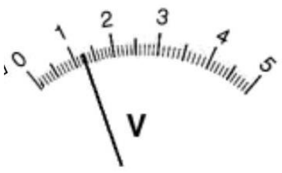
Unit 1
Binary Systems

Introduction

We are in "Information age" since digital systems have such a prominent and growing role in modern society. They are involved in our business transactions, communications, transportation, medical treatment and entertainment. In industrial world they are heavily employed in design, manufacturing, distribution and sales.

Analog System

Analog systems process analog signals (continuous time signals) which can take any value within a range, for example the output from a speaker or a microphone.



An analog meter can display any value within the range available on its scale. However, the precision of readings is limited by our ability to read them. E.g. meter on the right shows 1.25V because the pointer is estimated to be half way between 1.2 and 1.3. The analogue meter can show any value between 1.2 and 1.3 but we are unable to read the scale more precisely than about half a division.

Digital System

- Digital systems process digital signals, which can take only a limited number of values (discrete steps), usually just two values are used: the positive supply voltage (+Vs) and zero volts (0V).
- Digital systems contain devices such as logic gates, flip-flops, shift registers and counters.

A digital meter can display many values, but not every value within its range. For example the display on the right can show 6.25 and 6.26 but not a value between them. This is not a problem because digital meters normally have sufficient digits to show values more precisely than it is possible to read an analogue display.
The general purpose digital computer is a best known example of **digital system**.

Generic Digital computer structure

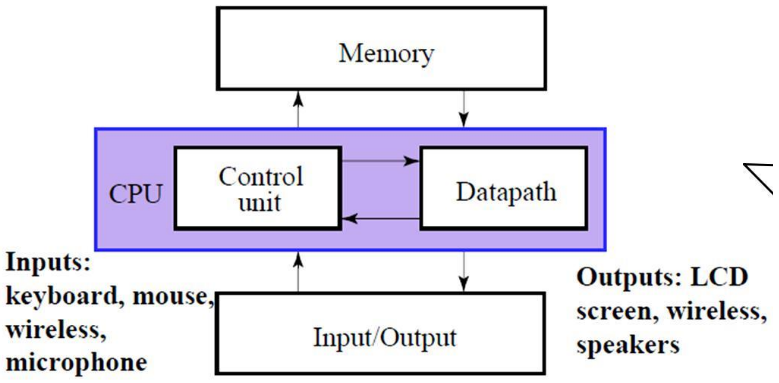

Memory
CPU
Control unit
Datapath
Inputs: keyboard, mouse, wireless, microphone
Input/Output
Outputs: LCD screen, wireless, speakers

Fig: Block diagram of digital computer

- ❖ Actually processor contains 4 functional modules: CPU, FPU, MMU and internal cache.
- ❖ Here only CPU is specified.

Working principle of generic digital computer: Memory stores programs as well as input, output and intermediate data. The datapath performs arithmetic and other data-processing operations as specified by the program. The control unit supervises the flow of information between the various units. A datapath, when combined with the control unit, forms a component referred to as a *central processing unit*, or CPU. The program and data prepared by the user are transferred into memory by means of an input device such as a keyboard. An output device, such as a CRT (cathode-ray tube) monitor, displays the results of the computations and presents them to the user.

Advantages of digital system:

- Have made possible many scientific, industrial, and commercial advances that would have been unattainable otherwise.
- Less expensive
- More reliable
- Easy to manipulate
- Flexibility and Compatibility
- Information storage can be easier in digital computer systems than in analog ones. New features can often be added to a digital system more easily too

Disadvantages of digital system:

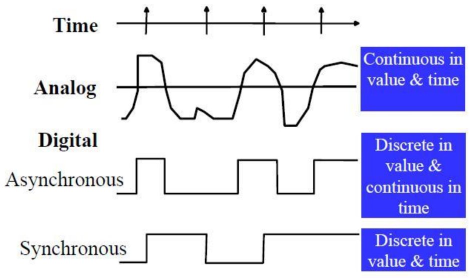
- Use more energy than analog circuits to accomplish the same tasks, thus producing more heat as well.
- Digital circuits are often fragile, in that if a single piece of digital data is lost or misinterpreted, the meaning of large blocks of related data can completely change.
- Digital computer manipulates discrete elements of information by means of a binary code.
- Quantization error during analog signal sampling.

Information Representation

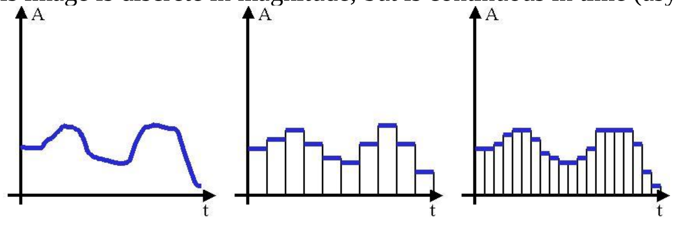
Signals

- Information variables represented by physical quantities.
- For digital systems, the variables take on discrete values.
- Two level or binary values are the most prevalent values in digital systems.
- Binary values are represented abstractly by:
  - digits 0 and 1
  - words (symbols) False (F) and True (T)
  - words (symbols) Low (L) and High (H)
  - and words On and Off.
- Binary values are represented by values or ranges of values of physical quantities

Signal Examples over time

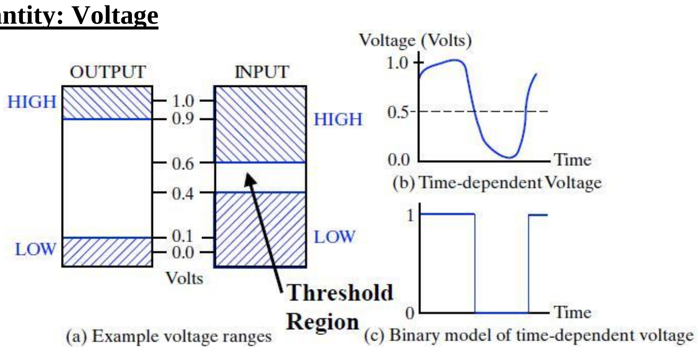

Time
Analog
Continuous in value & time
Digital
Asynchronous
Discrete in value & continuous in time
Synchronous
Discrete in value & time

Here is an example waveform of a quantized signal. Notice how the magnitude of the wave can only take certain values, and that creates a step-like appearance. This image is discrete in magnitude, but is continuous in time (asynchronous):



Analog signal – continuously varying | **Digital signal – large time divisions** | **Digital signal – small time divisions**

Signal Example – physical quantity: Voltage

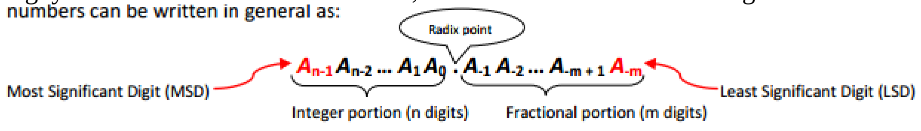


What are other physical quantities representing 0 and 1?

1. CPU: Voltage
- Disk: Magnetic Field Direction
2. CD: Surface Pits/Light
- Dynamic RAM: Electrical Charge

Number Systems

Here we discuss positional number systems with Positive radix (or base) ***r***. A number with radix ***r*** is represented by a string of digits as below i.e. wherever you guys see numbers of whatever bases, all numbers can be written in general as


numbers can be written in general as:
Radix point
A_{n-1} A_{n-2} ... A_1 A_0 . A_{-1} A_{-2} ... A_{-m+1} A_{-m}
Most Significant Digit (MSD)
Least Significant Digit (LSD)
Integer portion (n digits)
Fractional portion (m digits)

in which $\mathbf{0 \le A_i < r}$ (since each being a symbol for particular base system viz. for r = 10 (decimal number system) A_i will be one of 0,1,2,…,8,9). Subscript i gives the position of the coefficient and, hence, the weight r^i by which the coefficient must be multiplied. In general, a number in base *r* contains *r* digits, 0, l, 2... *r*- 1, and is expressed as a power series in *r* with the general form:

$$\mathbf{(Number)_r = A_{n-1}\, r^{n-1} + A_{n-2}\, r^{n-2} + \ldots + A_1\, r^1 + A_0\, r^0 + A_{-1}\, r^{-1} + A_{-2}\, r^{-2} + \ldots + A_{-m+1}\, r^{-m+1} + A_{-m}\, r^{-m}}$$

$$\text{(Number)}_r = \Big(\sum_{i=0}^{i=n-1} A_i \cdot r^i\Big) + \Big(\sum_{j=-m}^{j=-1} A_j \cdot r^j\Big)$$

(Integer Portion) + (Fraction Portion)

Decimal Number System (Base-10 system)
Radix (r) = 10
Symbols = 0 through r-1 = 0 through 10-1 = {0, 1, 2... 8, 9}
I am starting from base-10 system since it is used vastly in everyday arithmetic besides computers to represent numbers by strings of digits or symbols defined above, possibly with a *decimal point*. Depending on its position in the string, each digit has an associated value of an integer raised to the power of 10.
<u>Example</u>: decimal number 724.5 is interpreted to represent 7 hundreds plus 2 tens plus 4 units plus *5* tenths.

$$724.5 = 7 \text{ X } 10^2 + 2 \text{ X } 10^1 + 4 \text{ X } 10^0 + 5 \text{ X } 10^{-1}$$

Binary Number System (Base-2 system)
Radix (r) = 2
Symbols = 0 through r-1 = 0 through 2-1 = {0, 1}
A binary numbers are expressed with a string of 1'sand 0's and, possibly, a *binary point* within it. The decimal equivalent of a binary number can be found by expanding the number into a power series with a base of 2.
Example: $(11010.01)_2$ can be interpreted using power series as:
$(11010.01)_2 = 1 \text{ X } 2^4 + 1 \text{ X } 2^3 + 0 \text{ X } 2^2 + 1 \text{ X } 2^1 + 0 \text{ X } 2^0 + 0 \text{ X } 2^{-1} + 1 \text{ X } 2^{-2} = (26.25)_{10}$
Digits in a binary number are called bits (**B**inary dig**IT**s). When a bit is equal to 0, it does not contribute to the sum during the conversion. Therefore, the conversion to decimal can be obtained by adding the numbers with powers of 2 corresponding to the bits that are equal to 1. Looking at above example, $(11010.01)_2 = 16 + 8 + 2 + 0.25 = (26.25)_{10}$.

n	2^n	n	2^n	n	2^n
0	1	8	256	16	65,536
1	2	9	512	17	131,072
2	4	10	1,024	18	262,144
3	8	11	2,048	19	524,288
4	16	12	4,096	20	1,048,576
5	32	13	8,192	21	2,097,152
6	64	14	16,384	22	4,194,304
7	128	15	32,768	23	8,388,608

Table: Numbers obtained from 2 to the power of n

In computer work,
- 2^{10} is referred to as K (kilo),
- 2^{20} as M (mega),
- 2^{30} as G (giga),
- 2^{40} as T (tera) and so on.

Octal Number System (Base-8 system)
Radix (r) = 8
Symbols = 0 through r-1 = 0 through 8-1 = {0, 1, 2…6, 7}
An octal numbers are expressed with a strings of symbols defined above, possibly, an *octal point* within it. The decimal equivalent of a octal number can be found by expanding the number into a power series with a base of 8.
Example: $(40712.56)_8$ can be interpreted using power series as:
$(40712.56)_8 = 4 \text{ X } 8^4 + 0 \text{ X } 8^3 + 7 \text{ X } 8^2 + 1 \text{ X } 8^1 + 2 \text{ X } 8^0 + 5 \text{ X } 8^{-1} + 6 \text{ X } 8^{-2} = (16842.1)_{10}$

Hexadecimal Number System (Base-16 system)
Radix (r) = 16
Symbols = 0 through r-1 = 0 through 16-1 = {0, 1, 2…9, A, B, C, D, E, F}
A hexadecimal numbers are expressed with a strings of symbols defined above, possibly, a *hexadecimal point* with in it. The decimal equivalent of a hexadecimal number can be found by expanding the number into a power series with a base of 16.
Example: $(4D71B.C6)_{16}$ can be interpreted using power series as:

$$\begin{aligned}(4D71B.C6)_{16} &= 4 \text{ X } 16^4 + D \text{ X } 16^3 + 7 \text{ X } 16^2 + 1 \text{ X } 16^1 + B \text{ X } 16^0 + C \text{ X } 16^{-1} + 6 \text{ X } 16^{-2} \\ &= 4 \text{ X } 16^4 + 13 \text{ X } 16^3 + 7 \text{ X } 16^2 + 1 \text{ X } 16^1 + 11 \text{ X } 16^0 + 12 \text{ X } 16^{-1} + 6 \text{ X } 16^{-2} \\ &= (317211.7734375)_{10}\end{aligned}$$

Number Base Conversions
Case I: *<u>Base-r system to Decimal</u>*: Base-r system can be binary (r=2), octal (r=8), hexadecimal (r=16), base-60 system or any other. For decimal system as destination of conversion, we just use power series explained above with varying *r* and sum the result according to the arithmetic rules of base-10 system. I have already done examples for binary to decimal, octal to decimal and hexadecimal to decimal.
For refreshment lets assume base-1000 number $(458HQY)_{1000.}$ Where n = 6 and m = 0.

$(458HQY)_{1000} = A_{n-1}\, r^{n-1} + A_{n-2}\, r^{n-2} + \ldots + A_1\, r^1 + A_0\, r^0 + A_{-1}\, r^{-1} + A_{-2}\, r^{-2} + \ldots + A_{-m+1}\, r^{-m+1} + A_{-m}\, r^{-m}$
$= 4 \text{ X } 1000^5 + 5 \text{ X } 1000^4 + 8 \text{ X } 1000^3 + H \text{ X } 1000^2 + Q \text{ X } 1000^1 + Y \text{ X } 1000^0$
=Resulting number will be in decimal. Here I have supposed various symbols for base-1000 system. Don't worry, if someone gives you base-1000 number for conversion, he should also define all 1000 symbols (0-999).

Case II: _Decimal to Base-r system_: Conversion follows following algorithm.

1. Separate the number into **integer** and **fraction** parts if radix point is given.
2. Divide "*Decimal Integer part*" by base *r* repeatedly until quotient becomes zero and storing remainders at each step.
3. Multiply "*Decimal Fraction part*" successively by r and accumulate the integer digits so obtained.
4. Combine both accumulated results and parenthesize the whole result with subscript *r*.

Example I: Decimal to binary

- $(41.6875)_{10} = (?)_2$
  Here Integer part = 41 and fractional part = 0.6875

```
Integer = 41                 Fraction = 0.6875

41 |                           0.6875
20 | 1                              2
10 | 0                         1.3750
 5 | 0                            x 2
 2 | 1                         0.7500
 1 | 0                            x 2
 0 | 1                         1.5000
                                  x 2
                               1.0000
```

$(41)_{10} = (101001)_2$ $(0.6875)_{10} = (0.1011)_2$

$(41.6875)_{10} = (101001.1011)_2$

Example II: Decimal to octal

- $(153.45)_{10} = (?)_8$

Here integer part = 153 and fractional part = 0.45

```
153 |                  0.45
 19 | 1                 X 8
  2 | 3          ---------------
  0 | 2                3.60
                        X 8
                 ---------------
                       4.80
                        X 8
                 ---------------
                       6.40   (may not end, choice
                               is upon you to end up)
```

This is simply division by 8, I am writing Quotients and remainders only.

Multiply always the portion after radix point.

$(153)_{10} = (231)_8$

$(0.45)_{10} = (346)_8$

$(153.45)_{10} = (231.346)_8$

Example III: Decimal to Hexadecimal

- $(1459.43)_{10} = (?)_{16}$
  Here integer part = 1459 and fractional part = 0.43

```
1459 |                         0.43
  91 | 4                        X16
   5 | 11 (=B)          ---------------
   0 | 5                       6.80
                                X16
                        ---------------
                              12.80
                                X16
                        ---------------
                              12.80   (Never ending...)
```

$(1459)_{10} = (5B4)_{16}$

$(0.43)_{10} = (6CC)_8$

$(1459.43)_{10} = (5B4.6CC)_{16}$

Case III: _Binary to octal & hexadecimal and vice-versa_: Conversion from and to binary, octal and hexadecimal representation plays an important part in digital computers. Since,

- $2^3 = 8$, octal digit can be represented by at least 3 binary digits. (We have discussed this much better in class). So to convert given binary number into its equivalent octal, we divide it into groups of 3 bits, give each group an octal symbol and combine the result.
  - _Integer part_: Group bits from right to left of an octal point. 0's can be added to makeit multiple of 3 (**not compulsory**).
  - _Fractional part_: Group bits from left to right of an octal point. 0's must be added to if bits are not multiple of 3 (**Note it**).
- 24 = 16, each hex digit corresponds to 4 bits. So to convert given binary number into its equivalent hex, we divide it into groups of 4 bits, give each group a hex digit and combine the result. If hex point is given, then process is similar as of octal.
- 15 numbers in 4 systems summarized below for easy reference.

Decimal (base 10)	Binary (base 2)	Octal (base 8)	Hexadecimal (base 16)
00	0000	00	0
01	0001	01	1
02	0010	02	2
03	0011	03	3
04	0100	04	4
05	0101	05	5
06	0110	06	6
07	0111	07	7
08	1000	10	8
09	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F

Example:

1. Binary to octal:
   $(10110001101011.11110000011)_2 = (\underbrace{010}_{2}\,\underbrace{110}_{6}\,\underbrace{001}_{1}\,\underbrace{101}_{5}\,\underbrace{011}_{3}.\,\underbrace{111}_{7}\,\underbrace{100}_{4}\,\underbrace{000}_{0}\,\underbrace{110}_{6})_2 = (26153.7406)_8$
2. Binary to hexadecimal:
   $(10110001101011.11110000011)_2 = (\underbrace{0010}_{2}\,\underbrace{1100}_{C}\,\underbrace{0110}_{6}\,\underbrace{1011}_{B}.\,\underbrace{1111}_{F}\,\underbrace{0000}_{0}\,\underbrace{0110}_{6})_2 = (2C6B.F06)_{16}$
3. From hex & octal to binary is quite easy, we just need to remember the binary of particular hex or octal digit.
   $(673.12)_8 = 110\ \ 111\ \ 011.\ 001\ \ 010 = (110111011.00101)_2$
   $(3A6.C)_{16} = 0011\ 1010\ 0110.\ 1100 = (1110100110.11)_2$

Complements

Complements are used in digital computers for simplifying the subtraction operation and for logical manipulation. There are two types of complements for each base-r system: r's complement and the second as the (r - 1)'s complement. When the value of the base *r* is substituted, the two types are referred to as the 2's complement and 1's complement for binary numbers, the 10's complement and 9's complement for decimal numbers etc

(r-1)'s Complement (diminished radix compl.)
(r-1)'s complement of a number N is defined as $(r^n - 1) - N$
Where **N** is the given number
r is the base of number system
n is the number of digits in the given number
To get the (r-1)'s complement fast, subtract each digit of a number from (r-1).

Example:

- 9's complement of 835_{10} is 164_{10} (Rule: $(10^n - 1) - N$)
- 1's complement of 1010_2 is 0101_2 (bit by bit complement operation)

r's Complement (radix complement)
r's complement of a number N is defined as $r^n - N$
Where **N** is the given number
r is the base of number system
n is the number of digits in the given number
To get the r's complement fast, add 1 to the low-order digit of its (r-1)'s complement.

Example:

- 10's complement of 835_{10} is $164_{10} + 1 = 165_{10}$
- 2's complement of 10102 is $0101_2 + 1 = 0110_2$

Subtraction with complements

The direct method of subtraction taught in elementary schools uses the borrow concept. When subtraction is implemented with digital hardware, this method is found to be less efficient than the method that uses complements.
The subtraction of two n-digit unsigned numbers *M* - *N* in base-*r* can be done as follows:

1. Add the minuend *M* to the *r's* complement of the subtrahend *N*. This performs $M + (r^n - N) = M - N + r''$.
2. If $M >= N$, the sum will produce an end carry, *r''*, which is discarded; what is left is the result *M* - *N*.
3. If $M < N$, the sum does not produce an end carry and is equal to $r^n - (N - M)$, which is the *r's* complement of $(N - M)$. To obtain the answer in a familiar form, take the r's complement of the sum and place a negative sign in front.

Example I:

Using 10's complement, subtract 72532 − 3250.

$$\begin{array}{rr}
M = & 72532 \\
\text{10's complement of } N = & +\ \underline{96750} \\
\text{Sum} = & 169282 \\
\text{Discard end carry } 10^5 = & -\underline{100000} \\
\text{Answer} = & 69282
\end{array}$$

M has 5 digits and *N* has only 4 digits. Both numbers must have the same number of digits; so we can write *N* as 03250. Taking the 10's complement of *N* produces a 9 in the most significant position. The occurrence of the end carry signifies that *M* >= *N* and the result is positive.

Example II:

Using 10's complement, subtract 3250 − 72532.

$$\begin{array}{rr}
M = & 03250 \\
\text{10's complement of } N = & +\ \underline{27468} \\
\text{Sum} = & 30718
\end{array}$$

There is no end carry.

Answer: −(10's complement of 30718) = −69282

Example III:

Given the two binary numbers X = 1010100 and Y = 1000011, perform the subtraction (a) $X - Y$ and (b) $Y - X$ using 2's complements.

(a)

$$\begin{array}{rr}
X = & 1010100 \\
\text{2's complement of } Y = & +\ \underline{0111101} \\
\text{Sum} = & 10010001 \\
\text{Discard end carry } 2^7 = & -\underline{10000000} \\
\text{Answer: } X - Y = & 0010001
\end{array}$$

(b)

$$\begin{array}{rr}
Y = & 1000011 \\
\text{2's complement of } X = & +\ \underline{0101100} \\
\text{Sum} = & 1101111
\end{array}$$

There is no end carry.

Answer: $Y - X$ = −(2's complement of 1101111) = −0010001 ■

Example IV: Repeating Example III using 1's complement

(a) $X - Y$ = 1010100 − 1000011

$$\begin{array}{rr}
X = & 1010100 \\
\text{1's complement of } Y = & +\ \underline{0111100} \\
\text{Sum} = & 10010000 \\
\text{End-around carry} & \longrightarrow \underline{+\ 1} \\
\text{Answer: } X - Y = & 0010001
\end{array}$$

(b) $Y - X$ = 1000011 − 1010100

$$\begin{array}{rr}
Y = & 1000011 \\
\text{1's complement of } X = & +\ \underline{0101011} \\
\text{Sum} = & 1101110
\end{array}$$

There is no end carry.

Answer: $Y - X$ = −(1's complement of 1101110) = −0010001

Binary Codes

Electronic digital systems use signals that have two distinct values and circuit elements that have two stable states. There is a direct analogy among binary signals, binary circuit elements, and binary digits. A binary number of *n* digits, for example, may be represented by *n* binary circuit elements, each having an output signal equivalent to a 0 or a 1. Digital systems represent and manipulate not only binary numbers, but also many other discrete elements of information. Any discrete element of information distinct among a group of quantities can be represented by a binary code. Binary codes play an important role in digital computers. The codes must be in binary because computers can only hold 1's and 0's.

Advantages of Binary Code

Following is the list of advantages that binary code offers.

- Binary codes are suitable for the computer applications.
- Binary codes are suitable for the digital communications.
- Binary codes make the analysis and designing of digital circuits if we use the binary codes.
- Since only 0 & 1 are being used, implementation becomes easy.

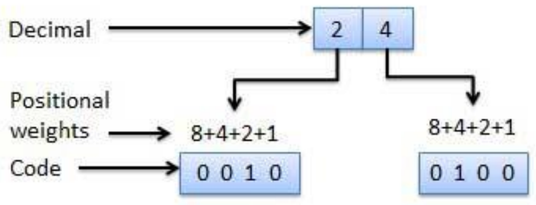
Classification of binary codes

The codes are broadly categorized into following four categories.

- Weighted Codes
- Non-Weighted Codes
- Binary Coded Decimal Code
- Alphanumeric Codes
- Error Detecting Codes
- Error Correcting Codes

Weighted Codes

Weighted binary codes are those binary codes which obey the positional weight principle. Each position of the number represents a specific weight. Several systems of the codes are used to express the decimal digits 0 through 9. In these codes each decimal digit is represented by a group of four bits.

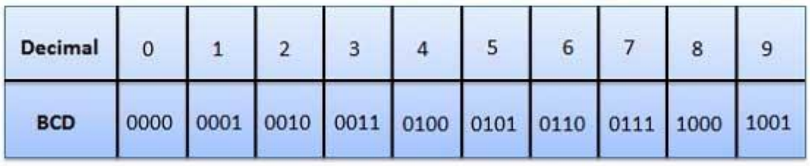


Non-Weighted Codes

In this type of binary codes, the positional weights are not assigned. The examples of non-weighted codes are Excess-3 code and Gray code.

1. Binary Coded Decimal (BCD)

The binary number system is the most natural system for a computer, but people are accustomed to the decimal system. So, to resolve this difference, computer uses decimals in coded form which the hardware understands. A binary code that distinguishes among 10 elements of decimal digits must contain at least four bits. Numerous different binary codes can be obtained by arranging four bits into 10 distinct combinations. The code most commonly used for the decimal digits is the straightforward binary assignment listed in the table below. This is called ***binary-coded decimal*** and is commonly referred to as **BCD**.



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

Table: 4-bit BCD code for decimal digits

- A number with *n* decimal digits will require 4n bits in BCD. E.g. decimal 396 is represented in BCD with 12 bits as 0011 1001 0110.
- Numbers greater than 9 has a representation different from its equivalent binary number, even though both contain 1's and 0's.
- Binary combinations 1010 through 1111 are not used and have no meaning in the BCD code.
- Example :
  $(185)_{10} = (0001\ 1000\ 0101)_{BCD} = (10111001)_2$

Advantages of BCD Codes

- It is very similar to decimal system.
- We need to remember binary equivalent of decimal numbers 0 to 9 only.

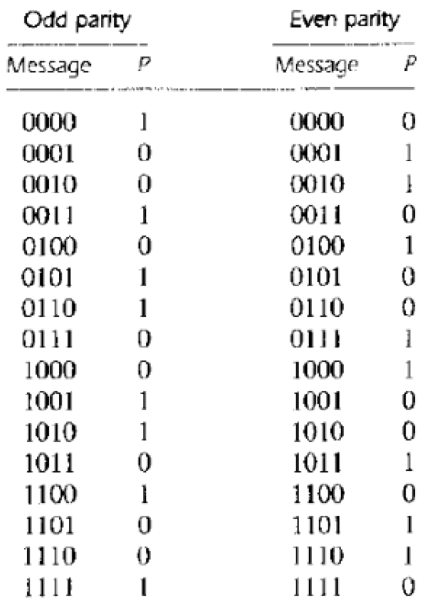
Disadvantages of BCD Codes

- The addition and subtraction of BCD have different rules.
- The BCD arithmetic is little more complicated.
- BCD needs more number of bits than binary to represent the decimal number. So BCD is less efficient than binary.

2. Error-Detection codes

Electric wires or other communication medium can transmit binary information from one location to another. Any external noise introduced into the physical communication medium may change some of the bits from 0 to 1 or vice versa.

The purpose of an error-detection code is to detect such bit-reversal errors. One of the most common ways to achieve error detection is by means of a **parity bit**. A *parity bit* is the extra bit included to make the total number of 1's in the resulting code word either even or odd. A message of 4-bits and a parity bit P are shown in the table below:



Odd parity		Even parity	
Message	P	Message	P
0000	1	0000	0
0001	0	0001	1
0010	0	0010	1
0011	1	0011	0
0100	0	0100	1
0101	1	0101	0
0110	1	0110	0
0111	0	0111	1
1000	0	1000	1
1001	1	1001	0
1010	1	1010	0
1011	0	1011	1
1100	1	1100	0
1101	0	1101	1
1110	0	1110	1
1111	1	1111	0

Error Checking Mechanism:

- During the transmission of information from one location to another, an even parity bit is generated in the sending end for each message transmission. The message, together with the parity bit, is transmitted to its destination. The parity of the received data is checked in the receiving end. If the parity of the received information is not even, it means that at least one bit has changed value during the transmission.
- This method detects one, three, or any odd combination of errors in each message that is transmitted. An even combination of errors is undetected. Additional error-detection schemes may be needed to take care of an even combination of errors.

3. Gray code (Reflected code)

It is a binary coding scheme used to represent digits generated from a mechanical sensor that may be prone to error. Used in telegraphy in the late 1800s, and also known as "reflected binary code".

Bell Labs researcher Frank Gray patented Gray code in 1947. In Gray code, there is **only one bit location different between two successive values**, which makes mechanical transitions from one digit to the next less error prone. The following chart shows normal binary representations from 0 to 15 and the corresponding Gray code.

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

Application of Gray code

- Gray code is popularly used in the shaft position encoders.
- A shaft position encoder produces a code word which represents the angular position of the shaft.

The Gray code is used in applications where the normal sequence of binary numbers may produce an error or ambiguity during the transition from one number to the next. If binary numbers are used, a change from 0111 to 1000 may produce an intermediate erroneous number 1001 if the rightmost bit takes more time to change than the other three bits. The Gray code eliminates this problem since only one bit changes in value during any transition between two numbers.

4. Alphanumeric codes

Alphanumeric character set is a set of elements that includes the 10 decimal digits, 26 letters of the alphabet and special characters such as $, %, + etc. It is necessary to formulate a binary code for this set to handle different data types. If only capital letters are included, we need a binary code of at least six bits, and if both uppercase letters and lowercase letters are included, we need a binary code of at least seven bits.

The following three alphanumeric codes are very commonly used for the data representation.

- American Standard Code for Information Interchange (ASCII).
- Extended Binary Coded Decimal Interchange Code (EBCDIC).
- Five bit Baudot Code.

A. ASCII character code

The standard binary code for the alphanumeric characters is called ASCII (American Standard Code for Information Interchange). It uses seven bits to code 128 characters as shown in the table below. The seven bits of the code are designated by B_1 through B_7 with B_7 being the most significant bit.

American Standard Code for Information Interchange (ASCII)

	$B_7B_6B_5$							
$B_4B_3B_2B_1$	000	001	010	011	100	101	110	111
0000	NULL	DLE	SP	0	@	P	`	p
0001	SOH	DC1	!	1	A	Q	a	q
0010	STX	DC2	"	2	B	R	b	r
0011	ETX	DC3	#	3	C	S	c	s
0100	EOT	DC4	$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	'	7	G	W	g	w
1000	BS	CAN	(	8	H	X	h	x
1001	HT	EM	)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	;	K	[	k	{
1100	FF	FS	,	<	L	\	l	|
1101	CR	GS	-	=	M	]	m	}
1110	SO	RS	.	>	N	^	n	~
1111	SI	US	/	?	O	_	o	DEL

NOTE:

Decimal digits in ASCII can be converted to BCD by removing the three higher order bits, 011.

Various control character symbolic notation stands for:

NULL	NULL	DLE	Data link escape
SOH	Start of heading	DC1	Device control 1
STX	Start of text	DC2	Device control 2
ETX	End of text	DC3	Device control 3
EOT	End of transmission	DC4	Device control 4
ENQ	Enquiry	NAK	Negative acknowledge
ACK	Acknowledge	SYN	Synchronous idle
BEL	Bell	ETB	End of transmission block
BS	Backspace	CAN	Cancel
HT	Horizontal tab	EM	End of medium
LF	Line feed	SUB	Substitute
VT	Vertical tab	ESC	Escape
FF	Form feed	FS	File separator
CR	Carriage return	GS	Group separator
SO	Shift out	RS	Record separator
SI	Shift in	US	Unit separator
SP	Space	DEL	Delete

Example: ASCII for each symbol is ($B_7B_6B_5B_4B_3B_2B_1$)

G←100 0111, (←010 1000, h ←110 1000, >← 011 1110 and so on.

B. BCDIC character code

EBCDIC (Extended Binary Coded Decimal Interchange Code) is another alphanumeric code used in IBM equipment. It uses **eight bits** for each character. EBCDIC has the same character symbols as ASCII, but the bit assignment for characters is different. As the name implies, the binary code for the letters and numerals is an extension of the binary-coded decimal (BCD) code. This means that the last four bits of the code range from 0000 through 1001 as in BCD.

Integrated Circuits (ICs)

An Integrated circuit is an association (or connection) of various electronic devices such as resistors, capacitors and transistors etched (or fabricated) to a semiconductor material such as silicon or germanium. It is also called as a **chip** or **microchip**. An IC can function as an amplifier, rectifier, oscillator, counter, timer and memory. Sometime ICs are connected to various other systems to perform complex functions.

Types of ICs

ICs can be categorized into two types

- Analog or Linear ICs

- Digital or logic ICs

Further there are certain ICs which can perform as a combination of both analog and digital functions.

Analog or Linear ICs: They produce continuous output depending on the input signal. From the name of the IC we can deduce that the output is a linear function of the input signal. Op-amp (operational amplifier) is one of the types of linear ICs which are used in amplifiers, timers and counters, oscillators etc.

Digital or Logic ICs: Unlike Analog ICs, Digital ICs never give a continuous output signal. Instead it operates only during defined states. Digital ICs are used mostly in microprocessor and various memory applications. Logic gates are the building blocks of Digital ICs which operate either at 0 or 1.

Advantages of ICs

- In consumer electronics, ICs have made possible the development of many new products, including personal calculators and computers, digital watches, and video games.
- They have also been used to improve or lower the cost of many existing products, such as appliances, televisions, radios, and high-fidelity equipment.
- The logic and arithmetic functions of a small computer can now be performed on a single VLSI chip called a microprocessor.
- Complete logic, arithmetic, and memory functions of a small computer can be packaged on a single printed circuit board, or even on a single chip.

Levels of Integration

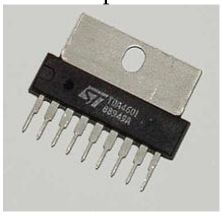
During 1959 two different scientists invented IC's. Jack Kilby from Texas Instruments made his first germanium IC during 1959 and Robert Noyce made his first silicon IC during the same year. But ICs were not the same since the day of their invention; they have evolved a long way. Integrated circuits are often classified by the number of transistors and other electronic components they contain:

- SSI (small-scale integration): Up to 100 electronic components per chip
- MSI (medium-scale integration): From 100 to 3,000 electronic components per chip
- LSI (large-scale integration): From 3,000 to 100,000 electronic components per chip
- VLSI (very large-scale integration): From 100,000 to 1,000,000 electronic components per chip
- ULSI (ultra large-scale integration): More than 1 million electronic components per chip

SIP (Single In-line Package) and DIP (Dual In-line Package)

SIP

A **single in-line package** is an electronic device package which has one row of connecting pins. It is not as popular as the dual in-line package (DIP) which contains two rows of pins, but has been used for packaging RAM chips and multiple resistors with a common pin.



SIPs group RAM chips together on a small board. The board itself has a single row of pin-leads that resembles a comb extending from its bottom edge, which plug into a special socket on a system or system-expansion board. SIPs are commonly found in memory modules. SIP is not to be confused with SIPP which is an archaic term referring to Single In-line Pin Package which was a memory used in early computers.

DIP

Dual in-line package (DIP) is a type of semiconductor component packaging. DIPs can be installed either in sockets or permanently soldered into holes extending into the surface of the printed circuit board. DIP is relatively broadly defined as any rectangular package with two uniformly spaced parallel rows of pins pointing downward, whether it contains an IC chip or some other device(s), and whether the pins emerge from the sides of the package and bend downwards. A DIP is usually referred to as a **DIP*n***, where *n* is the total number of pins.

For example, a microcircuit package with two rows of seven vertical leads would be a DIP14. The photograph below shows three DIP14 ICs.

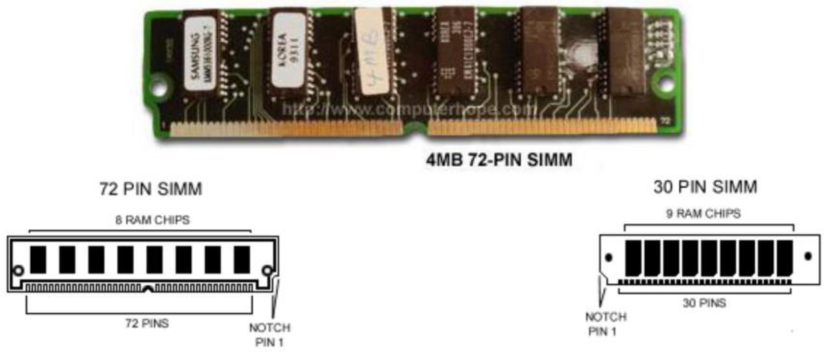
Several DIP variants for ICs exist, mostly distinguished by packaging material:

- **Ceramic Dual In-line Package (CERDIP or CDIP)**
- **Plastic Dual In-line Package (PDIP)**
- **Shrink Plastic Dual In-line Package (SPDIP)** -A denser version of the PDIP with a 0.07 in. (1.778 mm) lead pitch.
- **Skinny Dual In-line Package (SDIP)** – Sometimes used to refer to a 0.3 in. wide DIP, normally when clarification is needed e.g. for a 24 or 28 pin DIP.

Fig: Several PDIPs and CERDIPs. The large CERDIP in the foreground is an Intel 8080 microprocessor.

SIMM (Single In-line Memory Module) and DIMM (Dual In-line Memory Module)
These two terms (SIMM and DIMM) refer to a way series of dynamic random access memory integrated circuits modules are mounted on a printed circuit board and designed for use in personal computers, workstations and servers.

SIMM
Short for **Single In-line Memory Module, SIMM** is a circuit board that holds six to nine memory chips per board, the ninth chip usually an error checking chip (parity/non parity) and were commonly used with Intel Pentium or Pentium compatible motherboards. SIMMs are rarely used today and have been widely replaced by DIMMs. SIMMs are available in two flavors: 30 pin and 72 pin. 30-pin SIMMs are the older standard, and were popular on third and fourth generation motherboards. 72-pin SIMMs are used on fourth, fifth and sixth generation PCs.



DIMM
Short for **Dual In-line Memory Module**, **DIMM** is a circuit board that holds memory chips. DIMMs have a 64-bit path because of the Pentium Processor requirements. Because of the new bit path, DIMMs can be installed one at a time, unlike SIMMs on a Pentium that would require two to be added. Below is an example image of a 512MB DIMM memory stick.

SO-DIMM is short for **Small Outline DIMM** and is available as a 72-pin and 144-pin configuration. SO-DIMMs are commonly utilized in laptop computers.

Some of the advantages DIMMs have over SIMMs:

- DIMMs have separate contacts on each side of the board, thereby providing twice as much data as a single SIMM.
- The command address and control signals are buffered on the DIMMs. With heavy memory requirements this will reduce the loading effort of the memory.

Description of a few types of chips (not in syllabus)

CMOS: in computer science, acronym for complementary metal-oxide semiconductor. A semiconductor device that consists of two metal-oxide semiconductor field effect transistors (MOSFETs), one N-type and one P-type, integrated on a single silicon chip. Generally used for RAM and switching applications, these devices have very high speed and extremely low power consumption. They are, however, easily damaged by static electricity.
Digital Signal Processor (DSP): An integrated circuit designed for high-speed data manipulations, used in audio, communications, image manipulation, and other data-acquisition and data-control applications.

Dynamic RAM (DRAM): In computer science, a form of semiconductor random access memory (RAM). Dynamic RAMs store information in integrated circuits that contain capacitors. Because capacitors lose their charge over time, dynamic RAM boards must include logic to "refresh" (recharge) the RAM chips continuously. While a dynamic RAM is being refreshed, it cannot be read by the processor; if the processor must read the RAM while it is being refreshed, one or more wait states occur. Because their internal circuitry is simple, dynamic RAMs are more commonly used than static RAMs, even though they are slower. A dynamic RAM can hold approximately four times as much data as a static RAM chip of the same complexity.

EPROM: In computer science, acronym for erasable programmable read-only memory, also called reprogrammable read-only memory (RPROM). EPROMs are nonvolatile memory chips that are programmed after they are manufactured. EPROMs are a good way for hardware vendors to put variable or constantly changing code into a prototype system when the cost of producing many PROM chips would be prohibitive. EPROMs differ from PROMs in that they can be erased, generally by removing a protective cover from the top of the chip package and exposing the semiconductor material to ultraviolet light, and can be reprogrammed after having been erased. Although EPROMs are more expensive than PROMs, they can be more cost-effective in the long run if many changes are needed.

PROM: Acronym for programmable read-only memory. In computer science, a type of read-only memory (ROM) that allows data to be written into the device with hardware called a PROM programmer. After a PROM has been programmed, it is dedicated to that data, and it cannot be reprogrammed. Because ROMs are cost-effective only when produced in large volumes, PROMs are used during the prototyping stage of the design. New PROMs can be created and discarded as needed until the design is perfected.

Reduced Instruction Set Computer (RISC): Type of microprocessor that focuses on rapid and efficient processing of a relatively small set of instructions. RISC design is based on the premise that most of the instructions a computer decodes and executes are simple. As a result, RISC architecture limits the number of instructions that are built into the microprocessor but optimizes each so it can be carried out very rapidly-usually within a single clock cycle. RISC chips thus execute simple instructions faster than microprocessors designed to handle a much wider array of instructions.

ROM: Acronym for read-only memory. In computer science, semiconductor-based memory that contains instructions or data that can be read but not modified. To create a ROM chip, the designer supplies a semiconductor manufacturer with the instructions or data to be stored; the manufacturer then produces one or more chips containing those instructions or data. Because creating ROM chips involves a manufacturing process, it is economically viable only if the ROM chips are produced in large quantities; experimental designs or small volumes are best handled using PROM or EPROM. In general usage, the term ROM often means any read-only device, including PROM and EPROM.

Static RAM (SRAM): In computer science, a form of semiconductor memory (RAM). Static RAM storage is based on the logic circuit known as a flip-flop, which retains the information stored in it as long as there is enough power to run the device. A static RAM chip can store only about one-fourth as much data as a dynamic RAM chip of the same complexity, but static RAM does not require refreshing and is usually much faster than dynamic RAM. It is also more expensive. Static RAMs are usually reserved for use in caches.

Unit 2
Boolean algebra and Logic Gates

Binary logic

Binary logic consists of binary variables and logical operations. The variables are designated by letters of the alphabet such as *A*, *B*, C, *x*, *y*, Z, etc., with each variable having two and only two distinct possible values: 1 and 0. There are three basic logical operations: AND, OR, and NOT.

1. AND: This operation is represented by a dot or by the absence of an operator. For example, *x.y* = z or *xy* = z is read *"x* AND *y* is equal to *z."* The logical operation AND is interpreted to mean that z = 1 if and only if *x* = 1 *and y* = 1; otherwise z = 0. (Remember that *x*, *y*, and z are binary variables and can be equal either to 1 or 0, and nothing else.)
2. OR: This operation is represented by a plus sign. For example, *x* + *y* = z is read *"x* OR *y* is equal to *z,"* meaning that z = 0 if *x* = 0 *or* if *y* = 0 otherwise z = 1.
3. NOT: This operation is represented by a prime (sometimes by a bar). For example, *x'* = z is read "not *x* is equal to *z,"* meaning that z is what *x* is not. In other words, if *x* = 1, then z = 0; but if *x* = 0, then z = 1.

These definitions may be listed in a compact form using truth tables. A **truth table** is a table of all possible combinations of the variables showing the relation between the values that the variables may take and the result of the operation.

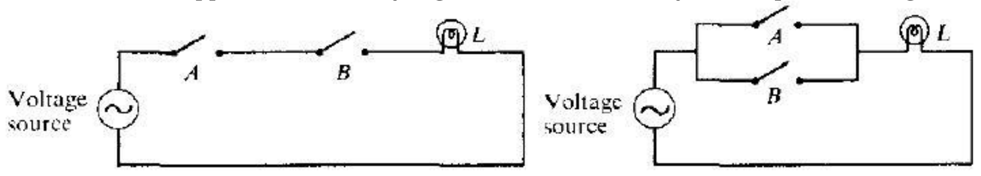
Truth Tables of Logical Operations

AND			OR			NOT	
x	y	$x \cdot y$	x	y	$x + y$	x	x'
0	0	0	0	0	0	0	1
0	1	0	0	1	1	1	0
1	0	0	1	0	1		
1	1	1	1	1	1		

Binary logic should not be confused with binary arithmetic (However we use same symbols here). You should realize that an arithmetic variable designates a number that may consist of many digits. A logic variable is always either 1 or 0. For example, in binary arithmetic, 1 + 1 = 10 (read: "one plus one is equal to 2"), whereas in binary logic, we have 1 + 1 = 1 (read: "one OR one is equal to one").

Switching circuits and Binary Signals

The use of binary variables and the application of binary logic are demonstrated by the simple switching circuits shown below:



(a)Switches in series-Logical AND (b) Switches in parallel-Logical OR

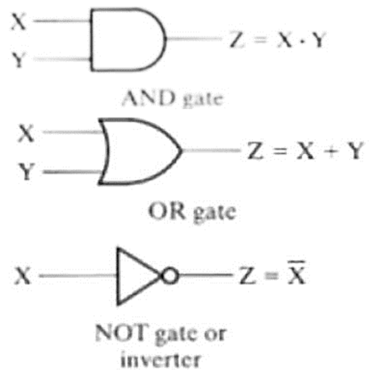
Let the manual switches *A* and *B* represent two binary variables with values equal to 0 when the switch is open and 1 when the switch is closed. Similarly, let the lamp L represent a third binary variable equal to 1 when the light is on and 0 when off.

Electronic digital circuits are sometimes called **switching circuits** because they behave like a switch, with the active element such as a transistor either conducting (switch closed) or not conducting (switch open). Instead of changing the switch manually, an electronic switching circuit uses **binary signals** to control the conduction or non-conduction state of the transistor.

Basic Logic Gates (Digital logic gates will be covered in detail later)

Logic gates are electronic circuits that operate on one or more input signals to produce an output signal. Electrical signals such as voltages or currents exist throughout a digital system in either one of two recognizable values (bi-state 0 or 1). Voltage-operated circuits respond to two separate voltage ranges (Example of voltage ranges is discussed in unit 1) that represent a binary variable equal to logic 1 or logic

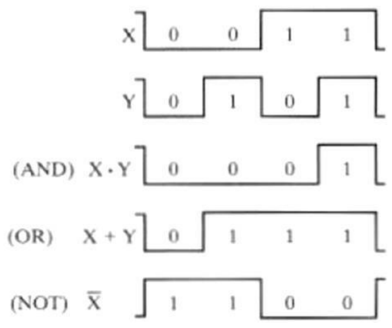
The graphics symbols used to designate the three types of gates AND, OR, and NOT are shown in Figure below:



Graphic symbols

- These circuits, called gates, are blocks of hardware that produce a logic-1 or logic-0 output signal if input logic requirements are satisfied.
- Note that four different names have been used for the same type of circuits: digital circuits, switching circuits, logic circuits, and gates.

- AND and OR gates may have more than two inputs.
- NOT gate is single input circuit, it simply inverts the input.



Timing diagram
The two input signals X and Y to the AND and OR gates take on one of four possible combinations: 00, 01, 10, or 11. These input signals are shown as timing diagrams, together with the timing diagrams for the corresponding output signal for each type of gate. The horizontal axis of a timing diagram represents time, and the vertical axis shows a signal as it changes between the two possible voltage levels. The low level represents logic 0 and the high level represents logic I. 1l1e AND gate responds with a logic-1 output signal when both input signals are logic-1. The OR gate responds with a logic-1 output signal if either input signal is logic-1

Boolean algebra
Boolean Algebra is used to analyze and simplify the digital (logic) circuits. It uses only the binary numbers i.e. 0 and 1. It is also called as Binary Algebra or logical Algebra. Boolean algebra was invented by George Boole in 1854.

Rule in Boolean Algebra
Following are the important rules used in Boolean algebra.
Variable used can have only two values. Binary 1 for HIGH and Binary 0 for LOW.

- Complement of a variable is represented by an overbar (-). Thus, complement of variable B is represented as $\overline{B}$. Thus if B = 0 then $\overline{B}$ = 1 and B = 1 then $\overline{B}$ = 0.
- ORing of the variables is represented by a plus (+) sign between them. For example ORing of A, B, C is represented as A + B + C.
- Logical ANDing of the two or more variable is represented by writing a dot between them such as A.B.C. Sometime the dot may be omitted like ABC.

Boolean Laws
There are six types of Boolean Laws.

Commutative law
Any binary operation which satisfies the following expression is referred to as commutative operation.
(i) $A.B = B. A$ (ii) $A + B = B + A$
Commutative law states that changing the sequence of the variables does not have any effect on the output of a logic circuit.

Associative law
This law states that the order in which the logic operations are performed is irrelevant as their effect is the same.
(i) $(A.B).C = A.(B.C)$ (ii) $(A + B) + C = A + (B + C)$

Distributive law
Distributive law states the following condition.
$A.(B + C) = A.B + A.C$

AND law
These laws use the AND operation. Therefore they are called as AND laws.
(i) $A.0 = 0$ (ii) $A.1 = A$
(iii) $A.A = A$ (iv) $A.\overline{A} = 0$

OR law
These laws use the OR operation. Therefore they are called as OR laws.
(i) $A + 0 = A$ (ii) $A + 1 = 1$
(iii) $A + A = A$ (iv) $A + \overline{A} = 1$

INVERSION law
This law uses the NOT operation. The inversion law states that double inversion of a variable results in the original variable itself.

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

In 1854 George Boole introduced a systematic treatment of logic and developed for this purpose an algebraic system now called *Boolean algebra.* In 1938 C. E. Shannon introduced a two-valued Boolean algebra called *switching algebra,* in which he demonstrated that the properties of bitable electrical switching circuits can be represented by this algebra.
Thus, the mathematical system of binary logic is known as **Boolean or switching algebra**. This algebra is conveniently used to describe the operation of complex networks of digital circuits. Designers of digital systems use Boolean algebra to transform circuit diagrams to algebraic expressions and vice versa. For any given algebra system, there are some initial assumptions, or postulates, that the system follows. We can deduce additional rules, theorems, and other properties of the system from this basic set of postulates. Boolean algebra systems often employ the postulates formulated by E. V. Huntington in 1904.

Postulates
Boolean algebra is an algebraic structure defined on a set of elements *B (Boolean system)* together with two binary operators + (OR) and • (AND) and unary operator ' (NOT), provided the following postulates are satisfied:

- P1→ Closure: Boolean algebra is closed under the AND, OR, and NOT operations.
- P2→ Commutatively: The • and + operators are commutative i.e. x + y = y + x and x • y = y • x, for all x, y ∈ B.
- P3→ Distribution: • and + are distributive with respect to one another i.e.
  X • (y + z) = (x • y) + (x • z).
  x + (y • z) = (x + y) • (x + z), for all x, y, z ∈ B.
- P4→ Identity: The identity element with respect to • is 1 and + is 0 i.e. x + 0 = 0 + x = x and x • 1=1• x = x. There is no identity element with respect to logical NOT.
- P5→ Inverse: For every value x there exists a value x' such that x • x' = 0 and x + x' = 1. This value is the logical complement (or NOT) of x.
- P6→ There exists at least two elements x, y ∈ B such that x ≠ y.

One can formulate many Boolean algebras (viz. set theory, n-bit vectors algebra), depending on the choice of elements of B and the rules of operation. Here, we deal only with a two-valued Boolean algebra, i.e., B = {0, 1}. Two-valued Boolean algebra has applications in set theory and in propositional logic. Our interest here is with the application of Boolean algebra to gate-type circuits.

Basic theorems and Properties of Boolean algebra
Duality
Postulates of Boolean algebra are found in pairs; one part may be obtained from the other if the binary operators and the identity elements are interchanged. This important property of Boolean algebra is called the **duality principle**. It states that "***Every algebraic expression deducible from the postulates of Boolean algebra remains valid if the operators and identity elements are interchanged***". In a two-valued Boolean algebra, the identity elements and the elements of the set *B* are the same: 1 and 0. If the ***dual*** of an algebraic expression is desired, we simply interchange OR and AND operators and replace 1's by 0's and 0's by 1's.

Basic theorems
The theorems, like the postulates, are listed in pairs; each relation is the dual of the one paired with it. The postulates are basic axioms of the algebraic structure and need no proof. The theorems must be proven from the postulates. six theorems of Boolean algebra are given below:

Theorem1:	Idempotence	(a) x + x = x	(b) x.x = x	One variable theorems
Theorem2:	Existence: 0&1	(a) x + 1 = 1	(b) x.0 = 0	
Theorem3:	Involution	(x')' = x		
Theorem4:	Associative	(a) x + (y + z) = (x + y) + z	(b) x(yz) = (xy)z	2 or 3 variable theorems
Theorem5:	Demorgan	(a) (x + y)' = x'y'	(b) (xy)' = x' + y'	
Theorem6:	Absorption	(a) x + xy = x	(b) x(x + y) = x	

Proofs:

(A) The proofs of the theorems with one variable are presented below:

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

$$\begin{aligned} x + x &= (x + x) . 1 && \text{(P4: Identity element)} \\ &= (x + x)(x + x') && \text{(P5: Existence of inverse)} \\ &= x + xx' && \text{(P3: Distribution)} \\ &= x + 0 && \text{(P5: Existence of inverse)} \\ &= x && \text{(P4: Identity element)} \end{aligned}$$

THEOREM 1(b): $x{\cdot}x = x$

$$\begin{aligned} x.x &= xx + 0 && \text{(P4: Identity element)} \\ &= xx + xx' && \text{(P5: Existence of inverse)} \\ &= x(x + x') && \text{(P3: Distribution)} \\ &= x.1 && \text{(P5: Existence of inverse)} \\ &= x && \text{(P4: Identity element)} \end{aligned}$$

Each step in theorem 1(b) and 1(a) are dual of each other.

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

$$\begin{aligned} x + 1 &= 1{\cdot} (x + 1) && \text{(P4: Identity element)} \\ &= (x + x')(x + 1) && \text{(P5: Existence of inverse)} \\ &= x + x'{\cdot}1 && \text{(P3: Distribution)} \\ &= x + x' && \text{(P4: Identity element)} \\ &= 1 && \text{(P5: Existence of inverse)} \end{aligned}$$

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

THEOREM 3: $(x')' = x$. From P5, we have $x + x' = 1$ and $x.x' = 0$, which defines the complement of x. The complement of x' is x and is also $(x')'$. Therefore, since the complement is unique, we have that $(x')' = x$.

(B) The theorems involving two or three variables may be proven algebraically from the postulates and the theorems that have already been proven. For example, lets prove **Demorgan's theorem:**

THEOREM 5(a): $(x + y)' = x'y'$

From postulate P5 (Existence of inverse), for every x in a Boolean algebra there is a unique x' such that

$$x + x' = 1 \text{ and } x \bullet x' = 0$$

So it is sufficient to show that x'y' is the complement of x + y.

We'll do this by showing that $(x + y) + (x'y') = 1$ and $(x + y) \bullet (x'y') = 0$.

$$\begin{aligned} (x + y) + (x'y') &= [(x + y) + x'] [(x + y) + y'] && \text{[OR distributes over AND (P3)]} \\ &= [(y + x) + x'] [(x + y) + y'] && \text{[OR is commutative (P2)]} \\ &= [y + (x + x')] [x + (y + y')] && \text{[OR is associative (Theorem 3(a)), used twice]} \\ &= (y + 1)(x + 1) && \text{[Complement, x + x' = 1 (P5), twice]} \\ &= 1 \bullet 1 && \text{[x + 1 = 1, (Theorem 2), twice]} \\ &= 1 && \text{[Idempotent, x • x = x (Theorem 1)]} \end{aligned}$$

Also,

$$\begin{aligned} (x + y)(x'y') &= (x'y') (x + y) && \text{[AND is commutative (P2)]} \\ &= [(x'y') x] + [(x'y') y] && \text{[AND distributes over OR (P3)]} \\ &= [(y'x')x] + [(x'y')y] && \text{[AND is commutative (P2)]} \\ &= [y'(x'x)] + [x'(y'y)] && \text{[AND is associative (Theorem 3(b)), twice]} \\ &= [y'(xx')] + [x'(yy')] && \text{[AND is commutative, twice]} \\ &= *y' \bullet 0+ + *x' \bullet 0+ && \text{[Complement, x • x' = 0, twice]} \\ &= 0 + 0 && \text{[x • 0 = 0, twice]} \\ &= 0 && \text{[Idempotent, x + x = x]} \end{aligned}$$

THEOREM 5(b): (xy)' = x' + y' can be proved similarly as in Theorem 5(a). Each step in the proof of 5(b) is the dual of its 5(a) counterparts.

Theorems above can also be proved using truth tables (alternative to algebraic simplification). Viz.
theorem 6(a) can be proved as:

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

Operator Precedence
The operator precedence for evaluating Boolean expressions is
1. Parentheses→ ()
2. NOT→'
3. AND→.
4. OR→+

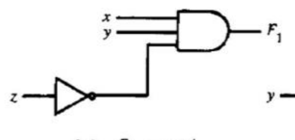
In other words, the expression inside the parentheses must be evaluated before all other operations. The next operation that holds precedence is the complement, then follows the AND, and finally the OR. Example: (a+b.c).d' → here we first evaluate 'b.c' and OR it with 'a' followed by ANDing with complement of 'd'.

Boolean Functions
A Boolean function is an expression formed with binary variables (variables that takes the value of 0 or 1), the two binary operators OR and AND, and unary operator NOT, parentheses, and an equal sign. For given value of the variables, the function can be either 0 or 1.

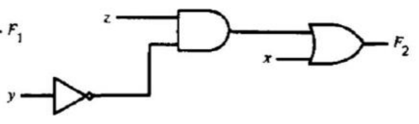
- **Boolean function represented as an algebraic expression:** Consider Boolean function F_1 = xyz'.
  Function F is equal to 1 if x=1, y=1 and z=0; otherwise F_1 =0. Other examples are: F_2 = x + y'z, F_3 = x'y'z + x'yz + xy', F_4 = xy' + x'z etc.
- **Boolean function represented in a truth table:**
  The number of rows in the truth table is 2^n, where n is the number of binary variables in the function, The 1's and 0's combinations for each row is easily obtained from the binary numbers by counting from 0 to $2^n – 1$.

x	y	z	F_1	F_2	F_3	F_4
0	0	0	0	0	0	0
0	0	1	0	1	1	1
0	1	0	0	0	0	0
0	1	1	0	0	1	1
1	0	0	0	1	1	1
1	0	1	0	1	1	1
1	1	0	1	1	0	0
1	1	1	0	1	0	0

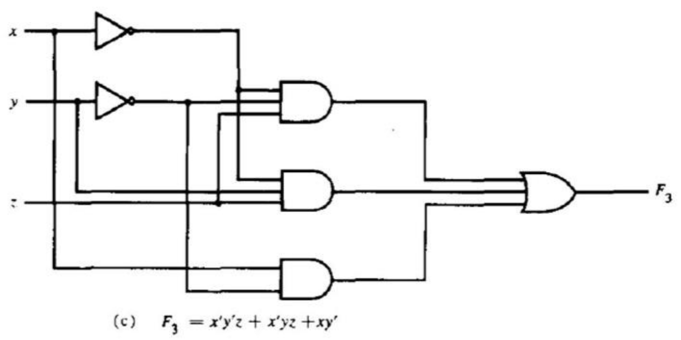
Question: Is it possible to find two algebraic expressions that specify the same function? Answer is: **yes**. Being straightforward, the manipulation of Boolean algebra is applied mostly to the problem of finding simpler expressions for the same function.
Example: Functions F_3 and F_4 are same although they have different combinations of binary variables with in them.
A Boolean function may be transformed from an algebraic expression into a logic diagram composed of AND, OR, and NOT gates. The implementation of the four functions introduced in the previous discussion is shown below:



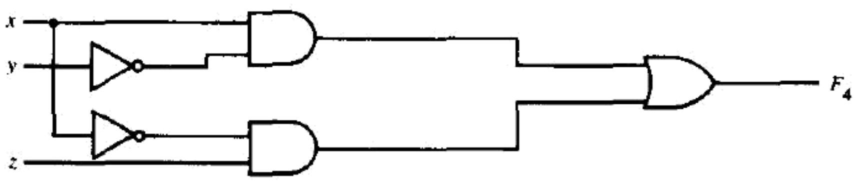
(a) $F_1 = xyz'$



(b) $F_2 = x + y'z$



(c) $F_3 = x'y'z + x'yz + xy'$



(d) $F_4 = xy' + x'z$

Fig: Implementation of Boolean functions with gates

Algebraic manipulation and simplification of Boolean function

- A ***literal*** is a primed or unprimed (i.e. complemented or un-complemented) variable. When a Boolean function is implemented with logic gates, each literal in the function designates an input to a gate, and each ***term*** is implemented with a gate.
- The minimization of the number of literals and the number of terms results in a circuit with less equipment. It is not always possible to minimize both simultaneously; usually, further criteria must be available. At the moment, we shall narrow the minimization criterion to literal minimization. We shall discuss other criteria in unit 3.
- The number of literals in a Boolean function can be minimized by algebraic manipulations. Unfortunately, there are no specific rules to follow that will guarantee the final answer. The only method available is a cut-and-try procedure employing the postulates, the basic theorems, and any other manipulation method that becomes familiar with use. The following examples illustrate this procedure.

Simplify the following Boolean functions to a minimum number of literals.

1. $x + x'y = (x + x')(x + y) = 1.(x + y) = x + y$
2. $x(x' + y) = xx' + xy = 0 + xy = xy$
3. $x'y'z + x'yz + xy' = x'z(y' + y) + xy = x'z + xy$
4. $xy + x'z + yz = xy + x'z + yz(x + x')$
   $= xy + x'z + xyz + x'yz$
   $= xy(1 + z) + x'z(1 + y)$
   $= xy + x'z$
5. $(x + y)(x' + z)(y + z) = (x + y)(x' + z)$ by duality from function 4.

Complement of a function

The complement of a function *F* is *F'* and is obtained from an interchange of 0's for 1's and 1's for 0's in the value of *F*. The complement of a function may be derived algebraically through DeMorgan's theorem. DeMorgan's theorems can be extended to three or more variables. The three-variable form of the first DeMorgan's theorem is derived below.

$(A + B + C)'$	$= (A + X)'$	let $B + C = X$
	$= A'X'$	(DeMorgan)
	$= A' \cdot (B + C)'$	substitute $B + C = X$
	$= A'.(B'C')$	(DeMorgan)
	$= A'B'C'$	(associative)

DeMorgan's theorems for any number of variables resemble in form the two variable case and can be derived by successive substitutions similar to the method used in the above derivation. These theorems can be generalized as follows:

$$(A + B + C + D + ... + F)' = A'B'C'D'\ldots F'$$
$$(ABCD ... F)' = A' + B' + C' + D' + ... + F'$$

The generalized form of De Morgan's theorem states that the complement of a function is obtained by interchanging AND and OR operators and complementing each literal.

Two ways of getting complement of a Boolean function:

1. *Applying DeMorgan's theorem:*

Question: Find the complement of the functions $F_1 = x'yz' + x'y'z$ *and* $F_2 = x(y'z' + yz)$.

By applying DeMorgan's theorem as many times as necessary, the complements are obtained as follows:

$F_1' = (x'yz' + x'y'z)' = (x'yz')'(x'y'z)' = (x + y' + z)(x + y + z')$

$F_2' = [x(y'z' + yz)]' = x' + (y'z' + yz)' = x' + (y'z')' \cdot (yz)' = x' + (y + z)(y' + z')$

2. *First finding dual of the algebraic expression and complementing each literal*

Question: Find the complement of the functions F_1 and F_2 of example above by taking their duals and complementing each literal.

- $F_1 = x'yz' + x'y'z$.
  The dual of F_1 is $(x' + y + z')(x' + y' + z)$.
  Complement each literal: $(x + y' + z)(x + y + z') = F_1'$.
- $F_2 = x(y'z' + yz)$.
  The dual of F_1 is $x + (y' + z')(y + z)$.
  Complement each literal: $x' + (y + z)(y' + z') = F_2'$.

Other logic operations

When the binary operators AND and OR are placed between two variables, x and y, they form two Boolean functions, x.y and x + y, respectively. There are 22 functions for n binary variables. For two variables, n2, and the number of possible Boolean functions is 16. Therefore, the AND and OR functions are only two of a total of 16 possible functions. It would be instructive to find the other 14 functions and investigate their properties.

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
Operator symbol			$\cdot$	$/$		$/$		$\oplus$	$+$	$\downarrow$	$\odot$	$'$	$\subset$	$'$	$\supset$	$\uparrow$	

The 16 functions listed in truth table form above can be expressed algebraically by means of Boolean expressions as in following table. Each of the functions is listed with an accompanying name and a comment that explains the function in some way. The 16 functions listed can be subdivided into three categories:

1. Two functions that produce a constant 0 or 1.
2. Four functions with unary operations: complement and transfer.
3. Ten functions with binary operators that define eight different operations: AND, OR, NAND. NOR, exclusive OR, equivalence, inhibition, and implication.

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
$F_9 = xy + x'y'$	$x \odot y$	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

Table: **Boolean Expressions for the 16 Functions of Two Variables**

Digital Logic gates (In detail)

Boolean functions are expressed in terms of AND, OR, and NOT logic operations, and hence are easier to implement with these types of gates. The possibility of constructing gates for the other logic operations is of **practical interest**. Factors to be weighed when considering the construction of other types of logic gates are:

- ➢ The feasibility and economy of producing the gate with physical components
- ➢ The possibility of extending the gate to more than two inputs
- ➢ The basic properties of the binary operator such as commutativity and associativity, and
- ➢ The ability of the gate to implement Boolean functions alone or in conjunction with other gates.

Of the 16 functions defined in Table above, two are equal to a constant and four others are repeated twice. There are only ten functions left to be considered as candidates for logic gates. Two, inhibition and implication, are not commutative or associative and thus are impractical to use as standard logic gates. The other eight: **complement**, **transfer**, **AND**, **OR**, **NAND**, **NOR**, **exclusive-OR**, and **equivalence**, are used as standard gates in digital design.

The graphic symbols and truth tables of the eight gates are shown below:

Boolean Algebra and Logic Gates

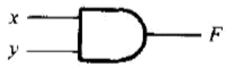
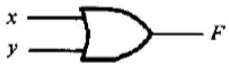
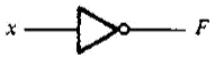
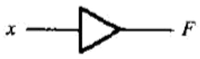
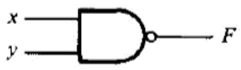
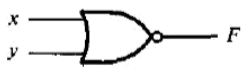
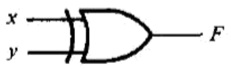
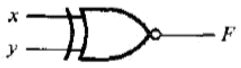
Name	Graphic symbol	Algebraic function	Truth table
AND		$F = xy$	x y | F 0 0 | 0 0 1 | 0 1 0 | 0 1 1 | 1
OR		$F = x + y$	x y | F 0 0 | 0 0 1 | 1 1 0 | 1 1 1 | 1
Inverter		$F = x'$	x | F 0 | 1 1 | 0
Buffer		$F = x$	x | F 0 | 0 1 | 1
NAND		$F = (xy)'$	x y | F 0 0 | 1 0 1 | 1 1 0 | 1 1 1 | 0
NOR		$F = (x + y)'$	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		$F = xy + x'y'$ $= x \odot y$	x y | F 0 0 | 1 0 1 | 0 1 0 | 0 1 1 | 1

Fig. 8 Logical gates

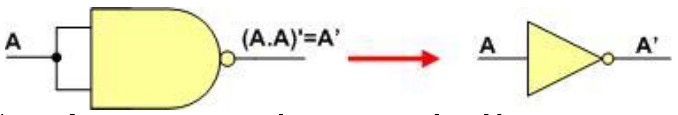
Universal gates

A universal gate is a gate which can implement any Boolean function without need to use any other gate type. The **NAND** and **NOR** gates are universal gates. In practice, this is advantageous since NAND and NOR gates are economical and easier to fabricate and are the basic gates used in all IC digital logic families.

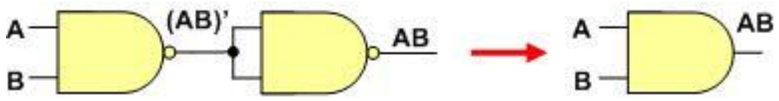
1. **NAND Gate is a Universal Gate**

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

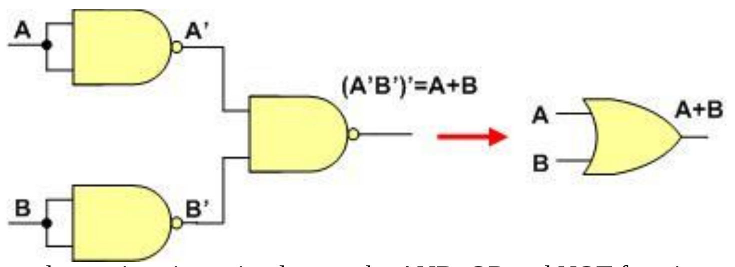
- **Implementing an Inverter Using only NAND Gate:** All NAND input pins connected to the input signal A gives an output A'.



- **Implementing AND Using only NAND Gates:** The AND is replaced by a NAND gate with its output complemented by a NAND gate inverter.



- **Implementing OR Using only NAND Gates:** The OR gate is replaced by a NAND gate with all its inputs complemented by NAND gate inverters.

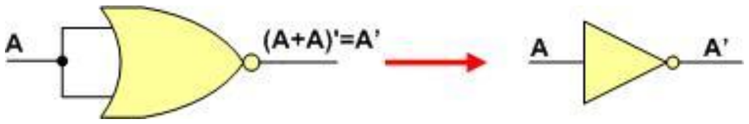


Thus, the **NAND gate** is a universal gate since it can implement the AND, OR and NOT functions.

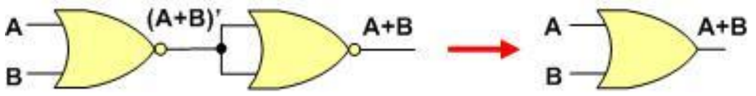
2. **NOR Gate is a Universal Gate:**

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

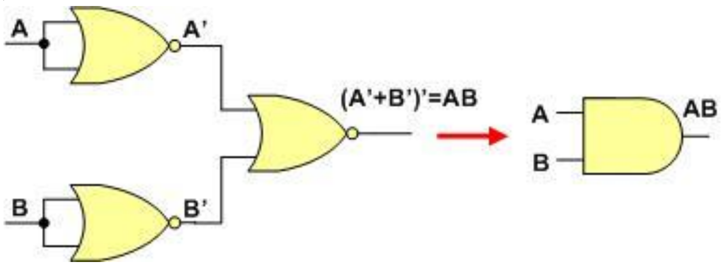
- ❖ **Implementing neither an Inverter Using only NOR Gate:** All NOR input pins connect to the input signal **A** gives an output **A'**.



- ❖ **Implementing OR Using only NOR Gates:** The OR is replaced by a NOR gate with its output complemented by a NOR gate inverter.



- ❖ **Implementing AND Using only NOR Gates:** The AND gate is replaced by a NOR gate with all its inputs complemented by NOR gate inverters.



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

Extending gates to multiple inputs

The gates shown in Fig above, except for the inverter and buffer, can be extended to have more than two inputs. A gate can be extended to have multiple inputs if the binary operation it represents is commutative and associative.

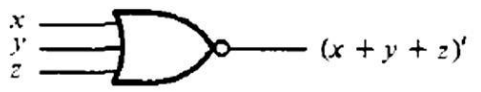
- ➢ The AND and OR operations, defined in Boolean algebra, possess these two properties. For the OR function, we have *x + y = y + x commutative* and *(x +* y) + z = *x +* (*y +* z) = *x +* y + z associative, which indicates that the gate inputs can be interchanged and that the OR function can be extended to three or more variables.
- ➢ The NAND and NOR functions are commutative and but not associative {x↓(y ↓ z) ≠ (x ↓ y) ↓z]

$$(x \downarrow y) \downarrow z = [(x + y)' + z]' = (x + y)z' = xz' + yz'$$

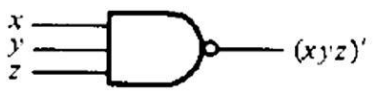
$$x \downarrow (y \downarrow z) = [x + (y + z)']' = x'(y + z) = x'y + x'z$$

Their gates can be extended to have more than two inputs, provided the definition of the operation is slightly modified. We define the multiple NOR (or NAND) gate as a complemented

OR (or AND) gate i.e. x ↓ y ↓ z = (x + y + z)' and x ↑y ↑ z = (xyz)'.

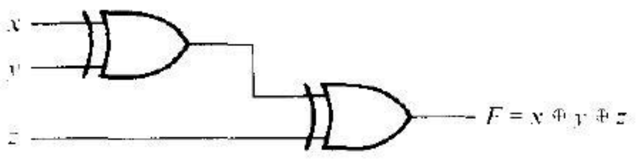


(a) Three-input NOR gate

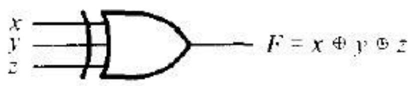


(b) Three-input NAND gate

- ➢ The exclusive-OR and equivalence gates are both commutative and associative and can be extended to more than two inputs



(a) Using 2-input gates



(b) 3-input gate

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

(c) Truth table

Fig: 3-input XOR gate

IC digital logic Families
Continuing the introduction of integrated Circuits (Chips) in unit 1,

Digital logic families

- Digital logic family refers to the specific circuit technology to which digital integrated circuits belong. Family has its own basic electronic circuit upon which more complex digital circuits and components are developed. The basic circuit in each technology is a NAND, NOR, or an inverter gate. The **electronic components used in the construction of the basic circuit are usually used as the name of the technology**. Different logic families have been introduced commercially. Some of most popular are:
- **TTL (transistor-transistor logic):** The TTL family evolved from a previous technology that used diodes and transistors for the basic NAND gate. This technology was called DTL for diode-transistor logic. Later the diodes were replaced by transistors to improve the circuit operation and the name of the logic family was changed to TTL.
- **ECL (emitter-coupled logic):** Emitter-coupled logic (ECL) circuits provide the highest speed among the integrated digital logic families. ECL is used in systems such as supercomputers and signal processors, where high speed is essential. The transistors in ECL gates operate in a non-saturated state, a condition that allows the achievement of propagation delays of 1 to 2 nanoseconds.
- **MOS (metal-oxide semiconductor):** The metal-oxide semiconductor (MOS) is a unipolar transistor that depends upon the flow of only one type of carrier, which may be electrons (n-channel) or holes (p-channel), this is in contrast to the bipolar transistor used in TTL and ECL gates, where both carriers exist during normal operation. A p-channel MOS is referred to as PMOS and an n-channel as NMOS. NMOS is the one that is commonly used in circuits with only one type of MOS transistor.
- **CMOS (complementary metal-oxide semiconductor):** Complementary MOS (CMOS) technology uses one PMOS and one NMOS transistor connected in a complementary fashion in all circuits. The most important advantages of MOS over bipolar transistors are the high packing density of circuits, a simpler processing technique during fabrication, and a more economical operation because of the low power consumption.

IIL (Integrated Injection Logic): Integrated injection logic (**IIL**, **I^{2}L**, or **I2L**) is a class of digital circuit technology built with multiple collector bipolar junction transistors (BJT). When introduced it had speed comparable to TTL yet was almost as low power as CMOS, making it ideal for use in VLSI (and larger) integrated circuits. Although the logic voltage levels are very close (High: 0.7V, Low: 0.2V), I^{2}L has high noise immunity because it operates by current instead of voltage. Sometimes also known as Merged Transistor Logic.
Currently, silicon-based Complementary Metal Oxide Semiconductor (CMOS) technology dominates due to its high circuit density, high performance, and low power consumption. Alternative technologies based on Gallium Arsenide (GaAs) and Silicon Germanium (SiGe) are used selectively for very high speed circuits.

Characteristics of digital logic families (Technology Parameters)
For each specific implementation technology, there are details that differ in their electronic circuit design and circuit parameters. The most important parameters used to characterize an implementation technology are:

1. **Fan-in**

For high-speed technologies, *fan-in,* the number of inputs to a gate, is often restricted on gate primitives to no more than four or five. This is primarily due to electronic considerations related to gate speed. To build gates with larger fan-in, interconnected gates with lower fan-in are used during technology mapping. A mapping for a 7-input NAND gate is made up of two 4- input NANOs and an inverter as shown in figure.

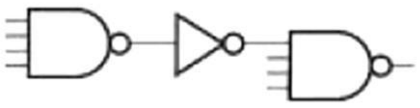


Fig: Implementation of a 7-input NAND Gate using NAND Gates with 4 or Fewer Inputs.

2. Propagation delay

The signals through a gate take a certain amount of time to propagate from the inputs to the output. This interval of time is defined as the **propagation delay of the gate**. Propagation delay is measured in nanoseconds (ns). 1 ns is equal to 10^{-9} of a second. The signals that travel from the inputs of a digital circuit to its outputs pass through a series of gates. The sum of the propagation delays through the gates is the total delay of the circuit.

The average propagation delay time of a gate is calculated from the input and output waveforms as:

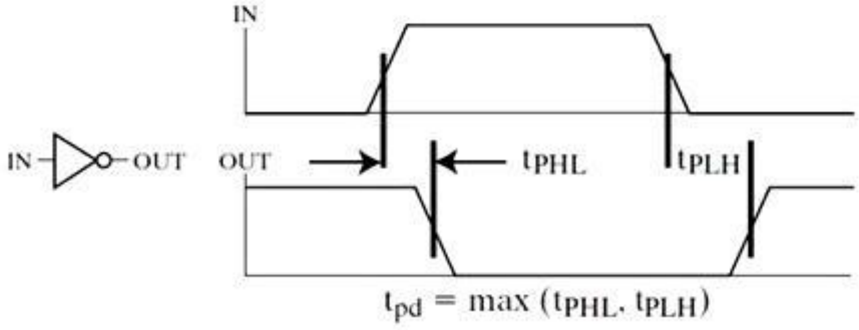


Fig: Measurement of propagation delay

- Delay is usually measured at the 50% point with respect to the H and L output voltage levels.
- High-to-low (t_{PHL}) and low-to-high (t_{PLH}) output signal changes may have different propagation delays.
- High-to-low (HL) and low-to-high (LH) transitions are defined with respect to the output, **not the input**.
- An HL input transition causes:
  - an LH output transition if the gate inverts and
  - An HL output transition if the gate does not invert.

3. Fan-out

Fan-out specifies the number of standard loads driven by a gate output i.e. Fan-out is a measure of the ability of a logic gate output to drive a number of inputs of other logic gates of the same type. Maximum Fan-out for an output specifies the fan-out that the output can drive with out exceeding its specified maximum transition time. Standard loads may be defined in a variety of ways depending upon the technology. For example: the input to a specific inverter can have load equal to 1.0 standard load. If a gate drives six such inverters, then the fan-out is equal to 6.0 standard loads.

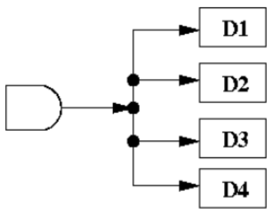


Fig: AND gate above is attached to the inputs of four other components so has a fan out of 4.

4. Power Dissipation

Every electronic circuit requires a certain amount of power to operate. The power dissipation is a parameter expressed in millwatts (mW) and represents the amount of power needed by the gate. The number that represents this parameter does not include the power delivered from another gate; rather, it represents the power delivered to the gate from the power supply. An IC with four gates will require, from its power supply, four times the power dissipated in each gate.

The amount of power that is dissipated in a gate is calculated as:

P_D (Power Dissipation) = V_{CC} * I_{CC} Where V_{cc} = *supply voltage and*
I_{cc} = *current drawn by the circuit*

The current drain from the power supply depends on the logic state of the gate. The current drawn from the power supply when the output of the gate is in the high-voltage level is termed I_{CCH}. When the output is in the low-voltage level, the current is I_{CCL}. The average current is

$$I_{CC}(\text{avg}) = \frac{I_{CCH} + I_{CCL}}{2}$$

And used to calculate the average power dissipation as,

$$P_D(\text{avg}) = I_{CC}(\text{avg}) \times V_{CC}$$

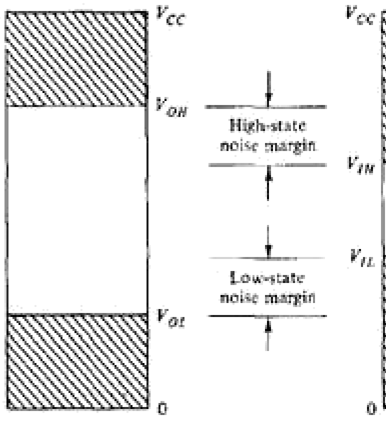
Example: A standard TTL NAND gate uses a supply voltage V_{cc} of 5V and has current drains I_{CCH} = 1 mA and I_{CCL} = 3 mA. The average current is (3 + 1)/2 = 2 mA. The average power dissipation is 5 x 2 = 10 mW. An IC that has four NAND gates dissipates a total of 10 x 4 = 40 mW. In a typical digital system there will be many ICs, and the power required by each IC must be considered. The total power dissipation in the system is the sum total of the power dissipated in all ICs.

5. **Noise Margin**

Undesirable or unwanted signals (e.g. voltages, currents etc) on the connecting wires between logic circuits are referred to as *noise.* There are two types of noise to be considered:

- **DC noise** is caused by a drift in the voltage levels of a signal.
- **AC noise** is a random pulse that may be created by other switching signals.

Thus, noise is a term used to denote an undesirable signal that is superimposed upon the normal operating signal. ***Noise margin*** is the maximum noise voltage added to an input signal of a digital circuit that does not cause an undesirable change in the circuit output. The ability of circuits to operate reliably in a noise environment is important in many applications. Noise margin is expressed in volts and represents the maximum noise signal that can be tolerated by the gate.



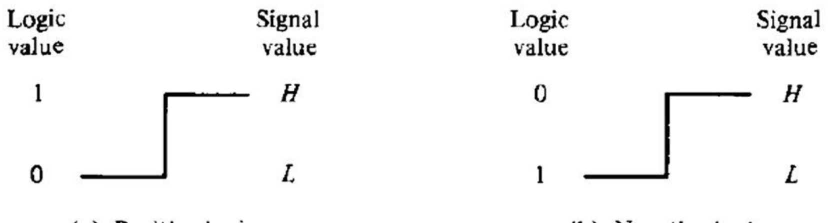
(a) Output voltage range (b) Input voltage range

In fig, V_{OL} is the maximum voltage that the output can be when in the low-level state. The circuit can tolerate any noise signal that is less than the noise margin *(V_{IL} - V_{OL})* because the input will recognize the signal as being in the low-level state. Any signal greater than V_{OL} plus the noise-margin figure will send the input voltage into the indeterminate range, which may cause an error in the output of the gate. In a similar fashion, a negative-voltage noise greater than V_{OH} - V_{IH} will send the input voltage into the indeterminate range.

The parameters for the noise margin in a standard TTL NAND gate are V_{OH} = 2.4 V, V_{OL} = 0.4 V, V_{IH} = 2 V, and V_{IL} = 0.8 V. The high-state noise margin is 2.4 - 2 = 0.4 V, and the low-state noise margin is 0.8 - 0.4 = 0.4 V. In this case, both values are the same.

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. So there is a possibility of two different assignments of signal level to logic value, as shown in Fig.

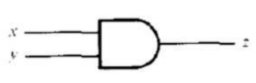


(a) Positive logic (b) Negative logic

- 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 signal values that determine the type of logic, but rather the assignment of logic values to the relative amplitudes of the two signal levels.

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

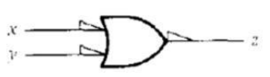
(c) Truth table for positive logic



(d) Positive logic AND gate

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