UNIT - I
NUMBER SYSTEMS & CODES

INTRODUCTION

Computers, uses 1's and 0's to represent the data. These 1's and 0's might bestored magnetically on a disk, or as a state in a transistor or vacuum tube. To performuseful operations on these 1's and 0's, the 1's and 0's are organized together into patterns thatmake up codes. Modern digital systems do not represent numeric values using the decimalsystem. Instead, they typically use a binary or two's complement numbering system.

Tounderstand the digital system arithmetic, we must understand how digital systems representnumbers.

The term digital is derived from the way computers perform operations, by counting digits. For many years, applications of digital electronics were related to computer systems. Today, digital technology is applied in a wide range of areas in addition to computers.

Digital Applications such as:

- Television,
- Communication systems,
- Radar,
- Navigation and Guidance systems,
- Military systems,
- Medical instrumentation,
- Industrial process control system for counting and controlling items for packaging on a conveyor line and consumer electronics use digital techniques.

Over the years digital technology has progressed from vacuum-tube circuits to discretetransistors to complex integrated circuits, some of which contain millions of transistors.

We can conclude, whether an electronic product contains digital circuitry based on the following:

- Does it have an Alphanumeric (shows letters and numbers) display?
- Does it has a memory or can it store information?
- Can the device be programmed?

If any one of the above is satisfied, then we can say that the electronic product contains digital circuitry.

DIGITAL AND ANALOG QUANTITIES:

Electronic circuits can be divided into two broad categories, digital and analog. Digital electronics involves quantities with discrete values, and analog electronics involves quantities with continuous values. Although we will be studying digital fundamentals in Switching theory and Logic Design, we should also know something about analog because many applications require both; and interfacing between analog and digital is important.

An analog quantity is one having continuous values. A digital quantity is one having a discrete set of values. Most things that can be measured quantitatively occur in nature is in analog form. For example, the air temperature changes over a continuous range of values.

Example for Analog quantity:

During a given day, the temperature does not go from, say, 70° to 71 ° instantaneously; it takes on all the infinite values in between. If we graph the temperature on a typical summer day. We would have a smooth, continuous curve similar to the curve shown in Figure below.

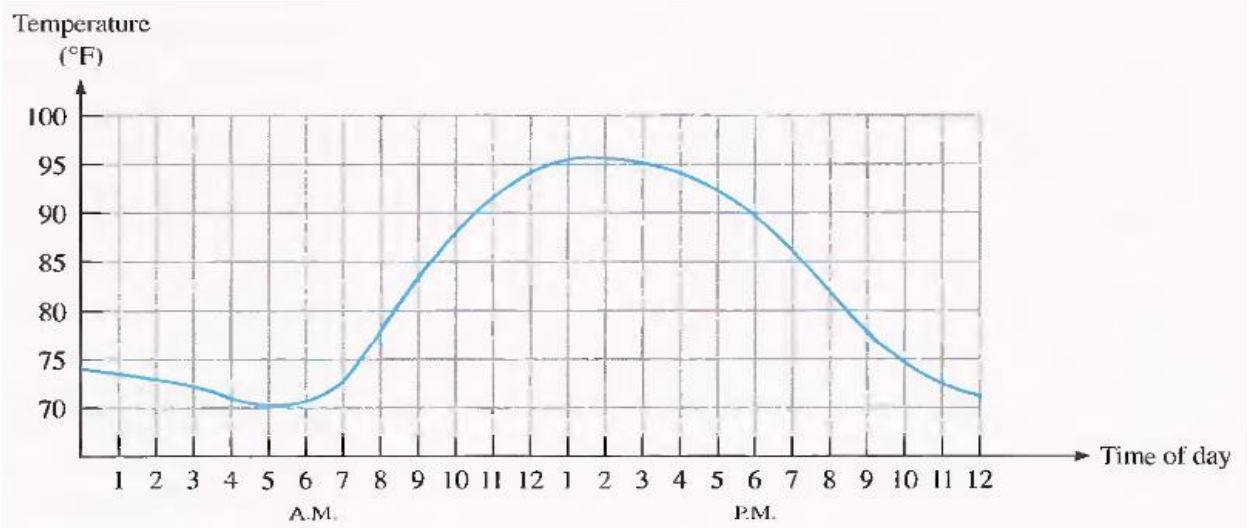


Fig: Graph of an analog quantity (temperature versus time).

Other examples of analog quantities are time, pressure, distance, and sound. Rather than graphing the temperature on a continuous basis, suppose if we just take a temperature reading every hour. Now we have sampled values representing the temperature at discrete points in time (every hour) over a 24-hour period, as indicated in Figure below. We have effectively converted an analog quantity to a form that can now be digitized by representing each sampled value by a digital code. It is important to realize that Figure below itself is not the digital representation of the analog quantity.

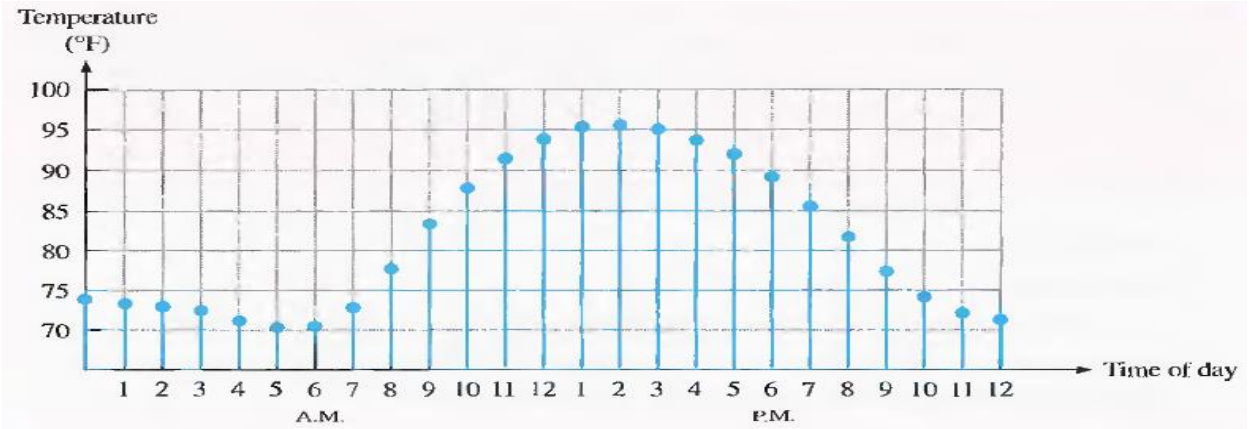


Fig: Sampled-value representation (quantization) of the analog quantity.

Each value represented by a dot can be digitized by representing it as a digital code that consists of a series of 1 s and Os.

Digital Advantage:

Digital representation has certain advantages over analog representation in electronicsapplications. Such as

- Digital data can be processed and transmitted more efficiently and reliably than, analog data.
- Generally, digital devices are easier to design using modern integrated circuits (ICs)
- Information storage is easy. For example music when converted to digital form can be stored more compactly and reproduced with greater accuracy and clarity than is possible when it is in analog form.
- Noise (unwanted voltage fluctuations) does not affect digital data much when compared to analog signals.
- Devices can be made programmable with digital.

An Analog Electronic System:

A public address system, used to amplify sound is one simple example of an application of analog electronics. The basic diagram in shown in Figure below illustrates that sound waves, which are analog in nature, are picked up by a microphone and converted to a small analog voltage called the audio signal. This voltage varies continuously as the volume and frequency of the sound changes and is applied to the input of a linear amplifier. The output of the amplifier which is an increased reproduction of input voltage goes to the speaker(s). The speaker changes the amplified audio signal back to sound waves that have a much greater volume than the original sound waves picked up by the microphone.

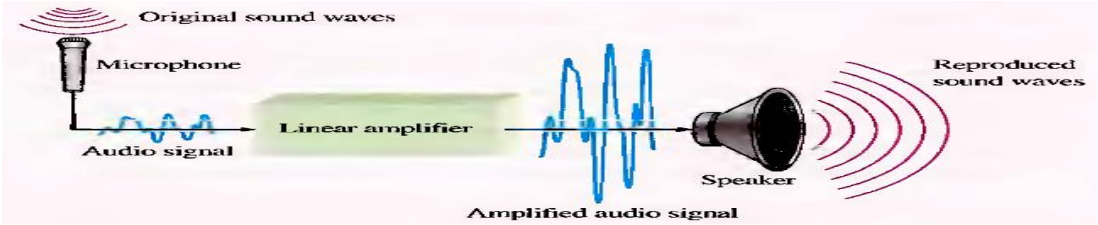
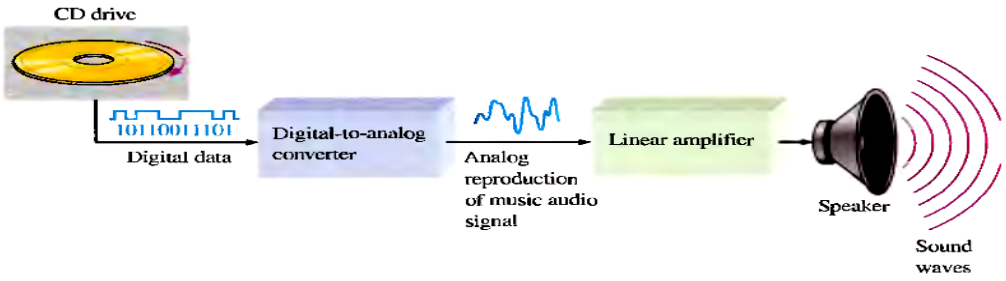


Fig: A basic audio public address system.

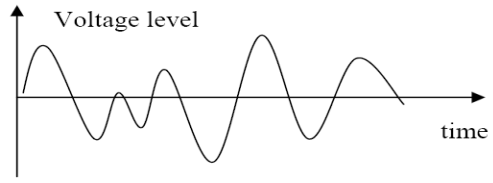
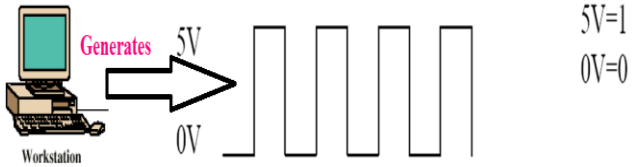
A System Using Digital and Analog Methods:

The compact disk (CD) player is an example of a system in which both digital and analog circuits are used. The simplified block diagram in shown in figure below illustrates the basic principle. Music in digital form is stored on the compact disk. A laser diode optical system picks up the digital data from the rotating disk and transfers it to the digital-to-analog converter (DAC).



The DAC changes the digital data into an analog signal that is an electrical reproduction of the original music. This signal is amplified and sent to the speaker for us to enjoy. When the music was originally recorded on the CD, a process, essentially the reverse of the one described here, using an analog-to-digital converter (ADC) was used.

Differences between analog signal and digital signal:

Analog signal	**Digital signal**
1. It is a continuous signal.	1. It is a discrete signal.
2. It is a continuously varying signal.	2. It is based on 0's and 1's.
3. Analog signal has more influence to noise.	3. Digital signal has less influence to noise.
4. Analog signals are difficult to transmit when compared to digital signals.	4. Digital signals are easier to transmit when compared to analog signals.
5. Cost of analog signal transmission is expensive.	5. Cost of digital signal transmission is less expensive.
6. Less reliable when compared to digital signals.	6. More reliable when compared to digital signals.
7. Analog signals do not provide continuous delivery, when compared to digital signals.	7. Digital signals provide better continuous delivery, when compared to analog signals.
8. Analog signal cannot be stored easily.	8. Digital signal can be stored easily.
9. More errors occur.	9. Only few errors occur.
10. Errors correction is not easy.	10. Error correction is easy.
11. Analog signals require lesser Bandwidth.	11. Digital signals require greater Bandwidth.
12. Analog signal has a slower rate of transmission when compared to digital signal.	12. Digital signal has a faster rate of transmission when compared to analog signal.
	
Example: ▪ Sound waves are continuous. ▪ Sin, cosine signals, triangular & saw tooth signals.	**Example:** ▪ Most computers used such as PCs work using digital signal. ▪ Morse code.

A REVIEW OF THE DECIMAL SYSTEM

People have been using the decimal (base 10) numbering system for so long that theymostly use. When one sees a number like "123", he doesn't think about thevalue 123; rather, he generates a mental image of how many items this value represents. Inreality, however, the number 123 represents:

$$1*10^2 + 2*10^1 + 3*10^0$$

or

$$100 + 20 + 3$$

Each digit appearing to the left of the decimal point represents a value between zero andnine times an increasing power of ten. Digits appearing to the right of the decimal pointrepresent a value between zero and nine times an increasing negative power of ten. Forexample, the value 123.456 means:

$$1*10^2 + 2*10^1 + 3*10^0 + 4*10^{-1} + 5*10^{-2} + 6*10^{-3}$$

or

$$100 + 20 + 3 + 0.4 + 0.05 + 0.006$$

Binary Formats:

Every binary number contains an infinite number of digits (or bits which is short for binary digits). Because any number of leading zero bits may precede the binary number without changing its value. For example, one can represent the number seven by:

111
00000111
000000000000111

Often several values are packed together into the same binary number. For convenience, a numeric value is assign to each bit position. Each bit is numbered as follows:

1. The rightmost bit in a binary number is bit position zero.
2. Each bit to the left is given the next successive bit number.

An eight-bit binary value uses bits zero through seven :$\mathbf{X_7\ X_6\ X_5\ X_4\ X_3\ X_2\ X_1\ X_0}$

A 16-bit binary value uses bit positions zero through fifteen:

$$\mathbf{X_{15}\ X_{14}\ X_{13}\ X_{12}\ X_{11}\ X_{10}\ X_9\ X_8\ X_7\ X_6\ X_5\ X_4\ X_3\ X_2\ X_1\ X_0}$$

Bit zero is usually referred to as the low order bit. The left-most bit is typically called the high order bit. The intermediate bits are referred by their respective bit numbers. The low order bit which is X_0 is called LEAST SIGNIFICANT BIT (LSB). The high order bit or left most bit. i.e., X15 is called MOST SIGNIFICANT BIT (MSB).

Data Organization

In mathematics a value may take an arbitrary number of bits. Digital systems, generally work with some specific number of bits. Common collections are single bits, groups of four bits (called nibbles), groups of eight bits (called bytes), groups of 16 bits (called words), and more. The sizes are not arbitrary.

Bit:

- The smallest "unit" of data on a binary computer or digital system is a single bit.
- **Bit**, an abbreviation for Binary Digit, can hold either a 0 or a 1.
- A bit is the smallest unit of information a computer can understand. Since a single bit is capable of representing only two different values (typically zero or one) one may get the impression that there are a very small number of items one can represent with a single bit. That's not true! There are an infinite number of items one can represent with a single bit. With a single bit, one can represent any two distinct items.

Examples include zero or one, true or false, on or off, male or female, and right or wrong. However, one is not limited.

Nibble:

A nibble is a collection of four bits. It wouldn't be a particularly interesting data structure except for two items:

- BCD (binary coded decimal) numbers and hexadecimal numbers. It takes four bits to represent a single BCD or hexadecimal digit. With a nibble, one can represent up to 16 distinct values. In the case of hexadecimal numbers, the values 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F are represented with four bits.
- BCD uses ten different digits (0, 1, 2, 3, 4, 5, 6, 7, 8, and 9) and requires four bits.
- In fact, any sixteen distinct values can be represented with a nibble, but hexadecimal and BCD digits can be represented with a single nibble.

Byte:

- Computer memory must be able to store letters, numbers, and symbols. A single bit by itself cannot be of much use. Bits are combined to represent some meaningful data.
- A group of eight bits is called a byte. It can represent a character and is the smallest addressabledatum (data item) on the most of the digital systems (e.g. 80 × 86 microprocessor).
- The mostimportant data type is the byte.

  Note: Byte also contains exactly 2 Nibbles.

Word:A word is a group of 16 bits. Bits in a word are numbered starting from zero on up to fifteen.

Double Word:A double word is exactly what its name implies, a pair of words. Therefore, a double word quantity is 32 bits long as shown in Fig. below.

Note: This double word can be divided into a high order word and a low order word, or four different bytes, or eight different nibbles.

Binary Equivalents:

- 1 Nibble (or nibble) = 4 Bits
- 1 Byte = 2 nibbles = 8 Bits
- 1 Kilobyte (KB) $=2^{10}$ Bytes = 1024 Bytes
- 1 Megabyte (MB) $=2^{20}$ Bytes = 1024 Kilo Bytes = 1,048,576 Bytes
- 1 Gigabyte (GB) $=2^{30}$ Bytes = 1024 Mega Bytes = 1,073,741,824 Bytes

BINARY NUMBERING SYSTEM

Most modern digital systems operate using binary logic. The digital systems represent values using two voltage levels (usually 0V and +5V). With two such levels one can represent exactly two different values. These could be any two different values, but by convention we use the values zero and one. These two values, coincidentally, correspond to the two digits used by the binary numbering system.

Computers use binary numbers to select memory locations. Each location is assigned a unique number called an address. Some Pentium microprocessors, for example, have 32 address lines which can select 2^{32} (4,294,967,296) unique locations.

- The binary number system is another way to represent quantities. It is less complicated than the decimal system because it has only two digits.
- The decimal system with its ten digits is a base-ten system; the binary system with its two digits is a base-two system.
- The two binary digits (bits) are 1 and 0. The position of a 1 or 0 in a binary number indicates its weight, or value within the number, just as the position of a decimal digit determines the value of that digit.
- The weights in a binary number are based on powers of two.

DECIMAL NUMBER	BINARY NUMBER			
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
10	1	0	1	0
11	1	0	1	1
12	1	1	0	0
13	1	1	0	1
14	1	1	1	0
15	1	1	1	1

From the above Table, we observe that, four bits are required to count from zero to 15. In general, with 'n' bits you can count up to a number equal to 2^n - 1.

Largest decimal number $= 2^n - 1$

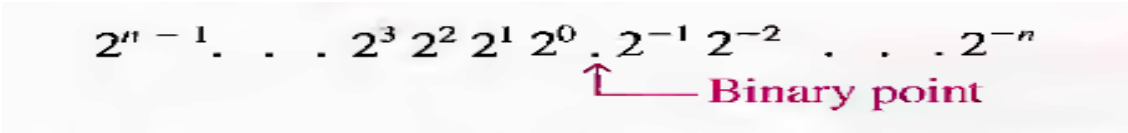
For example, with five bits (n = 5) you can count from 0 to 31.i.e., $2^5 - 1 = 32 - 1 = 31$

Note: The value of a bit is determined by its position in the number.

The Weighting Structure of Binary Numbers

A binary number is a weighted number.

- The right-most bit is the LSB (least significant bit)in a binary whole number and has a weight of $2^0 = 1$.
- The weights increase from right toleft by a power of two for each bit.
- The left-most bit is the MSB (most significant bit); its weight depends on the size of the binary number.
- Fractional numbers can also be represented in binary by placing bits to the right of the binary point, just as fractional decimal digits are placed to the right of the decimal point.
- The left-most bit is the MSB in a binary fractional number and has a weight of $2^{-1} = 0.5$.
- The fractional weights decrease from left to right by a negative power of two for each bit.
- The weight structure of a binary number is



Where n is the number of bits from the binary point.

- Thus, all the bits to the left of the binary point have weights that are positive powers of two, as previously discussed for whole numbers.
- All bits to the right of the binary point have weights that are negative powers of two, or fractional weights.

The powers of two and their equivalent decimal weights for an 8-bit binary whole number and a 6-bit binary fractional number are shown in Table.

POSITIVE POWERS OF TWO (WHOLE NUMBERS)									NEGATIVE POWERS OF TWO (FRACTIONAL NUMBER)					
2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}
256	128	64	32	16	8	4	2	1	1/2	1/4	1/8	1/16	1/32	1/64
									0.5	0.25	0.125	0.0625	0.03125	0.015625

Note: The weight or value of a bit increases from right to left in a binary number.

OCTAL NUMBERS:

- The octal number system provides a convenient way to express binary numbers and codes. The octal number system has a base of 8.
- However, it is used less frequently than hexadecimal in combination with computers and microprocessors to express binary quantities for input and output purposes.
- The octal number system is composed of eight digits, which are 0, 1, 2, 3, 4, 5, 6, and 7.
- To obtain the equivalent octal number above 7, divide the number with 8 and write the remainders obtained at each stage from bottom to top to give the equivalent octal number:
  10, 11, 12, 13, 14, 15, 16, 17, 20, 21...
- Counting in octal is similar to counting in decimal, except that the digits 8 and 9 are not used. To distinguish octal numbers from decimal numbers or hexadecimal numbers, we will use the subscript 8 to indicate an octal number.
- For example, 15_8 in octal is equivalent to 13_{10} in decimal and D in hexadecimal. The octal number is also represented by “o" or a "Q" following an octal number.

HEXADECIMAL NUMBERS:

- The hexadecimal number system has sixteen characters; it is used primarily as a compact way of displaying or writing binary numbers because it is very easy to convert between binary and hexadecimal.
- As we are probably aware that, long binary numbers are difficult to read and write because it is easy to drop or transpose a bit. Since computers and microprocessors understand only 1's and 0's, it is necessary to use these digits when we program in "machine language." Imagine writing a sixteen bit instruction for a microprocessor system in 1's and 0's. It is much more efficient to use hexadecimal or octal.
- Hexadecimal is widely used in computer and microprocessor applications because of the compact representation of long string of binary 1's and 0's.
- The hexadecimal number system has a base of 16; i.e., it is composed of 16 numeric and alphabetic characters.
- Most digital systems process binary data in groups that are multiples of four bits, making the hexadecimal number very convenient because each hexadecimal digit represents a 4-bit binary number.

The hexadecimal number system consists of digits 0-9 and letters A-F as shown in table below.

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

- The use of letters A, B, C, D, E, and F to represent numbers may seem strange at first, but keep in mind that any number system is only a set of sequential symbols. If we understand what quantities these symbols represent, then the form of the symbols themselves is less important once you get adjusted using them.
- Hexadecimal numbers are designated using the subscript 16 to avoid confusion with decimal numbers. Sometimes you may see an "h" following a hexadecimal number.

Counting in Hexadecimal:

- To obtain the equivalent hexadecimal number above 15, divide the number with 16 and write the remainders obtained at each stage from bottom to top to give the equivalent hexadecimal number as follows: 10,11, 12,13,14,15, 16, 17, 18, 19, lA, IB, 1C, ID, IE, IF, 20, 21, 22, 23, 24,25,26,27,28,29, 2A, 2B, 2C, 2D, 2E, 2F, 30, 31,. . .
- With two hexadecimal digits, we can count up to FF_{16} , which is decimal 255. To count beyond this, three hexadecimal digits are needed. For instance, $(100)_{16}$ is decimal 256, $(101)_{16}$ are decima1257, and so forth. The maximum 3-digit hexadecimal number is $(FFF)_{16}$ or decimal 4095. The maximum 4-digit hexadecimal number is $(FFFF)_{16}$, which is decimal 65,535.

NOTE:

With computer memories in the gigabyte (GB) range, specifying a memory address in binary is quite, cumbersome. For example, it takes 32 bits to specify an address in a 4 GB memory. It is much easier to express a 32-bit code using 8 hexadecimal digits. Hexadecimal is a convenient way to represent binary numbers.

NUMBER BASE CONVERSIONS:

The humans use decimal number system while the 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 into decimal equivalent while displaying the result to the humans. However, dealing with large quantity of binary number of many bits is inconvenient for humans. Therefore, octal or hexadecimal numbers are used as a shorthand means of expressing large binary numbers. But it is necessary to keep it in mind that the digital circuits and systems work strictly in binary; we use octal and hexadecimal for the operator convenience.

CONVERSION DECIMAL NUMBER TO ANY RADIX:

Conversion of decimal number to any radix / base can be achieved in two steps:

- Conversion of integer part to any radix / base number by successive division method &
- Conversion of fractional part to any radix / base number by successive multiplication method.

Steps in successive division method to convert integer part to any radix / base number:

1. Divide the integer part of decimal number by desired base number, store quotient (Q) and remainder (R).
2. Consider quotient as the new decimal number and repeat step 1 untill quotient becomes 1.
3. List the remainders from bottom to top (i.e. in reverse order).

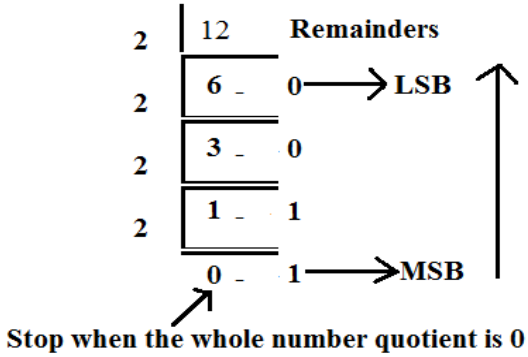
Steps in successive multiplication method to convert fractional part to any radix / base number:

1. Multiply the fractional part of decimal number by desired radix / base.
2. Store the integer part of the product as carry and fractional part as new fractional part.
3. Repeat steps 1 and 2 untill fractional part of product becomes 0 or until you have as many digits necessary for your application.
4. Write carries from top to bottom to get the desired radix / base number.

Example:

Steps for converting the decimal number 12 to binary:

- Divide the decimal number by 2 producing a dividend and a remainder. This number is the LSB (least significant bit of the desired binary number).
- Again divide the dividend obtained above by 2. This produces another dividend and remainder. The remainder is the next digit of the binary number.
- Continue this process of division until the dividend becomes 0. The remainder obtained in the final division is the MSB (most significant bit of the binary number).



Note:
To obtain the binary number for a given decimal number,

Steps:

1. Divide the decimal number by 2 until the quotient is 0.

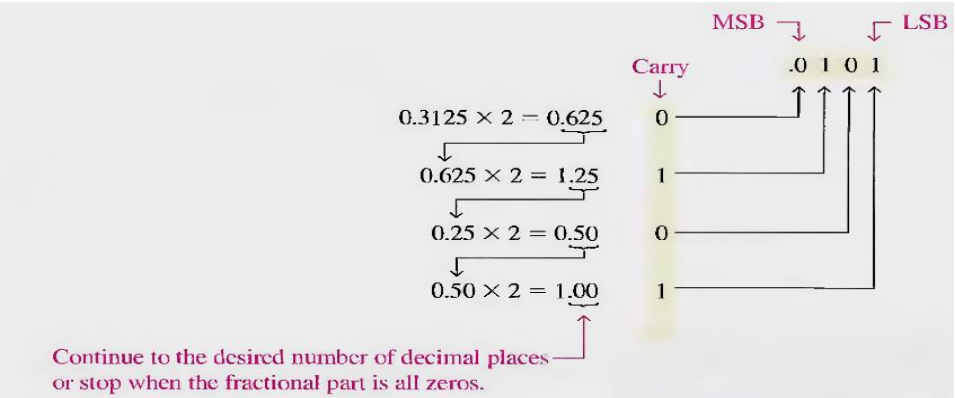
2. And then , Start writing the Remainders form the bottom to top to obtain the Binary number.

Repeated Multiplication by 2:

As we have seen, decimal whole numbers can be converted to binary by repeated division by 2.

Decimal fractions can be converted to binary by repeated multiplication by 2.

For example, to convert the decimal fraction 0.3125 to binary, begin by multiplying 0.3125 by 2 and then multiplying each resulting fractional part of the product by 2 until the fractional product is zero or until the desired number of decimal places is reached. The carry digits, or carries, generated by the multiplications produce the binary number. The first carry produced is the MSB, and the last carry is the LSB. This procedure is illustrated as follows:

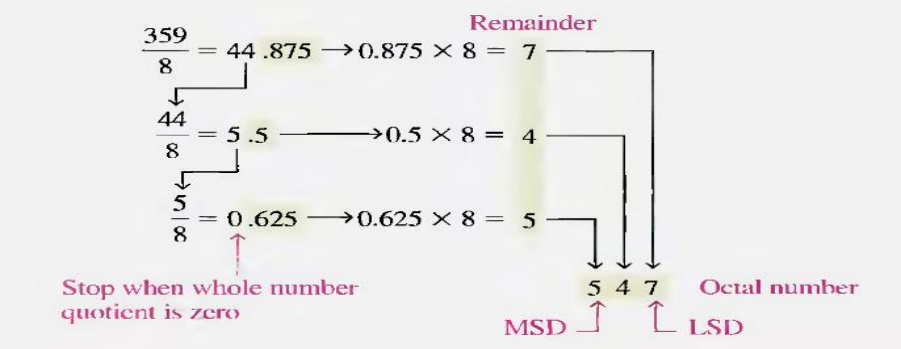


Decimal-to-Octal Conversion:

Converting a decimal number to an octal number is obtained by repeated division-by- 8 methods, which is similar to the method used in the conversion of decimal numbers to binary or to hexadecimal.

The decimal number 359 to octal is converted as follows.

Each successive division by 8 yields a remainder that becomes a digit in the equivalent octal number. The first remainder generated is the least significant digit (LSD).



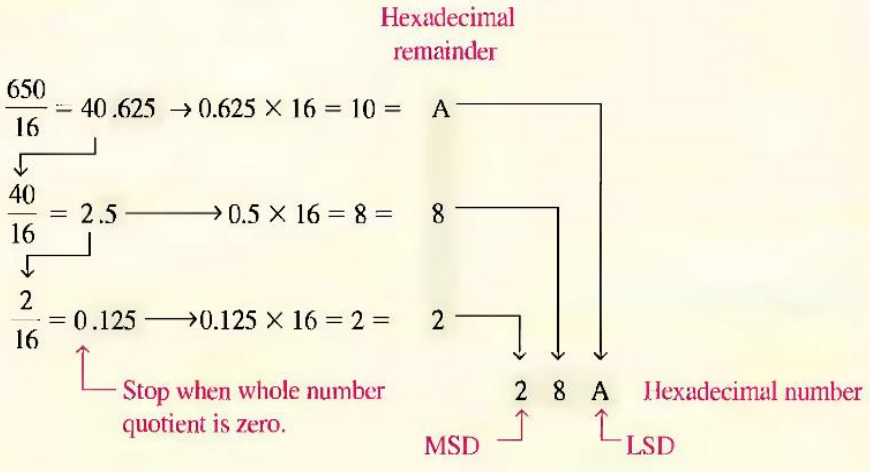
Decimal-to-Hexadecimal Conversion:

- Repeated division of a decimal number by 16 will produce the equivalent hexadecimal number, formed by the remainders of the divisions.
- The first remainder produced is the least significant digit (LSD).
- Each successive division by 16 yields a remainder that becomes a digit in the equivalent hexadecimal number. This procedure is similar to repeated division by 2 for decimal-to-binary conversion.
- When a quotient has a fractional part, the fractional part is multiplied by the divisor to get the remainder.

EXAMPLE 2–28

Convert the decimal number 650 to hexadecimal by repeated division by 16.

Solution



CONVERSION OF ANY RADIX / BASE TO DECIMAL NUMBER:

$$N = A_{n-1} X r^{n-1} + A_{n-2} X r^{n-2} + \ldots + A_1 X r^1 + A_0 X r^0 + A_{-1} X r^{-1} + A_{-2} X r^{-2} + \ldots + A_{-m} X r^{-m}$$

Where **N = Number in decimal,**
A = Digit,
r = Radix / Base of a number system,
n = The number of digits in the integer portion of the number &
m = The number of digits in the fractional portion of the number

Binary-to-Decimal Conversion

The decimal value of any binary number can be found by adding the weights of all bits that are 1 and discarding the weights of all bits that are 0.

Note: Add the weights of all 1's in a binary number to get the decimal value.

EXAMPLE

Convert the binary whole number 1101101 to decimal.

Solution Determine the weight of each bit that is a 1, and then find the sum of the weights to get the decimal number.

Weight: $2^6\ 2^5\ 2^4\ 2^3\ 2^2\ 2^1\ 2^0$
Binary number: 1 1 0 1 1 0 1

$$1101101 = 2^6 + 2^5 + 2^3 + 2^2 + 2^0$$
$$= 64 + 32 + 8 + 4 + 1 = \mathbf{109}$$

EXAMPLE

Convert the fractional binary number 0.1011 to decimal.

Solution Determine the weight of each bit that is a 1, and then sum the weights to get the decimal fraction.

Weight: $2^{-1}\ 2^{-2}\ 2^{-3}\ 2^{-4}$
Binary number: 0 . 1 0 1 1

$$0.1011 = 2^{-1} + 2^{-3} + 2^{-4}$$
$$= 0.5 + 0.125 + 0.0625 = \mathbf{0.6875}$$

Octal-to-Decimal Conversion:

Since the octal number system has a base of eight, each successive digit position is an increasing power of eight, beginning in the right-most column with 8^0. The evaluation of an octal number in terms of its decimal equivalent is obtained by multiplying each digit by its weight and summing the products, as explained below for 2374_8

Weight: $8^3\ 8^2\ 8^1\ 8^0$
Octal number: 2 3 7 4

$$2374_8 = (2 \times 8^3) + (3 \times 8^2) + (7 \times 8^1) + (4 \times 8^0)$$
$$= (2 \times 512) + (3 \times 64) + (7 \times 8) + (4 \times 1)$$
$$= 1024 + 192 + 56 + 4 = 1276_{10}$$

Hexadecimal-to-Decimal Conversion:

- One way to find the decimal equivalent of a hexadecimal number is to first convert the hexadecimal number to binary and then convert from binary to decimal.

EXAMPLE

Convert the following hexadecimal numbers to decimal:

(a) $1C_{16}$ (b) $A85_{16}$

Solution Remember, convert the hexadecimal number to binary first, then to decimal.

(a) 1 C

$\overbrace{0001}\overbrace{1100} = 2^4 + 2^3 + 2^2 = 16 + 8 + 4 = \mathbf{28_{10}}$

(b) A 8 5

$\overbrace{1010}\overbrace{1000}\overbrace{0101} = 2^{11} + 2^9 + 2^7 + 2^2 + 2^0 = 2048 + 512 + 128 + 4 + 1 = \mathbf{2693_{10}}$

- Another way to convert a hexadecimal number to its decimal equivalent is to multiply the decimal value of each hexadecimal digit by its weight and then take the sum of these products. The weights of a hexadecimal number are increasing powers of 16 (from right to left). For a 4-digit hexadecimal number, the weights are

16^3	16^2	16^1	16^0
4096	256	16	1

EXAMPLE

Convert the following hexadecimal numbers to decimal:

(a) $E5_{16}$ (b) $B2F8_{16}$

Solution Recall from Table . that letters A through F represent decimal numbers 10 through 15, respectively.

(a) $E5_{16} = (E \times 16) + (5 \times 1) = (14 \times 16) + (5 \times 1) = 224 + 5 = \mathbf{229_{10}}$

(b) $B2F8_{16} = (B \times 4096) + (2 \times 256) + (F \times 16) + (8 \times 1)$
$= (11 \times 4096) + (2 \times 256) + (15 \times 16) + (8 \times 1)$
$= 45{,}056 + 512 + 240 + 8 = \mathbf{45{,}816_{10}}$

Binary-to-Octal Conversion:

Conversion of a binary number to an octal number is the reverse of the octal-to-binary conversion.

The procedure is as follows:

- Start with the right-most group of three bits and, moving from right to left, convert each 3-bit group to the equivalent octal digit.
- If the number of bits available for the left-most group is less than 3; add either one or two zeros to make a complete group of 3-bits. These leading zeros do not affect the value of the binary number.

EXAMPLE

Convert each of the following binary numbers to octal:

(a) 110101 (b) 101111001 (c) 100110011010 (d) 11010000100

Solution

(a) $\underbrace{110}\underbrace{101}$ $6\ 5 = \mathbf{65_8}$

(b) $\underbrace{101}\underbrace{111}\underbrace{001}$ $5\ 7\ 1 = \mathbf{571_8}$

(c) $\underbrace{100}\underbrace{110}\underbrace{011}\underbrace{010}$ $4\ 6\ 3\ 2 = \mathbf{4632_8}$

(d) $\underbrace{011}\underbrace{010}\underbrace{000}\underbrace{100}$ $3\ 2\ 0\ 4 = \mathbf{3204_8}$

Octal-to-Binary Conversion:

Each octal digit can be represented by a 3-bit binary number ($8 = 2^3$, where 3 represents the number of bits used to encode), so it is very easy to convert from octal to binary. Each octal digit is represented by three bits as shown below.

OCTAL DIGIT	0	1	2	3	4	5	6	7
BINARY	000	001	010	011	100	101	110	111

To convert an octal number to a binary number, simply replace each octal digit with the appropriate three bits.

EXAMPLE 2–31

Convert each of the following octal numbers to binary:

(a) 13_8 (b) 25_8 (c) 140_8 (d) 7526_8

Solution

(a) 1 3 → 001011 (b) 2 5 → 010101 (c) 1 4 0 → 001100000 (d) 7 5 2 6 → 111101010110

Note: Octal is a convenient way to represent binary numbers, but it is not as commonly used as Hexadecimal.

Binary-to-Hexadecimal Conversion:

Converting a binary number to hexadecimal is a straightforward procedure. Simply break the binary number into 4-bit groups, starting at the right-most bit and replaces each 4-bit group with the equivalent hexadecimal symbol.

EXAMPLE

Convert the following binary numbers to hexadecimal:

(a) 1100101001010111 (b) 1111110001011010 01

Solution

(a) 1100 1010 0101 0111 → C A 5 7 $= CA57_{16}$

(b) 0011 1111 0001 0110 1001 → 3 F 1 6 9 $= 3F169_{16}$

Two zeros have been added in part (b) to complete a 4-bit group at the left.

Hexadecimal-to-Binary Conversion:

To convert a hexadecimal number to a binary number, reverse the process and replace each hexadecimal symbol with the appropriate four bits.

EXAMPLE

Determine the binary numbers for the following hexadecimal numbers:

(a) $10A4_{16}$ (b) $CF8E_{16}$ (c) 9742_{16}

Solution

(a) 1 0 A 4 → 1000010100100

(b) C F 8 E → 1100111110001110

(c) 9 7 4 2 → 1001011101000010

In part (a), the MSB is understood to have three zeros preceding it, thus forming a 4-bit group.

It is clear that it is much easier to deal with a hexadecimal number than with the equivalent binary number. Since conversion is so easy, the hexadecimal system is widely used for representing binary numbers in programming, printouts, and displays.

Note: Conversion between hexadecimal and binary is direct and easy.

Hex to Octal and Octal to Hex Conversion

These conversions are done through the binary conversion. Recall that, a group of 4-bitsrepresent a hexadecimal digit and a group of 3-bits represent an octal digit.

Hex to Octal Conversion

1. Convert the given hexadecimal number into binary.
2. Starting from right make groups of 3-bits and designate each group an octal digit.

Example. *Convert* $(1A3)_{16}$ *into octal.*

Solution.

1. Converting hex to binary

$$(1\ A\ 3)_{16} = \underbrace{0001}_{1}\ \underbrace{1010}_{A}\ \underbrace{0011}_{3}$$

2. Grouping of 3-bits

$$(1A3)_{16} = \underset{0}{\underbrace{000}}\ \underset{6}{\underbrace{110}}\ \underset{4}{\underbrace{100}}\ \underset{3}{\underbrace{011}}$$

so $(1A3)_{16} = (0643)_8 = (643)_8$

Octal to Hex Conversion

1. Convert the given octal number into binary.
2. Starting from right make groups of 4-bits and designate each group as a Hexadecimaldigit.

Example. *Convert* $(76)_8$ *into hexadecimal.*

Solution. 1. Converting octal to binary

$$(76)_8 = \underbrace{111}_{7}\ \underbrace{110}_{6}$$

2. Grouping of 4-bits

$$(76)_8 = \underset{3}{\underbrace{11}}\ \underset{E}{\underbrace{1110}} \equiv \underset{3}{\underbrace{0011}}\ \underset{E}{\underbrace{1110}}$$

$\therefore$ $(76)_8 = (3E)_{16}$

BINARY ARITHMETIC:

Binary arithmetic is essential in all digital computers and in many other types of digital systems. To understand digital systems, we must know the basics of binary addition, subtraction,multiplication, and division.

Binary Addition:

The four basic rules for adding binary digits (bits) are as follows:

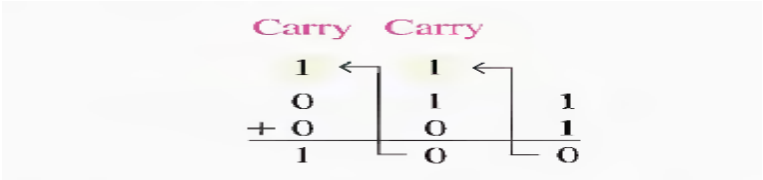
Augend	Addend	Carry	Sum	Result
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	0	10

The procedure of adding 2 binary numbers is same as that of 2 decimal numbers. Addition is carried out from LSB and it proceeds to higher significant bits, adding the carry resulting from the addition of two previous bits each time.

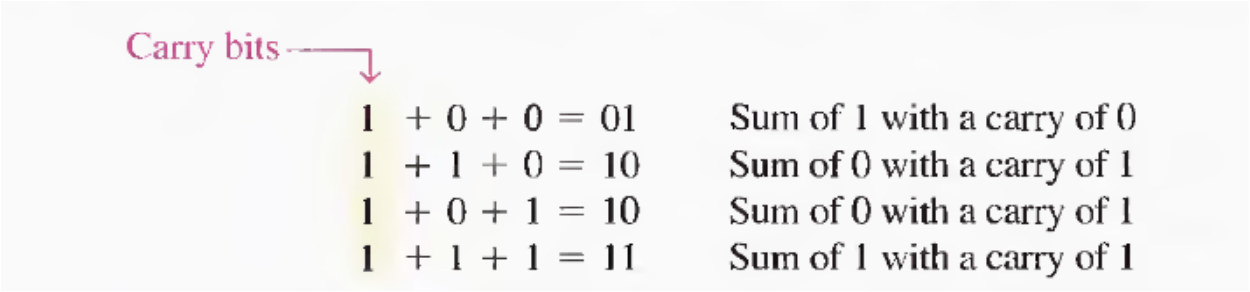
- 0+0=0 Sum of 0 with a carry of 0
- 0+1=1 Sum of 1 with a carry of 0
- 1 + 0= 1 Sum of 1 with a carry of 0
- 1+1=10 Sum of 0 with a carry of 1

Note: Remember, in binary 1+1 = 10, not 2.

The first three rules result in a single bit and in the fourth rule the addition of two1's yields a binary two (10). When binary numbers are added, the last condition create10 a sum of 0 in a given column and a carry of 1 over to the next column to the left, as illustrated in the following addition of 11 + 1:



- In the right column, 1 + 1 = 0 with a carry of 1 to the next column to the left. In the middle column, 1 + 1 + 0 = 0 with a carry of 1 to the next column to the left. In the left column, 1 + 0 + 0 = 1.
- When there is a carry of 1, we have a situation in which three bits are being added (a bitin each of the two numbers and a carry bit). This situation is explained as follows:



EXAMPLE

Add the following binary numbers:

(a) 11 + 11 (b) 100 + 10 (c) 111 + 11 (d) 110 + 100

Solution The equivalent decimal addition is also shown for reference.

(a)	11	3	(b)	100	4	(c)	111	7	(d)	110	6
	+11	+3		+10	+2		+ 11	+3		+100	+4
	110	6		**110**	6		**1010**	10		**1010**	10

Example *Add the binary numbers:*

(a) 1010 and 1101 *(b) 0110 and 1111*

Solution.

(a)

	1	0	1	0
(+)	1	1	0	1
1	0	1	1	1

↑ Carry

(b)

	(1)	(1)	←	Carry
	0	1	1	0
(+)	1	1	1	1
1	0	1	0	1

↑ Carry

Binary Subtraction:

The four basic rules for subtracting bits are as follows:

Minuend	Subtrahend	Borrow	Difference	Result
0	0	0	0	00
0	1	1	1	11
1	0	1	0	10
1	1	0	0	00

Binary subtraction is also carried out in a similar method to decimal subtraction. The subtraction is carried out from LSB and proceeds to the higher significant bits. When borrow is 1, as in the second row, this is to be subtracted from the next higher binary bit as is performed in decimal subtraction.

Actually, the subtraction between two numbers can be performed in three ways,

1. The Direct method,
2. The r's complement method, and
3. The (r-1)'s complement method.

0-0 = 0
1-1 = 0
1-0 = 1
10-1 = 1 0 - 1 with a borrow of 1

Note: Remember in binary 10 - 1 = 1, not 9.

When subtracting numbers, we sometimes have to borrow from the next column to the left. A borrow is required in binary only when we try to subtract a 1 from a 0. In this case, when a 1 is borrowed from the next column to the left, a 10 is created in the column being subtracted, and the last of the four basic rules just listed must be applied.

The binary subtraction and equivalent decimal subtractions are as follows:

Subtraction Using the Direct Method

The direct method of subtraction uses the concept of borrow. In this method, we borrow a 1 from a higher significant position when the minuend digit is smaller than the corresponding subtrahend digit.

Example *Using the direct method to perform the subtraction 1001 – 1000.*

Solution:

```
    1 0 0 1
(–) 1 0 0 0
    0 0 0 1
```

Example *Using the direct method to perform the subtraction 1000 – 1001.*

Solution.

```
      1 0 0 0
(–)   1 0 0 1
End carry → 1 1 1 1 1
```

End carry has to be ignored.

Answer: 1111 = (2's complement of 0001).

When the minuend is smaller than the subtrahend the result of subtraction is negative and in the direct method the result obtained is in 2's complement form. So to get back the actual result we have to perform the 2's complement again on the result thus obtained.

But to tackle the problem shown in Example 1.29 we have applied a trick. When a

EXAMPLE

Perform the following binary subtractions:

(a) 11 − 01 (b) 11 − 10

Solution

(a)	11	3	(b)	11	3
	− 01	− 1		− 10	− 2
	10	2		**01**	1

No borrows were required in this example. The binary number 01 is the same as 1.

EXAMPLE

Subtract 011 from 101.

Solution

101	5
−011	−3
010	2

Let's examine exactly what was done to subtract the two binary numbers since a borrow is required. Begin with the right column.

Left column: When a 1 is borrowed, a 0 is left, so 0 − 0 = 0.

Middle column: Borrow 1 from next column to the left, making a 10 in this column, then 10 − 1 = 1.

Right column: 1 − 1 = 0

```
  0
 1'01
−011
 010
```

Binary Multiplication:

The four basic rules for multiplying bits are as follows:

- 0X0=0
- 0X1=0
- 1x0=0
- 1 X1 =1

Note:

- Binary multiplication of two bits is the same as multiplication of the decimal digits 0 and 1.Multiplication is performed with binary numbers in the same manner as with decimal numbers.
- It involves forming partial products, shifting each successive partial product left one place, and then adding all the partial products. The equivalent decimal multiplications are shown below for reference.

EXAMPLE

Perform the following binary multiplications:

(a) 11×11 (b) 101×111

Solution

(a)	11	3	(b)	111	7
	× 11	× 3		× 101	× 5
Partial products	11	9	Partial products	111	35
	+11			000	
	1001			+111	
				100011	

Example *Multiply the following binary numbers:*
(a) 0111 and 1101 and (b) 1.011 and 10.01.
Solution.

(a) 0111 × 1101

			0	1	1	1		Multiplicand
		×	1	1	0	1		Multiplier
			0	1	1	1		
		0	0	0	0			Partial
	0	1	1	1				Products
0	1	1	1					
1	0	1	1	0	1	1		Final Product

(b) 1.011 × 10.01

			1.	0	1	1		Multiplicand
		×	1	0.	0	1		Multiplier
			1	0	1	1		
		0	0	0	0			Partial
	0	0	0	0				Products
1	0	1	1					
1	1 .	0	0	0	1	1		Final Product

Binary Division:

Division in binary follows the same procedure as division in decimal. The equivalent decimal divisions are also given.

Dividend	Divisor	Result
0	0	Not Allowed
0	1	0
1	0	Not Allowed
1	1	1

Note: A calculator can be used to perform arithmetic operations with binary numbers as long as the capacity of the calculator is not exceeded.

EXAMPLE

Perform the following binary divisions:

(a) $110 \div 11$ **(b)** $110 \div 10$

Solution

$$\begin{array}{r} \mathbf{10} \\ 11\overline{)110} \\ \underline{11} \\ 000 \end{array} \qquad \begin{array}{r} 2 \\ 3\overline{)6} \\ \underline{6} \\ 0 \end{array}$$

(a)

$$\begin{array}{r} \mathbf{11} \\ 10\overline{)110} \\ \underline{10} \\ 10 \\ \underline{10} \\ 00 \end{array} \qquad \begin{array}{r} 3 \\ 2\overline{)6} \\ \underline{6} \\ 0 \end{array}$$

(b)

Example 1.31. *Divide the following binary numbers:*

(a) 11001 and 101 and (b) 11110 and 1001.

Solution.

(a) $11001 \div 101$

$$\begin{array}{r} 1\ 0\ 1 \\ 1\ 0\ 1\overline{)1\ 1\ 0\ 0\ 1} \\ \underline{1\ 0\ 1} \\ 0\ 0\ 1\ 0\ 1 \\ \underline{1\ 0\ 1} \\ 0\ 0\ 0 \end{array}$$

Answer: 101

(b) $11110 \div 1001$

$$\begin{array}{r} 1\ 1. \\ 1\ 0\ 0\ 1\overline{)1\ 1\ 1\ 1\ 0} \\ \underline{1\ 0\ 0\ 1} \\ 0\ 1\ 1\ 0\ 0 \\ \underline{1\ 0\ 0\ 1} \\ 1\ 1 \end{array}$$

Answer: 11

1'S AND 2'S COMPLEMENTS OF BINARY NUMBERS:

The l's complement and the 2's complement of a binary number are important because they are the means for the representation of negative numbers in digital technology. The method of 2's complementarithmetic is commonly used in computers to handle negative numbers.

Finding the 1's Complement:

The l's complement of a binary number is found by changing all 1's to 0's and all 0's to 1's, as shown below:

```
1 0 1 1 0 0 1 0     Binary number
↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓
0 1 0 0 1 1 0 1     1's complement
```

Note: Change each bit in a number to get the l's complement.

The simplest way to obtain the l's complement of a binary number with a digital circuit is to use parallel inverters (NOT circuits), as shown in Figure 2-2 for an 8-bit binary number.

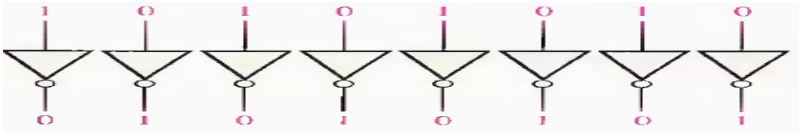


Fig: Inverters used to obtain the 1's complement of a binary number.

Finding the 2' s Complement:

The 2's complement of a binary number is found by adding 1 to the LSB of the l's complement.

2's complement = (l's complement) + 1

Note: Add 1 to the l' s complement to get the 2's complement.

EXAMPLE

Find the 2's complement of 10110010.

Solution

```
  10110010    Binary number
  01001101    1's complement
+        1    Add 1
  01001110    2's complement
```

An **alternative method** of finding the 2's complement of a binary number is as follows:

1. Start at the right with the LSB and write the bits as they are up to and including the first 1.

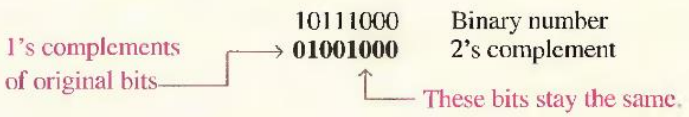
2. Take the 1's complements of the remaining bits.

Note: Change all bits to the left of the least significant 1 to get 2's complement.

EXAMPLE

Find the 2's complement of 10111000 using the alternative method.

Solution



The 2's complement of a negative binary number can be realized using inverters and anadder, as shown in Figure below. Figure below gives, how an 8-bit number can be converted toits 2's complement by first inverting each bit (taking the l's complement) and then adding 1 to the l's complement with an adder circuit.

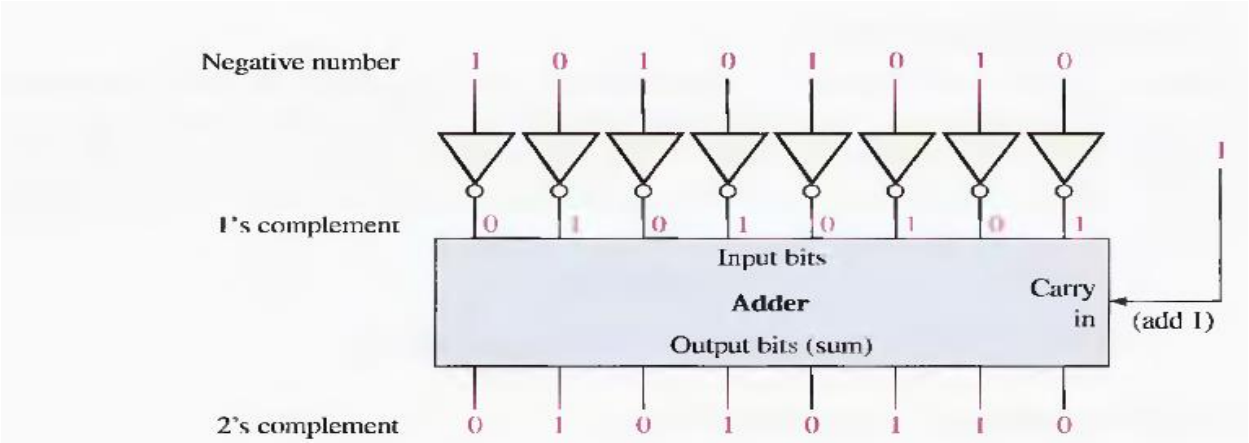


Fig: Obtaining the 2's complement of a negative binary number.

To convert from an l's or 2's complement back to the true (un-complemented) binary form, use the same two procedures described previously.

- To go from the l's complement back to true binary, reverse all the bits.
- To go from the 2's complement form back to true binary, take the 1's complement of the 2's complement number and add 1 to the least significant bit.

SIGNED NUMBERS:

Digital systems, such as the computer, must be able to handle both positive and negative numbers. A signed binary number consists of both sign and magnitude information. Thesign indicates whether a number is positive or negative, and the magnitude is the value of the number.
There are three forms in which signed integer (whole) numbers can be represented in binary:

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

Of thesethe 2's complement is the most important and the sign-magnitude is the least used. Non-integer and very large or small numbers can be expressed in floating-point format.

To explain the effect of these 3 representations, we consider 4-bit binary representation as in below table. Carefully observe the differences in three methods.

Decimal	*Signed Magnitude*	*1's complement*	*2's complement*
+0	0 0 0 0	0 0 0 0	0 0 0 0
+1	0 0 0 1	0 0 0 1	0 0 0 1
+2	0 0 1 0	0 0 1 0	0 0 1 0
+3	0 0 1 1	0 0 1 1	0 0 1 1
+4	0 1 0 0	0 1 0 0	0 1 0 0
+5	0 1 0 1	0 1 0 1	0 1 0 1
+6	0 1 1 0	0 1 1 0	0 1 1 0
+7	0 1 1 1	0 1 1 1	0 1 1 1
–8	—	—	1 0 0 0
–7	1 1 1 1	1 0 0 0	1 0 0 1
–6	1 1 1 0	1 0 0 1	1 0 1 0
–5	1 1 0 1	1 0 1 0	1 0 1 1
–4	1 1 0 0	1 0 1 1	1 1 0 0
–3	1 0 1 1	1 1 0 0	1 1 0 1
–2	1 0 1 0	1 1 0 1	1 1 1 0
–1	1 0 0 1	1 1 1 0	1 1 1 1
–0	1 0 0 0	1 1 1 1	—

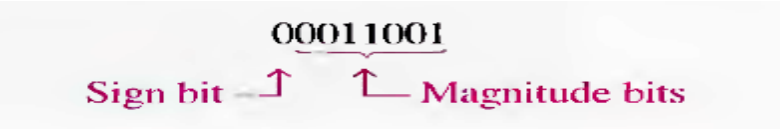
From the table, it is clear that both signed Magnitude and 1's complement methodsintroduce two zeros +0 and – 0 which is awkward. This is not the case with 2's complement.This is one among the reasons that why all the modern digital systems use 2's complementmethod for the purpose of signed representation. From the above table, it is also clear thatin signed representation $\frac{2^n}{2}$ positive numbers and$\frac{2^n}{2}$ negative numbers can be representedwith n-bits. Out of 2^ncombinations of n-bits, first $\frac{2^n}{2}$ combinations are used to denote thepositive numbers and next $\frac{2^n}{2}$ combinations represent the negative numbers.

The Sign Bit:

The left-most bit in a signed binary number is the sign bit, which tells you whether the number is positive or negative. A '0' sign bit indicates a positive number, and a '1' sign bit indicates a negative number.

Sign-Magnitude Form:

When a signed binary number is represented in sign-magnitude, the left-most bit is the sign bit and the remaining bits are the magnitude bits. The magnitude bits are in true (un-complemented) binary for both positive and negative numbers. For example, the decimal number + 25 is expressed as an 8-bit signed binary number using the sign-magnitude form as



The decimal number - 25 is expressed as1001100l

- Notice that the only difference between + 25 and - 25 is the sign bit because the magnitude bits are in true binary for both positive and negative numbers.
- In the sign-magnitude form, a negative number has the same magnitude bits as the corresponding positive number but the sign bit is a 1 rather than a zero.

Example: Find the decimal equivalent of the following binary numbers assuming the binary numbers have been represented in sign- magnitude form.

a. 0101100 b. 101000 c.1111 d.011011

Solution.

(*a*) Sign bit is 0, which indicates the number is positive.
Magnitude $101100 = (44)_{10}$
Therefore $(0101100)_2 = (+44)_{10}$.

(*b*) Sign bit is 1, which indicates the number is negative.
Magnitude $01000 = (8)_{10}$
Therefore $(101000)_2 = (-8)_{10}$.

(*c*) Sign bit is 1, which indicates the number is negative.
Magnitude $111 = (7)_{10}$
Therefore $(1111)_2 = (-7)_{10}$.

(*d*) Sign bit is 0, which indicates the number is positive.
Magnitude $11011 = (27)_{10}$
Therefore $(011011)_2 = (+27)_{10}$.

l's Complement Form: Positive numbers in 1's complement form are represented the same way as the positive signmagnitude numbers. Negative numbers, however, are the l's complements of the corresponding positive numbers.

Example: using eight bits, the decimal number -25 is expressed as the 1's complement of + 25 (00011001) as11100110

In the l's complement form, a negative number is the l's complement of the corresponding positive number.

2's Complement Form:

Positive numbers in 2's complement form are represented the same way as in the sign magnitude and l's complement forms. Negative numbers are the 2's complements of the corresponding positive numbers.

Example: The eight bits 2's complement form of the decimal number -25 is equivalent to 2's complement of +25 (00011001) i.e.11100111

In the 2's complement form, a negative number is the 2's complement of the corresponding positive number.

Note:

Computers use the 2's complement for negative integer numbers in all arithmetic operations. The reason is that subtraction of a number is the same as adding the 2' complement of the number. Computers form the 2's complement by inverting the bits and adding 1, using special instructions that produce the same result as the adder.

EXAMPLE

Express the decimal number −39 as an 8-bit number in the sign-magnitude, 1's complement, and 2's complement forms.

Solution First, write the 8-bit number for +39.

00100111

In the *sign-magnitude form*, −39 is produced by changing the sign bit to a 1 and leaving the magnitude bits as they are. The number is

10100111

In the *1's complement form*, −39 is produced by taking the 1's complement of +39 (00100111).

11011000

In the *2's complement form*, −39 is produced by taking the 2's complement of +39 (00100111) as follows:

```
 11011000   1's complement
+       1
 11011001   2's complement
```

The Decimal Value of Signed Numbers:

Sign-magnitude:

Decimal values of positive and negative numbers in the sign-magnitude form are determined by summing the weights in all the magnitude bit positions where there are 1's and ignoring those positions where there are zeros. The sign is determined by examination of the sign bit (i.e., Left Most bit).

EXAMPLE

Determine the decimal value of this signed binary number expressed in sign-magnitude: 10010101.

Solution The seven magnitude bits and their powers-of-two weights are as follows:

2^6	2^5	2^4	2^3	2^2	2^1	2^0
0	0	1	0	1	0	1

Summing the weights where there are 1s,

$$16 + 4 + 1 = 21$$

The sign bit is 1; therefore, the decimal number is **−21.**

1's Complement:

- Decimal values of positive numbers in the l's complement form are determined by summing the weights in all bit positions where there are 1's and ignoring those positions where there are 0's.
- Decimal values of negative numbers are determined by assigning a negative value to the weight of the sign bit, summing all the weights where there are 1's, and adding 1 to the result.

EXAMPLE

Determine the decimal values of the signed binary numbers expressed in 1's complement:

(a) 00010111 **(b)** 11101000

Solution **(a)** The bits and their powers-of-two weights for the positive number are as follows:

-2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
0	0	0	1	0	1	1	1

Summing the weights where there are 1s,

$$16 + 4 + 2 + 1 = +23$$

(b) The bits and their powers-of-two weights for the negative number are as follows. Notice that the negative sign bit has a weight of -2^7 or -128.

-2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
1	1	1	0	1	0	0	0

Summing the weights where there are 1s,

$$-128 + 64 + 32 + 8 = -24$$

Adding 1 to the result, the final decimal number is

$$-24 + 1 = -23$$

2's Complement:

Decimal values of positive and negative numbers in the 2's complement form are determined by summing the weights in all bit positions where there are 1's and ignoring those positions where there are zeros. The weight of the sign bit in a negative number is given a negative value.

EXAMPLE

Determine the decimal values of the signed binary numbers expressed in 2's Complement:

(a) 01010110

(b) 10101010

Solution **(a)** The bits and their powers-of-two weights for the positive number are as follows:

-2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
0	1	0	1	0	1	1	0

Summing the weights where there are 1s,

$$64 + 16 + 4 + 2 = +86$$

(b) The bits and their powers-of-two weights for the negative number are as follows. Notice that the negative sign bit has a weight of $-2^7 = -128$.

-2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
1	0	1	0	1	0	1	0

Summing the weights where there are 1s,

$$-128 + 32 + 8 + 2 = -86$$

From above examples, we can see why the 2's complement form is preferred for representingsigned integer numbers:

To convert to decimal, it simply requires a summation of weights regardless of whether the number is positive or negative.

The 1's complement is not preferred because of the following reasons:

- The l's complement system requires adding 1 to the summation of weights for negative numbers but not for positive numbers.
- Also, the 1's complement form is generally not used because two representations of zero (00000000 or 11111111) are possible.

Range of Signed Integer Numbers That Can Be Represented:

We have used 8-bit numbers because the 8-bit grouping is common in most computers and has been given the special name byte. With one byte or eight bits, we can represent 256 different numbers. With two bytes or sixteen bits, you can represent 65,536 different numbers. With four bytes or 32 bits, you can represent 4.295×10^9 different numbers.

The formula for finding the number of different combinations of n bits is

Total combinations = 2^n

For 2's complement signed numbers, the range of values for n-bit numbers is

Range = $-(2^{n-1})$ to $+(2^{n-1} - 1)$

where in each case there is one sign bit and n - 1 magnitude bits.

For example, with four bits we can represent numbers in 2's complement ranging from $-(2^3) = -8$ to $2^3 - 1 = +7$. Similarly, with eight bits you can go from - 128 to + 127, with sixteen bits you can go from - 32,768 to + 32,767, and so on.

Signed integer representation.	**Range of signed integer representations.**
Sign magnitude	$-(2^{n-1} - 1)$ to $+(2^{n-1} - 1)$
1's complement	$-(2^{n-1} - 1)$ to $+(2^{n-1} - 1)$
2's complement	$-(2^{n-1})$ to $+(2^{n-1} - 1)$

ARITHMETIC OPERATIONS WITH SIGNED NUMBERS

Subtraction using 1's complement:

- Binary subtraction can be performed by adding the 1's complement of the subtrahend to the minuend.
- If a carry is generated, then the result is positive and in true form. Add carry to the LSB position to get the final result called end around - carry.
- If the subtrahend is larger than the minuend, then no carry is generated. The answer obtained is in 1's complement and is negative. To get a true answer take the 1's complement of the result and put a negative sign in front.

Example:

Subtract $(1100)_2$ from $(1001)_2$ using the 1's complement method. Also subtract using the direct method and compare.

Solution.

Direct Subtraction		1's complement method		
1 0 0 1		1 0 0 1		(+)
− 1 1 0 0	1's complement →	0 0 1 1		
Carry → 1 1 1 0 1		1 1 0 0		
2's complement 0 0 1 1	1's complement →	0 0 1 1		
True result 0 0 1 1	True result	− 0 0 1 1		

- In the direct method, whenever a larger number is subtracted from a smaller number, the result obtained is in 2's complement form and opposite in sign. To get the true result we have to discard the carry and make the 2's complement of the result obtained and put a negative sign before the result.
- In 1's complement subtraction, no carry is obtained and the result obtained is in 1's complement form. To get the true result we have to make the 1's complement of the result obtained and put negative sign before the result.

Example: Subtract $(1010)_2$ from $(1001)_2$ using the 1's complement method. Also subtract using the direct method and compare.

Solution.

Direct Subtraction		1's complement method		
	1 0 0 1		1 0 0 1	(+)
	− 1 0 1 0	2's complement →	0 1 1 0	
Carry →	1 1 1 1 1		1 1 1 1	
2's complement	0 0 0 1	2's complement →	0 0 0 1	
True result	−0001	True result	−0001	

Subtraction using 2's complement:

In the last section, we learned how signed numbers are represented in three different forms. In this section, we will learn how signed numbers are added, subtracted, multiplied, and divided. Because the 2's complement form for representing signednumbers is the most widely used in computers and microprocessor-based systems.

Addition:

The two numbers in an addition are the addend and the augend. The result is the sum.

There are four cases that can occur when two signed binary numbers are added.

1. Both numbers positive ,
2. Positive number with magnitude larger than negative number ,
3. Negative number with magnitude larger than positive number &
4. Both numbers negative.

Let's take one case at a time using 8-bit signed numbers as examples. The equivalent decimal numbers are shown below for reference.

Both numbers positive:

00000111	7
+ 00000100	+ 4
00001011	11

Note: Addition of two positive numbers yields a positive number.

The sum is positive and is therefore in true (un-complemented) binary.

Positive number with magnitude larger than negative number:

	00001111	15
	+ 11111010	+ −6
Discard carry →	1 00001001	9

The final carry bit is discarded. The sum is positive and therefore in true (un-complemented) binary.

Note: Addition of a positive number and a smaller negative number yields a positive number.

Negative number with magnitude larger than positive number:

00010000	16
+ 11101000	+ −24
11111000	−8

The sum is negative and therefore in 2's complements form.

Both numbers negative:

	11111011	−5
	+ 11110111	+ −9
Discard carry ⟶	1 11110010	−14

The final carry bit is discarded. The sum is negative and therefore in 2's complements form. In a computer, the negative numbers are stored in 2's complement form.

The addition process is very simple: Add the two numbers and discard any final carry bit.

Note: Addition of a positive number and a larger negative number or two negative numbers yields a negative number in 2's complement.

Subtraction using 2's complement:

- Binary subtraction can be performed by adding the 2's complement of the subtrahend to the minuend.
- If a carry is generated, discard the carry.
- If the subtrahend is larger than the minuend, then no carry is generated. The answer obtained is in 2's complement and is negative.
- To get a true answer take the 2's complement of the result and put a negative sign in front.

Example: Subtract $(0111)_2$ from $(1101)_2$ using the 2's complement method. Also subtract using the direct method and compare.

Solution.

Direct Subtraction	1's complement method	
1 1 0 1	1 1 0 1	(+)
− 0 1 1 1	2's complement ⟶ 1 0 0 1	
0 1 1 0	Carry ⟶ 1 0 1 1 0	
	Discard Carry 0 1 1 0 (Result)	

Note: In 2's complement subtraction, no carry is obtained and the result obtained is in 2's complement form. To get the true result we have to make the 2's complement of the result obtained and put negative sign before the result.

Advantage: The End-Around carry operation present in the 1's complement method is not present in 2's complement.

Comparison between 1's and 2's complements

A comparison between 1's and 2's complement reveals the advantages and disadvantages of each.

1. The 1's complement has the advantage of being easier to implement by digitals components (Viz. inverter) since only thing to be done is to change the 1's to 0's and vice versa. To implement 2's complement we have two ways:
   - By finding out the 1's complement of the number and then adding 1 to the LSB of the 1's complement, and
   - By leaving all leading 0s in the LSB positions and the first 1 unchanged, and only then changing all 1s to 0s and vice versa.

2. During subtraction of two numbers by a complement method, the 2's complement is advantageous since only one arithmetic addition is required. The 1's complement requires two arithmetic additions when an end –around carry occurs.

3. The 1's complement has an additional disadvantage of having two arithmetic zeros: one with all 0s and one with all 1s. The 2's complement has only one arithmetic zero. The fact is explained as follows:

We consider the subtraction of two equal binary numbers 1010 – 1010.
Using 1's complement:

```
   1010
+  0101  (1's complement of 1010)
+  1111  (negative zero)
```

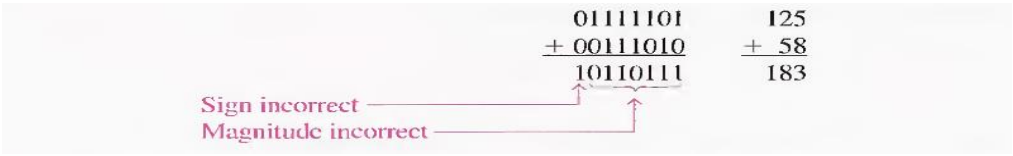
We complement again to obtain (– 0000) (positive zero).
Using 2's complement:

```
   1010
+  0110  (2's complement of 1010)
+  0000
```

In this 2's complement method no question of negative or positive zero arises.

Overflow Condition:

- When two numbers are added and the number of bits required to represent the sum exceeds the number of bits in the two numbers, an overflow results as indicated by an incorrect sign bit.
- An overflow can occur only when both numbers are positive or both numbers are negative. The following 8-bit example will illustrate this condition.



In this example the sum of 183 requires eight magnitude bits. Since there are seven magnitude bits in the numbers (one bit is the sign), there is a carry into the sign bit which produces the overflow indication.

Numbers are Added Two at a Time:

Now let's look at the addition of a string of numbers, added two at a time. This can be accomplished by adding the first two numbers, then adding the third number to the sum of the first two, then adding the fourth number to this result, and so on. This is how computers add strings of numbers. The addition of numbers taken two at a time is as follows:

EXAMPLE

Add the signed numbers: 01000100, 00011011, 00001110, and 00010010.

Solution The equivalent decimal additions are given for reference.

68	01000100	
+ 27	+ 00011011	Add 1st two numbers
95	01011111	1st sum
+ 14	+ 00001110	Add 3rd number
109	01101101	2nd sum
+ 18	+ 00010010	Add 4th number
127	**01111111**	Final sum

Subtraction:

Subtraction is a special case of addition. For example, subtracting +6 (the subtrahend) from +9 (the minuend) is equivalent to adding -6 to +9. Basically, the subtraction operation changes the sign of the subtrahend and adds it to the minuend. The result of a subtraction is called the difference.

The sign of a positive or negative binary number is changed by taking its 2's complement.

For example, when you take the 2's complement of the positive number 00000100 (+4), we get 11111100, which is -4 as the following sum-of-weights evaluation shows:

-128 + 64 + 32 + 16 + 8 + 4 = -4

As another example, when you take the 2's complement of the negative number 11101101 (-19), we get 00010011, which are + 19 as the following sum-of-weights evaluation shows:

16 + 2 + 1 = 19

Since subtraction is simply an addition with the sign of the subtrahend changed, the process is stated as follows:

To subtract two signed numbers, take the 2's complement of the subtrahend and add. Discard any final carry bit.

EXAMPLE:

Perform each of the following subtractions of the signed numbers:

(a) 00001000 − 00000011 **(b)** 00001100 − 11110111

(c) 11100111 − 00010011 **(d)** 10001000 − 11100010

Solution Like in other examples, the equivalent decimal subtractions are given for reference.

(a) In this case, 8 − 3 = 8 + (−3) = 5.

```
          00001000    Minuend (+8)
        + 11111101    2's complement of subtrahend (−3)
Discard carry → 1 00000101    Difference (+5)
```

(b) In this case, 12 − (−9) = 12 + 9 = 21.

```
  00001100    Minuend (+12)
+ 00001001    2's complement of subtrahend (+9)
  00010101    Difference (+21)
```

(c) In this case, −25 − (+19) = −25 + (−19) = −44.

```
          11100111    Minuend (−25)
        + 11101101    2's complement of subtrahend (−19)
Discard carry → 1 11010100    Difference (−44)
```

(d) In this case, −120 − (−30) = −120 + 30 = −90.

```
  10001000    Minuend (−120)
+ 00011110    2's complement of subtrahend (+30)
  10100110    Difference (−90)
```

Octal Arithmetic: 7's and 8's Complement Arithmetic

Subtraction using 7's Complement:

The 7's complement of an octal number can be found by subtracting each digit in the number from 7. The 8's complement can be obtained by subtracting the LSB from 8 & rest of each digit in the number from 7. The 7's and 8's complement of the octal digits 0 to 7 is shown in table below.

- The method of subtraction using 7's complement method is same as 1's complement method in binary system. Here also the carry obtained is added to the result to get the true result.
- And as in the previous case, if the minuend is larger than subtrahend, no carry is obtained and the result is obtained in 7's complement form. To get the true result we have to perform 7's complement over the result obtained and put negative sign in front.
- Similarly, the method of subtraction using 8's complement method is same as 2's complement method in binary system. Here also the carry obtained is discarded to get the true result.
- And as in the previous case, if the minuend is larger than subtrahend, no carry is obtained and the result is obtained in 8's complement form. To get the true result we have to perform 8's complement over the result obtained and put negative sign in front.

Octal Digit	7's Complement	8's Complement
0	7	8
1	6	7
2	5	6
3	4	5
4	3	4
5	2	3
6	1	2
7	0	1

Subtraction Using 7's Complement

Example *Subtract $(372)_8$ from $(453)_8$ using the 7's complement method. Also subtract using the direct method and compare.*

Solution.

Direct Subtraction		7's complement method	
4 5 3		4 5 3	(+)
− 3 7 2	7's complement →	4 0 5	
6 1		1 0 6 0	
	Add Carry →	1	
		$(6\ 1)_8$ (Result)	

Example *Subtract $(453)_8$ from $(372)_8$ using the 7's complement method. Also subtract using the direct method and compare.*

Solution.

Direct Subtraction		7's complement method	
3 7 2		3 7 2	(+)
4 5 3	7's complement →	3 2 4	
1 7 1 7		7 1 6	
Discard Carry 7 1 7	7's complement →	6 1	

8's complement : (61)

True result : $(-61)_8$ True result : $(-61)_8$

Note: In the direct method, whenever a larger number is subtracted from a smaller number, the result obtained is in 8's complement form and opposite in sign. To get the true result we have to discard the carry and make the 8's complement of the result obtained and put a negative sign in front.

Subtraction using 8's Complement:

Example *Subtract $(256)_8$ from $(461)_8$ using the 8's complement method. Also subtract using the direct method and compare.*

Solution.

Direct Subtraction		8's complement method	
4 6 1		4 6 1	(+)
− 2 5 6	8's complement →	5 2 2	
2 0 3	Carry →	1 2 0 3	
	Discard Carry	$(2\ 0\ 3)_8$ (Result)	

Example *Subtract $(461)_8$ from $(256)_8$ using the 8's complement method. Also subtract using the direct method and compare.*

Solution.

Direct Subtraction		8's complement method	
2 5 6		2 5 6	(+)
− 4 6 1	8's complement →	3 1 7	
1 5 7 5		5 7 5	
Discard Carry 5 7 5	8's complement →	2 0 3	
8's complement 2 0 3			
True result $(-203)_8$	True result	$(-203)_8$	

Note: In the direct method, whenever a larger number is subtracted from a smaller number, the result obtained is in 8's complement form and opposite in sign. To get the true result we have to discard the carry and make the 8's complement of the result obtained and put a negative sign in front.

Hexadecimal Addition:

Addition can be done directly with hexadecimal numbers by remembering that the hexadecimal digits 0 through 9 are equivalent to decimal digits 0 through 9 and that hexadecimal digits A through F are equivalent to decimal numbers 10 through 15. Addition of twohexadecimal numbers is as follows:

EXAMPLE

Add the following hexadecimal numbers:

(a) $23_{16} + 16_{16}$ (b) $58_{16} + 22_{16}$ (c) $2B_{16} + 84_{16}$ (d) $DF_{16} + AC_{16}$

Solution

(a) 23_{16} + 16_{16} = 39_{16}

right column: $3_{16} + 6_{16} = 3_{10} + 6_{10} = 9_{10} = 9_{16}$
left column: $2_{16} + 1_{16} = 2_{10} + 1_{10} = 3_{10} = 3_{16}$

(b) 58_{16} + 22_{16} = $7A_{16}$

right column: $8_{16} + 2_{16} = 8_{10} + 2_{10} = 10_{10} = A_{16}$
left column: $5_{16} + 2_{16} = 5_{10} + 2_{10} = 7_{10} = 7_{16}$

(c) $2B_{16}$ + 84_{16} = AF_{16}

right column: $B_{16} + 4_{16} = 11_{10} + 4_{10} = 15_{10} = F_{16}$
left column: $2_{16} + 8_{16} = 2_{10} + 8_{10} = 10_{10} = A_{16}$

(d) DF_{16} + AC_{16} = $18B_{16}$

right column: $F_{16} + C_{16} = 15_{10} + 12_{10} = 27_{10}$
$27_{10} - 16_{10} = 11_{10} = B_{16}$ with a 1 carry
left column: $D_{16} + A_{16} + 1_{16} = 13_{10} + 10_{10} + 1_{10} = 24_{10}$
$24_{10} - 16_{10} = 8_{10} = 8_{16}$ with a 1 carry

Hexadecimal Subtraction:

Weknow that, the 2's complement allows us to subtract by adding binary numbers. Since a hexadecimal number can be used to represent a binary number, it can also be used to represent the 2's complement of a binary number.

There are three ways to get the 2's complement of a hexadecimal number. Method 1 is the most common and easiest to use. Methods 2 and 3 are alternate methods.

Method 1: Convert the hexadecimal number to binary. Take the 2's complement of the binary number. Convert the result to hexadecimal. This is shown in Figure below.

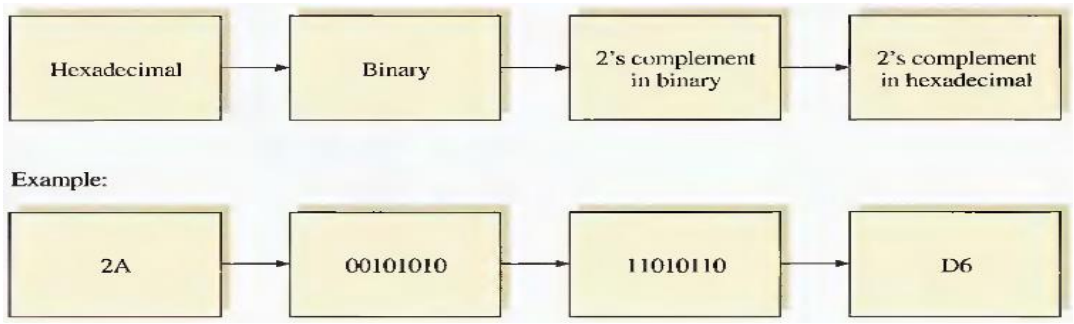


Fig: Obtaining the 2's complement of a hexadecimal number, Method 1.

Method 2: Subtract the hexadecimal number from the maximum hexadecimal number and add 1. This is shown in Figure below.

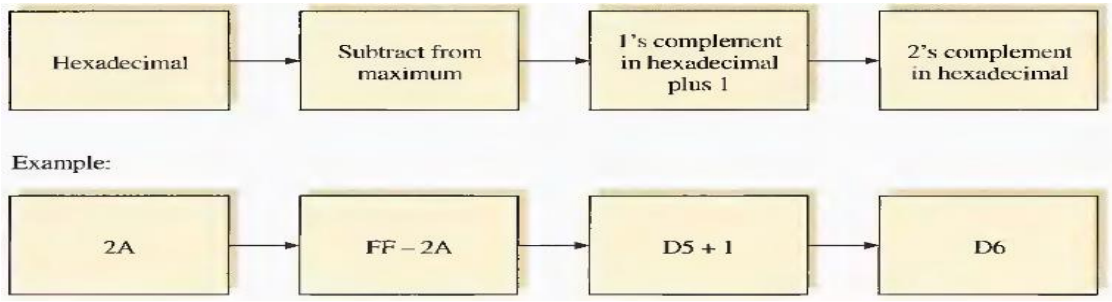


Fig: Obtaining the 2's complement of a hexadecimal number, Method 2.

Method 3: Write the sequence of single hexadecimal digits. Write the sequence in reverse below the forward sequence. The l's complement of each hex digit is the digit directly below it. Add 1 tothe resulting number to get the 2's complement. This is shown in Figure.

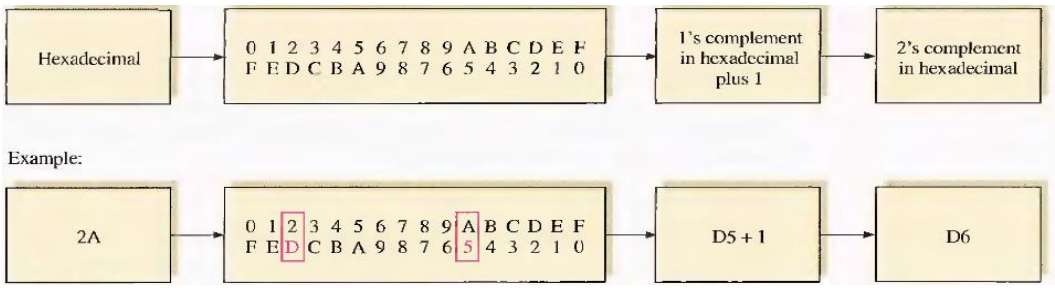


Fig: Obtaining the 2's complement of a hexadecimal number, Method 3.

EXAMPLE

Subtract the following hexadecimal numbers:

(a) $84_{16} - 2A_{16}$ (b) $C3_{16} - 0B_{16}$

Solution (a) $2A_{16} = 00101010$

2's complement of $2A_{16} = 11010110 = D6_{16}$ (using Method 1)

$$\begin{array}{rl} & 84_{16} \\ + & D6_{16} \quad \text{Add} \\ \hline & \not{1}5A_{16} \quad \text{Drop carry, as in 2's complement addition} \end{array}$$

The difference is $\mathbf{5A}_{16}$.

(b) $0B_{16} = 00001011$

2's complement of $0B_{16} = 11110101 = F5_{16}$ (using Method 1)

$$\begin{array}{rl} & C3_{16} \\ + & F5_{16} \quad \text{Add} \\ \hline & \not{1}B8_{16} \quad \text{Drop carry} \end{array}$$

The difference is $\mathbf{B8}_{16}$.

15's and 16's complement Arithmetic:

The 15's complement **of a hexadecimal number can be found by subtracting each digit in the number from 15. The 16's complement can be obtained by subtracting the LSB from 16 and the rest of each digit in the number from 15.**

Subtraction using 15's complement:

Example *Subtract $(2B1)_{16}$ from $(3A2)_{16}$ using the 15's complement method. Also subtract using the direct method and compare.*

Solution.

Direct Subtraction		15's complement method		
3 A 2			3 A 2	(+)
– 2 B 1	15's complement →		D 4 E	
F 1			1 0 F 0	
	Add Carry →		1	
			$(F\ 1)_{16}$ (Result)	

Example *Subtract $(3A2)_{16}$ from $(2B1)_{16}$ using the 15's complement method. Also subtract using the direct method and compare.*

Solution.

Direct Subtraction		15's complement method		
2 B 1			2 B 1	(+)
– 3 A 2	15's complement →		C 5 D	
1 F 0 F			F 0 E	
Discard Carry F 0 F	15's complement →		F 1	
16's complement F 1				
True result $(-F1)_{16}$	True result		$(-F1)_{16}$	

In the direct method, whenever a larger number is subtracted from a smaller number, the result obtained is in 16's complement form and opposite in sign. To get the true result we have to discard the carry and make the 16's complement of the result obtained and put a negative sign before the result.

Subtraction using 16's complement:

Example *Subtract $(1FA)_{16}$ from $(2DC)_{16}$ using the 16's complement method. Also subtract using the direct method and compare.*

Solution.

Direct Subtraction		16's complement method		
2 D C			2 D C	(+)
– 1 F A	16's complement →		E 0 6	
E 2	Carry		1 0 E 2	
	Discard Carry		$(E\ 2)_{16}$ (Result)	

Example *Subtract $(2DC)_{16}$ from $(1FA)_{16}$ using the 16's complement method. Also subtract using the direct method and compare.*

Solution.

Direct Subtraction		16's complement method		
1 F A			1 F A	(+)
–2 D C	16's complement →		D 2 4	
1 0 1 E			F 1 E	
Discard carry 1 E	16's complement →		E 2	
16's complement E 2				
True result $(-E2)_{16}$	True result		$(-E2)_{16}$	

Types of binary codes:

Binary codes are codes which are represented in binary system with modification from the original ones. The classification of binary codes is as follows:

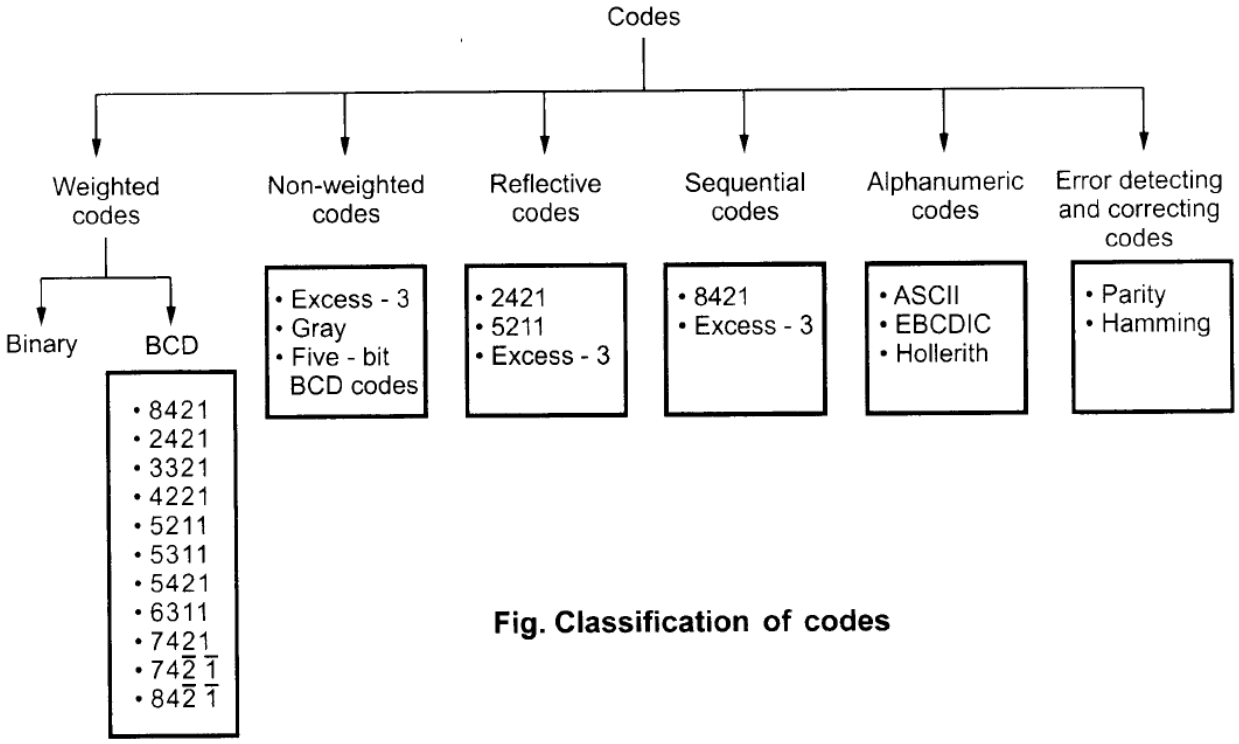


Fig. Classification of codes

S.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 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	4 – Bit BCD code, 4221, 5211, 8421, 6421, 84-2-1.	XS – 3 code and Gray code.
4.	Applications	These codes are used in: (a) Data manipulation during arithmetic operations.	These codes are used in: (a) To perform certain arithmetic operations (i.e., used in K – Map simplification).
		(b) For input/output operations in digital circuits.	(b) Shift position encoders
		(c) To represent the decimal digits in calculators, voltmeters etc.	(c) Used for error detecting purpose.

Weighted binary codes are those which obey the positional weighting principles, each position of the number represents a specific weight. The binary counting sequence is an example.

Decimal	8421	4221	5211	$74\bar{2}\bar{1}$
0	0000	0000	0000	0000
1	0001	0001	0001	0111
2	0010	0010	0011	0110
3	0011	0011	0110	0101
4	0100	1000	0111	0100
5	0101	0111	1000	1010
6	0110	1100	1001	1001
7	0111	1101	1100	1000
8	1000	1110	1110	1111
9	1001	1111	1111	1110

8421 code/BCD code:

The BCD (Binary Coded Decimal) is a straight assignment of the binary equivalent. It is possible to assign weights to the binary bits according to their positions. The weights in the BCD code are 8,4,2,1.

Example: The bit assignment 1001 can be seen by its weights to represent the decimal 9 because

1x8+0x4+0x2+1x1 = 9.

Weights	**8**	**4**	**2**	**1**	**Result**
Binary No.	1	0	0	1	
Equivalent Decimal No.	1 X 8 + 0 X 4 + 0 X 2 + 1 X 1				**9**

2421 code:

This is a weighted code; its weights are 2, 4, 2 and 1. A decimal number is represented in 4-bit form and the total four bits weight is 2 + 4 + 2 + 1 = 9. Hence the 2421 code represents the decimal numbers from 0 to 9.

5211 code:

This is a weighted code; its weights are 5, 2, 1 and 1. A decimal number is represented in 4-bit form and the total four bits weight is 5 + 2 + 1 + 1 = 9. Hence the 5211 code represents the decimal numbers from 0 to 9.

Reflective code / Self – Complementing codes:

A code is said to be reflective code, if the code word of the 9's complement of a number 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 code 0, 8 for 1, 7 for 2, 6 for 3 and 5 for 4 and so on. Codes 4211, 2421, 5211, 642-3 and excess-3 are reflective, whereas the 8421 code is not.

Advantage: Self – complementing codes have an advantage that their logical complement is the same as the arithmetic complement.

Sequential code: In sequential codes,each succeeding code is one binary number greater than its preceding code, i.e., each succeeding code differs by one.
Examples: 8421 and Excess-3 codes, whereas the 2421 and 5211 codes are not sequential codes.
Non-Weighted code:

Non weighted codes are codes that are not positionally weighted. That is, each position within the binary number is not assigned a fixed value.

Excess-3 code:

Excess-3 is a non weighted code used to express decimal numbers. The code derives its name from the fact that each binary code is the corresponding 8421 code plus 0011(3).

Example: 1000 of 8421 = 1011 in Excess-3.

Decimal digit	Xs-3 code
0	0011
1	0100
2	0101
3	0110
4	0111
5	1000
6	1001
7	1010
8	1011
9	1100
10	01000011
11	01000100
12	01000101
13	01000110
14	01000111
15	01001000

Example: Find the XS-3 code and its 9's complement for the following decimal no.s.

(a). $(592)_{10}$

Sol) $(592)_{10}$

BCD of 592:	0101	1001	0010
	0011	0011	0011
XS-3 (592)	1000	1100	0101

9's complement of 592: 0111 0011 1010 (XS-3 code is self-complementary codes. i.e., Reflective codes).

(b). $(403)_{10}$:

BCD of 403:	0100	0000	0011
	0011	0011	0011
XS-3 (403)	0111	0011	0110

9's complement of 403: 1000 1100 1001 (XS-3 code is self-complementary codes. i.e., Reflective codes)

Note: In the above two examples, the 9's complement of a XS – 3 code / number is obtained directly by interchanging 1's to 0's and vice – versa.

XS-3 Addition:

- Add two XS-3 numbers.
  - If carry = 1,add0011(3) to the sum of 2 digits (i.e., append or add 3 zeros to the left of carry and add 0011 (3)). And if any carry occurs by adding 0011 (3), add it to the next higher order bit position.

  = 0 subtract 0011(3).

Example:

1. **8+6**

	8	1011	(XS-3 for 8)
+	6	1001	(XS-3 for 6)
	14	0001 0100	
		0011 0011	
		0100 0111	(XS-3 for 14)
		(1) (4)	

2. **1+2**

	1	0100	(XS-3 for 1)
+	2	0101	(XS-3 for 2)
	3	1001	(No carry, so subtract 3)
		0011	
		0110	(XS-3 for 3)
		(3)	

1. Add 27 & 48 using XS-3 method

Ans:-

27 in XS-3 :	0101	1010
48 in XS-3 :	0111	1011
	1100	10101
	-0011	+0011
XS-3 No:	1010	1000
	7	5.

27 + 48 = 75

2. Add 127 & 248 using XS-3 method.

Ans:-

127 in XS-3 :	0100	0101	1010
248 in XS-3 :	0101	0111	1011
	1001	1101	10101
	-0011	-0011	+0011
XS-3 NO:	0110	1010	1000
	3	7	5

127 + 248 = 375

3. Add 458 & 264 using XS-3 method.

Ans:-

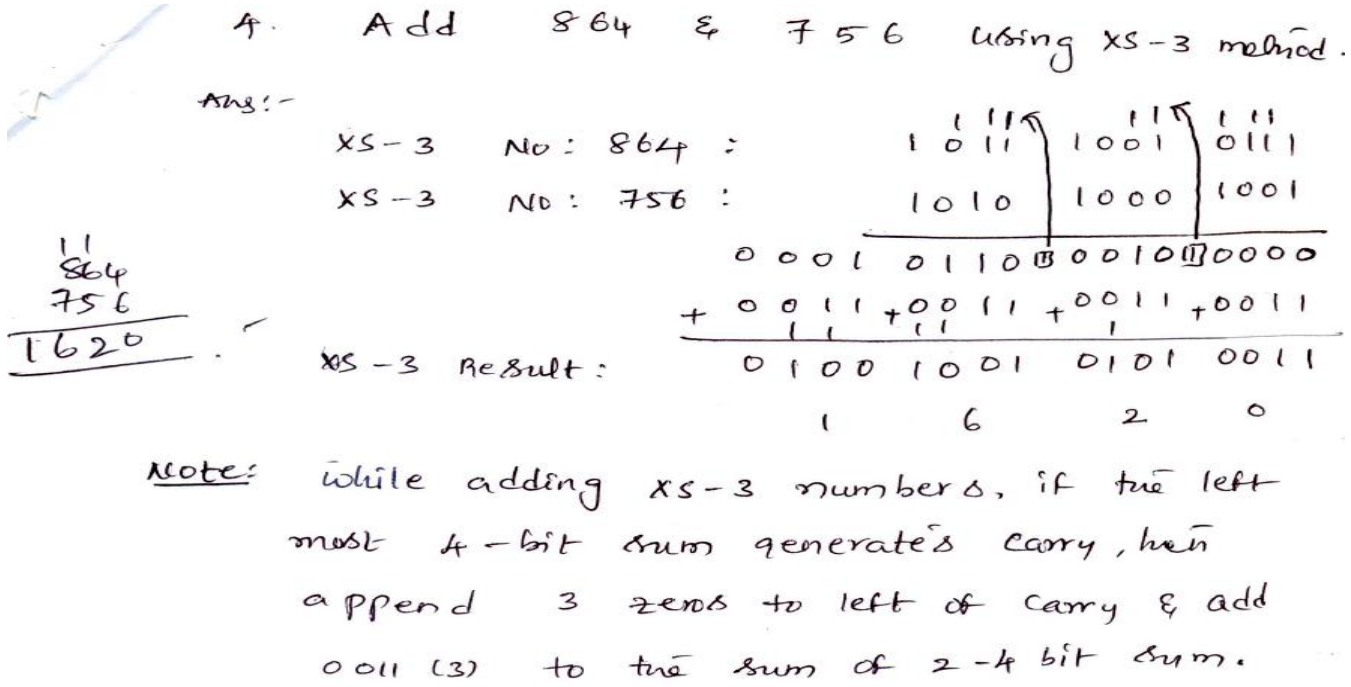
458 in XS-3 :	0111	1000	1011
264 in XS-3 :	0101	1001	0111
	1101	10010	10010
	-0011	+0011	+0011
XS-3 NO:	1010	0101	0101
	7	2	2.

458 + 264 = 722

Note: In addition of XS-3 numbers.

i, If carry is generated, then add 0011(3) to the result of 4-bit sum & add the carry to next higher order position.

ii, If carry is not generated subtract -0011(3)



XS-3 Subtraction

- Complement the subtrahend
- Add the complemented subtrahend to minuend
- If carry = 1, Result is positive. Add 0011(3) and end around carry
  = 0, Result is negative. Subtract 0011(3) and the result is in excess – 3 negative number and to get its true form complement the result and put negative in front.

Example:

1. **8 -5**

XS-3 for 5 : 1000
Complement for 5 in XS-3 :0111

XS – 3 for 8 : 1011
Complement of subtrahend : 0111

10010
\+ 0011

0101
1

0110 (XS-3 for 3)

Note: In excess – 3, 9's complement and complement of any number is same. So this excess – 3 subtraction method is also known as excess – 3 subtraction using 9's complement.

2. 5 -8

XS-3 for 8 : 1011
Complement for 8 in XS-3 :0100

XS – 3 for 5 : 1000
Complement of subtrahend : 0100

1100
\- 0011

1001 (XS-3 for - 3)

Subtraction :-

1. Subtract 28 from 46 using XS-3 method.

Ans:- Direct method

46
−28
18

XS-3 NO: 46 : 0111 1001 (Minuend)
XS-3 NO: 28 : − 0101 1011 (subtrahend)

Step 1: Find the complement of subtrahend ~~i.e~~ (9's complement).

i.e 0101 1011

9's complement : 1010 0100
of subtrahend:

Step 2: Add the complemented subtrahend to minuend, & follow addition rules of XS-3. except, if carry is generated by adding left most 4-bit sum, then add the carry to ~~LSB~~ LSB position. i.e end around carry.

i.e 0111 1001 → Minuend
1010 0100 → complemented Subtrahend.

1 0001 1 + 10 1
0
+0011 −0011
1 1

End Around carry: 0100 1010 → 1

XS-3 Result: 0100 1011
+(1 8)

2. Subtract 46 from 28 using XS-3 method.

Ans: Direct method:

$$\begin{array}{r} 28 \\ -46 \\ \hline -(18) \end{array}$$

XS-3 No 28 : 0101 1000 (Minuend)

XS-3 No 46 : 0111 1001 (Subtrahend)

Step 1: Find the complement of subtrahend, (9's complement).

i.e • XS-3 No: 46 : 0111 1001

Complement of XS-3 No. 46: 1000 0110 (i.e 9's complement of subtrahend).

Step 2:- Add the complemented subtrahend to minuend & follow addition rules of XS-3, i.e. if carry is not generated, then to obtain true result find the complement of result & put a negative sign infront.

i.e., 0101 (Minuend)

1000 (complemented subtrahend).

$$\begin{array}{r} 0101 \\ 1000 \\ \hline 1\,1\,0\,1 \\ +0\,0\,1\,1 \\ \hline 0\,1\,0\,0 \end{array} \qquad \begin{array}{r} 1011 \\ 0110 \\ \hline 0001 \\ +0011 \\ \hline 0100 \end{array}$$

Result: (1011 0100) → XS-3 No −18

To obtain true form, obtain the complement of result & put a −ve sign infront.

i.e 0101 0100

Note:

- BCD and 2421 are weighted codes.
- XS-3 is an un-weighted code.
- 2421 and XS-3 are self complementary codes. i.e., Reflective codes.
- Such codes have the property that 9's complement of a decimal number is obtained directly by changing 1's to 0's & 0's to 1's in the code.

Example:

$(395)_{10}$ is represented in XS-3 as : 0110 1100 1000

The 9's complement of 395 is 604 (0110 1100 1000), which is obtained by complementing each bit of the code.

Gray code:

- The Gray code is non-weighted and is not an arithmetic code; that is, there are no specific weights assigned to the bit positions.
- The important feature of the Gray code is that it exhibits only a single bit change from one code word to the next in sequence. This property is important in many applications, such as shaft position encoders, where error susceptibility increases with the number of bit changes between adjacent numbers in a sequence.
- The gray code is a reflective digital code which has the special property that any two subsequent numbers codes differ by only one bit. This is also called a unit-distance code. In digital Gray code has got a special place.

The Gray Code:

Note: The single bit change characteristic of the Gray code minimizes the chance for error.

Table is a listing of the 4-bit Gray code for decimal numbers 0 through 15. Binary numbers are shown in the table for reference. Like binary numbers, the Gray code can have any number of bits. Notice the single-bit change between successive Gray code words. For instance, in going from decimal 3 to decimal 4, the Gray code changes from 001 0 to 0110, while the binary code changes from 0011 to 01 00, a change of three bits. The only bit change is in the third bit from the right in the Gray code; the others remain the same.

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

Binary-to-Gray Code Conversion:

Conversion between binary code and Gray code is sometimes useful. The following rules explain how to convert from a binary number to a Gray code word:

1. The most significant bit (left-most) in the Gray code is the same as the corresponding MSB in the binary number.
2. Going from left to right, add each adjacent pair of binary code bits to get the next Gray code bit. Discard carries.

For example, the conversion of the binary number 10110 to Gray code is as follows:

The Gray code is 11101.

Gray-to-Binary Conversion:

To convert from Gray code to binary, use a similar method; however, there are some differences. The following rules apply:

1. The most significant bit (left-most) in the binary code is the same as the corresponding bit in the Gray code.
2. Add each binary code bit generated to the Gray code bit in the next adjacent position. Discard carries.

For example, the conversion of the Gray code word 11011 to binary is as follows:

The binary number is 10010.

Application

The gray code is used in applications where the sequence of binary numbers may produce an error during the transition from one number to the next. If the binary numbers are used, a change from 0111 to 1000 may produce an intermediate erroneous number 1001 if the right most bit takes no longer to change in value than the other 3 – Bits.

This above problem can be eliminated by using Gray code.

BINARY CODED DECIMAL (BCD):

Binary coded decimal (BCD) is a way to express each of the decimal digits with a binary code. There are only ten code groups in the BCD system, so it is very easy to convert between decimal and BCD. Because we like to read and write in decimal, the BCD code provides an excellent interface to binary systems. Examples of such interfaces are keypad inputs and digital readouts.

Note: In BCD, 4 bits represent each decimal digit.

The 8421 Code:

The 8421 code is a type of BCD (binary coded decimal) code. Binary coded decimal means that each decimal digit, 0 through 9, is represented by a binary code of four bits. The designation 8421 indicates the binary weights of the four bits (2^3 , 2^2 , 2^1 ,2^0). The ease of conversion between 8421 code numbers and the familiar decimal numbers is the main advantage of this code. All that we have to remember are the ten binary combinations that represent the ten decimal digits as shown in Table. The 8421 code is the predominant BCD code, and when we refer to BCD, we always mean the 8421 code unless otherwise stated.

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

Invalid Codes: With four bits, sixteen numbers (0000 through 1111) can be represented in the 8421 code. Only ten of these are used. The six code combinations that are not used are 1010, 1011, 1100, 1101, 1110, and 1111 are invalid in the 8421 BCD code.

To express any decimal number in BCD, simply replace each decimal digit with the appropriate 4-bit code, as shown below.

EXAMPLE

Convert each of the following decimal numbers to BCD:

(a) 35 (b) 98 (c) 170 (d) 2469

Solution

(a) 3 5 → 00110101

(b) 9 8 → 10011000

(c) 1 7 0 → 000101110000

(d) 2 4 6 9 → 0010010001101001

It is equally easy to determine a decimal number from a BCD number. Start at the right- most bit and break the code into groups of four bits. Then write the decimal digit represented by each 4-bit group.

EXAMPLE

Convert each of the following BCD codes to decimal:

(a) 10000110 (b) 001101010001 (c) 1001010001110000

Solution

(a) 10000110 → 8 6

(b) 001101010001 → 3 5 1

(c) 1001010001110000 → 9 4 7 0

BCD Addition:

BCD is a numerical code and can be used in arithmetic operations. Addition is the most important operation because the other three operations (subtraction, multiplication, and division) can be accomplished by the use of addition.

Steps for addition of two BCD numbers:

1. Add the two BCD numbers, using the rules for binary addition.
2. If a 4-bit sum is equal to or less than 9, it is a valid BCD number.
3. If a 4-bit sum is greater than 9, or if a carry out of the 4-bit group is generated, it is an invalid result. Add 6 (0110) to the 4-bit sum in order to skip the six invalid states and return the code to 8421. If a carry results when 6 is added, simply add the carry to the next 4-bit group.

EXAMPLE

Add the following BCD numbers:

(a) 0011 + 0100 (b) 00100011 + 00010101

(c) 10000110 + 00010011 (d) 010001010000 + 010000010111

Solution The decimal number additions are shown for comparison.

(a)
```
  0011        3
+ 0100      + 4
  0111        7
```

(b)
```
  0010  0011        23
+ 0001  0101      + 15
  0011  1000        38
```

(c)
```
  1000  0110        86
+ 0001  0011      + 13
  1001  1001        99
```

(d)
```
  0100  0101  0000        450
+ 0100  0001  0111      + 417
  1000  0110  0111        867
```

Note that in each case the sum in any 4-bit column does not exceed 9, and the results are valid BCD numbers.

EXAMPLE

Add the following BCD numbers

(a) 1001 + 0100 (b) 1001 + 1001

(c) 00010110 + 00010101 (d) 01100111 + 01010011

Solution The decimal number additions are shown for comparison.

(a)
```
    1001                                  9
  + 0100                                +4
    1101    Invalid BCD number (>9)      13
  + 0110    Add 6
  0001 0011 Valid BCD number
    ↓   ↓
    1   3
```

(b)
```
    1001                                  9
  + 1001                                + 9
  1 0010    Invalid because of carry     18
  + 0110    Add 6
  0001 1000 Valid BCD number
    ↓   ↓
    1   8
```

(c)
```
  0001  0110                             16
+ 0001  0101                           + 15
  0010  1011  Right group is invalid (>9),  31
                left group is valid.
        + 0110  Add 6 to invalid code. Add
                  carry, 0001, to next group.
  0011  0001  Valid BCD number
    ↓   ↓
    3   1
```

(d)
```
        0110  0111                          67
      + 0101  0011                        + 53
        1011  1010  Both groups are invalid (>9)  120
      + 0110  + 0110  Add 6 to both groups
  0001  0010  0000  Valid BCD number
    ↓   ↓     ↓
    1   2     0
```

BCD Subtraction:

There are two methods that can be followed for BCD subtraction.

METHOD 1:

In order to subtract any number from another number we have to add the 9's complement of the subtrahend to the minuend. We can use the 10's complement also to perform the subtraction operation.

The 9's complement of a decimal number can be found by subtracting each digit in the number from 9. The 10's complement can be obtained by subtracting the LSB from 10 and the rest of the each digit in the number from 9.

The 9's and 10's Complement of the decimal digits 0 to 9 is shown in table below.

Decimal Digit	9's Complement	10's Complement
0	9	10
1	8	9
2	7	8
3	6	7
4	5	6
5	4	5
6	3	4
7	2	3
8	1	2
9	0	1

Subtraction Using 9's Complement

Example *Subtract $(358)_{10}$ from $(592)_{10}$ using the 9's complement method. Also subtract using the direct method and compare.*

Solution.

Direct Subtraction		9's complement method	
5 9 2		5 9 2	(+)
– 3 5 8	9's complement →	6 4 1	
2 3 4		1 2 3 3	
	Add Carry →	1	
		$(2\ 3\ 4)_{10}$ (Result)	

Example *Subtract $(592)_{10}$ from $(358)_{10}$ using the 9's complement method. Also subtract using the direct method and compare.*

Solution.

Direct Subtraction		9's complement method	
3 5 8		3 5 8	(+)
– 5 9 2	9's complement →	4 0 7	
– 1 7 6 6		7 6 5	
Discard carry 7 7 6	9's complement →	2 3 4	
10's complement 2 3 4	True result	$(-234)_{10}$	
True result $(-234)_{10}$			

Subtraction Using 10's Complement

Example *Subtract $(438)_{10}$ from $(798)_{10}$ using the 10's complement method. Also subtract using the direct method and compare.*

Solution.

Direct Subtraction		10's complement method		
7 9 8			7 9 8	(+)
– 4 3 8	10's complement	→	5 6 2	
3 6 0	Carry	→	1 3 6 0	
	Discard Carry		$(3\ 6\ 0)_{10}$ (Result)	

Example *Subtract $(798)_{10}$ from $(438)_{10}$ using the 10's complement method. Also subtract using the direct method and compare.*

Solution.

Direct Subtraction		10's complement method		
4 3 8			4 3 8	(+)
– 7 9 8	10's complement	→	2 0 2	
1 6 4 0			6 4 0	
Discard carry 6 4 0	10's complement	→	3 6 0	
10's complement 3 6 0	True result		$(-360)_{10}$	
True result $(-360)_{10}$				

Example *Carry out BCD subtraction for (893) – (478) using 9's complement method.*

Solution.

9's complement of 478 is 999

$$\begin{array}{r} 999 \\ -\ 478 \\ \hline 521 \end{array}$$

Direct method

$$\begin{array}{r} 893 \\ -\ 478 \\ \hline 415 \end{array}$$

Now in BCD form we may write

$$\begin{array}{rlll} & 1000 & 1001 & 0011 \\ +\ & 0101 & 0010 & 0001 \\ \hline & 1101 & 1011 & 0100 \\ +\ & 0110 & 0110 & \\ \hline 1 & 0100 & 0001 & 0100 \\ & & & \rightarrow 1 \\ \hline & 0100 & 0001 & 0101 \end{array}$$

Left and middle groups are invalid

Add 6

End around carry

Hence, the final result is $(0100\ 0001\ 0101)_2$ or $(415)_{10}$.

Example *Carry out BCD subtraction for (768) – (274) using 10's complement method.*

Solution.

10's complement of 274 is 9910

$$\begin{array}{r} 9910 \\ -\ 274 \\ \hline 726 \end{array}$$

Direct method

$$\begin{array}{r} 768 \\ -\ 274 \end{array}$$

Now in BCD form we may write

$$\begin{array}{rrrl} 0111 & 0110 & 1000 & \\ +\ 0111 & 0010 & 0110 & \\ \hline 1110 & 1000 & 1110 & \text{Left and right groups are invalid} \\ +\ 0110 & & 0110 & \text{Add 6} \\ \hline 1\ 0100 & 1001 & 0100 & \end{array}$$

Ignore Carry →

Hence, the final result is $(0100\ 1001\ 0100)_2$ or $(494)_{10}$.

Total result positive

$$\begin{array}{rrrrl} 736 & & 0111 & 0011 & 0110 \\ -273 & & 1101 & 1000 & 1100 \\ +\ 463 & 1 & 0100 & 1011 & 0010 \\ & & \downarrow & 1 \leftarrow & 1 \\ & & 0100 & 1100 & 0011 \\ & & 0000 & 1010 & 0000 \\ & & 0100 & 10110 & 0011 \\ & & 4 & \uparrow 6 & 3 \end{array}$$

← 1's complement of 0010 0111 0011

Ignore this carry 1

Total result negative

$$\begin{array}{rrrrl} 427 & & 0100 & 0010 & 0111 \\ -572 & & 1010 & 1000 & 1101 \\ 145 & & 1110 & 1010 & 0100 \\ & & \downarrow & 1 \leftarrow & \downarrow \\ & & & 1011 & \\ & & 0001 & 0100 & 1011 \\ & & 0000 & 0000 & 1010 \\ & & 0001 & 0100 & 10101 \\ & & 1 & 4 & \uparrow 5 \end{array}$$

← 1's complement of 0101 0111 0010

Transfer 1's complement of adder 1 output

Ignore this carry 1

DIGITAL CODES:

Many specialized codes are used in digital systems. BCD is also one of the digital codes; some codes are strictly numeric, like BCD, and others are alphanumeric; that is, they are used to represent numbers, letters, symbols, and instructions.

Alpha numeric codes:

- For communication, we not only require numbers. But, also other symbols known as non – numeric data.
- The binary codes that can be used to represent all the letters of the alphabet, numbers and mathematical symbols, punctuation marks, are known as alphanumeric codes or character codes. These codes enable us to interface the input-output devices like the keyboard, printers, video displays with the computer.

The most commonly used Alpha numeric codes are:

- ASCII (American standard code for information interchange) &
- EBCDIC (Extended Binary code for Decimal Interchange code)

ASCII codes: (American standard code for information interchanges)

- ASCII is the abbreviation for American Standard Code for Information Interchange. Pronounced "askee," ASCII is a universally accepted alphanumeric code used in most computers and other electronic equipment. Most computer keyboards are standardized with the ASCII. When we enter a letter, a number, or control command, the corresponding ASCII code goes into the computer.
- It is a 7-bit code representing $2^7 = 128$ different characters.
- ASCII has 128 characters and symbols represented by a 7-bit binary code, Actually, ASCII can be considered as an 8-bit code with the MSB always 0.
- This 8-bit code is 00 through 7F in hexadecimal. The first thirty-two ASCII characters are non-graphic commands that are never printed or displayed and are used only for control purposes.
- These codes represent 26 upper case letters (A to Z), 26 lowercase letters (a to z), 10 numbers (0 to 9), 33 special characters and symbols and 33 control characters.
- Examples of the control characters are ""null," "line feed," "start of text," and "escape."
- The other characters are graphic symbols that can be printed or displayed and include the letters of the alphabet (lowercase and uppercase). The ten decimal digits, punctuation signs and other commonly used symbols form the ASCII code.

Note: A computer keyboard has a dedicated microprocessor that constantly scans keyboard circuits to detect when a key has been pressed and released. A unique scan code is produced by
Computer software representing that particular key. The scan code is then converted to an alphanumeric code (ASCII) for use by the computer.

ASCII code: It is a 7 - Bit code in which the decimal digits are represented by the BCD code.

LSB's	MSB's							
	000(0)	**001(1)**	**010(2)**	**011(3)**	**100(4)**	**101(5)**	**110(6)**	**111(7)**
0000 (0)	NUL	DLE	!	0	@	P	.	p
0001 (1)	SOH		"	1	A	Q	a	q
0010 (2)	STX		#	2	B	R	b	r
0011 (3)	ETX		$	3	C	S	c	s
0100 (4)			%	4	D	T	d	t
0101 (5)			&	5	E	U	e	u
0110 (6)			'	6	F	V	f	v
0111 (7)			(	7	G	W	g	w
1000 (8)			)	8	H	X	h	x
1001 (9)			*	9	I	Y	i	y
1010(10)			+	:;	J	Z	j	z
1011(11)				;	K		k	
1100(12)				<	L		l	
1101(13)				=	M		m	
1110(14)					N		n	
1111(15)			/		O		o	

Table: ASCII code.

ASCII Control Characters:

- The first thirty-two codes in the ASCII table above represent the control characters. These are used to allow devices such as a computer and printer to communicate with each other when passing information and data.
- The control characters and the control key function that allows them to be entered directly from an ASCII keyboard by pressing the control key (CTRL) and the corresponding symbol are also shown in the table. A brief description of each control character is also given.

ASCII control characters:

NAME	DECIMAL	HEX	KEY	DESCRIPTION
NUL	0	00	CTRL @	null character
SOH	1	01	CTRL A	start of header
STX	2	02	CTRL B	start of text
ETX	3	03	CTRL C	end of text
EOT	4	04	CTRL D	end of transmission
ENQ	5	05	CTRL E	enquire
ACK	6	06	CTRL F	acknowledge
BEL	7	07	CTRL G	bell
BS	8	08	CTRL H	backspace
HT	9	09	CTRL I	horizontal tab
LF	10	0A	CTRL J	line feed
VT	11	0B	CTRL K	vertical tab
FF	12	0C	CTRL L	form feed (new page)
CR	13	0D	CTRL M	carriage return
SO	14	0E	CTRL N	shift out
SI	15	0F	CTRL O	shift in
DLE	16	10	CTRL P	data link escape
DC1	17	11	CTRL Q	device control 1
DC2	18	12	CTRL R	device control 2
DC3	19	13	CTRL S	device control 3
DC4	20	14	CTRL T	device control 4
NAK	21	15	CTRL U	negative acknowledge
SYN	22	16	CTRL V	synchronize
ETB	23	17	CTRL W	end of transmission block
CAN	24	18	CTRL X	cancel
EM	25	19	CTRL Y	end of medium
SUB	26	1A	CTRL Z	substitute
ESC	27	1B	CTRL [	escape
FS	28	1C	CTRL /	file separator
GS	29	1D	CTRL]	group separator
RS	30	1E	CTRL ^	record separator
US	31	1F	CTRL _	unit separator

EBCDIC code:

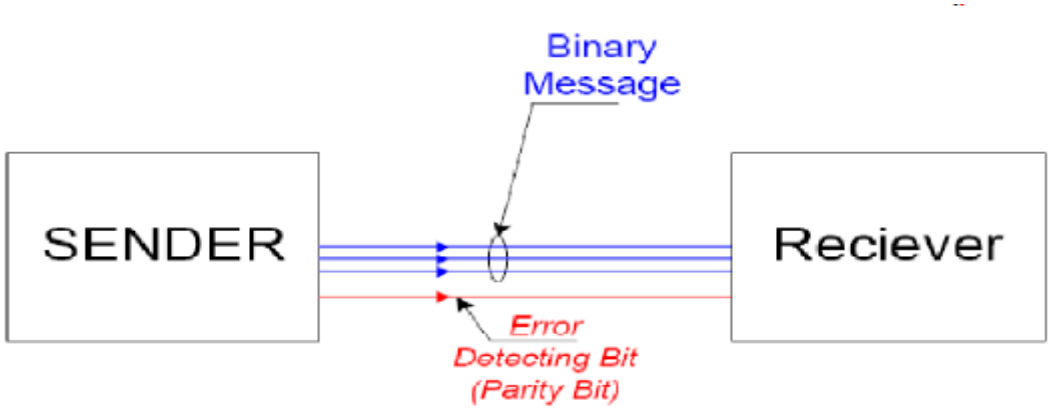
- The extended binary code decimal interchange code is an 8 – bit fixed length character set.
- With 8 – Bits there will be 2^8= 256 codes possible.
- In EBCDIC codes, only 139 out of 256 are used and the remaining codes are assigned to special characters.
- In EBCDIC code the LSB is designated as b_0 and MSB as b_7. Therefore, the higher bit b_7 is transmitted first and lower order bit b_0 is transmitted last.
- It is mainly used with large computer systems like mainframes.

LSB	Most Significant Byte (MSB)															
	0000 (0)	0001 (1)	0010 (2)	0011 (3)	0100 (4)	0101 (5)	0110 (6)	0111 (7)	1000 (8)	1001 (9)	1010 (A)	1011 (B)	1100 (C)	1101 (D)	1110 (E)	1111 (F)
0000 (0)	NUL		DS		SP	&	-									0
0001 (1)			SOS				/		a	j			A	J		1
0010 (2)			FS						b	k	s		B	K	S	2
0011 (3)		TM							c	l	t		C	L	T	3
0100 (4)	PF	RES	BYP	PN					d	m	u		D	M	U	4
0101 (5)	HT	NL	LF	RS					e	n	v		E	N	V	5
0110 (6)	LC	BS	EOB	UC					f	o	w		F	O	W	6
0111 (7)	DL	IL	PRE	EOT					g	p	x		G	P	X	7
1000 (8)		.							h	q	y		H	Q	Y	8
1001 (9)									i	r	z		I	R	Z	9
1010 (A)		CC	SM		¢	!		:								
1011 (B)						$	,	#								
1100 (C)					<	*	%	@								
1101 (D)					(	)	_	'								
1110 (E)					+	;	>	=								
1111 (F)	CU1	CU2	CU3		|		?	"							

Table Partial EBCDIC table

ERROR DETECTION AND CORRECTION CODES:

Binary information may be transmitted through some communication medium, e.g. using wires or wireless media. A corrupted bit will have its value changed from 0 to 1 or vice versa. To be able to detect errors at the receiver end, the sender sends an extra bit (parity bit) with the original binary message.



Parity Method for Error Detection:

A parity bit is an extra bit included with the n-bit binary message to make the total number of 1's in this message (including the parity bit) either odd or even. If the parity bit makes the total number of 1's an odd (even) number, it is called odd (even) parity.

Table below gives:

- Idea about how parity bits are attached to a code;
- Lists the parity bits for each BCD number for both even and odd parity. The parity bit for each BCD number is in the P column.

EVEN PARITY		ODD PARITY	
PARITY BIT	**BCD**	**PARITY BIT**	**BCD**
0	0000	1	0000
1	0001	0	0001
1	0010	0	0010
0	0011	1	0011
1	0100	0	0100
0	0101	1	0101
0	0110	1	0110
1	0111	0	0111
1	1000	0	1000
0	1001	1	1001

The parity bit can be attached to the code at either the beginning or the end, depending on system design.

- The total number of 1's, including the parity bit, is always even for even parity and
- The total number of 1's including the parity bit is always odd for odd parity.

Detecting an Error:

- At the receiver end, an error is detected if the message does not match have the proper parity (odd/even).
- Parity bits can detect the occurrence 1, 3, 5 or any odd number of errors in the transmitted message. At the receiver end, an error is detected if the message does not match have the proper parity (odd/even). No error is detectable if the transmitted message has 2 bits in error since the total number of 1's will remain even (or odd) as in the original message.
- In general, a transmitted message with even number of errors cannot be detected by the parity bit.

Parity bit is used for the detection of a single bit error (or any odd number of errors, which is very unlikely) but cannot check for two errors in one group.

Example:

For example, let's assume that we wish to transmit the BCD code 0101. (Parity can be used with any number of bits; we are using four for explanation) The total code transmitted, including the even parity bit, is

Now let's assume that an error occurs in the third bit from the left (i.e., the 1 becomes a 0).

EXAMPLE

Assign the proper even parity bit to the following code groups:

(a) 1010 (b) 111000 (c) 101101

(d) 1000111001001 (e) 101101011111

Solution Make the parity bit either 1 or 0 as necessary to make the total number of 1s even. The parity bit will be the left-most bit (color).

(a) 01010 (b) 1111000 (c) 0101101

(d) 0100011100101 (e) 1101101011111

EXAMPLE

An odd parity system receives the following code groups: 10110, 11010, 110011, 110101110100, and 1100010101010. Determine which groups, if any, are in error.

Solution Since odd parity is required, any group with an even number of 1s is incorrect. The following groups are in error: **110011** and **1100010101010.**

Error-correcting codes not only detect errors, but also correct them. This is used normally in Satellite communication, where turn-around delay is very high as is the probability of data getting corrupt.

There are two methods for adding bits to codes to either detect a single-bit error or detect and correct a single-bit error.

1. The parity method of error detection.

2. The Hamming method of single-error detection and correction (When a bit in a given code word is found to be in error, it can be corrected by simply inverting it) and double bit error detection.

The Hamming Error Correction Code:

It is one of the most common errors correcting code developed by R.W.Hamming. Hamming code adds a minimum number of bits to the data transmitted in a noisy channel, to be able to correct every possible one-bit error. It can detect (not correct) two-bit errors and cannot distinguish between 1-bit and 2-bits inconsistencies. It can't - in general - detect 3(or more)-bits errors.

- We know that, a single parity bit is appended for detection of single-bit errors in a code word. A single parity bit can indicate that there is an error in a certain group of bits. In order to correct a detected error, more information is required because the position of the bit in error must be identified before it can be corrected. More than one parity bit must be included in a group of bits to be able to correct a detected error.

- In a 7-bit code, there are seven possible single-bit errors. In this case, three parity bits can not only detect an error but can specify the position of the bit in error. The Hamming code is one such code used for detection of error and single-error correction.

- FEC is suited for data communication systems, when acknowledgements are impossible, such as when simplex transmissions are used to transmit messages to many receivers or when the transmission, acknowledgement and retransmission time is excessive.

- However, the addition of FEC bits to each message waste time itself.

Construction of a 7-bit Hamming code for detection of error and single-error correction:

Number of Parity Bits:

If the number of data bits is designated d, then the number of parity bits, p, is determined by the following relationship:

$2^p \geq d+p+1$

For example, if we have 4 data bits, then p is found by trial and error using the above equation. Let p=2. Then

$2^2 \geq 4+2+1$

$4 \geq 7$

Since $2p \geq d+p+1$, the relationship in above equation is not satisfied. We have to try again. Let p=3. Then

$2^3 \geq 4+3+1$

$8 \geq 8$, this value of p satisfies the relationship of equation above, so 3 parity bits are required to provide single error correction for 4 data bits. It should be noted that error detection and correction is provided for all bits, both parity and data, in a code group; i.e., the parity bits also check themselves.

Placement of the Parity Bits in the Code:

The data and parity bits must be arranged properly in the code. In the above example the code is composed of the four data bits and the three parity bits. The left-most bit is designated bit 1, the next bit is bit 2, and so on as follows:

Bit 1, bit 2, bit 3, bit 4, bit 5, bit 6, and bit 7

The parity bits are located in the positions that are numbered corresponding to ascending powers of two (i.e., 1, 2, 4, 8…), as indicated;

$2^0 = 1$	$2^1 = 2$		$2^2 = 4$				$2^3 = 8$		
P_1	P_2	D_3	P_4	D_5	D_6	D_7	P_8	D_9	D_{10}

$P_1, P_2, D_3, P_4, D_5, D_6, D_7$

The symbol P_n designates a particular parity bit, and D_n designates a particular data bit.

Assignment of Parity Bit Values:

- Finally, we must properly assign a 1 or 0 values to each parity bit. Since each parity bit provides a check on certain other bits in the total code, we must know the value of these others in order to assign the parity bit value.

- To find the bit values, first number each bit position in binary, that is, write the binary number for each decimal position number, as shown in the second two rows of Table below. Next, indicate the parity and data bit locations, as shown in the first row of Table below.

- Notice that the binary position number of parity bit P1 has a 1 for its right-most digit. This parity bit checks all bit positions, including itself that has 1's in the same location in the binary position numbers. Therefore, parity bit P1 checks bit positions 1, 3, 5, and 7.

Bit position table for a 7 -bit error correction code:

BIT DESIGNATION	P_1	P_2	D_3	P_4	D_5	D_6	D_7
BIT POSITION	1	2	3	4	5	6	7
BINARY POSITION NUMBER	001	010	011	100	101	110	111
Data Bits (D_n)							
Parity Bits (P_n)							

- The binary position number for parity bit P_2 has a 1 for its middle bit. It checks all bit positions, including it, that have 1s in this same position. Therefore, parity bit P2 checks bit positions 2, 3, 6, and 7.

- The binary position number for parity bit P_3 has a 1 for its left-most bit. It checks all bit Positions, including itself, that have 1s in this same position. Therefore, parity bit P_3 checks bit positions 4, 5, 6, and 7.

- In each case, the parity bit is assigned a value to make the number of 1s in the set of bits It checks either odd or even, depending on which is specified.

THE FOLLOWING EXAMPLES MAKE THIS PROCEDURE CLEAR.

Idea about how parity bits are attached to a code:

EXAMPLE 1: Determine the Hamming code for the BCD number 1001 (data bits), using even parity.

Solution:

Step 1: Find the number of parity bits required. We Use trial and error method. Let p = 3. Then

$2^p \geq d+p+1$
$2^3 \geq 4+3+1$ (True)

Therefore 3 parity bits are sufficient.
Total code bits (Length of Hamming code) = d + p
= 4 + 3
= 7

Step 2: Construct a bit position table, as shown below, and enter the data bits. Parity bits are determined in the following steps.

BIT DESIGNATION	**P_1**	**P_2**	**D_3**	**P_4**	**D_5**	**D_6**	**D_7**
BIT POSITION	**1**	**2**	**3**	**4**	**5**	**6**	**7**
BINARY POSITION NUMBER	**001**	**010**	**011**	**100**	**101**	**110**	**111**
Data Bits (D_n)			1		0	0	1
Parity Bits (P_n)	0	0		1			

Step 3: Determine the parity bits as follows:

- Bit PI checks bit positions 1, 3, 5, and 7 and it must be a 0 to make the total number of 1s (2) even in this group to obtain even parity.
- Bit P 2 checks bit positions 2, 3, 6, and 7 and must be a 0 to make the total number of 1s (2) even in this group to obtain even parity.
- Bit P 3 checks bit positions 4, 5, 6, and 7 and must be a 1 to make the total number of 1s (2) even in this group to obtain even parity.

Step 4: These parity bits are entered in Table, and the resulting combined code is called Hamming Code:0011001.

EXAMPLE 2: Determine the Hamming code for the data bits 10110 using odd parity.

Solution:

Step 1: Determine the number of parity bits required. In this case the number of data bits, d, is five. We Use trial and error method. Let p = 4. Then

$2^p \geq d+p+1$
$2^4 \geq 5+4+1$ (True)
Therefore 4 parity bits are sufficient.
Total code bits = 5 + 4 = 9

Step 2: Construct a bit position table, as shown below, and enter the data bits. Parity bits are determined in the following steps.

BIT DESIGNATION	**P_1**	**P_2**	**D_3**	**P_4**	**D_5**	**D_6**	**D_7**	**P_8**	**D_9**
BIT POSITION	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
BINARY POSITION NUMBER	**0001**	**0010**	**0011**	**0100**	**0101**	**0110**	**0111**	**1000**	**1001**
Data Bits (D_n)			1		0	1	1		0
Parity Bits (P_n)	1	0		1				1	

Step 3: Determine the parity bits as follows:

- Bit P1 checks bit positions 1,3,5, 7 and 9 and it must be a 1 to make the total number of 1s (3) odd in this group to obtain odd parity.
- Bit P 2 checks bit positions 2, 3, 6, and 7 and must be a 0 to make the total number of 1s (3) odd in this group to obtain odd parity.
- Bit P 3 checks bit positions 4, 5, 6, and 7 and must be a 1 to make the total number of 1s (3) odd in this group to obtain odd parity.
- Bit P4 checks bit positions 8 and 9 and must be a 1 to make the total number of 1s (1) odd in this group to obtain odd parity.

Step 4: These parity bits are entered in Table, and the resulting combined code is called
Hamming Code: 101101110.

Detecting and Correcting an Error using the Hamming Code:

The Hamming method for constructing an error-correction code is as follows:

Detection of error bit position and correction:

Each parity bit, along with its corresponding group of bits, must be checked for the proper parity. If there are three parity bits in a code word, then three parity checks are made. If there are four parity bits, four checks must be made, and so on. Each parity check will yield a good or a bad result. The total result of all the parity checks indicates the bit, if any, that is in error, as follows:

Step 1: Start with the group checked by P1

Step 2: Check the group for proper parity. A '0' represents a good parity check, and '1' represents a bad check.

Step 3: Repeat step 2 for each parity group.

Step 4: The binary number formed by the results of all the parity check designates the position of the code bit that is in error. This is the error position code. The first parity check generates the least significant bit (LSB). If all checks are good, there is no error.

EXAMPLE 3: Assume that the code word 0011001 is transmitted and that 001 0001 is received. The receiver does not "know" what was transmitted and must look for proper parities to determine if the code is correct. Designate any error that has occurred in transmission if even parity is used.

Solution: First, make a bit position table, as indicated below

BIT DESIGNATION	**P_1**	**P_2**	**D_3**	**P_4**	**D_5**	**D_6**	**D_7**
BIT POSITION	**1**	**2**	**3**	**4**	**5**	**6**	**7**
BINARY POSITION NUMBER	**001**	**010**	**011**	**100**	**101**	**110**	**111**
Received Code	0	0	1	0	0	0	1

First parity check:
Bit PI checks positions 1, 3, 5, and 7.
There are two 1's in this group.
Parity check is good.----------→ 0 (LSB)

Second parity check:
Bit P 2 checks positions 2, 3, 6, and 7.
There are two 1's in this group.
Parity check is good.----------→ 0

Third parity check:
Bit P 3 checks positions 4, 5, 6, and 7.
There is one 1 in this group.
Parity check is bad.----------→ 1 (MSB)

Result:

The error position code is 100 (binary four). This says that the bit in position 4 is in error. It is a 0 and should be a 1. The corrected code is 001100 1, which agrees with the transmitted code.

EXAMPLE 4: The code 101101010 is received. Correct any errors. There are four parity bits, and odd parity is used.

Solution:

First, make a bit position table as indicated below

BIT DESIGNATION	**P_1**	**P_2**	**D_3**	**P4**	**D5**	**D6**	**D7**	**P8**	**D9**
BIT POSITION	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
BINARY POSITION NUMBER	0001	0010	0011	0100	0101	0110	0111	1000	1001
Received Code	1	0	1	1	0	1	0	1	0

First parity check:
Bit PI checks positions 1, 3, 5, 7, and 9.
There are two 1's in this group.
Parity check is bad.----------→ 1 (LSB)

Second parity check:
Bit P 2 checks positions 2, 3, 6, and 7.
There are two 1's in this group.
Parity check is bad.----------→ 1

Third parity check:
Bit P 3 checks positions 4, 5, 6, and 7.
There are two 1's in this group.
Parity check is bad.----------→ 1

Fourth parity check:
Bit P 4 checks positions 8 and 9.
There is one 1 in this group.
Parity check is good.----------→ 0 (MSB)

Result:
The error position code is 0111 (binary seven). This says that the bit in position 7 is in error. The corrected code is therefore 101101110.

EXAMPLE5 : Determine the Hamming code for the data bits 1011 using even parity.

Solution:

Step 1: Determine the number of parity bits required. In this case the number of data bits, d, is five. We Use trial and error method. Let p = 4. Then

$2^p \geq d + p + 1$
$2^4 \geq 5 + 4 + 1$ (True)

Therefore 4 parity bits are sufficient.
Length of Hamming code (Total code bits) = 5 + 4 = 9

Step 2: Construct a bit position table, as shown below, and enter the data bits. Parity bits are determined in the following steps.

BIT DESIGNATION	**P_1**	**P_2**	**D_3**	**P_4**	**D_5**	**D_6**	**D_7**
BIT POSITION	**1**	**2**	**3**	**4**	**5**	**6**	**7**
BINARY POSITION NUMBER	**001**	**010**	**011**	**100**	**101**	**110**	**111**
Data Bits (D_n)			1		0	1	1
Parity Bits (P_n)	0	1		0			

Step 3: Determine the parity bits as follows:

- Bit P_1 checks bit positions 1,3,5 and 7, and it must be a 1 to make the total number of 1's (4) even in this group to obtain even parity.

  Parity bit P_1 checks : 3 5 7 - 1
  1 1 1 (Number of 1's = 3 odd, so P_1 = 1 to make it even parity)

- Bit P_2 checks bit positions 2, 3, 6, and 7 and must be a 0 to make the total number of 1s (2) even in this group to obtain even parity.

  Parity bit P_2 checks : 3 6 7 - 0
  1 0 1 (Number of 1's = 2 even, so $P_2 = 0$ to make it even parity)

- Bit P_4 checks bit positions 4, 5, 6, and 7 and must be a 0 to make the total number of 1s (2) even in this group to obtain even parity.

  Parity bit P_4 checks : 5 6 7 - 0
  1 0 1 (Number of 1's = 2 even, so $P_4 = 0$ to make it even parity)

Step 4: These parity bits are entered in Table, and the resulting combined code is called **Hamming Code :** 101101110.

BIT DESIGNATION	**P_1**	**P_2**	**D_3**	**P_4**	**D_5**	**D_6**	**D_7**
BIT POSITION	**1**	**2**	**3**	**4**	**5**	**6**	**7**
BINARY POSITION NUMBER	**001**	**010**	**011**	**100**	**101**	**110**	**111**
Data Bits (D_n)			1		0	1	1
Parity Bits (P_n)	0	1		0			
Hamming code	0	1	1	0	0	1	1

Correction of single bit errors if any:

BIT DESIGNATION	**P_1**	**P_2**	**D_3**	**P_4**	**D_5**	**D_6**	**D_7**	**Error position**
BIT POSITION	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**Bit value**
BINARY POSITION NUMBER	**001**	**010**	**011**	**100**	**101**	**110**	**111**	
Received code	0	1	1	0	0	1	1	
For P_1 :	P_1 checks locations 1,3,5,7. There are four 1's in this group ∴parity check for even parity is correct and indicate it with '0'.							**$C_1 = 0$ (LSB)**
For P_2 :	P_2 checks locations 3,6,7. There are two 1's in this group ∴parity check for even parity is correct and indicate it with '0'.							**$C_2 = 0$**
For P_4 :	P_4 checks locations 5,6,7. There are two 1's in this group ∴parity check for even parity is correct and indicate it with '0'.							**$C_4 = 0$ (MSB)**

The resultant bit position that is in error is: **C_4 C_2 C_1 = 0 0 0**

i.e No errors.

Single error correction and Double error detection:

Hamming code explained above provides the detection and correction of only a single error. With a slight modification, it is possible to construct a hamming code for single error correction and double error detection. A one more parity bit is added to the hamming code obtained to make sure that the hamming code (including all parity bits) contains an even number of 1's. The added parity bit is not used in determining the values of the other parity bits. The resulting hamming code makes single error correction and detects double error detection.

Resulting hamming code after adding overall parity bit is as follows:

BIT DESIGNATION	**OP**	**D_7**	**D_6**	**D_5**	**P_4**	**D_3**	**P_2**	**P_1**
Hamming code	0	0	1	1	0	0	1	1

Case 1:

i) OP (Overall parity) --------→ correct
ii) Check for correction -------→ C4 C2 C1

0 0 0

If (i) & (ii) are satisfied, No Error.

Case 2:

i) OP (Overall parity) --------→ In correct, Single Error
ii) Check for correction -------→ C4 C2 C1

1 0 1 (5^{th} bit in error)

If (i) & (ii) are satisfied, Single Error and that can be corrected by detecting error bit position.

Case 3:

i) OP (Overall parity) --------→ correct
ii) Check for correction -------→ C4 C2 C1

0 1 0 (Other than '0')

If (i) & (ii) are satisfied, Double Error and that cannot be corrected.

Example : 6 Finding no. of bits required for hamming code:

- $2k \geq n+k+1$
- $2k \geq 12+k+1$
- $2k \geq 13+k$
- for k=5 the equation holds true. so no. of parity bits required=5

Hamming code:

Bit designation	P_1	P_2	D_3	P_4	D_5	D_6	D_7	P_8	D_9	D_{10}	D_{11}	D_{12}	D_{13}	D_{14}	D_{15}	P_{16}	D_{17}
Bit loc No.	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Binary loc	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111	10000	10001
DataBits			1		0	1	1		0	0	0	1	0	0	1		0
ParityBits	1	0		0				0								0	

Rules for finding the parity bits:

- P_1 => (3,5,7,9,11,13,15,17)
  (1,0,1,0, 0, 0, 1, 0) =1
- P_2 => (3,6,7,10,11,14,15)
  (1,1,1, 0, 0, 0, 1) =0
- P_4 => (5,6,7,12,13,14,15)
  (0,1,1, 1, 0, 0, 1) =0
- P_8 => (9,10,11,12,13,14,15)
  (0, 0, 0, 1, 0, 0, 1) =0
- P_{16} => (17)
  (0) =0

Error detection:

- **Hamming code** **:** 1 0 1 0 0 1 1 0 0 0 0 1 0 0 1 0 0.
- **Error in bit 6** **:** 1 0 1 0 0 0 1 0 00 0 1 0 0 1 0 0.

- C_1 (P1,3,5,7,9,11,13,15,17)
  (1, 1,0,1,0, 0, 0, 1, 0) =0
- C_2 (P2,3,6,7,10,11,14,15)
  (0, 1,0,1, 0, 0, 0, 1) =1
- C_4 (P4,5,6,7,12,13,14,15)
  (0, 0,0,1, 1, 0, 0, 1) =1
- C_8 (P8,9,10,11,12,13,14,15)
  (0, 0, 0, 0, 1, 0, 0, 1) =0
- C_{16} (P16,17)
  (0, 0) =0

Correction code	C16	C8	C4	C2	C1	Decimal equivalent of correction code
	0	0	1	1	0	6

Example 7: Determine which Bit, if any, is in error in the even parity. Hamming coded character is 1100111. Decode the message.

BIT DESIGNATION	P_1	P_2	D_3	P_4	D_5	D_6	D_7	Bit Value
BIT POSITION	1	2	3	4	5	6	7	Error Position
BINARY POSITION NUMBER	001	010	011	100	101	110	111	
Received Data	1	1	0	0	1	1	1	
P_1 checks Bit Positions 1,3,5, & 7 for even Parity check	1		0		1		1	C_1 = 1 (LSB)
P_2 checks Bit Positions 2,3,6, & 7 for even Parity check		1	0			1	1	C_2 = 1
P_4 checks Bit Positions 4,5,6 & 7 for even Parity check				0	1	1	1	C_4 = 1 (MSB)

The resultant bit position that is in error is: **C_4 C_2 C_1 = 1 1 1**

This means that the bit in position 7 is in error. So, the correction is done by simply changing 1 to 0.

Hence the corrected code: 1 1 0 0 1 1 0.

BOOLEAN ALGEBRA AND SWITCHING FUNCTIONS

OUTLINE

- Fundamental postulates of Boolean Algebra,
- Basic theorems and properties,
- Switching functions,
- Canonical and Standard forms,
- Algebraic simplification Digital logic gates,
- Properties of XOR gates,
- Universal gates,
- Multilevel NAND/NOR realizations.

Why Boolean algebra?

- It is highly desirable to find the simplest circuit implementation (Logic) with less number of gates or wires.
- We can use the Boolean minimization process to reduce the function (expression) to its simplest form.
- The result is an expression with the less number of literals and requires less number of wires for the final gate implementation. If number of terms in the expression is reduced, then the number of gates reduces.
- So, simplification plays an important role in digital designs, which reduces the hardware required, i.e. the number of gates and number of wires required. And it also reduces cost, power required and increases the speed of the digital system.

Boolean algebra:

Basic mathematics for the study of logic design is **Boolean algebra.**

Boolean algebra is a mathematical system for the manipulation of variables that can have one of two values.

- In formal logic, these values are “true” and “false.”
- In digital systems, these values are “on” and “off,” 1 and 0, or “high” and “low.”
- Boolean expressions are created by performing operations on Boolean variables.
- Common Boolean operators include AND, OR, and NOT.
- Networks of Logic gates allow us to manipulate digital signals
    - Can perform numerical operations on digital signals such as addition, multiplication
    - Can perform translations from one binary code to another.

(OR)

Boolean algebra may be defined as a set of elements, a set of operators, and a number of unproved axioms or postulates.

Common Postulates in Algebraic Structure

- Closure
- Associative Law
- Commutative Law
- Identity Element
- Inverse
- Distributive Law

AXIOMATIC DEFINITION OF BOOLEAN ALGEBRA

- In 1854, George Boole developed an algebraic system to deal with '1' and '0' now called Boolean algebra.
- In 1904, E. V. Huntington developed some postulates required to deal with the symbols '1' or '0'called Boolean algebra.
- In 1938, Claude E. Shannon introduced a two-valued Boolean algebra called switching algebra that represented the properties of bi - stable electrical switching circuits.

Definition: Boolean algebra is an algebraic structure defined by a set of elements, B, together with two binary operators, + and . , provided that the following (Huntington) postulates are satisfied:

Postulates:

1. (a) The structure is **closed** with respect to the operator +.
(b) The structure is **closed** with respect to the operator . .

2. (a) The element 0 is an **identity** element with respect to +; that is, x + 0 = 0 + x = x.
(b) The element 1 is an **identity** element with respect to . ; that is, x . 1 = 1 .x = x.

3. (a) The structure is **commutative** with respect to +; that is, x + y = y + x.
(b) The structure is **commutative** with respect to . ; that is, x . y = y .x.

4. (a) The operator. is**distributive** over +; that is, x . (y + z) = (x .y) + (x . z).
(b) The operator + is **distributive** over . ; that is, x + (y . z) = (x + y) . (x + z).

5. For every element x ϵ B, there exists an element x^1 ϵB (called the **complement** of x) such that
(a) x+ x^1 = 1 and
(b) x .x^1 = 0.

6. There exist at least two elements x, y ϵ Bsuch that x not equal to y.

Comparing Boolean algebra with arithmetic and ordinary algebra (the field of real numbers), we note the following differences:

1. The distributive law of + over. (i.e., x + (y.z) = (x + y). (x + z)) Is valid in Boolean algebra, but not for ordinary algebra.
2. Boolean algebra does not have additive or multiplicative inverses; therefore, there are no subtraction or division operations.
3. The **complement** operator is not available in ordinary algebra.
4. Ordinary algebra deals with the real numbers, which constitute an infinite set of elements. But two-valued Boolean algebra deals with finite set of elements B ('0' and '1'). B = {0,1}.

Two valued Boolean algebra:

A two-valued Boolean algebra is defined on a set of two elements, B = {0, 1}, with rules for the two binary operators + and .as shown in the following operator tables:

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

x	y	$x + y$
0	0	0
0	1	1
1	0	1
1	1	1

x	x'
0	1
1	0

These rules are exactly the same as the AND, OR, and NOT operations, respectively.

1. That the structure is closed with respect to the two operators '.' or '+' , i.e. the result is either '0' or '1' and (1,0) ϵ B.
2. **Identity element:**

(a) '+'
X + 0 = 0 + x
1 + 0 = 0 + 1 = 1;
0 + 0 = 0 + 0 = 0;

(b) '.'
X . 1 = 1 .x
1 . 1 = 1 . 1 = 1
0 . 1 = 1 . 0 =0;

This establishes the two identity elements, 0 for + and 1 for .

3. The commutative laws are clear from the symmetry of the binary operator tables.

(a) X + Y = Y + X

X	Y	X + Y
0	0	0
0	1	1
1	0	1
1	1	1

X	Y	Y + X
0	0	0
0	1	1
1	0	1
1	1	1

(b) X . Y = Y . X

X	Y	X. Y
0	0	0
0	1	0
1	0	0
1	1	1

X	Y	Y. X
0	0	0
0	1	0
1	0	0
1	1	1

4. (a) The distributive law
x . (y + z) = (x .y) + (x .z):

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

(b) The distributive law of + over .is similar to the one shown above.

5. From the complement table, it is easily shown that
(a) $x + x^1 = 1$, since $0 + 0^1 = 0 + 1 = 1$ and $1 + 1^1 = 1 + 0 = 1$.
(b) $x . x^1 = 0$, since $0 . 0^1 = 0 . 1 = 0$ and $1 . 1^1 = 1 . 0 = 0$.

6. Postulate 6 is satisfied because the two-valued Boolean algebra has two elements, '1' and '0', with '1' not equal to '0'.

DUALITY:

Definition of duality:A dual of a Boolean expression is derived by replacing AND operations by ORs, OR operations by ANDs, constant 0's by 1's, and 1's by 0's (everything else is left unchanged).

Steps are to be followed to obtain the Duality of a Boolean expression.

- Changing OR to AND sign
- Changing AND to OR sign
- Complementing any '0' by '1', '1' by a '0' in the expression.

Principle of duality: If a statement is true for an expression, then it is also true for the dual of the expression.

Basic Theorems and Properties of Boolean algebra

Theorem 1:

(a) x + x = x;
(b) x . x = x;

THEOREM 1(a): $x + x = x.$

Statement	Justification
$x + x = (x + x) \cdot 1$	postulate 2(b)
$= (x + x)(x + x')$	5(a)
$= x + xx'$	4(b)
$= x + 0$	5(b)
$= x$	2(a)

THEOREM 1(b): $x \cdot x = x.$

Statement	Justification
$x \cdot x = xx + 0$	postulate 2(a)
$= xx + xx'$	5(b)
$= x(x + x')$	4(a)
$= x \cdot 1$	5(a)
$= x$	2(b)

Theorem 2:

(a) x + 1 = 1;
(b) x . 0 = 0;

THEOREM 2(a): $x + 1 = 1.$

Statement	Justification
$x + 1 = 1 \cdot (x + 1)$	postulate 2(b)
$= (x + x')(x + 1)$	5(a)
$= x + x' \cdot 1$	4(b)
$= x + x'$	2(b)
$= 1$	5(a)

THEOREM 2(b): x . 0 = 0 by duality.

Theorem 3:

(a) Involution: $(x^1)^1 = x$

X	X^1	$(X^1)^1$
0	1	0
1	0	1

Theorem 4: Associative

(a) x + (y + z) = (x + y) + z;
(b) x . (y . z) = (x . y) . z;

Theorem 5: Demorgans Theorem

(a) $(x + y)^1 = x^1 . y^1$;
(b) $(x . y)^1 = x^1 + y^1$;

(a)

x	y	$x + y$	$(x + y)'$
0	0	0	1
0	1	1	0
1	0	1	0
1	1	1	0

x'	y'	$x'y'$
1	1	1
1	0	0
0	1	0
0	0	0

(b)

A	B	$(AB)^1$	$A^1 + B^1$
0	0	1	1
0	1	1	1
1	0	1	1
1	1	0	0

Complement of a function:

The complement of a function F is F^1& is obtained by:

- Interchanging 0's for 1's & 1's for 0's in the value of F.
- The complement of a function may be derieved algebraically through Demorgans theorem.

Theorem 6: Absorption

(a) x + xy = x;
(b) x . (x + y) = x;

THEOREM 6(a): $x + xy = x$.

Statement	Justification
$x + xy = x \cdot 1 + xy$	postulate 2(b)
$= x(1 + y)$	4(a)
$= x(y + 1)$	3(a)
$= x \cdot 1$	2(a)
$= x$	2(b)

THEOREM 6(b): x(x + y) = x by duality.

NOTE: The postulates are basic axioms of algebraic structure and need no proof. The theorems are proved by using postulates.

Consensus therorem:

(i) $$AB + \overline{A}C + BC = AB + \overline{A}C$$

Proof:

$$\begin{aligned} AB + \overline{A}C + BC &= AB + \overline{A}C + BC \cdot 1 \\ &= AB + \overline{A}C + BC(A + \overline{A}) \qquad (\because A + \overline{A} = 1) \\ &= AB + \overline{A}C + ABC + \overline{A}BC \\ &= AB(1 + C) + \overline{A}C(1 + B) \qquad (\because 1 + B = 1 = 1 + C) \\ &= AB + \overline{A}C \end{aligned}$$

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

Proof:

$$\begin{aligned} (A + B)(\overline{A} + C)(B + C) &= (A + B)(\overline{A} + C)(B + C + 0) \\ &= (A + B)(\overline{A} + C)(B + C + A\overline{A}) \\ &= (A + B)(\overline{A} + C)(B + C + A)(B + C + \overline{A}) \\ &\quad [\because A + BC = (A + B)(A + C)] \\ &= (A + B)(A + B + C)(\overline{A} + C)(\overline{A} + C + B) \\ &= (A + B)(\overline{A} + C) \\ &\quad [\because A(A + B) = A] \end{aligned}$$

Recognition of consensus term:

Step1: Find pair of terms, one of which contains a variable & other contains its complement.

Step 2: Find a 3rd term which should contain the remaining variables from pair of terms eliminating selected variables and its complement.

Operator Precedence:

The operator precedence for evaluating Boolean expressions is:

1. Parentheses,
2. NOT,
3. AND, and
4. OR.

BOOLEAN FUNCTIONS:

- A Boolean function can be represented in a truth table. The number of rows in the truth table is 2^n, where n is the number of variables in the function. The binary combinations for the truth table are obtained from the binary numbers by counting from 0 through 2^n- 1.
- Table below shows the truth table for the function F_1. There are eight possible binary combinations for assigning bits to the three variables x, y, and z. The column labeled F_1 contains either 0 or 1 for each of these combinations. The table shows that the function is equal to 1 when x = 1 or when yz= 01 and is equal to 0 otherwise.

Truth Tables for F_1 and F_2

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

- A Boolean function can be transformed from an algebraic expression into a circuit diagram composed of logic gates connected in a particular structure.
- The logic-circuit diagram (also called a schematic) for F1 is shown in Fig. below. There is an inverter for input y to generate its complement. There is an AND gate for the term y^1z and an OR gate.

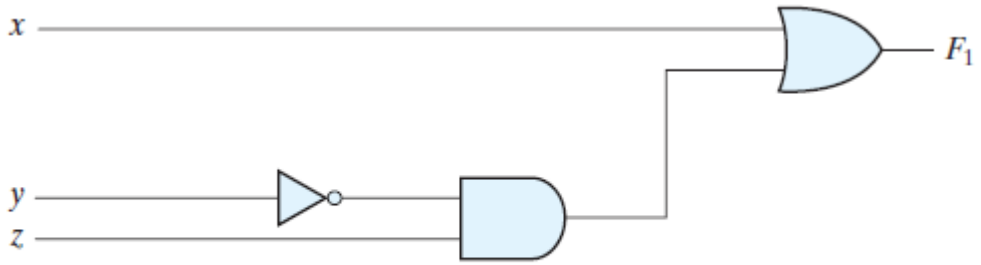


FIGURE
Gate implementation of $F_1 = x + y'z$

OTHER LOGIC OPERATIONS:

For n variables, there are $\mathbf{2^{2^n}}$ **functions**. Thus, for two variables, n = 2, the number of possible Boolean functions is $2^{2X2} = 16$.

Truth table for the 16 functions of 2 – variables is shown below.

Truth Tables for the 16 Functions of Two Binary Variables

x	y	F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}	F_{11}	F_{12}	F_{13}	F_{14}	F_{15}
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

Boolean Expressions for the 16 Functions of Two Variables

Boolean Functions	Operator Symbol	Name	Comments
$F_0 = 0$		Null	Binary constant 0
$F_1 = xy$	$x \cdot y$	AND	x and y
$F_2 = xy'$	x/y	Inhibition	x, but not y
$F_3 = x$		Transfer	x
$F_4 = x'y$	y/x	Inhibition	y, but not x
$F_5 = y$		Transfer	y
$F_6 = xy' + x'y$	$x \oplus y$	Exclusive-OR	x or y, but not both
$F_7 = x + y$	$x + y$	OR	x or y
$F_8 = (x + y)'$	$x \downarrow y$	NOR	Not-OR
$F_9 = xy + x'y'$	$(x \oplus y)'$	Equivalence	x equals y
$F_{10} = y'$	y'	Complement	Not y
$F_{11} = x + y'$	$x \subset y$	Implication	If y, then x
$F_{12} = x'$	x'	Complement	Not x
$F_{13} = x' + y$	$x \supset y$	Implication	If x, then y
$F_{14} = (xy)'$	$x \uparrow y$	NAND	Not-AND
$F_{15} = 1$		Identity	Binary constant 1

Example: find the dual of the following equalities

1) XY+Z = 0
Sol) X+YZ = 1

2) a (b + c) = a b + ac
Sol) a + (b c) = (a + b) (a + c)

Duality is very important property of Boolean Algebra

Minimization (Simplification) of Boolean Expressions

***Example* 1.** Prove that $AB + BC + \overline{B}C = AB + C$,

Solution
$$\begin{aligned} AB + BC + \overline{B}C &= AB + C(B + \overline{B}) \\ &= AB + C \cdot 1 \\ &= AB + C \end{aligned}$$

2.
$$\begin{aligned} \overline{A} \cdot B + A \cdot B + \overline{A} \cdot \overline{B} &= (\overline{A} + A) \cdot B + \overline{A} \cdot \overline{B} \\ &= 1 \cdot B + \overline{A} \cdot \overline{B} \\ &= B + \overline{A} \cdot \overline{B} \\ &= B + \overline{A} \quad (\because \quad A + \overline{A} \cdot B = A + B) \end{aligned}$$

***Example* 3.** Simplify the given expression $A + A \cdot \overline{B} + \overline{A} \cdot B$.

Solution
$$\begin{aligned} A + A \cdot \overline{B} + \overline{A} \cdot B &= A(1 + \overline{B}) + \overline{A} \cdot B \\ &= A \cdot 1 + \overline{A} \cdot B \\ &= A + B \end{aligned}$$

***Example* 4.** Complement the expression $\overline{A}B + C\overline{D}$.

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

***Example* 5.** Simplify the expression $AB + \overline{AC} + A\overline{B}C(AB + C)$.

Solution
$$\begin{aligned} AB + \overline{AC} + A\overline{B}C(AB + C) &= AB + \overline{AC} + A\overline{B}C \cdot AB + A\overline{B}C \cdot C \\ &= AB + \overline{AC} + A\overline{B}C \quad [\because C \cdot C = C \text{ and } \overline{B} \cdot B = 0] \\ &= AB + \overline{A} + \overline{C} + A\overline{B}C \\ &= AB + \overline{A} + \overline{C} + \overline{B}C \quad [\because \quad A + \overline{A}B = A + B] \\ &= \overline{A} + AB + \overline{C} + C\overline{B} \\ &= \overline{A} + B + \overline{C} + \overline{B} \\ &= \overline{A} + \overline{C} + 1 \quad [\because \quad B + \overline{B} = 1] \\ &= \overline{A} + 1 \\ &= 1 \end{aligned}$$

Example:

1. Expand the given Boolean function using shannon's expansion theorem.
   F (A, B, C, D) = A B^1 + (AC + B) D

   Sol) F (A, B, C, D) = A B^1 + (AC + B) D
   = A[1. B^1 + (1.C + B) D] + A^1 [0.B^1 + (0 . C + B) D]
   = A[B^1 + (C + B) D] + A^1 [B D]
   = AB^1 + A (C + B) D + A^1 B D

2. Expand the given Boolean function using shannon's expansion theorem.
   F (A, B, C, D) = A ^{1} C + (B + AD) C

   Sol) F (A, B, C, D) = A ^{1} C + (B + AD) C
   = A [1^{1} . C + (B + 1.D) C] + A ^{1} [0^{1} . C + (B + 0.D) C]

   = A [0 . C + (B + D) C] + A ^{1} [1. C + (B + 0.D) C]

   = A (B+D) C + A ^{1} (C + BC)

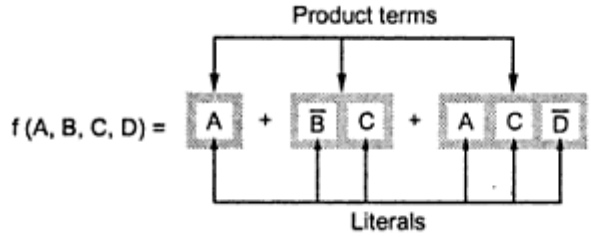
Switching functions / Boolean function

- Boolean algebra is an algebra that deals with binary variables and logic operations.
- A Boolean function described by the algebraic expression consists of binary variables, the constants '0'&'1', and logic operation symbols.

For a given value of binary variables, the function is equal to 1 or 0.

Examples:

Four variable Boolean functions is shown below:



... (1.1)

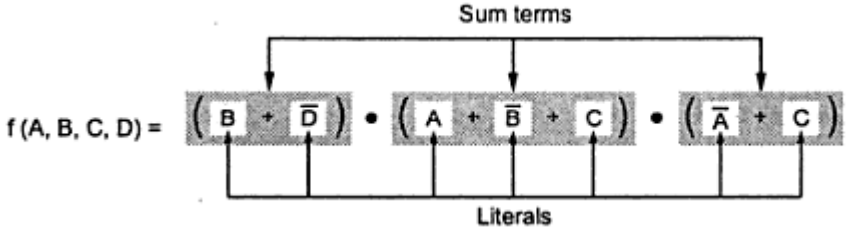
In this boolean function, the variables are appeared either in a complemented form or an uncomplemented form called **Literal**.

Literal: Literal is a binary variable that can occur either in complemented or un complemented form.

A product term is defined as either a literal or a product of literals.

The above function contains 6 literals and 3 product terms.

Example:



... (1.2)

Sum term: It is defined as either a literal or a sum of literal.

The Boolean function can be expressed in any one of the two forms, depending on arrangement of literals and terms.

Boolean function is of two types:

1. SOP (sum of products) &
2. POS (product of sums).

Sum of products

The sum of products is a Boolean function containing AND terms (product terms) of one or more literals each.

Products of sums

The products of sum is a Boolean function containing OR terms (sum terms) of one or more literals each.

Eg: (A'+C) (A'+C') (A+B+C'D)

Algebraic forms of switching functions

- **Sum of products form (SOP)**

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

- **Product of sums form (POS)**

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

Logic representations:

(a) truth table

Row	X	Y	Z	F
0	0	0	0	1
1	0	0	1	0
2	0	1	0	0
3	0	1	1	1
4	1	0	0	1
5	1	0	1	0
6	1	1	0	1
7	1	1	1	1

(b) boolean equation

from 1-rows in truth table:

F = X'Y'Z' + X'YZ + XY'Z' + XYZ' + XYZ

F = Y'Z' + XY + YZ

from 0-rows in truth table:

F = (X + Y + Z')(X + Y' + Z)(X' + Y + Z')

F = (X + Y' + Z)(Y + Z')

Definitions:

Literal : A variable or complemented variable (e.g., X or X')

Product term : Single literal or logical product of literals (e.g., X or X'Y)

Sum term **:** Single literal or logical sum of literals (e.g. X' or (X' + Y))

Sum-of-products **:** Logical sum of product terms (e.g. X'Y + Y'Z)

Product-of-sums : Logical product of sum terms (e.g. (X + Y')(Y + Z))

Normal term : Sum term or product term in which no variable appears more than once

(e.g. X'YZ but not X'YZX or X'YZX' (X + Y + Z') but not (X + Y + Z' + X))

Minterm : Normal product term containing all variables (e.g. XYZ')

Maxterm : Normal sum term containing all variables (e.g. (X + Y + Z'))

Canonical sum : Sum of minterms from truth table rows producing a '1'.

Canonical product : Product of maxterms from truth table rows producing a '0'.

Product and Sum Terms –Definitions:

Definitions:

Literal: A Boolean variable or its complement
X X' A B'

Product term:

A literal or the logical product (AND) of multiple literals:
X XY XYZ X'YZ' A'BC

Note: X(YZ)'

Sum term:
A literal or the logical sum (OR) of multiple literals:
X X'+Y X+Y+Z X'+Y+Z' A'+B+C

Note: X+(Y+Z)'

SOP & POS – Definitions:

Sum of products (SOP) expression: The logic sum (OR) of multiple product terms:
AB + A'C + B' + ABC
AB'C + B'D' + A'CD'

Product of sums (POS) expression: The logic product (AND) of multiple sum terms:
(A+B).(A'+C).B'.(A+B+C)
(A' + B + C).(C' + D)

Note:
SOP expressions ==> 2-level AND-OR circuit
POS expressions ==> 2-level OR-AND circuit

Minterms and Maxterms

Minterm:

Each individual term in a standard SOP form is called minterm.

For n variable there are 2^n minterms.

Minterm	Designation	Bit Combination
X'Y'Z'	m0	000
X'Y'Z	m1	001
X'YZ'	m2	010
X'YZ	m3	011
XY'Z'	m4	100
XY'Z	m5	101
XYZ'	m6	110
XYZ	m7	111

Example: if X, Y and Z are the input variables, the minterms are:
X'Y'Z' X'Y'Z X'YZ' X'YZ XY'Z' XY'Z XYZ' XYZ

Standard or canonical SOP:

In the SOP form, all the individual minterms do not contain all the literals. If each term in a SOP form contains all the literals then the SOP form is known as "standard or canonical SOP" form.

Maxterm:

Each individual term in a standard POS form is called Maxterm.

For n variable there are 2^n max-terms.

Maxterm	Bit Combination	Designmation
X + Y + Z	000	M0
X + Y + Z'	001	M1
X + Y' + Z	010	M2
X + Y' + Z'	011	M3
X' + Y + Z	100	M4
X' + Y + Z'	101	M5
X' + Y' + Z	110	M6
X' + Y' + Z'	111	M7

Example: if X, Y and Z are the input variables, the maxterms are:

X'+Y'+Z' X'+Y'+Z X'+Y+Z' X'+Y+Z X+Y'+Z' X+Y'+Z X+Y+Z' X+Y+Z

Standard or canonical POS :

It each term in a POS form contains all the literals then the POS form is known as Standard or canonical POS'.

Deriving Boolean Expression from Truth Table:

Input			Output	Minterm	
A	B	C	F	term	designation
0	0	0	1	A'B'C'	m0
0	0	1	0	A'B'C	m1
0	1	0	0	A'BC'	m2
0	1	1	1	A'BC	m3
1	0	0	0	AB'C'	m4
1	0	1	0	AB'C	m5
1	1	0	0	ABC'	m6
1	1	1	0	ABC	m7

Sum of minterms form:

A Boolean function is equal to the sum of minterms for which the output is one.

The sum of minterms (also called the standard SOP) form

Example: F = Σ m (0, 3)

Deriving Boolean Expression from Truth Table:

Input			Output	Minterm		Maxterm	
A	B	C	F	term	Designation	term	Designation
0	0	0	1	A'B'C'	m0	A + B + C	M0
0	0	1	0	A'B'C	m1	A + B + C'	M1
0	1	0	0	A'BC'	m2	A + B' + C	M2
0	1	1	1	A'BC	m3	A + B' + C'	M3
1	0	0	0	AB'C'	m4	A' + B + C	M4
1	0	1	0	AB'C	m5	A ' + B + C'	M5
1	1	0	0	ABC'	m6	A' + B' + C	M6
1	1	1	0	ABC	m7	A' + B' + C'	M7

Product of Maxterms:

A Boolean function is equal to the product of Maxterms for which the output is 0.

The product of Maxterms (also called the standard Product of Sums) form

Example: F = ΠM(1,2,4,5,6,7)

Truth table vs. minterms&maxterms:

Row	X	Y	Z	F	Minterm	Maxterm
0	0	0	0	F(0,0,0)	$X' \cdot Y' \cdot Z'$	$X + Y + Z$
1	0	0	1	F(0,0,1)	$X' \cdot Y' \cdot Z$	$X + Y + Z'$
2	0	1	0	F(0,1,0)	$X' \cdot Y \cdot Z'$	$X + Y' + Z$
3	0	1	1	F(0,1,1)	$X' \cdot Y \cdot Z$	$X + Y' + Z'$
4	1	0	0	F(1,0,0)	$X \cdot Y' \cdot Z'$	$X' + Y + Z$
5	1	0	1	F(1,0,1)	$X \cdot Y' \cdot Z$	$X' + Y + Z'$
6	1	1	0	F(1,1,0)	$X \cdot Y \cdot Z'$	$X' + Y' + Z$
7	1	1	1	F(1,1,1)	$X \cdot Y \cdot Z$	$X' + Y' + Z'$

Shortcut notation:

Row	X	Y	Z	F
0	0	0	0	1
1	0	0	1	0
2	0	1	0	0
3	0	1	1	1
4	1	0	0	1
5	1	0	1	0
6	1	1	0	1
7	1	1	1	1

$$F = X'Y'Z' + X'YZ + XY'Z' + XYZ' + XYZ$$
$$= \Sigma (0, 3, 4, 6, 7)$$

$$F = (X + Y + Z')(X + Y' + Z)(X' + Y + Z')$$
$$= \Pi (1, 2, 5)$$

Note equivalences:

$\Sigma (0, 3, 4, 6, 7) = \Pi (1, 2, 5)$

$\Sigma (0, 3, 4, 6, 7)]' = \Sigma (1, 2, 5) = \Pi (0, 3, 4, 6, 7)$

$\Pi (1, 2, 5)]' = \Pi (0, 3, 4, 6, 7) = \Sigma (1, 2, 5)$

Exclusive-OR (XOR) Function

- XOR is often denoted by the symbol ⊕

Logic operation of XOR

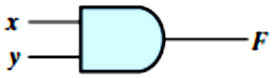
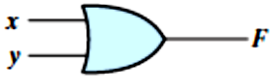
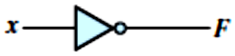
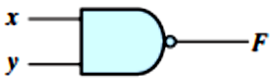
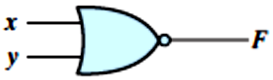
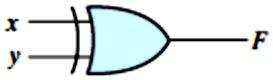
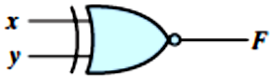
- $X \oplus Y = X\,{}^{1}Y. + X.Y\,{}^{1}$
- Equal to 1 if **only x** is equal to 1 or if **only y** is equal to 1, but not when both are equal to 1
- It.s complement, exclusive-NOR (XNOR), is often denoted by the symbol ⊙

Logic operationof X - NOR

- $X \odot Y = XY + X\,{}^{1}Y\,{}^{1}$
- It is equal to 1 if **both x and y** is equal to 1 or if both are equal to 0
- Seldom used in general Boolean functions
- Particularly useful in arithmetic operations and error detection and correction circuits

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 number binary inputs in order to perform a particular logical function. The type of gates available are NOT, AND, OR, NAND, NOR, EX – OR & EX – NOR.
- 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 also.
- 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. The truth table indicates the outputs for different possibilities of the inputs.

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

Positive and Negative Logic:

The binary signal at the inputs and outputs of any gate has one of two values, except during transition. One signal value represents logic 1 and the other logic 0. Since two signal values are assigned to two logic values, there exist two different assignments of signal level to logic value, as shown in Fig. below.

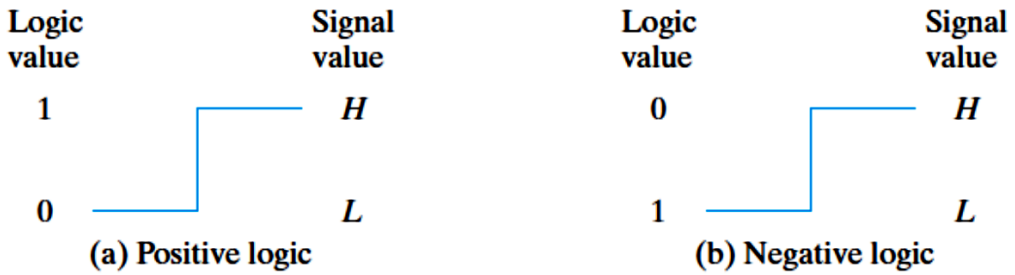


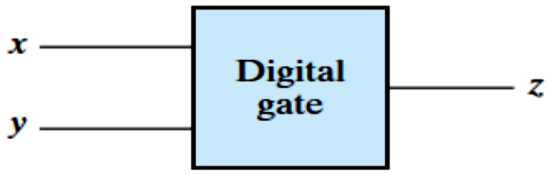
Fig: Signal assignment and logic polarity

- The higher signal level is designated by H and the lower signal level by L. **Choosing the high-level H to represent logic 1 defines a positive logic system.**
- **Choosing the low-level L to represent logic 1 defines a negative logic system.**
- The terms positive and negative are somewhat misleading, since both signals may be positive or both may be negative. It is not the actual values of the signals that determine the type of logic, but rather the assignment of logic values to the relative amplitudes of the two signal levels.
- Hardware digital gates are defined in terms of signal values such as H and L. It is up to the user to decide on a positive or negative logic polarity.

Consider, for example, the electronic gate shown in Fig. (b) below. The truth table for this gate is listed in Fig. (a) below. It specifies the physical behavior of the gate when H is 3 V and L is 0 V.

x	*y*	*z*
L	*L*	*L*
L	*H*	*L*
H	*L*	*L*
H	*H*	*H*

(a) Truth table with *H* and *L*



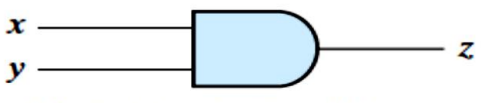
(b) Gate block diagram

The truth table of Fig. (c) assumes a positive logic assignment, with H = 1 and L = 0. This truth table is the same as the one for the AND operation.

The graphic symbol for a positive logic AND gate is shown in Fig.(d) below.

x	*y*	*z*
0	0	0
0	1	0
1	0	0
1	1	1

(c) Truth table for positive logic

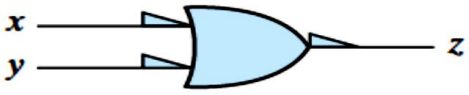


(d) Positive logic AND gate

Now consider the negative logic assignment for the same physical gate with L = 1 and H = 0. The result is the truth table of Fig. (e) Below. This table represents the OR operation, even though the entries are reversed. The graphic symbol for the negative logic OR gate is shown in Fig. (f) Below. The small triangles in the inputs and output designate a polarity indicator, the presence of which along a terminal signifies that negative logic is assumed for the signal. Thus, the same physical gate can operate either as a positive-logic AND gate or as a negative-logic OR gate.

x	y	z
1	1	1
1	0	1
0	1	1
0	0	0

(e) Truth table for negative logic



(f) Negative logic OR gate

Fig: Demonstration of positive and negative logic

PROPERTIES OF X-OR GATE:

1. **$A \oplus A = 0$;** Output is '0' when inputs are same.

   Proof: W.K.T, $x \oplus y = xy^1 + x^1y$

   From above equation, $A \oplus A = AA^1 + A^1A$
   $= 0 + 0$
   $A \oplus A = 0$

2. **$A \oplus A^1 = 1$** Output is '1' when inputs are different.

   Proof: W.K.T, $x \oplus y = xy^1 + x^1y$

   From above equation, $A \oplus A^1 = A(A^1)^1 + A^1A^1$
   $= AA + A^1$

   $= A + A^1$
   $A \oplus A^1 = 1$

3. **$A \oplus 1 = A^1$** XOR as inverter.

   When one input of XOR gate is connected to logic 1 .we get the complement of the other input of XOR gate.

   $1 \oplus 0 = 1$
   $1 \oplus 1 = 0$ (When one input is tied to logic "1", Output is complement form of the other input).

   Proof: W.K.T, $x \oplus y = xy^1 + x^1y$

   From above equation, $A \oplus 1 = A.1^1 + A^1.1$
   $= A \cdot 0 + A^1$
   $= 0 + A^1$
   $A \oplus 1 = A^1$

4. **$A \oplus 0 = A$** XOR as Non - inverter.

When one input of XOR gate is connected to logic 0 .we get the same output as other input of XOR gate.

$0 \oplus 0 = 0$
$0 \oplus 1 = 1$ (When one input is tied to logic "0", Output is same as the other Input) .

Proof: W.K.T, $x \oplus y = xy^1 + x^1y$

From above equation, $A \oplus 0 = A.0^1 + A^1.0$

$= A \cdot 1 + A^1 \cdot 0$

$= A + 0$

$A \oplus 0 = A$

5. **XOR as Modulo – 2 adder:**

The XOR gate can be used as Modulo – 2 adder. Because its truth table is same as truth table of the Modulo – 2 adder.

0 + 0 =	0		0 ⊕ 0 =	0
0 + 1 =	1		0 ⊕ 1 =	1
1 + 0 =	1	(Equal)	1 ⊕ 0 =	1
1 + 1 =	0		1 ⊕ 1 =	0

6. **$AB \oplus AC = A(B \oplus C)$**

A B C	AB	AC	AB ⊕ AC	B ⊕ C	A (B ⊕ C)
0 0 0	0	0	0	0	0
0 0 1	0	0	0	1	0
0 1 0	0	0	0	1	0
0 1 1	0	0	0	0	0
1 0 0	0	0	0	0	0
1 0 1	0	1	1	1	1
1 1 0	1	0	1	1	1
1 1 1	1	1	0	0	0

7. **If** $A \oplus B = C$, **then**
   $A \oplus C = B$
   $B \oplus C = A$ **&**
   $A \oplus B \oplus C = 0$

A	B	$A \oplus B = C$	$A \oplus C = B$	$B \oplus C = A$	$A \oplus B \oplus C = 0$
0	0	0	0	0	0
0	1	1	1	0	0
1	0	1	0	1	0
1	1	0	1	1	0

Note:

For 3 input XOR gate, output is one only for odd number of logic 1 inputs.

Alternate Logic Gate Symbols:

- Most of the logic networks use standard symbols. But in some networks an alternative set of symbols is used in addition to standard symbols.
- The alternative set of symbols for the 5 basic gates is shown in figure below. These alternate symbols are equivalent to standard and their equivalence can be proved using DeMorgan's theorems.
- For example, we know that the output expression for standard NAND symbol is $(AB)^1 = A^1 + B^1$, which is same as the output expression of alternate gate symbol.

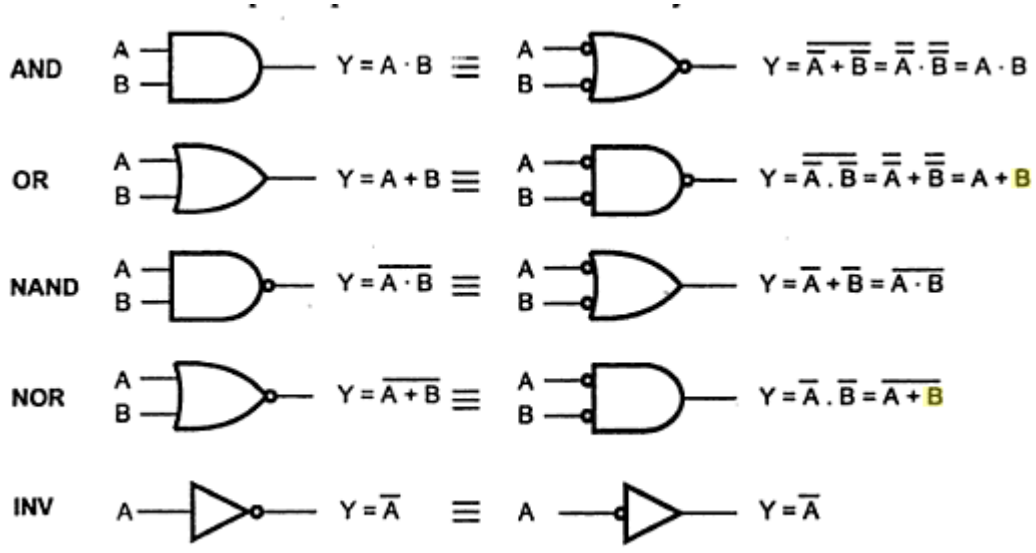


Fig. 2.25 Standard and alternate symbols for basic logic gates

Example 2.1 : *Draw the circuit shown in Fig. 2.26 using alternate symbols.*

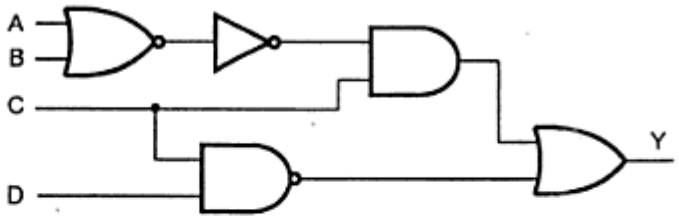


Fig. 2.26

Solution :

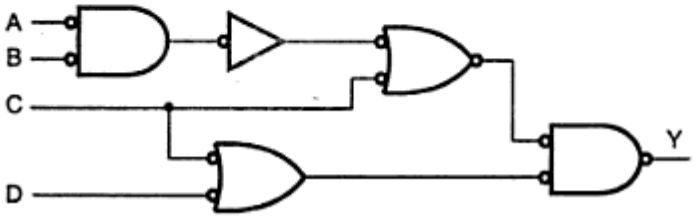


Fig. 2.26 (a)

IC'S OF DIFFERENT LOGIC GATES:

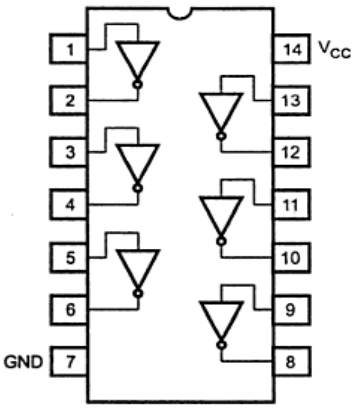


Fig. 2.4 Pin Diagram of a 7404

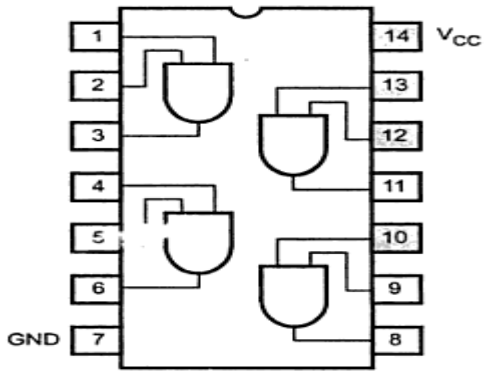


Fig. 2.10 Pin diagram of a 7408

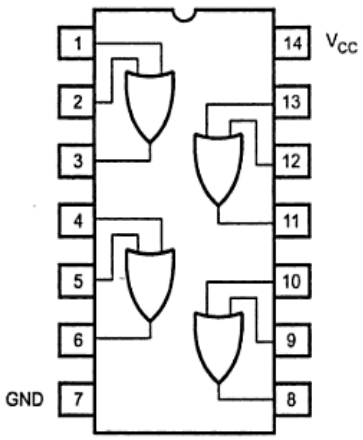


Fig. 2.16 Pin diagram of a 7432

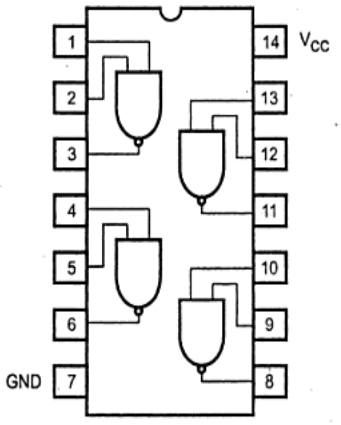


Fig. 2.20 Pin diagram of a 7400

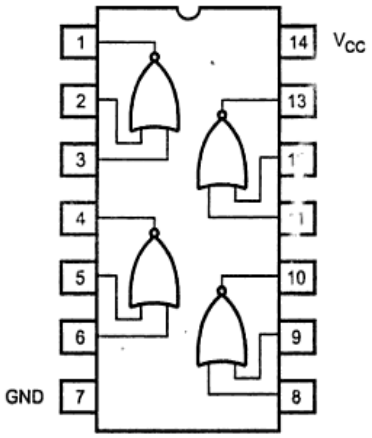


Fig. 2.24 Pin diagram of a 7402

UNIVERSAL GATES

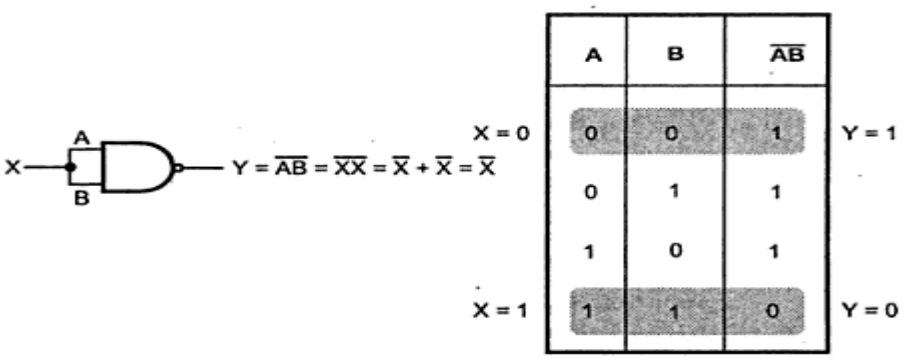
The NAND and NOR gates are known as universal gates, since any logic function can be implemented using NAND or NOR gates. For any logic gate to be fabricated, the NAND and NOR plays an important role.

Universality of NAND Gates:

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

NOT gate realization using NAND:

An inverter can be made from a NAND gate by connecting all of the inputs together and creating a to produce a single common input as shown in figure for a two – input gate.



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

Fig. 2.27 NOT function using NAND gate

AND gate realization using NAND:

An AND gate function can be generated using only NAND gates. It is generated by simply inverting output of NAND gate; i.e. $((AB)^1)^1$ = AB. Figure below shows the 2 input AND gate using NAND gates.

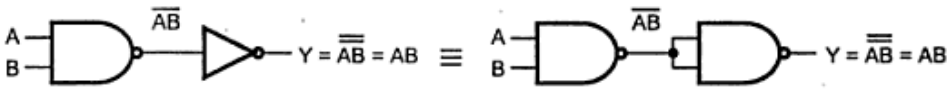


Fig. 2.28 AND function using NAND gates

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 2.7 Truth table

OR gate realization using NAND:

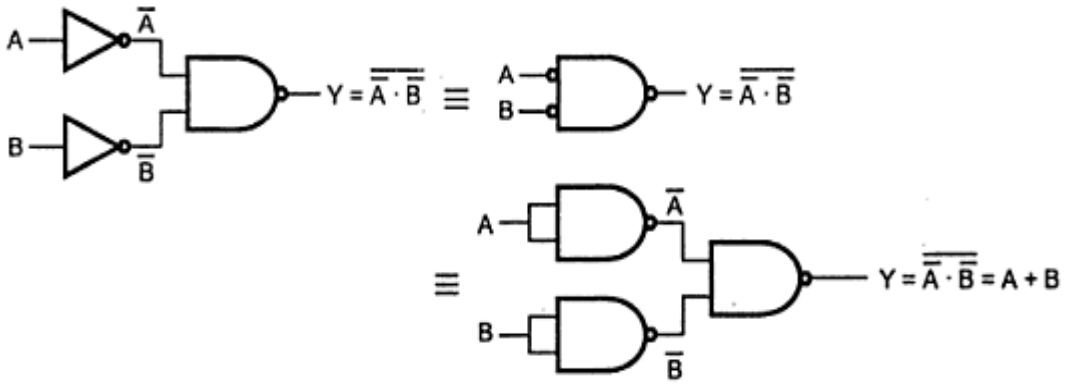


Fig. 2.29 OR function using only NAND gates

NOR gate realization using NAND:

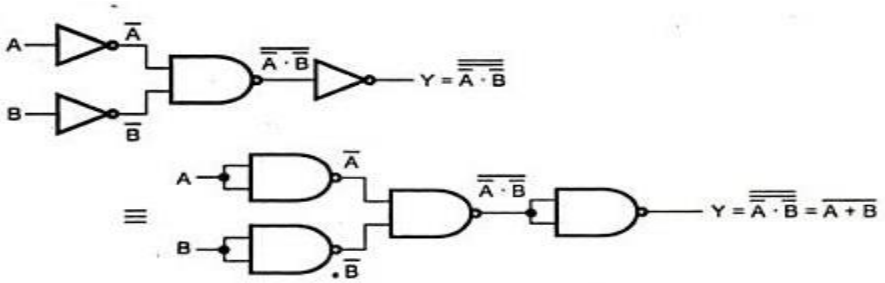


Fig. 2.30 NOR function using only NAND gates

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

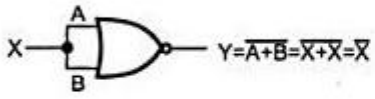
≡

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

Universality of NOR Gates:

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

NOT gate realization using NOR:



	A	B	$\overline{A+B}$	
X = 0	0	0	1	Y = 1
	0	1	0	
	1	0	0	
X = 1	1	1	0	Y = 0

Fig. 2.31 NOT function using NOR gate

OR gate realization using NOR:

Fig. 2.32 OR function using NOR gates

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

=

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

Table 2.9 Truth table

AND gate realization using NOR:

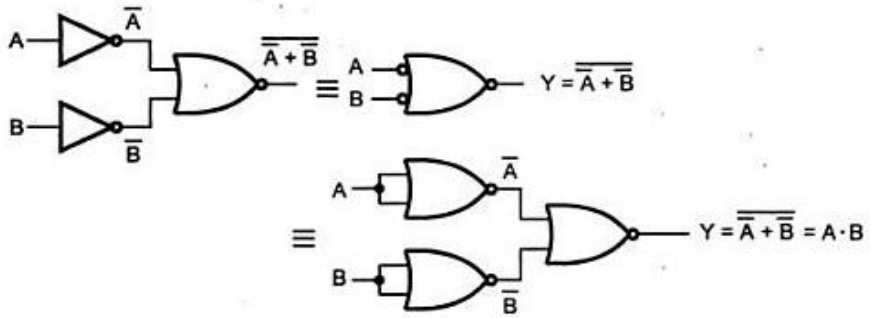


Fig. 2.33 AND function using NOR gates

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

A	B	A · B
0	0	0
0	1	0
1	0	0
1	1	1

≡

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

Table 2.10 Truth table

NAND gate realization using NOR:

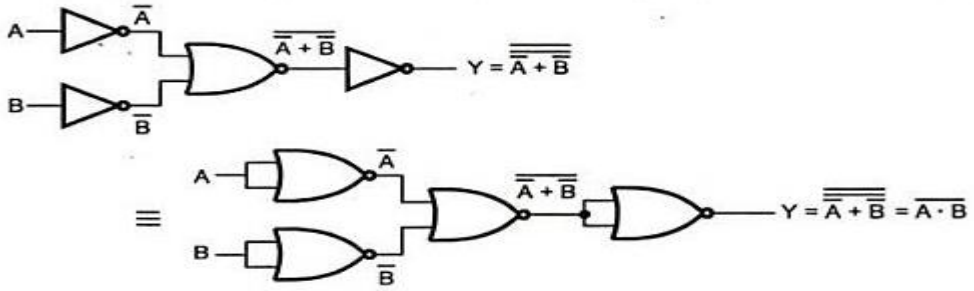


Fig. 2.34 NAND function using only NOR gates

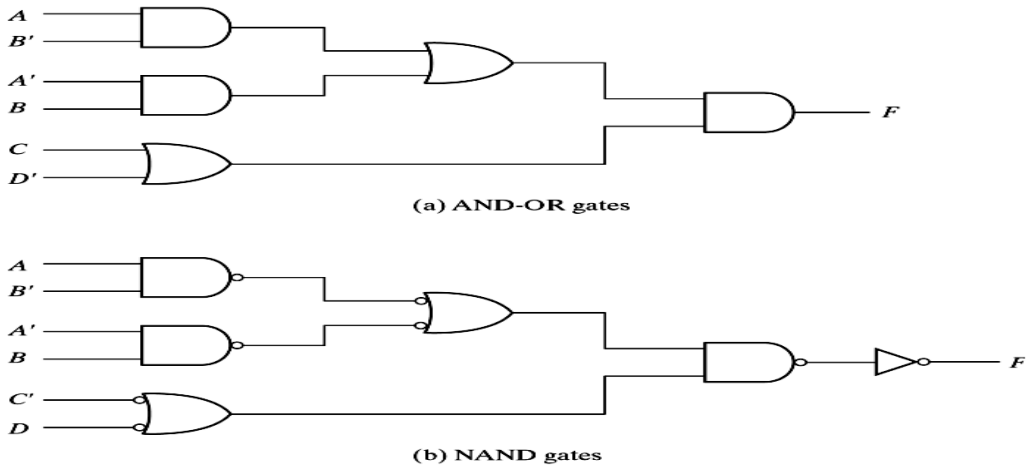
A	B	$\overline{A\cdot B}$
0	0	1
0	1	1
1	0	1
1	1	0

=

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

Truth table 2.11

Multi-Level Gate Implementation

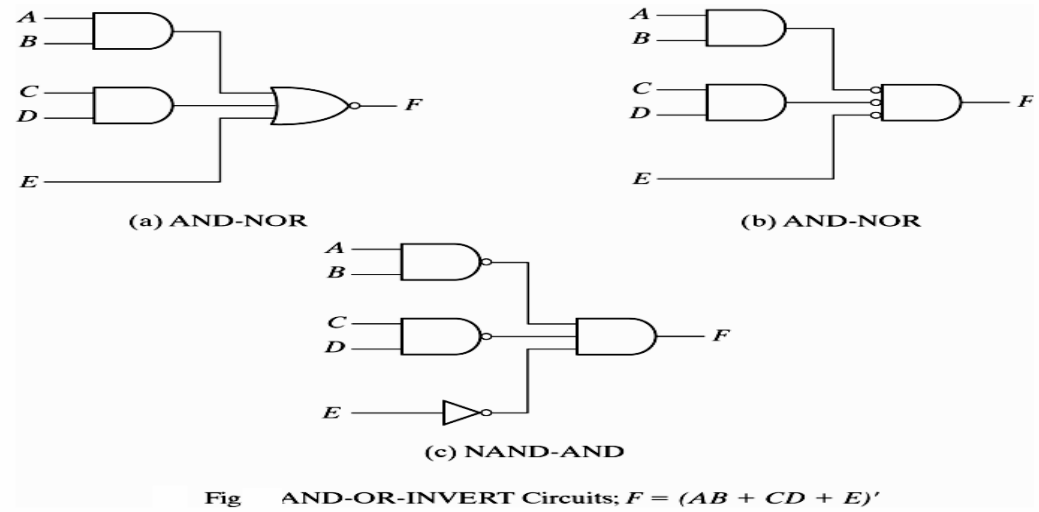


(a) AND-OR gates

(b) NAND gates

Fig. : Implementing $F = (AB' + A'B)(C + D')$

ND-OR-INVERT Gate Implementation



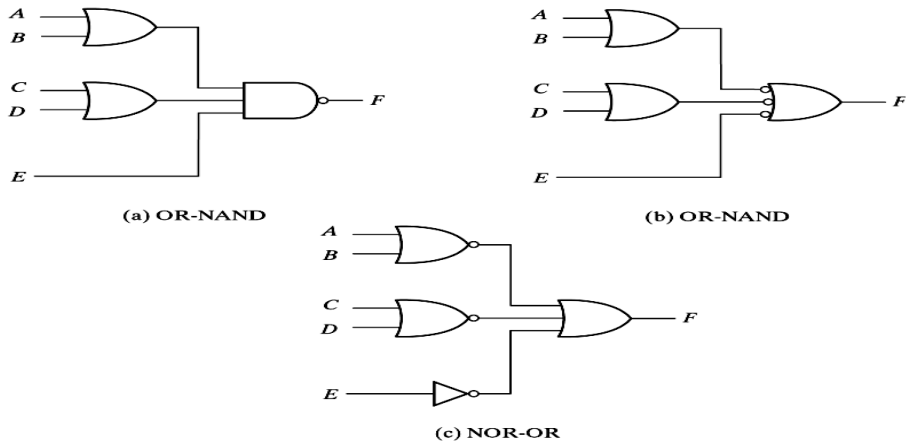
(a) AND-NOR

(b) AND-NOR

(c) NAND-AND

Fig AND-OR-INVERT Circuits; $F = (AB + CD + E)'$

OR-AND-INVERT Gate Implementation



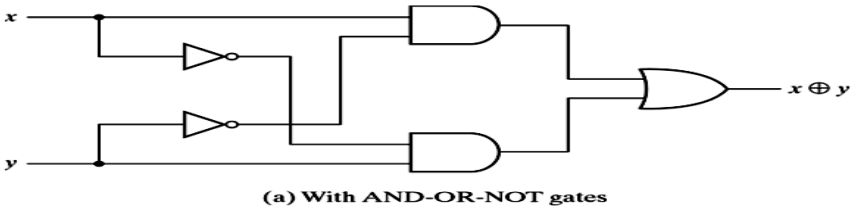
(a) OR-NAND

(b) OR-NAND

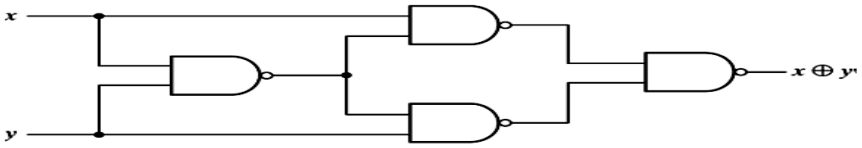
(c) NOR-OR

Fig. OR-AND-INVERT Circuits; $F = [(A + B)(C + D)E]'$

EXCLUSIVE-OR FUNCTION



(a) With AND-OR-NOT gates



(b) With NAND gates

Fig. Exclusive-OR Implementations

Sum of Minterms

- A Boolean function can be expressed algebracially from a given truth table by
  - forming a minterm for each combination of the variables that produces a 1 in the function
  - And then taking the OR of all those terms
- $f_1 = x'y'z + xy'z' + xyz = m_1 + m_4 + m_7$
- $f_2 = x'yz + xy'z + xyz' + xyz = m_3 + m_5 + m_6 + m_7$

x	y	z	f_1	f_2
0	0	0	0	0
0	0	1	1	0
0	1	0	0	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	0

The Complementation

The minterms that produce a 0

f1' = m0 + m2 + m3 + m5 + m6
= x'y'z'+x'yz'+x'yz+xy'z+xyz'

f1 = (f1')'
= (x+y+z) (x+y'+z) (x+y'+z') (x'+y+z') (x'+y'+z)
= M0 M2 M3 M5 M6

Example: Sum of Minterms

F = A+B'C
= A (B+B') + B'C
= AB +AB' + B'C
= AB(C+C') + AB'(C+C') + (A+A')B'C
=ABC+ABC'+AB'C+AB'C'+A'B'C
F = A'B'C +AB'C' +AB'C+ABC'+ ABC
= m 1 + m4 + m5 + m6 + m7
F(A,B,C) = Σ(1, 4, 5, 6, 7)

Σ stands for the ORing of number of minterms

From Truth Table for Sum of Minterms

A	B	C	F
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

F = A+B'C

- All terms that A=1 in Truth table
- All terms that B' AND C
-

F =m1 + m4 +m5 + m6 + m7
F(A B C) = Σ(1 4 5 6 7)

Product of Maxterms

Bring into a form of OR terms

- By using the distributive law
- x + yz = (x + y)(x + z)
- Then any missing variable x in each OR terms is ORed with xx'

= (x+y+zz')(x+z+yy')
=(x+y+z)(x+y+z')(x+y'+z)

Example:

F = xy + x'z
= (xy + x') (xy +z)
= (x+x')(y+x')(x+z)(y+z)
= (x'+y)(x+z)(y+z)
x'+y = x' + y + zz'
= (x'+y+z)(x'+y+z')
F = (x+y+z)(x+y'+z)(x'+y+z)(x'+y+z')
= M0 M2 M4 M5
F(x,y,z) = Π(0,2,4,5)

Conversion between canonical forms:

- The complement of a function expressed as the sum of minterms equals the sum of minterms missing from the original function
  - Original function: F(A,B,C) = $\Sigma(1, 4, 5, 6, 7)$
  - The complement of F: F'(A,B,C) = $\Sigma(0, 2, 3)=m_0+m_2+m_3$
- If take the complement of F' by DeMorgan's theorem
  - $F=(F')' =(m_0+m_2+m_3)'=M_0M_2M_3= \Pi(0,2,3)$
- The maxterm with subscript *j* is a complement of the minterm with the same subscript *j*, and vice versa $m_j' = M_j, \qquad M_j' = m_j$

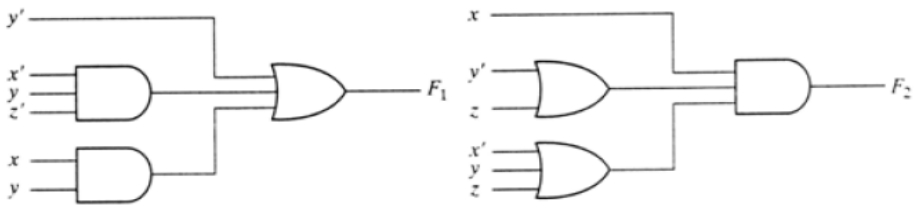
HOW TO CONVERT?

- Interchanging the symbols Σ and Π and list those numbers missing from the original form by using a truth table
- Example: F=xy+x'z
  - In sum of minterms:
    - $F(x,y,z)= \Sigma(1, 3, 6, 7)$
  - In product of maxterms:
    - $F(x,y,z)= \Pi(0,2,4,5)$

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

Standard Forms:

- Canonical forms are seldom used
- Standard form
  - sum of products: ORing all the product terms
    - Example: F_1 = y' + zy+ x'yz'
  - product of sums: ANDing all the sum terms
    - Example: F_2 = x(y'+z)(x'+y+z'+w)



(a) Sum of Products (b) Product of Sums

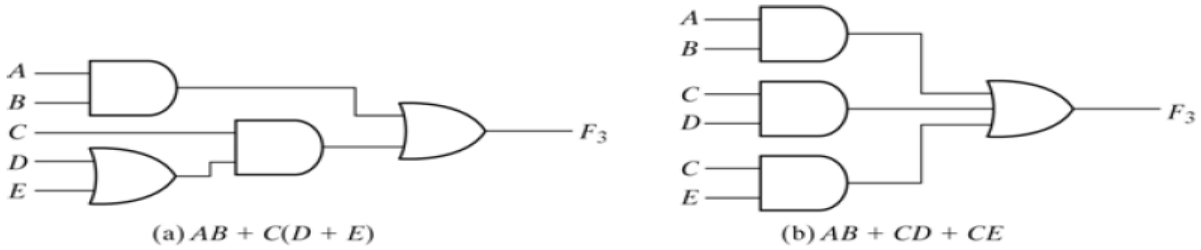
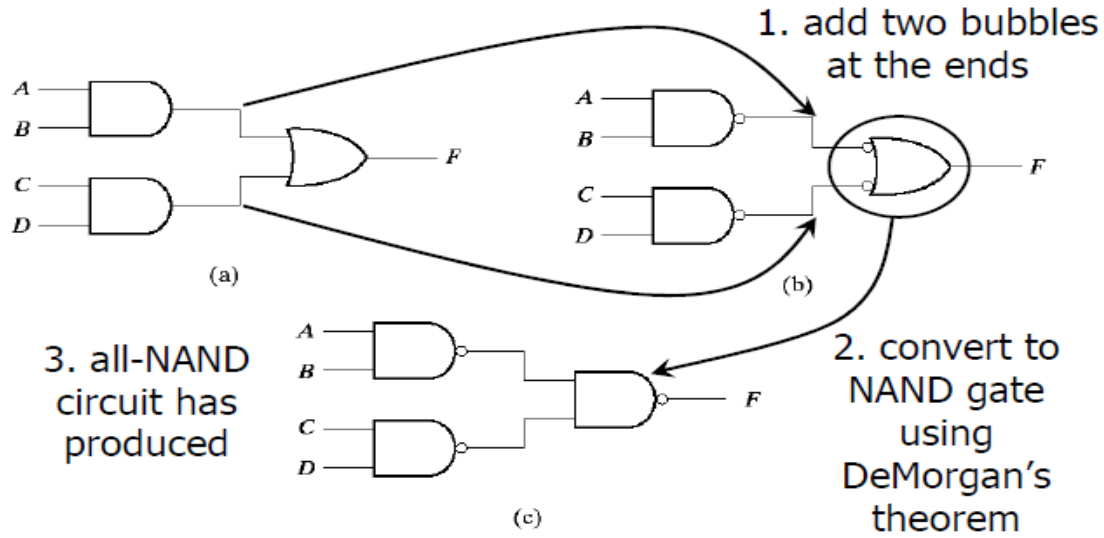


Fig. Three- and Two-Level implementation

Two-Level Implementation (NAND):

It's easy to implement a Boolean function with only NAND gates if converted from a **sum of products** form

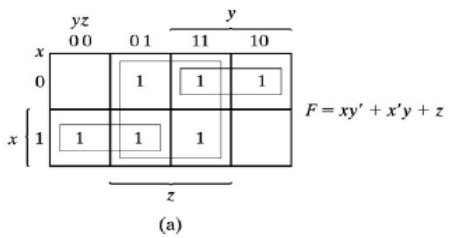
Ex: F = AB+CD = ((AB)'(CD)')'

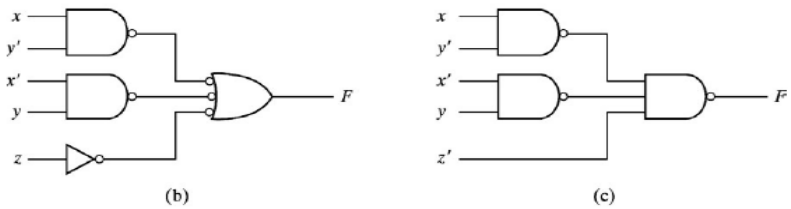


$$F(x, y, z) = \sum(1,2,3,4,5,7)$$

Procedures:

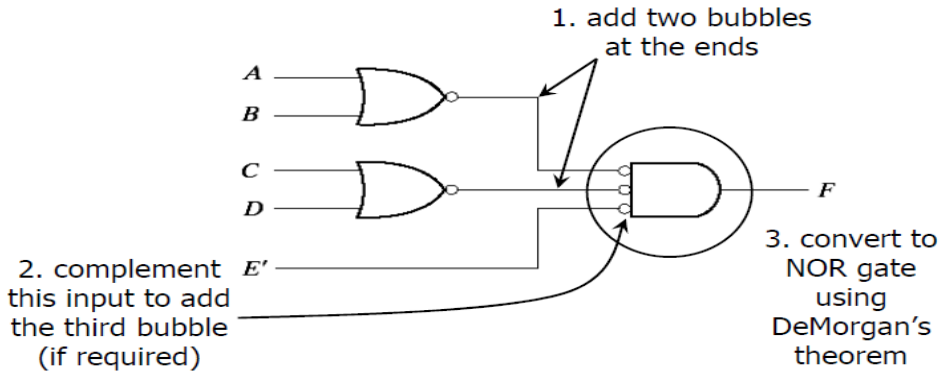
1. Simplify the function in sum of products
2. Draw NAND gates for the first level
3. Draw a single AND-invert or invert-OR in the second level
4. Add an inverter at the first level for the term with a single literal





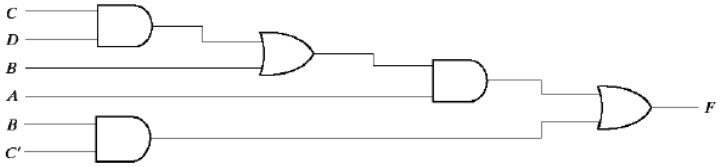
Two-Level Implementation (NOR):

- It's easy to implement a Boolean function with only NOR gates if converted from a **product of sums** form
- Ex: F=(A+B)(C+D)E

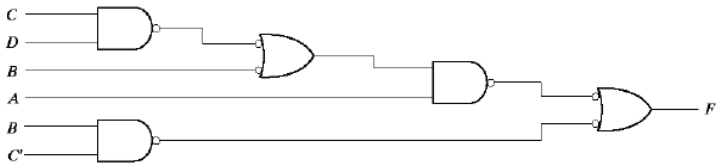


Multilevel NAND Circuits:

F=A(CD+B)+BC'



(a) AND-OR gates



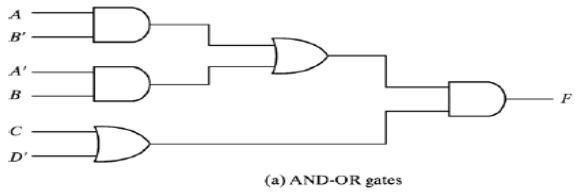
(a) NAND gates

- Procedures:
  1. Convert all AND gates to NAND gates with AND-invert symbols
  2. Convert all OR gates to NAND gates with invert-OR symbols
  3. Check all bubbles and insert an inverter for the bubble that are not compensated by another bubble

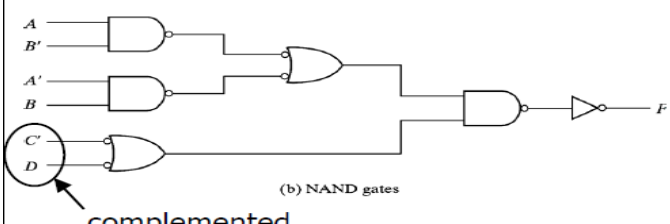
Multilevel NOR Circuits:

- For NOR gates, AND → invert-AND, OR → OR-invert
- Other procedures are the same as those for NAND

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



(a) AND-OR gates



(b) NAND gates

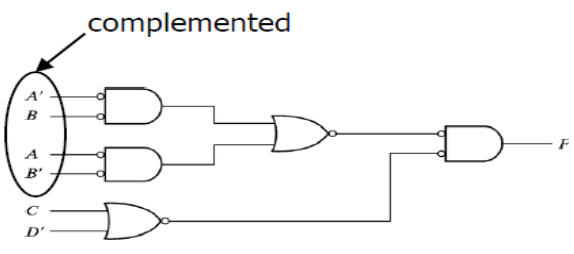


Fig. 3-27 Implementing $F = (AB' + A'B)(C + D')$ with NOR Gates

IMPLEMENT WITH TWO-LEVEL FORMS:

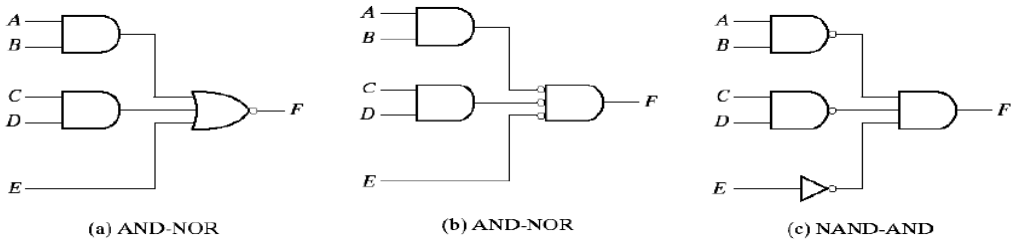
Table 3-3
Implementation with Other Two-Level Forms

Equivalent Nondegenerate Form (a)	Equivalent Nondegenerate Form (b)*	Implements the Function	Simplify F' in	To Get an Output of
AND-NOR	NAND-AND	AND-OR-INVERT	Sum of products by combining 0's in the map	F
OR-NAND	NOR-OR	OR-AND-INVERT	Product of sums by combining 1's in the map and then complementing	F

*Form (b) requires an inverter for a single literal term.

AND-OR-INVERT Implementation:

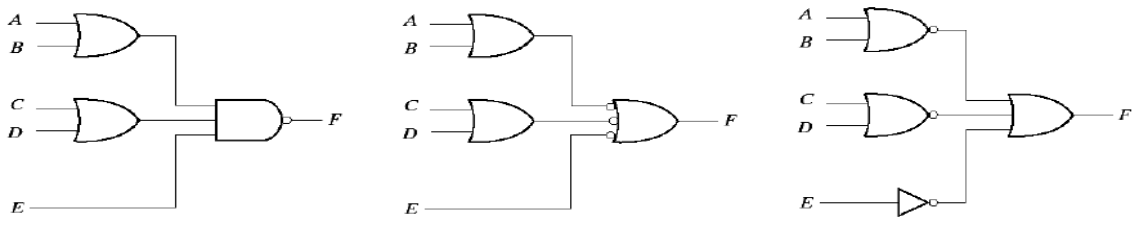
- NAND-AND and AND-NOR are equivalent and both perform the AND-OR-INVERT (**AOI**) function
- Require sum-of-products form in nature
  - When starting from product-of-sums form, complement it using DeMorgan's theorem to obtain sum-of-products form
- Ex: F = (AB + CD + E)'



(a) AND-NOR (b) AND-NOR (c) NAND-AND

OR-AND-INVERT Implementation:

- OR-NAND and NOR-OR are equivalent and both perform the OR-AND-INVERT (**OAI**) function
- Require product-of-sums form in nature
  - When starting from sum-of-products form, complement it using DeMorgan's theorem to obtain product-of-sums form
- Ex: F = [(A+B)(C+D)E]'



(a) OR-NAND (b) OR-NAND (c) NOR-OR

Problems

1. **Reduce A (A + B)**

A (A + B) = AA + AB

= A (1 + B) [1 + B = 1]

= A.

2. **Reduce A'B'C' + A'BC' + A'BC**

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

= A'C' + A'BC [A + A' = 1]

= A'(C' + BC)

= A'(C' + B) [A + A'B = A + B]

3. **Reduce AB + (AC)' + AB'C (AB + C)**

AB + (AC)' + AB'C (AB + C) = AB + (AC)' + AAB'BC + AB'CC

= AB + (AC)' + AB'CC [A.A' = 0]

= AB + (AC)' + AB'C [A.A = 1]

= AB + A' + C' +AB'C [(AB)' = A' + B']

= A' + B + C' + AB'C [A + AB' = A + B]

= A' + B'C + B + C' [A + A'B = A + B]

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

=A' + B + C' + B'

=A' + C' + 1 = 1 [A + 1 =1]

4. **Simplify the following expression Y = (A + B) (A + C') (B' + C')**

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

= (AA' + AC +A'B +BC) (B' + C') [A.A' = 0]

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

= AB'C + ACC' + A'BB' + A'BC' + BB'C + BCC'

= AB'C + A'BC'

5. **Show that (X + Y' + XY) (X + Y') (X'Y) = 0**

(X + Y' + XY)(X + Y')(X'Y) = (X + Y' + X) (X + Y') (X' + Y) [A + A'B = A + B]

= (X + Y') (X + Y') (X'Y) [A + A = 1]

= (X + Y') (X'Y) [A.A = 1]

= X.X' + Y'.X'.Y = 0 [A.A' = 0]

6. **Prove that ABC + ABC' + AB'C + A'BC = AB + AC + BC**

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

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

=A(B + C) + A'BC

=AB + AC + A'BC

=B(A + C) + AC

=AB + BC + AC

=AB + AC +BC ...Proved

7. **Convert the given expression in canonical SOP form Y = AC + AB + BC**

Y = AC + AB + BC

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

=ABC + ABC' + AB'C + AB'C' + ABC + ABC' + ABC

=ABC + ABC' +AB'C + AB'C' [A + A =1]

8. **Find the complement of the functions F1 = x'yz' + x'y'z and F2 = x (y'z' + yz). By applying De-Morgan's theorem.**

F1' = (x'yz' + x'y'z)'

= (x'yz')'(x'y'z)'

= (x + y' + z)(x + y +z')

F2' = [x (y'z' + yz)]'

= x' + (y'z' + yz)'

= x' + (y'z')'(yz)'

= x' + (y + z) (y' + z')

9. Simplify the following expression

Y = (A + B) (A + C) (B + C)

= (A A + A C + A B + B C) (B + C)

= (A C + A B + B C) (B + C)

= A B C + A C C + A B B + A B C + B B C + B C C

= A B C

Problem: 2-1

Demonstrate by means of truth tables the validity of the following identities:

(a) DeMorgan's theorem for three variables: (x+y+z)' = x'y'z' and (xyz)'=x'+y'+z'
(b) The distributive law: x+yz = (x+y)(x+z)

Solution:

(a)

x	Y	z	x+y+z	(x+y+z)'
0	0	0	0	1
0	0	1	1	0
0	1	0	1	0
0	1	1	1	0
1	0	0	1	0
1	0	1	1	0
1	1	0	1	0
1	1	1	1	0

x'	y'	z'	x'y'z'
1	1	1	1
1	1	0	0
1	0	1	0
1	0	0	0
0	1	1	0
0	1	0	0
0	0	1	0
0	0	0	0

x	Y	z	xyz	(xyz)'
0	0	0	0	1
0	0	1	0	1
0	1	0	0	1
0	1	1	0	1
1	0	0	0	1
1	0	1	0	1
1	1	0	0	1
1	1	1	1	0

x'	y'	z'	x'+y'+z'
1	1	1	1
1	1	0	1
1	0	1	1
1	0	0	1
0	1	1	1
0	1	0	1
0	0	1	1
0	0	0	0

(b)

x	Y	z	yz	x+yz
0	0	0	0	0
0	0	1	0	0
0	1	0	0	0
0	1	1	1	1
1	0	0	0	1
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

x+y	x+z	(x+y)(x+z)
0	0	0
0	1	0
1	0	0
1	1	1
1	1	1
1	1	1
1	1	1
1	1	1

Problem: 2-2 : Reduce the following Boolean expressions to the indicated number of literals:

(a) A'C' + ABC + AC' to three literals
(b) (x'y'+z)' + z + xy + wz to three literals
(c) A'B(D'+C'D) + B(A+A'CD) to one literal
(d) (A'+C)(A'+C')(A+B+C'D) to four literals

Solution:

(a) A'C' + ABC + AC' = A'C' + AC' + ABC
= C'(A'+ A) + ABC

= C'·1 + ABC

= C' + ABC

= (C'+ AB)(C'+C) [distributive]

= AB + C'

(b) (x'y'+z)' + z + xy + wz = (x'y'+z)' + z + wz + xy
= (x'y'+z)' + z(1+ w) + xy

= (x'y'+z)' + z + xy

= (x + y)z' + z + xy [DeMorgan]

= (z + (x + y)) · (z + z') + xy [distributive]

= (z + (x + y)) · 1 + xy

= x + y + z + xy

= x + y + z [absorption]

(c) A'B(D' + C'D) + B(A+A'CD) = A'BD' + A'BC'D + AB+ A'BCD
= A'BD(C+C')+ A'BD'+ AB

= A'BD+ A'BD'+ AB

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

= A'B+ AB

= B(A'+ A)

= B

(d) (A'+C)(A'+C')(A+B+C'D) = (A'+C)(A'+C')(A+B+C'D)
= (A' + CC')(A + B + C'D)

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

= A'A + A'B + A'C'D

= A'B + A'C'D

= A'(B + C'D)

Problem 2-3:

Find the complement of $F = x + yz$; then show that $FF' = 0$ and $F + F' = 1$

Solution:

$F = x + yz$

The dual of F is: $x \bullet (y + z)$

Complement each literal: $x' \bullet (y' + z') = F'$

$$FF' = (x + yz) \bullet (x' \bullet (y' + z')) = (xx' + x'yz) \bullet (y' + z') = x'yz \bullet (y' + z') = x'yy'z + x'yzz' = 0$$

$$F + F' = (x + yz) + (x' \bullet (y' + z')) = (x + yz + x') + (x + yz + y' + z') = (1 + yz) + (x + yz + y' + z')$$
$$= 1 + (x + yz + y' + z') = 1$$

Problem 2-4:

List the truth table of the function: $F = xy + xy' + y'z$

Solution:

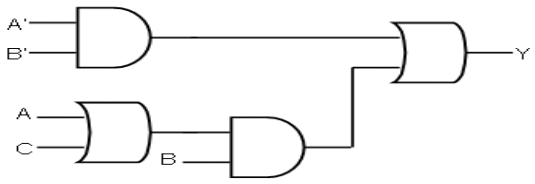
The truth table is:

x	y	Z	F
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

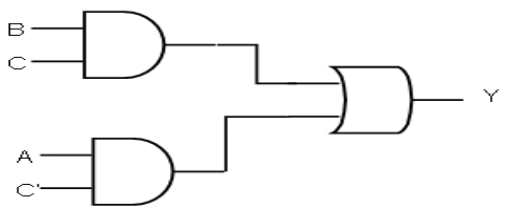
Problem 2-4:

Draw the logical diagrams for the following Boolean expressions:

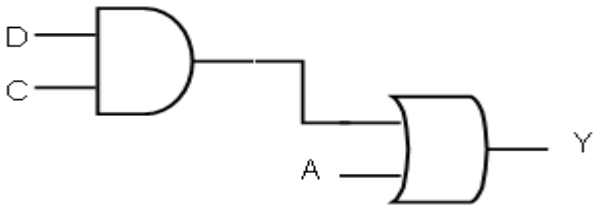
(a) Y=A'B'+B(A+C).



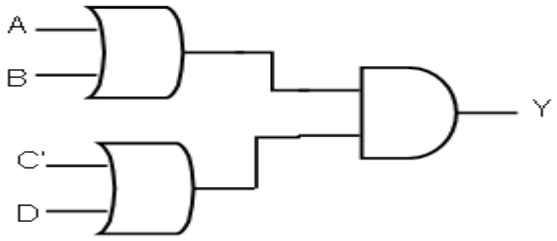
(b) Y=BC+AC'.



(c) Y=A+CD.



(d) Y=(A+B)(C'+D).



Problem 2-5

Given the Boolean function F=xy'z+x'y'z+w'xy+wx'y+wxy.

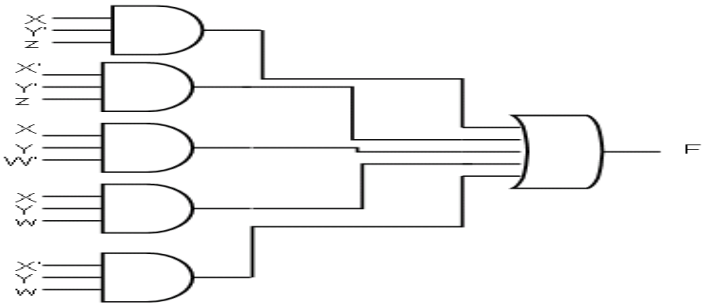
(a) Obtain the truth table of the function.
(b) Draw the logical diagram using the original Boolean expression.
(c) Simplify the function to a minimum number of literals using Boolean algebra.
(d) Obtain the truth table of the function using the simplified expression.
(e) Draw the logical diagram from the simplified expression and compare the total number of gates with the diagram of part (b).

Solutions:

(a) The truth table of the function:

X	Y	z	w	F	F(simplified)
0	0	0	0	0	0
0	0	0	1	0	0
0	0	1	0	1	1
0	0	1	1	1	1
0	1	0	0	0	0
0	1	0	1	1	1
0	1	1	0	0	0
0	1	1	1	1	1
1	0	0	0	0	0
1	0	0	1	0	0
1	0	1	0	1	1
1	0	1	1	1	1
1	1	0	0	1	1
1	1	0	1	1	1
1	1	1	0	1	1
1	1	1	1	1	1

(b)The logic diagram:



(b) The simplified function:

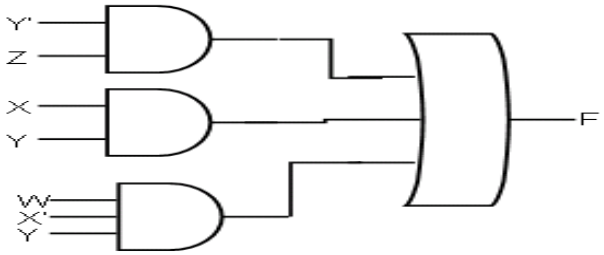
F =XY'Z+X'Y'Z+ WXY + W'XY+ WX'Y

=Y'Z (X+X') +XY (W+W') + WX'Y

=Y'Z+ XY+ WX'Y

(c) The truth table of the simplified function has been added in the truth table of (a) as F(simplified)

(d) Logic circuit for simplified function



For 1st design there are 5 AND gates with 3 inputs and 1 OR gate with 5 inputs.

For 2nd design there are 2 AND gates with 2 inputs and 1 OR gate with 3 inputs and 1 AND gate with 3 inputs.

Problem 2-6:

Convert the following to the other canonical form:

(a) F(x, y, z)= $\sum(1,3,7)$ (b) F(A,B,C,D)= $\prod(0,1,2,3,4,6,12)$

Solution:

(a) F(x, y, z)= $\sum(1,3,7) = \prod(0,2,4,5,6)$

$F(x, y, z)=(x+y+z)\bullet(x+\bar{y}+z)\bullet(\bar{x}+y+z)\bullet(\bar{x}+y+\bar{z})\bullet(\bar{x}+\bar{y}+z)$

(b) F(A,B,C,D)= $\prod(0,1,2,3,4,6,12)= \sum(5,7,8,9,10,11,13,14,15)$

F(A,B,C,D)=

$(\bar{A}\bar{B}\bar{C}D)+(\bar{A}BCD)+(A\bar{B}\bar{C}\bar{D})+(A\bar{B}\bar{C}D)+(A\bar{B}C\bar{D})+(A\bar{B}CD)+(AB\bar{C}D)+(ABC\bar{D})+(ABCD)$

Problem 2-7:

Show that the dual of the exclusive-OR is equal to its complement.

Solution:

XOR: $X \oplus Y = XY' + X'Y$

Dual of XOR: $= (X + Y') \bullet (X' + Y)$

$= XX' + XY + X'Y' + YY'$

$= XY + X'Y'$

Complement of XOR (XNOR) $= (X \oplus Y)'$

$= (XY' + X'Y)'$

$= (X' + Y) \bullet (X + Y')$

$= XX' + XY + X'Y' + YY'$

$= XY + X'Y'$

Problem 2-8:

Show that a positive logic NAND gate is a negative logic NOR gate and vice versa.

Solution:

Truth table for a NAND gate:

Truth table for positive logic NAND gate (L = 0 H = 1) with H and L:

X	Y	Z
L	L	H
L	H	H
H	L	H
H	H	L

Truth table for negative logic let L = 1, H = 0

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

This resulting truth table is that of the NOR gate using negative logic.