Unit 1

BINARY SYSTEMS

Unit 1: Binary Systems (6 Hrs.)
Digital Systems, Binary numbers, Number base conversion, Octal and hexadecimal numbers, compliments, Signed Binary numbers, Decimal codes (BCD, 2 4 2 1,8 4 -2 -1,Excess 3, Gray Code), Binary Storage and Registers, Binary logic

Digital Systems

Introduction:

The general-purpose digital computer is the best example of digital systems. Other examples include, digital watch, digital calculator, digital display, telephone switching exchanges etc. The most widely used digital system is the digital computer itself. Digital computers have made possible many scientific, industrial and commercial advances that would have been unattainable otherwise. Advance scientific studies, researches and experiments require real-time, continuous computer monitoring, and many businesses enterprises function efficiently only with the aid of automatic data processing. Computers are used in scientific calculations, commercial and business data processing, air traffic control, space guidance, the educational field, and many other areas. Digital computers are largely used in design, manufacture, distribution and sales. Besides, we use digital computers in education, exploration, entertainment and many other fields to accomplish wide variety of data processing tasks.

The major characteristic of digital system is manipulation of discrete elements of information. Such elements of information may be electric impulses, numbers, alphabets, operations etc. mostly the computers are known for numerical computations. In this case digits are the discrete elements and hence the name digital computer has emerged. A more appropriate word for a digital computer would be “discrete information processing system”.

Discrete elements of information are represented in digital systems by physical quantities called signals. Electrical signals such as voltage and currents are the most common. A piece of signal has two possible values and hence is binary in nature. The digital system designer is restricted to use two binary signals. Multivalued discrete systems are found to be less reliable than binary system. A circuit in binary system will be in two states. A system with ten states could be designed with ten states circuit using one discrete voltage value for each state, but it would possess very less reliability. For example, a transistor circuit has two states “on” and “off” and is extremely reliable. Moreover, human logic also tend to be binary.

The two constraints in digital systems are:

- Information representation in discrete format. It means, system can be in either of the fixed, predefined, discrete states. Not continuous.

- The information is stored in binary format. System must have only two states.

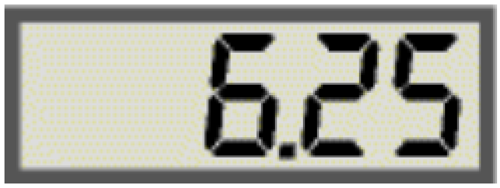
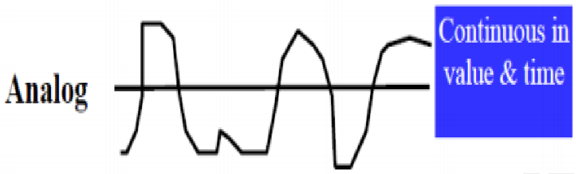
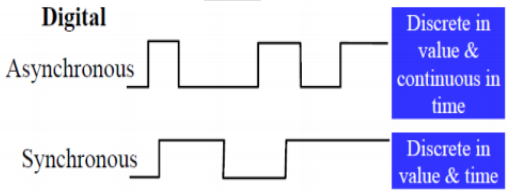
Discrete quantities of information arise either from the nature of the process or may be quantized from a continuous process. For example, number of people is by nature discrete information. The weight of a person is continuous in nature. In a digital weighing machine, the continuous information is needed to be quantized.

Information representation in digital systems:

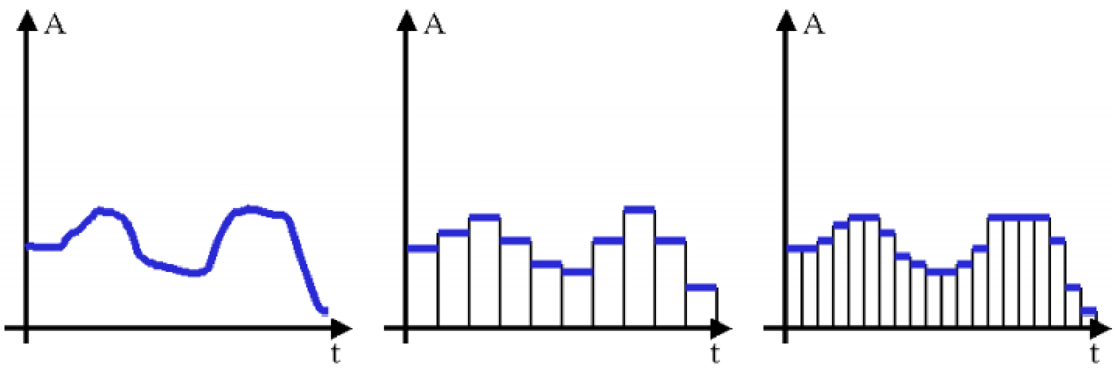
- Process variables (Information) are represented in the form of discrete signals such as voltage.
- The discrete signals (presence and absence of voltage) are represented by two values (for discrete binary systems) 0 and 1.
- Signal values are obtained after quantization of the process variables (information).
- After quantization, a single piece of discrete value may represent a certain range of real-time-value.
- Lower the quantization range, higher is the precision of digital system.

Differences between analog and digital systems:

Analog System	Digital System
Can perform direct simulation of the physical processes. If the variables of the process are discrete in nature, they are converted to continuous signals by using digital to analog converter (modulator).	To simulate a physical process in a digital system, the quantities (process variable) must be quantized. If the variables of the process are real time continuous signals, they are quantized by analog to digital converter (demodulator).
Variables in an analog system are represented by the continuous signals. Usually by the electric voltage that vary with time.	Variables in a digital system are represented by the discrete signals. Usually in binary state.
The process variables and the signals are analogous in behave in the same manner. For example, in an analog voltmeter, the process variable is voltage and same can be used as signal.	The process variables and the signals are not analogous in behave in the same manner. For example, in a digital speedometer, the rotation of wheel (process variable) is represented by the quantized electrical signals.
The analog signals are continuous in nature.	The digital signals are discrete in nature.
An analog signal may represent infinite states.	A digital signal may represent either of the fixed number of states. (two states in binary digital system).
A physical system whose behavior is described by mathematical equations is simulated in analog system by using calculus.	A physical system whose behavior is described by mathematical equations is simulated in digital system by using numerical methods.

There are infinite possible values between any two values, however the ability to read them precisely may differ.	There are only fixed number of values between any two values. They can be read precisely.
 Above speedometer is an analog system representing probably 1.25. it can represent infinite values between 1.25 and 1.26. we can increase the precision of the system by o	 Above speedometer is a digital system representing exactly 6.25. The next value it can represent is 6.26. It can't represent any value between 6.25 and 6.26. we can increase the precision of the system by using better architecture.
Analog systems use microdevices such as shifts and gares.	Digital systems contain devices such as logic gates, flip-flops, shift registers and counters.
Signal represents continuous values. 	Signal represents discrete values. 

Here is an example waveform of a quantized signal. In case of the digital 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**

Note that, in digital systems, the signal represent two states (normally indicated by 0 and 1) of the corresponding physical quantity. The two states are the quantized representation of corresponding physical variable. The physical variable being:

- High and low voltage in CPU.
- Magnetized and non-magnetized spot in magnetic disk.
- Pits and lands in CD.

- Electrical charge + and – in RAM etc.

Working principle of digital computers:

The best example of a digital system is a digital computer system. Following diagram represent the basic concept of digital computer system.

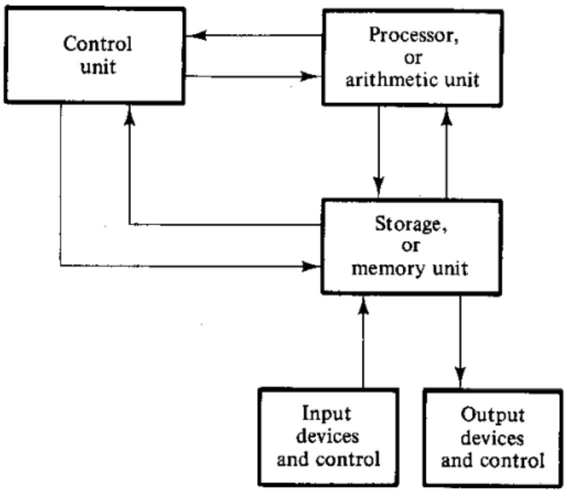


Fig: Block diagram of a digital computer system

The memory unit stores data regarding input, output or intermediate. It also stores the program to be executed. The processing unit performs the arithmetic and logic operations as instructed by the program. The control unit generates and gives control signals to various units to implement any instruction mentioned in the program. The program and data prepared by the user are transferred into the memory unit by means of an input device such as a keyboard. And output device such as printer receives the result of a computation and provides the result to the user. The input and output devices are special digital systems driven by electromechanical parts and controlled by electronic digital circuits.

Advantages of digital systems:

- It made many scientific, industrial, and commercial advances possible that would have been unattainable otherwise.
- It is less expensive and more reliable.
- Provides higher flexibility and compatibility
- Information storage can be easier in digital computer systems than in analog ones.
- Digital systems are easier to add new features and upgrade.

Disadvantages of digital systems:

- It is highly fragile system. Misinterpretation of tiny bit results in large error.
- Physical systems cannot be directly represented, rather quantization is needed which may result in quantization error.
- Poor architecture may result into higher data limitation.

The digital computer manipulates discrete elements of information represented in binary form. Data needed for calculation are represented in binary number system. Decimal digits are stored in binary codes. Data storage is done in binary storage elements. The instructions are represented in binary systems. It is the most important concept to have the idea of binary numbers to better understand the digital systems.

Number System:

Number system is a system of representing values by using a set of specified symbols following the predefined rules.

The two types of number systems are:

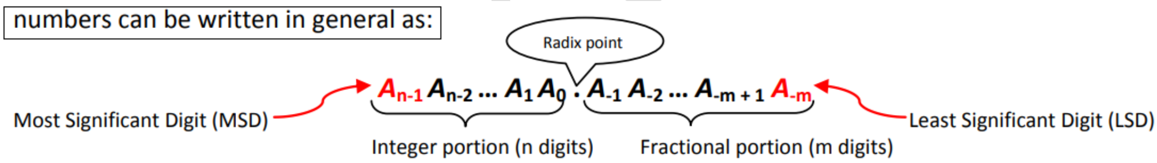
- Non-positional number system
- Positional number system

In the non-positional number systems, the digits do not have their place values. Each symbol has a fixed value, irrespective of the position it is placed. For example, roman number system.

In the positional number system, numbers are expressed in a sequence of digits. Each digit has its face value and place value. Face value of the digit is fixed and the place value of the digits depends in its position in the sequence of digits.

Representation of positional number system:

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:



Value of each A_i is such that 0<= A_i <= r. Where;

- **A_i =>** symbols used in the number system (Eg: 0-9 for decimal & 0,1 for binary number system)
- **r =>** radix of the number system. It is equal to the total number of symbols used in the system. For example, r=10 for decimal number system & r=2 for binary number system.
- **i =>** Position of the digit/symbol being used in the number.

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:

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

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

$$(\text{Integer Portion}) + (\text{Fraction Portion})$$

Number Systems Overview:

1. **Decimal number system:**

Radix/Base (r)	10
Symbols	0 to r-1 => 0,1,2,3,4,5,6,7,8,9
Example: 426.56_{10}	($4x10^2 + 2x10^1 + 6x10^0$) + ($5x10^{-1} + 6x10^{-2}$) Whole part fractional part
Use	Daily arithmetic calculation in real life.

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.

2. **Binary Number System:**

Radix/Base (r)	2
Symbols	0 to r-1 => 0,1
Example: 11010.01_2 is written in decimal as	($1x2^4$+$1x2^3$ +$0x2^2$ + $1x2^1$ + $0x2^0$ +($1x2^{-1}$ + $0x2^{-2}$) =$(26.25)_{10}$ Whole part fractional
Use	Used to represent values in binary digital systems. The binary number system is very useful in computer technology and computer programming languages also uses binary number system that is helpful in digital encoding. The binary number system can also be used in Boolean algebra.

Digits in a binary number are called bits (Binary digITs). 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,

$$(11010.01)_2 = 16 + 8 + 2 + 0.25 = (26.25)_{10}$$

Note: Please revise the binary arithmetic (addition, subtraction and multiplication and division of binary numbers).

3. **Octal Number System:**

Radix/Base (r)	8
Symbols	0 to r-1 => 0,1,2,3,4,5,6,7
Example: $(426.56)_8$ Written in decimal as	($4x8^2 + 2x8^1 + 6x8^0$) + ($5x8^{-1} + 6x8^{-2}$) = $(278.72)_8$ Whole fractional
Use	The octal numbers are not as common as they used to be. However, Octal is used when the number of bits in one word is a multiple of 3. It is also used as a shorthand for

	representing file permissions on UNIX systems and representation of UTF8 numbers, etc.

Octal number are expressed with a string 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 as shown above.

4. Hexadecimal Number System:

Radix/Base (r)	16
Symbols	0 to r-1 => 0,1,2,3,4,5,6,7,8,9, A, B, C, D, E, F Where; A = 10, B=11, C=12, D=13, E=14, F=15
Example: $(4B6.56)_{16}$	$(4x16^2 + 11x16^1 + 6x16^0) + (5x16^{-1} + 6x16^{-2}) = (1206.34)_{16}$ Whole part fractional part
Use	Hexadecimal Number System is commonly used in Computer programming and Microprocessors. It is also helpful to describe colors on web pages. Each of the three primary colors (i.e., red, green and blue) is represented by two hexadecimal digits to create 255 possible values, thus resulting in more than 16 million possible colors. Hexadecimal number system is used to describe locations in memory for every byte. These hexadecimal numbers are also easier to read and write than binary or decimal numbers for Computer Professionals.

A hexadecimal number is expressed with a string of symbols defined above (0-F), 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.

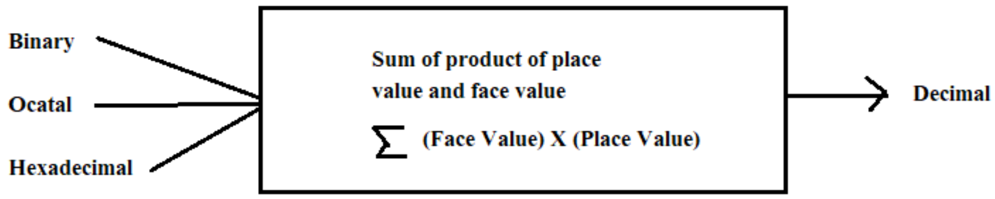
Following figure shows the representation of first 16 decimal numbers in different number systems.

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

Fig: First 16 numbers in different number system

Number Base Conversion:

1. Conversion from any number system to decimal:



Example: $(100110)_2 = (?)_{10}$

Face Value	1	0	0	1	1	0
Place Value	32	16	8	4	2	1

Answer = $(1x32 + 0x16 + 0x8 + 1x4 + 1x2 + 0x1)_{10}$

= $(38)_{10}$

Example: $(245)_8 = (?)_{10}$

Face Value	2	4	5
Place Value	64	8	1

Answer = $(2x64 + 4x8 + 5x1)_{10}$

= (165)

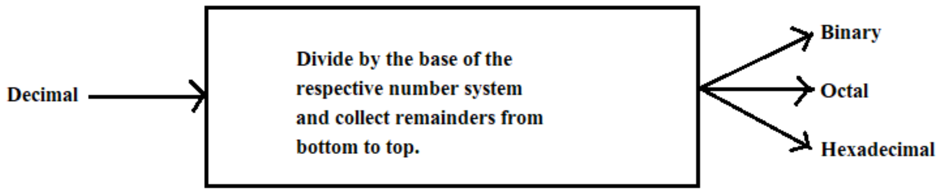
Example: $(4B6)_{16} = (?)_{10}$

Face Value	4	11	6
Place Value	256	16	1

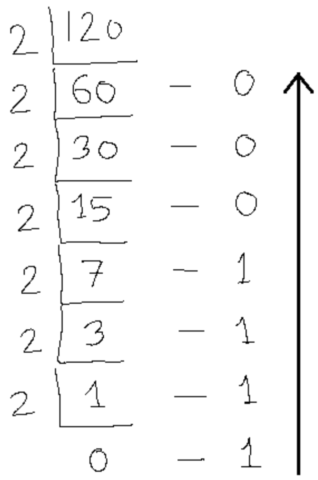
Answer = $(4x256 + 11x16 + 6x1)_{10}$

= $(1206)_{10}$

2. Conversion from decimal to any other number system:

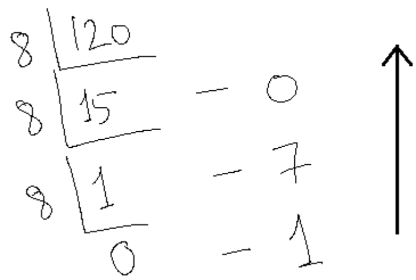


Example: $(120)_{10} = (?)_2$



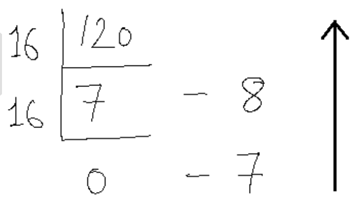
Answer = $(1111000)_2$

Example: $(120)_{10} = (?)_8$



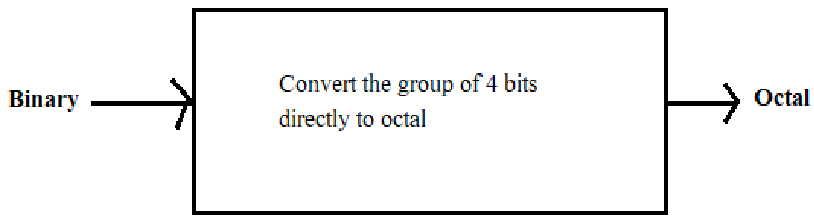
Answer = $(170)_8$

Example: $(120)_{10} = (?)_{16}$

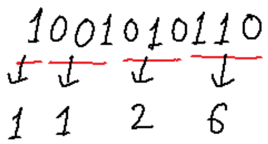


Answer = $(78)_{16}$

3. Binary to octal conversion:

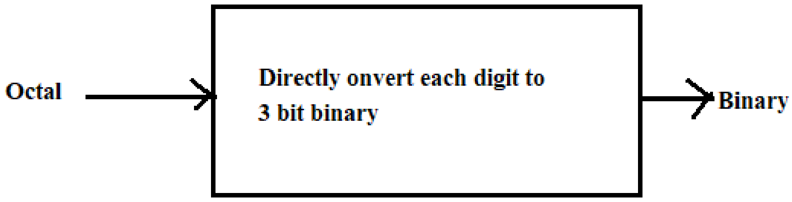


Example: $(1001101010110)_2 = (?)_8$

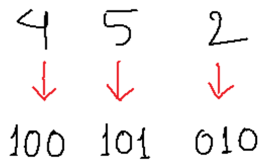


Anser = $(1126)_8$

4. Octal to binary conversion:

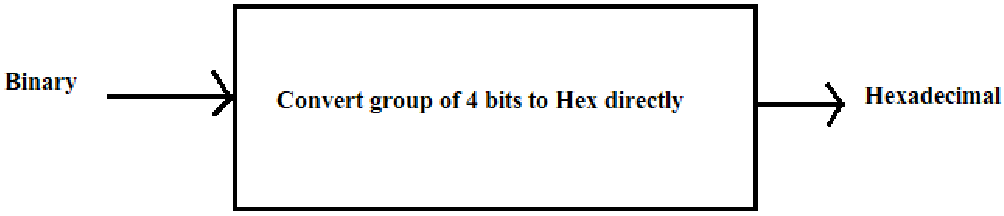


Example: $(452)_8 = (?)_2$

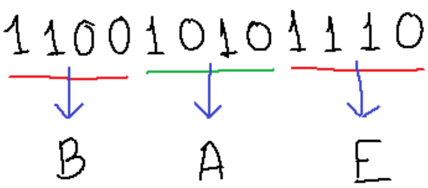


Ans = $(100101010)_2$

5. Binary to hexadecimal conversion:

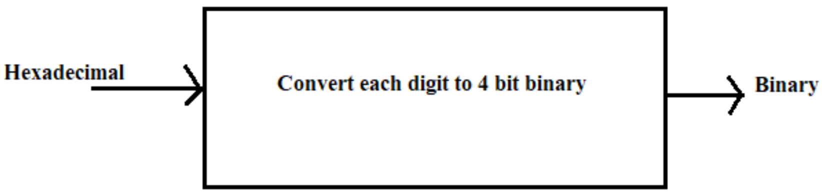


Example: $(110010101110)_2 = (?)$

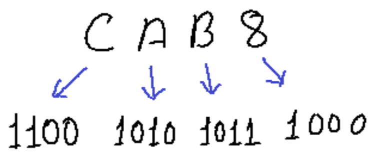


Ans = $(BAE)_{16}$

6. Hexadecimal to binary conversion:

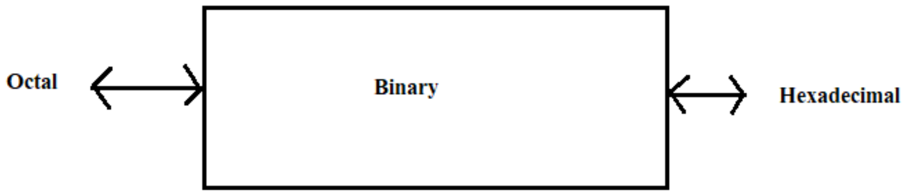


Example: $(CAB8)_{16} = (?)_2$



Ans = $(1100101010111000)_2$

7. **Octal to hexadecimal conversion (And vice-versa):**



To convert from octal to hexadecimal, first we convert octal to binary and then binary to hexadecimal. Similarly, for hexadecimal to octal.

Note that, computer understands only binary numbers. If he human need to communicate, use of binary system is hectic. Use of binary numbers in human communication is too complex because it needs large numbers of digits to represent a simple value. For example, to represent $(B5C)_{16}$ 3 digits, we need $(101101011100)_2$ 12 digits.

Complements:

Complements are used in digital computers for simplifying the subtraction operation and for logical manipulations. There are two types of complements for base-r system.

a. The r's complement (Example: 2's complement for binary, 10's complement for decimal)
b. The (r-1)'s complement (Example: 1's complement for binary, 9's complement for decimal)

a. The r's complement (radix complement):

Given a positive number N in base r with an integer part of n digits, the r's complement of N is defined as r^n-N for N>0 and 0 for N=0.

Example: Find 10's complement of $(256)_{10}$.

Answer : Here,

N = 256, n = 3 and r = 10

r's complement of N = 10's complement of 256_{10}

$= r^n - N$

$= 10^3 – 256$

$= (744)_{10}$

Note: We can easily find the 10's complement of a decimal number by following procedure:

- Leave all the least significant zeros unchanged.
- Subtract first non-zero least significant digit from 10.
- Subtract all the other higher significant digits from 9.
- Eg: 10's complement of 256 is (9-2)(9-5)(10-6) = 744.
- For fraction also, same rule works. For example, 10's complement of 25.639 is 74.361.

Example: Find 2's complement of $(10110)_2$.

Answer: Here,

N = 10110, n = 5 and r = 2

r's complement of N = 2's complement of 10110

$= r^n - N$

$= 2^5 – 10110_2$

$= 32 - 10110_2$

$= 100000_2 - 10110_2$

$= 01010_2$

Note: We can easily find the 2's complement of a binary number by following procedure:

- Leave all the least significant zeros unchanged.
- Leave the first least significant non-zero digit.
- Replace all 0 by 1 and 1 by 0 in the higher significant digits.
- Eg: 2's complement of 10110 is 01010.

b. The (r-1)'s complement (Diminished radix complement):

Given a positive number N in base r with an integer part of n digits and a fraction part of m digits, the (r-1)'s complement of N is defined as $r^n – r^{-m}$ - N.

Note: In case of a whole number, the (r-1)'s complement is defined as r^n -1 – N.

Example: Find 9's complement of 256.

Ans: Here, N = 256, n = 3, m = 0, r = 10.

9's complement of 256 = (r-1)'s complement of N

$= r^n – r^{-m} – N$

$= 10^3 - 10^{-0} – 256$

= 1000-1-256

= 743

Note: To find 9's complement of a decimal number, we can simply subtract each digit from 9. Whether it is whole number or fraction.

Try yourself:

- 9's complement of 52520 is 47479.
- 9's complement of 0.32670.6732

Example: Find 1's complement of 10110.

Ans: Here, N = 10110, n=5, m=0, r=2

1's complement of 10110 = (r-1)'s complement of N

$= r^n - r^{-m} - N$

$= 2^5 - 2^{-0} - 10110$

= 32 – 1 – 10110

= 31 – 10110

= 11111 – 10110

= 01001

Note: To find 1's complement of a binary number, we can simply replace 1's by 0's and 0's by 1's.

Try yourself:

- 1's complement of 101100 is 010011
- 1's complement of 0.0110 is 0.1001.

Note: If we add 1 to (r-1)'s complement of a number, we will get r's complement.

Subtraction with r's complement:

The general method of subtraction uses a borrow method when a larger digit is to be subtracted from smaller one. This seems to be hard to implement in digital components. Moreover, computers are more efficient on adding rather than subtracting. We can apply the r's complement to perform subtraction in more efficient way.

Let, A and B be two positive numbers in base-r system. We may evaluate A-B by following the procedure given below.

- Add the minuend A to the r's complement of subtrahend B.
- Inspect the result obtained in above step for the end carry:
  - If an end carry occurs, discard the end carry and remaining value gives the desired result.

- If an end carry doesn't occur, take the r's complement of the obtained value and place a negative sign to get the final result.

Proof: Let A and B be two positive numbers in base-r system. Let the numbers be represented in N digits. Then,

$A + r^n - B \geq r^n$ if A>B ----------------- (a)

$A + r^n - B < r^n$ if A<B ----------------- (b)

Our desired result is to obtain A-B.

In case (a), we can obtain positive A-B simply by removing r^n which is equivalent to discarding end carry.

In case (b), the result should be –(B-A), which is obtained by taking r's complement once again and giving a minus sign in front as;

Complement once again => $r^n - (A+r^n-B)$

Attach minus sign => $-[\, r^n - (A+r^n-B)\,] = -(B-A)$

Example: Using 10's complement method, subtract 752-435.

Here,
Minuend(A) = 752
Subtrahend (B) = 435
10's Complement of subtrahend = 565

Now,

$$\begin{array}{r} 752 \\ +\,565 \\ \hline \boxed{1}\,317 \end{array}$$

end carry

∴ Ans = 317

Example: Using 10's complement method, subtract 245-275.

Here,
Minuend (A) = 245
Subtrahend (B) = 275
10's Complement of subtrahend = 725

Now,

$$\begin{array}{r} 245 \\ +\,725 \\ \hline 970 \end{array}$$

No end carry

Answer = −10's Complement of 970
= −030
= −30

Example: Using 2's complement method, subtract 10110 – 10011.

Here,

Minvend (A) = 10110

Subtrahend (B) = 10011

2's Complement of B = 01101

Now,

$$
\begin{array}{r}
10110 \\
+\ 01101 \\
\hline
\boxed{1}00011
\end{array}
$$

End Carry ←

∴ Ans = 11

Example: Subtract 101001 – 110110 using 2's complement method.

Here, Minuend (A) = 101001

Subtrahend (B) = 110110

2's Complement of B = 001010

Now,

$$
\begin{array}{r}
101001 \\
+\ 001010 \\
\hline
110011
\end{array}
$$

No end carry ←

Ans = – 2's complement of the result

= – 001101

= – 1101

Subtraction with (r-1)'s complement:

Let A and B be two positive numbers in base-r system. We may evaluate A-B as:

- Add the minuend A with (r-1)'s complement of subtrahend B.
- Inspect the result obtained in above step for end carry as:
  - If an end carry occurs, drop the end carry and add 1 to the least significant digit to get the final result.
  - If end carry doesn't occur, take (r-1)'s complement of the value to obtain final result.

Example: Subtract 536 – 345 using 9's complement.

Here,

Minuend (A) = 536

Subtrahend (B) = 345

9's complement of B = 654

Now,

$$\begin{array}{r} 536 \\ +654 \\ \hline \boxed{1}190 \end{array}$$

End carry ←

∴ Ans = 190+1

= 191

Example: Subtract 345-432 using 9's complement.

Here,

Minuend (A) = 345

Subtrahend (B) = 432

9's complement of B = 567

Now,

$$\begin{array}{r} 345 \\ +567 \\ \hline 912 \end{array}$$

No end carry! ←

Ans = − 9's complement of 912

= − 087

= −87

Example: Subtract 10110-101 using 1's complement.

Here,

Minuend (A) = 10110

Subtrahend (B) = 101 = 00101

1's complement of B = 11010

Now,

$$\begin{array}{r} 10110 \\ +11010 \\ \hline \boxed{1}10000 \end{array}$$

End carry ←

Ans = 10000 + 1

= 10001

Example: Subtract 1101-10110 using 1's complement method.

Here,

Minuend (A) = 1101 = 01101

Subtrahend (B) = 10110

1's Complement of B = 01001

Now,

$$\begin{array}{r} 01101 \\ +\ 01001 \\ \hline 10110 \end{array} \quad \text{No End Carry} \leftarrow$$

Ans = − 1's complement of 10110

= − 01001

= −1001

Note: The number of digits must match while doing any subtraction. If the minuend and subtrahend are not having same number of digits, we fix this by adding additional zeros in the most significant digit position (that means at the beginning of the number).

Comparison of 1's complement and 2's complement methods for subtraction:

2's complement method seems to be more appropriate than 1's complement method in computers due to the following reasons:

- In case of 1's complement method, if an end carry occurs, two addition operations are to be performed. Which leads to operational overhead.
- Zero in 2's complement has only one notation but in 1's complement, zero has two notations (+0 = 0000 and -0 = 1111).

Unsigned and Signed Binary Numbers:

Binary numbers can be represented in signed and unsigned way. Unsigned binary numbers do not have sign bit, whereas signed binary numbers uses signed bit as well or these can be distinguishable between positive and negative numbers.

1. **Unsigned Binary Numbers:** Unsigned numbers don't have any sign, these can contain only magnitude of the number. So, representation of unsigned binary numbers are all positive numbers only. For example, representation of positive decimal numbers are positive by default. We always assume that there is a positive sign symbol in front of every number.
   Since there is no sign bit in this unsigned binary number, so N bit binary number represent its magnitude only. Zero (0) is also unsigned number. This representation has only one zero (0), which is always positive. Every number in unsigned number representation has only one unique binary equivalent form, so this is unambiguous representation technique.

The range of unsigned binary number is from 0 to (2^n-1).

Example: Represent 43_{10} in unsigned binary number.

Ans: Here, we simply convert 43_{10} into binary form. The unsigned representation considers only magnitude. Hence, 43_{10} = 101011_2. All the bits are magnitude bits.

Example: Determine the range of 8-bits unsigned binary number. Also evaluate the minimum and maximum values in this range.

Ans: The range of unsigned binary numbers is 0 – (2^n-1). Hence the range of 8-bits unsigned binary numbers is from 0 to 2^8-1= 256_{10} which is 11111111_2.

2. **Signed binary numbers:** Signed numbers contain sign flag. This representation distinguishes positive and negative numbers. This technique contains both sign bit and magnitude of a number.

   There are three representations for the signed binary numbers.

   a. **Signed magnitude form:** For n bit binary number, 1 bit is reserved for sign symbol. If the value of sign bit is 0, then the given number will be positive, else if the value of sign bit is 1, then the given number will be negative. Remaining (n-1) bits represent magnitude of the number.

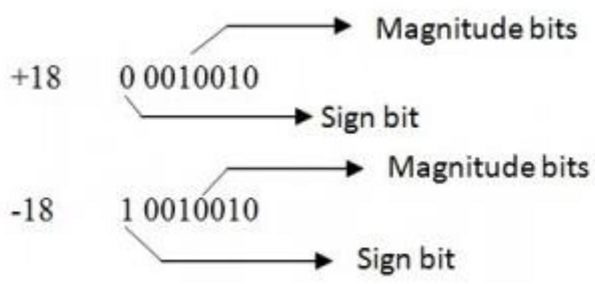
      The range of n-bit number in this representation is from –(2^{n-1}-1) to +(2^{n-1}-1).

      The limitation of this representation is that, the number 0 has two representations.

      +0 = 0 0000

      - 0 = 1 0000

      Example:



Q: Represent 45_{10} and -62_{10} in signed magnitude form.

Ans:

Here, $45_{10} = 101101_2$

In sign magnitude form, $+45_{10}$ can be represented as;

0	101101
Sign	magnitude

Again, $62_{10} = 111110_2$

In sign magnitude form, -62_{10} can be represented as;

1	111110
Sign	magnitude

b. **1's complement form:** we represent positive numbers in signed magnitude form and negative numbers in 1's complement form. There is extra bit for sign representation. If value of sign bit is 0, then number is positive and directly represents it in simple binary form, but if value of sign bit is 1, then number is negative and you have to take 1's complement to get the actual magnitude.

The range of n-bit number in this representation is from $-(2^{n-1}-1)$ to $+(2^{n-1}-1)$.

1's complement method is also ambiguous since 0 has two representations.

+0 = 0 0000

\- 0 = 1 1111

Example: Represent 57_{10} and -38_{10} in 1's complement form.

Ans: Here, 57_{10} is a positive number so we simply represent it in signed magnitude form as;

0	111001
Sign	magnitude

Again, to represent -38_{10}, we use 1 as sign bit and 1's complement of 38 in remaining bits as;

1	011001
Sign	Magnitude in 1's complement

c. **2's complement form:** we represent positive numbers in signed magnitude form and negative numbers in 2's complement form. There is extra bit for sign representation. If value of sign bit is 0, then number is positive and directly represents it in simple binary form, but if value of sign bit is 1, then number is negative and you have to take 2's complement to get the actual magnitude.

The range of n-bit number in this system is from $-(2^{n-1})$ to $+(2^{n-1}-1)$.

This method of representing negative numbers is most efficient for computers. Usually computers use 2's complement method for representing negative numbers.

Example: Represent 57_{10} and -38_{10} in 2's complement form.

Ans: Here, 57_{10} is a positive number so we simply represent it in signed magnitude form as;

0	111001
Sign	magnitude

Again, to represent -38_{10}, we use 1 as sign bit and 2's complement of 38 in remaining bits as;

1	011010
Sign	Magnitude in 1's complement

Decimal 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.

n-bit binary number can be used to code maximum of 2^n distinct items. For example, 3 bit binary numbers can be used to code $2^3 = 8$ distinct values. Binary

Binary codes for decimal digits require at least 4 bits. Different codes can be obtained by arranging four or more bits in ten distinct possible combinations. Some common binary codes for decimal digits are discussed below.

1. Binary Coded Decimal (BCD)

In this code each decimal digit is represented by a 4-bit binary number. BCD is a way to express each of the decimal digits with a binary code. In the BCD, with four bits we can represent sixteen numbers (0000 to 1111). But in BCD code only first ten of these are used (0000 to 1001). The remaining six code combinations i.e. 1010 to 1111 are invalid in BCD.

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

BCD codes are also called 8421 code as the four bits from left to right have the positional weight 8, 4, 2 and 1 respectively.

While converting BCD code to equivalent decimal, we convert each group of 4-bits separately. For example,

BCD value of 45_{10} is $(0100\ 0101)_{BCD}$

and

$(100100100011)_{BCD} = 1001\ \ 0010\ \ 1011_{BCD} = 923_{10}$

Though the BCD system is easy for humans, it is quite complicated to be applied in digital electronics. Moreover, it needs a greater number of bits to represent a simple number. It needs complex architecture.

Note: If we take weight of the 4-bits as 2, 4, 2, 1 respectively from left to right, the corresponding code is called 2421 code.

2. Error Detection Codes

Binary information can be transmitted from one location to another by electric wires or other communication medium. 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:

Message	0000	0101	1010	1011	1111
Even Parity	0				
Odd Parity	1				

In odd parity, 1 represents even number of 1’s in the message. In even parity, 0 represents an even number of 1’s in the message.

During the message transmission, a parity bit along with the message is sent to the destination. If the parity of the message at sending and receiving end doesn’t match, at least one bit in the message has changed during transmission.

This method has following limitations:

- Error detection becomes erroneous if the parity bit itself gets affected during transmission.

- It can only detect the change in odd number of bits during transmission. If even number of bits get affected, parity is not affected.

3. Reflected Code (Also called Gray Code or Unit Distance Code):

It is the non-weighted code and it is not arithmetic codes. That means there are no specific weights assigned to the bit position. It has a very special feature that, only one bit will change each time the decimal number is incremented as shown in fig. It is also called Gray code named after Frank Gray. As only one bit changes at a time, the gray code is called as a unit distance code. The gray code is a cyclic code. Gray code cannot be used for arithmetic operation. 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.

	Binary	Gray	
0.	0000	0	000
1.	0001	0	001
2.	0010	0	011
3.	0011	0	010
4.	0100	0	110
5.	0101	0	111
6.	0110	0	101
7.	0111	0	100
8.	1000	1	100
9.	1001	1	101
10.	1010	1	111
11.	1011	1	110
12.	1100	1	010
13.	1101	1	011
14.	1110	1	001
15.	1111	1	000

MIRROR (last3 bits of 7 & 8 are same, similarly for 6&9, 5&10 till 0&15)

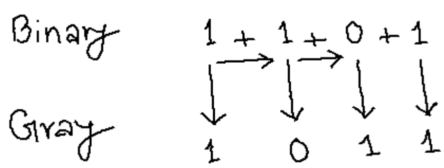
Gray code is also called reflected binary code due to mirror values as shown in above figure. The Gray code is used in applications where the normal sequence of binary numbers may produce an error or ambiguity during the transition from one number to the next.

<u>Binary to Gray conversion:</u>

Starting from left to right:

- If it is MSB, place it as it is in gray code.
- Otherwise get the gray code bit by adding current bit with previous bit and ignore any carry.
- Repeat step 2 till end.

Example: Convert 1101_2 to Gray code.

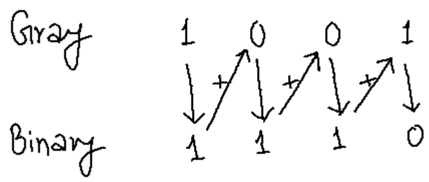


<u>Gray to Binary Conversion:</u>

Starting from left to left:

- If it is MSB, place it as it is in Binary code.
- Otherwise get the Binary code bit by adding current Binary bit with previous Gray code bit and ignore any carry.
- Repeat step 2 till end.

Example: Convert the Gray code 1001 to Binary code.



4. Access 3 (XS3) Codes:

XS3 Code is the unweighted binary code for decimal numbers. It is an extension to the BCD code. It is the self-complementary binary coded decimal code and numeral system. Excess-3 code was used on some older computers as well as in cash registers and hand-held portable electronic calculators of the 1970s, among other uses.

We obtain XS3 code by adding 3_{10} or 0011_2 to each digit in BCD code.

Decimal	BCD 8421	Excess-3 BCD+8421(0110)
0	0000	0011
1	0001	0100
2	0010	0101
3	0011	0110
4	0100	0111
5	0101	1000
6	0110	1001
7	0111	1010
8	1000	1011
9	1001	1100

Excess-3 code is self-complementary code; that means, 1's complement of XS3 code also points to 9's complement of corresponding decimal number.

Example: Find excess-3 code of 45_{10}.

Ans:

Decimal	BCD	XS3 = BCD+3 (each digit)
45	0100 0101	0100+0011 0101+0011 = 0111 1000

5. 2 4 2 1 Code: The Aiken code (also known as 2421 code) is a complementary binary-coded decimal (BCD) code. A group of four bits is assigned to the decimal digits from 0 to 9 according to the following table. The code was developed by Howard Hathaway Aiken and is still used today in digital clocks, pocket calculators and similar devices.

The Aiken code differs from the standard 8421 BCD code in that the Aiken code does not weight the fourth digit as 8 as with the standard BCD code but with 2. It is a four bit code having positional value 2, 4, 2 and 1 from MSB to LSB.

Decimal	0	1	2	3	4	5	6	7	8	9
2421	0000	0001	0010	0011	0100	1011	1100	1101	1110	1111

The code is self-complementary. 1's complement of 0 is 9 and that of 1 is 8 and so on.

6. Alphanumeric Codes:

The alphanumeric codes are the codes that represent numbers and alphabetic characters. Mostly such codes also represent other characters such as symbol and various instructions necessary for conveying information. An alphanumeric code should at least represent 10 digits and 26 letters of alphabet i.e. total 36 items. The following three alphanumeric codes are very commonly used for the data representation.

Two such codes are ASCII code and EBCDIC codes.

a. ASCII Code: ASCII stands for "American Standard for Information Interchange". It is a globally accepted character encoding standard. ASCII is a 7-bit character set containing 128 characters. It contains the numbers from 0-9, the upper and lower case English letters from A to Z, and some special characters. The character sets used in modern computers, in HTML, and on the Internet, are all based on ASCII.

ASCII codes for the common characters are:

For decimal digits 0 to 9	48 – 57
For lowercase letters a to z	65 – 90
For uppercase letters A to Z	97 - 122

The Extended Binary Coded Decimal (EBCDIC) was used earlier to ASCII as character encoding system specially for IBM mainframe computers. It was the extended form of BCD code to encode alphabets and other characters as well. Though it was used By IBM in the past, todays modern standard for character encoding is either ASCII itself or it is based on ASCII.

Binary Storage and Registers:

Binary Storage:

Binary storage is made up of binary cells. A binary cell is a device that possess two stable states and is capable of storing one bit of information (0 or 1). The excitation signals trigger the cell to generate the output signal representing one of the two states of the binary cell. The output signal represents 0 or 1 based on the interpretation of the two stable states of the output signal. Following diagram represents a typical binary storage system.

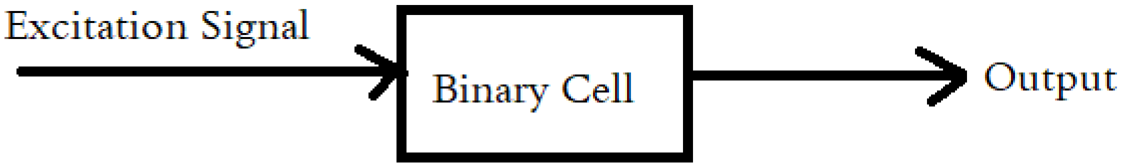


Fig: Binary storage System

For example, in a CD, the binary cell is represented by the surface element with two stable states pits or lands. Excitation signal is the laser beam and the output signal is the light detected by the sensor after reflection from pits or lands. Similarly we can consider a punched card as binary cell as well. In a digital computer, a single binary cell is well represented by a flip flop.

Register:

A register is a group of binary cells. A n bit register contains n different binary cells (flipflops) each storing a single bit. It can represent 2^n different states but only one at a time.

0	0	1	1	0	0	1	1

Fig: A 8-bit register

The content of the register is a string of binary digits. The interpretation of the content gives the complete information about the content. For example, above register contains 01011011, which may represent different values depending on the interpretation. If it is storing unsigned binary number, the content is +51. If it is a ASCII character, the content is the character 3 and so on.

Register Transfer:

A digital computer is characterized by its registers. The memory unit is the collection of thousands of registers for storing digital information. Processor unit also contains various registers to hold the operands on which operation are performed. The control unit also uses registers to store probably the commands. Input and output devices are also associated with I/O registers to store the information temporarily to transfer information to or from the device.

Register transfer operation is the operation that involves information transfer from one register to another. Following figure illustrates register transfer operation to illustrate the input from keyboard to memory unit.

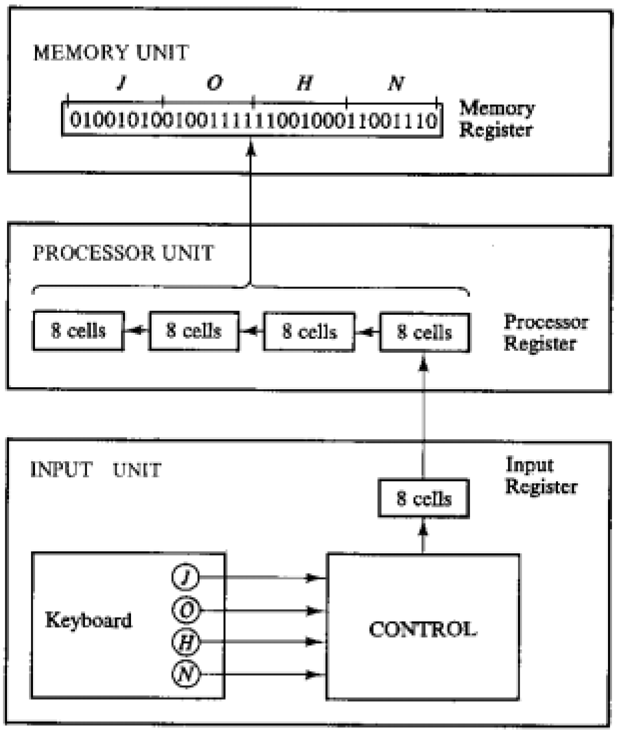


Fig: Transfer of information with register

Four characters “J”, “O”, “H”, “N” are entered from keyboard and are transferred from input register to processor register (register transfer). Capacity of processor register is four times the word length. The content of processor register is ultimately transferred to the memory register associated with memory unit.

For nay binary information processing, we need two fundamental components.

- Devices that can hold the data to be processed.
- Circuit that can manipulate individual bit of information.

The device commonly used for holding data is register and the bits manipulation circuit includes logic circuits. Following diagram shows the process of adding two 10-bit binary numbers.

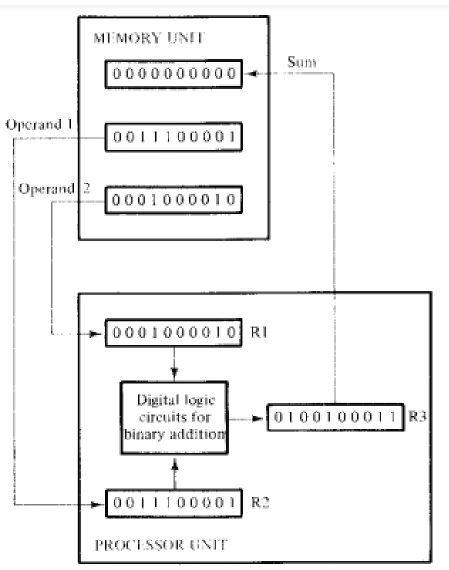


Fig: Example of binary information processing

The operands are fetched from memory unit to the registers R1 and R2 in the processor unit. the digital logic circuit performs binary addition and stores the result in register R3 which is then transferred to the memory unit. This process clearly illustrates the use of two components (registers and circuitry) of binary information processing system.

Binary Logic:

Binary logic deals with manipulation of binary information and generate logical output. Binary logic uses manipulation of binary variables. The binary variable may have any one of two possible states; (true or false, on or off, 1 or 0). For our convenient, we assume 0 or 1 as two possible states of a binary variable (also called as Boolean variable). Binary logic describes manipulation and processing of binary information. Binary logic helps to describe the working of a digital circuits. Binary logic can be explained by using Boolean algebra.

In binary logic, there are three basic logic operations: AND, OR and NOT.

1. **AND Operation**: This operation is represented by a dot operator or by the absence of any operator. It is a binary operator. The result of expression xy is 1 if and only if both x=1 and y=1. In other cases, 0. Following truth table shows various possible results of AND logic.

AND		
x	y	Z=xy
0	0	0
0	1	0
1	0	0
1	1	1

Fig: Truth table of AND logic

The AND logic may be demonstrated by the following switching circuit.

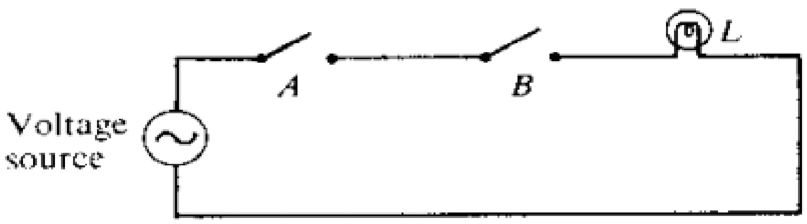


Fig: Switches in series- logic AND

2. **OR Operation**: This operation is represented by + operator. It is a binary operator. The result of expression x+y is 1 if any one of x or y is 1, otherwise 0. Following truth table shows various possible results of OR logic.

OR		
x	y	Z=x+y
0	0	0
0	1	1
1	0	1
1	1	1

Fig: Truth table of OR logic

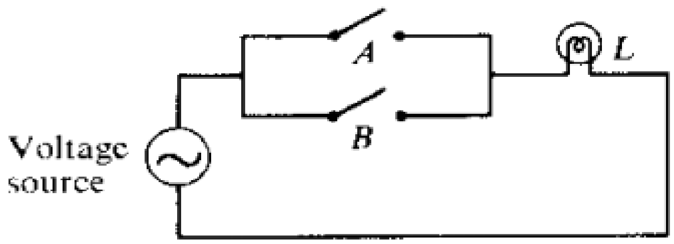


Fig: Switches in Parallel- Logic AND

3. **NOT Operation:** This operation is represented by a prime or a bar symbol above the Boolean variable. It is unary operator. The result of expression x' is 0 if x = 1 and x' = 1 if x=0. That means, the result of x' is opposite of x. following truth table shows the various possible results of NOT logic.

NOT	
x	X'
0	1
1	0

Fig: Truth table of NOT logic

Note: One should not be confused with binary arithmetic and binary logic. You can understand the difference by the fact that, In binary arithmetic, 1+1 = 10 (read as 1 plus 1 is 2, but in binary logic, 1+1 = 1 (read as 1 OR 1 is 1).

Logic Gates:

Electronic digital circuits are also called logic circuits because, with the proper input, they establish logical manipulation paths. Any particular path in the circuit can pass binary information (a single bit of information) with respect to presence or absence of the signal. The digital electronic circuits for AND, OR and NOT logic are called logic gates. Logic gates are the hardware blocks that produce a logic-1 or logic-0 output based on the input logic information. The NOT gate is often called as invertor due to its behavior.

Following are the symbols used for the fundamental logic gates.

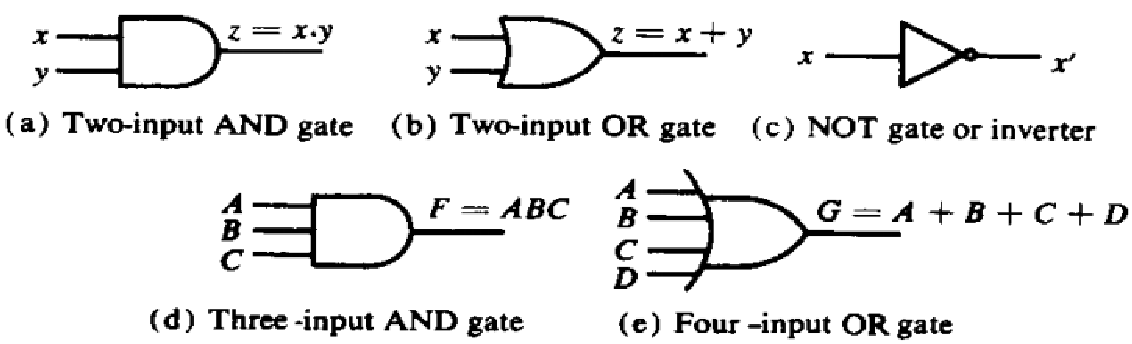


Fig: Fundamental Logic gates

From the above figure, it is clear that, the AND and OR logit gates have two or more inputs and the NOT gate has only one input. All logic gates have only one output.

Unit 2

Boolean Algebra & Logic Gates

Unit 2: Boolean algebra and Logic Gates (5 Hrs.)
Basic and Axiomatic definitions of Boolean algebra, Basic Theorems and properties of Boolean Algebra, Boolean Functions, Logic Operations, Logic Gates, Integrated Circuits

Basic and Axiomatic Definitions of Boolean Algebra

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 bistable 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.

That means, the result of any AND, OR and NOT operation results in {0,1}.

P2→ Commutativity: The • and + operators are commutative

i.e. $x + y = y + x$ and $x \bullet y = y \bullet x$, for all $x, y \in B$

P3→ Distribution: • and + are distributive with respect to one another

i.e. $X \bullet (y + z) = (x \bullet y) + (x \bullet z)$.

$x + (y \bullet z) = (x + y) \bullet (x + z)$, for all $x, y, z \in B$.

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. tyey are 0 and 1.

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:

One variable theorem:

Theorem1: Idempotence (a) x + x = x (b) x.x = x

Theorem2: Existence: 0&1 (a) x + 1 = 1 (b) x.0 = 0

Theorem3: Involution (x')' = x

Two variable theorem:

Theorem4: Associative (a) x + (y + z) = (x + y) + z (b) x(yz) = (xy)z

Theorem5: Demorgan (a) (x + y)' = x'y' (b) (xy)' = x' + y'

Theorem6: Absorption (a) x + xy = x (b) x(x + y) = x

Proofs:

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

$x + x = (x + x) . 1$ (P4: Identity element)
$= (x + x)(x + x')$ (P5: Existence of inverse)
$= x + xx'$ (P3: Distribution)
$=x + 0$ (P5: Existence of inverse)
$=x$ (P4: Identity element)

THEOREM 1(b): x.x=x

$x \cdot x = x$
$= xx + 0$ (P4: Identity element)
$=xx + xx'$ (P5: Existence of inverse)
$=x(x + x')$ (P3: Distribution)
$= x.1$ (P5: Existence of inverse)
$=x$ (P4: Identity element)

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

THEOREM 2(a): x+1=1

$x + 1 = 1\ x + 1$
$= 1\cdot (x + 1)$ (P4: Identity element)
$= (x + x')(x + 1)$ (P5: Existence of inverse)
$=x + x'\cdot 1$ (P3: Distribution)
$= x + x'$ (P4: Identity element)
$= 1$ (P5: Existence of inverse)

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

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.

The theorems involving two or three variables may be proven algebraically from the postulates and the theorems that have already been proven.

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

$x + x' = 1$ 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 $(x + y) \bullet (x'y') = 0$.

$(x + y) + (x'y') = [(x + y) + x'] [(x + y) + y']$ [OR distributes over AND (P3)]
$= [(y + x) + x'] [(x + y) + y']$ [OR is commutative (P2)]
$= [y + (x + x')] [x + (y + y')]$ [OR is associative (Theorem 3(a)), used twice]
$= (y + 1)(x + 1)$ [Complement, x + x' = 1 (P5), twice]

= 1 • 1 [x + 1 = 1, (Theorem 2), twice]
= 1 [Idempotent, x • x = x (Theorem 1)]

Also,

(x + y)(x'y') = (x'y') (x + y) [AND is commutative (P2)]
= [(x'y') x] + [(x'y') y] [AND distributes over OR (P3)]
= [(y'x')x] + [(x'y')y] [AND is commutative (P2)]
= [y'(x'x)] + [x'(y'y)] [AND is associative (Theorem 3(b)), twice]
= [y'(xx')] + [x'(yy')] [AND is commutative, twice]
= *y' • 0+ + *x' • 0+ [Complement, x • x' = 0, twice]
= 0 + 0 [x • 0 = 0, twice] = 0 [Idempotent, x + x = x]

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

Boolean function is an expression formed with:

- Binary variables (each binary variable van take values either 0 or 1).
- Operators (AND, OR, NOT, parenthesis and equality sign).

The resultant value of the Boolean function is either 0 or 1.

For example, consider a Boolean function $F_1 = xyz'$. Here the function F_1 is 0 if x=1, y=1 and z=0, otherwise it is 0.

A Boolean function can be represented either in algebraic form or in the form of truth table. The above Boolean function can be represented in truth table as shown below.

x	**y**	**z**	**z'**	**$F_1 = xyz'$**
0	0	0	1	0
0	0	1	0	0
0	1	0	1	0
0	1	1	0	0
1	0	0	1	0
1	0	1	0	0
1	1	0	1	1
1	1	1	0	0

To represent a Boolean function having n Boolean variables, in truth table, we need 2^n different input values set. As in above Boolean function with 3 variables, there are $2^3 = 8$ different rows in the truth table. Different combinations of 0's and 1's in the rows is easily obtained by going from binary 0 to 2^n-1. For each row, value of the Boolean function is either 0 or 1.

In a Boolean function, different operators are represented differently.

- AND operator is represented by dot '.' Or simply by the absence of any operator.
- OR operator is represented by '+'.
- NOT operator is represented by a dash superscripted in the variable name or simply by a bar above the variable name.
- Parenthesis is simply represented by '()'.

A Boolean function can have multiple representations. In other words, there may be a simple representation equivalent to a complex looking Boolean function. Two Boolean functions are said to be same or equivalent if their truth table is identical. For example, consider the two Boolean functions as

$F_2 = x'y'z + x'yz + xy'$ and $F_3 = xy' + x'z$

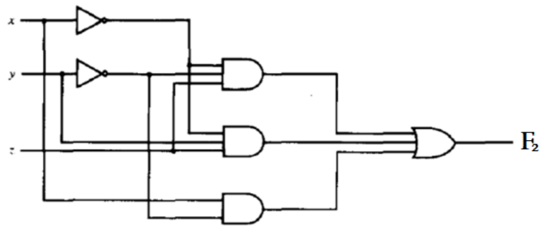
Constructing the truth tables for these Boolean functions;

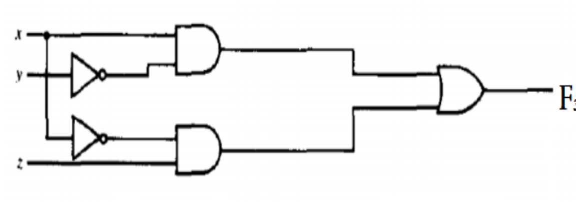
x	**y**	**z**	**x'**	**y'**	**z'**	**$F_2 = x'y'z + x'yz + xy'$**	**$F_3 = xy' + x'z$**
0	0	0	1	1	1	0	0
0	0	1	1	1	0	1	1
0	1	0	1	0	1	0	0
0	1	1	1	0	0	1	1
1	0	0	0	1	1	1	1
1	0	1	0	1	0	1	1
1	1	0	0	0	1	0	0
1	1	1	0	0	0	0	0

From the above truth table, we can say that the Boolean functions F_2 and F_3 are same. We can simplify the Boolean Function to get as simpler expression as possible. This will help to transform a Boolean function to a logic circuit (ie. From logic expression to a logic diagram).

We can transform a logic expression (Boolean expression) into logic diagram by using the AND, OR and NOT logic gates.

We can represent the Boolean functions $F_2 = x'y'z + x'yz + xy'$ and $F_3 = xy' + x'z$ in logic diagram as shown below.





Both of the above logic diagrams represent the same logic, however the logic for F_2 requires more logic gates and is complicated as well in comparison to that of F_3. Thus, it is always important to manipulate the Boolean expression to get a simple form.

Algebraic Manipulation or simplification of the Boolean Functions

Algebraic manipulation or simplification of Boolean functions is important to simplify the corresponding logic circuit and hence reduce the circuit equipment and costs. We can simplify a Boolean expression by reducing the literals (primes or unprimed Boolean variable) and the terms. Each literal in an expression indicates an input to the logic circuit and each term in an Boolean expression represents a logic gate. So, by reducing the literals, we reduce the number of inputs and by reducing the terms, we reduce the number of equipment needed.

The detail about simplifying the Boolean expression will be discussed more in chapter 3. Here, we discuss only the basics of simplifying Boolean expression using algebraic manipulation. By using algebraic manipulation, we can reduce the number of literals.

Unfortunately, there are no specific rules to give the guarantee of getting simplified final answer. The only method available is to cut-and-try following the basic postulates of Boolean algebra and the basic theorems.

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

1. **$x + x'y$**

Ans→ $x + x'y = (x+x')(x+y)$

$= 1.(x+y)$

$= x+y$

(number of literals reduced from 3 to 2)

2. **$x(x'+y)$**

Ans→ $x(x'+y) = xx' + xy$

$= 0 + xy$

$= xy$

(number of literals reduced from 3 to 2 & number of terms reduced from 2 to 1)

3. $x'y'z + x'yz + xy'$

Ans→ $x'y'z + x'yz + xy' = x'z\,(y' + y) + xy'$

$= x'z + xy'$

(number of literals reduced from 8 to 4 & terms reduced from 3 to 2)

4. $xy + x'z + yz$

Ans→ $xy + x'z + yz \quad = xy + x'z + yz(x+x')$

$= xy + x'z + xyz + x'yz$

$= xy(1+z) + x'z\,(1+y)$

$= xy + x'z$

(number of literals reduced from 6 to 4 and terms reduced from 3 to 2)

5. $(x+y)(x'+z)(y+z)$

Ans→ we already proved that, $xy + x'z + yz = xy + x'z$.

From the duality principle, we can write the following expression.

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

Complement of a Boolean function

The complement of a function F is denoted by F' and is obtained by interchanging 0's for 1's and 1's for 0's in the values of F. Complement of a function may be derived by using the De Morgan's theorem.

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, which can be represented by following equations.

- $(x+y)' = x'y'$
- $(x+y+z)' = x'y'z'$
- $(xy)' = (x' + y')$
- $(xyz)' = (x'+y'+z')$

Q. Find the complement of Boolean function $x'yz' + x'y'z$.

Ans→ Here $F = x'yz' + x'y'z$

Complement of F is written as;

F' = (x'yz' + x'y'z)'

Applying De Morgan's law successively, we can write;

F' = (x'yz')' (x'y'z)'

= (x+y'+z)(x+y+z') which is the complement of F.

Q. Find the complement of Boolean function x(y'z' +yz).

Ans→ Here F = x (y'z' +yz)

Complement of F is written as;

F' = [x (y'z' +yz)]'

Applying De Morgan's law successively, we can write;

F' = x'+ (y'z' +yz)'

= x' + (y'z')'.(yz)'

= x' + (y+z)(y'+z') which is the complement of F.

Note: Easiest method to find the complement of a function is;

- Firstly, find the dual of the function.
- And then complement each literal.

For example, for the function F = x'yz' + x'y'z
The dual of F is (x'+y+z')(x'+y'+z)
Now, complementing each literal we get the complement of F as follows;
F' = (x+y'+z)(x+y+z')

Logic Operations:

AND, OR and NOT are the fundamental logic operations. With the combinations of these basic logic operations, we can construct other compound logic operations. It is found that with n binary variables, we can construct 2^{2n} different logic operations or Boolean functions. For 2 binary variables, we can construct 2^{2x2} =16 different logic operations as shown below.

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**

The truth table for above 16 functions with two binary variables is as shown below.

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

he 16 functions listed can be subdivided into three categories:

1. Two functions that produce a constant 0 or 1 (F_0 and F_{15}).
2. Four functions with unary operations: complement and transfer (F_3, F_5, F_{10}, F_{12}).
3. Ten functions with binary operators that define eight different operations: AND, OR, NAND, NOR, exclusive OR, equivalence (F_9), inhibition (F_2 and F_4), and implication.

Operation	Description
Constant Boolean Functions	It is a unary operation. That generates 0 or 1 irrespective of any input (See F_3 and F_5).
Complement Functions	It is a unary operation that generates the output opposite of the input (See F_{10} and F_{12}).
Transfer Functions	It is a unary operation that simply generates the output same as input (See F_3 and F_5).

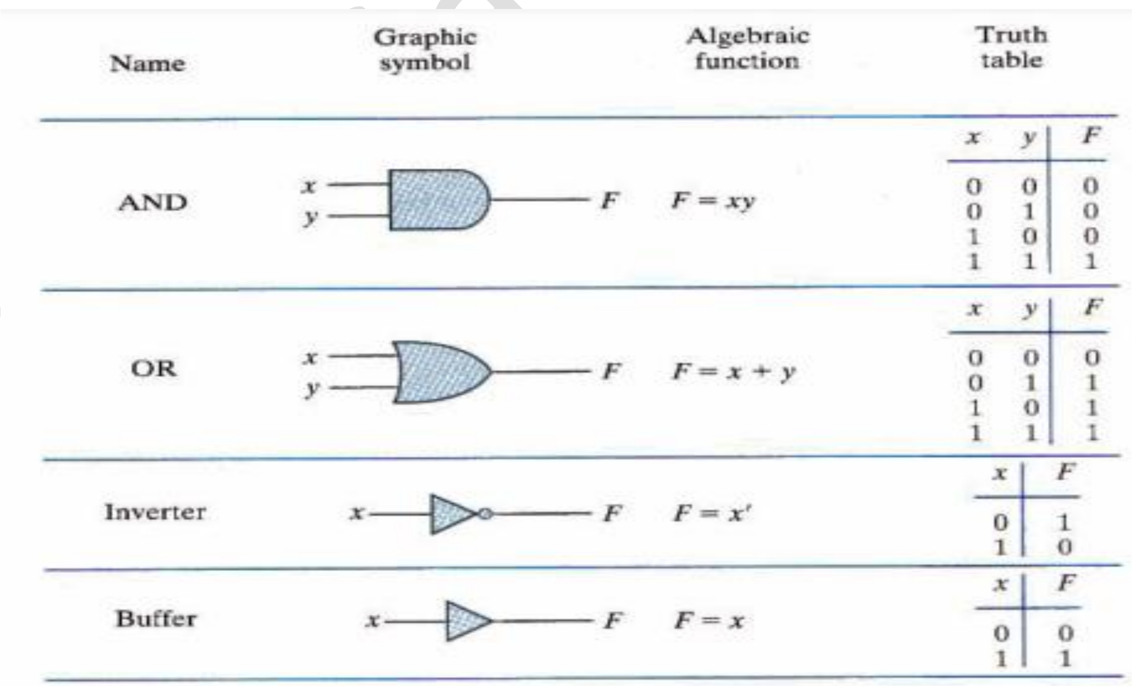
Equivalence	It is a binary operation that generates true if both the inputs are identical, otherwise false. In fact, if there are even number of 0's, the result is 1.
Inhibition and implication	They are the compound Boolean operations used by logician as per requirement. These are not commutative and associative, so they are not practical to use as standard Boolean functions.
AND, OR, NAND, NOR, XOR	They are the binary operations most commonly used in logic design.

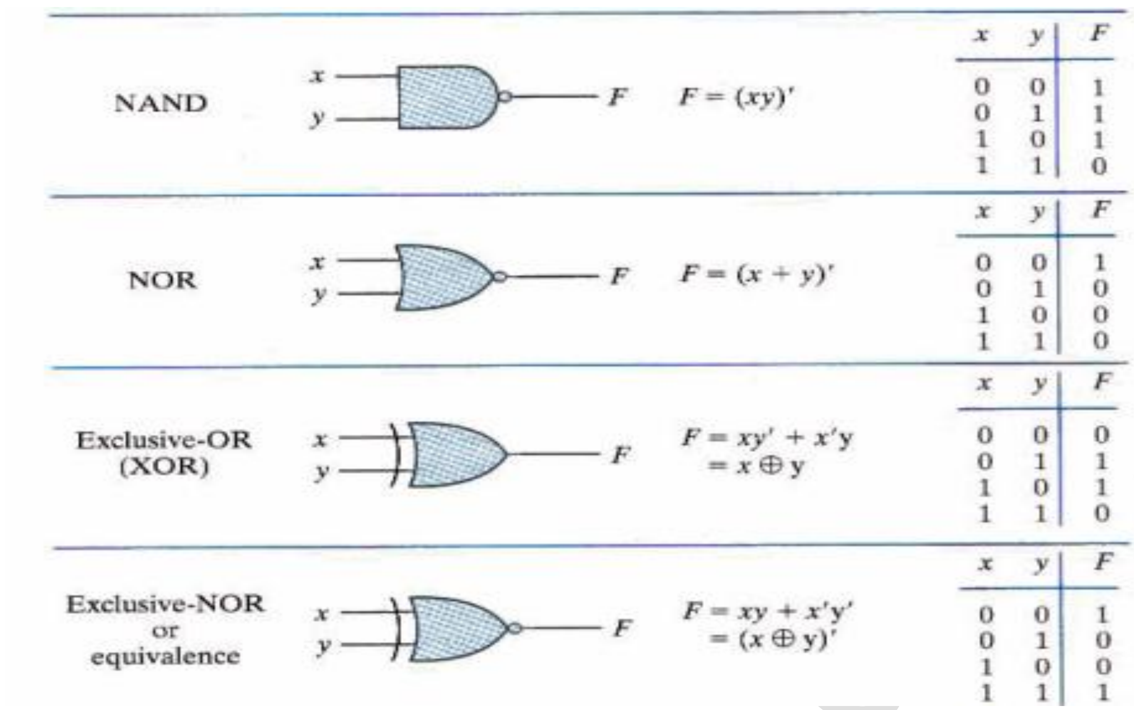
Logic Gates

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.
- 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:





NAND:

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

NOR:

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

Exclusive-OR (XOR):

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

Exclusive-NOR or equivalence:

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

Note:

- In XOR gate, odd no of 1's in the input give 1 as output.
- In Equivalence Boolean operation, even no of 0's in input give 1 as output. It is not standard operation and has no separate logic gate for this operation.

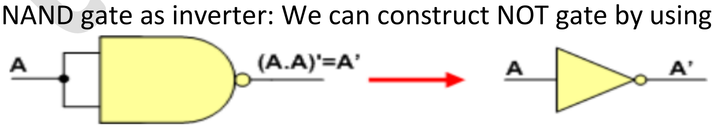
You must be able to draw the logic gate circuit diagram for any given Boolean functions using any of the above logic gates.

NAND and NOR Gates as Universal Logic gates:

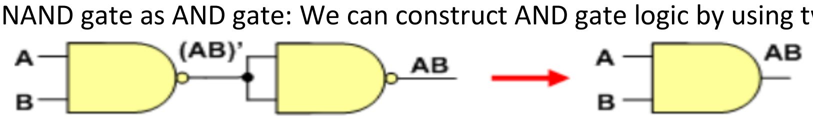
A universal gate is a gate which can implement any Boolean function without need to use any other gate type. The NAND and NOR gates are universal gates. In practice, this is advantageous since NAND and NOR gates are economical and easier to fabricate and are the basic gates used in all IC digital logic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

We can use NAND gate as universal Gate. Since Boolean expression uses AND, OR and NOT logic, to prove that NAND gate can be used as universal gate, we have to show that we can construct AND, OR and NOT logic by using only NAND gate.

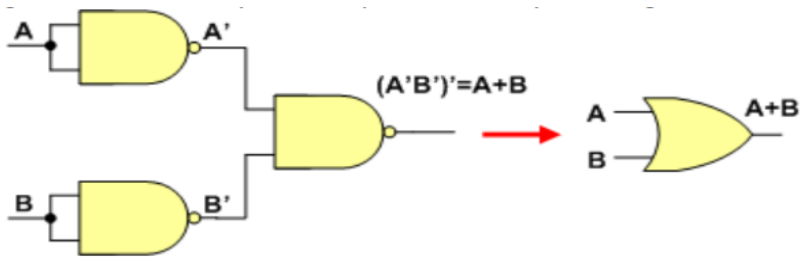
- NAND gate as inverter: We can construct NOT gate by using NAND gate as follows.



- NAND gate as AND gate: We can construct AND gate logic by using two NAND gates as follows.

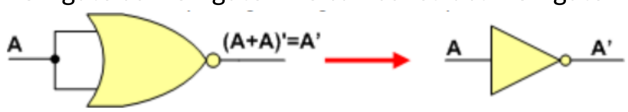


- NAND gate as OR gate: We can construct OR gate logic by using three NAND gates as follows.

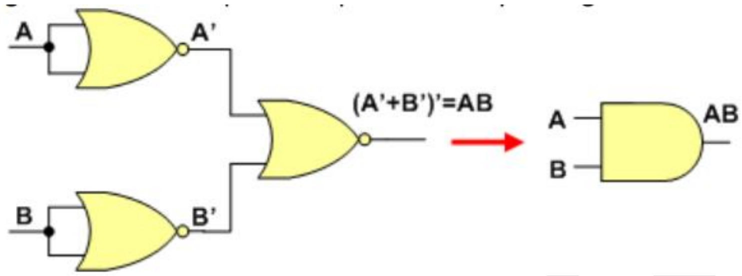


Similarly, NOR gat can also be used as universal gate. We can construct the fundamental AND, OR and NOT gates by using only NOR gate as follows.

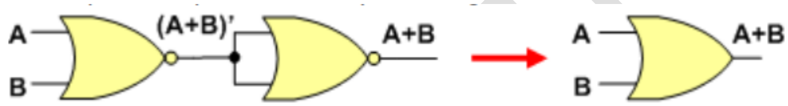
- NOR gate as NOT gate: We can construct NOT gate from a single NOR gate as follows;



- NOR gate as AND gate: We can construct AND gate from three NOR gates as follows;



- NOR gate as OR gate: We can construct AND gate from two NOR gates as follows;



Hence, we can use NAND gate and NOR gates to implement any logical expression into circuits.

Multiple Input Logic Gates:

Single input logic gates like NOT gate and buffers always have single input. The other logic gates can be extended to have more than one inputs. A gate can be extended to have multiple inputs if the binary operation it represents is commutative and associative.

For example, consider an OR gate. 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. Similarly, the AND function also possesses commutative and associative property. Hence, we can have multiple input AND gate and OR gate.

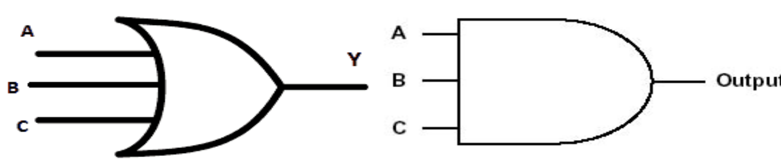


Fig: Three input OR gate and AND gate

The NOR function is commutative ie. $X\downarrow Y\downarrow Z = Z\downarrow Y\downarrow X$ but not associative $X\downarrow(Y\downarrow Z) \neq (X\downarrow Y)\downarrow Z$.

$$X\downarrow(Y\downarrow Z) = [X+(Y+Z)']' = X'(Y+Z) = X'Y + X'Z$$

$$(X\downarrow Y)\downarrow Z = [(X+Y)'+Z]' = (X+Y)Z' = XZ' + YZ'$$

But we can define NOR gate as the complement of OR gate. Then it becomes commutative.

$$X\downarrow Y\downarrow Z = (X+Y+Z)'$$

Similar case (as that of NOR gate) exists for NAND gate also.

Integrated Circuits

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.

Following are the benefits of using IC's.

- 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.

On the basis of the level of integration, the IC's are of following types;

- 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 and DIP IC's

On the basis of fabrication model, IC's may be SIP or DIP.

1. **SIP**: A single in-line package (SIP) 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.

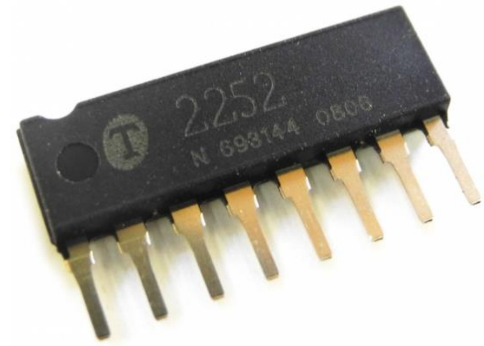


Fig: SIP8 IC

2. **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 DIPn, where n is the total number of pins.

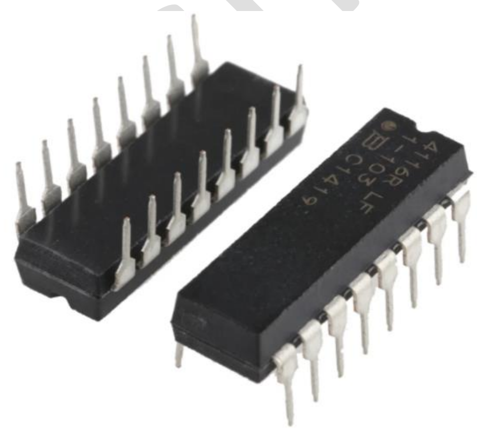


Fig: DIP6 IC

Based on the packaging materials, DIP ICs are of various types:

- 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

SIMM (Single In-line Memory Module) and DIMM (Dual In-line Memory Module)

SIMM stands 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) 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.

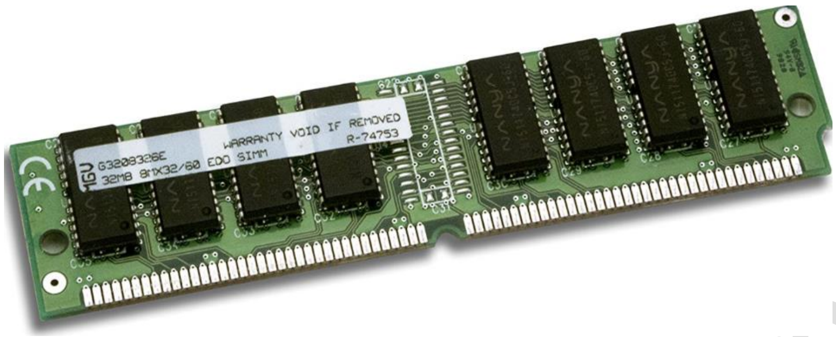


Fig: 32 MB 72 Pin SIMM

DIMM stands 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.

Fig: 512MB 168 pin DIMM

Following are the major differences between SIMM and DIMM:

SIMM	DIMM
SIMM supports 32 bit channel for data transferring.	DIMM supports 64 bit channel for data transferring.
SIMM consumes 5 volts of power.	DIMM consumes 3.3 volts of power.
SIMM provides less storage capacity.	DIMM provides high storage capacity.
The classic or most common pin configuration of the SIMM module is 72 pins.	The foremost common pin configuration of the DIMM module is 168 pins.
SIMMs are the older technology.	DIMMs are the replacement of the DIMMs.
SIMMs are installed in pairs at a time.	DIMMs are installed one at a time.

There is single notch in SIMM.	There are two notches in DIMM.

IC 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. It is ***more common*** in use.

- **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.

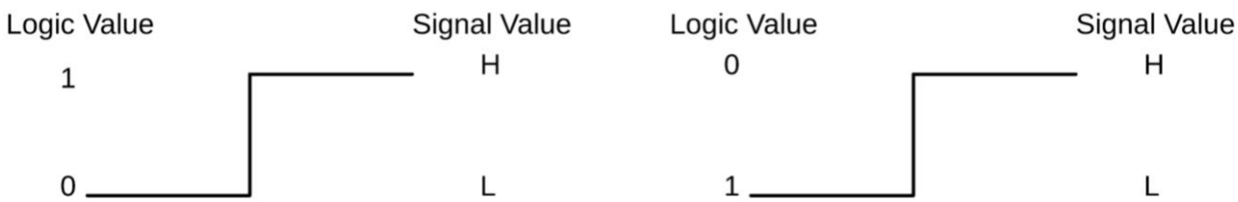
- **MOS (metal-oxide semiconductor):** The metal-oxide semiconductor (MOS) is a unipolar transistor that depends upon the flow of only one type of carrier, which may be electrons (n-channel) or holes (p-channel), this is in contrast to the bipolar transistor used in TTL and ECL gates, where both carriers 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. It is used when ***high component*** density is required.

- **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. 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.

- **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), I2 L has high noise immunity because it operates by current instead of voltage. Sometimes also known as Merged Transistor Logic.

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.

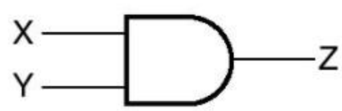


(a) Positive Logic (b) Negative Logic

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

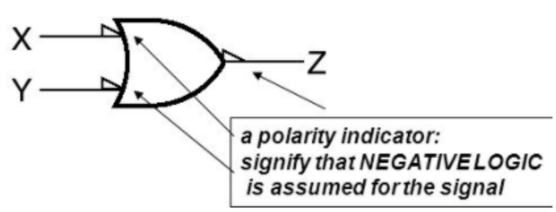
(c) Truth table for positive logic



(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

Special Characteristics of IC logic families

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:

- **Fan Out:** The total number of logic gates that can be powered by the output of a single logic gate is called fan out. It is desired to have an IC with higher Fan Out value.

- **Fan In:** Maximum number of inputs in a logic gate inside an IC is called Fan In.

- **Power Dissipation:** Power consumed by a single logic level. Or, we can also understand it as the power consumed per logic gate. It is desired to have an IC having less power dissipation.

- **Propagation Delay:** It is the time interval between the value of an input is changed and the corresponding result is obtained. It is desired to have less propagation delay for faster ICs.

- **Noise Margin:** It is the amount of voltage that can be increased without affecting the output logic. An IC having high noise margin can tolerate higher voltage fluctuation.

- **Breadth:** The total number of functionalities an IC possesses is called as its breadth. An IC with higher breadth means a versatile IC that can be used for multiple operations.

Following table shows the typical characteristics or parameters of the ICs of different logic families.

IC Logic Family	Fan out	Power Dissipation (mw)	Propagation delay (ns)	Noise Margin (v)
Standard TTL	10	10	10	0.4
Shottky TTL	10	22	3	0.4
Low power Shottky TTL	20	2	10	0.4
ECL	25	25	2	0.2
CMOS	50	0.1	25	3

We can use this table to compare different IC logic families and select an appropriate IC according to the requirement. For example, if we want faster ICs, we may choose one having low propagation delay (ie ECL). If we need ICs having low power consumption, we may choose COMS logic family.