UNIT-I

Contents:

NUMBER SYSTEM AND BOOLEAN ALGEBRA

UNIT-I

1.1.1 Number Systems

Number system is a basis for counting varies items. Modern computers communicate and operate with binary numbers which use only the digits 0 &1. Basic number system used by humans is Decimal number system.

For Ex: Let us consider decimal number 18. This number is represented in binary as 10010.

We observe that binary number system take more digits to represent the decimal number. For large numbers we have to deal with very large binary strings. So this fact gave rise to three new number systems.

i) Octalnumber systems
ii) HexaDecimalnumbersystem
iii) BinaryCodedDecimalnumber(BCD)system

To define any number system we have to specify

- Base of the number system such as 2,8,10 or16.
- The base decides the total number of digits available in that number system.
- First digit in the number system is always zero and last digit in the number system is always base-1.

1.1.1.2 The Decimal Number system:

The Decimal number system contains ten unique symbols. 0,1,2,3,4,5,6,7,8,9. Since Counting in decimal involves ten symbols its base or radix is t`en. There is no symbol for its base .i.e, for ten. It is a positional weighted system i.e, the value attached to a symbol depends on its location w.r.t. the decimal point. In this system, any number (integer, fraction or mixed) of any magnitude can be rep. by the use of these ten symbols only. Each symbol in the no. is called a Digit. The leftmost digit in any number representation, which has the greatest positional weight out of all the digits present in that no. is called the MSD (Most Significant Digit) and the right most digit which has the least positional weight out of all the digits present in that no. is called the LSD (Least Significant Digit).The digits on the left side of the decimal pt. form the integer part of a decimal number & those on the right side form the fractional part. The digits to the right of the decimal point have weights which are negative powers of 10 and the digits to the left of the decimal point have weights are positive powers of 10. The value of a decimal no. is the sum of the products of the digit of that no. with their respective column weights. The weights of each column is 10 times greater than the weight of unity or 10^{10}. The first digit to the right of the decimal pt. has a weight of 1/10 or 10^{-1}.for the second 1/100 & for third 1/1000. In general the value of any mixed decimal no. is

$d_n\, d_{n-1}\, d_{n-2}$ ………$d_1\, d_0.d_{-1}\, d_{-2}\, d_{-3}$ …….d_{-k} is given by

$(d_n\, x10_n)+(d_{n-1}\, x10_{\,n-1})+$ ………$(d_1\, x10_1)+(d_0\, x10^1)+(d_{-1}\, x10^2)(d_{-2}\, x10^3)$ …….

1.1.1.2 The Binary Number System:

It is a positional weighted system. The base or radix of this no. system is 2 Hence it has two independent symbols. The basic itself can‘t be a symbol. The symbols used are 0 and 1.The binary digit is called a bit. A binary number consist of a sequence of bits each of which is either a 0 or 1. The binary point separates the integer and fraction parts. Each digit (bit) carries a weight based on its position relative to the binary point. The weight of each bit position is on power of 2 greater than the weight of the position to its immediate right. The first bit to the left of the binary point has a weight of 2^0 & that column is called the **Units Column**. The second bit to the left has a weight of 2^1 & it is in the 2's column & the third has weight of 2^2& so on. The first bit to the right of the binary point has a weight of 2^{-1} & it is said to be in the ½ 's column , next right bit with a weight of 2^{-2} is in ¼'s column so on..The decimal value of the binary no. is the sum of the products of all its bits multiplied by the weight of their respective positions. In general, binary number with an integer part of (n+1) bits & a fraction parts of k bits can be

$d_n\ d_{n-1}\ d_{n-2}$ ………$d_1\ d_0.d_{-1}\ d_{-2}\ d_{-3}$ …….d_{-k}

In decimal equivalent is

$(d_n\ x2^n)+(d_{n-1}\ x2^{n-1})+$ ………$(d_1\ x2^1)+(d_0\ x2^0)+(d_{-1}\ x2^{-1})(d_{-2}\ x2^{-2})$ …….

The decimal equivalent of the no. system

$d_n\ d_{n-1}\ d_{n-2}$ ………$d_1\ d_0.d_{-1}\ d_{-2}\ d_{-3}$ …….d_{-k} in any system with base b is $(d_n\ xb^n)+(d_{n-1}\ xb^{n-1})+$ ………$(d_1\ xb^1)+(d_0\ xb^0)+(d_{-1}\ xb^{-1})(d_{-2}\ xb^{-2})$ …….

The binary no. system is used in digital computers because the switching circuits used in these computers use two-state devices such as transistors , diodes etc. A transistor can be OFF or ON a switch can be OPEN or CLOSED, a diode can be OFF or ON etc (two possible states). These two states represented by the symbols 0 & 1 respectively.

Counting in binary:

Easy way to remember to write a binary sequence of n bits is

- The rightmost column in the binary number begins with a 0 & alternates between 0 & 1.
- Second column begins with $2(=2^1)$ zeros & alternates between the groups of 2 zeros & 2 ones. So on

Decmal no.	Binary no.	Decimal no.	Binary no.
0	0	20	10100
1	1	21	10101
2	10	22	10110
3	11	23	10111
4	100	24	11000
5	101	25	11001
6	110	26	11010
7	111	27	11011
8	1000	28	11100
9	1001	29	11101
10	1010	30	11110
11	1011	31	11111

12	1100	32	100000
13	1101	33	100001
14	1110	34	100010
15	1111	35	100011
16	10000	36	100100
17	10001	37	100101
18	10010	38	100110
19	10011	39	100111

1.1.1.3 The Octal Number System:

It is used by early minicomputers. It is also a positional weights system. Its base or radix is 8.It has 8 independent symbols 0, 1,2,3,4,5,6,7. Since its base $8=2^3$, every 3-bit group of binary can be rep by an octal digit. An octal no. is, 1/3 rd the length of the corresponding binary number.

1.1.1.4 The Hexadecimal number system:

Binary numbers are long & fine for machines but are too lengthy to be handled by human beings. So representing binary numbers concisely with their objective is the hexadecimal no system (or hex) . It is a positional weighted system. The base or radix of there is 16 i.e, it has 16 independent symbols 0,1,2, 9,A,B,C,D,E,F. since its base is $16=2^4$, every 4 binary digit combination can be represented by one hexa decimal digit, so a hexadecimal no is ¼ th the length of the corresponding binary number. A 4 bit group is **nibble.**

Hexadecimal counting system:

0 1 2 3 4 5 6 7 8 9 A B C D E F

10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F

…

F0 F1 F2 -- FF

100 101 --10F

….

1F0 1F1 --1FF

1.1.2 Base Conversions

1.1.2.1 Binary to Decimal Conversion:

It is by the positional weights method. In this method, each binary digit of the no. is multiplied by its position weight. The product terms are added to obtain the decimal no.

<u>**Example**</u>: convert **10101**$_2$ to decimal

Positional weights 2^4 2^3 2^2 2^1 2^0

Binary no. $\mathbf{10101}_2 = (1x\,2^4)+(0x2^3)+(1x2^2)+(0x2^1)+(1x2^0)$

$=16+0+4+0+1$

$= 21_{10}$

<u>Example</u>: convert $\mathbf{11011.101}_2$ to decimal

Positional weights $2^4 \;\; 2^3 \;\; 2^2 \;\; 2^1 \;\; 2^0 \;\; 2^{-1} \;\; 2^{-2} \;\; 2^{-3}$

$=16+8+0+2+1+.5+0+.125$

$= 27.625_{10}$

An integer binary no. can also converted to a an integer decimal no as follows

- Left bit MSB, multiply this bit by 2 & add the provided to next bit to the right.
- Multiply the result obtained in the previous step by 2 & add the product to the next bit to the right.

Example: 1001011_2

1	0	0	1	0	1	1
↓	↓	↓	↓	↓	↓	↓
	1x2+0	2x2+0	4x2+1	9x2+0	18x2+1	37x2+1
	=2	=4	=9	=18	=37	=75

Result=75_{10}

1.1.2.2 Decimal to Binary conversion:

There are reverse processes of the two methods used to convert a binary no. to a decimal number. It converts decimal integer no. to binary integer no by successive division by 2 & the decimal fraction is converted to binary fraction by double –dabble method

<u>Example:</u> 163.875^{10} binary

$163_{10}= 10000000_2+100000_2+10_2+1_2= 10100011_2.$

The fraction part is 0.875_{10}

The largest fraction, which is a power of 2, not exceeding 0.875 is 0.5 $0.5=2^{-1}=0.100_2$

Remainder is 0.875-.5=0.3752.

0.375 is 0.25

$0.25 =2^{-2}=0.01_2$

Remainder is 0.375-.25=0.125.

0.125 is 0.125 itself

$0.125 =2^{-3} =0.001_2$

$0.875_{10}=0.100_2+0.01_2+0.001_2=0.111_2$

final result is $163.875_{10} =10100011.111_2$

Example: convert 52_{10} to binary using double-dabble method

Divide the given decimal no successively by 2 &read the remainders upwards to get the equivalent binary no.

Successive division remainder

```
2 | 52
2 | 26   ---  0
2 | 13   ---  0
2 |  6   ---  1
2 |  3   ---  0        ↓
2 |  1   ---  1        ↓   = 110100₂
2 |  0   ---  1        ↓
```

Example: 0.75_{10} using double – dabble method

Multiply give fraction by 2

Keep the integer in the product as it is & multiply the new fraction in the product by 2

Multiply 0.75 by 2	1.50	↓	
Multiply 0.50 by 2	1.00	↓	$=0.11_2$

1.2.3 Binary to Octal conversion:

Starting from the binary point make groups of 3 bits each, on either side of the binary point & replace each 3 bit binary group by the equivalent octal digit.

Example: Convert 110101.101010_2 to octal Group of 3

110 101 . 101 010

6 5 . 5 2

$=65.52_8$

Example: 10101111001.0111_2

010 101 111 001 . 011 100

2 5 7 1 . 3 4

$=2571.34_8$

1.1.2.4 Octal to decimal Conversion:

Multiply each digit in the octal no by the weight of its position & add all the product terms Decimal value of the octal no.

$d_n\, d_{n-1}\, d_{n-2}\, \ldots\ldots\ldots d_1\, d_0.d_{-1}\, d_{-2}\, d_{-3}\, \ldots\ldots.d_{-k}$ is

$(d_n\, x8^n)+(d_{n-1}\, x8^{n-1})+\ldots\ldots\ldots(d_1\, x8^1)+(d_0\, x8^0)+(d_{-1}\, x8^{-1})(d_{-2}\, x8^2)\ldots\ldots.$

Example: Convert 4057.06_8 to octal

$=4x8^3+0x8^2+5x8^1+7x8^0+0x8^{-1}+6x8^{-2}$

$=2048+0+40+7+0+0.0937 = 2095.0937_{10}$

1.1.2.5 Decimal to Octal Conversion:

To convert a mixed decimal number to a mixed octal number convert the integer and fraction parts separately. To convert decimal integer number to octal, successively divide the given no by 8 till the quotient is 0. The last remainder is the MSD. The remainder read upwards give the equivalent octal integer number to convert the given decimal fraction to octal, successively multiply the decimal fraction & the subsequent decimal fractions by 8 till the product is 0 or till the required accuracy is the MSD. The integers to the left of the octal pt read downwards give the octal fraction.

Example: Convert 378.93_{10} to octal

Successive division:

```
8 | 378
8 | 47      ---   2
8 | 5       ---   7  ↑
    0       ---   5
```

$=572_8$

0.93_{10} to octal

$0.93x8=7.44$

$0.44x8=3.52$↓

$0.53x8=4.16$

$0.16x8=1.28$

$=0.7341_8$

$378.93_{10}=572.7341_8$

Example: 5497_{10} to binary

```
8 | 5497
8 | 687     ---   1
8 | 85      ---   7        ↑
8 | 10      ---   5
8 | 1       ---   2
    0       ---   1        ↑
```

$=12571_8=001010101111001_2$

Example: 101111010001_2 to decimal

$101111010001_2 = 5721_8 = 5x8^3+7x8^2+2x8^1+1x8^0$
$=2560+448+16+1= 3025_{10}$

1.1.2.6 Binary to Hexadecimal conversion:

For this make groups of 4 bits each, on either side of the binary pt & replace each 4 bit group by the equivalent hexadecimal digit.

Examples: 1011011011_2

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

groups of 4-bits: 0010 1101 1011
2 D B $=2DB_{16}$

1.1.2.7 Hexadecimal to binary conversion:

Replace each hex digit by its 4-bit binary group.

Examples: $4BAC_{10}$ to binary

4 B A C
01 1011 1010 1100
00

$=0100101110101100_2$

1.1.2.8 Hexadecimal to Decimal conversion:

Multiply each digit in the hexadecimal no. by its position weight & add those entire product terms.

Hex no is: $d_n\ d_{n-1}\ d_{n-2}\ \ldots\ldots\ldots d_1\ d_0.d_{-1}\ d_{-2}\ d_{-3}\ \ldots\ldots\ldots d_{-k}$

In decimal equivalent is given by

$(d_n \times 16^n)+(d_{n-1} \times 16^{n-1})+ \ldots\ldots\ldots(d_1 \times 16^1)+(d_0 \times 16^0)+(d_{-1} \times 16^{-1})+(d_{-2} \times 16^{-2})$

Example: $5C7_{16}$ to decimal

$(5 \times 16^2)+(C \times 16^1)+ (7 \times 16^0)$

$=1280+192+7$

$=147_{10}$

1.1.2.9 Decimal to Hexadecimal conversion:

It is successively divide the given decimal no. by 16 till the quotient is zero. The last remainder is the MSB. The remainder read from bottom to top gives the equivalent hexadecimal integer. To convert a decimal fraction to hexadecimal successively multiply the given decimal fraction & subsequent decimal fractions by 16, till the product is zero. Or till the required accuracy is obtained, and collect all the integers to the left of decimal point. The first integer is MSB & the integer read from top to bottom give the hexadecimal fraction known as **the hexadabble method.**

Example: 2598.675_{10}

```
16|2598
16|162         --- 6
   10          --- 2
```

$= A26_{16}$

$0.675_{10}=0.675x16$ -- 10.8

$=0.800x16$ -- 12.8

$=0.800x16$ -- 12.8↓

$=0.800x16$ -- $12.8=0.ACCC_{16}$

$2598.675_{10} = A26.ACCC_{16}$

Example: 49056_{10}

```
16 |49056                    decimal     hexa        binary
16  |3066      ---      0          0           0000
16  |191      ---          10                A              1010
16  |11       ---          15↑     F                    1111
     0          ---     11  B                  1011
```

$= BFA0_{16}= 1011,1111,1010,0000_2$

1.1.2.10 Octal to hexadecimal conversion:

The simplest way is to first convert the given octal no. to binary & then the binary no. to hexadecimal.

Example: 756.603_8

7	5	6	.	6	0	3
111	101	110	.	110	000	011
0001	1110	1110	.	1100	0001	1000
1	E	E	.	C	1	8

1.1.2.11 Hexadecimal to octal conversion:

First convert the given hexadecimal number to binary & then the binary to octal.

Example: B9F.AE16

B		9		F	.	A	E	
1011		1001		1111	.	1010	1110	
101	110		011	111	.	101	011	100
5		6	3	7	.	5	3	4

=5637.534

Arithmetic Operations on Number Systems:

Binary Addition:

Rules:

0+0=0

0+1=1

1+0=1

1+1=10 i.e, 0 with a carry of 1.

Example: add binary no.s 1101.101 & 111.011

```
 8421  2^-1 2^-2 2^-3
 1101.101
  111.011
10101.000
```

In 2^{-3} column 1+1=0 with a carry of 1 to the 2^{-2} column

In 2^{-2} column	0+1+1=0	2^{-1}
1	1+0+1=0	1's
2	1+1+1=1	2's
4	0+1+1=0	4's
8	1+1+1=1	8's
16	1+1 =0	16's

Binary Subtraction:

Rules:

0-0=0

1-1=0

1-0=1

0-1=1 with a borrow of 1

Example: subtract binary no.s 111.1_2& 1010.01_2

```
8421  2^-1 2^-2 2^-3
1010.010
111.111
0010.011
```

result is 0010.011_2

Octal Arithmetic:

The rules are similar to the decimal or binary arithmetic. This no. system used to enter long strings of binary data in a digital system like a microcomputer. Arithmetic operations can be performed by converting the octal numbers to binary numbers & then using the rules of binary arithmetic. Octal subtraction can be performed using 1‘s compliment method or 2‘s comp method & can also be performed directly by 7‘s & 8‘s comp methods of decimal system.

		Add 27.5_8 & 74.4_8	Subtract 45_8 from 66_8
27.5_8	=	$010\ 111.101_2$	66_8 $=00\ 110\ 110_2$
$+74.4_8$	=	$+1111000.100_2$	-45_8 $=+11\ 011\ 011_2$
124.1_8		1010100.001_2	22_8 $=(1)00010\ 011$

Multiplication & division can also be performed using the binary representation of octal numbers & then making use of multiplication & division rules of binary numbers.

Hexadecimal Arithmetic:

The rules for arithmetic is same as decimal octal & binary. Arithmetic operations are not done directly in hex. The hexa numbers are first converted into binary & arithmetic operations are done in binary. Hex decimal subtraction can be performed using 1‘s compliment method or 2‘s compliment methods performed directly by 15‘s & 16‘s compliment methods. Similar to the 9‘s & 10‘s compliment of decimal system..

Example: Add $6E_{16}$ & $C5_{16}$ | Subtract $7B_{16}$from$C4_{16}$

$6E_{16}$	$=0110\ 1110_2$	$C4_{16}$	$=1100\ 0100_2$
$C5_{16}$	$=+1100\ 0101_2$	$-7B_{16}$	$=+100001\ 01_2$
133_{16}	$1010\ 100.\ 001$	49_{16}	$(1)010\ 010\ \ 01_2$

Ignore carry answer: Positive.

1.1.3. Complements of Numbers

1.1.3.1 Unsigned Number Representation

Unsigned Integers: The simplest numbers to consider are the integers. We will begin by considering positive integers and then expand the discussion to include negative integers. Numbers that are positive only are called unsigned, and numbers that can also be negative are called signed.

An n-bit unsigned binary number $B = b_{n-1}\ b_{n-2} \cdot \cdot \cdot b_1b_0$

Represents an integer that has the value

$$V(B) = b_{n-1} \times 2^{n-1} + b_{n-2} \times 2^{n-2} + \cdots + b_1 \times 2^1 + b_0 \times 2^0$$

$$= \sum_{i=0}^{n-1} b_i \times 2^i$$

Octal and Hexadecimal Representations:

1. The positional number representation can be used for any radix. If the radix is r, then the number $K = k_{n-1}\ k_{n-2} \cdot \cdot \cdot k_1\ k_0$

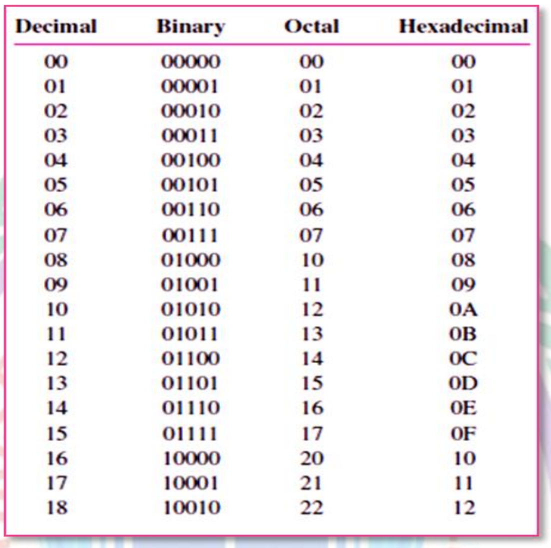
Has the value $V(K) = \sum_{i=0}^{n-1} k_i \times r^i$

2. Our interest is limited to those radices that are most practical. We will use decimal numbers because they are used by people, and we will use binary numbers because they are used by computers.

3. In addition, two other radices are useful—8 and 16. Numbers represented with radix 8 are called octal numbers, while radix-16 numbers are called hexadecimal numbers.

4. In octal representation the digit values range from 0 to 7. In hexadecimal representation (often abbreviated as hex), each digit can have one of 16 values. The first ten are denoted the same as in the decimal system, namely, 0 to 9. Digits that correspond to the decimal values 10, 11, 12, 13, 14, and 15 are denoted by the letters, A, B, C, D, E, and F.
5. In computers the dominant number system is binary. The reason for using the octal and hexadecimal systems is that they serve as a useful shorthand notation for binary numbers.
6. Below table 2.1 gives the first 18 integers in these number systems.

Table 1.1. Numbers in different systems.



Decimal	Binary	Octal	Hexadecimal
00	00000	00	00
01	00001	01	01
02	00010	02	02
03	00011	03	03
04	00100	04	04
05	00101	05	05
06	00110	06	06
07	00111	07	07
08	01000	10	08
09	01001	11	09
10	01010	12	0A
11	01011	13	0B
12	01100	14	0C
13	01101	15	0D
14	01110	16	0E
15	01111	17	0F
16	10000	20	10
17	10001	21	11
18	10010	22	12

7. One octal digit represents three binary bits. Thus a binary number is converted into an octal number by taking groups of three bits, starting from the least-significant bit, and replacing them with the corresponding octal digit. For example, 101011010111 is converted as

$$\underbrace{101}_{5}\quad\underbrace{011}_{3}\quad\underbrace{010}_{2}\quad\underbrace{111}_{7}$$

8. Which means that $(101011010111)_2 = (5327)_8$.
9. If the number of bits is not a multiple of three, then we add 0s to the left of the most-significant bit. For example, $(10111011)_2 = (273)_8$ because of the grouping

$$\underbrace{0\,1\,0}_{2}\quad\underbrace{1\,1\,1}_{7}\quad\underbrace{0\,1\,1}_{3}$$

10. Conversion from octal to binary is just as straightforward; each octal digit is simply replaced by three bits that denote the same value.
11. Similarly, a hexadecimal digit represents four bits. For example, a 16-bit number is represented by four hex digits, as in $(1010111100100101)_2 = (AF25)_{16}$

Using the grouping

$$\underbrace{1\,0\,1\,0}_{A}\quad\underbrace{1\,1\,1\,1}_{F}\quad\underbrace{0\,0\,1\,0}_{2}\quad\underbrace{0\,1\,0\,1}_{5}$$

12. Zeros are added to the left of the most-significant bit if the number of bits is not a multiple of four. For example, $(1101101000)_2 = (368)_{16}$ because of the grouping

$$\underbrace{0\,0\,1\,1}_{3}\quad\underbrace{0\,1\,1\,0}_{6}\quad\underbrace{1\,0\,0\,0}_{8}$$

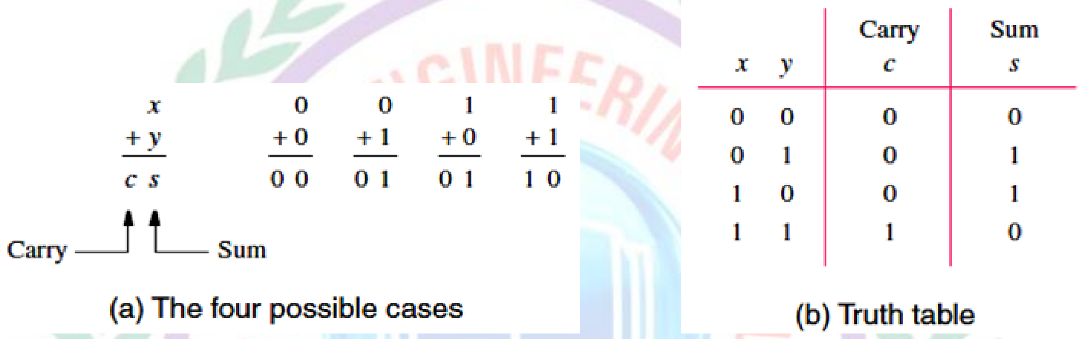
13. Conversion from hexadecimal to binary involves straightforward substitution of each hex digit by four bits that denote the same value.

14. Binary numbers used in modern computers often have 32 or 64 bits. Written as binary *n*-tuples (sometimes called bit vectors), such numbers are awkward for people to deal with. It is much simpler to deal with them in the form of 8- or 16-digit hex numbers.

1.1.3.1 Addition of Unsigned Numbers:

1. Binary addition is performed in the same way as decimal addition except that the values of individual digits can be only 0 or 1.

2. The one-bit addition entails four possible combinations, as indicated in Figure 1.1a. Two bits are needed to represent the result of the addition. The right-most bit is called the sum, s. The left-most bit, which is produced as a carry-out when both bits being added are equal to 1, is called the carry,



x	y	Carry c	Sum s
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Fig.1.1 One bit addition

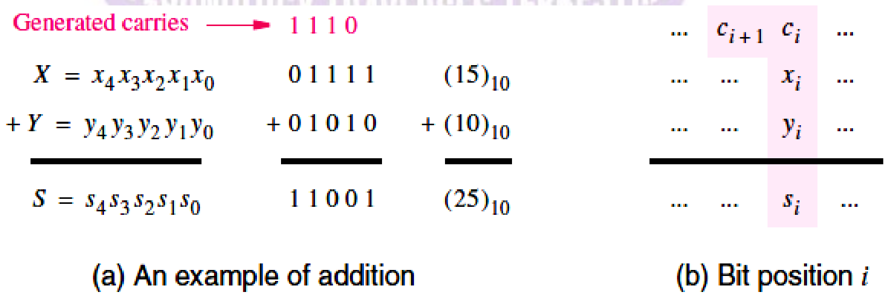
3. The addition operation is defined in the form of a truth table in part 1.1(b) of the figure

4. A more interesting case is when larger numbers that have multiple bits are involved. Then it is still necessary to add each pair of bits, but for each bit position *i*, the addition operation may include a *carry-in* from bit position *i* − 1.

5. Below figure 2.2.a presents an example of the addition operation. The two operands are X = $(01111)_2 = (15)_{10}$ and Y = $(01010)_2 = (10)_{10}$.

6. Five bits are used to represent X and Y, making it possible to represent integers in the range from 0 to 31; hence the sum S = X + Y = $(25)_{10}$ can also be denoted as a five-bit integer.

7. Note the labeling of individual bits, such that $X = x_4\, x_3\, x_2\, x_1\, x_0$ and $Y = y_4\, y_3\, y_2\, y_1\, y_0$. The figure shows, in a pink color, the carries generated during the addition process. For example, a carry of 0 is generated when x_0 and y_0 are added; a carry of 1 is produced when x_1 and y_1 are added, and so on.



Generated carries → 1 1 1 0
X = x4x3x2x1x0 0 1 1 1 1 (15)10
+ Y = y4y3y2y1y0 + 0 1 0 1 0 + (10)10
S = s4s3s2s1s0 1 1 0 0 1 (25)10
(a) An example of addition
... ci+1 ci ...
... ... xi ...
... ... yi ...
... ... si ...
(b) Bit position i

Fig.1.2 Addition of multibit numbers

8. For bit position 0, there is no carry-in, for each other bit position i, the addition involves bits x_i and y_i , and a carry-in c_i , as illustrated in Figure 1.2b. This observation leads to the design of a logic circuit that has three inputs x_i , y_i , and c_i , and produces the two outputs s_i and c_{i+1}.

1.1.3.2. Signed Number Representation

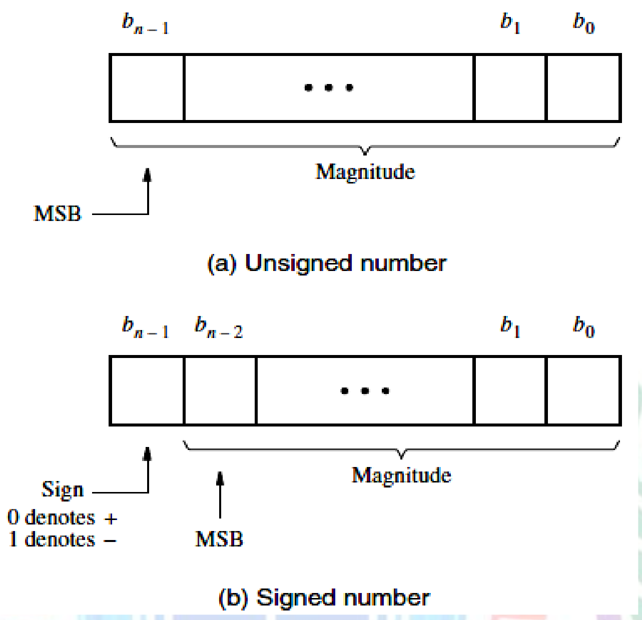


Fig. 1.3 Formats for representation of integers

1. In the decimal system the sign of a number is indicated by a + or − symbol to the left of the most-significant digit. In the binary system the *sign* of a number is denoted by the left-most bit. For a positive number the left-most bit is equal to 0, and for a negative number it is equal to 1.
2. Therefore, in signed numbers the left-most bit represents the sign, and the remaining $n - 1$ bits represent the magnitude,
3. In unsigned numbers all bits represent the magnitude of a number; hence all n bits are *significant* in defining the magnitude. Therefore, the MSB is the left-most bit, b_{n-1}. In signed numbers there are $n - 1$ significant bits, and the MSB is in bit position b_{n-2}.

Negative Numbers:

Negative numbers can be represented in three different ways:

1. Sign-and-magnitude
2. 1's complement
3. 2's complement

Sign-and-Magnitude Representation:

1. In the familiar decimal representation, the magnitude of both positive and negative numbers is expressed in the same way. The sign symbol distinguishes a number as being positive or negative. This scheme is called the sign-and-magnitude number representation.
2. The same scheme can be used with binary numbers in which case the sign bit is 0 or 1 for positive or negative numbers, respectively. For example, if we use four-bit numbers, then +5 = 0101 and −5 = 1101.

3. This representation is not well suited for use in computers. More suitable representations are based on complementary systems.

1's Complement Representation:

1. In a complementary number system, the negative numbers are defined according to a subtraction operation involving positive numbers. We will consider two schemes for binary numbers: the 1's complement and the 2's complement.
2. In the *1's complement* scheme, an *n*-bit negative number, *K*, is obtained by subtracting its equivalent positive number, *P*, from $2^n - 1$; that is, $K = (2^n - 1) - P$.
3. For example, if $n = 4$

Then $K = (2^4 - 1) - P$

$= (15)_{10} - P = (1111)_2 - P$

4. If we convert +5 to a negative

We get $-5 = 1111 - 0101 = 1010$

Similarly $+3 = 0011$

And $-3 = 1111 - 0011 = 1100$

5. Clearly, the 1's complement can be obtained simply by complementing each bit of the number, including the sign bit.
6. While 1's complement numbers are easy to derive, they have some drawbacks when used in arithmetic operations, as we will see in the next section.

2's Complement Representation

1. In the 2's complement scheme, a negative number, K, is obtained by subtracting its equivalent positive number, P, from 2^n; namely, $K = 2^n - P$.

Using our four-bit example

$-5 = 10000 - 0101 = 1011$

And $-3 = 10000 - 0011 = 1101$

2. Finding 2's complements in this manner requires performing a subtraction operation that involves borrows.
3. However, we can observe that if K_1 is the 1's complement of P and K_2 is the 2's complement of P, then

$$K_1 = (2^n - 1) - P$$
$$K_2 = 2^n - P$$

4. It follows that $K_2 = K_1 + 1$. Thus a simpler way of finding a 2's complement of a number is to add 1 to its 1's complement because finding a 1's complement is trivial.
5. This is how 2's complement numbers are obtained in logic circuits that perform arithmetic operations. There is a simple rule that can be used for this purpose.

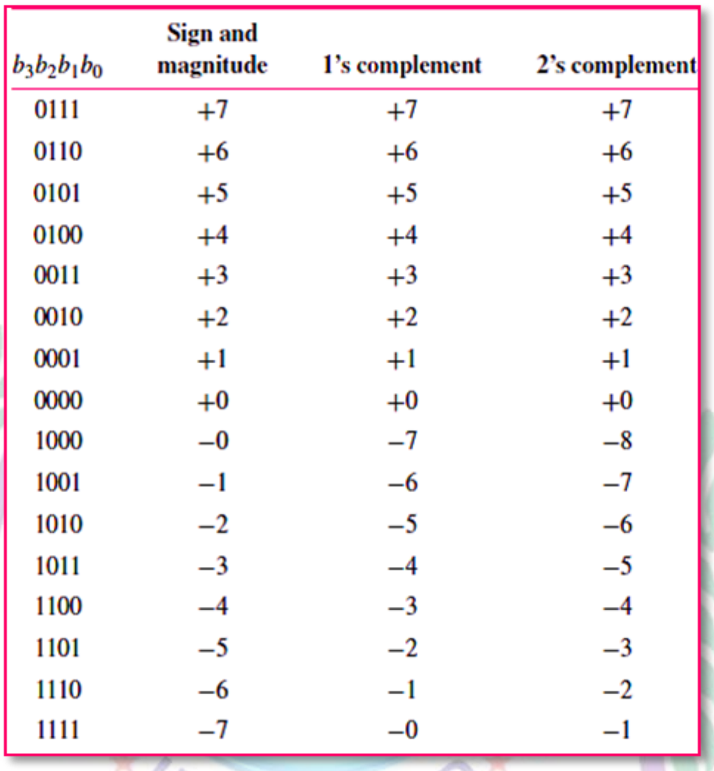
Rule for Finding 2's Complements

1. Given a number $B = b_{n-1}\, b_{n-2} \cdot\cdot\cdot b_1 b_0$, its 2's complement, $K = k_{n-1}\, k_{n-2} \cdot\cdot\cdot k_1\, k_0$, can be found by examining the bits of *B* from right to left and taking the following action: copy all bits of *B* that are 0 and the first bit that is 1; then simply complement the rest of the bits.
2. For example, if $B = 0110$, then we copy $k_0 = b_0 = 0$ and $k_1 = b_1 = 1$, and complement the rest so that $k_2 = \bar{b}_2 = 0$ and $k_3 = \bar{b}_3 = 1$. Hence $K = 1010$.

3. As another example, if $B = 10110100$ then we copy $k_0 = b_0 = 0$,$k_1 = b_1 = 0$, $k_2 = b_2 = 1$ and complement the rest so that $k_3 = b_3 = 1$ and $k_4 = b_4 = 0$ and $k_5 = b_5 = 0$ and $k_6 = b_6 = 1$ and $k_7 = b_7 = 0$, Then $K = 01001100$.

4. Below table 1.2 illustrates the interpretation of all 16 four-bit patterns in the three signed number representations that we have considered.

Table 1.2. Interpretation of four-bit signed integers.



$b_3b_2b_1b_0$	Sign and magnitude	1's complement	2's complement
0111	+7	+7	+7
0110	+6	+6	+6
0101	+5	+5	+5
0100	+4	+4	+4
0011	+3	+3	+3
0010	+2	+2	+2
0001	+1	+1	+1
0000	+0	+0	+0
1000	−0	−7	−8
1001	−1	−6	−7
1010	−2	−5	−6
1011	−3	−4	−5
1100	−4	−3	−4
1101	−5	−2	−3
1110	−6	−1	−2
1111	−7	−0	−1

5. Note that for both sign-and-magnitude representation and for 1's complement representation there are two patterns that represent the value zero.

6. For 2's complement there is only one such pattern. Also, observe that the range of numbers that can be represented with four bits in 2's complement form is −8 to +7, while in the other two representations it is −7 to +7.

7. Using 2's-complement representation, an n-bit number $B = b_{n-1}\, b_{n-2} \cdot \cdot \cdot\, b_1b_0$ represents the value $V(B) = (-b_{n-1} \times 2^{n-1}) + b_{n-2} \times 2^{n-2} +\cdot \cdot \cdot+ b_1 \times 2^1 + b_0 \times 2^0$

8. Thus the largest negative number, 100 . . . 00, has the value -2^{n-1}. The largest positive number, 011 . . . 11, has the value $2^{n-1} - 1$.

1.3.3. Addition and Subtraction:

1. To assess the suitability of different number representations, it is necessary to investigate their use in arithmetic operations—particularly in addition and subtraction.

2. We can illustrate the good and bad aspects of each representation by considering very small numbers. Use four-bit numbers, consisting of a sign bit and three significant bits.

3. Addition of positive numbers is the same for all three number representations. It is actually the same as the addition of unsigned numbers.

4. But there are significant differences when negative numbers are involved. The difficulties that arise become apparent if we consider operands with different combinations of signs.

Sign-and-Magnitude Addition

1. If both operands have the same sign, then the addition of sign-and-magnitude numbers is simple. The magnitudes are added, and the resulting sum is given the sign of the operands.
2. However, if the operands have opposite signs, the task becomes more complicated. Then it is necessary to subtract the smaller number from the larger one.
3. This means that logic circuits that compare and subtract numbers are also needed. We will see shortly that it is possible to perform subtraction without the need for this circuitry. For this reason, the sign-and-magnitude representation is not used in computers.

1's Complement Addition

1. An obvious advantage of the 1's complement representation is that a negative number is generated simply by complementing all bits of the corresponding positive number.
2. Below figure 1.4 shows what happens when two numbers are added. There are four cases to consider in terms of different combinations of signs. As seen in the top half of the figure, the computation of 5 + 2 = 7 and *(*−5*)* + 2 = *(*−3*)* is straightforward; a simple addition of the operands gives the correct result. Such is not the case with the other two possibilities.
3. Computing 5 + *(*−2*)* = 3 produces the bit vector 10010. Because we are dealing with four-bit numbers, there is a carry-out from the sign-bit position. Also, the four bits of the result represent the number 2 rather than 3, which is a wrong result.
4. Interestingly, if we take the carry-out from the sign-bit position and add it to the result in the least-significant bit position, the new result is the correct sum of 3.

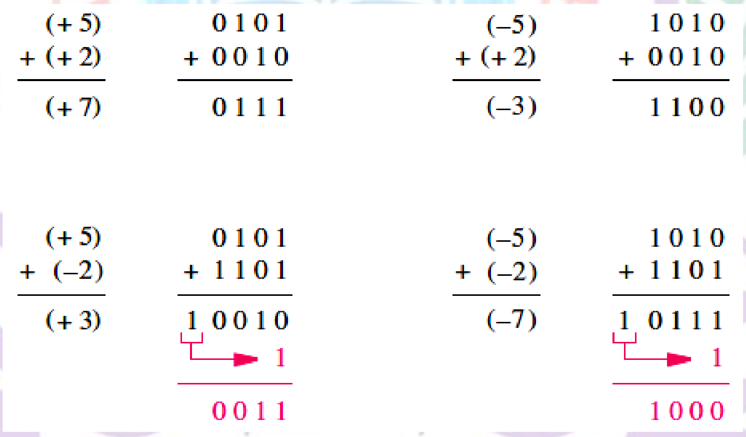


Fig. 1.4 Examples of 1's complement addition

5. A similar situation arises when adding *(*−5*)* + *(*−2*)* = *(*−7*)*. After the initial addition the result is wrong because the four bits of the sum are 0111, which represents +7 rather than −7. But again, there is a carry-out from the sign-bit position, which can be used to correct the result by adding it in the LSB position
6. The conclusion from these examples is that the addition of 1's complement numbers may or may not be simple. In some cases a correction is needed, which amounts to an extra addition that must be performed.

2's Complement Addition

1. Consider the same combinations of numbers as used in the 1's complement example. Below figure 1.5 indicates how the addition is performed using 2's complement numbers.

$$\begin{array}{rr} (+5) & 0\,1\,0\,1 \\ +\,(+2) & +\,0\,0\,1\,0 \\ \hline (+7) & 0\,1\,1\,1 \end{array} \qquad\qquad \begin{array}{rr} (-5) & 1\,0\,1\,1 \\ +\,(+2) & +\,0\,0\,1\,0 \\ \hline (-3) & 1\,1\,0\,1 \end{array}$$

$$\begin{array}{rl} (+5) & 0\,1\,0\,1 \\ +\,(-2) & +\,1\,1\,1\,0 \\ \hline (+3) & 1\,0\,0\,1\,1 \\ & \uparrow \\ & \text{ignore} \end{array} \qquad\qquad \begin{array}{rl} (-5) & 1\,0\,1\,1 \\ +\,(-2) & +\,1\,1\,1\,0 \\ \hline (-7) & 1\,1\,0\,0\,1 \\ & \uparrow \\ & \text{ignore} \end{array}$$

Fig.1.5 Examples of 2's complement addition.

2. Adding 5 + 2 = 7 and *(*−5*)* + 2 = *(*−3*)* is straightforward. The computation 5 + *(*−2*)* = 3 generates the correct four bits of the result, namely 0011. There is a carry-out from the sign-bit position, which we can simply ignore.

3. The fourth case is *(*−5*)* + *(*−2*)* = *(*−7*)*. Again, the four bits of the result, 1001, give the correct sum *(*−7*)*. In this case also, the carry-out from the sign-bit position can be ignored.

4. As illustrated by these examples, the addition of 2's complement numbers is very simple. When the numbers are added, the result is always correct. If there is a carry-out from the sign-bit position, it is simply ignored.

5. Therefore, the addition process is the same, regardless of the signs of the operands. It can be performed by an adder circuit.

6. Hence the 2's complement notation is highly suitable for the implementation of addition operations.

2's Complement Subtraction

$$\begin{array}{rr} (+5) & 0\,1\,0\,1 \\ -\,(+2) & -\,0\,0\,1\,0 \\ \hline (+3) & \end{array} \quad \Longrightarrow \quad \begin{array}{rl} & 0\,1\,0\,1 \\ & +\,1\,1\,1\,0 \\ \hline & 1\,0\,0\,1\,1 \\ & \uparrow \\ & \text{ignore} \end{array}$$

$$\begin{array}{rr} (-5) & 1\,0\,1\,1 \\ -\,(+2) & -\,0\,0\,1\,0 \\ \hline (-7) & \end{array} \quad \Longrightarrow \quad \begin{array}{rl} & 1\,0\,1\,1 \\ & +\,1\,1\,1\,0 \\ \hline & 1\,1\,0\,0\,1 \\ & \uparrow \\ & \text{ignore} \end{array}$$

$$\begin{array}{rr} (+5) & 0\,1\,0\,1 \\ -\,(-2) & -\,1\,1\,1\,0 \\ \hline (+7) & \end{array} \quad \Longrightarrow \quad \begin{array}{r} 0\,1\,0\,1 \\ +\,0\,0\,1\,0 \\ \hline 0\,1\,1\,1 \end{array}$$

$$\begin{array}{rr} (-5) & 1\,0\,1\,1 \\ -\,(-2) & -\,1\,1\,1\,0 \\ \hline (-3) & \end{array} \quad \Longrightarrow \quad \begin{array}{r} 1\,0\,1\,1 \\ +\,0\,0\,1\,0 \\ \hline 1\,1\,0\,1 \end{array}$$

Fig. 1.6 Examples of 2's complement subtraction.

1. The easiest way of performing subtraction is to negate the subtrahend and add it to the minuend. This is done by finding the 2's complement of the subtrahend and then performing the addition. Above figure 1.6 illustrates the process.
2. The operation 5 − *(*+2*)* = 3 involves finding the 2's complement of +2, which is 1110. When this number is added to 0101, the result is 0011 = *(*+3*)* and a carry-out from the sign-bit position occurs, which is ignored. A similar situation arises for *(*−5*)* − *(*+2*)* = *(*−7*)*. In the remaining two cases there is no carry-out, and the result is correct.
3. Subtraction operation can be realized as the addition operation, using a 2's complement of the subtrahend, regardless of the signs of the two operands. Therefore, it should be possible to use the same adder circuit to perform both addition and subtraction.

1.1.4. Binary to Gray & Gray to Binary

The Gray code (reflective –code):

Gray code is a non-weighted code & is not suitable for arithmetic operations. It is not a BCD code. It is a cyclic code because successive code words in this code differ in one bit position only i.e, it is a unit distance code. Popular of the unit distance code. It is also a reflective code i.e, both reflective & unit distance. The n least significant bits for 2^n through 2^{n+1}-1 are the mirror images of those for 0 through 2^n-1.An N bit gray code can be obtained by reflecting an N- 1 bit code about an axis at the end of the code, & putting the MSB of 0 above the axis & the MSB of 1 below the axis.

Reflection of gray codes:

Gray Code				Decimal	4 bit binary
1 bit	2 bit	3 bit	4 bit		
0	00	000	0000	0	0000
1	01	001	0001	1	0001
	11	011	0011	2	0010
	10	010	0010	3	0011
		110	0110	4	0100
		111	0111	5	0101
		101	0101	6	0110
		110	0100	7	0111
			1100	8	1000
			1101	9	1001
			1111	10	1010
			1110	11	1011
			1010	12	1100
			1011	13	1101
			1001	14	1110
			1000	15	1111

1.1.4.1 Binary to Gray conversion:

N bit binary no is rep by B_n B_{n-1} -------- B_1

Gray code equivalent is by G_n G_{n-1} ---------- G_1

B_n,, G_n are the MSB's then the gray code bits are obtaind from the binary code as

$G_n=B_n$	$G_{n-1}=B_n \oplus B_{n-1}$	$Gn\text{-}2=B_{n-1} \oplus B_{n-2}$	-----------	$G_1=B_2 \oplus B_1$

$\oplus$→EX-or symbol

Procedure: ex-or the bits of the binary no with those of the binary no shifted one position to the right. The LSB of the shifted no. is discarded & the MSB of the gray code number is the same as the MSB of the original binary number.

Example: 10001

(a). Binary : 1 →0 →0 →1

Gray : $\oplus$ 1 $\oplus$ 1 $\oplus$ 0 1

(b). Binary: 1 0 0 1

Shifted binary: 1 0 0 (1)

1 1 0 1→gray

1.1.4.2 Gray to Binary Conversion:

If an n bit gray no. is rep by G_n G_{n-1} ------------------- G_1

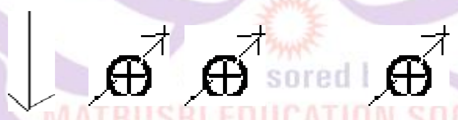
its binary equivalent by B_n B_{n-1} -----------B_1 then the binary bits are obtained from gray bits as

$B_n= G_n$ $B_{n-1}=B_n \oplus G_{n-1}$ $B_{n-2}= B_{n-1} \oplus G_{n-2}$ ----------- $B1=B_2 \oplus G_1$

To convert no. in any system into given no. first convert it into binary & then binary to gray. To convert gray no into binary no & convert binary no into require no system.

Example: 10110010(gray) = 11011100_2= DC_{16}=334_8=220_{10} EX:1101

Gray: 1 1 0 1



Binary: 1 0 0 1

Example: $3A7_{16}$= $0011,1010,0111_2$=1001110100(gray)

527_8=$101,011,011_2$=111110110(gray)

652_{10}=1010001100_2= 1111001010(gray)

1.1.5. Logic Gates

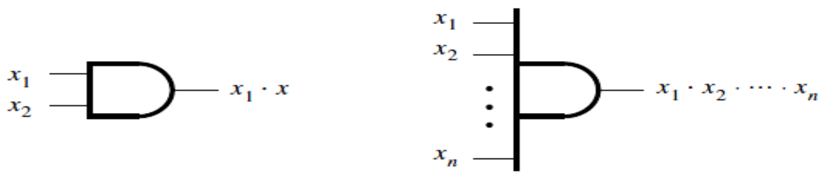
Logic gates are fundamental building blocks of digital systems. Logic gate produces one output level when some combinations of input levels are present. & a different output level when other combination of input levels is present. In this, 3 basic types of gates are there. AND OR & NOT

The interconnection of gates to perform a variety of logical operation is called *Logic Design*. Inputs & outputs of logic gates can occur only in two levels.1, 0 or High, Low or True , False or On , Off. A table which lists all the possible combinations of input variables & the corresponding outputs is called a Truth Table. It shows how the logic circuits output responds to various combinations of logic levels at the inputs. *Level Logic*, a logic in which the voltage levels represent logic 1 & logic 0.Level logic may be Positive Logic or Negative Logic. In *Positive Logic* the higher of two voltage levels represent logic 1 & Lower of two voltage levels represent logic 0. In *Negative Logic* the lower of two voltage levels represent logic 1 & higher of two voltage levels represent logic 0.

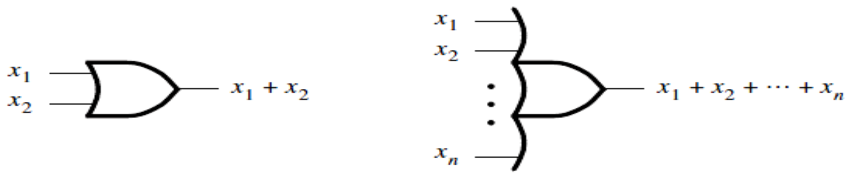
1. Logic circuits are used in digital computers.
2. Such circuits also form the foundation of many other digital systems, such as those that perform control applications or are involved in digital communications. All such applications are based on some simple logical operations that are performed on input information.
3. Information in computers is represented as electronic signals that can have two discrete values. Although these values are implemented as voltage levels in a circuit, we refer to them simply as logic values, 0 and 1.
4. Any circuit in which the signals are constrained to have only some number of discrete values is called a **logic circuit.** Logic circuits can be designed with different numbers of logic values, such as three, four, or even more.
5. The *binary* logic circuits that have two logic values.
6. Binary logic circuits have the dominant role in digital technology.

1.1.5.1. Variables and Functions:

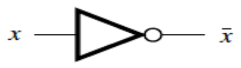
1. A complex function may require many of the basic operations for its implementation. Each logic operation can be implemented electronically with transistors, resulting in a circuit element called a *logic gate.*
2. A logic gate has one or more inputs and one output that is a function of its inputs. It is often convenient to describe a logic circuit by drawing a circuit diagram, or *schematic*, consisting of graphical symbols representing the logic gates.
3. The dominance of binary circuits in digital systems is a consequence of their simplicity, which results from constraining the signals to assume only two possible values.
4. The **“·”** symbol is called **the *AND operator***, and the circuit in Figure 1.7*a* is said to implement a ***logical AND function*.**
5. The + symbol is called the ***OR operator***, and the circuit in Figure 1.7*b* is said to implement a ***logical OR function*.**
7. The value of this function is the inverse of the value of the input variable. Instead of using the word *inverse*, it is more common to use the term *complement*. Thus we say that *L(x)* is a complement of *x* in this example. Another frequently used term for the same operation is the *NOT operation.*
8. There are several commonly used notations for indicating the complementation.
   Thus the following are equivalent:
   $= x' = !\,x = \sim x = NOT\ x$
9. The graphical symbols for the AND, OR, and NOT gates are shown in Figure 1.7.



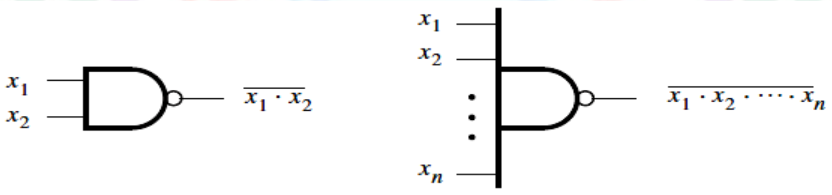
(a) AND gates



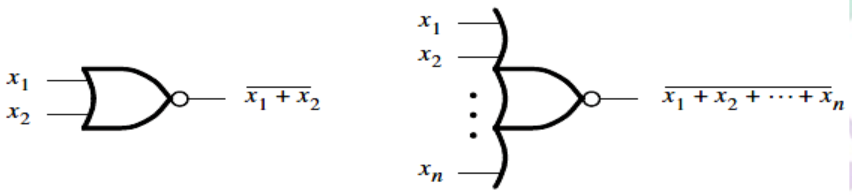
(b) OR gates



(c) NOT gate


x_1
x_2
$\overline{x_1 \cdot x_2}$
x_1
x_2
x_n
$\overline{x_1 \cdot x_2 \cdot \cdots \cdot x_n}$

(a) NAND gates

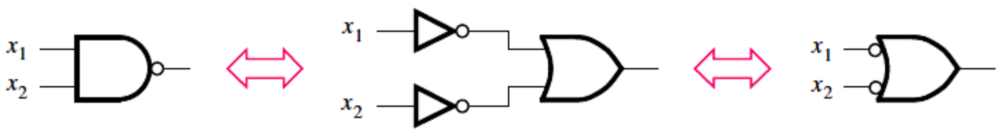


(b) NOR gates

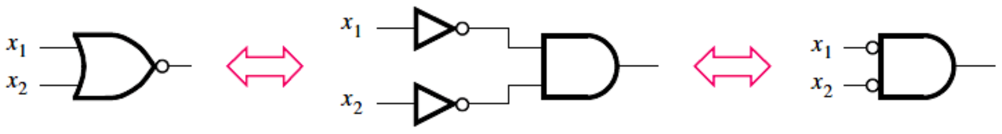
Fig.1.7 Logic gates

10. NAND and NOR functions which are obtained by complementing the output generated by AND and OR operations, respectively.
11. These functions are attractive because they are implemented with simpler electronic circuits than the AND and OR functions.
12. Figure 1.7 gives the graphical symbols for the NAND and NOR gates. A bubble is placed on the output side of the AND and OR gate symbols to represent the complemented output signal.
13. NAND and NOR gates are realized with simpler circuits than AND and OR gates
14. Its logic gate interpretation is shown in below Figure 1.8.
15. From DeMorgan's theorem identity 15*a* is interpreted in part (*a*) of the figure. It specifies that a NAND of variables *x*1 and *x*2 is equivalent to first complementing each of the

variables and then ORing them. Notice on the far-right side that we have indicated the NOT gates simply as bubbles, which denote inversion of the logic value at that point.



(a) $\overline{x_1 x_2} = \bar{x}_1 + \bar{x}_2$



(b) $\overline{x_1 + x_2} = \bar{x}_1 \bar{x}_2$

Fig.1.8 DeMorgan's theorem in terms of logic gates

16. The other half of DeMorgan's theorem, identity 15*b*, appears in part (*b*) of the figure. It states that the NOR function is equivalent to first inverting the input variables and then ANDing them.
8. Any logic function can be implemented either in sumof- products or product-of-sums form, which leads to logic networks that have either an AND-OR or an OR-AND structure, respectively. We will now show that such networks can be implemented using only NAND gates or only NOR gates.

Truth Tables:

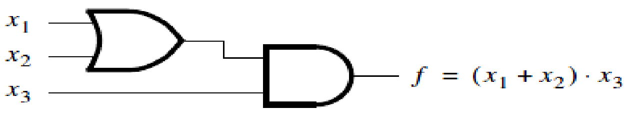
1. All the logical operations can also defined in the form of table1.3 called a truth table.
2. As shown in Table the first two columns (to the left of the double vertical line) give all four possible combinations of logic values that the variables *x*1 and *x*2 can have. The next column defines the AND operation for each combination of values of *x*1 and *x*2, and the last column defines the OR operation

Table. 1.3 A truth table for the AND and OR operation

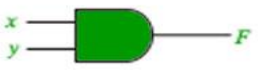
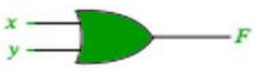
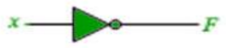
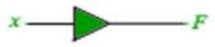
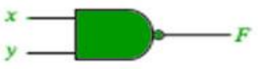
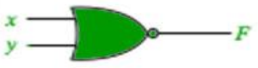
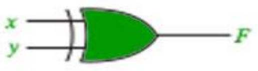
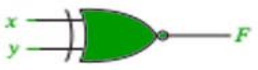
x_1	x_2	. *(AND)*	+ *(OR)*
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

3. The truth table is a useful aid for depicting information involving logic functions. We will use it to define specific functions and to show the validity of certain functional relations.
4. Small truth tables are easy to deal with. However, they grow exponentially in size with the number of variables. A truth table for three input variables has eight rows because there are eight possible valuations of these variables.
5. In general, for n input variables the truth table has 2^n rows.
6. The figure indicates on the left side how the AND and OR gates are drawn when there are only a few inputs. On the right side it shows how the symbols are augmented to accommodate a greater number of inputs.

7. A larger circuit is implemented by a *network* of gates.



8. For example, the logic function from above Figure shows series-parallel connection can be implemented by the network in below figure. The complexity of a given network has a direct impact on its cost. Because it is always desirable to reduce the cost of any manufactured product, it is important to find ways for implementing logic circuits as inexpensively as possible.

Name	Graphic symbol	Algebraic function	Truth table
AND		$F = x \cdot y$	x y | F 0 0 | 0 0 1 | 0 1 0 | 0 1 1 | 1
OR		$F = x + y$	x y | F 0 0 | 0 0 1 | 1 1 0 | 1 1 1 | 1
Inverter		$F = x'$	x | F 0 | 1 1 | 0
Buffer		$F = x$	x | F 0 | 0 1 | 1
NAND		$F = (xy)'$	x y | F 0 0 | 1 0 1 | 1 1 0 | 1 1 1 | 0
NOR		$F = (x + y)'$	x y | F 0 0 | 1 0 1 | 0 1 0 | 0 1 1 | 0
Exclusive-OR (XOR)		$F = xy' + x'y$ $= x \oplus y$	x y | F 0 0 | 0 0 1 | 1 1 0 | 1 1 1 | 0
Exclusive-NOR or equivalence		$F = xy + x'y'$ $= (x \oplus y)'$	x y | F 0 0 | 1 0 1 | 0 1 0 | 0 1 1 | 1

1.1.6. Realization Using Gates

1. With some basic ideas, we can implement arbitrary functions using the AND, OR, and NOT gates.

2. The function of the circuit is to continuously monitor the state of the switches and to produce an output logic value 1 whenever the switches (x_1, x_2) are in states (0, 0),(0, 1), or (1, 1). If the state of the switches is (1, 0), the output should be 0.

3. We can express the required behavior using a truth table, as shown in table 1.3.

Table 1.3 A function to be synthesized

x_1	x_2	$f(x_1, x_2)$
0	0	1
0	1	1
1	0	0
1	1	1

4. A possible procedure for designing a logic circuit that implements this truth table is to create a product term that has a value of 1 for each valuation for which the output function f has to be 1.

5. Then we can take a logical sum of these product terms to realize *f*.

6. Let us begin with the fourth row of the truth table, which corresponds to $x_1 = x_2 = 1$. The product term that is equal to 1 for this valuation is $x_1 \cdot x_2$, which is just the AND of x_1 and x_2.

7. Next consider the first row of the table, for which to $x_1 = x_2 = 0$. For this valuation the value 1 is produced by the product term $x_1 \cdot x_2$. Similarly, the second row leads to the term. $x_1 \cdot x_2$.

8. Thus *f* may be realized as $f(x_1, x_2) = x_1 \cdot x_2 + x_1' \cdot x_2 + x_1' \cdot x_2'$

9. The logic network that corresponds to this expression is shown in Figure 1.9 a

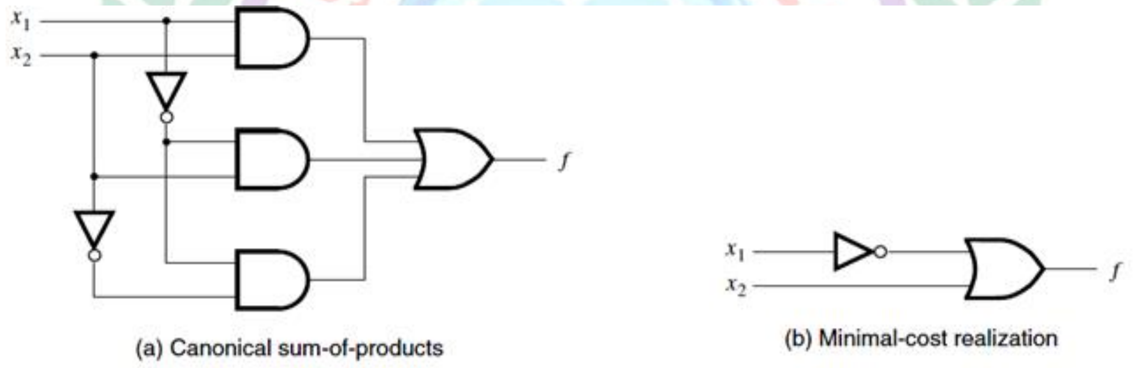


Fig.1.9 Two implementations of the function

10. Although this network implements *f* correctly, it is not the simplest such network. To find a simpler network, we can manipulate the obtained expression using the theorems and properties.

11. According to theorem 7*b*, we can replicate any term in a logical sum expression. Replicating the third product term, the above expression becomes

$$f(x_1, x_2) = x_1 \cdot x_2 + x_1' \cdot x_2 + x_2' \, x_1' + x_1' \cdot x_2$$

12. Now the distributive property 12*a* allows us to write

$$f(x_1, x_2) = (x_2 + x_2').\ x_1' + (x_1 + x_1').\ x_2$$

13. Applying theorem 8*b* we get

$$f(x_1, x_2) = 1.\ x_1' + 1.\ x_2$$

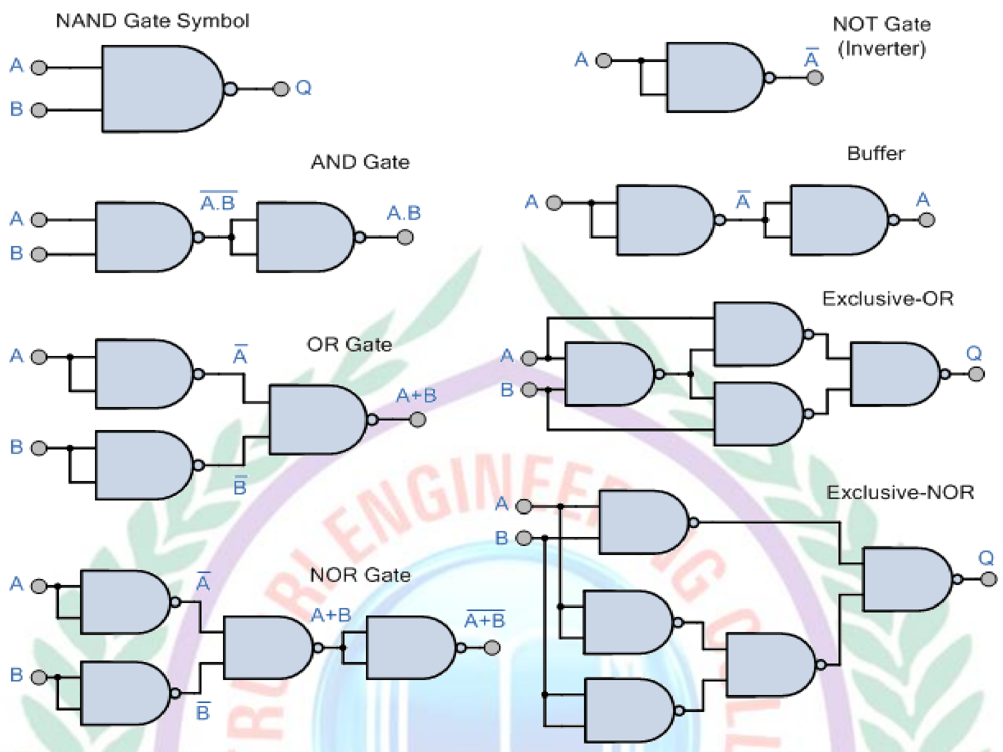
14. Finally, theorem 6*a* leads to $f(x_1, x_2) = x_2 + x_1'$

15. The network described by this expression is given in Figure 1.9 *b*. Obviously the cost of this network is much less than the cost of the network in part (*a*) of the figure 1.9.

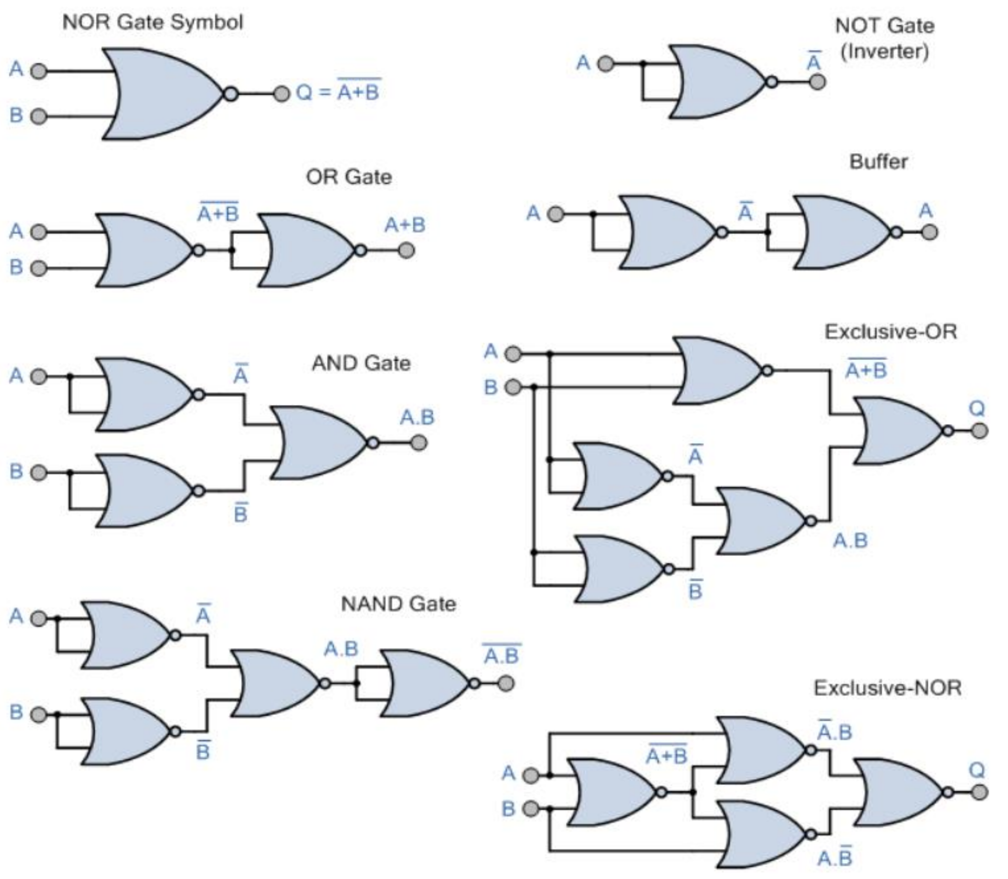
16. Algebraic manipulation used to derive simplified logic expressions and thus lower-cost networks.

Implementation of Logic Gate using Universal Gates

Universal Logic Gates using only NAND Gates



Universal Logic Gates using only NOR Gates



1.1.7. Postulates & Theorems of Boolean Algebra

1. In 1849 George Boole published a scheme for the algebraic description of processes involved in logical thought and reasoning. Subsequently, this scheme and its further refinements became known as **Boolean algebra**. It was almost 100 years later that this algebra found application in the engineering sense.
2. Algebra is a powerful tool that can be used for designing and analyzing logic circuits. It provides the foundation for much of our modern digital technology.

Axioms of Boolean algebra:

1. Like any algebra, Boolean algebra is based on a set of rules that are derived from a small number of basic assumptions. These assumptions are called **axioms**. Let us assume that Boolean algebra involves elements that take on one of two values, 0 and 1.
2. Assume that the following axioms are true:

1a. $0 \cdot 0 = 0$

1b. $1 + 1 = 1$

2a. $1 \cdot 1 = 1$

2b. $0 + 0 = 0$

3a. $0 \cdot 1 = 1 \cdot 0 = 0$

3b. $1 + 0 = 0 + 1 = 1$

4a. *If* $x = 0$, *then* $x' = 1$

4b. *If* $x = 1$, *then* $x' = 0$

Single-Variable Theorems:

1. From the axioms we can define some rules for dealing with single variables. These rules are often called theorems.
2. If x is a Boolean variable, then the following theorems hold:

5a. $x \cdot 0 = 0$

5b. $x + 1 = 1$

6a. $x \cdot 1 = x$

6b. $x + 0 = x$

7a. $x \cdot x = x$

7b. $x + x = x$

8a. $x \cdot x' = 0$

8b. $x + x' = 1$

9. $(x')' = x$

3. It is easy to prove the validity of these theorems by perfect induction, that is, by substituting the values $x = 0$ and $x = 1$ into the expressions and using the axioms given above.

Two- and Three-Variable Properties:

1. To enable us to deal with a number of variables, it is useful to define some two- and three-variable algebraic identities. For each identity, its dual version is also given. These identities are often referred to as properties.

2. They are known by the names indicated below.If x, y, and z are Boolean variables, then the following properties hold:

***10a.** x · y = y · x*	***Commutative***
***10b.** x + y = y + x*	
***11a.** x · (y · z) = (x · y) · z*	***Associative***
***11b.** x + (y + z) = (x + y) + z*	
***12a.** x · (y + z) = x · y + x · z*	***Distributive***
***12b.** x + y · z = (x + y) · (x + z)*	
***13a.** x + x · y = x*	***Absorption***
***13b.** x · (x + y) = x*	
***14a.** x · y + x · y' = x*	***Combining***
***14b.** (x + y) · (x +y') = x*	
***15a.** (x · y)' = x'+y'*	***DeMorgan's theorem***
***15b.** (x+ y)' = x'· y'*	
***16a.** x + x'· y = x + y*	
***16b.** x · (x'+ y) = x · y*	
***17a.** x · y + y · z + y' · z = x · y + y'· z*	***Consensus***
***17b.** (x + y) · (y + z) · (+ z) = (x + y) · (+ z)*	

3. Again, we can prove the validity of these properties either by perfect induction or by performing algebraic manipulation.
4. Table 1.2 illustrates how perfect induction can be used to prove DeMorgan's theorem, using the format of a truth table. The evaluation of left-hand and right-hand sides of the identity in 15*a* gives the same result.

Table 1.2. Proof of DeMorgan's theorem in 15*a*

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

5. We have a number of axioms, theorems, and properties. Not all of these are necessary to define Boolean algebra.
6. The preceding axioms, theorems, and properties provide the information necessary for performing algebraic manipulation of more complex expressions.

Consensus Theorem

Theorem 1. AB+ A'C+ BC=AB+A'C
Theorem 2. (A+B).(A'+C).(B+C)=(A+B).(A'+C)

The BC term is called the consensus term and is redundant. The consensus term is formed from a PAIR OF TERMS in which a variable (A) and its complement (A') are present; the consensus term is formed by multiplying the two terms and leaving out the selected variable and its complement

Consensus Theorem 1 Proof:

AB+A’C+BC=AB+A’C+(A+A’)BC
=AB+A’C+ABC+A’BC
=AB(1+C)+A’C(1+B)
= AB+A’C

Duality:

1. Given a logic expression, its **dual** is obtained by replacing all + operators with · operators, and vice versa, and by replacing all 0s with 1s, and vice versa.
2. The dual of any true statement (axiom or theorem) in Boolean algebra is also a true statement
3. Duality implies that at least two different ways exist to express every logic function with Boolean algebra
4. Often, one expression leads to a simpler physical implementation than the other and is thus preferable.

Principle of Duality:

Each postulate consists of two expressions statement one expression is transformed into the other by interchanging the operations (+) and (·) as well as the identity elements 0 and 1. Such expressions are known as duals of each other.
If some equivalence is proved, then its dual is also immediately true.
If we prove:(x.x)+(x’+x’)=1, then we have by duality: (x+x)·(x’.x’)=0

Complement of a Function

- The complement of a function is derived by interchanging (• and +), and(1 and 0), and complementing each variable.
- Otherwise, interchange 1s to 0s in the truth table column showing F.
- The *complement* of a function IS NOT THE SAME as the *dual* of a function.
- Example: Find G(x,y,z), the complement of F(x,y,z)= xy’z’+x’yz
  Ans: G = F’ = (xy’z’ + x’yz)’

=(xy’z’)’• (x’yz)’ *De Morgan*
=(x’+y+z)•(x+y’+z’) *De Morgan* again

Note: The complement of a function can also be derived by finding the function’s *dual*, and then complementing all of the literals.

1.1.7.1 Sum-of-Products and Product-of-Sums Forms

If a function f is specified in the form of a truth table, then an expression that realizes f can be obtained by considering either the rows in the table for which f = 1,as we have already done, or by considering the rows for which f = 0.

Minterms:

1. For a function of *n* variables, a product term in which each of the *n* variables appears once is called a ***minterm***.

2. The variables may appear in a minterm either in uncomplemented or complemented form. For a given row of the truth table, the minterm is formed by including *x* if *x = 1* and by including *x'* if *x' = 0*.

Row number	x_1	x_2	x_3	Minterm	Maxterm
0	0	0	0	$m_0 = \bar{x}_1\bar{x}_2\bar{x}_3$	$M_0 = x_1 + x_2 + x_3$
1	0	0	1	$m_1 = \bar{x}_1\bar{x}_2x_3$	$M_1 = x_1 + x_2 + \bar{x}_3$
2	0	1	0	$m_2 = \bar{x}_1x_2\bar{x}_3$	$M_2 = x_1 + \bar{x}_2 + x_3$
3	0	1	1	$m_3 = \bar{x}_1x_2x_3$	$M_3 = x_1 + \bar{x}_2 + \bar{x}_3$
4	1	0	0	$m_4 = x_1\bar{x}_2\bar{x}_3$	$M_4 = \bar{x}_1 + x_2 + x_3$
5	1	0	1	$m_5 = x_1\bar{x}_2x_3$	$M_5 = \bar{x}_1 + x_2 + \bar{x}_3$
6	1	1	0	$m_6 = x_1x_2\bar{x}_3$	$M_6 = \bar{x}_1 + \bar{x}_2 + x_3$
7	1	1	1	$m_7 = x_1x_2x_3$	$M_7 = \bar{x}_1 + \bar{x}_2 + \bar{x}_3$

Fig.1.10 Three-variable minterms and maxterms.

3. To illustrate this concept, consider the truth table in Figure 1.10. We have numbered the rows of the table from 0 to 7, so that we can refer to them easily.

4. From the discussion of the binary number representation in previous section, we can observe that the row numbers chosen are just the numbers represented by the bit patterns of variables x_1, x_2 and x_3

5. The figure 1.10 shows all minterms for the three-variable table

6. For example, in the first row the variables have the values $x_1 = x_2 = x_3 = 0$, which leads to the minterm $x_1'x_2'x_3'$. In the second row $x_1 = x_2 = 0$ and $x_3 = 1$, which gives the minterm $x_1'x_2'x_3$, and so on.

7. To be able to refer to the individual minterms easily, it is convenient to identify each minterm by an index that corresponds to the row numbers shown in the figure. We will use the notation m_i to denote the minterm for row number *i*. Thus $m_0 = x_1'x_2'x_3'$, $m_1 = x_1'x_2'x_3$, and so on.

Sum-of-Products Form :

1. Afunction f can be represented by an expression that is a sum of minterms, where each minterm is ANDed with the value of *f* for the corresponding valuation of input variables.

2. For example, the two-variable minterms are $m_0 = x_1'x_2'$, $m_1 = x_1'x_2$, $m_2 = x_1x_2'$, and $m_3 = x_1x_2$. The function in Figure 1.14 can be represented as

$$f = x_1'x_2'\cdot 1 + x_1'x_2\cdot 1 + x_1x_2'\cdot 0 + x_1x_2\cdot 1$$
$$= x_1'x_2' + x_1'x_2 + x_1x_2$$
$$= m_0 + m_1 + m_3$$

3. Any function f can be represented by a sum of minterms that correspond to the rows in the truth table for which *f* = 1. The resulting implementation is functionally correct and unique, but it is not necessarily the lowest-cost implementation of *f*.

4. A logic expression consisting of product (AND) terms that are summed (ORed) is said to be in the su**m-of products (SOP) form**. If each product term is a minterm, then the expression is called a **canonical sum-of-products** for the function *f*.

5. The first step in the synthesis process is to derive a canonical sum-of-products expression for the given function. Then we can manipulate this expression, using the theorems and

properties, with the goal of finding a functionally equivalent sum-of-products expression that has a lower cost.

Example: Consider the three-variable function $f(x_1, x_2, x_3)$, specified by the truth table in Figure 1.11 and minimize the function and realize the logic diagram

Row number	x_1	x_2	x_3	$f(x_1, x_2, x_3)$
0	0	0	0	0
1	0	0	1	1
2	0	1	0	0
3	0	1	1	0
4	1	0	0	1
5	1	0	1	1
6	1	1	0	1
7	1	1	1	0

Fig.1.11 A three-variable function

Sol: To synthesize this function, we have to include the minterms m_1, m_4, m_5, m_6. Copying these minterms from Figure 1.11 leads to the following canonical sum-of-products expression for f

$$f(x_1, x_2, x_3) = x_1'x_2'x_3 + x_1x_2'x_3' + x_1x_2'x_3 + x_1'x_2x_3$$

1. This expression can be manipulated as follows

$$f(x_1, x_2, x_3) = x_1'x_3(x_2' + x_2) + x_1x_2'(x_3' + x_3)$$

$$f(x_1, x_2, x_3) = 1.\ x_1'x_3 + 1.\ x_1x_2'$$

$$f(x_1, x_2, x_3) = x_1'x_3 + x_1x_2'$$

2. This is the minimum-cost sum-of-products expression for f. It describes the circuit shown in below figure 1.12.

3. A good indication of the *cost* of a logic circuit is the total number of gates plus the total number of inputs to all gates in the circuit.

4. A good indication of the *cost* of a logic circuit is the total number of gates plus the total number of inputs to all gates in the circuit. Using this measure, the cost of the network in Figure 1.12 is 13, because there are five gates and eight inputs to the gates.

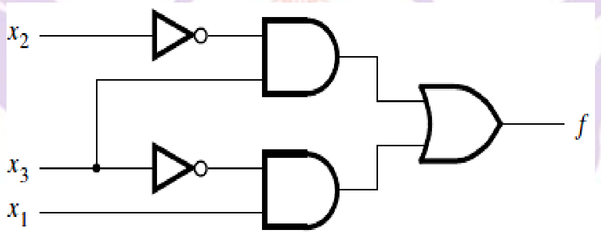


Fig. 1.12 A minimal sum-of-products realization

5. By comparison, the network implemented on the basis of the canonical sum-of- products would have a cost of 27

6. Minterms, with their row-number subscripts, can also be used to specify a given function in a more concise form

7. For example, the above function can be specified $f(x_1, x_2, x_3) = \Sigma(m_1, m_4, m_5, m_6)$ or even more simply as $f(x_1, x_2, x_3) = \Sigma m(1,4,5,6)$

The Σ sign denotes the logical sum operation. This shorthand notation is often used in practice.

Maxterms:

Row number	x_1	x_2	x_3	Minterm	Maxterm
0	0	0	0	$m_0 = \bar{x}_1\bar{x}_2\bar{x}_3$	$M_0 = x_1 + x_2 + x_3$
1	0	0	1	$m_1 = \bar{x}_1\bar{x}_2x_3$	$M_1 = x_1 + x_2 + \bar{x}_3$
2	0	1	0	$m_2 = \bar{x}_1x_2\bar{x}_3$	$M_2 = x_1 + \bar{x}_2 + x_3$
3	0	1	1	$m_3 = \bar{x}_1x_2x_3$	$M_3 = x_1 + \bar{x}_2 + \bar{x}_3$
4	1	0	0	$m_4 = x_1\bar{x}_2\bar{x}_3$	$M_4 = \bar{x}_1 + x_2 + x_3$
5	1	0	1	$m_5 = x_1\bar{x}_2x_3$	$M_5 = \bar{x}_1 + x_2 + \bar{x}_3$
6	1	1	0	$m_6 = x_1x_2\bar{x}_3$	$M_6 = \bar{x}_1 + \bar{x}_2 + x_3$
7	1	1	1	$m_7 = x_1x_2x_3$	$M_7 = \bar{x}_1 + \bar{x}_2 + \bar{x}_3$

Fig.1.13 Three-variable minterms and maxterms.

1. The principle of duality suggests that if it is possible to synthesize a function *f* by considering the rows in the truth table for which *f* = 1, then it should also be possible to synthesize *f* by considering the rows for which *f* = 0. This alternative approach uses the complements of minterms, which are called *maxterms*.

2. All possible maxterms for three variable functions are listed in Figure. We will refer to a maxterm M_j by the same row number as its corresponding minterm m_j as shown in the figure 1.13.

Product-of-Sums Form:

1.If a given function *f* is specified by a truth table, then its complement *f* can be represented by a sum of minterms for which *f* = 1, which are the rows where *f* = 0. For example, for the function in below Figure 1.14

$$f(x_1, x_2)' = (m_0 + m_1 + m_3)'$$

x_1	x_2	$f(x_1, x_2)$
0	0	1
0	1	1
1	0	0
1	1	1

Fig.1.14 Two variable function

Complement this expression using DeMorgan's theorem, the result is= $f = f(x_1, x_2) = \Sigma(m_2)'$

2. Note that we obtained this expression previously by algebraic manipulation of the canonical sum-of-products form for the function *f*. The key point here is that *f* = M(2) =M_2
Where M_2 is the maxterm for row 2 in the truth table.

3. As another example, consider the function in below Figure 1.15. The complement of this function can be represented as $f(x1, x2, x3)' = m_0 + m_2 + m_3 + m_7$

$$= x_0'x_1'x_2' + x_0'x_1x_2' + x_0'x_1x_2 + x_0x_1x_2$$

Then *f* can be expressed as

$$f = (m_0 + m_2 + m_3 + m_7)'$$
$$= M_0 . M_2 . M_3 . M_7$$
$$= (x_1+x_2+x_3)\ (x_1+x_2'+x_3)\ (x_1+x_2'+x_3')\ (x_1'+x_2'+x_3')$$

This expression represents *f* as a product of maxterms.

Row number	x_1	x_2	x_3	$f(x_1, x_2, x_3)$
0	0	0	0	0
1	0	0	1	1
2	0	1	0	0
3	0	1	1	0
4	1	0	0	1
5	1	0	1	1
6	1	1	0	1
7	1	1	1	0

Fig.1.15 A three-variable function

4. A logic expression consisting of sum (OR) terms that are the factors of a logical product (AND) is said to be of the *product-of-sums* (*POS*) form. If each sum term is a maxterm, then the expression is called a *canonical product-of-sums* for the given function.

5. Any function *f* can be synthesized by finding its canonical product-of-sums. This involves taking the maxterm for each row in the truth table for which *f* = 0 and forming a product of these maxterms.

6. We can reduce the complexity of the derived expression that comprises a product of maxterms. Using the commutative property 10*b* and the associative property 11*b* ,this expression can be written as

$$f = (x_1+x_2+x_3)\,(x_1+x_2'+x_3)\,(x_1+x_2'+x_3')\,(x_1'+x_2'+x_3')$$

7. Then, using the combining property 14*b*, the expression reduces to

$$f = (x_1+x_3)\,(x_3'+x_2')$$

8. The corresponding network is given in Figure

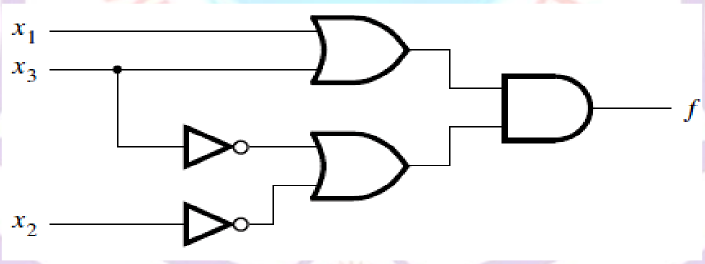


Fig.1.16 A minimal product-of-sums realization

9. The cost of this network is 13

10. Using the shorthand notation, an alternative way of specifying our sample function is

$$f(x_1, x_2) = \Pi\,(M_0, M_2)$$
$$= \Pi\, M(0, 2)$$

The Π sign denotes the logical product operation.

11. Consider the network in below figure 1.17 as a representative of general AND-OR networks. This network can be transformed into a network of NAND gates as shown.

12. First, each connection between the AND gate and an OR gate is replaced by a connection that includes two inversions of the signal: one inversion at the output of the AND gate and the other at the input of the OR gate. Such double inversion has no effect on the behavior of the network.

13. The OR gate with inversions at its inputs is equivalent to a NAND gate. Thus we can redraw the network using only NAND gates.

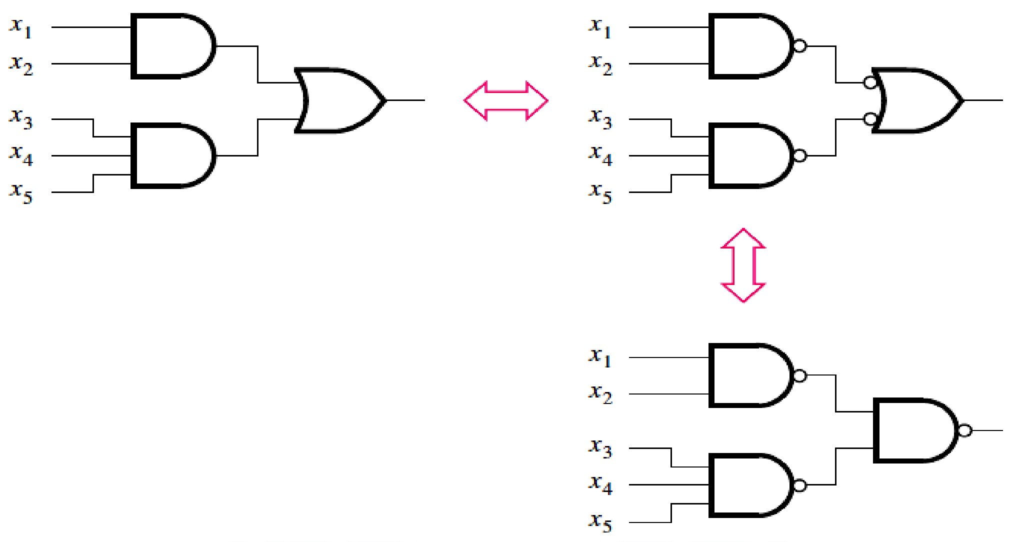


Fig.1.17 Using NAND gates to implement a sum-of-products.

14. This example shows that any AND-OR network can be implemented as a NAND-NAND network having the same topology.
15. Below figure 1.18 gives a similar construction for a product-of-sums network, which can be transformed into a circuit with only NOR gates.

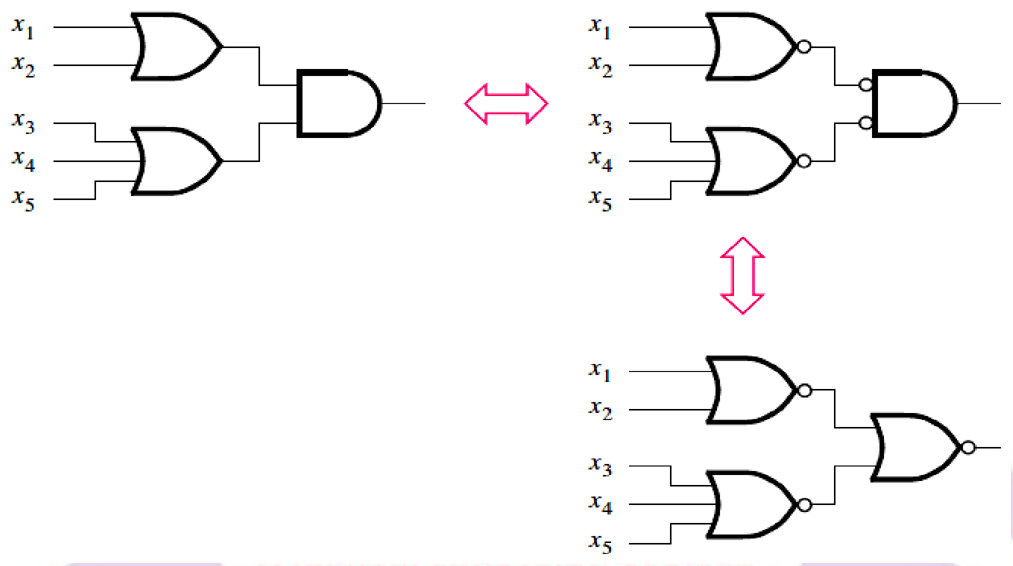


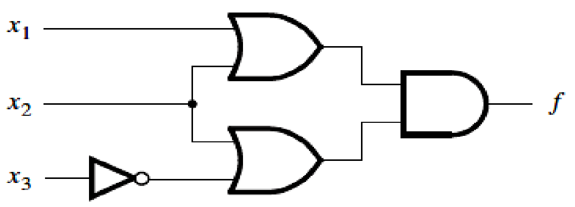
Fig.1.18 Using NOR gates to implement a product-of-sums.

16. The procedure is exactly the same but the AND gate with inversions at its inputs is equivalent to a NOR gate. Thus we can redraw the network using only NOR gates
17. The conclusion is that any OR-AND network can be implemented as a NOR-NOR network having the same topology.

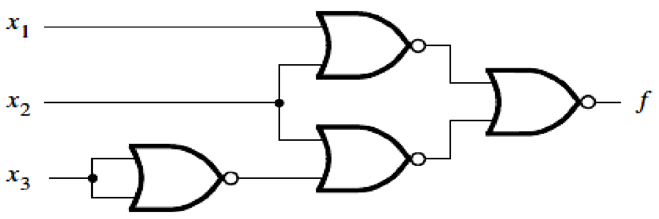
Example: Let us implement the function $f(x_1, x_2, x_3) = \Sigma m(2, 3, 4, 6, 7)$ using NOR gates only

This function can be represented by the POS expression as $f = (x_1 + x_2)(x_2 + x_3')$

- An OR-AND circuit that corresponds to this expression is shown in Figure 1.19*a*. Using the same structure of the circuit, a NOR-gate version is given in Figure 1.19*b*. Note that x_3 Is inverted by a NOR gate that has its inputs tied together.



(a) POS implementation

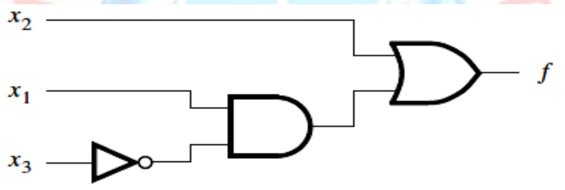


(b) NOR implementation

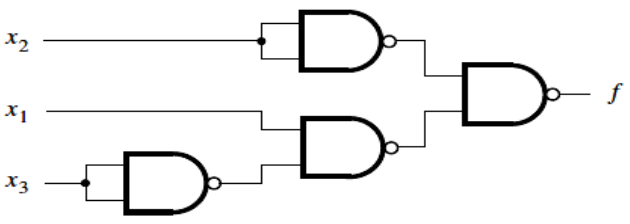
Fig.1.19 NOR-gate realization of the function

Example: Let us implement the function $f\,(x_1, x_2, x_3) = \Sigma\, m(2, 3, 4, 6, 7)$ using NAND gates only

The SOP expression of given function $f = x_2 + x_1x_3'$



(a) SOP implementation



(b) NAND implementation

Fig. 1.20 NAND-gate realization of the function

1. An AND-OR circuit that corresponds to this expression is shown in Figure 1.20*a*. Using the same structure of the circuit, a NAND -gate version is given in Figure 1.20*b*.
2. The only difference is signal x_3 passes only through an OR gate, instead of passing through an AND gate and an OR gate.
3. If we simply replace the OR gate with a NAND gate, this signal would be inverted which would result in a wrong output value. Since x_3 must either not be inverted, or it can be inverted twice, we can pass it through two NAND gates as shown in Figure 1.20b

Observe that for this circuit the output f is $f = ((x_2')' \cdot (x_1' + x_3)')'$

Applying DeMorgan's theorem, this expression becomes

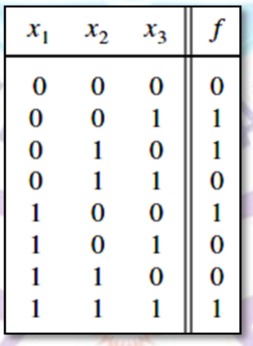
$f = x_2 + x_1x_3'$

Design Examples

1. Logic circuits provide a solution to a problem. They implement functions that are needed to carry out specific tasks. Within the framework of a computer, logic circuits provide complete capability for execution of programs and processing of data.
2. A designer of logic circuits is always deal with the same basic issues. First, it is necessary to specify the desired behavior of the circuit. Second, the circuit has to be synthesized and implemented. Finally, the implemented circuit has to be tested to verify that it meets the specifications.

Example: Three-Way Light Control

1. Assume that a large room has three doors and that a switch near each door controls a light in the room. It has to be possible to turn the light on or off by changing the state of any one of the switches.
2. As a first step, let us turn this word statement into a formal specification using a truth table. Let x_1, x_2, and x_3 be the input variables that denote the state of each switch.
3. Assume that the light is off if all switches are open. Closing any one of the switches will turn the light on. Then turning on a second switch will have to turn off the light. Thus the light will be on if exactly one switch is closed, and it will be off if two (or no) switches are closed. If the light is off when two switches are closed, then it must be possible to turn it on by closing the third switch.
4. If $f\,(x_1, x_2, x_3)$ represents the state of the light, then the required functional behavior can be specified as shown in the truth table in Figure 1.28.



x_1	x_2	x_3	f
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

Fig.1.21 Truth table for the three-way light control.

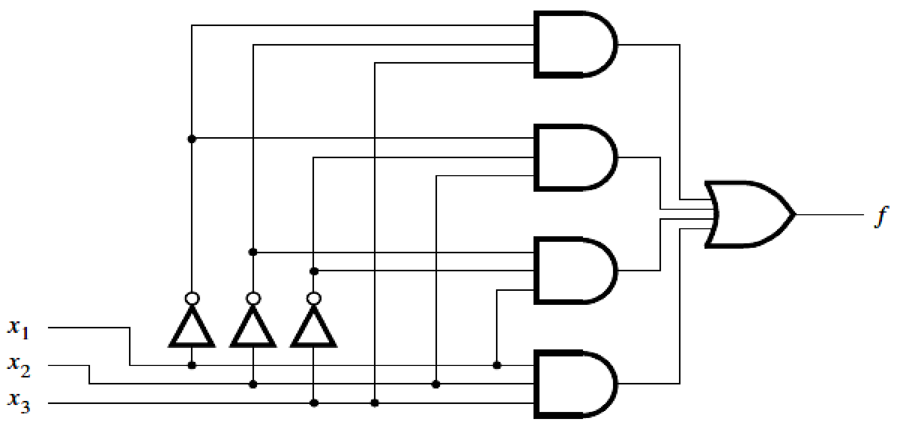
5. The canonical sum-of-products expression for the specified function is

$$f = m_1 + m_2 + m_4 + m_7$$
$$= x_1' \, x_2' x_3 + x_1' \, x_2 x_3' + x_1, x_2' \, x_3' + x_1 \, x_2 \, x_3$$

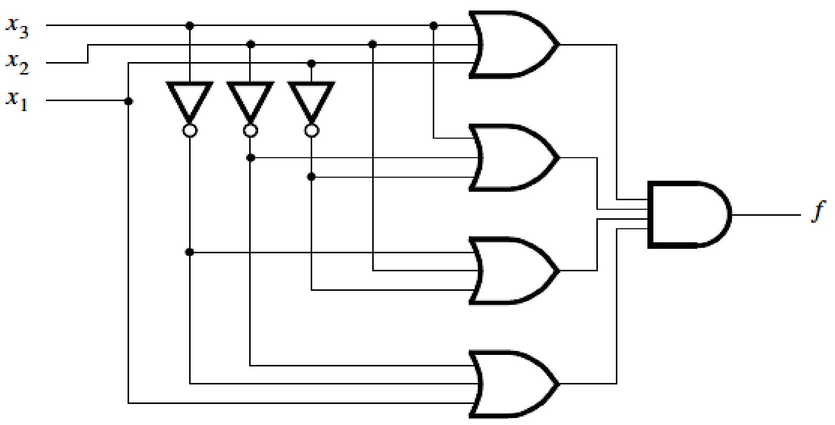
6. This expression cannot be simplified into a lower-cost sum-of-products expression. The resulting circuit is shown in Figure 1.21*a*.
7. An alternative realization for this function is in the product-of-sums form. The canonical expression of this type is

$$f = M_0 \,.\, M_3 \,.\, M_5 \,.\, M_6$$
$$= (x_1 + x_2 + x_3) \,.\, (x_1 + x_2' + x_3') \,.\, (x_1' + x_2 + x_3') \,.\, (x_1' + x_2' + x_3)$$

8. The resulting circuit is depicted in Figure 1.21*b*



(a) Sum-of-products realization



(b) Product-of-sums realization

Fig.1.21 Implementation of the function

1.1.8 Reduction of Boolean Expressions using Boolean Algebra

- Boolean algebra is a useful tool for simplifying digital circuits.
- Why do it? Simpler can mean cheaper, smaller, faster.

Procedure:

1. Multiply all variables necessary to remove parenthesis.
2. Look for identical terms. Only one of those terms to be retained & other dropped.
   Ex: AB+AB+AB+AB=AB
3. Look for a variable & its negation in the same term. This term can be dropped one.
   Ex: ABC D+ABC = ABC (D+1)=ABC .1=ABC
4. Look for pairs of terms which have the same variables,with one or more variables complemented.
5. If a variable in one term of such a pair is complemented while in the second term it is not then such terms can be combined into a single term with variable dropped.
   Ex: ABC D+ABC D'= ABC (D +D')=ABC .1=ABC

Example: Simplify F=x'yz+x'yz'+xz.
F=x'yz+x'yz'+xz
=x'y(z+z')+xz
=x'y•1+xz
=x'y+xz

Example: Prove x'y'z'+x'yz'+xyz'= x'z'+yz'
Proof:
x'y'z'+x'yz'+xyz'
=x'y'z'+x'yz'+ x'yz' +xyz'
=x'z'(y'+y)+ yz'(x'+x)
=x'z'•1+ yz'•1
=x'z'+yz'

Example: Let us minimize the boolean function given as follows using the method of algebraic manipulation-

F = ABC'D' + ABC'D + AB'C'D + ABCD + AB'CD + ABCD' + AB'CD'

Solution – The properties used here refer to the 3 most common laws mentioned above.

F = ABC' (D' + D) + AB'C'D + ACD (B + B') + ACD' (B + B')

= ABC' + AB'C'D + ACD + ACD'

= ABC' + AB'C'D + AC (D + D')

= ABC' + AB'C'D + AC

= A (BC' + C) + AB'C'D

= A (B +C) + AB'C'D

= AB + AC + AB'C'D

= AB + AC + AC'D

= AB + AC + AD

Example: Reduce the given Boolean function

F= AB(B'C+AC)

= A.B.B'.C + A.B.A.C

= A.0.C + A.B.C

= 0 + A.B.C

= ABC

Example: Reduce the given Boolean function
F= ABC' + ABC + AB
= AB (C' + C) + A'B
= AB + A'B
= B (A + A') = B. 1 = B

Example: Reduce the given Boolean function

$F = A.(A' + C)\,(A'.B + C')$

$= (A.\,A' + AC)\,(A'.B + C')$

$= AC\,(A'.B + C')$ [Since, A.A = 0]

$= AC.\,A'.B + AC.\,C'$

$= 0$

Example: Reduce the given Boolean function

$F = (A+B'+C')(A+B'+C)(A+B+C')$

$= A(A+B'+C)(A+B+C') + B'(A+B'+C)(A+B+C') + C'(A+B'+C)(A+B+C')$

$= (AA+AB'+AC)(AA+AB+AC') + (AB'+B'B'+B'C)(AB'+B'B'+B'C) +$
$(AC'+B'C'+CC')(AC'+BC'+C'C')$

$= (A+AB'+AC)(A+AB+AC') + (AB'+B'+B'C)(AB'+0+B'C) +$
$(AC'+B'C'+0)(AC'+BC'+C')$

$= (A+AC)(A+AB+AC') + (AB'+B'+B'C)(AB'+B'C) + (AC'+BC')(AC'+BC'+C')$

$= A(A+AB+AC') + (B'+B'C)(AB+BC) + (AC'+BC')(BC'+C')$

$= A(A+AC') + B'(AB+BC) + C'(AC'+BC')$

$= AA + (AB+B'BC) + (AC'C'+BC'C')$

$= A + (AB+B'C) + (AC'+BC')$

$= A + (AB) + (B'C) + (AC')$

$= A + (B'C) + (AC')$

$= A + (B'C)$

Example: Simplify the given Boolean function

$F = (A+(BC)')'(AB'+ABC')$

$= A'.BC.\,(AB' + ABC')$ [De-Morgan's Law]

$= A'.BC\,(AB'+ABC')$

$= A'.BC.\,AB' + A'.BC.\,ABC'$

$= A'.A.B.B'.C + A'.A.B.B.C.C'$ [Manipulation]

$= 0 + 0 = 0$