

# **DEPARTMENT OF ELECTRICAL ENGINEERING**

## **DIGITAL CIRCUITS AND DESIGN (4 credit)**

**Course Code: BEC1405**

**(5<sup>TH</sup> SEMESTER)**

### **SYLLABUS**

#### **DIGITAL CIRCUITS AND DESIGN (3-1-0)**

##### **MODULE-I**

**Number system & codes: Binary Number base conversion, Octal & hexadecimal**

**numbers, complements, signed binary numbers, binary codes-BCD codes, gray codes, ASCII**

**Character Code, Codes for serial data transmission & storage.**

**Boolean Algebra & Logic gates: Axiomatic definition of boolean Algebra .Property of Boolean**

**Algebra, boolean functions, Canonical & standard form; min terms & max terms, standard forms; Digital Logic Gates, Multiple inputs.**

##### **MODULE-II**

**Gate level Minimization: The Map Method, K Map up to five variables, Product of Sum simplification, Sum of Product simplification, Don't care conditions. NAND and NOR**

**Implementation, AND-OR inverter, OR-AND inverter implementation, Ex-OR Function, parity generation & checking, Hardware Description Language (HDL).**

**Combinational Logic: Combinational Circuits, Analysis & Design procedure; Binary Adder-subtractor, Decimal Adder, Binary Multiplier, Magnitude comparator, Multiplexers and demultiplexers, Decoders, Encoders, Multipliers, Combinational Circuits design**

### **MODULE-III**

**Synchronous Sequential logic: Sequential Circuit, latches, Flip-flop, Analysis of Clocked Sequential circuits, HDL for Sequential Circuits, State Reduction & Assignment, Design procedure. Register & Counters: Shift Register, Ripple Counters, Synchronous Counter, Asynchronous Counter,**

**Ring Counters, Module-n Counters, HDL for Register & Counters .**

### **MODULE-IV**

**Memory & Programmable logic: Random Access Memory (RAM), Memory , Decoding, Error detection & correction, Read only Memory, Programmable logic array , Sequential Programmable Devices.**

**Register Transfer levels: Register transfer level notion, Register transfer level in HDL, Algorithm, State machine, Design Example, . HDL Description of Design, Examples, Binary Multiplier, HDL Description,**

**Digital Integrated logic Circuits: RTL, DTL, TTL, ECL, MOS & C-MOS Logic circuits, . Switch level modeling with HDL**

### **BOOKS**

**[1]. Digital Design, 3rd edition by M. Morris Mano, Pearson Education**

**[2]. Digital Design-Principle & practice, 3rd edition by John F. Wakerley, Pears**

## **Disclaimer**

**This document does not claim any originality and cannot be used as a substitute for prescribed textbooks. The information presented here is merely a collection by the committee members for their respective teaching assignments. Various sources as mentioned at the end of the document as well as freely available material from internet were consulted for preparing this document. The ownership of the information lies with the respective authors or institutions. Further, this document is not intended to be used for commercial purpose and the committee members are not accountable for any issues, legal or otherwise, arising out of use of this document. The committee members make no representations or warranties with respect to the accuracy or completeness of the contents of this document and specifically disclaim any implied warranties of merchantability or fitness for a particular purpose. The committee members shall be liable for any loss of profit or any other commercial damages, including but not limited to special, incidental, consequential, or other damages.**

# **MODULE-I**

## **NUMBER SYSTEMS**

Many number systems are in use in digital technology. The most common are the decimal, binary, octal, and hexadecimal systems. The decimal system is clearly the most familiar to us because it is a tool that we use every day. Examining some of its characteristics will help us to better understand the other systems. In the next few pages we shall introduce four numerical representation systems that are used in the digital system. There are other systems, which we

will look at briefly.

- **Decimal**
- **Binary**
- **Octal**
- **Hexadecimal**

### **Decimal System**

The decimal system is composed of 10 numerals or symbols. These 10 symbols are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. Using these symbols as digits of a number, we can express any quantity. The decimal system is also called the base-10 system because it has 10 digits.

#### **Decimal Examples**

3.14<sub>10</sub>  
52<sub>10</sub>  
1024<sub>10</sub>  
64000<sub>10</sub>

## Binary System

In the binary system, there are only two symbols or possible digit values, 0 and 1. This base-2 system can be used to represent any quantity that can be represented in decimal or other base system. In digital systems the information that is being processed is usually presented in binary form. Binary quantities can be represented by any device that has only two operating states or possible conditions.

E.g., a switch is only open or closed. We arbitrarily (as we define them) let an open switch represent binary 0 and a closed switch represent binary 1. Thus we can represent any binary number by using series of switches.

## Octal System

The octal number system has a base of eight, meaning that it has eight possible digits: 0,1,2,3,4,5,6,7.

### octal to Decimal Conversion

$$237_8 = 2 \times (8^2) + 3 \times (8^1) + 7 \times (8^0) = 159_{10}$$

$$24.6_8 = 2 \times (8^1) + 4 \times (8^0) + 6 \times (8^{-1}) = 20.75_{10}$$

$$11.1_8 = 1 \times (8^1) + 1 \times (8^0) + 1 \times (8^{-1}) = 9.125_{10}$$

$$12.3_8 = 1 \times (8^1) + 2 \times (8^0) + 3 \times (8^{-1}) = 10.375_{10}$$

## Hexadecimal System

The hexadecimal system uses base 16. Thus, it has 16 possible digit symbols. It uses the digits 0 through 9 plus the letters A, B, C, D, E, and F as the 16 digit symbols.

### Hexadecimal to Decimal Conversion

$$24.6_{16} = 2 \times (16^1) + 4 \times (16^0) + 6 \times (16^{-1}) = 36.375_{10}$$

$$11.1_{16} = 1 \times (16^1) + 1 \times (16^0) + 1 \times (16^{-1}) = 17.0625_{10}$$

$$12.3_{16} = 1 \times (16^1) + 2 \times (16^0) + 3 \times (16^{-1}) = 18.1875_{10}$$

## Code Conversion

Converting from one code form to another code form is called code conversion, like converting from binary to decimal or converting from hexadecimal to decimal.

### Binary-To-Decimal Conversion

Any binary number can be converted to its decimal equivalent simply by summing together the weights of the various positions in the binary number which contain a 1.e.g.

$$11011_2 = 2^4 + 2^3 + 0 \cdot 2^2 + 2^1 + 2^0 = 16 + 8 + 0 + 2 + 1 = 27_{10}$$

### Octal-To-Binary Conversion

Each Octal digit is represented by three binary digits.

Example:

$$472_8 = (100)(111)(010)_2 = 100111010_2$$

### Octal-To-Hexadecimal Hexadecimal-To-Octal Conversion

- Convert Octal (Hexadecimal) to Binary first.
- Regroup the binary number by three bits per group starting from LSB if Octal is required.
- Regroup the binary number by four bits per group starting from LSB if Hexadecimal is required.

## Binary Codes

Binary codes are codes which are represented in binary system with modification from the original ones. Below we will be seeing the following:

- Weighted Binary Systems
- Non Weighted Codes

### Weighted Binary Systems

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

### 8421 Code/BCD Code

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

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

$$1 \times 8 + 0 \times 4 + 0 \times 2 + 1 \times 1 = 9$$

### 2421 Code

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

### 5211 Code

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

## Reflective Code

A code is said to be reflective when code for 9 is complement for the code for 0, and so is for 8 and 1 codes, 7 and 2, 6 and 3, 5 and 4. Codes 2421, 5211, and excess-3 are reflective, whereas the 8421 code is not.

## Excess-3 Code

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

## Gray Code

The gray code belongs to a class of codes called minimum change codes, in which only one bit in the code changes when moving from one code to the next. The Gray code is non-weighted code, as the position of bit does not contain any weight. The gray code is a reflective digital code which has the special property that any two subsequent numbers codes differ by only one bit. This is also called a unit-distance code. In digital Gray code has got a special place.

## Error Detecting and Correction Codes

For reliable transmission and storage of digital data, error detection and correction is required. Below are a few examples of codes which permit error detection and error correction after detection.

## Error Detecting Codes

When data is transmitted from one point to another, like in wireless transmission, or it is just stored, like in hard disks and memories, there are chances that data may get corrupted. To detect these data errors, we use special codes, which are error detection codes.

## Parity

In parity codes, every data byte, or nibble (according to how user wants to use it) is checked if they have even number of ones or even number of zeros. Based on this information an



additional bit is appended to the original data. Thus if we consider 8-bit data, adding the parity bit will make it 9 bit long.

At the receiver side, once again parity is calculated and matched with the received parity (bit 9), and if they match, data is ok, otherwise data is corrupt.

There are two types of parity:

- **Even parity:** Checks if there is an even number of ones; if so, parity bit is zero. When the number of ones is odd then parity bit is set to 1.
- **Odd Parity:** Checks if there is an odd number of ones; if so, parity bit is zero. When number of ones is even then parity bit is set to 1.

## **Error-Correcting Codes**

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

ECC (Error correcting codes) are used also in memories, networking, Hard disk, CDROM, DVD etc. Normally in networking chips (ASIC), we have 2 Error detection bits and 1 Error correction bit.

### **Hamming Code**

Hamming code adds a minimum number of bits to the data transmitted in a noisy channel, to be able to correct every possible one-bit error. It can detect (not correct) two-bits errors and cannot distinguish between 1-bit and 2-bits inconsistencies. It can't - in general – detect 3(or more)-bits errors The idea is that the failed bit position in an n-bit string (which we'll call X) can be represented in binary with  $\log_2(n)$  bits, hence we'll try to get it adding just  $\log_2(n)$  bits.

### **ASCII Code**

ASCII stands for American Standard Code for Information Interchange. It has become a world standard alphanumeric code for microcomputers and computers. It is a 7-bit code representing  $2^7 = 128$  different characters. These characters represent 26 upper case letters (A to Z), 26 lowercase letters (a to z), 10 numbers (0 to 9), 33 special characters and symbols and 33 control characters.

## BOOLEAN ALGEBRA AND LOGIC GATES

The English mathematician George Boole (1815-1864) sought to give symbolic form to Aristotle's system of logic. Boole wrote a treatise on the subject in 1854, titled *An Investigation of the Laws of Thought, on Which Are Founded the Mathematical Theories of Logic and Probabilities*, which codified several rules of relationship between mathematical quantities limited to one of two possible values: true or false, 1 or 0. His mathematical system became known as Boolean algebra. All arithmetic operations performed with Boolean quantities have but one of two possible Outcomes: either 1 or 0. There is no such thing as  $\sqrt{2}$  or  $\sqrt{-1}$  or  $\sqrt{1/2}$  in the Boolean world.

It is a world in which all other possibilities are invalid by fiat. As one might guess, this is not the kind of math you want to use when balancing a check book or calculating current through a resistor.

However, Claude Shannon of MIT fame recognized how Boolean algebra could be applied to on-and-off circuits, where all signals are characterized as either **high** (1) or **low** (0). His 1938 thesis, titled *A Symbolic Analysis of Relay and Switching Circuits*, put Boole's theoretical work to use in a way Boole never could have imagined, giving us a powerful mathematical tool for designing and analyzing digital circuits.

Like **normal** algebra, Boolean algebra uses alphabetical letters to denote variables. Unlike **normal** algebra, though, Boolean variables are always CAPITAL letters, never lowercase.

Because they are allowed to possess only one of two possible values, either 1 or 0, each and every variable has a complement: the opposite of its value. For example, if variable **A** has a value of 0, then the complement of A has a value of 1. Boolean notation uses a bar above the variable character to denote complementation, like this:

If:  $A=0$   
Then:  $\bar{A}=1$

If:  $A=1$   
Then:  $\bar{A}=0$

In written form, the complement of  $A$  denoted as  $A$ -not or  $A$ -bar. Sometimes a prime symbol is used to represent complementation. For example,  $A'$  would be the complement of  $A$ , much the same as using a prime symbol to denote differentiation in calculus rather than the fractional notation dot. Usually, though, the bar symbol finds more widespread use than the prime symbol, for reasons that will become more apparent later in this chapter.

## Boolean Arithmetic:

Let us begin our exploration of Boolean algebra by adding numbers together:

$$0 + 0 = 0$$

$$0 + 1 = 1$$

$$1 + 0 = 1$$

$$1 + 1 = 1$$

The first three sums make perfect sense to anyone familiar with elementary addition. The last sum, though, is quite possibly responsible for more confusion than any other single statement in digital electronics, because it seems to run contrary to the basic principles of mathematics.

Well, it does contradict principles of addition for real numbers, but not for Boolean numbers. Remember that in the world of Boolean algebra, there are only two possible values for any quantity and for any arithmetic operation: 1 or 0. There is no such thing as  $||2||$  within the scope of Boolean values. Since the sum  $||1 + 1||$  certainly isn't 0, it must be 1 by process of elimination.

### **Principle of Duality:**

It states that every algebraic expression is deducible from the postulates of Boolean algebra, and it remains valid if the operators & identity elements are interchanged. If the inputs of a NOR gate are inverted we get a AND equivalent circuit. Similarly when the inputs of a NAND gate are inverted, we get a OR equivalent circuit. This property is called duality.

### **Theorems of Boolean algebra**

The theorems of Boolean algebra can be used to simplify many a complex Boolean expression and also to transform the given expression into a more useful and meaningful equivalent expression. The theorems are presented as pairs, with the two theorems in a given pair being the dual of each other. These theorems can be very easily verified by the method of 'perfect induction'. According to this method, the validity of the expression is tested for all possible combinations of values of the variables involved. Also, since the validity of the theorem is based on its being true for all possible combinations of values of variables, there is no reason why a variable cannot be replaced with its complement, or vice versa, without disturbing the validity. Another important point is that, if a given expression is valid, its dual will also be valid.

**Theorem 1 (Operations with '0' and '1')**

**(a)  $0.X = 0$  and (b)  $1+X = 1$**

Where X is not necessarily a single variable – it could be a term or even a large expression.

Theorem 1(a) can be proved by substituting all possible values of X, that is, 0 and 1, into the given expression and checking whether the LHS equals the RHS:

- For  $X = 0$ ,  $LHS = 0.X = 0.0 = 0 = RHS$ .
- For  $X = 1$ ,  $LHS = 0.1 = 0 = RHS$ .

Thus,  $0.X = 0$  irrespective of the value of X, and hence the proof.

Theorem 1(b) can be proved in a similar manner. In general, according to theorem 1,  
 $0. (\text{Boolean expression}) = 0$  and  $1+ (\text{Boolean expression}) = 1$ .

For example:  $0. (A.B+B.C +C.D) = 0$  and  $1+ (A.B+B.C +C.D) = 1$ , where A, B and C are Boolean variables.

**Theorem 2 (Operations with '0' and '1')**

**(a)  $1.X = X$  and (b)  $0+X = X$**

where X could be a variable, a term or even a large expression. According to this theorem, ANDing a Boolean expression to '1' or ORing '0' to it makes no difference to the expression:

- For  $X = 0$ ,  $LHS = 1.0 = 0 = RHS$ .
- For  $X = 1$ ,  $LHS = 1.1 = 1 = RHS$ .

Also,

$1. (\text{Boolean expression}) = \text{Boolean expression}$  and  $0 + (\text{Boolean expression}) = \text{Boolean expression}$ .

For example,

$1.(A+B.C +C.D) = 0+(A+B.C +C.D) = A+B.C +C.D$

### Theorem 3 (Idempotent or Identity Laws)

(a)  $X.X.X.....X = X$  and (b)  $X+X+X +\dots+X = X$

Theorems 3(a) and (b) are known by the name of idempotent laws, also known as identity laws.

Theorem 3(a) is a direct outcome of an AND gate operation, whereas theorem 3(b) represents an OR gate operation when all the inputs of the gate have been tied together. The scope of idempotent laws can be expanded further by considering  $X$  to be a term or an expression. For example, let us apply idempotent laws to simplify the following Boolean expression:

$$\begin{aligned}(A.\bar{B}.\bar{B} + C.C).(A.\bar{B}.\bar{B} + A.\bar{B} + C.C) &= (A.\bar{B} + C).(A.\bar{B} + A.\bar{B} + C) \\ &= (A.\bar{B} + C).(A.\bar{B} + C) = A.\bar{B} + C\end{aligned}$$

### Theorem 4 (Complementation Law)

(a)  $X.X = 0$  and (b)  $X+X = 1$

According to this theorem, in general, any Boolean expression when ANDed to its complement yields a '0' and when ORed to its complement yields a '1', irrespective of the complexity of the expression:

- For  $X = 0$ ,  $\bar{X} = 1$ . Therefore,  $X.\bar{X} = 0.1 = 0$ .
- For  $X = 1$ ,  $\bar{X} = 0$ . Therefore,  $X.\bar{X} = 1.0 = 0$ .

Hence, theorem 4(a) is proved. Since theorem 4(b) is the dual of theorem 4(a), its proof is implied.

The example below further illustrates the application of complementation laws:

$$(A + B.C)(\overline{A + B.C}) = 0 \quad \text{and} \quad (A + B.C) + \overline{(A + B.C)} = 1$$

**Theorem 5 (Commutative property)**

Mathematical identity, called a **property** or a **law**, describes how differing variables relate to each other in a system of numbers. One of these properties is known as the **commutative property**, and it applies equally to addition and multiplication.

In essence, the commutative property tells us we can reverse the order of variables that are either added together or multiplied together without changing the truth of the expression:

**Commutative property of addition**

$$A + B = B + A$$

**Commutative property of multiplication**

$$AB = BA$$

**Theorem 6 (Associative Property)**

The **Associative Property**, again applying equally well to addition and multiplication.

This property tells us we can associate groups of added or multiplied variables together with parentheses without altering the truth of the equations.

**Associative property of addition**

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

**Associative property of multiplication**

$$A (BC) = (AB) C$$

**Theorem 7 (Distributive Property)**

The **Distributive Property**, illustrating how to expand a Boolean expression formed by the product of a sum, and in reverse shows us how terms may be factored out of Boolean

sums-of-products:

Distributive property

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

Theorem 8 (Absorption Law or Redundancy Law)

$$(a) X + X.Y = X \text{ and } (b) X.(X+Y) = X$$

The proof of absorption law is straightforward:

$$X + X.Y = X.(1+Y) = X.1 = X$$

Theorem 8(b) is the dual of theorem 8(a) and hence stands proved.

The crux of this simplification theorem is that, if a smaller term appears in a larger term, then the larger term is redundant. The following examples further illustrate the underlying concept:

$$A + A.\bar{B} + A.\bar{B}.\bar{C} + A.\bar{B}.C + \bar{C}.B.A = A$$

and

$$(\bar{A} + B + \bar{C}).(\bar{A} + B).(C + B + \bar{A}) = \bar{A} + B$$

## De-Morgan's First Theorem

It States that —The complement of the sum of the variables is equal to the product of the complement of each variable . This theorem may be expressed by the following Boolean expression.

$$\overline{A + B} = \bar{A} . \bar{B}$$



## De-Morgan's Second Theorem

It states that the —Complement of the product of variables is equal to the sum of complements of each individual variables|. Boolean expression for this theorem is

$$\overline{A \cdot B} = \overline{A} + \overline{B}$$

## Boolean Function

Boolean functions are represented in various forms. The two popular forms are *truth tables* and *Venn diagrams*. Truth tables represent functions in a tabular form, while Venn diagrams provide a graphic representation. In addition, there are two algebraic representations known as the *standard* (or *normal*) *form* and the *canonical form*.

Draw the truth table for  $Z = AB' + A'C + A'B'C$ .

There are three component variables:  $A$ ,  $B$ , and  $C$ . Hence, there will be  $2^3 = 8$  combinations of values. These eight combinations are shown in the first three columns of Table 2.2. These combinations correspond to binary numbers 000 through 111, or  $(0)_{10}$  through  $(7)_{10}$ .

To evaluate  $Z$  in the example function, knowing the values for  $A$ ,  $B$ , and  $C$  at each row of the truth table (Table 2.2), we would first generate values for  $A'$  and  $B'$  and then generate values for  $AB'$ ,  $A'C$  and  $A'B'C$  by ANDing the values in the appropriate columns for each row. Finally, we would derive the value of  $Z$  by ORing the values in the last three columns for each row. Note that evaluating  $A'B'C$  corresponds to ANDing  $A'$  and  $B'$  values, followed by ANDing the value of  $C$ . Similarly, if more than two values are to be ORed, they are ORed

$$Z=AB'+A'C+A'B'C'$$

A	B	C	A'	B'	AB'	A'C	A'B'C	Z
0	0	0	1	1	0	0	0	0
0	0	1	1	1	0	1	1	1
0	1	0	1	0	0	0	0	0
0	1	1	1	0	0	1	0	1
1	0	0	0	1	1	0	0	1
1	0	1	0	1	1	0	0	1
1	1	0	0	0	0	0	0	0
1	1	1	0	0	0	0	0	0

### Canonical Form of Boolean Expressions

An expanded form of Boolean expression, where each term contains all Boolean variables in their true or complemented form, is also known as the canonical form of the expression. As an illustration,  $f(A,B,C) = \overline{A}.\overline{B}.\overline{C} + \overline{A}.\overline{B}.C + A.B.C$  is a Boolean function of three variables expressed in canonical form. This function after simplification reduces to  $\overline{A}.\overline{B} + A.B.C$  and loses its canonical form.

### MIN TERMS AND MAX TERMS

Any boolean expression may be expressed in terms of either minterms or maxterms. To do this we must first define the concept of a literal. A literal is a single variable within a term which may or may not be complemented. For an expression with N variables, minterms and maxterms are defined as follows :

- A minterm is the product of N distinct literals where each literal occurs exactly once.
- A maxterm is the sum of N distinct literals where each literal occurs exactly once.

## Product-of-Sums Expressions

A product-of-sums expression contains the product of different terms, with each term being either a single literal or a sum of more than one literal. It can be obtained from the truth

table by considering those input combinations that produce a logic '0' at the output. Each such input combination gives a term, and the product of all such terms gives the expression.

Different terms are obtained by taking the sum of the corresponding literals. Here '0' and '1' respectively mean the uncomplemented and complemented variables, unlike sum-of-products

expressions where '0' and '1' respectively mean complemented and uncomplemented variables.

Since each term in the case of the product-of-sums expression is going to be the sum of literals, this implies that it is going to be implemented using an OR operation. Now, an OR gate produces a logic '0' only when all its inputs are in the logic '0' state, which means that the first term corresponding to the second row of the truth table will be  $A+B+C$ . The product of-

sums Boolean expression for this truth table is given by Transforming the given product of-sums expression into an equivalent sum-of-products expression is a straightforward process. Multiplying out the given expression and carrying out the obvious simplification provides the equivalent sum-of-products expression:

A given sum-of-products expression can be transformed into an equivalent product-of-sums expression by (a) taking the dual of the given expression, (b) multiplying out different terms to get the sum-of products form, (c) removing redundancy and (d) taking a dual to get the

**equivalent product-of-sums expression. As an illustration, let us find the equivalent product of sums expression of the sum-of products expression**

$$A.B + \bar{A}.\bar{B}$$

The dual of the given expression =  $(A + B).(\bar{A} + \bar{B})$ :

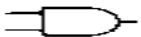

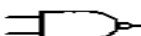
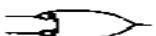

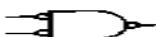
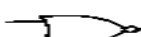
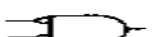


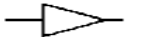
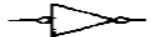
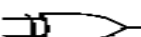
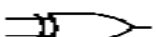
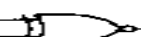
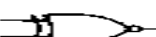
$$(A + B).(\bar{A} + \bar{B}) = A.\bar{A} + A.\bar{B} + B.\bar{A} + B.\bar{B} = 0 + A.\bar{B} + B.\bar{A} + 0 = A.\bar{B} + \bar{A}.B$$

The dual of  $(A.\bar{B} + \bar{A}.B) = (A + \bar{B}).(\bar{A} + B)$ . Therefore

$$A.B + \bar{A}.\bar{B} = (A + \bar{B}).(\bar{A} + B)$$

## **Digital Logic Gates**

**The basic logic gates are AND, OR, NAND, NOR, XOR, INV, and BUF. The last two are not standard terms; they stand for 'inverter' and 'buffer', respectively. The symbols for these gates and their corresponding Boolean expressions are given in Fig. 2.**

Name	Expression	Symbol	Negative true symbol
AND	$AB$		
NAND	$\overline{AB}$		
OR	$A + B$		
NOR	$\overline{A + B}$		
INVERT	$\overline{A}$		
BUFFER	$A$		
XOR	$A \oplus B$		
XNOR	$\overline{A \oplus B}$		

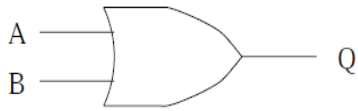
**Figure 2:**

All of the logical gate functions, as well as the Boolean relations discussed in the next section, follow from the truth tables for the AND and OR gates. We reproduce these below. We also show the XOR truth table, because it comes up quite often, although, as we shall see, it is not elemental.



A	B	Q
0	0	0
1	0	0
0	1	0
1	1	1

Figure 3: AND gate.



A	B	Q
0	0	0
1	0	1
0	1	1
1	1	1

Figure 4: OR gate.



A	B	Q
0	0	0
1	0	1
0	1	1
1	1	0

Figure 5: XOR (exclusive OR) gate.

## WORK IN PROGRESS

**CONTD.....**

### **References**

- [1] Digital Design,3rd edition by M. Morris Mano, Pearson Education
- [2] Digital Design-Principle& practice, 3rd edition by John F. Wakerley, Pears
- [3] Digital Electronics by R.Anitha NPRCET



