 1's complement form.

$$\therefore \text{True answer} = - (0.0001)_2$$

$$= \left(-\frac{1}{16} \right)_{10}$$

$$(e) \quad \frac{5}{8} = 0.101 \quad \text{Minuend} = 0.101$$

$$\frac{7}{8} = 0.111 \quad 1\text{'s complement of } \frac{7}{8} \text{ subtrahend} = \underline{0.000}$$

$$0.101$$

There is no carry, so the answer is in 1's complement form.

$$\therefore \text{True answer} = - (0.010)_2 = (-2/8)_{10}$$

$$(f) \quad 16.875 - 11.125$$

$$16.875 = 10000.111 \quad \text{Minuend} = 01011.001$$

$$11.125 = 01011.001 \quad 1\text{'s complement of subtrahend} = \underline{01111.000}$$

$$01010.001$$

There is no carry, so the answer is in 1's complement form.

$$\therefore \text{True answer} = - (0101.110)_2$$

$$= - (5.75)_{10}$$

Advantages of 1's complement subtraction

1. The 1's complement subtraction can be accomplished with a binary adder. Therefore, this method is useful in arithmetic logic circuits.
2. It is very easy to find the 1's complement of a number i.e., to get the $(-N)$, where N is a number.

Disadvantage :

1. In 1's complement method, hardware implementation is difficult and it gives the concept of negative zero.
2. It also gives one value less to represent negative values.

2's complement system (representation)

The 2's complement of any binary number is determined by adding 1 to 1's complement of that number. If N is a binary word and \bar{N} is its 1's complement. Then the 2's complement of the binary word N is $\bar{N}+1$.

- ★ 2's complement of number = 1's complement of a number +1.
- ★ 2's complement form is used to represent -ve numbers.
- ★ 2's complement of one (1) is 1.
- ★ 2's complement of zero is 10.

Example 1.58. Find the 2's complement of the following binary numbers.

(a) $(1010)_2$

(b) 1111

(c) 11.01

Solution. (a) Given binary number = 1010

1's complement of 1010 = 0101

By adding 1 to 1's complement = $\frac{+1}{0110}$

2's complement of 1010 = $(0110)_2$

(b) Given binary number = 1111

1's of 1111 is = 0000

By adding 1 to 1's complement = $\frac{1}{0001}$

2's complement of 1111 = $(0001)_2$

(c) Given binary number = 11.01

1's complement of 11.01 = 00.10

By adding 1 to 1's complement = $\frac{1}{00.11}$

2's complement of 11.01 is = $(00.11)_2$

Note : From the above examples, we can conclude that the 2's complement of a number can be obtained by leaving all least significant 0's and the first one (1) unchanged, and replacing all 1's with 0's and 0's with 1's in all other higher significant bits.

Example 1.59. Convert the following decimal numbers into binary and find their 2's complements.

(a) 40

(b) 18

(c) 16.5

Solution. (a) The binary equivalent of 40 is $(101000)_2$

$$1\text{'s complement of } 40 = (010111)_2$$

$$\text{By adding 1 to its 1's complement} = \frac{\quad + 1}{011000}$$

$$2\text{'s complement of } 40 = (011000)_2$$

(b) The binary equivalent of 18 = $(10010)_2$

$$2\text{'s complement of } 18 = (01110)_2$$

(c) The binary equivalent of 16.5 = 10000.1

$$2\text{'s complement of } 16.5 = (01111.1)_2$$

2's complement subtraction method

Like 1's complement subtraction in 2's complement subtraction also, the subtraction is accomplished by addition only. For subtraction of two numbers we have two cases.

Case (i) : Subtraction of a smaller number from larger number.

Procedure :

1. Determine the 2's complement of the small number (subtrahend)
2. Add 2's complement to the larger number (minuend)
3. Discard the carry generated.

Note. Always a carry is generated in this case.

Example 1.60. Perform the following subtraction operations using 2's complement method

$$(a) (111001)_2 - (101011)_2 \quad (b) (1111)_2 - (1010)_2 \quad (c) (112)_{10} - (65)_{10}$$

$$(d) 22-7 \quad (e) (100.5)_{10} - (50.75)_{10}$$

Solution. (a) Minuend $111001 = 57$

$$2\text{'s complement of subtrahend } 010101 \quad 43$$

$$\text{discard carry } \textcircled{1}001110 = 14$$

$$\therefore 111001 - 101011 \Rightarrow (1110)_2 = (14)_{10}$$

(b) Minuend = 1111

$$2\text{'s complement of subtrahend} = 0110$$

$$\text{discard carry } \textcircled{1}0101$$

$$\therefore 1111 - 1010 \Rightarrow (101)_2 = (5)_{10}$$

(c) $(112)_{10} - (65)_{10}$

$$\text{binary equivalent of } 112 = 1110000$$

$$\text{binary equivalent of } 65 = 1000001$$

$$\text{Minuend} = 1110000$$

$$2\text{'s complement of subtrahend} = 0111111$$

$$\text{discard carry } \textcircled{1}0101111$$

$$112 - 65 \Rightarrow (101111)_2 = (47)_{10}$$

(d) $22 = 10110$ and $7 = 00111$

Minuend = 10110

2's complement of subtrahend = 11001

discard carry $\textcircled{0}1111$

$\therefore 22 - 7 \Rightarrow (1111)_2 = (15)_{10}$

(e) $(100.5)_{10} = (1100100.10)_2$

$(50.75)_{10} = (0110010.11)_2$

Minuend = 1100100.10

2's complement of subtrahend = 1001101.01

discard carry $\textcircled{0}110001.11$

$\therefore (100.5)_{10} - (50.75)_{10} = (110001.11)_2 = (49.75)_{10}$

Case (ii) : Subtraction of a larger number from smaller number.

Procedure :

1. Determine the 2's complement of the larger number (subtrahend)
2. Add the 2's complement to small number (minuend)
3. Answer is in 2's complement form. To get the answer in the true form, take the 2's complement and assign negative sign to the answer.

Note : Here, no carry is generated.

Example 1.61. Perform the following subtraction operations using 2's complement method.

(a) $7 - 22$

(b) $13 - 27$

(c) $10 - 28$

(d) $16.5 - 24.75$

Solution. (a)

$(7)_{10} = 00111$

Minuend = 00111

$(22)_{10} = 10110$ 2's complement of subtrahend = 01010

No carry

$\begin{array}{r} 01001 \\ + 01010 \\ \hline 10011 \end{array}$

10001

\therefore True answer = The 2's complement of (1001) is $-(1111)_2 = (-15)_{10}$

(b) $(13)_{10} = 01101$

Minuend = 01101

$(27)_{10} = 11011$ 2's complement of subtrahend = 00101

No carry

10010

\therefore True answer = The 2's complement of (10010) is $-(1110)_2 = -(14)_{10}$

(c) $(10)_{10} = 01010$

Minuend = 01010

$(28)_{10} = 11100$ 2's complement of subtrahend = 00100

No carry

1110

\therefore True answer = The 2's complement of (01110) = $(-10010)_2 = -(18)_{10}$

(d) $16.5 = 10000.10$

Minuend = 10000.10

$24.75 = 11000.11$ 2's complement of subtrahend = 00111.01

10111.11

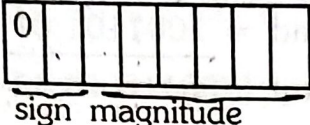
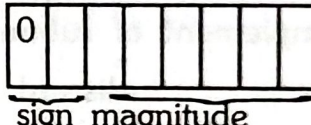
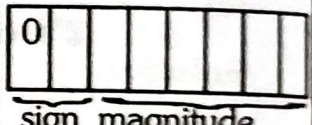
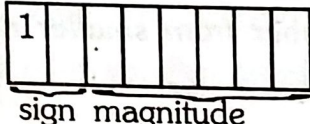
True answer = 2's complement of 10111.11 = $-(01000.01)_2 = -(8.25)_{10}$

Advantage : Reduction in hardware.

Note : In 2's complement subtraction, carry indicates whether the answer is positive or negative. When the carry is 1, the answer is positive. When there is no carry, the answer is negative and is the 2's complement of the actual magnitude.

The comparison of the three signed-number systems is shown in Table 1.7.

Table 1.7. Comparison of three signed-number systems

Number system → parameters ↓	Signed magnitude	Signed 1's complement	Signed 2's complement
1. Representation of a positive number			
2. Representation of negative number		$2^m - N - 1$	$2^m - N$
3. Form of zero	Plus zero = 00...0 Minus zero = 10...0	Plus zero = 00...0 Minus zero = 11...1	Plus zero = 00...00 No minus zero
4. Negative numbers in complement form	No	Yes	Yes
5. Sign bit involved in addition or subtraction	No	Yes	Yes
6. Use end-around carry	Yes	Yes	No
7. Method for detecting over flow in adding two positive numbers	Carry from the mag. bits	Negative sum	Negative sum
8. Method for detecting over flow in adding two negative numbers	Carry from the mag. bits	Positive sum	Positive sum

1.8.3 Binary Multiplication

The process of binary multiplication is almost similar to the process of decimal multiplication. *Binary multiplication* is just a matter of shifting and adding. The rules for binary multiplication are given in Table 1.8.

Table 1.8. Rules for binary multiplication

Multiplicand	Multiplier	Product
A	B	P
0	0	0
0	1	0
1	0	0
1	1	1

Note : The product of two binary numbers is 1, only if both the bits are 1, otherwise it is 0.

To handle large numbers, number with several bits: Binary multiplication involves the partial products, shifting each successive partial product left to one place (bit), and then adding all the partial products.

Example 1.62. Perform the binary multiplication for the following

(a) 7×5

(b) 22×6

(c) 27×21

(d) $\frac{3}{8} \times \frac{1}{4}$

(e) 4.75×3.625

Solution. (a)

$(7)_{10} = 111$

Multiplicand = 111

$(5)_{10} = 101$

Multiplier = 101

$$\begin{array}{r} 111 \\ \times 101 \\ \hline 111 \quad \text{Partial product 1} \\ 000 \quad \text{Partial product 2} \\ \overline{111} \quad \text{Partial product 3} \\ 100011 \quad \text{Final product} \end{array}$$

$(100011)_2 = (35)_{10}$

\Rightarrow

(b)

$(22)_{10} = 10110$

$(6)_{10} = \times 110$

$\frac{132}{132} \quad \frac{00000}{00000} \Rightarrow (10000100)_2 = (132)_{10}$

10110

10110

$\frac{10000100}{10000100} = \text{Final product} \therefore (10000100)_2 = (132)_{10}$

(c)

27 = 11011

21 = \times 10101

11011

00000

11011

00000

11011

$\frac{1000110111}{1000110111} \text{ Final product} \Rightarrow (1000110111)_2 = (567)_{10}$

(d)

$\frac{3}{8} = 0.011$ 0.011

$\frac{1}{4} = 0.01$ $\times \frac{0.01}{0011}$

0000

0.00011

\Rightarrow

$(0.00011)_2 \Rightarrow \left(\frac{3}{32}\right)_{10}$

$$\begin{array}{r}
 \text{(e)} \quad 4.75 = 100.110 \\
 3.625 = \times 11.101 \\
 \hline
 100110 \\
 000000 \\
 100110 \\
 100110 \\
 100110 \\
 \hline
 10001.001110
 \end{array}$$

$$= (10001.00111)_2 \Rightarrow (17.21875)_{10} = \left(17\frac{7}{32}\right)_{10}$$

1.8.4 Binary Division

Binary division is carried out in a similar manner to division using decimal numbers i.e., comparison and subtraction till the final remainder is obtained.

Suppose the divisor has two bits, then, if the first three digits of the dividend are equal to, or larger than the divisor, put 1 in the quotient and subtract the divisor from the first two bits. Next, bring down the next bit of the dividend to the LSB of the remainder. If the remainder is larger than the divisor, put another 1 into the quotient. If it is smaller, insert 0. If 1 is inserted, subtract the divisor from the remainder and bring down the next bit of the dividend and repeat the procedure.

The rules for division by binary bits is as follows.

$$0 \div 1 = 0$$

$$1 \div 1 = 1$$

Note : Division by zero is not permitted.

Let us consider the different examples for binary division.

Example 1.63. (a) Divide 50 by 5 using binary division.

(b) Divide 11001 by 100.

(c) Divide 10.5 by 5.25 by using binary division.

(d) Divide 1100.10 by 010.

Solution. (a) Divisor = 5 = 101

Dividend = 50 = 110010

The quotient = 101 = $(5)_{10}$

$$\begin{array}{r}
 101 \leftarrow \text{quotient} \\
 101 \overline{) 110010} \\
 101 \\
 \hline
 00101 \\
 101 \\
 \hline
 000 \leftarrow \text{remainder}
 \end{array}$$

(b)
$$\begin{array}{r} 110.01 \\ 100 \overline{) 11001} \\ \underline{100} \\ 0100 \\ \underline{100} \\ 00100 \\ \underline{100} \\ 000 \end{array}$$
 Hence, result = $110.01 = (6.25)_{10}$

(c) Dividend = $10.5 = 1010.10$
 Divisor = $5.25 = 101.01$

$$\begin{array}{r} 10 \\ 101.01 \overline{) 1010.10} \\ \underline{101} \\ 0000.00 \\ \underline{000.00} \\ 0000 \end{array}$$

quotient = $(10)_2 = (2)_{10}$

(d)
$$\begin{array}{r} 110.01 \\ 10 \overline{) 1100.00} \\ \underline{10} \\ 10 \\ \underline{10} \\ 00 \\ \underline{00} \\ 01 \\ \underline{00} \\ 10 \\ \underline{10} \\ 00 \end{array}$$
 result = $(110.01)_2 = (6.25)_{10}$

1.9. THE 9'S AND 10'S COMPLEMENTS

In any number system, the complement of a digit a , is denoted by a' or \bar{a} is defined as

$$\bar{a} = (b - 1) - a$$

i.e., the complement \bar{a} is the difference between the largest digit in base b and the digit a .

9's complement : The $(r - 1)$'s complement of a decimal system ($r = 10$), is called 9's complement. The 9's complement of a number $N = \bar{N} = [(10^n - 10^{-m}) - N]$

1.12 BINARY CODES

The electronic digital systems like computers, microprocessors, ...etc., are required to handle data (represent and manipulate) which may be consisting of numbers, alphabets or special characters. But the digital systems use signals that have two distinct values and circuit elements that have two stable states. Hence the numerals, alphabets, special characters and control functions are to be converted into binary format.

The process of conversion into binary format is known as '*binary coding*'. The combination of binary bits that represent numbers, (may be decimal numbers, octal, hexadecimal number etc.), alphabets, symbols or control functions are called '*binary codes (or) digital codes*'. When decimal number is represented by its equivalent binary, we call it "straight binary coding". The codes must be in binary because computers can only process 1's and 0's. Representation of a group of 2^n distinct elements in a binary code requires a minimum of n bits. This is because it is possible to arrange n bits in 2^n distinct ways.

For example, a group of four distinct quantities can be represented by a two bit code, with each quantity assigned one of the following bit combinations : 00, 01, 10, 11. A group of eight elements requires a three-bit code, with each element assigned to one and only one of the following: 000, 001, 010, 011, 100, 101, 110, 111.

The examples show that the distinct bit combinations of an n -bit code can be found by counting in binary from 0 to $(2^n - 1)$. The minimum number of bits required to code 2^n distinct quantities is n , there is no maximum number of bits that may be used for a binary code.

Basically, binary codes are classified as numeric or alphanumeric. *Numeric codes* are used to represent numbers. On the other hand, *alphanumeric codes* are used to represent characters: alphabetic letters and numerals.

In these codes, a numeral is treated simply as another symbol rather than of a number or numeric value.

1.12.1 Classification of Binary Codes

The binary codes are classified in different ways as (1) Weighted Codes (2) Non-weighted Codes (3) Reflective Codes (4) Sequential Codes (5) Alphanumeric Codes (6) Error Detecting and Correcting Codes.

(1) *Weighted Codes*. The main characteristic of a *weighted code* is, each binary bit is assigned by a "*weight*" and values depend on the position of the binary bit.

The sum of the weights of these binary bits, whose value is 1 is equal to the decimal digit which they represent.

In other words, if $w_1, w_2, w_3,$ and w_4 are the weights of the binary digits, and x_1, x_2, x_3 and x_4 are the corresponding bit values, then the decimal digit

$N = w_4 x_4 + w_3 x_3 + w_2 x_2 + w_1 x_1$ is represented by the binary sequence $x_4 x_3 x_2 x_1$. A sequence of binary bits which represents a decimal digit is called a "*code word*". Thus $x_4 x_3 x_2 x_1$ is a code word of N . Examples for these codes are: BCD, 8421, 6421, 4221, 5211, 3321, $84\bar{2}\bar{1}$ etc.

(2) *Non-weighted Codes*. Non-weighted codes or un-weighted codes are those codes in which the digit value does not depend upon their position i.e., each digit position within the

number is not assigned fixed value. Most commonly used non-weighted codes are : unweighted BCD code, Excess-3 code and gray code.

Comparison between weighted and non-weighted codes is shown in Table 1.9.

Table 1.9. Comparison between weighted and non-weighted codes

Sl. No.	Particulars	Weighted codes	Non-weighted codes
1.	Weight	In this code, each bit position is assigned a specific weight.	In this code, no specific weights are assigned to the bit positions.
2.	Value	Each bit position represents a fixed value.	Each position within the binary number is not assigned any fixed value.
3.	Examples	BCD, 4221, 5211, 8421, 6421, $84\bar{2}\bar{1}$.	Examples. Ex-3 code, Gray code.
4.	Applications	These codes are used in (a) Data manipulation during arithmetic operations. (b) For input/output operations in digital circuits. (c) To represent the decimal digits in calculators, volt meters etc.	These codes are used (a) To perform certain arithmetic operations. (b) Shift position encodes. (c) Used for error detecting purpose.

(3) *Reflective Codes (or) Self Complimenting Codes.* A code is said to be a reflective code, if the code word of the 9's complement of N i.e., $9 - N$ can be obtained from the code word of N by interchanging all the 1's as 0's and 0's as 1's. i.e., the code word for 9 is the complement for the code 0, 8 for 1, 7 for 2, 6 for 3 and 5 for 4. Note that 4221, 5211, 6, 4, 2, -3 and Ex-3 codes are reflective, whereas 8421 code is not.

Example. In $642\bar{3}$, the decimal 5 is represented by = 1011

$9 - N = 9 - 5 = 4$ decimal 4 is represented by = 0100

i.e., 9's complement of 5 is obtained just by complementing each bit in 5.

Necessary condition. A weighted code is said to be self complementing code if the sum of all weights is equal to '9'.

Totally there are 13 possible self complementing weighted codes which include positive and negative weights.

Example. 5211, 4221, 3321, $84\bar{2}\bar{1}$ etc.

Note: Ex-3 code is a non-weighted self complementing code.

Reflective codes are used to perform 9's complement subtraction.

(4) *Sequential Codes.* In sequential codes, each succeeding code is one binary number greater than its preceding code. The examples for this code are 8421, and Ex-3 codes, whereas 4221 and 5211 are not sequential codes. These codes are greatly used in mathematic manipulation of data.

(5) **Alphanumeric Codes.** Many applications of digital computers require the handling of data consisting of not only numbers, but also of letters. To represent the letters (characters), it is necessary to formulate a binary code. An alphanumeric character set is a set of elements that includes the 10 decimal digits, the 26 letters of alphabets, and a number of special characters. Such a set contains between 36 to 64 elements if only capital letters are included, or between 64 to 128 elements if both uppercase and lowercase letters are included. In the first case, we need a binary code of six bits and in the second, we need a binary code of seven bits.

The codes which represent the alphanumeric character set are called 'alphanumeric codes'.

Most of these codes, however, also represent symbols and various instructions necessary for conveying intelligible information.

The most commonly used alphanumeric codes are :

ASCII (American Standard Code for Information Interchange)

EBCDIC (Extended Binary Coded Decimal Interchange Code) and

Hollerith code. All these codes are non-weighted codes.

(6) **Error Detecting and Correcting Codes.** When the binary information is transmitted from one circuit or system to another circuit or system, there is a chance of error occurrence. The error may occur at the transmitter, receiver or in the channel. This means a 0 may be changed as 1 and vice-versa due to the presence of different types of noise. To maintain the data integrity between transmitter and receiver, extra bit(s) are added to the data. These extra bits allow the detection and some times correction of error(s) in the data.

The data along with extra bit(s) form the code. The codes which allow only error detection are called "error detecting codes" and codes which allow error detection and correction are called 'error detecting and correcting codes'. Example for error detecting code is parity and error detection and correction or error correcting code is Hamming code.

Fig. 1.8 represents the classification of various digital codes (or) binary codes.

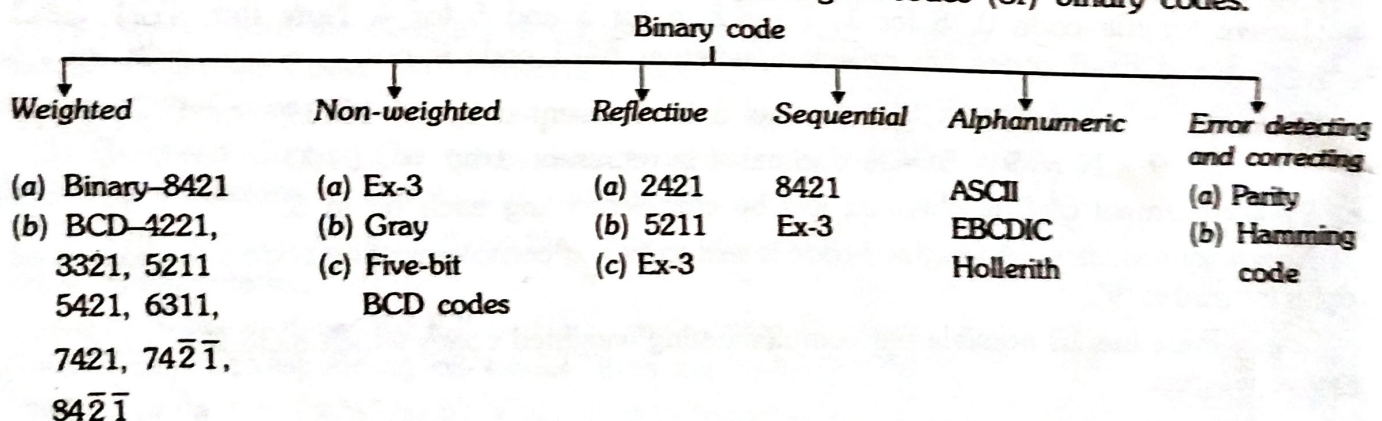


Fig. 1.8 Classification of binary codes.

1.12.2 BCD Codes or 8421 Code

BCD is an abbreviation for Binary-Coded-Decimal in which decimal digits 0 through 9 are represented by their binary equivalents using four bits i.e., BCD is a numeric code in which each decimal digit is represented by a separate binary code of four bits. The most common BCD code is 8-4-2-1. In this code, the binary weights associated with four bits are 8 4 2 1

(i.e., 2^3 , 2^2 , 2^1 , and 2^0) from left to right. This means that bit 3 has weight 8, bit 2 has weight 4, bit 1 has weight 2 and bit 0 has weight 1.

The 8421 code is a *mixed-base code*; it is binary within each group of 4 bits, but it is decimal from group to group.

Note: The four bit BCD codes use only 10 of 16 possible states (because $2^4 = 16$). There are $\frac{16!}{16! \times 10!}$ possible codes, approximately 8008 codes.

Table 1.10 shows the 4 bit 8421 BCD-code used to represent a single decimal digit.

Table 1.10. 8421 BCD code

Decimal digit	BCD code			
	8	4	2	1
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1

With four bits, 2^4 i.e., sixteen numbers can be represented. But in this code, we are using only ten. The result of six codes combinations are not used i.e., 1010 to 1111 are invalid in this BCD code.

In multi-digit coding, each decimal digit is individually coded with 8421 BCD code. For example, 78 in decimal can be encoded in 8421 BCD as 0111 1000.

Comparison between BCD and binary

In multi-digit coding of 8421 BCD numbers, we require 4-bits per decimal digit. Therefore, total 8-bits are required to encode 78 in 8421 BCD. When we represent the same number 78 in binary : 1001110, we require only 7 bits. The straight binary code takes the complete decimal number and represents it in binary: the BCD code converts each decimal digit to binary individually. This means that representing numbers 8421 BCD is less efficient than pure binary number system. The advantage of a BCD code is that it is easy for conversion to decimal and vice-versa. The disadvantage of BCD is, in this arithmetic operations are more complex than in pure binary.

Example 1.71. Convert each of the following decimal numbers into BCD.

(a) $(97)_{10}$

(b) $(63.4)_{10}$

(c) 78.216

Solution.

- (a) $(97)_{10} \rightarrow 1001 \ 0111$
 (b) $(63.4)_{10} \rightarrow 0110 \ 0011.0100$
 (c) $78.216 \rightarrow 0111 \ 1000.001000010110$

Note : If a number in any number system other than decimal is to be converted into BCD-code, first it must be converted into decimal and, from decimal to BCD. Similarly if the conversion is required from BCD to other number system, it must always be done via decimal.

Example 1.72. Convert given numbers into BCD

- (a) $(7A)_{16}$ (b) 1000010 (c) 1110110.0001 (d) $(36)_8$

Solution.

- (a) $(7A)_{16} = 7 \times 16 + 10 = 112 + 10 = 122$
 $\therefore 122 = 0001 \ 0010 \ 0010$
 (b) $1000010 = 64 + 2 = 66$
 $= 0110 \ 0110$
 (c) $1110110.0001 = 64 + 32 + 16 + 4 + 2.0625$
 $= 118.0625$
 $= 0001 \ 0001 \ 1000 . 0000 \ 0110 . 0010 \ 0101$
 (d) $(36)_8 = 24 + 6 = 30 = 0110 \ 0000$

Example 1.73. Convert the given BCD numbers into

- (a) decimal (b) binary (c) octal and (d) hexadecimal
 (i) 10001101 (ii) 10100.001 (iii) 100110.100110

Solution.

- (i) (a) $\frac{1000}{8} \frac{1001}{9} = (89)_{10}$ (b) 1011001
 (c) $(131)_8$ (d) $(59)_{16}$
 (ii) 10100.0010
 (a) $(14.2)_{10}$ (c) $(16.146)_8$
 (b) 1110.0011 (d) $(E.33)_{16}$
 (iii) 100110.100110
 (a) $(26.98)_{10}$ (c) $(32.7655)_8$
 (b) 11110.1111 (d) $(1A.FA)_{16}$

1.12.2.1 BCD addition

We can perform BCD addition provided that in each case, the sum in any four-bit column does not exceed 9. Then only the results are valid BCD numbers.

Examples 1.74. Add the following BCD numbers.

- (i) 1000 0100 and 0001 0011
- (ii) 0101 0010 and 0011 0110
- (iii) 0100 0110.010 and 01000010.001
- (iv) 1001 1000 and 0010 1000

Solution.

<p>(i) $\begin{array}{r} 10000100 \\ + \\ 00010101 \\ \hline 10011001 \end{array}$</p> <p>(iii) $\begin{array}{r} 0100\ 0110.0100 \\ + \\ 0100\ 0010.0010 \\ \hline 1000\ 1000.0110 \end{array}$</p>	<p style="text-align: right;">84 (ii) 01010010 52</p> <p style="text-align: right;">15</p> <p style="text-align: right;">99</p> <p style="text-align: right;">46.4</p> <p style="text-align: right;">42.2</p> <p style="text-align: right;">88.6</p>
<p style="text-align: right;">84</p> <p style="text-align: right;">15</p> <p style="text-align: right;">99</p> <p style="text-align: right;">46.4</p> <p style="text-align: right;">42.2</p> <p style="text-align: right;">88.6</p>	<p style="text-align: right;">52</p> <p style="text-align: right;">36</p> <p style="text-align: right;">88</p> <p style="text-align: right;">98</p> <p style="text-align: right;">28</p> <p style="text-align: right;">126</p>

Note: In the above examples (i), (ii) and (iii) are valid BCD additions, because in all cases, the four-bit sum is equal to or less than 9.

But example (iv) is an invalid BCD addition. Because here four bit sum is greater than 9 and also carry out of group is generated. Hence BCD addition can't be performed.

1.12.3 Other 4-bit BCD Codes

There are various other weighted 4-bit BCD codes, each developed to have certain properties useful for special applications. Table 1.11 shows these codes, identified by the weights assigned to their bit positions. Among those 2421, 3321 and 4221 are the self complementing codes. In the 2421 BCD code, the weights are 2-4-2-1, meaning that bit 1 and bit 3 have the same weight two (2). It is sometimes referred to as 2*-4-2-1 code, where the asterisk simply distinguishes one position with weight 2 from the other. In the above three codes, two positions have same weight, there are two possible bit patterns that could be used to represent same decimal digit, but only one of these patterns is actually assigned.

The $74\bar{2}\bar{1}$ and $84\bar{2}\bar{1}$ codes are some what different than other codes. Here $\bar{2}\bar{1}$ indicates that they are negative weights. When 1 occurs in either of the two right most positions, it means that the weight of that position is subtracted, rather than added to determine the decimal value. In the above two codes, $84\bar{2}\bar{1}$ is a self complementing code.

Table 1.11. Weighted 4-bit BCD codes

Decimal	2421	3321	4221	5211	5311	5421	6311	7421	$74\bar{2}\bar{1}$	$84\bar{2}\bar{1}$
0	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
1	0001	0001	0001	0001	0001	0001	0001	0001	0111	0111
2	0010	0010	0010	0011	0011	0010	0011	0010	0110	0110
3	0011	0011	0011	0101	0100	0011	0100	0011	0101	0101

4	0100	0101	1000	0111	0101	0100	0101	0100	0100	0100
5	1011	1010	0111	1000	1000	1000	0111	0101	1010	1010
6	1100	1100	1100	1001	1001	1001	1000	0110	1001	1010
7	1101	1101	1101	1011	1011	1010	1001	1000	1000	1001
8	1110	1110	1110	1101	1100	1011	1011	1001	1111	1000
9	1111	1111	1111	1111	1101	1100	1100	1010	1110	1111

1.12.4 Non-Weighted Codes

1.12.4.1 Excess-3 code

Excess-3 code is a modified form of BCD code. The Excess-3 code can be obtained from the natural BCD code by adding 3 to each coded number *i.e.*, the digital code which is derived by adding 3 to each decimal digit and then converting the result into four-bit binary is known as 'Excess-3 code' (XS3) (or) EX-3 code.

Here no definite weights are assigned to the four bit positions and hence this code is an unweighted code.

In this code also, we have $16(2^4)$ possible code combinations, out of these, only ten code combinations are used as shown in Table 1.12. The remaining six code combinations are invalid. These are 0000, 0001, 00010, 1101, 1110 and 1111.

Table 1.12. The excess-3 code

Decimal	BCD	Excess-3
0	0000	0011
1	0001	0100
2	0010	0101
3	0011	0110
4	0100	0111
5	0101	1000
6	0110	1001
7	0111	1010
8	1000	1011
9	1001	1100

Example 1.75. Find the Ex-3 code for the following decimal numbers.

(a) 7

(b) 36

(c) 72.9

(d) 5481

Solution.

(a) The excess-3 code for the decimal 7 is

$$7 + 3 = 10 \rightarrow 1010$$

\therefore The excess-3 code for the decimal 7 = 1010.

$$\begin{array}{r} \text{(b) } 3 \ 6 \\ +3 \ +3 \\ \hline 6 \ 9 \end{array} = 0110 \ 1001$$

$$\begin{array}{r} \text{(c) } 7 \ 2 \ . \ 9 \quad (1010 \ 0101.1100) \\ +3 \ +3 \ +3 \\ \hline 10 \ 5 \ . \ 12 \end{array}$$

$$\begin{array}{r} \text{(d) } 5 \ 4 \ 8 \ 1 \quad (1000 \ 0111 \ 1011 \ 0100) \\ 3 \ 3 \ 3 \ 3 \\ \hline 8 \ 7 \ 11 \ 4 \end{array}$$

Note : (1) The special feature of this unweighted code is self-complementing. That is, 1's complement of an excess-3 code number is the excess-3 code for the 9's complement of the respective decimal number.

For example: The excess-3 code for decimal 4 is 0111. Then it's one's complement is 1000. This 1000 is the excess-3 code for the decimal 5, which is the 9's complement of 4.

This self complementing property makes the excess-3 code useful in some arithmetic operations.

(2) For a digit in any number system to be converted into Ex-3 code, first it should be converted into BCD and then into Ex-3 code and vice versa i.e. conversions between any number system and Ex-3 can be done via BCD code (decimal).

Example 1.76. Convert each of the following numbers of various number systems into Ex-3 code.

- (a) $(1011)_2$ (b) $(436)_8$ (c) $(3A)_{16}$ (d) $(1100.011)_2$ (e) (10.8)

Solution.

$$\begin{array}{l} \text{(a) } \quad 1011 = (11)_{10} \\ \text{Ex-3 code} = \begin{array}{r} 1 \ 1 \ (0100 \ 0100) \\ +3 \ +3 \\ \hline 4 \ 4 \end{array} \end{array}$$

$$\begin{array}{l} \text{(b) } \quad (436)_8 = 64 \times 4 + 3 \times 8 + 6 \\ = 256 + 24 + 6 = (286)_{10} \\ \text{Ex-3 code} = \begin{array}{r} 2 \ 8 \ 6 \ (0101 \ 1011 \ 1001) \\ 3 \ 3 \ 3 \\ \hline 5 \ 11 \ 9 \end{array} \end{array}$$

$$\begin{array}{l} \text{(c) } \quad (3A)_{16} = 16 \times 3 + 10 \\ = (58)_{10} \\ \text{Ex-3 code} = \begin{array}{r} 5 \ 8 \\ +3 \ +3 \\ \hline 8 \ 11 \\ \hline = 1000 \ 1011 \end{array} \end{array}$$

$$\begin{array}{l} \text{(d) } (1100.011)_2 = 12.375 \\ \text{Ex-3 code} = \begin{array}{r} 33 \ 333 \\ \hline 45 \ 6 \ 10 \ 8 \\ = 01000101 \ 0110.1010 \ 1000 \end{array} \end{array}$$

$$\begin{array}{l} \text{(e) } \quad (1C.8)_{16} = 16 \times 1 + 12 + \frac{8}{16} \\ = 16 + 12 + 0.5 = (28.5)_{10} \end{array}$$

Case (ii) : Whenever the sum of decimal digits exceeds 9, there will be a carry from one group to the next. When this happens, the group that produced the carry will revert to 8421 form. This occurs because of the Ex-6 and six unused four-bit groups. To restore the answer to Ex-3 code, we must add 3 to the group that produced the carry.

Example 1.79. Add the following in Ex-3 code

- (a) 28 + 36 (b) 9 + 8 (c) 87.6 + 12.6

Solution.

(a) 28	=	0 1 0 1	1 0 1 1	Excess -3 code for 28
36		0 1 1 0	1 0 0 1	Excess -3 code for 36
		1 1 0 0	0 1 0 0	First result
	-	0 0 1 1	+ 0 0 1 1	Subtract and add 3
		1 0 0 1	0 1 1 1	Ex-3 for 64

(b) Ex-3 code for 9	=	1 1 0 0		
Ex-3 code for 8	=	1 0 1 1		
First result		1 0 1 1		
Add 3		0 0 1 1		
		1 1 0 1 0		= Ex-3 code for 17.

(c) Ex-3 code for 87.6	=	1000 0111 . 0110		
		0011 0011 0011		
		1011 1010 . 1001		
Ex-3 code for 12.6		0001 0010 . 0110		
		0011 0011 0011		
		0100 0101 . 1001		

Add Ex-3 of 87.6 and 12.6	=	1011 1010 . 1001		
		0100 0101 . 1001		
		10000000 . 0010		
Add 0011		0011		
		10000000 . 0101		
				= 100.5

- Advantages :** (1) In the Ex-3 code, addition operation use the ordinary binary addition.
 (2) The 1's and 2's complements can be used to subtract Ex-3 numbers.

1.12.4.3 Gray code

The code which exhibits only a single-bit change from one number to the next is known as 'gray code' i.e., in this code between any two successive code words, there will be change

in only one position. This code is also called as 'unit distance code' or 'cyclic code'. As there is no specific weight assigned to each bit position, this code is an unweighted code. Gray code is also known as 'reflected code', because n -bit gray code can be generated by reflecting the $(n-1)^{\text{th}}$ bit code.

The Table 1.13 shows the bit pattern assigned for gray code from decimal 0 to decimal 15.

Table 1.13. Gray code representation

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

Gray code belongs to a class of codes called 'minimum' changes codes', in which only one bit in the code group changes when going from one step to the next. Because of this, this code is not suitable for arithmetic operations.

Another property of gray code is that the gray-coded number corresponding to the decimal number $2^n - 1$, for any n ; differs from gray coded 0 (0000) in one bit position only (i.e., at n value position). For example, for $n = 2, 3$ and 4 , we see that $2^2 - 1 = 3, 10 = 0010$ in gray code \therefore change at position 1.

$2^3 - 1 = 7 = 0100$ in gray code \therefore change at position 3.

$2^4 - 1 = 15 = 1000$ in gray code \therefore change at position 4.

We can observe the reflection property of the gray code in Table 1.14(a), (b) and (c).

Table 1.14. 2 bit gray code, 3 bit gray code, and 4 bit gray code

2-bit gray code (a)	3-bit gray code (b)	4-bit gray code (c)
$g_1 g_0$	$g_2 g_1 g_0$	$g_3 g_2 g_1 g_0$
0 0	0 0 0	0 0 0 0
0 1	0 0 1	0 0 0 1

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

Advantages

- (1) This code is useful for Input/Output devices, A to D convertes, and other peripheral equipment.
- (2) The Gray code exhibits a single bit change from one code number to the next. This property is useful in shaft position encoders, where error susceptibility increases with the number of bit changes between adjacent numbers in sequence.
- (3) This code is useful when other codes such as binary etc. produce undefined or ambiguous results during the transition from one number to next. For example, to change from 7 to 8 i.e., 0111 to 1000, four changes are needed.
- (4) Very easy for conversion between gray and binary.
- (5) Gray code is used to represent the analog data by continous change of a shaft position. The shaft is partitioned into segments, and each segment is assigned a number. If the adjacent segments are made to correspond with the gray code sequence, ambiguity is eliminated when detection is sensed in the line that separates any two segments shown in Fig. 1.9.

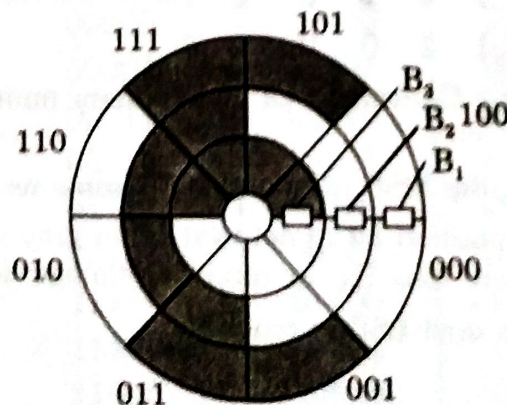


Fig. 1.9 Rotating disk.

Gray to binary conversion : The Gray to binary code conversion can be achieved using the following steps :

- (1) The MSB of the binary number is same as the MSB of the gray code number. So write it down.
- (2) To obtain the next binary bit, perform exclusive OR mod-2 addition or addition with neglecting carry between the just written down binary bit and the next gray code bit. Write down the result.
- (3) Repeat the step 2 until all gray code bits have been completed i.e., until LSB of Gray code number is reached.

Note : (1) The number of bits in a gray code = number of bits in the equivalent binary number and vice-versa.

(2) The conversion process starts from MSB and moves towards LSB.

Let $g_n \dots g_1 g_0$ designate a code word in the $(n + 1)^{th}$ bit gray code. Let $b_n \dots b_1 b_0$ designate the corresponding binary number, where,

suffix 0 indicates LSB and n indicates MSB. Then, i^{th} bit of g_i can be obtained as

$$g_i = b_i \oplus b_{i+1} \quad 0 \leq i \leq n - 1$$

where, b_i represents the i^{th} bit in the binary code.

$$g_n = b_n$$

for gray-binary

$$b_n = g_n$$

$$b_i = b_{i+1} \oplus g_i$$

Example 1.80. Convert the following gray code numbers into binary

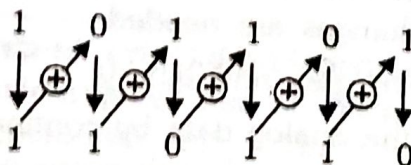
(a) 101101

(b) 10101111

(c) 101011

Solution.

(a) Gray code



Binary code

(b) Gray code

1 0 1 0 1 1 1 1

Binary code

1 1 0 0 1 0 1 0

(c) Gray code

1 0 1 0 1 1

Binary code

1 1 0 0 1 0

Binary to gray conversion : Conversion of given binary number into gray code involves the following steps :

- (1) Record the MSB, that is, the MSB of gray code is same as MSB of the binary number.
- (2) Add this bit to the next position bit to get next bit in gray code, neglecting carry if any and record the sum. (Here also Ex- or mod-2 arithmetic is used for addition).
- (3) Continue recording sums until LSB is reached.

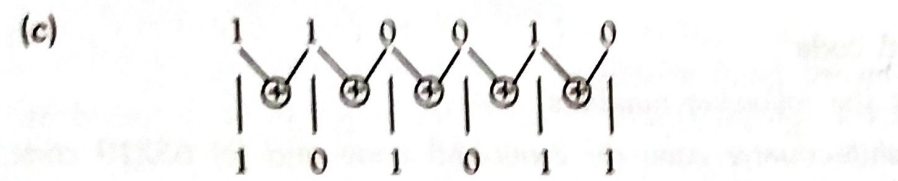
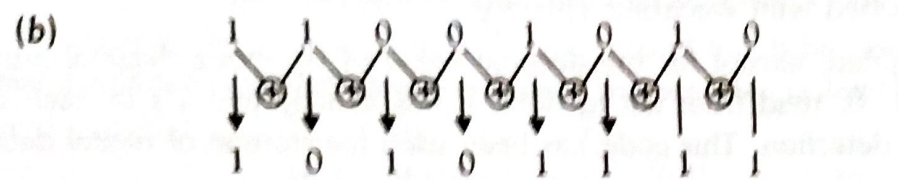
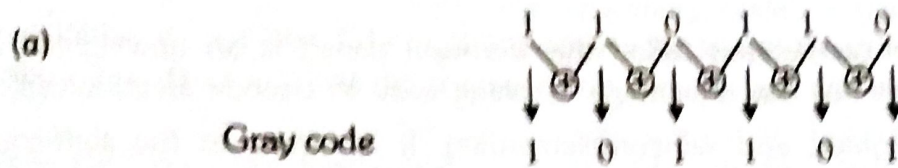
Example 1.81. Convert the following binary number into gray code :

(a) 110110

(b) 11001010

(c) 110010.

Solution.



1.12.4.4 Unweighted BCD code

We know that BCD code is 8421 code. Using this code each decimal digit is represented as weighted 8421 code. If we take decimal number in which each digit is represented using 8421 code hence it becomes unweighted code. The decimal numbers after 9, that is 10, 11, 12 etc. are represented separately for each digit.

i.e., 10 in unweighted BCD is 00010000

11 in unweighted BCD is 00010001 etc.

1.13 5-BIT CODES

Five bit codes also exist. Although only 4-bits are needed to encode any decimal digit from 0 to 9, an extra-bit will allow us to decode the number more easily. Table 1.15 shows some 5-bit BCD codes having special characteristics. These special characteristics of the code are useful for error detection.

Table 1.15. Five-bit BCD codes

Decimal	51111	Shift counter	86421	63210	(2-out of-5)
0	00000	00000	00000	00110	00011
1	00001	00001	00001	00011	00101
2	00011	00011	00010	00101	00110
3	00111	00111	00011	01001	01001
4	01111	01111	00100	01010	01010
5	10000	11111	00101	01100	01100
6	11000	11110	01000	10001	10001
7	11100	11100	01001	10010	10010
8	11110	11000	10000	10100	10100
9	11111	10000	10001	11000	11000

The 9876543210 code, sometimes called the 'counter code' uses only a single 1 in each code group; this makes it easy to decode and to detect errors. The code has the disadvantage of requiring more electronic circuitry than the simpler 4-and 5-bit codes.

The 5043210 code, also called the 'biquinary code' is occasionally used in electronic counters (note that biquinary means two-five). Each member of this code contains a group of 2 bits and a group of 5 bits. The group of two bits indicates whether the number is more or less than 5. The group of five bits denotes the count. Reliable error detection is possible because a single 1 is in the group of 2 bits and a single 1 is in the group of 5 bits.

The 543210 code is like biquinary code, except that 6 bits are used instead of 7. The most significant digit (for left) indicates whether the number is less than 5 or not. This code is sometimes used in electronic counters.

1.15 ALPHA NUMERIC CODES

In addition to numerical data, a computer must be able to handle non-numerical information. In other words, a computer should recognize codes that represent letters of the alphabet, punctuation marks and special characters as well as numbers. We can say that an alpha numeric code represents all of the various characters and functions that are found on a standard typewriter (or computer) keyboard.

The standard binary code for the alphanumeric characters is ASCII. It uses 7 bits to code ($2^7 = 128$) characters, as shown in Table 1.17. The ASCII (Pronounced "ask-ee") code contains 94 graphic characters that can be printed and 34 non printing characters used for various control functions. The graphic characters consist of the 26 upper case letters. (A through Z), 26 lower case letters (a through z), 10 numbers (0 to 9), and 32 special printable characters such as %, *, \$, &, + and - so on. The 34 control characters are designed in the ASCII table with abbreviated names. They are lighted in the table with their full functional names. The control characters are used for routing data and arranging the printed text into a prescribed format. There are three types of control characters : format effectors, information separators, and communication-control characters.

Table 1.17. ASCII codes

ASCII Code	Character	ASCII Code	Character	ASCII Code	Character	ASCII Code	Character
00	NUL	21	!	42	B	63	c
01	SOH	22	"	43	C	64	d
02	STX	23	#	44	D	65	e
03	ETX	24	\$	45	E	66	f
04	EOT	25	%	46	F	67	g
05	ENQ	26	&	47	G	68	h
06	ACK	27	'	48	H	69	i
07	BEL	28	(49	I	6A	j
08	BS	29)	4A	J	6B	k
09	HT	2A	*	4B	K	6C	l
0A	LF	2B	+	4C	L	6D	m

0B	VT	2C	,	4D	M	6E	n
0C	FF	2D	-	4E	N	6F	o
0D	CR	2E	.	4F	O	70	p
0E	SO	2F	/	50	P	71	q
0F	SI	30	0	51	Q	72	r
10	DLE	31	1	52	R	73	s
11	DC1 (X-on)	32	2	53	S	74	t
12	DC2 (Tape)	33	3	54	T	75	u
13	DC3 (X-off)	34	4	55	U	76	v
14	DC4	35	5	56	V	77	w
15	NAK	36	6	57	W	78	x
16	SYN	37	7	58	X	79	y
17	ETB	38	8	59	Y	7A	z
18	CAN	39	9	5A	Z	7B	{
19	EM	3A	:	5B	[7C	
1A	SUB	3B	;	5C	\	7D	} (ALT mode)
1B	ESC	3C	<	5D]	7E	~
1C	FS	3D	=	5E	^(↑)	7F	DEL
1D	GS	3E	>	5F	- (←)		(RUB OUT)
1E	RS	3F	?	60	,		
1F	US	40	@	61	a		
20	SP	41	A	62	b		

Format effectors are characters that control the layout of printing. They include the familiar typewriter controls such as back space (BS), horizontal tabulation (HT), and carriage return (CR).

Information separators are used to separate the data into divisions such as paragraphs and pages. They include characters such as record separator (RS) and file separator (FS).

The *communication-control characters* are useful during the transmission of text between remote terminals. They are STX (start-of-text) and ETX (end of text) which are used to frame a text message when transmitted through telephone wires.

The seven bit code format is $b_6b_5b_4b_3b_2b_1b_0$. The first 3-bits (rightmost 3 bits) are taken from the column in which symbol appears and the last 4 bits correspond to the row. For instance, the letter 5 appears as follows : 101 0011.

ASCII is a 7-bit code, but most computers manipulate an 8-bit quantity as a single unit called a 'byte'. So the ASCII characters most often are stored one per byte., The extra bit is used for other purposes, depending on the application. For example, simple printers recognise 8-bit ASCII characters with the MSB set to 0. When used in data communication, the eighth bit may be used to indicate parity of the character.

The ASCII code is used for the transfer of alphanumeric information between a computer and input/output devices such as video terminals or printers. A computer also uses it internally to store the information that an operator types in at the computer's keyboard.

There are three common ways to display alphanumerics. They are

(a) *Discrete method*— Here, single light source produces each symbol ex : nixie tube.

(b) *Bar-matrix method* – In this method, one or more light sources may be involved in the display of a particular symbol.

(c) *Dot-matrix method* – Here, many individual sources are shaped like dots. A typical example is the 5 × 7 LED matrix.

EBCDIC codes: EBCDIC stands for Extended Binary Coded Decimal Interchange Code. It is pronounced as “ebb-see-dick”. It is a standard code for large computers. It is an 8-bit-code. It provides more extensive character set than ASCII. But it does not provide for parity with an 8-bit byte. It is longer than ASCII. Table 1.18 shows EBCDIC code.

Table 1.18. EBCDIC codes

Characters or Control characters	Hex codes for EBCDIC	Characters or Control characters	Hex codes for EBCDIC
NUL	00	SUB	3F
SOH	01	SP	40
STH	02	BLANK	E0
ETS	03	¢	4A
HT	05	.	4B
DEL	07	,	6B
VT	0B	?	6F
FF	0C	:	7A
CR	0D	;	5E
SO	0E	!	5A
S1	0F	'	7D
DLE	10	“	7F
DC1	11	+	4E
DC2	12	-	60
DC3	13	—	6D
RES	14	*	5C
NL	15	/	61
BS	16	=	7E
CAN	18	<	4C
EM	19	>	6E
FLS	1C	(4D
GS	1D)	5D
RDS	1E	{	8B
US	1F	}	9B
BYP	24	[AD

LF	25	J	DD
EOB	26		4F
ENQ	2D	&	50
ACK	2E	\$	5B
BEL	2F		5F
SYN	32	%	6C
DC4	35	#	7B
EOT	37	@	7C
NAK	3D		
a	81	F	C6
b	82	G	C7
c	83	H	C8
d	84	I	C9
e	85	J	D1
f	86	K	D2
g	87	L	D3
h	88	M	D4
i	89	N	D5
j	91	O	D6
k	92	P	D7
l	93	Q	D8
m	94	R	D9
n	95	S	E2
o	96	T	E3
p	97	U	E4
q	98	V	E5
r	99	W	E6
s	A2	X	E7
t	A3	Y	E8
u	A4	Z	E9
v	A5	0	F0
w	A6	1	F1
x	A7	2	F2
y	A8	3	F3
z	A9	4	F4
A	C1	5	F5
B	C2	6	F6
C	C3	7	F7
D	C4	8	F8
E	C5	9	F9

1.16 ERROR CORRECTING AND DETECTING CODES

The digital information in the form of binary is transmitted from one device to another or circuit (or) system to another. In this, there is a chance for occurring of error. This means a signal corresponding to 0 may change to 1 or vice-versa due to the presence of different types of noise. A reliable system must have a mechanism for detecting and correcting such errors. For this purpose, extra bit (or) more than one bit are added in the data. These extra bits allow the detection and some times correction of errors in the data.

1.16.1 Types of Errors

When ever an electromagnetic signal flows from one point to another, it is subjected to unpredictable interference from heat, magnetism and other forms of electricity. This interference can change the shape or timing of the signal. If the signal is carrying binary data, such changes can alter the meaning of the data changing 0 to 1 or 1 to 0. Bits can be changed singly or in clumps. There are three types of possible errors that may occur: single bit, multiple-bit and burst errors.

Single-bit Error : The term single bit error means that only one bit of a given data unit is changed from 1 to 0 or 0 to 1. The data unit may be a byte, character or packet etc. Example: The sender transmitting a data is 00100100, if the received message is 00000100. Here the 6th position of data is changed from 1 to 0.

Multiple-bit Error : The term multiple-bit error means that two or more non consecutive bits in data unit have changed from 1 to 0 or from 0 to 1.

The Fig. 1.10(a) shows an example of multiple-bit error.

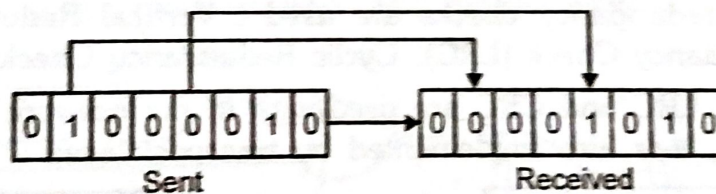


Fig. 1.10(a) : Multiple-bit error.

Burst Error : The term burst error means that two or more consecutive bits in the data unit have changed from 1 to 0 or 0 to 1 see Fig. 1.10(b).

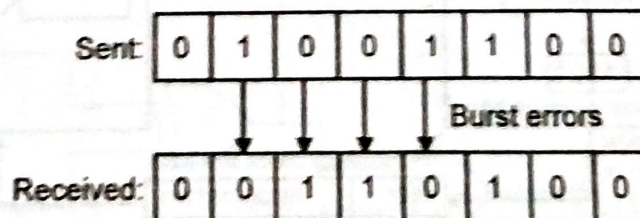


Fig. 1.10(b) : Burst error.

1.16.2 Error Detection

Even if we know, what type of errors can occur, we can recognize those, if we have a copy of the intended transmission for comparison. But if we don't have a copy of the original we will have no way of knowing that we have received an error. There are several mechanisms, they are exact count encoding and redundancy. These codes are used only to detect errors, not to prevent the occurrence of error or not to correct the occurred error.

Exact-count Encoding : With exact count encoding, the no. of 1's in each data unit is same. An example of exact count encoding scheme is the ARQ code. In ARQ code each data unit (character) has three 1's in it, and therefore a simple count of the number of 1's received can determine whether a transmission error has occurred or not.

Redundancy : The redundancy involves transmitting of each data unit twice. If same data unit is not received twice in succession, a transmission error has occurred. The same concept can be used for messages. This system is accurate but it would be in supportable slow. Not only would the transmission time double, but the time it takes to compare every unit by bit must be added.

The major problem with this is, if the error has occurred at the same position of both data units, it is not possible to recognize it even though there is an error, because in comparison all bits of both data units are same.

So instead of transmitting every data unit twice, a shorter group of bits may be appended to the end of each data unit. This technique is called redundancy, because the extra bits are redundant to the information; and they are discarded as soon as the accuracy of transmission has been determined.

As soon as data unit is generated, it passes through a device called generating function that analyzes it and adds an appropriately coded redundancy check bits. This expanded data unit is traveled over the physical channel to the receiver. The receiver puts the entire stream through a checking function, if there is no error, the data position of data unit is accepted and the redundant bits are discarded (See Fig. 1.11).

Four types of redundancy checks are used : Vertical Redundancy Check (VRC), Longitudinal Redundancy Check (LRC), Cyclic Redundancy Check (CRC) and checksum.

Generally VRC, LRC and CRC are used with in the network and checksum is used in the inter-network, it is also implemented by transport layer.

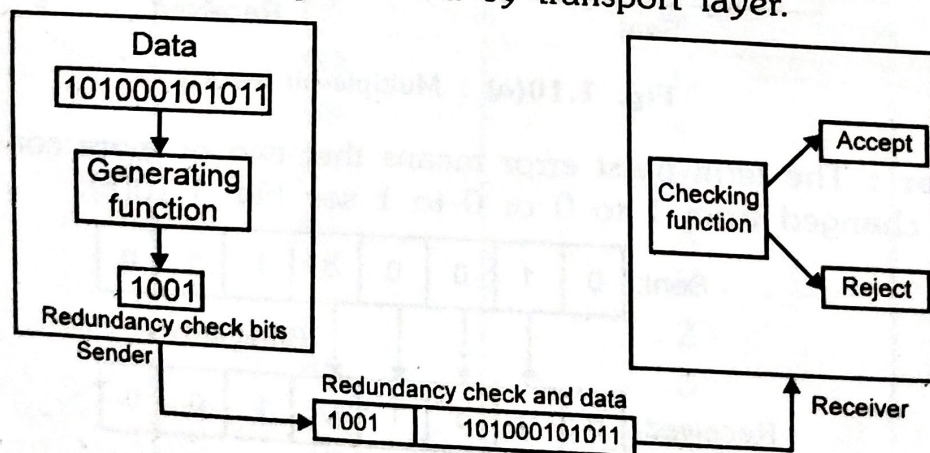


Fig. 1.11 Redundancy.

1.16.2.1 Vertical redundancy check (VRC)

It is most common and least expensive mechanism for error detection. This VRC is also called a parity check. The definition of parity is equivalence of equality. In this technique one bit is appended to each data unit, this bit is called a parity bit or redundant bit, with this bit the total number of 1's in the data unit becomes either even (even parity) or (odd parity). Both sending and receiving systems must be of same parity. If an even-parity data unit is transmitted and an odd parity is received, the receiver knows that the data unit has an error.

For example, if we consider each data unit as a ASCII character, the length of each character is seven bits, including parity, total length of data unit is eight bits.

Suppose we want to transmit a character 'C', the binary value is 1000011 (43 hex). Before transmitting we pass it through a generating function (See Fig. 1.12). Here the generating function is a even-parity generator, it counts the number of 1's and append the appropriate parity bit i.e., 1 with this parity bit the data unit is 11000011, the total number of 1's is a even number. The system now transmits this data unit to the receiver. At the receiver, the checking function counts number of 1's, if the number of 1's is an even number, then the data unit passes, otherwise it will be rejected.

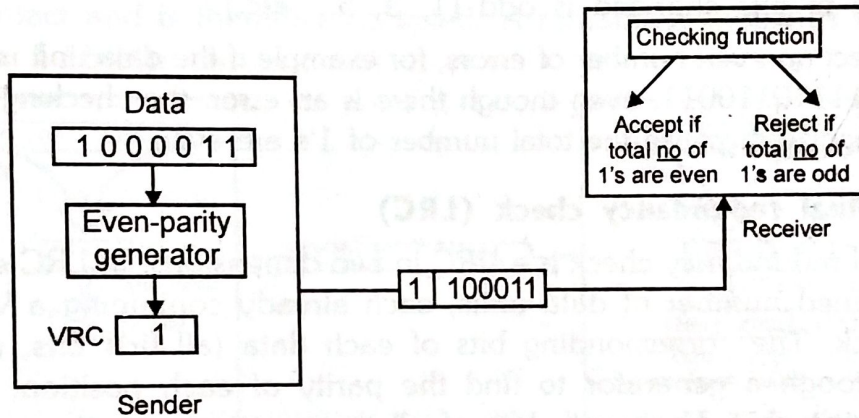


Fig. 1.12 Even parity VRC.

Two types of parity generator circuits are used as a generating function at the transmitter and checking function at the receiver: serial and parallel see Fig. 1.13.

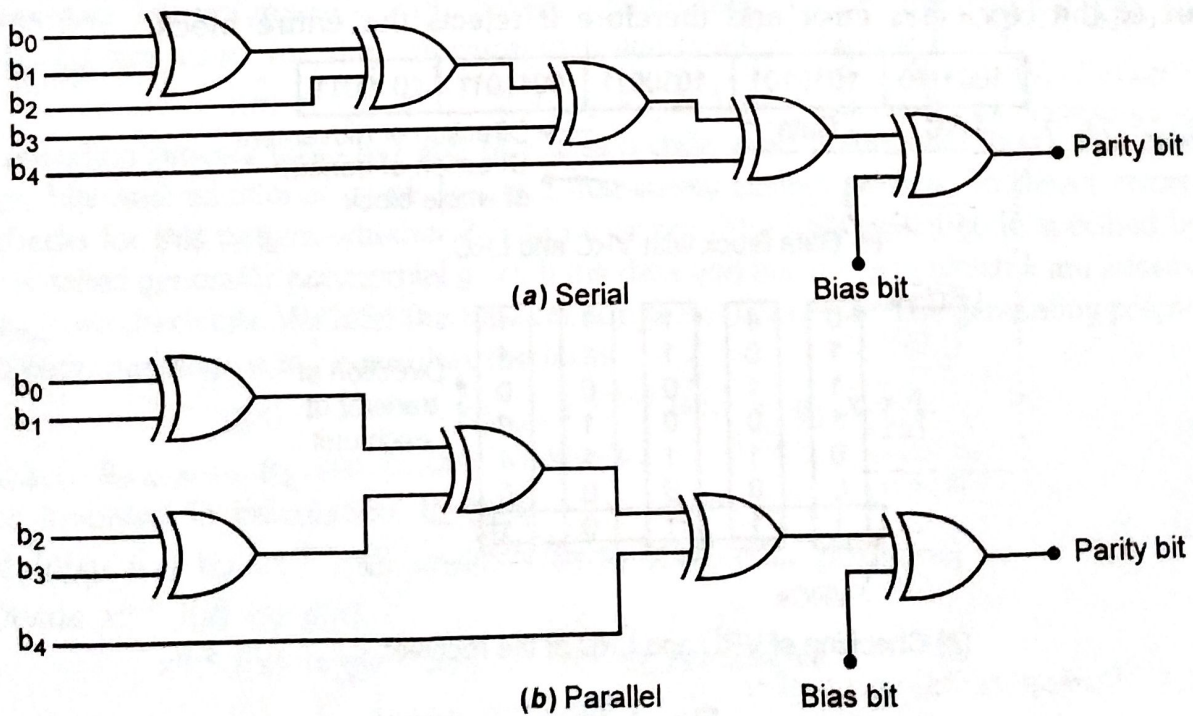


Fig. 1.13 Parity generators.

In the sequential parity generator b_0 is XORed with b_1 , the result is XORed with b_2 and so on. The result of last XOR operation is compared with bias bit. For even parity,

this bit made logic 0, and for odd parity it is logic '1'. The output of the circuit is a parity bit, which is appended to the data unit same as in parallel or combinational generator.

Some circuits can be used as parity checkers in the receiver. A parity checker uses the same procedure as a parity generator except that logic condition of the final comparison is used to determine if a parity violation has occurred (for odd parity a 1 indicates an error and '0' indicates no error, for even parity a 1 indicates an error and 0 indicates no error).

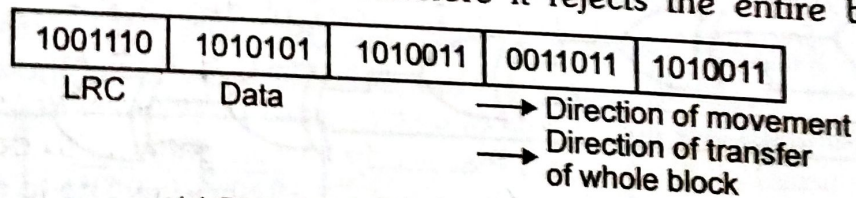
Reliability

1. VRC can detect all single errors. It can also detect multiple-bit or burst errors as long as the number of bits changed is odd (1, 3, 5... etc.).

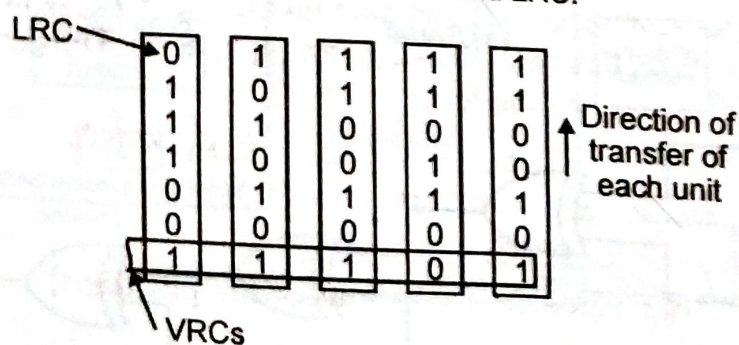
2. It can not detect an even number of errors, for example if the data unit is 11000110011, if it has received as 11110110011, even though there is an error, the checking function at the receiver can not recognize, because the total number of 1's are even.

1.16.2.2 Longitudinal redundancy check (LRC)

The longitudinal redundancy check is a VRC in two dimensions. In LRC error detection, groups a predetermined number of data units, each already containing a VRC parity bit, together into a block. The corresponding bits of each data (all first bits, all second bits etc.) are passed through a generator to find the parity of each position. Each position then gets its own parity bit. The parity bits of all the positions are then assembled into a new data unit which is added at the end of the data block (message) see Fig. 1.14(a). As the data block reaches the receiver, it passes through a LRC checker. The total parity including VRC and LRC must match the parity expected. If so the receiver discards the VRC and LRC, accepts the data units. If any values are wrong, the receiver knows that some part of the block has error and therefore it rejects the entire block.



(a) Data Block with VRC and LRC.



(b) Checking of VRC and LRC at the receiver.

Fig. 1.14

Reliability

1. It determines all single-bit, multiple-bit and burst errors except one pattern of errors.

2. If two bits in one data unit are damaged, and two bits in exactly the same positions in another data unit are also damaged, the LRC checker will not detect an error.

1.16.2.3 Cyclic redundancy check (CRC)

It is the most powerful redundancy checking technique, based on the binary division. Polynomial codes are used for generating check bits in the form of cyclic redundancy check.

The CRC bits are appended at the end of each data unit so that the resulting data unit becomes exactly divisible by a second, predetermined number. At the receiver, the incoming data unit is divided by the same number. If there is no remainder, the data unit is assumed to be intact and is therefore accepted. A remainder indicates that data unit is damaged (See Fig. 1.15(a)).

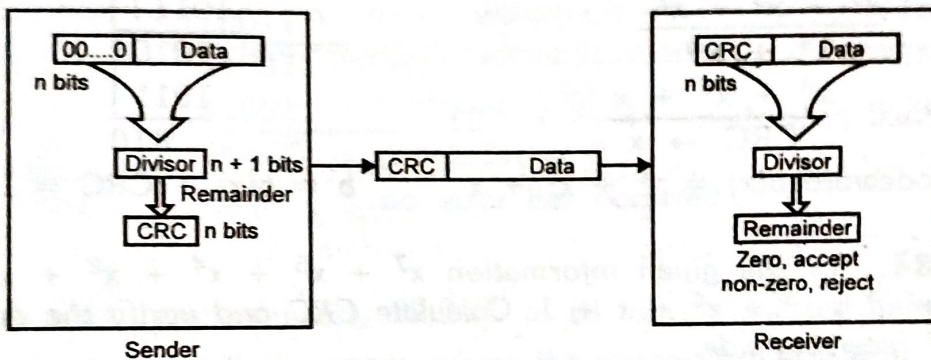


Fig. 1.15(a) CRC generator and checker.

In polynomial codes the information symbols, the code words and the error vectors are represented by polynomial with binary coefficients. The K information bits ($i_{k-1}, i_{k-2}, \dots, i_1, i_0$) are used for the information polynomial of degree $k - 1$.

$$i(x) = i_{k-1}x^{k-1} + i_{k-2}x^{k-2} + \dots + i_1x + i_0$$

The encoding process takes $i(x)$ and produces a code word polynomial ' $b(x)$ ' that contains information bits and additional check bits and that satisfy certain pattern. To detect errors, the receiver checks for this pattern-whether it is same or not. The polynomial code specified by the generator is called generator polynomial $g(x)$. If the data unit has n bits in which k are information bits and $n - k$ are check bits. We refer this type of code as an (n, k) code. The generating polynomial for such a code has degree $n - k$ and has the form

$$g(x) = x^{n-k} + g_{n-k-1}x^{n-k-1} + \dots + g_1x + 1.$$

where $g_{n-k-1}, g_{n-k-2}, \dots, g_1$ are binary numbers.

Steps involved in calculation of CRC:

1. Multiply $i(x)$ by x^{n-k} (put zeros in $(n-k)$ low order positions)
2. Divide $x^{n-k} i(x)$ by $g(x)$

$$x^{n-k} i(x) = g(x) q(x) + r(x) \rightarrow \text{remainder}$$

└─── quotient

3. The remainder polynomial $r(x)$ provides the CRCs, add remainder $r(x)$ to $x^{n-k} i(x)$ (i.e., put check bits in the $n-k$ low order position)

$$b(x) = x^{n-k} i(x) + r(x)$$

where $b(x)$ is a transmitted codeword/data unit.

Example 1.83. The given information is $x^3 + x^2$ and the generating polynomial is $x^3 + x + 1$, find the CRC and transmitted code word.

Solution. Generating polynomial $g(x) = x^3 + x + 1$.

Information: $(1, 1, 0, 0) \rightarrow i(x) = x^3 + x^2$.

Encoding $x^3 i(x) = x^6 + x^5 = 1100000$.

$$\begin{array}{r}
 x^3 + x^2 + x \\
 x^2 + x + 1 \overline{) x^6 + x^5} \\
 \underline{x^6 + x^4 + x^3} \\
 x^5 + x^4 + x^3 \\
 \underline{x^5 + x^3 + x^2} \\
 x^4 + x^2 \\
 \underline{x^4 + x^2 + x} \\
 \text{CRC} \rightarrow x
 \end{array}$$

$$\begin{array}{r}
 1110 \\
 1011 \overline{) 1100000} \\
 \underline{1011} \\
 1110 \\
 \underline{1011} \\
 1010 \\
 \underline{1011} \\
 010
 \end{array}$$

Transmitted codeword $b(x) = x^6 + x^5 + x$

$b = b(x) + \text{CRC} = 1100010$
data CRC

Example 1.84. For the given information $x^7 + x^5 + x^4 + x^2 + x + 1$ and generating polynomial is $x^5 + x^4 + x + 1$. Calculate CRC and verify the data is error received or not at receiving side.

Solution. $g(x) = x^5 + x^4 + x + 1$ or 110011

$i(x) = x^7 + x^5 + x^4 + x^2 + x + 1$ or 10110111

Encoding $= i(x) x^5 = x^{12} + x^{10} + x^9 + x^7 + x^6 + x^5$
 $= 1011011100000$

$$\begin{array}{r}
 11010111 \\
 110011 \overline{) 1011011100000} \\
 \underline{110011} \\
 111101 \\
 \underline{110011} \\
 111010 \\
 \underline{110011} \\
 100100 \\
 \underline{110011} \\
 101110 \\
 \underline{110011} \\
 111010 \\
 \underline{110011} \\
 01001 \rightarrow \text{CRC}
 \end{array}$$

Transmitted data $= b(x) = i(x) + \text{CRC}$
 $= \begin{array}{r} 10110111 \\ \hline \text{data} \end{array} \begin{array}{r} 01001 \\ \hline \text{CRC} \end{array}$

At the receiver, the transmitted data is again divided by $g(x)$.

$$\begin{array}{r}
 11010111 \\
 110011 \overline{) 1011011101001} \\
 \underline{110011} \\
 111101 \\
 \underline{110011} \\
 111010 \\
 \underline{110011} \\
 100110 \\
 \underline{110011} \\
 101010 \\
 \underline{110011} \\
 110011 \\
 \underline{110011} \\
 110011 \\
 \underline{110011} \\
 00000 \rightarrow \text{remainder} = 0 \\
 \text{no error has occurred.}
 \end{array}$$

Reliability

1. With CRC, approximately 99.95% of all transmission errors are detected.
2. It cannot detect those errors, where the change in bit value of a block of code is exactly the value of the divisor or its multiple values. The standard polynomial code or CRC codes are shown in Table 1.19.

Table 1.19. Standard generator polynomials

Name	Polynomial	Used in
CRC-8	$x^8 + x^2 + x + 1$	ATM header error check
CRC-10	$x^{10} + x^9 + x^5 + x^4 + x + 1$	ATM AALCRC
CRC-12	$x^{12} + x^{11} + x^3 + x^2 + x + 1$	Bisync
CRC-16	$x^{16} + x^{15} + x^2 + x + 1$	Bisync
CCITT-16	$x^{16} + x^{12} + x^5 + 1$	HDLC, XMODEM, V.41
CCITT-32	$x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$	IEEE802, DOD, V.42AALs

To generate the CRC check, we can use the shift register circuit. Let us see a circuit for generating CRC bits of example 2.1 (see Fig. 1.15(b)) after 7 clock pulses the contents of registers are $r(x) = 010$

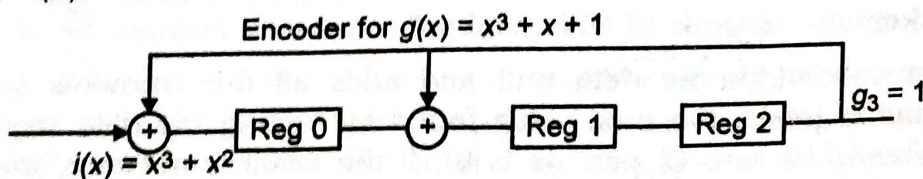


Fig. 1.15(b) Shift register circuit for generated polynomial.

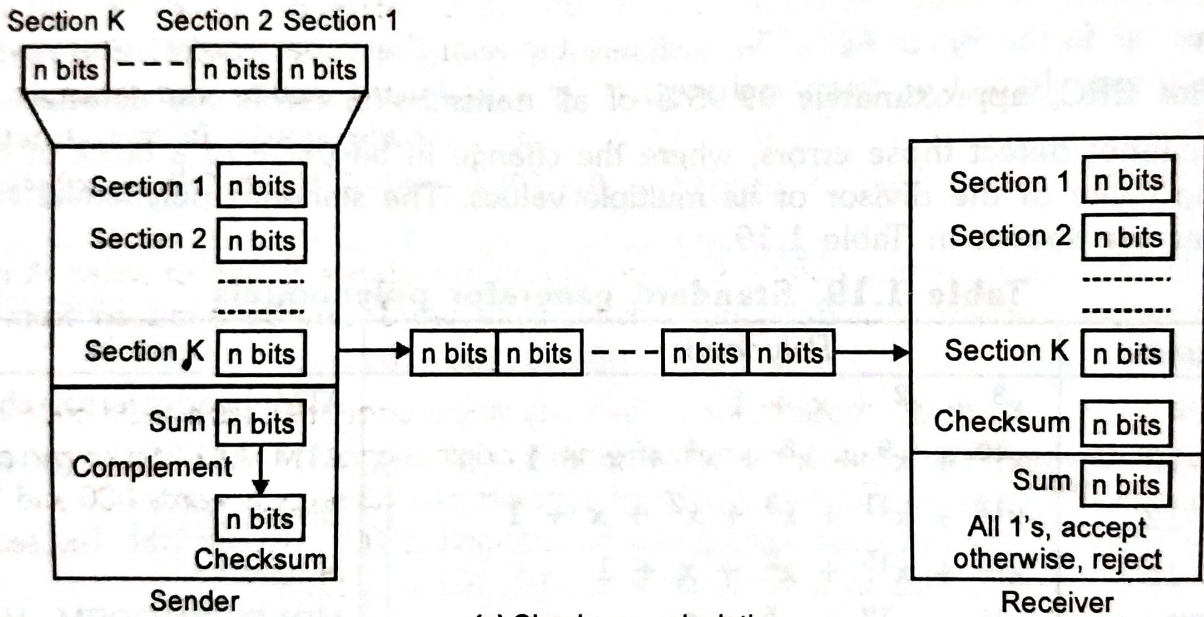
1.16.2.4 Check sum

The error detection method used in the network higher layer protocol is called check sum. The check sum is based on the concept of redundancy. Several internet protocols (IP, TCP, UDP) use check bits to detect errors.

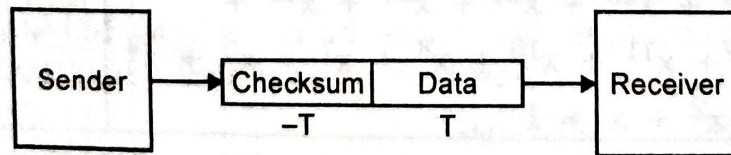
Check sum generator

At the sender, the checksum, is generated by the following steps:

1. The data is divided into k sections, each of n bits (usually-16)
2. Sections 1 and 2 are added together using 1's complement arithmetic in such a way that the total is also n bits long.
3. In this way all the k sections are added.
4. The final result (sum) is then complemented and appended to the end of original data unit as redundancy bits, called checksum field.
5. The extended data unit is transmitted across the network, so the sum of data segment is T , the checksum is $-T$ (see Fig. 1.16).



(a) Checksum calculation



(b) Checksum and data unit

Fig. 1.16

Checksum checker

The receiver subdivides the data unit and adds all the segments together. If the extended data unit is intact, the total value found by adding the data segments and the checksum field should be zero (T plus $-T$ is 0). If the result is not zero, there is an error in the data block. The actual result of the addition will be n 1's if the data is unchanged i.e., represents the value -0 .

Reliability

1. It detects all errors involving odd number of bits, as well as most errors involving even number of bits.

2. However, if one or more bits of a segment are errors and the corresponding bit or bits of opposite value in a second segment are also damaged, the sum of these columns will not change and receiver will not detect a problem.

1.16.3 Error Correction

The mechanisms that we have covered upto this point detect error but cannot correct errors. The error correction can be handled in two ways

1. When an error is discovered, the receiver can have the sender re-transmit the entire data unit.

2. A receiver can use an error correcting code, which automatically corrects certain errors.

The number of bits required to correct a multiple-bit or burst error is so high that in most cases it is inefficient to do so. For this reason in most cases error correction is limited to one, two or three bit errors. For a code to be error detecting, its minimum distance must be further increased. The distance between the code words is the number of digits that must change in one word so that the other word results. For example, the distance between the 1010 and 0100 is three, since the code words differ in three bit positions. The minimum distance of a code is the smallest number of bits in which any code word differ.

A code is an error detecting code only if its minimum distance is two or more. A code is said to be error correcting code if the correct code word can be deduced from the erroneous word.

If the minimum distance of a code is three, then any single error changes a valid code word into an invalid one, which is a distance one away from the original code word a distance two away from any other valid code word. For example, if we consider two valid code words 000 and 111. A single error occurs in the first code word it can be changed to 001, 010 or 100. The second code word can be changed due to a single error to 110, 101 or 011.

1.16.3.1 Single bit error correction

The key to error correction, is that it must be possible to detect and locate erroneous digits. If the location of an error has been determined, then by complementing the erroneous digit the message is corrected.

For example, to correct a single bit in an ASCII character the error correction code must determine which of the seven bits has changed. In this case we have to distinguish between eight different states: no error, error in position 1, error in position 2 and so on up to error in position 7. To do so requires enough redundancy bits to show all eight states.

At first glance, it looks a three bit redundancy code should be adequate because three bits can show eight different states (000 to 111) and can therefore indicate the locations of eight different possibilities. But what if an error occurs in the redundancy bits themselves 7 bits of data plus 3-bits of redundancy equals to 10 bits. Three bits, however can identify only eight possibilities. Additional bits are necessary to cover all possible error locations.

1.16.3.2 Redundancy bits

To calculate the number of redundancy bits (r) required to correct a given number of data bits (m), we must find the relation between m and r . The total length of resulting code is $m + r$ bits.

If the total transmitting unit is ' $m + r$ ' length then ' r ' must be able to indicate at least $m + r + 1$ different states, of these one state means no error and $m + r$ state indicates the location of an error in each of the $m + r$ positions.

So $m + r + 1$ states must be discoverable by r -bits and r -bits indicate 2^r different states.

$$\therefore 2^r \geq m + r + 1 \quad \dots(1.1)$$

The relation between r and m is shown in Table 1.20.

Table 1.20. Relation between data and redundancy bits

Number of data bits (m)	Number of redundancy bits (r)	Total bits ($m + r$)
1	2	3
2	3	5
3	3	6
4	3	7
5	4	9
6	4	10
7	4	11

1.16.3.3 Hamming code

So far, we have examined the number of bits required to cover all of the the possible single bit error states in a transmission. But how to manipulate those bits to discover which bit (digit) has error? A technique was developed by R.W. Hamming.

Positions of the redundancy bits

The Hamming code can be applied to data units of any length by using the equation (1.1).

For example, a seven bit ASCII code requires four redundancy bits, that can be added to the end of the data unit or interspersed with original data bits. Generally, these bits are placed in a positions 1, 2, 4 and 8 (powers of 2). These bits are referred as r_1 , r_2 , r_4 and r_8 (see Fig. 1.17).

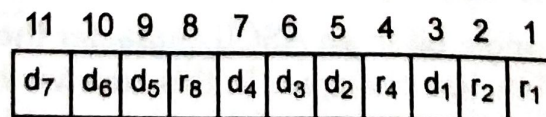


Fig. 1.17 Positions of redundancy bits in Hamming code.

In the Hamming code, each r bit is the vertical redundancy check bit for one combination of data bits.

r_1 is selected so as to establish even parity in bit positions 1, 3, 5, 7, 9, 11

r_2 is selected to establish even parity in bit positions 2, 3, 6, 7, 10, 11

r_4 is selected to establish even parity in bit positions 4, 5, 6, 7

r_8 is selected to establish even parity in bit positions 8, 9, 10, 11

To see the pattern behind this strategies, the r_1 bit is calculated using all bit positions whose binary representation includes 1 in the right most position (at weight equal to 1) and r_2 is calculated using all bit positions with 1 in second position (at weight equal to 2) and so on (see Table 1.21).

Table 1.21. Redundancy bits calculation

Position number	Binary rule			
	r_8	r_4	r_2	r_1
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1
10	1	0	1	0
11	1	0	1	1

See Table 1.21, under column r_4 the bit 1 contains by the positions 4, 5, 6 and 7 only as explained earlier.

Error detection and Error Correction

Now, as soon as the data unit is received, the receiver starts calculating new vertical redundancy checks using some set of bits used by the sender plus relevant parity (r) bit for each set. If in any combination, the number of 1's are even number, assign the corresponding 'c' value as '0' else '1'. Like this, after knowing the values of c_8, c_4, c_2 and c_1 the error bit location can be identified by c's position ($c_8c_4c_2c_1$). All the values of c's is zero, indicates no error has occurred.

Example 1.85. Consider the message is 1001101 is transmitted through the channel, obtain the redundancy bits and transmitting unit needed. Assume bit number 8 has been changed. How to locate it.

Solution.

Data bits = 1001101

Total number of data bits = $m = 7$

The number of redundancy bits required $r = 4$ (from Table 1.20).

Total number of bits in a transmitted data = $m + r = 7 + 4 = 11$

Positions of redundancy bits is 1, 2, 4 and 8. (r_1, r_2, r_4 and r_8).

Calculation of r's:

r_1 is calculated so that the bit positions 1, 3, 5, 7, 9, 11 contains an even number of 1's.

Positions	11	10	9	8	7	6	5	4	3	2	1
	m_7	m_6	m_5	r_8	m_4	m_3	m_2	r_4	m_1	r_2	r_1
Original data	1	0	0		1	1	0		1		
r_1 (to give even-parity at positions 1, 3, 5, 7, 9 and 11)	1		0		1		0		1		1
r_2 (to give even-parity at positions 2, 3, 6, 7, 10, 11)	1	0			1	1			1	0	
r_4 (to give even-parity at positions 4, 5, 6, 7)					1	1	0	0			
r_8 (to give even-parity at positions 8, 9, 10, 11)	1	0	0	1							

coded message = 1001 11 00101

As per the problem, the 8th bit is changed from 1 to 0. i.e., the message is received as 10001 00101. To calculate the position of error occurred at the receiver, recalculate all the values of c_1, c_2, c_4, c_8 with the received data.

Positions	11	10	9	8	7	6	5	4	3	2	1	
Message received	1	0	0	0	1	1	0	0	1	0	1	
1, 3, 5, 7, 9, 11 bits even parity check	1		0		1		0		1		1	$c_1 = 0$ since even parity
2, 3, 6, 7, 10, 11 bits even parity check	1	0			1	1			1	0		$c_2 = 0$ since even parity
4, 5, 6, 7 bits even parity check					1	1	0	0				$c_4 = 0$ since even parity
8, 9, 10, 11 bits even parity check	1	0	0	1								$c_8 = 1$ since odd parity

The error location = $c_8 c_4 c_2 c_1$
 1 0 0 0

which means that the location of error is in position 8. To correct the error, the digit in position 8 is complemented and the correct message 10011100101 is obtained.

Example 1.86. The Hamming code 101101101 is received. Correct it if any errors. There are four parity bits and odd parity is used.

Solution.

Bit positions	9	8	7	6	5	4	3	2	1	
Bit designation	m_5	r_8	m_4	m_3	m_2	r_4	m_1	r_2	r_1	Error position
Received data	1	0	1	1	0	1	1	0	1	Bit value
1, 3, 5, 7 and 9 bits for odd parity check	1		1		0		1		1	$c_1 = 1$ since odd parity
2, 3, 6 and 7 bit positions checking for odd parity			1	1			1	0		Number of 1s is odd: $c_2 = 0$
Bit positions 4, 5, 6 and 7 for checking odd parity			1	1	0	1				Number of 1s is odd: $c_4 = 0$
Bit positions 8 and 9 for checking odd parity	1	0								Number of 1s is odd: $c_8 = 0$

∴ The resultant word, that is, error position = $c = c_8 c_4 c_2 c_1 = 0001$. This says that the bit in the number 1 location is in error. First bit position value should be 0 instead of 1. Hence the correct code is 101101100 and the message is 11101.

Example 1.87. Determine which bit, if any, is in error in the even parity. Hamming coded character is 1100111. Decode the message.

Solution.

Bit positions	7	6	5	4	3	2	1	Error position
Bit designation	m_4	m_3	m_2	r_4	m_1	r_2	r_1	Bit value
Received data	1	1	0	0	1	1	1	
Bit positions 1, 3, 5, and 7 for even parity check	1		0		1		1	$c_1 = 1$
Bit positions 2, 3, 6 and 7 for even parity check	1	1			1	1		$c_2 = 0$
Bit positions 4, 5, 6 and 7 for even parity check	1	1	0	0				$c_4 = 0$

The resultant word, that is, error position

$$c = c_4 c_2 c_1 = 001.$$

This says that the bit in the number 1 location is in error. First bit positional value should be 0 instead of 1. Hence the correct code is 1100110 and the message is 1101.

Example 1.88. Message has been coded in Hamming code for BCD and transmitted through a noisy channel. Decode the message assuming that at most a single error has occurred in each codeword. 1001001011100111101100011011.

Solution. The given bit stream has four Hamming coded BCD codes of 7 bits each. Therefore, the received words are

1. 1001001
2. 0111001
3. 1110110
4. 0011011

For finding the error positions in each word, each word can be checked as in Ex. 1.87 and then get the correct data.

Example 1.89. Encode the information character 01101110101 according to the 15-bit hamming code.

Solution. The number of bits in the message is 11, hence number of redundant bits needed is 4. The location of the parity bits are (1, 2, 4, and 8). As we know that parity bits are assigned by checking message bits having 1 in the same location as parity bit in their binary location numbers.

Bit designation	m_{11}	m_{10}	m_9	m_8	m_7	m_6	m_5	r_8	m_4	m_3	m_2	r_4	m_1	r_2	r_1
Bit location	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
Data bits	0	1	1	0	1	1	1		0	1	0		1		
r_1 (to give even parity at positions 1, 3, 5, 7, 9, 11, 13, 15)	0		1		1		1		0		0		1		0
r_2 (to give even parity at positions 2, 3, 6, 7, 10, 11, 14, and 15)	0	1			1	1			0	1			1	1	
r_4 (to give even parity at positions 4, 5, 6, 7, 12, 13, 14 and 15)	0	1	1	0					0	1	0	1			

r_8 (to give even parity at positions 8, 9, 10, 11, 12, 13, 14 and 15)	0	1	1	0	1	1	1	1							
--	---	---	---	---	---	---	---	---	--	--	--	--	--	--	--

1.17 SOLVED PROBLEMS

Problem 1. List the first 16 numbers in base 12. Use the letters A and B to represent the last two digits.

Solution. 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, 10, 11, 12, 13.

Problem 2. What is the largest binary number that can be obtained with 16 bits? What is its decimal equivalent?

Solution. The largest 16-bit binary number is 1111 1111 1111 1111 and its decimal equivalent = 65,535.

Problem 3. Convert the following numbers.

- (a) 101110, 1110101.11, 110110100 and 11001010.0101 to base 10
- (b) $(.121\ 21)_3$, $(4310)_5$, $(50)_7$, $(198)_{12}$, $(1431)_8$ to base 10
- (c) (1231) , (673.23) , 104, 2004, $(3.1415\dots)_{10}$ and 175.175 to base 2.
- (d) Convert the following decimal numbers to the indicated bases
 - (i) 7562.45 to octal
 - (ii) 1938.257 to hexadecimal
 - (iii) (1776) to base 6
 - (iv) 1984 to base 8.
 - (v) 3.1415... to base 8.
 - (vi) 0.875 into binary trinary and quinary.
- (e) Convert the following binary numbers into octal, hexadecimal and base-4 number system.
 - (i) 11011
 - (ii) 100110.11001
 - (iii) 1111000.0001
- (f) Convert the following octal number into binary, hexadecimal and decimal
 - (i) 347
 - (ii) 444
 - (iii) 576.402
 - (iv) 1431
- (g) Convert the following hexadecimal numbers into binary, octal and decimal.
 - (i) AC
 - (ii) BD
 - (iii) 2002.A3

Solution.

(a) $101110 = 32 + 8 + 4 + 2 = 46$
 $111010.11 = 32 + 16 + 8 + 2 + 0.5 + 0.25 = 58.75$
 $110110100 = 256 + 128 + 32 + 16 + 4 = 336$

3.2.6 Simplification of Product of Sums Expressions (Minimal Products)

In the above discussion, we have considered the boolean expression in sum of products form and grouped 2, 4, and 8 adjacent ones to get the simplified boolean expression in the same form. In practice, the designer should examine both the sum of products and product of sums reductions to ascertain which is more simplified. We have already seen the representation of product of sums on the Karnaugh map. Once the expression is plotted on the K-map instead of making the groups of ones, we have to make groups of zeros. Each group of zero results a sum term and it is nothing but the prime implicate. The technique for using maps for POS reductions is a simple step by step process and it is similar to the one used earlier.

1. Plot the K-map and place 0s in those cells corresponding to the 0s in the truth table or maxterms in the products of sum expression.
2. Check the K-map for adjacent 0s and encircle those 0s which are not adjacent to any other 0s. These are called isolated 0s.
3. Check for those 0s which are adjacent to only one other 0 and encircle such pairs.
4. Check for quads and octets of adjacent 0s even if it contains some 0s that have already been encircled. While doing this make sure that there are minimum number of groups.
5. Combine any pairs necessary to include any 0s that have not yet been grouped.
6. Form the simplified SOP expression for \bar{F} by summing product terms of all the groups.

(Note : The simplified expression is in the complemented form because we have grouped 0s to simplify the expression.)

7. Use DeMorgan's theorem on \bar{F} to produce the simplified expression in POS form.

To get familiar with these steps we will solve some examples.

► **Example 3.14 :** Minimize the expression

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

Solution : $(A + B + \bar{C}) = M_1, (A + \bar{B} + \bar{C}) = M_3, (\bar{A} + \bar{B} + \bar{C}) = M_7,$
 $(\bar{A} + B + C) = M_4, (A + B + C) = M_0$

Step 1 : Fig. 3.22 (a) shows the K-map for three variable and it is plotted according to given maxterms

Step 2 : There are no isolated 0s

Step 3 : 0 in the cell 4 is adjacent only to 0 in the cell 0 and 0 in the cell 7 is adjacent only to 0 in the cell 3. These two pairs are combined and referred to as group 1 and group 2 respectively.

Step 4 : There are no quads and octets.

	BC	$\bar{B}\bar{C}$	$\bar{B}C$	$B\bar{C}$	BC
A	\bar{A}	0	0	0	
	1	0		0	

Fig. 3.22 (a)