DIGITAL COMPUTER FUNDAMENTALS

Number Systems

Most of the number systems follow the positional notation system to represent any number. The base or radix of a number system is the number of digits that can occur in each position in the number system. For example, the general system, which is basically practiced by us viz., the decimal system has a base or radix value as 10, as the system uses 10 different digits (0, 1, 2, .8, 9) to represent a number in the decimal number system. The binary system has its radix as 2 with the digits as 0 and 1.

In general, any number expressed in a number system that has a radix value r, has co-efficients multiplied by powers of r and can be represented as:

$a_n r^n + a_{n-1} r^{n-1} + a_{n-2} r^{n-2} + \ldots\ldots\ldots a_2 r^2 + a_1 r^1 + a_0 + a_{-1} r^{-1} + a_{-2} r^{-2} + \ldots\ldots\ldots a_{-m} r^{-m}$

The coefficients can have the value within a range 0 to r-1.

Binary Number System

A binary number system consists of two digits namely, 0 and 1. The base for the binary number system is 2. As in the case of decimal number system, positional notation is used in the binary number system, with powers of two. The value of a binary number can therefore be represented as

$a_n 2_n + a_{n-1} 2^{n-1} + a_{n-2} 2^{n-2} + \ldots\ldots + a_2 2^2 + a_1 2^1 + a_0 + a_{-1} 2^{-1} + a_{-2} 2^{-2} + a_{-m} 2^{-m}$

where a_i is either 0 or 1, n is the number of digits that occurs to the left of the binary point and m is the number of digits that occurs to the right of the binary point.

For example, the binary number $(10110)_2$ is equivalent to 22 in the decimal system. This can be illustrated as follows:

$(10110)_2 = 1 * 2^4 + 0 * 2^3 + 1 * 2^2 + 1 * 2^1 + 0 * 2^0$

$= 16 + 0 + 4 + 2 + 0$

$= (22)_{10}$

Consider another example using binary fraction:

$(1.101)_2 = 1 * 2^0 + 1 * 2^{-1} + 0 * 2^{-2} + 1 * 2^{-3}$

$= \quad 1 + 0.5 + 0 + 0.125$

$= \quad (1.625)_{10}$

Octal Number System

As in the case of decimal number system, positional notation is used in the octal number system, with powers of eight. The value of an octal number can therefore be represented as

$a_n8^n + a_{n-1}8^{n-1} + a_{n-2}8^{n-2} + \ldots\ldots + a_28^2 + a_18^1 + a_0 + a_{-1}8^{-1} + a_{-2}8^{-2} + a_{-m}8^{-m}$

where a_i can take any value ranging from 0 through 7, n is the number of digits that occurs to the left of the octal point and m is the number of digits that occurs to the right of the octal point.

For example, the octal number $(134)_8$ is equivalent to 92 in the decimal system. This can be illustrated as follows:

$(134)_8 \quad = \quad 1 * 8^2 + 3 * 8^1 + 4 * 8^0$

$= \quad 64 + 24 + 4$

$= \quad (92)_{10}$

Hexadecimal Number System

As in the case of decimal number system, positional notation is used in the hexadecimal number system, with powers of sixteen. The value of a hexadecimal number can therefore be represented as

$a_n16^n + a_{n-1}16^{n-1} + a_{n-2}16^{n-2} + \ldots\ldots + a_216^2 + a_116^1 + a_0 + a_{-1}16^{-1} + a_{-2}16^{-2} + a_{-m}16^{-m}$

where a_i can take any value ranging from 0 through 9 and A through F, n is the number of digits that occurs to the left of the hexadecimal point and m is the number of digits that occurs to the right of the hexadecimal point.

For example, the hexadecimal number $(1B5)_{16}$ is equivalent to 437 in the decimal system. This can be illustrated as follows:

$(1B5)_{16} \quad = \quad 1 * 16^2 + B * 16^1 + 5 * 16^0$

$= \quad 256 + 176 + 5$

$= \quad (437)_{10}$

Note that the decimal value of B is 11.

Consider another example using hexadecimal fraction:

$(1.A2)_{16}$ = $1 * 16^0 + A * 16^{-1} + 2 * 16^{-2}$

= $1 + 10 * 0.0625 + 2 * 0.00390625$

= $(1.6328125)_{10}$

Number System Conversions

A value in any number system can be converted into its corresponding value in any other number system. For example, a number in the decimal system can be represented in binary, octal or hexadecimal number system. The conversion from decimal to binary or to any other base –r- system is done as follows: the given decimal number is separated into an integer part and the fraction part and then the conversion for each part is carried individually using the following rules:

(i) Decimal to other number systems

- When converting from decimal to binary repeatedly divide the value by 2 and accumulate the remainders until the dividend is less than 2.
- When converting from decimal to octal repeatedly divide the value by 8 and accumulate the remainders until the dividend is less than 8.
- When converting from decimal to hexadecimal repeatedly divide the value by 16 and accumulate the remainders until the dividend is less than 16.

(ii) Binary to octal or hexadecimal number systems

- When converting from binary to octal, arrange the binary numbers in group of three from LSB (Left Significant Bit) and substitute the octal value.
- When converting from binary to hexadecimal, arrange the binary numbers in group of four from LSB (Left Significant Bit) and substitute the hexadecimal value.

(iii) Other number systems to Decimal number system

- When converting from binary to decimal multiply each binary digit with 2^p (where p is the positional value) and sum all the products. (Note: the positional value for the right most bit is 0).
- When converting from octal to decimal multiply each octal digit with 8^p (where p is the positional value) and sum all the products. (Note: the positional value for the right most bit is 0).
- When converting from hexadecimal to decimal multiply each binary digit with 16^p (where p is the positional value) and sum all the products. (Note: the positional value for the right most bit is 0)

(iv) Octal, Hexadecimal to binary number system

- When converting an octal number to binary, substitute the binary value for each octal digit (3 bits).
- When converting a hexadecimal number to binary, substitute the binary value for each hexadecimal digit (4 bits).

(v) Octal to Hexadecimal and vice versa

- When converting an octal number to hexadecimal, first convert the octal number to binary equivalent and then arrange the binary numbers in groups of four and substitute the hexadecimal value for each group of 4 bits.
- When converting a hexadecimal to octal number, first convert the hexadecimal number to binary equivalent and then arrange the binary numbers in groups of three and substitute the hexadecimal value for each group of 3 bits.

Binary Arithmetic

Like decimal number system, we can perform four types of operations in binary arithmetic, namely, addition, subtraction, multiplication and division. The operations can be performed for both signed numbers and unsigned numbers. In the case of unsigned numbers the sign of the operands will be similar (i.e., either both positive or both negative.). In which case, the sign of the numbers will be ignored and the result will be taken into account.

Negative Numbers

In number representation system, the sign bit assumes 0 for a positive number and a 1 for a negative number. These set of switches are commonly known as registers. There is a set of seven to eight registers in the computer. Each register is used to store a number. The left-most bit of the register is used to represent the sign of the number while the remaining bits are used to store the magnitude of the number.

One of the methods to represent negative number is signed-bit magnitude. In this method, the magnitude of the positive number is stored in a normal form, whereas the magnitude of the negative number is stored by introducing a 1 in the MSB called signed bit position. For a positive number, a 0 is introduced in the MSB.

The sign bit is placed at the leftmost position (MSB). (Underlined in the above example.) However, the sign bit representation has a lot of disadvantages like having two representations for 0 (zero) and the cost of hardware to store the extra bit (sign bit) will increase.

Complements

Given a non-negative number X with a radix r, the X' is the complement of X with respect to r called r's complement of X, where r=X+X'. The integer X is said to be in true form and the integer X' is said to be in the complement form. Complements make use of a single fundamental circuit for performing operations like addition and subtraction. There are two types of complements, namely, (r-1)'s complement and r's complement.

The (r-1)'s complement: The (r-1)'s complement of an integer in a radix r is obtained by subtracting the number from (r-1). For example, if we want to find the 9's complement of 8, then we should subtract 8 from 9 (r-1). In decimal system, if we want to find the (r-1)'s complement, simply subtract each digit by 9. Thus 9's complement of 8 = 9 - 8 = 1. In general the (r-1)'s complement can be generalized by the formula $(r^n - 1) - P$, where n is the total number of digits in positive value P.

The r's complement: The r's complement of positive value P in a radix r is obtained by using the formula r^n –P, where n is the total number of digits in P. For example, if we want to find the 10's complement of 8, then we should subtract 8 from 10 (r).

Binary Subtraction using r's Complement

The first and foremost application of complements is to deal with negative numbers and perform binary subtraction. Let P1 and P2 be two numbers with the radix r, then to subtract P2 from P1, the following procedure should be followed:

1. Obtain r's complement of the subtrahend (P2)
2. Add the Minuend (P1) with r's complement of P2
3. If the sum produces an end carry discard it and declare the result else find the r's complement of the sum and declare the result as negative.

Note: When we want to find the difference between two numbers of m and n bits (where m > or < or = n) then the number of bits in the difference should not exceed the size of m or n whichever is maximum. If it exceeds, then the exceeded bit should be discarded.

Binary Subtraction using (r-1)'s complement

Let P1 and P2 be two numbers with the radix r, then to subtract P2 from P1, the following procedure should be followed:

1. Obtain (r-1)'s complement of the subtrahend (P2)
2. Add the Minuend (P1) with (r-1)'s complement of P2
3. If the sum produces an end carry add 1 to the LSB, called end around carry and declare the result else find the (r-1)'s complement of the sum and declare the result as negative.

Representing Negative Numbers using Complements

We already discussed as how to represent negative numbers using signed- bit magnitude and also the disadvantages of the method. Now, we shall discuss about using complements to represent negative numbers. The negative numbers are represented by complementing the numbers to enhance operations like addition, subtraction, logical manipulations, etc to be carried in an easier and efficient manner using the circuitry alone.

Binary Codes

When dealing with large quantities of data, it is desirable to encode the data instead of converting them into binary form. The binary codes can be classified broadly into the following classes:

Weighted Codes: In these types of codes, the decimal value of a code is obtained by summing up the positional values, i.e. % w(i)b(i), where w represents the weights (positional value) and b represents binary value either 0 or 1.
8421 code is thus known as Natural Binary Coded Decimal (NBCD) or simply, BCD (Binary Coded Decimal).

Non-Weighted Codes: These codes are contrary to the weighted codes. In the case of weighted codes, each bit position is designed with a weight but the non-weighted codes do not designate any weight with respect to the bit position. An example of non-weighted code is excess-3 code. The excess-3 code can be framed by adding 3 to the BCD code.

Excess-3 Arithmetic: In the case of Excess-3 code, the binary subtraction is carried in the same manner as was done with 8421 code. In an excess-3 code, we can employ 1's complement and 2's complement for subtraction.

Self-Complementing Codes: In a 4-bit construction if we interchange 0's and 1's in a digit d, we will obtain the value equivalent to (9-d), and such a code is called self-complementing code. In simple terms the 9's complement of a decimal can be obtained by interchanging 0's and 1's. The familiar weighted self-complementing codes are 2421, 84-2-14 and the non-weighted self-complementing code is excess-3 code.

Cyclic, Gray or Reflected Code:
A common type of ADC is the shaft encoder, which directly converts a physical position into a digital value. Experience proves that there are many possibilities of producing wrong output by the shaft encoder due to misalignment, wearing out of brushes, etc. Hence a

coding technique was developed in such a manner that only one bit varies from one code to the next. Such a code is called cyclic code or gray code or a reflected code.

Error Detection and Correction Codes

During transmission, these signals are distorted and disturbed by noise. The infinite set of the received signals is then converted back to 0s and 1s by the demodulator using a decision rule. These decisions, however, are not free of errors. The probability of error would certainly be reduced if the transmitted power or the duration of the signals were increased. These methods are not used because neither poor efficiency nor low transfer rate is desirable. Fortunately, there is a procedure called error control coding to keep the probability of transmission errors at an acceptably low level.

Single Parity-Check Code: The simplest error detection code is the single parity-check code. The code is generated in such a way that the message is extended by a single parity-check digit the value of which depends on the bits of the message. Parity-check can be chosen as even or odd. Even-parity check is defined as modulo 2 sum of the elements while odd-parity check is its inverse, i.e. parity-check code is C(n,n-1).

Parity bit: The parity checking uses an extra bit called a parity bit or parity-check bit along with each code group. The parity bit checking can fall under even parity or odd parity depending upon the number of 1's in the group. For instance, if the data to be transmitted is 0111 (equivalent to 7) in a coding scheme, we need to add an extra bit (parity bit) to the four-bit group, the fifth bit that is introduced may be a 1 or 0 depending upon the type of parity-checking system. The two types of parity arte odd-parity and even-parity.

Hamming Code: Hamming code is an error correction code. In this case the addition of the parity bit creates a minimum Hamming distance of two between any two codes. Hamming displayed that by introducing more number of bits in the codes, it is possible to locate the position of the error along with error detection. Increasing the number of parity bits will enable to identify more than one error and also correct them. Hamming code between two successive code groups in a cyclic code is unity.

Alphanumeric Codes: As illustrated earlier, all digital computers should possess the capability of manipulating alphabets and special characters along with digital data. The digital computer should be able to recognize and process alphabets (both uppercase and lowercase), numeric characters, arithmetic operators and other special characters like ? , “ ‘ % & # ! ~ ( ) _ { } [ ] etc.

The ASCII code: The **A**merican **S**tandard **C**ode for **I**nformation **I**nterchange [or as-key] was invented 50 years ago as a means of displaying text on a computer screen. This was accomplished by assigning a number to each letter and the using a look up table to convert the number into a screen display. The capital letter A, for instance, corresponds to 65. (B=66). The ASCII code was a very compact convention and was available on every computer platform.

The EBCDIC Code: EBCDIC is an acronym for Extended Binary Coded Decimal Interchange Code. EBCDIC is a character encoding set developed by IBM. EBCDIC uses the full 8 bits available to it, so parity checking cannot be used on a 8-bit system.

Boolean Algebra

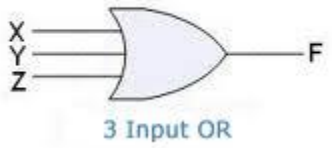
George Boole (1815- 1864), Mathematician and logistician developed ways of expressing logical processes using algebraic symbols, creating a branch of mathematics known as symbolic logic.

In 1847 he published The Mathematical Analysis of Logic, a short volume that first introduced Boole’s early ideas on symbolic logic to the world. The publication demonstrated that logic, as presented and verbalized by Aristotle, could be rendered as algebraic equations.

Boolean algebra differs in a major way from ordinary algebra in that Boolean constants and variables are allowed to have only two possible values, 0 or 1. Boolean 0 and 1 do not represent actual numbers but instead represent the state of a voltage variable, or what is called its logic level.

Truth Table: A truth table is a means for describing how a logic circuit's output depends on the logic levels present at the circuit's inputs.

Logic Addition (OR Operation): If any one of the conditions is True, then the result is True. If all the conditions fail, then the result is False. In the above truth table, the value 0 specifies False and the value 1 specifies True. Note that with the OR operation 1+1=1 (specifying the result is TRUE). The OR operation requires a minimum of two input variables and can be constructed with any number of variables.



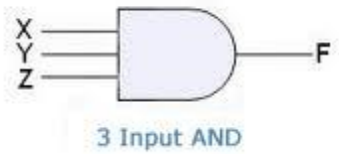
3 Input OR

The truth table for the three inputs OR gate is as follows:

A	B	C	X=A+B+B
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

Logical Multiplication (AND Operation): If and only if all the conditions are True, the result is True. If any one of the conditions fails, then the result is False. In the above truth table, the value 0 specifies False and the value 1 specifies True. Note that with the AND operation is similar to the Multiplication operation. The AND operation requires a minimum of two input variables and can be constructed with any number of variables.

An example of three input OR gate and its truth table is as follows:



The truth table for the three inputs AND gate is as follows:

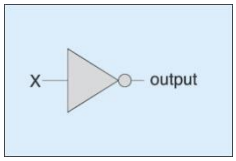
A	B	C	X=A+B+B
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

Inversion or Complementation (NOT Operation): Unlike AND and OR operations, the NOT operation can be performed only on a single input variable. For example, if the variable A is subjected to the NOT operation, the result x can be expressed as

X=A'

Where the prime (') represents the NOT operation. This expression is read as x equals NOT A (or) x equals the inverse of A (or) x equals the complement of A. The other methods of representing A' (A complement are ~A, ┐A and Ā.

Each of these is in common usage and all indicate that the logic value of x=A' is opposite to the logic value of A.



The Truth table of the NOT operation is as follows:

A	X=A'

0	1
1	0

The NOT operation reverses the truth of the condition. Suppose, if the input is 0, the result is complemented to1 and if the input is 1, the result is complemented to 0. Thus,

0'=1 and

Derived Gates: The gates that can be derived from the basic gates are called derived gates. NOR, XOR, XNOR are some of the examples of derived gates.

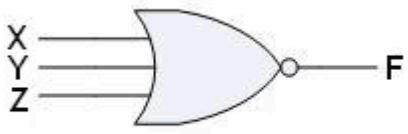
NOR Operation: The NOR operation is the complement of OR operation and is also called not-OR. The NOR is same as the OR gate symbol except that it has a small circle on the output. This small circle represents the inversion operation. Therefore the output expression of the two-input NOR gate is:

X = (A+B)'

The Truth table for the NOR operation is given in the following table.

A	**B**	**A+B**	**X=(A+B)'**
0	0	0	1
0	1	1	0
1	0	1	0
1	1	1	0

The following figure shows NOR gate that can be constructed using a OR gate and a NOT gate.



3 Input NOR

The Truth table for the above equation is as follows:

A	**B**	**C**	**A+B+C**	**(A+B+C)'**	**X=((A+B+C)')'**

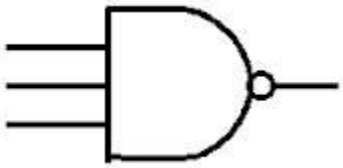
0	0	0	0	1	0
0	0	1	1	0	1
0	1	0	1	0	1
0	1	1	1	0	1
1	0	0	1	0	1
1	0	1	1	0	1
1	1	0	1	0	1
1	1	1	1	0	1

From the above truth table we can see that A+B+C=((A+B+C)')'

NAND Operation: NAND is the same as the AND gate symbol except that it has a small circle on the output. This small circle represents the inversion operation. Therefore the output expression of the two input NAND gate is:

X=(AB)'

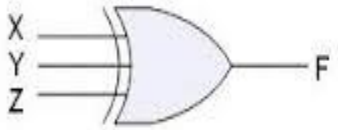
The following figure illustrates two inputs NAND gate.



The Truth table for the two inputs NAND gate is given below:

A	**B**	**A.B**	**X=(A.B)'**
0	0	0	1
0	1	0	1
1	0	0	1
1	1	1	0

XOR Operation: This operation is also called Exclusive-OR operation. The XOR gate has a graphic symbol similar to that of the OR gate with an additional curve line on the input side. The following diagram illustrates the XOR gate:



3 Input XOR

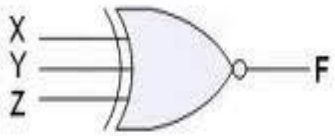
The Truth table for the XOR operation is given below:

A	B	AB'+A'B
0	0	0
0	1	1
1	0	1
1	1	0

The symbol A ⊕ B is read as A XOR B, which is equivalent to writing AB'+A'B. From the above truth table is clear that if all the values are true or if all the values are false then the result is false, otherwise it is true, that is, the output is low when both the inputs are high or when both inputs A and B are low.

XNOR Operation: The XNOR operation is the complement of XOR operation. XNOR operation is also called exclusive-NOR operation. The X NOR operation otherwise known as equivalence. The small circle on the output side of the graphic symbol indicates the inversion of XOR.

The following figure illustrates XNOR gate.



3 Input XNOR

The Truth table for the XNOR operation is as follows:

A	B	A+B	AB+A'B'
0	0	0	1

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

The symbol A ⊙ B is read as A XNOR B, which is equivalent to writing AB+A'B'. From the above truth table it is clear that if all the values are true or if all the values are false then the result is true, otherwise it is false. The output is high when both inputs A and B are high and when neither A nor B is high.

Hierarchy of Logic Circuits: While evaluating the logical circuits output, the following rules with respect to the hierarchy of operation should be followed:

1. Parentheses have the highest priority, i.e. all terms within the parentheses are evaluated first.
2. Inversions have next highest priority over other operations, i.e. if a single term with a 0 is inverted it becomes 1 and if value 1 is inverted it becomes 0.
3. AND operations are evaluated.
4. OR operations are evaluated after AND operations.
5. If an expression has a bar over it, the operation should be performed first and then the result should be inverted next.

Evaluating Logic circuits: Any logic circuit, no matter how complex may be completely described using Boolean operations, because the OR gate, AND gate, and NOT gate circuit are the basic building blocks of digital systems. The output logic level of a Boolean expression can be determined for any set of input levels.

Boolean Theorems and Postulates:

Boolean algebra is a systematic treatment of logic developed by George Boole. In 1904 E.V. Huntington formulated postulates, which are basic assumptions form which it is possible to deduce the rules, theorems and properties for an algebraic structure. These postulates are popularly called, Huntington Postulates. In 1938 C.E. Shanon introduced a

two-valued Boolean algebra called Switching algebra. Boolean algebra is an algebraic structure defined on a Set of elements S with two Binary operators '+' and '.'.

Boolean Functions

A Boolean function (or Boolean expression) consists of one or more binary variables that can take the value either 0 or 1. If more than one variable is used then the variables are combined by the basic logical operators AND, OR and NOT. The operators AND and OR are binary operators where as the NOT operator is a unary operator.

A Boolean function may be represented using a truth table. A truth table shows all possible combinations of the input variables (i.e. 2n combinations of n input variables). However, we will substitute the output of those combinations that is been specified in the Boolean function.

A Boolean function may be transformed from an algebraic expression into a logical circuit called the combinational circuit consisting of the AND, OR and the NOT gates to realize the functions. A term in the Boolean function is implemented using the AND gates and the OR gates is used to combine two or more terms. The NOT gate is essential if complementing a variable is required. Any given Boolean function has to be reduced further to produce simplified expressions for easy manipulation of the logical circuits. This can be accomplished using many methods, which will be explained in the following sections.

Duality Principle: One of the important properties of Boolean algebra is the duality principle, which states that every algebraic expression deducible from the postulates of Boolean algebra holds true if the operators and the identity elements are interchanged. Using this principle, we can simplify interchange OR (+) and AND (.) operators and replace 1s by 0s and 0s by 1s, the variables and complements are left unchanged in the function.

Complements

The complement of a function F, denoted as F' is obtained by interchanging the zeros for ones and ones for zeros, interchanging the operators '+' with '.' And '.' With '+' and

complementing all the variables in the function F. DeMorgan's theorems can be applied to complement a Boolean function.

Considering the above example, with the given function

F=(A+B).(C'+D')

The dual of the function is derived as:

F=(A.B)+(C'.D')

And the complement of the function would be

F=A'B'+CD.

Law of Involution

1. (X')' = X

X	X'	(X')'
0	1	0
1	0	1

Consensus Laws

2. XY + X'Z + YZ = XY +X'Z

X	Y	Z	X'	XY	X'Z	YZ	XY+X'Z+YZ	XY+X'Z
0	0	0	1	0	0	0	0	0
0	0	1	1	0	1	0	1	1
0	1	0	1	0	0	0	0	0
0	1	1	1	0	1	1	1	1
1	0	0	0	0	0	0	0	0
1	0	1	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0
1	1	1	0	1	0	1	1	1

From the above truth table, it is clear that XY + X'Z + YZ = XY +X'Z

3. (X+Y) (X'+Z)(Y+Z) = (X+Y).(X'+Z)

DeMorgan's Theorems

The following two rules are known as DeMorgan's theorems. Using DeMorgan's theorems, the complement of any Boolean expression, or part of any expression can be found.

1. The + symbols (OR expression) are replaced with . symbols (AND expression) and . symbols with + symbols.
2. Each of the terms in the expression is complemented.

Two Variable DeMorgan's Theorems:

4. (X + Y)' = X'Y'
5. (XY)' = X' + Y'

Proof: The above two theorems can be proved using truth tables

X	Y	X+Y	(X+Y)'	X'	Y'	X'Y'
0	0	0	1	1	1	1
0	1	1	0	1	0	0
1	0	1	0	0	1	0
1	1	1	0	0	0	0

DeMorgan's theorems are extremely useful in simplifying expressions in which a

Three variables DeMorgan's Theorem

6. (X + Y + Z)' = X'Y'Z'
7. (XYZ)' = X' + Y' + Z'

Summary of Boolean Theorems	
S.No	**Classification & Algebraic Definition**
	I. Single Variable Theorems
	a. laws of Intersection
1.	X * 0 = 0

2.	X * 1 = X
3.	X * X = X (Idempotent law or Tautology)
4.	X * X' = 0 (Complement Law)
	b. Laws of Union
5.	X + 0 = X
6.	X + 1 = 1
7.	X + X = X (Idempotent law or Tautology)
8.	X + X' = 1 (Complement law)
	II. Multi-variable theorems
	c. Commutative laws
9.	X + Y = Y + X
10.	X.Y = Y.X
	d. Associative laws
11.	X + (Y+Z) = (X+Y) + Z = X + Y + Z
12.	X(YZ)= (XY)Z = XYZ
	e. Distributive laws
13.	X+YZ=(X+Y).(X+Z)
14.	W+X)(Y+Z) = WY+WZ+XY+XZ
15.	X(Y+Z)= XY + XZ
	f. Reverse Distributive laws
16.	XY+YZ+WY=Y(W+X+Z)
17.	XYZ+WXY=XY(W+Z)
	g. Transposition laws
18.	XY+X'Z = (X'+Y).(X+Z)
19.	(X+Y).(X'+Z)=X'Y+XZ
	h. Absorption laws
20.	X(X+Y)=X
21.	X+X.Y=X
22.	X(X'+Y) =X.Y
23.	X.Y + Y' = X+ Y'

24.	X.Y' + Y = X + Y
	i. Involution law
25.	(X')' = X
	j. Consensus laws
26.	XY + X'Z + YZ = XY +X'Z
27.	(X+Y) (X'+Z)(Y+Z) = (X+Y).(X'+Z)
	k. DeMorgan's Theorem
	i. Two Variables DeMorgan's Theorem
28.	(X + Y)' = X'Y'
29.	(XY)' = X' + Y'
	ii. Three Variables deMorgan's Theorem
30.	(X + Y + Z)' = X'Y'Z'
31.	(XYZ)' = X' + Y' + Z'

Universality of NAND Gate: It is possible to implement to implement any logic expression using only NAND gates without using any other gate. This is because NAND gates, in the proper combination can be used to represent each of the basic Boolean operators NOT, AND and OR.

Universality of NOR Gate: It is possible to implement to implement any logic expression using only NOR gates without using any other gate. This is because NOR gates, in the proper combination can be used to represent each of the basic Boolean operators NOT, AND and OR.

Binary Adders & Subtractor:

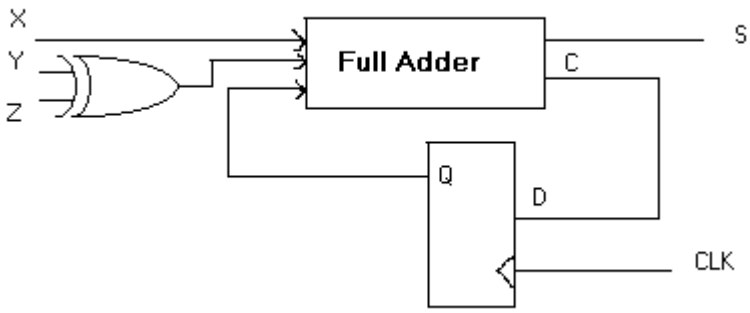
A binary adder is the circuit that generates the arithmetic sum of two binary numbers of any length.

The subtraction of binary numbers can be done most conveniently by means of complements of numbers.

Subtraction A - B can be done by taking the 2's complement of B and adding it to A.
The 2's complement can be obtained by taking the 1's complement and adding 1 to the least significant pair of bits.
The 1's complement can be implemented with inverters and a 1 can be added to the sum through the input carry.

The addition and subtraction operations can be combined into one common circuit by including an exclusive-OR (XOR) gate with a full-adder.

A bit sequential adder-subtractor is shown in the following figure:dders and Subtractors



Input X is used to represent the bit of the binary number A.
Input Y is used to represent the bit of the binary number B.
The control signal Z controls the type of operation.

When Z = 0 the circuit is an adder, meanwhile, the D flip-flop
should be initialized to 0.
When Z = 1 the circuit becomes a subtractor. The D flip-flop
should be initialized to 1.

The XOR gate receives input of signal bit Z and the input of the operand bit Y.

When Z = 0, we have Y XOR Z = Y.

The full adder receives the value of Y, the input carry is 0 because
D flip-flop was initialized to 0, and the circuit performs A plus B.

When Z = 1, we have Y XOR Z = Y' and the initial carry to 1.

The Y input is complemented and a 1 is added through the initial input carry.
The circuit performs the operation: A plus the 2's complement of B.

For unsigned numbers A and B, this gives the A - B if A >= B or
2's complement of (B - A) if A < B.
For signed numbers, the result is A - B provided there is no overflow.