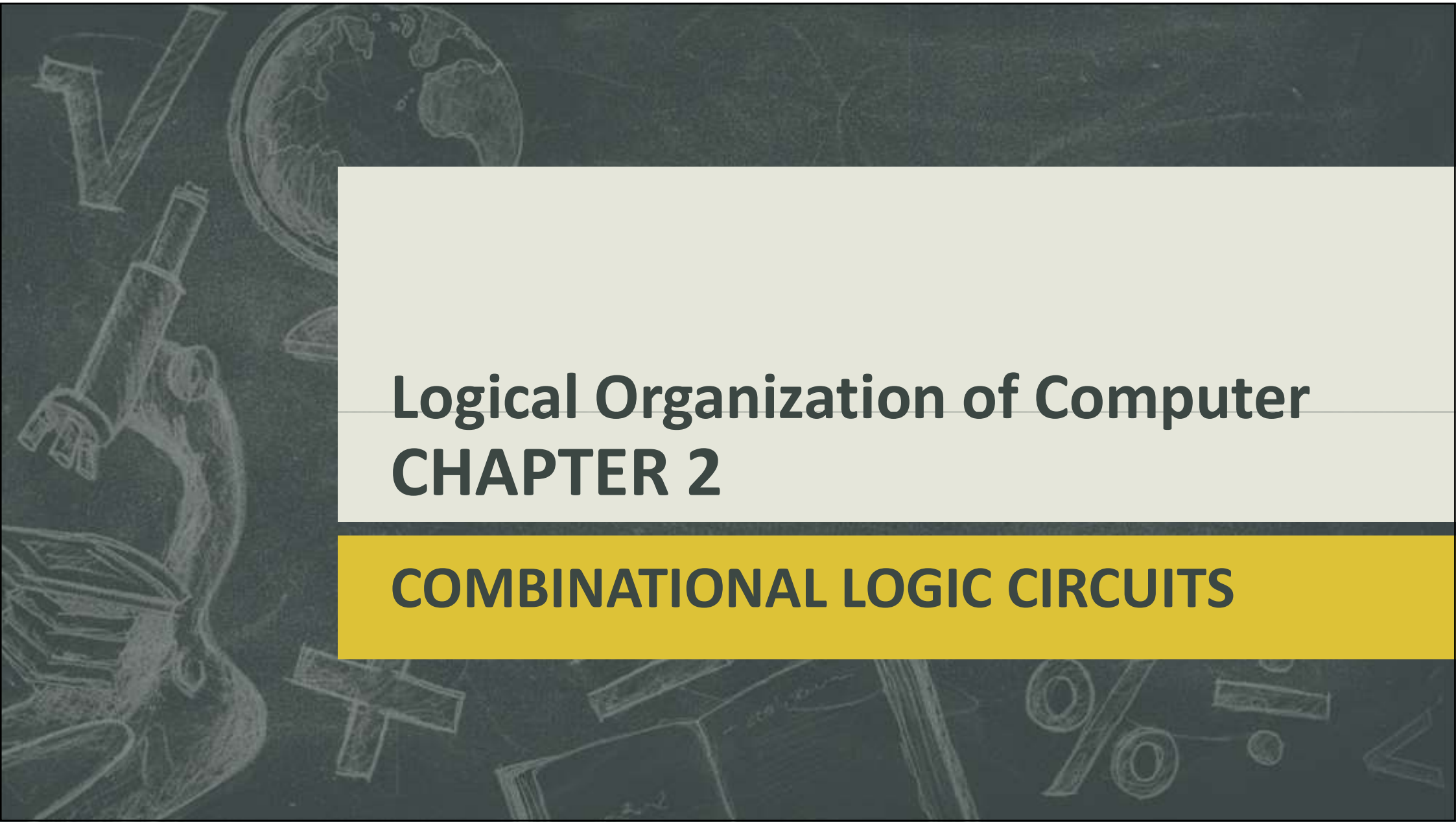


Logical Organization of Computer
CHAPTER 2

COMBINATIONAL LOGIC CIRCUITS

1. BINARY LOGIC AND GATES

- Digital circuits are hardware components that manipulate binary information. The circuits are implemented using transistors and interconnections in complex semiconductor devices called *integrated circuits*. Each basic circuit is referred to as a *logic gate*.
- Each gate performs a specific logical operation. The outputs of gates are applied to the inputs of other gates to form a digital circuit.
- In order to describe the operational properties of digital circuits, we need to introduce a mathematical notation that specifies the operation of each gate and that can be used to analyze and design circuits known as *Boolean algebras*.

❑ Binary Logic

- Binary logic deals with binary variables, which take on two discrete values, and with the operations of mathematical logic applied to these variables.
- Associated with the binary variables are three basic logical operations called AND, OR, and NOT.
- **1. AND.** This operation is represented by a dot or by the absence of an operator. For example, $Z=X \cdot Y$ or $Z=XY$ is read "Z is equal to X AND Y." The logical operation AND is interpreted to mean that $Z=1$ if and only if $X=1$ and $Y=1$; otherwise $Z=0$.

$$0 \cdot 0 = 0$$
$$0 \cdot 1 = 0$$
$$1 \cdot 0 = 0$$
$$1 \cdot 1 = 1$$

❑ Binary Logic

2. OR. This operation is represented by a plus symbol. For example, $Z = X + Y$ is read "Z is equal to X OR Y," meaning that $Z = 1$ if $X = 1$ or if $Y = 1$, or if both $X = 1$ and $Y = 1$. $Z = 0$ if and only if $X = 0$ and $Y = 0$.

$$0 + 0 = 0$$
$$0 + 1 = 1$$
$$1 + 0 = 1$$
$$1 + 1 = 1$$

3. NOT. This operation is represented by a bar over the variable. For example, $Z = \overline{X}$ is read "Z is equal to NOT X," meaning that Z is what X is not. In other words, if $X = 1$, then $Z = 0$; but if $X = 0$, then $Z = 1$. The NOT operation is also referred to as the *complement* operation, since it changes a 1 to 0 and a 0 to 1.

❑ Binary Logic

- A *truth table* for an operation is a table of combinations of the binary variables showing the relationship between the values that the variables take on and the values of the result of the operation.
- The truth tables for the operations AND, OR, and NOT are shown below.

Binary Logic

TABLE 1
Truth Tables for the Three Basic Logical Operations

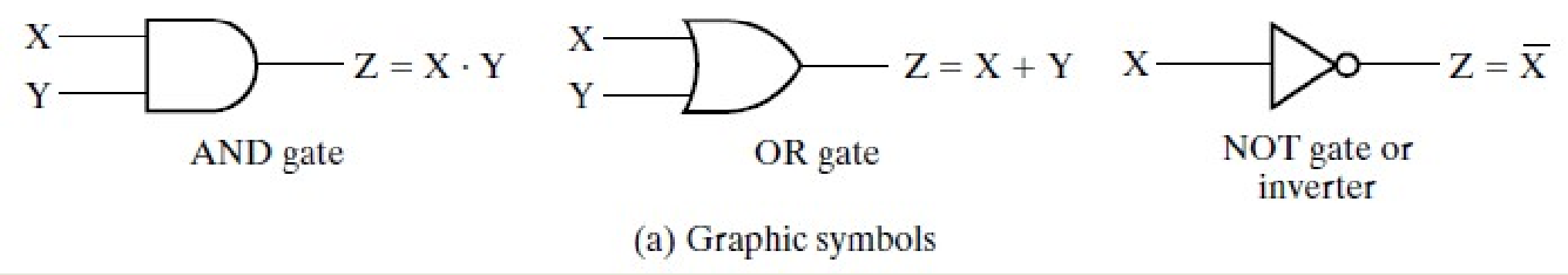
AND			OR			NOT	
X	Y	$Z = X \cdot Y$	X	Y	$Z = X + Y$	X	$Z = \overline{X}$
0	0	0	0	0	0	0	1
0	1	0	0	1	1	1	0
1	0	0	1	0	1		
1	1	1	1	1	1		

❑ Logic Gates

- Logic gates are electronic circuits that operate on one or more input signals to produce an output signal.
- The input terminals of logic gates accept binary signals within the allowable range and respond at the output terminals with binary signals that fall within a specified range.
- The intermediate regions between the allowed ranges in the figure are crossed only during changes from 1 to 0 or from 0 to 1.
- These changes are called *transitions*, and the intermediate regions are called the *transition regions*.

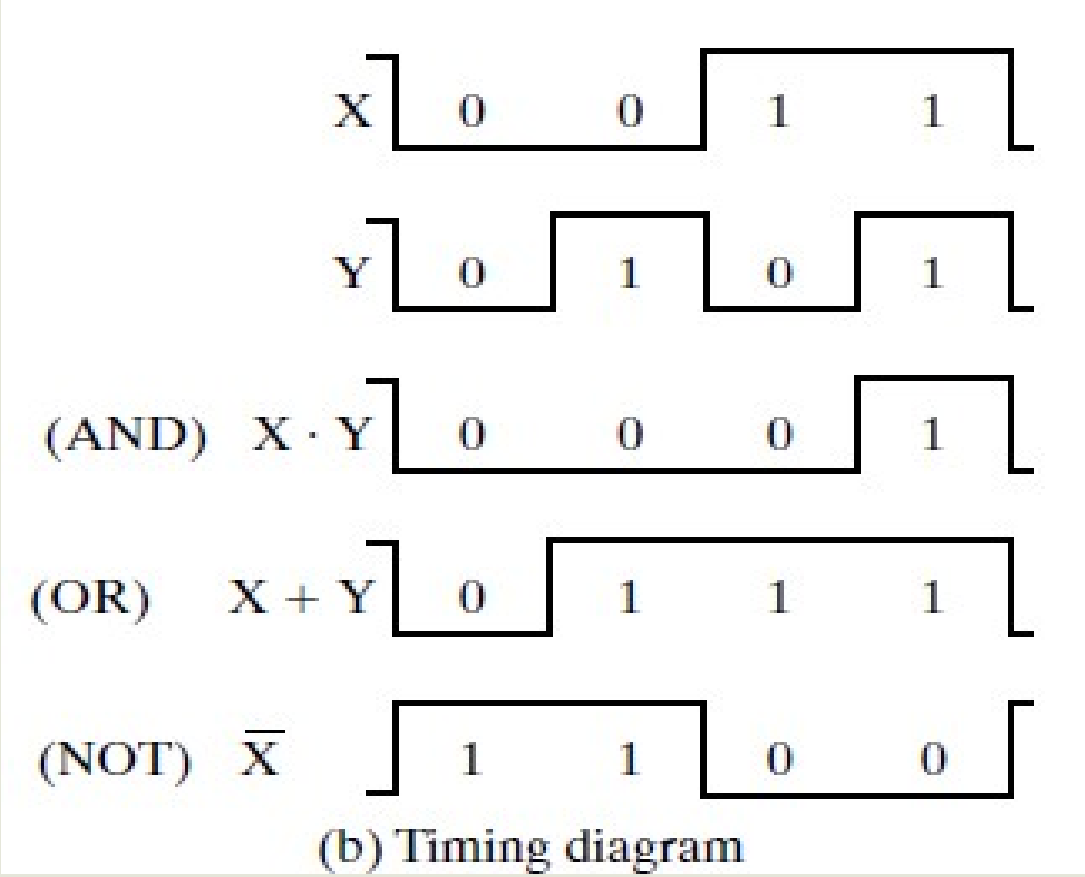
❑ Logic Gates

- The graphics symbols used to designate the three types of gates—AND, OR, and NOT—are shown below.



(a) Graphic symbols

❑ Logic Gates



(b) Timing diagram

2. BOOLEAN ALGEBRA

- A *Boolean expression* is an algebraic expression formed by using binary variables, the constants 0 and 1, the logic operation symbols, and parentheses.
- *Boolean function* can be described by a Boolean equation consisting of a binary variable identifying the function followed by an equals sign and a Boolean expression.

❑ Basic Identities of Boolean Algebra

Annulment Law – A term AND´ed with a "0" equals 0 or OR´ed with a "1" will equal 1

A . 0 = 0 A variable AND'ed with 0 is always equal to 0
A + 1 = 1 A variable OR'ed with 1 is always equal to 1

Identity Law – A term OR´ed with a "0" or AND´ed with a "1" will always equal that term

A + 0 = A A variable OR'ed with 0 is always equal to the variable
A . 1 = A A variable AND'ed with 1 is always equal to the variable

❑ Basic Identities of Boolean Algebra

Idempotent Law – An input that is AND´ed or OR´ed with itself is equal to that input

$A + A = A$	A variable OR'ed with itself is always equal to the variable
$A \cdot A = A$	A variable AND'ed with itself is always equal to the variable

Complement Law – A term AND´ed with its complement equals "0" and a term OR´ed with its complement equals "1"

$A \cdot \overline{A} = 0$	A variable AND'ed with its complement is always equal to 0
$A + \overline{A} = 1$	A variable OR'ed with its complement is always equal to 1

❑ Basic Identities of Boolean Algebra

Commutative Law – The order of application of two separate terms is not important

$A \cdot B = B \cdot A$ The order in which two variables are AND'ed makes no difference

$A + B = B + A$ The order in which two variables are OR'ed makes no difference

Double Negation Law – A term that is inverted twice is equal to the original term

$\overline{\overline{A}} = A$ A double complement of a variable is always equal to the variable

❑ Basic Identities of Boolean Algebra

Distributive Law – This law permits the multiplying or factoring out of an expression.

A(B + C) = A.B + A.C (OR Distributive Law)	
A + (B.C) = (A + B).(A + C) (AND Distributive Law)	

Absorptive Law – This law enables a reduction in a complicated expression to a simpler one by absorbing like terms.

A + (A.B) = A (OR Absorption Law)	Since A+AB = A(1+B) = A.1 =A;
A(A + B) = A (AND Absorption Law)	A(A+B) = AA+AB = A+AB = A(1+B) = A.1 =A;

❑ Basic Identities of Boolean Algebra

Associative Law – This law allows the removal of brackets from an expression and regrouping of the variables.

$A + (B + C) = (A + B) + C = A + B + C$ (OR Associate Law)

$A(B.C) = (A.B)C = A \,.\, B \,.\, C$ (AND Associate Law)

de Morgan´s Theorem – There are two "de Morgan´s" rules or theorems,

(1) Two separate terms NOR´ed together is the same as the two terms inverted (Complement) and AND´ed for example: $\overline{A+B} = \overline{A} \,.\, \overline{B}$

(2) Two separate terms NAND´ed together is the same as the two terms inverted (Complement) and OR´ed for example: $\overline{A.B} = \overline{A} + \overline{B}$

❑ Basic Identities of Boolean Algebra

Truth Tables to Verify DeMorgan's Theorem

(a) X	Y	X + Y	$\overline{X+Y}$
0	0	0	1
0	1	1	0
1	0	1	0
1	1	1	0

(b) X	Y	$\overline{X}$	$\overline{Y}$	$\overline{X} \cdot \overline{Y}$
0	0	1	1	1
0	1	1	0	0
1	0	0	1	0
1	1	0	0	0

❑ Boolean Functions

- A Boolean function refers to a function having n number of entries or variables, so it has 2n number of possible combinations of the given variables. Such functions would only assume 0 or 1 in their output.
- An example of a Boolean function is, f(p,q,r) = p X q + r. We are implementing these functions with the logic gates.

❑ Boolean Functions

- We make use of an algebraic expression known as the Boolean Expression to describe the Boolean Function.
- The Boolean Expression contains the logic operation symbols, binary variables, and the constants 1 and 0. Let us now consider the example given below:
- $F(W, X, Y, Z) = W + X\bar{Y} + WZY$ Equation No. 1
- The left side of this equation here represents the output B. So we can state equation no. 1
- $B = W + X\bar{Y} + WZY$

❑ Boolean Functions

- **Truth Table Formation**
- We use a truth table to represent a table that has all the combinations of inputs along with their corresponding results.

$$F(W, X, Y) = W + XY$$

Inputs			Output
W	X	Y	F
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

❑ Boolean Functions

- **Truth Table Formation**
  - A Boolean function may be transferred from an algebraic expression into a logic diagram composed of AND, OR and NOT gates

TABLE 2-2
Truth Tables for $F_1 = xyz'$, $F_2 = x + y'z$, $F_3 = x'y'z + x'yz + xy'$, and $F_4 = xy' + x'z$

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

❑ Algebraic Manipulation

- Boolean algebra is a useful tool for simplifying digital circuits. Consider, for example, the Boolean function represented by

$$F = \overline{X}YZ + \overline{X}Y\overline{Z} + XZ$$

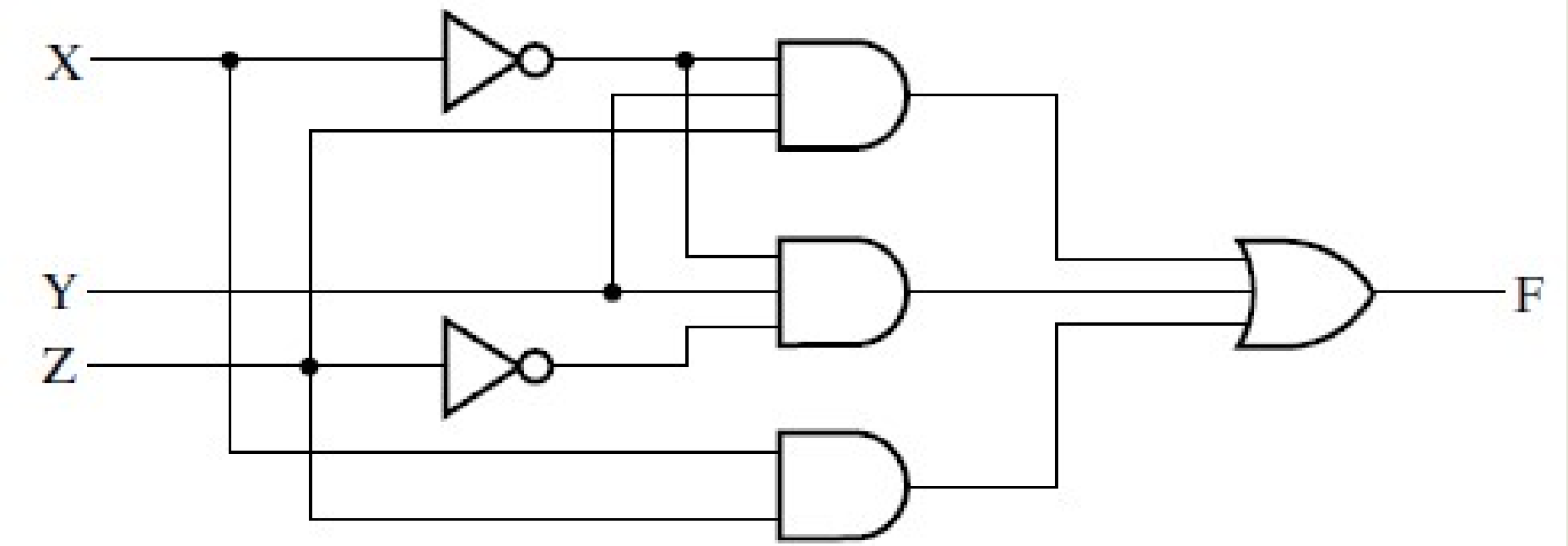
$$\begin{aligned} F &= \overline{X}YZ + \overline{X}Y\overline{Z} + XZ \\ &= \overline{X}Y(Z + \overline{Z}) + XZ \\ &= \overline{X}Y \cdot 1 + XZ \\ &= \overline{X}Y + XZ \end{aligned}$$

Distributive Law

Complement Law

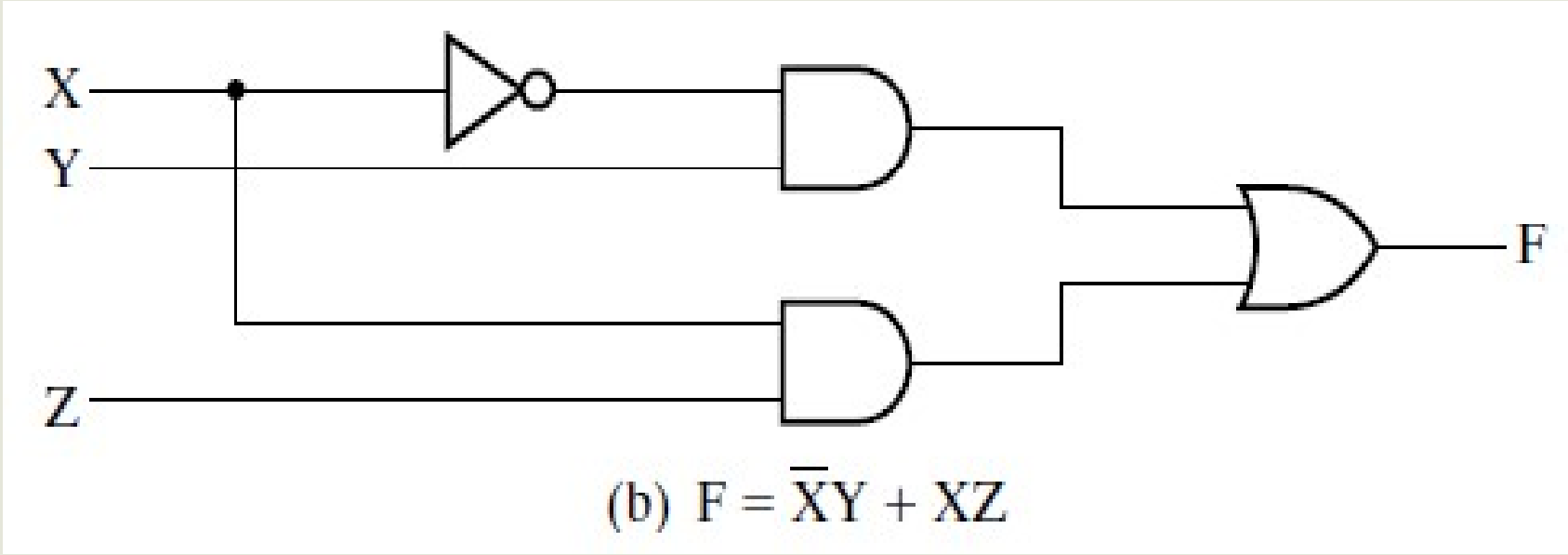
Identity Law

❑ Algebraic Manipulation



(a) $F = \overline{X}YZ + \overline{X}Y\overline{Z} + XZ$

❑ Algebraic Manipulation



(b) $F = \overline{X}Y + XZ$

❑ Algebraic Manipulation

Truth Table for Boolean Function

X	Y	Z	(a) F	(b) F
0	0	0	0	0
0	0	1	0	0
0	1	0	1	1
0	1	1	1	1
1	0	0	0	0
1	0	1	1	1
1	1	0	0	0
1	1	1	1	1

❑ Algebraic Manipulation

- Minimise the following functions using algebraic method:

1. ab + ab' + a'b
2. a'b + a'b' + b‘
3. b(a+c) + ab' + bc' + c
4. ab’ +a(b+c)+b(b+c)
5. [ab’(c+bd)+a’b’]c
6. a’bc+ab’c’+a’b’c’+abc’+abc

3. CANONICAL AND STANDARD FORMS

- A Boolean function expressed algebraically can be written in a variety of ways.
- There are, however, specific ways of writing algebraic equations that are considered to be standard forms.
- The standard forms facilitate the simplification procedures for Boolean expressions and, in some cases, may result in more desirable expressions for implementing logic circuits.

3. STANDARD FORMS

The standard forms contain *product terms* and *sum terms*. An example of a product term is $X\overline{Y}Z$. This is a logical product consisting of an AND operation among three literals. An example of a sum term is $X + Y + \overline{Z}$. This is a logical sum consisting of an OR operation among the literals. In Boolean algebra, the words "product" and "sum" do not imply arithmetic operations; instead, they specify the logical operations AND and OR, respectively.

3. CANONICAL AND STANDARD FORMS

- In Boolean algebra, Boolean function can be expressed as Canonical Disjunctive Normal Form known as **minterm** and some are expressed as Canonical Conjunctive Normal Form known as **maxterm** .

❑ Minterms and Maxterms

- A product term in which all the variables appear exactly once, either complemented or uncomplemented, is called a *minterm*.
- Its one characteristic is that it represents exactly one combination of binary variable values in the truth table. It has the value 1 for that combination and 0 for all others.

❑ Minterms and Maxterms

Minterms for Three Variables

X	Y	Z	Product Term	Symbol	m_0	m_1	m_2	m_3	m_4	m_5	m_6	m_7
0	0	0	$\overline{X}\,\overline{Y}\,\overline{Z}$	m_0	1	0	0	0	0	0	0	0
0	0	1	$\overline{X}\,\overline{Y}Z$	m_1	0	1	0	0	0	0	0	0
0	1	0	$\overline{X}Y\overline{Z}$	m_2	0	0	1	0	0	0	0	0
0	1	1	$\overline{X}YZ$	m_3	0	0	0	1	0	0	0	0
1	0	0	$X\overline{Y}\,\overline{Z}$	m_4	0	0	0	0	1	0	0	0
1	0	1	$X\overline{Y}Z$	m_5	0	0	0	0	0	1	0	0
1	1	0	$XY\overline{Z}$	m_6	0	0	0	0	0	0	1	0
1	1	1	XYZ	m_7	0	0	0	0	0	0	0	1

❑ Minterms and Maxterms

- A sum term that contains all the variables in complemented or uncomplemented form is called a *maxterm*.
- Again, it is possible to formulate 2^n maxterms with n variables.
- Each maxterm is a logical sum of the three variables, with each variable being complemented if the corresponding bit of the binary number is 1 and uncomplemented if it is 0.

❑ Minterms and Maxterms

Maxterms for Three Variables

X	Y	Z	Sum Term	Symbol	M_0	M_1	M_2	M_3	M_4	M_5	M_6	M_7
0	0	0	$X + Y + Z$	M_0	0	1	1	1	1	1	1	1
0	0	1	$X + Y + \overline{Z}$	M_1	1	0	1	1	1	1	1	1
0	1	0	$X + \overline{Y} + Z$	M_2	1	1	0	1	1	1	1	1
0	1	1	$X + \overline{Y} + \overline{Z}$	M_3	1	1	1	0	1	1	1	1
1	0	0	$\overline{X} + Y + Z$	M_4	1	1	1	1	0	1	1	1
1	0	1	$\overline{X} + Y + \overline{Z}$	M_5	1	1	1	1	1	0	1	1
1	1	0	$\overline{X} + \overline{Y} + Z$	M_6	1	1	1	1	1	1	0	1
1	1	1	$\overline{X} + \overline{Y} + \overline{Z}$	M_7	1	1	1	1	1	1	1	0

❑ Minterms and Maxterms

- Consider the Boolean function F in Table

Boolean Functions of Three Variables

(a) X	Y	Z	F	$\overline{F}$
0	0	0	1	0
0	0	1	0	1
0	1	0	1	0
0	1	1	0	1
1	0	0	0	1
1	0	1	1	0
1	1	0	0	1
1	1	1	1	0

$$F = \overline{X}\overline{Y}\overline{Z} + \overline{X}Y\overline{Z} + X\overline{Y}Z + XYZ = m_0 + m_2 + m_5 + m_7$$

$$\overline{F}(X,Y,Z) = \overline{X}\overline{Y}Z + \overline{X}YZ + X\overline{Y}\overline{Z} + XY\overline{Z} = m_1 + m_3 + m_4 + m_6$$

❑ Minterms and Maxterms

			Minterms	Maxterms
X	Y	Z	Product Terms	Sum Terms
0	0	0	$m_0 = \overline{X} \cdot \overline{Y} \cdot \overline{Z} = \min(\overline{X}, \overline{Y}, \overline{Z})$	$M_0 = X + Y + Z = \max(X, Y, Z)$
0	0	1	$m_1 = \overline{X} \cdot \overline{Y} \cdot Z = \min(\overline{X}, \overline{Y}, Z)$	$M_1 = X + Y + \overline{Z} = \max(X, Y, \overline{Z})$
0	1	0	$m_2 = \overline{X} \cdot Y \cdot \overline{Z} = \min(\overline{X}, Y, \overline{Z})$	$M_2 = X + \overline{Y} + Z = \max(X, \overline{Y}, Z)$
0	1	1	$m_3 = \overline{X} \cdot Y \cdot Z = \min(\overline{X}, Y, Z)$	$M_3 = X + \overline{Y} + \overline{Z} = \max(X, \overline{Y}, \overline{Z})$
1	0	0	$m_4 = X \cdot \overline{Y} \cdot \overline{Z} = \min(X, \overline{Y}, \overline{Z})$	$M_4 = \overline{X} + Y + Z = \max(\overline{X}, Y, Z)$
1	0	1	$m_5 = X \cdot \overline{Y} \cdot Z = \min(X, \overline{Y}, Z)$	$M_5 = \overline{X} + Y + \overline{Z} = \max(\overline{X}, Y, \overline{Z})$
1	1	0	$m_6 = X \cdot Y \cdot \overline{Z} = \min(X, Y, \overline{Z})$	$M_6 = \overline{X} + \overline{Y} + Z = \max(\overline{X}, \overline{Y}, Z)$
1	1	1	$m_7 = X \cdot Y \cdot Z = \min(X, Y, Z)$	$M_7 = \overline{X} + \overline{Y} + \overline{Z} = \max(\overline{X}, \overline{Y}, \overline{Z})$

❑ Minterms and Maxterms

The following is a summary of the most important properties of minterms:

1. There are 2^n minterms for n Boolean variables. These minterms can be generated from the binary numbers from 0 to $2^n - 1$.
2. Any Boolean function can be expressed as a logical sum of minterms.
3. The complement of a function contains those minterms not included in the original function.
4. A function that includes all the 2^n minterms is equal to logic 1.

❑ Minterms and Maxterms

- We perform Sum of minterm also known as Sum of products (SOP) .
- We perform Product of Maxterm also known as Product of sum (POS).
- Boolean functions expressed as a sum of minterms or product of maxterms are said to be in canonical form.

❑ Sum of Products (SOP)

- The sum of products (SOP) expression of a Boolean function can be obtained from its truth table summing or performing OR operation of the product terms corresponding to the combinations containing a function value of 1.
- In the product terms the input variables appear either in true (uncomplemented) form if it contains the value 1, or in complemented form if it possesses the value 0.
- Now, consider the following truth table in Figure 2.11, for a three-input function Y. Here the output Y value is 1 for the input conditions of 010, 100, 101, and 110, and their corresponding product terms are A'BC', AB'C', AB'C, and ABC' respectively

❑ Sum of Products (SOP)

- The final sum of products expression (SOP) for the output Y is derived by summing or performing an OR operation of the four product terms as shown below.
- Y = A′BC′ + AB′C′ + AB′C + ABC′

Inputs			*Output*	*Product terms*	*Sum terms*
A	*B*	*C*	*Y*		
0	0	0	0		A + B + C
0	0	1	0		A + B + C′
0	1	0	1	A′BC′	
0	1	1	0		A + B′ + C′
1	0	0	1	AB′C′	
1	0	1	1	AB′C	
1	1	0	1	ABC′	
1	1	1	0		A′ + B′ + C′

Figure 2.11

❑ Sum of Products (SOP)

- In general, the procedure of deriving the output expression in SOP form from a truth table can be summarized as below.
- 1. Form a product term for each input combination in the table, containing an output value of 1.
- 2. Each product term consists of its input variables in either true form or complemented form. If the input variable is 0, it appears in complemented form and if the input variable is 1, it appears in true form.
- 3. To obtain the final SOP expression of the output, all the product terms are OR operated.

❑ Product of Sums (POS)

- The product of sums (POS) expression of a Boolean function can also be obtained from its truth table by a similar procedure.
- Here, an AND operation is performed on the sum terms corresponding to the combinations containing a function value of 0.
- In the sum terms the input variables appear either in true (uncomplemented) form if it contains the value 0, or in complemented form if it possesses the value 1.
- Now, consider the same truth table as shown in Figure 2.11, for a three-input function Y. Here the output Y value is 0 for the input conditions of 000, 001, 011, and 111, and their corresponding product terms are A + B + C, A + B + C′, A + B′ + C′, and A′ + B′ + C′ respectively.

❑ Product of Sums (POS)

- Final product of sums expression (POS) for the output Y is derived by performing an AND operation of the four sum terms as shown below.
- Y = (A + B + C) (A + B + C′) (A + B′ + C′) (A′ + B′ + C′)
- In general, the procedure of deriving the POS form from a truth table can be summarized as below.
- 1. Form a sum term for each input combination in the table, containing an output value of 0.
- 2. Each product term consists of its input variables in either true form or complemented form. If the input variable is 1, it appears in complemented form and if the input variable is 0, it appears in true form.
- 3. To obtain the final POS expression of the output, all the sum terms are AND operated.

❑ Karnaugh Map Boolean Algebraic Simplification Technique

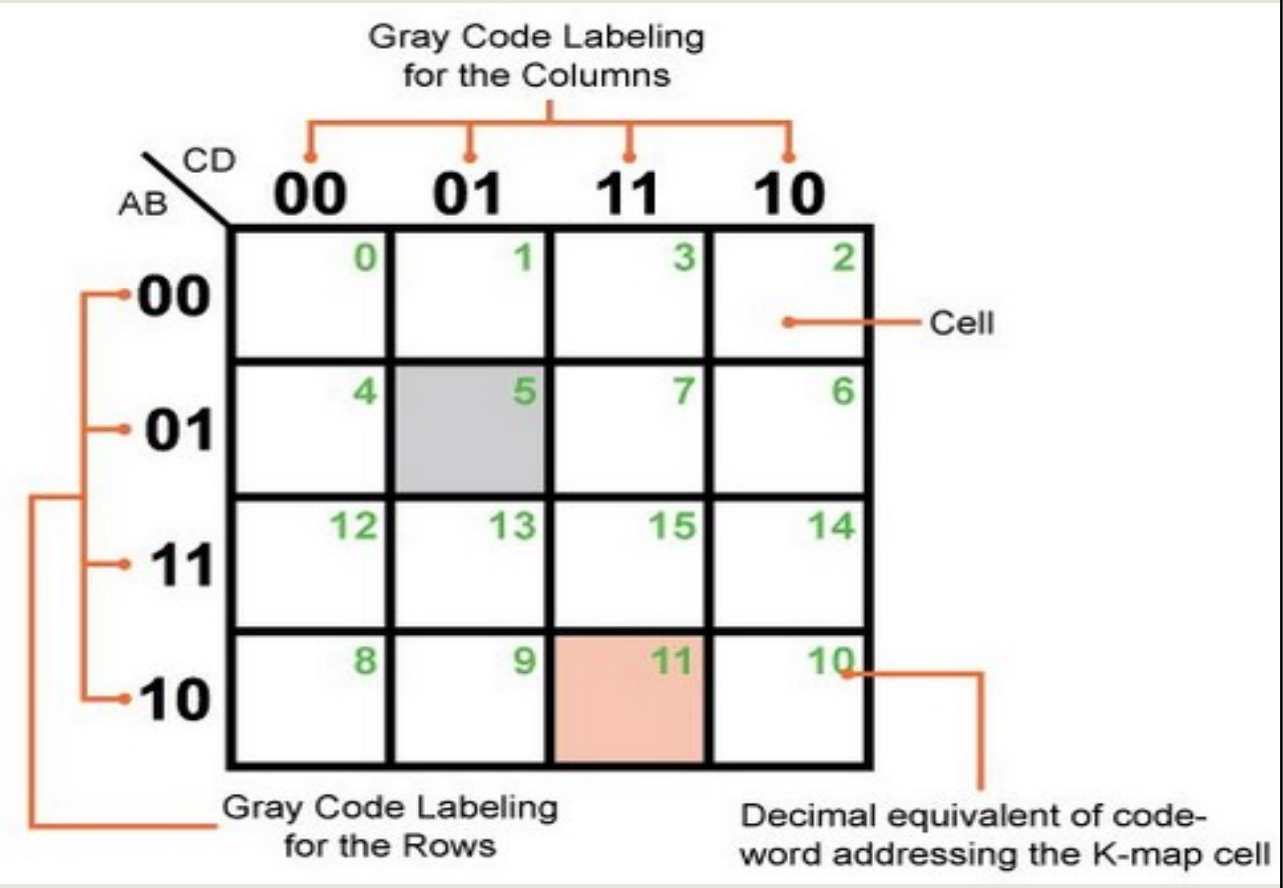
- Karnaugh map (K-map), introduced by Maurice Karnaugh in 1953.
- The K-map method of solving the logical expressions is referred to as the graphical technique of simplifying Boolean expressions. K-maps are also referred to as 2D truth tables .
- K-maps basically deal with the technique of inserting the values of the output variable in cells within a rectangle or square grid according to a definite pattern.
- The number of cells in the K-map is determined by the number of input variables and is mathematically expressed as two raised to the power of the number of input variables, i.e., 2^n, where the number of input variables is n.

❑ Karnaugh Map Boolean Algebraic Simplification Technique

- **Gray Coding**
- Each cell within a K-map has a definite place-value which is obtained by using an encoding technique known as Gray code.
- The specialty of this code is the fact that the adjacent code values differ only by a single bit.
- That is, if the given code-word is 01, then the previous and the next code-words can be 11 or 00, in any order, but cannot be 10 in any case.
- In K-maps, the rows and the columns of the table use Gray code-labeling which in turn represent the values of the corresponding input variables. This means that each K-map cell can be addressed using a unique Gray Code-Word.

❑ Karnaugh Map Boolean Algebraic Simplification Technique

- These concepts are further emphasized by a typical 16-celled K-map shown in below diagram, which can be used to simplify a logical expression comprising of 4-variables (A, B, C and D mentioned at its top-left corner).

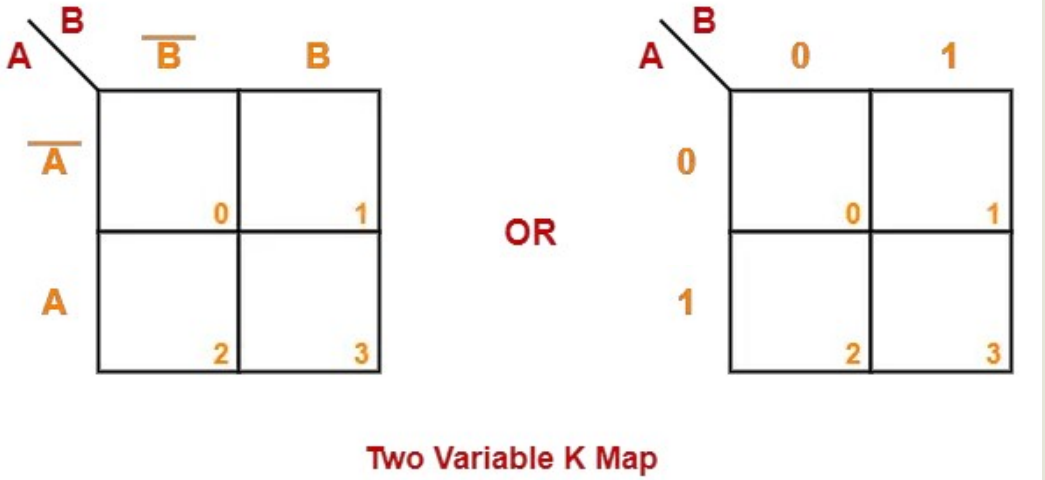


❑ Karnaugh Map Boolean Algebraic Simplification Technique

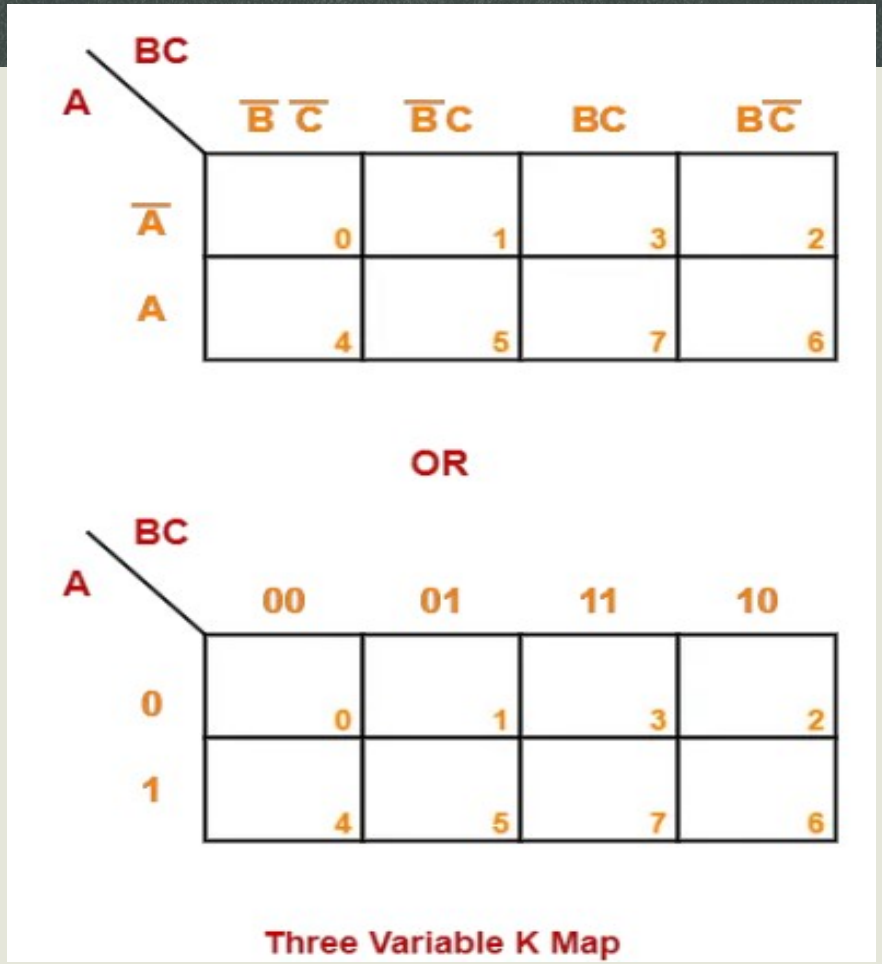
For example, the grey colored cell of the K-map shown can be addressed using the code-word "0101" which is equivalent to 5 in decimal (shown as the green number in the figure) and corresponds to the input variable combination $\overline{A}B\overline{C}D$ or $A+\overline{B}+C+\overline{D}$, depending on whether the input–output relationship is expressed in SOP (sum of products) form or POS (product of sums) form, respectively.

Similarly, $A\overline{B}CD$ or $\overline{A}+B+\overline{C}+\overline{D}$ refers to the Gray code-word of "1011", equivalent to 11 in decimal (again, shown in green in the figure), which in turn means that we are addressing the pink-colored K-map cell in the figure.

❑ Karnaugh Map Boolean Algebraic Simplification Technique



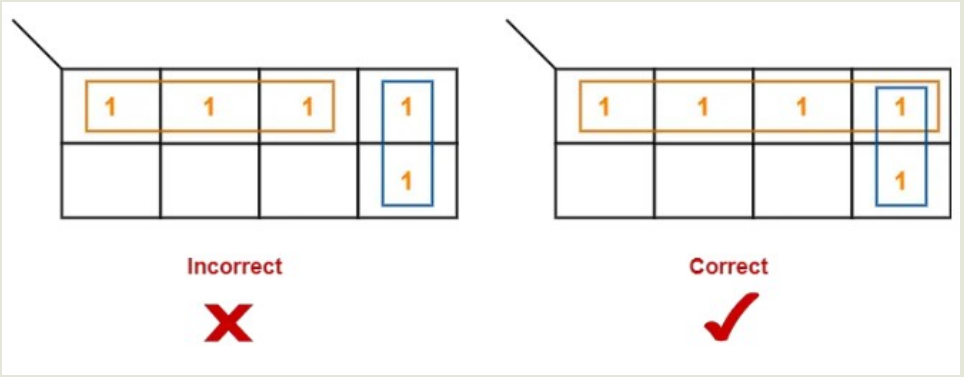
Two Variable K Map



Three Variable K Map

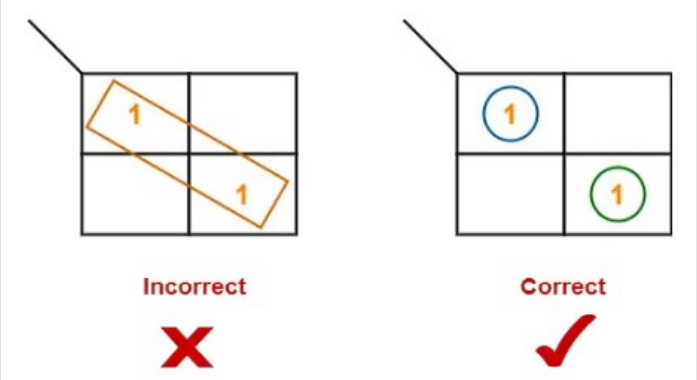
❑ Karnaugh Map Boolean Algebraic Simplification Technique

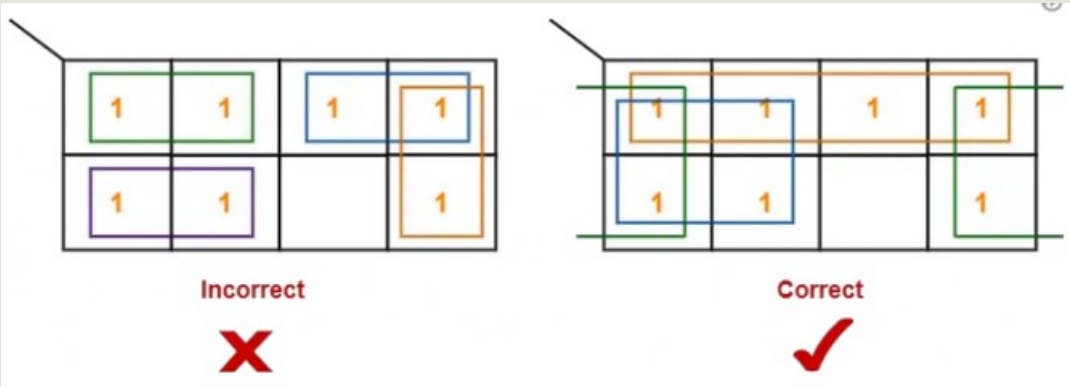
- **Rule-01:**
- We can either group 0's with 0's or 1's with 1's but we can not group 0's and 1's together.
- X representing don't care can be grouped with 0's as well as 1's.
- **Rule-02:** Groups may overlap each other.
- **Rule-03:**
- We can only create a group whose number of cells can be represented in the power of 2.
- In other words, a group can only contain 2^n i.e. 1, 2, 4, 8, 16 and so on number of cells.



❑ Karnaugh Map Boolean Algebraic Simplification Technique

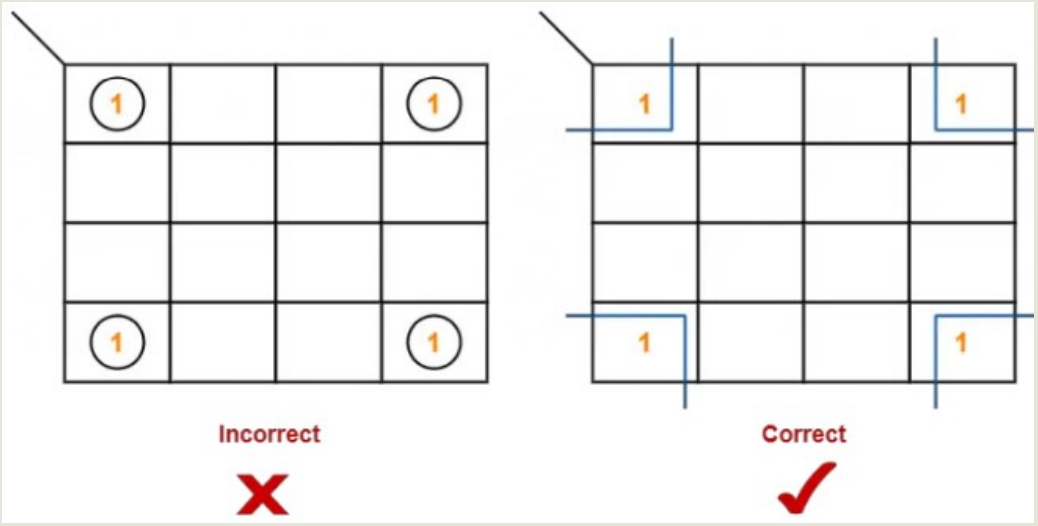
- **Rule-04:**
- Groups can be only either horizontal or vertical.
- We can not create groups of diagonal or any other shape.
- **Rule-05:**
- Each group should be as large as possible.





❑ Karnaugh Map Boolean Algebraic Simplification Technique

- **Rule-06:**
- Opposite grouping and corner grouping are allowed.
- The example of opposite grouping is shown illustrated in Rule-05.
- The example of corner grouping is shown below.



- **Rule-07:**
- There should be as few groups as possible.

❑ Karnaugh Map Boolean Algebraic Simplification Technique

- With this idea of K-maps, let us now move on to the procedure employed in designing an optimal (in terms of the number of gates used to realize the logic) digital system.
- We'll start with a given problem statement.
- **Example 1:**
- **Design a digital system whose output is defined as logically low if the 4-bit input binary number is a multiple of 3; otherwise, the output will be logically high. The output is defined if and only if the input binary number is greater than 2.**

❑ Karnaugh Map Boolean Algebraic Simplification Technique

- **Step 1: Truth Table**
- In the given example:
- Number of input variables = 4, which we will call A, B, C and D.
- Number of output variables = 1, which we will call Y where
- Y = Don't Care, if the input number is less than 3 (orange entries in the truth table)
- Y = 0, if the input number is an integral multiple of 3 (green entries in the truth table)
- Y = 1, if the input number is not an integral multiple of 3 (blue entries in the truth table)

❑ Karnaugh Map Boolean Algebraic Simplification Technique

Truth Table

Inputs				Decimal Equivalent	Output
A	B	C	D		Y
0	0	0	0	0	X
0	0	0	1	1	X
0	0	1	0	2	X
0	0	1	1	3	0
0	1	0	0	4	1
0	1	0	1	5	1
0	1	1	0	6	0
0	1	1	1	7	1
1	0	0	0	8	1
1	0	0	1	9	0
1	0	1	0	10	1
1	0	1	1	11	1
1	1	0	0	12	0
1	1	0	1	13	1
1	1	1	0	14	1
1	1	1	1	15	0

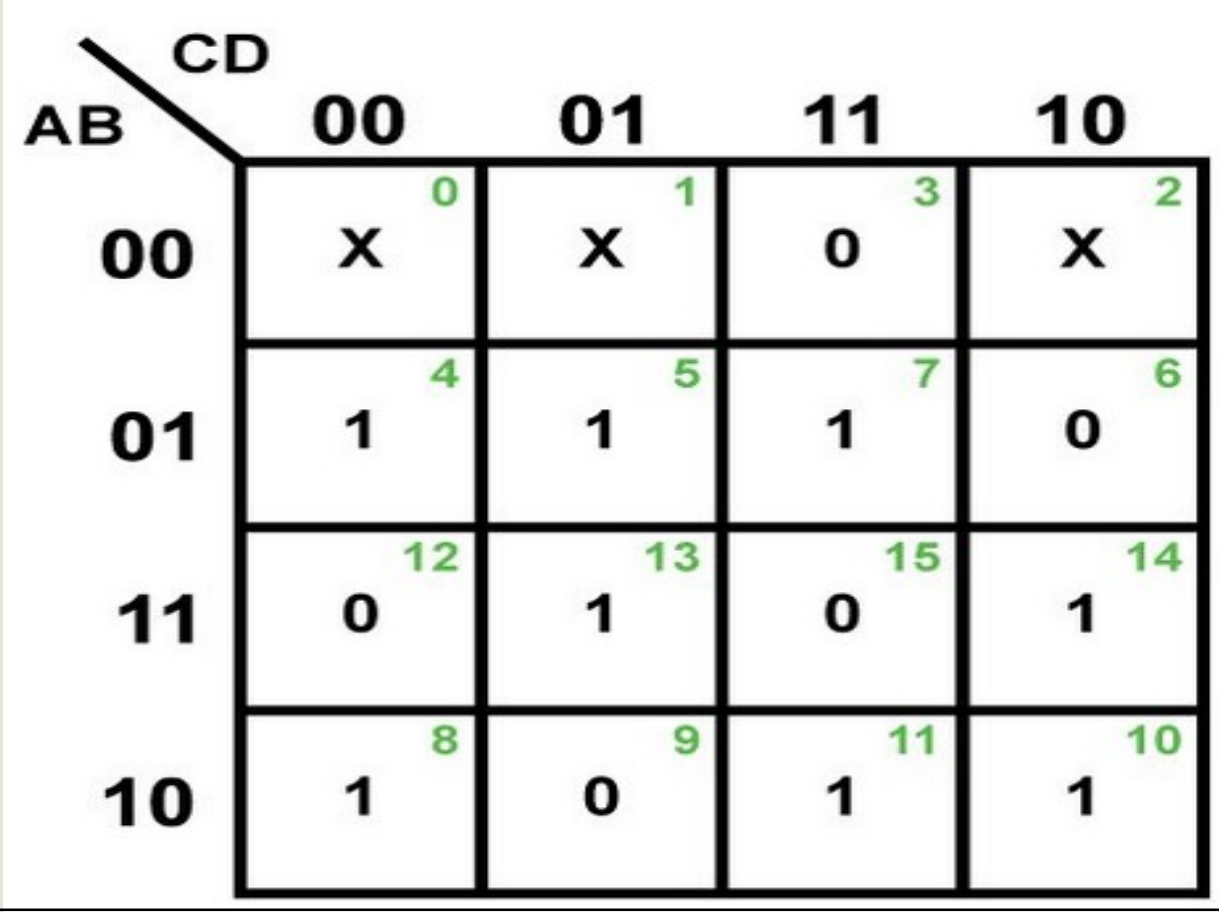
where X indicates Don't Care Condition

❑ Karnaugh Map Boolean Algebraic Simplification Technique

- **Step 2: Select and Populate K-Map**
- From Step 1, we know the number of input variables involved in the logical expression from which size of the K-map required will be decided.
- Further, we also know the number of such K-maps required to design the desired system as the number of output variables would also be known definitely.
- This means that, for the example considered, we require a single (due to one output variable) K-map with 16 cells (as there are four input variables).

❑ Karnaugh Map Boolean Algebraic Simplification Technique

- Next, we have to fill the K-map cells with one for each minterm, zero for each maxterm, and X for Don't Care terms. The procedure is to be repeated for every single output variable.
- Hence for this example, we get the K-map as shown in diagram.



❑ Karnaugh Map Boolean Algebraic Simplification Technique

- **Step 3: Form the Groups**
- K-map simplification can also be referred to as the "simplification by grouping" technique as it solely relies on the formation of clusters.
- That is, the main aim of the entire process is together as many ones (for SOP solution) or zeros (for POS solution) under one roof for each of the output variables in the problem stated.
- However, while doing so we have to strictly abide by certain rules and regulations:

❑ Karnaugh Map Boolean Algebraic Simplification Technique

- 1. The process has to be initiated by grouping the bits which lie in adjacent cells such that the group formed contains the maximum number of selected bits.
- This means that for an n-variable K-map with 2^n cells, try to group for 2^n cells first, then for 2^{n-1} cells, next for 2^{n-2} cells, and so on until the "group" contains only 2^0 cells, i.e., isolated bits (if any).
- Note that the number of cells in the group must be equal to an integer power to 2, i.e., 1, 2, 4, 8. . . .

❑ Karnaugh Map Boolean Algebraic Simplification Technique

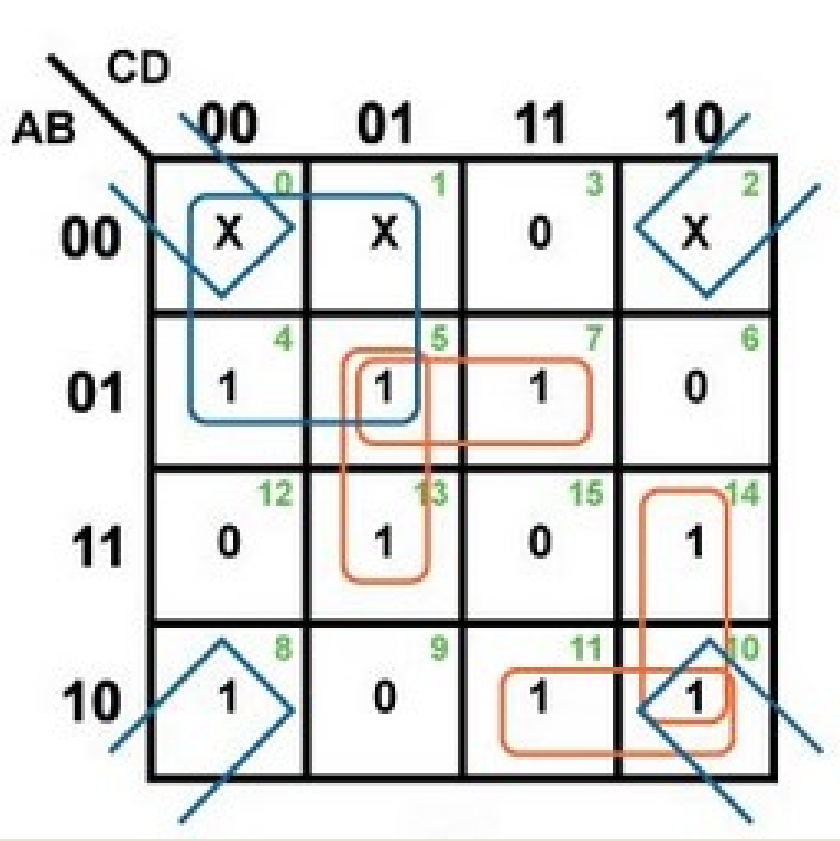
- 2. The procedure must be applied for all adjacent cells of the K-map, even when they appear to be not adjacent—the top row is considered to be adjacent to the bottom row and the rightmost column is considered to be adjacent to the leftmost column, as if the K-map wraps around from top to bottom and right to left.

❑ Karnaugh Map Boolean Algebraic Simplification Technique

- 3. A bit appearing in one group can be repeated in another group provided that this leads to the increase in the resulting group-size.
- 4. Don't Care conditions are to be considered for the grouping activity if and only if they help in obtaining a larger group. Otherwise, they are to be neglected.

❑ Karnaugh Map Boolean Algebraic Simplification Technique

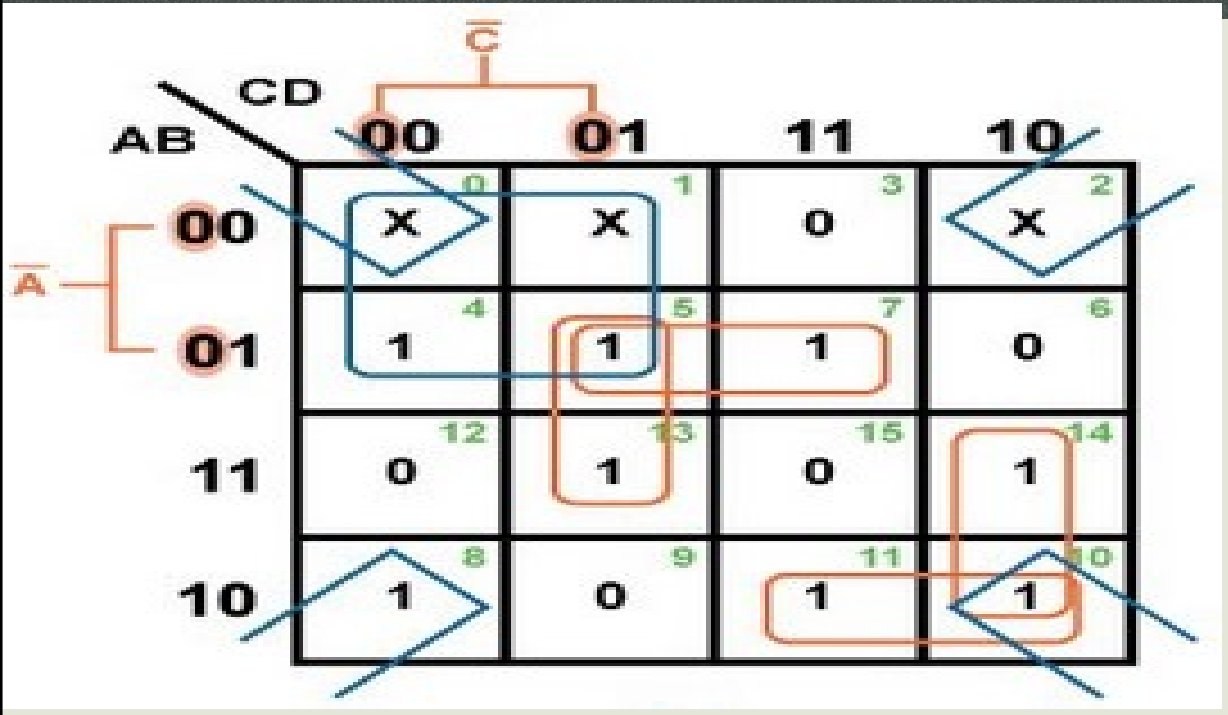
	SOP Form Solution		
Number of groups having 16 cells	0		
Number of groups having 8 cells	0		
Number of groups having 4 cells (Blue Enclosures in Figure 3)	2	Group 1 (Cells 0,2,8,10)	
		Group 2 (Cells 0,1,4,5)	
Number of groups having 2 cells (Orange Enclosures in Figure 3)	4	Group 3 (Cells 5,7)	Group 4 (Cells 5,13)
		Group 5 (Cells 10,11)	Group 6 (Cells 10,14)



❑ Karnaugh Map Boolean Algebraic Simplification Technique

- **Step 4: Simplified Logical Expression**
- For each of the resulting groups, we have to obtain the corresponding logical expression in terms of the input-variables.
- This can be done by expressing the bits which are common amongst the Gray code-words which represent the cells contained within the considered group.
- Finally, all these group-wise logical expressions need to be combined appropriately to form the simplified Boolean equation for the output variable.
- The same procedure must be repeated for every output variable of the given problem.

❑ Karnaugh Map Boolean Algebraic Simplification Technique

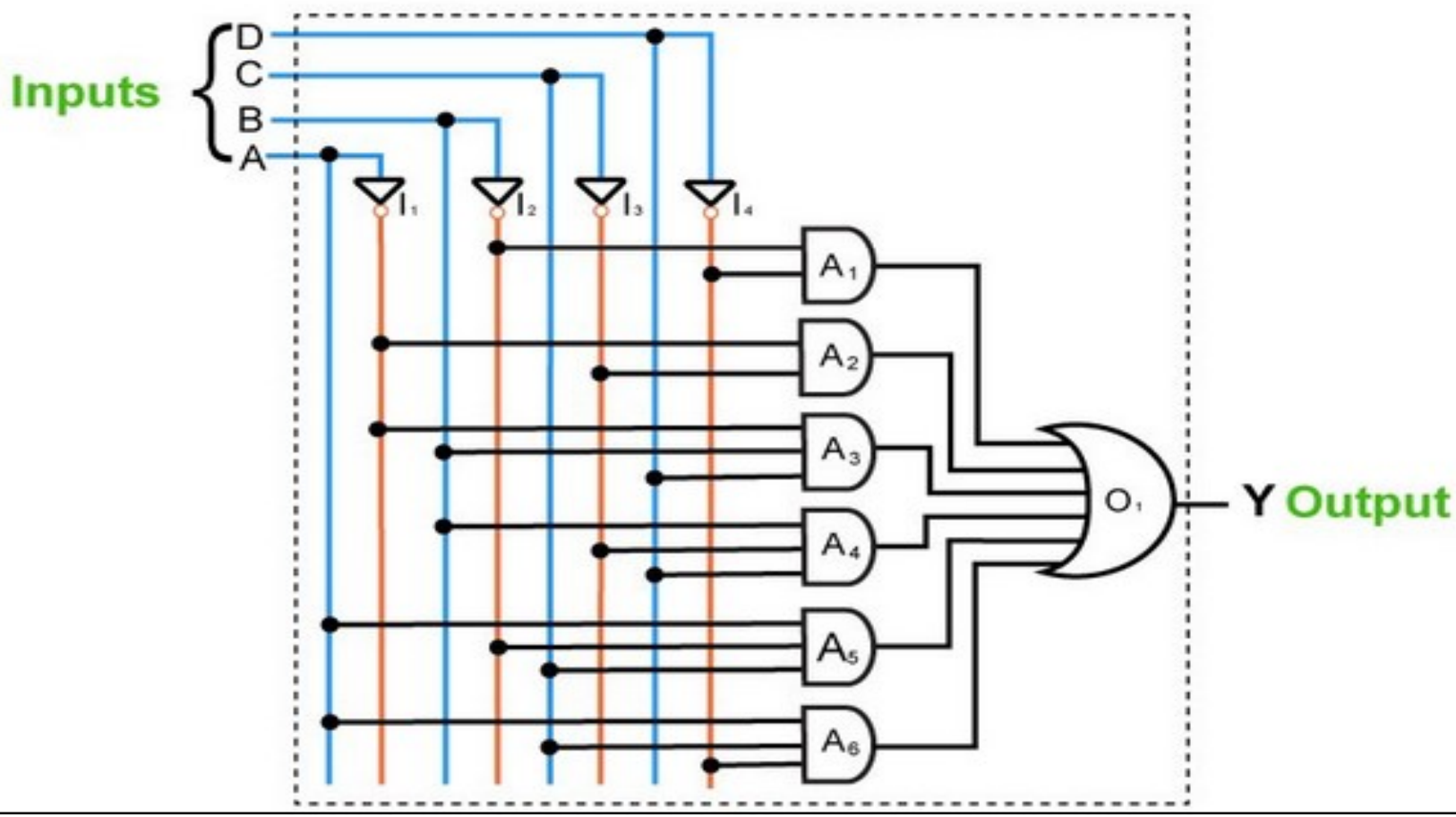


SOP Form Solution	
Groups	**Logical Expression**
Group 1	$\bar{B}\bar{D}$
Group 2	$\bar{A}\bar{C}$
Group 3	$\bar{A}BD$
Group 4	$B\bar{C}D$
Group 5	$A\bar{B}C$
Group 6	$AC\bar{D}$

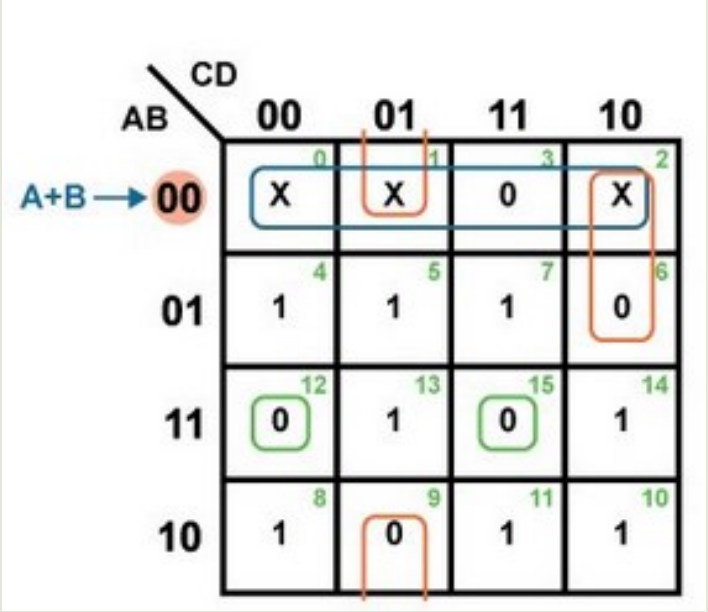
Thus, $Y = \bar{B}\bar{D} + \bar{A}\bar{C} + \bar{A}BD + B\bar{C}D + A\bar{B}C + AC\bar{D}$

❑ Karnaugh Map Boolean Algebraic Simplification Technique

- **Step 5: System Design**



❑ Karnaugh Map Boolean Algebraic Simplification Technique



POS Form Solution	
Groups	**Logical Expression**
Group 1	A+B
Group 2	B+C+$\overline{D}$
Group 3	A+$\overline{C}$+D
Group 4	$\overline{A}$+$\overline{B}$+C+D
Group 5	$\overline{A}$+$\overline{B}$+$\overline{C}$+$\overline{D}$

Thus, Y = (A+B) (B+C+$\overline{D}$) (A+$\overline{C}$+D) ($\overline{A}$+$\overline{B}$+C+D) ($\overline{A}$+$\overline{B}$+$\overline{C}$+$\overline{D}$)

❑ Don’t Care Conditions

- Don’t cares in a Karnaugh map, or truth table, may be either **1**s or **0**s, as long as we don’t care what the output is for an input condition we never expect to see.
- We plot these cells with an asterisk, * or x, among the normal **1**s and **0**s.
- When forming groups of cells, treat the don’t care cell as either a **1** or a **0**, or ignore the don’t cares.
- This is helpful if it allows us to form a larger group than would otherwise be possible without the don’t cares. There is no requirement to group all or any of the don’t cares.
- Only use them in a group if it simplifies the logic.

Exercise

- Minimize the following Boolean functions
- 1. $F(A, B, C, D) = \Sigma m(0, 2, 8, 10, 14) + \Sigma d(5, 15)$
- 2. $F(A, B, C) = \Sigma m(0, 1, 6, 7) + \Sigma d(3, 4, 5)$
- 3. $F(A, B, C) = \Sigma m(1, 2, 5, 7) + \Sigma d(0, 4, 6)$

❑ Exercise

- Minimize the following boolean functions
- 1 F(A, B, C, D) = ACD’ + B’D’
- 2. F(A, B, C) = A + B’
- 3. F(A, B, C) = A + B’ + C’

❑ Exercise

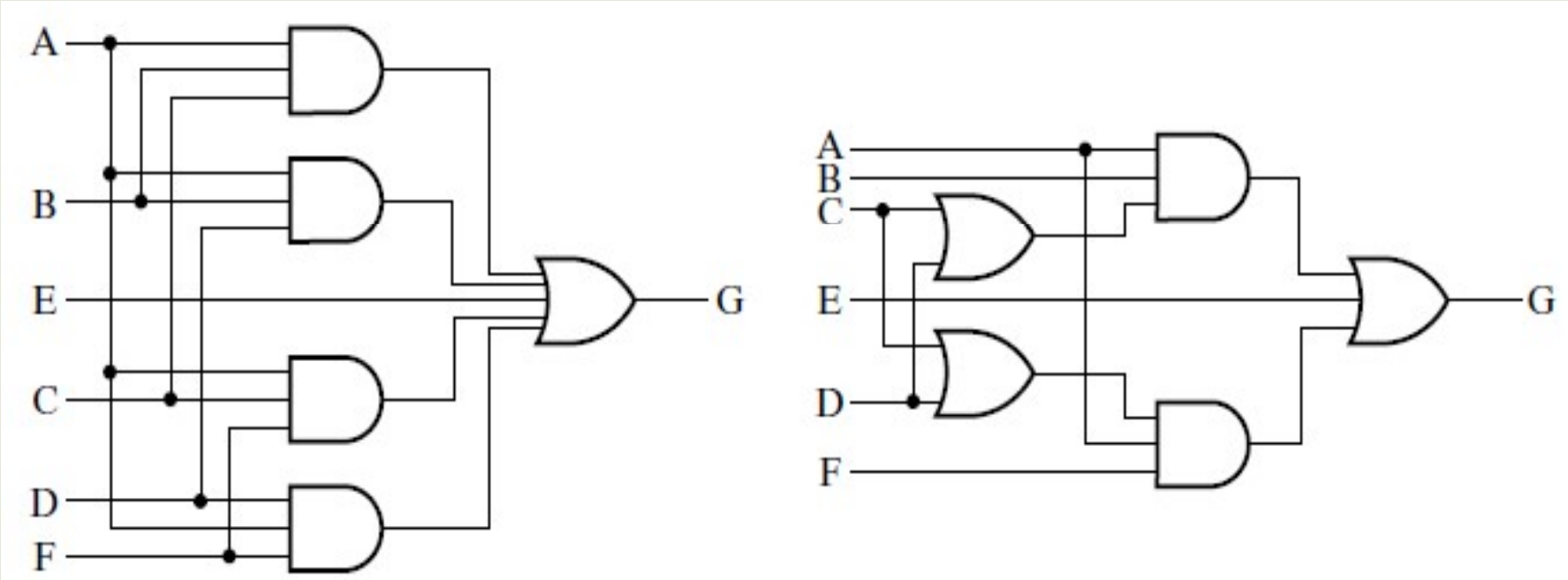
$$G = ABC + ABD + E + ACF + ADF$$

$$G = (AB + AF)(C + D) + E$$

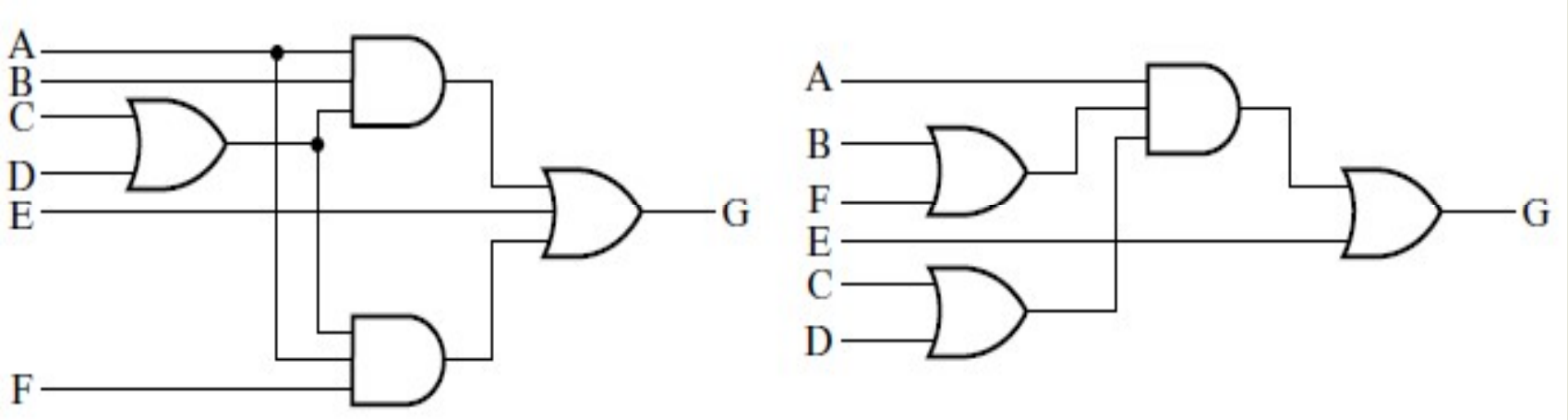
$$G = AB(C + D) + E + A(C + D)F$$

$$G = A(B + F)(C + D) + E$$

❑ Exercise



❑ Exercise



❑ Exercise

- (A + B).(A + C) = A + (B.C)

A.A + A.C + A.B + B.C	– Distributive law
A + A.C + A.B + B.C	– Idempotent AND law (A.A = A)
A(1 + C) + A.B + B.C	– Distributive law
A.1 + A.B + B.C	– Identity OR law (1 + C = 1)
A(1 + B) + B.C	– Distributive law
A.1 + B.C	– Identity OR law (1 + B = 1)
A + (B.C)	– Identity AND law (A.1 = A)

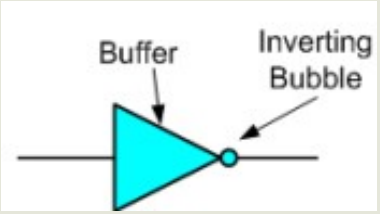
❑ Exercise

~(A * B) * (~A + B) * (~B + B)	=	~A

~(A * B) * (~A + B) * 1	Complement law
~(A * B) * (~A + B)	Identity law
(~A + ~B) * (~A + B)	DeMorgan's law
~A + ~B * B	Distributive law
~A + 0	Complement law
~A	Identity law

❑ Universal Gates

- In addition to AND, OR, and NOT gates, other logic gates like NAND and NOR are also used in the design of digital circuits. The NOT circuit inverts the logic sense of a binary signal.
- The small circle (bubble) at the output of the graphic symbol of a NOT gate is formally called a negation indicator and designates the logical complement.



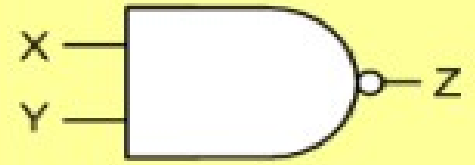
❑ Universal Gates

- A universal gate is a gate which can implement any Boolean function without need to use any other gate type. The NAND and NOR gates are universal gates.
- In practice, this is advantageous since NAND and NOR gates are economical and easier to fabricate and are the basic gates used in all IC digital logic families.
- In fact, an AND gate is typically implemented as a NAND gate followed by an inverter not the other way around. Likewise, an OR gate is typically implemented as a NOR gate followed by an inverter not the other way around.

❑ Universal Gates

- **NAND Gate**
- The NAND gate represents the complement of the AND operation. Its name is an abbreviation of NOT AND.
- The graphic symbol for the NAND gate consists of an AND symbol with a bubble on the output, denoting that a complement operation is performed on the output of the AND gate.
- The truth table and the graphic symbol of NAND gate is shown in the figure.
- The truth table clearly shows that the NAND operation is the complement of the AND.

X	Y	NAND
0	0	1
0	1	1
1	0	1
1	1	0

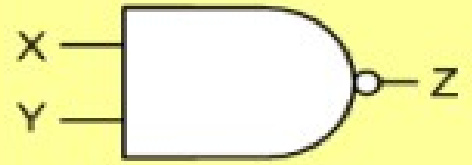


$$Z = \overline{X \cdot Y}$$

❑ Universal Gates

- **NOR Gate**
- The NOR gate represents the complement of the OR operation.
- Its name is an abbreviation of NOT OR.
- The graphic symbol for the NOR gate consists of an OR symbol with a bubble on the output, denoting that a complement operation is performed on the output of the OR gate.
- The truth table and the graphic symbol of NOR gate is shown in the figure.

X	Y	NAND
0	0	1
0	1	1
1	0	1
1	1	0

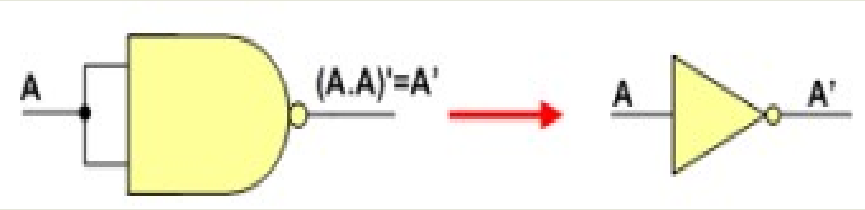


$$Z = \overline{X \cdot Y}$$

❑ Universal Gates

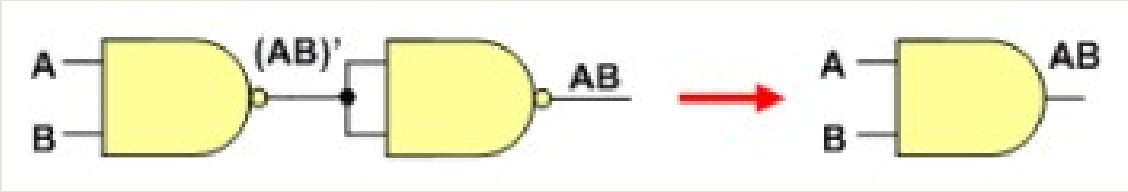
- **NAND Gate is a Universal Gate**
- To prove that any Boolean function can be implemented using only NAND gates, we will show that the AND, OR, and NOT operations can be performed using only these gates.
- Implementing an Inverter Using only NAND Gate The figure shows two ways in which a NAND gate can be used as an inverter (NOT gate).
- 1. All NAND input pins connect to the input signal A gives an output A'.

- 2. One NAND input pin is connected to the input signal A while all other input pins are connected to logic 1. The output will be A'.

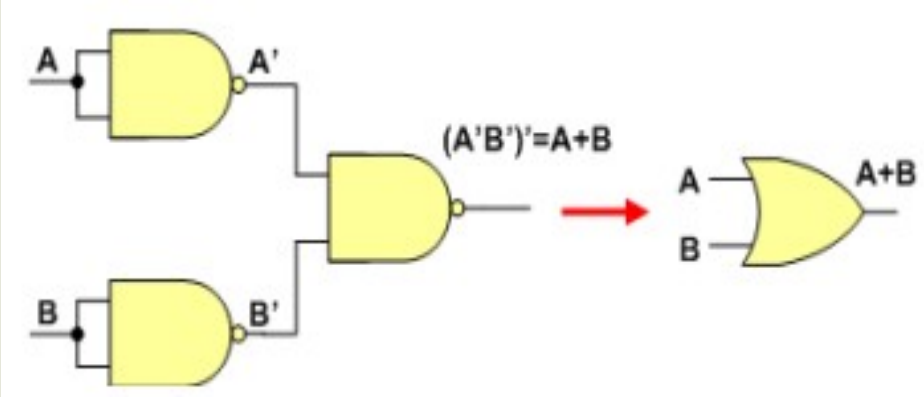


❑ Universal Gates

- Implementing AND Using only NAND Gates An AND gate can be replaced by NAND gates as shown in the figure (The AND is replaced by a NAND gate with its output complemented by a NAND gate inverter).



- Implementing OR Using only NAND Gates An OR gate can be replaced by NAND gates as shown in the figure (The OR gate is replaced by a NAND gate with all its inputs complemented by NAND gate inverters).



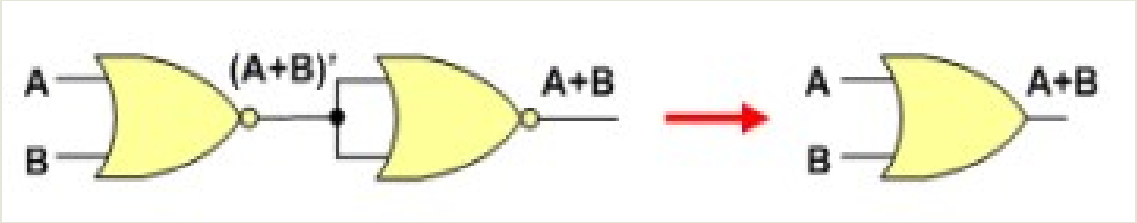
❑ Universal Gates

- **NOR Gate is a Universal Gate**
- The figure shows two ways in which a NOR gate can be used as an inverter (NOT gate). 1. All NOR input pins connect to the input signal A gives an output A’

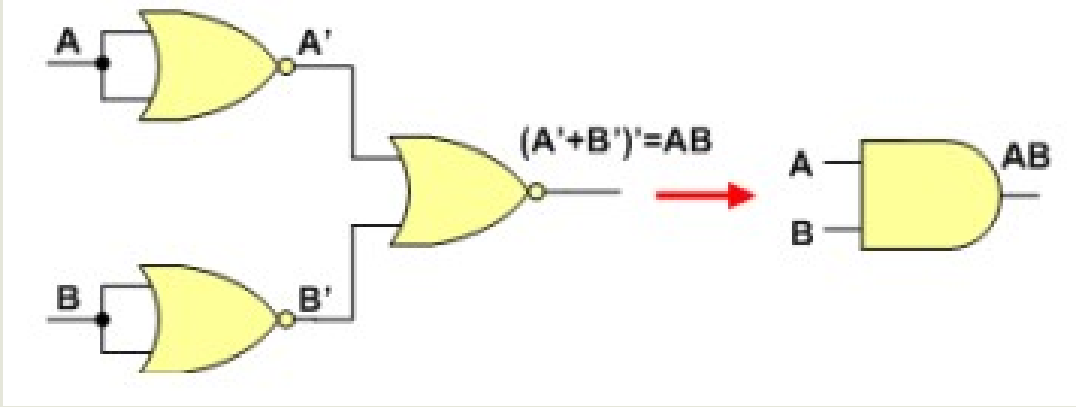
- 2. One NOR input pin is connected to the input signal A while all other input pins are connected to logic 0. The output will be A’.

❑ Universal Gates

- Implementing OR Using only NOR Gates An OR gate can be replaced by NOR gates as shown in the figure (The OR is replaced by a NOR gate with its output complemented by a NOR gate inverter)

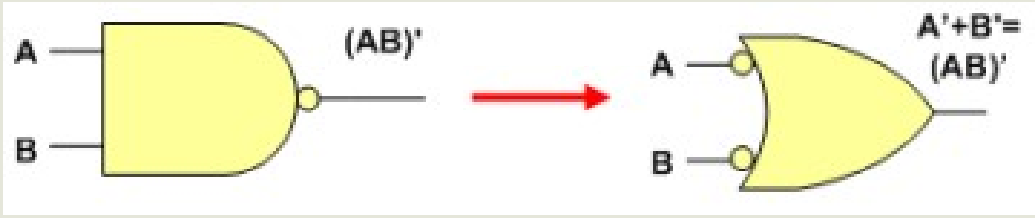


- Implementing AND Using only NOR Gates An AND gate can be replaced by NOR gates as shown in the figure (The AND gate is replaced by a NOR gate with all its inputs complemented by NOR gate inverters)

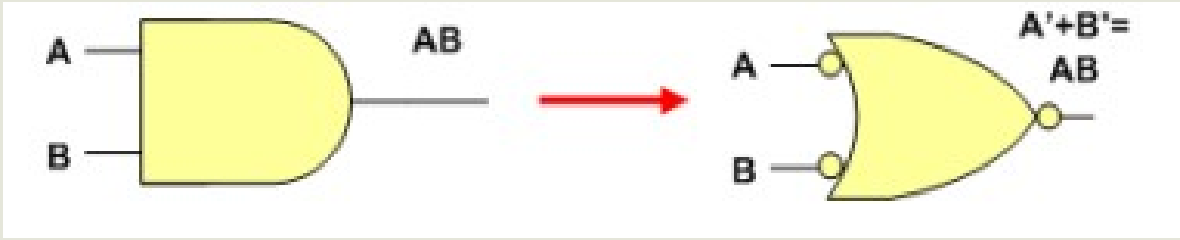


❑ Universal Gates

- **Equivalent Gates:**
- The shown figure summarizes important cases of gate equivalence. Note that bubbles indicate a complement operation (inverter). A NAND gate is equivalent to an inverted-input OR gate

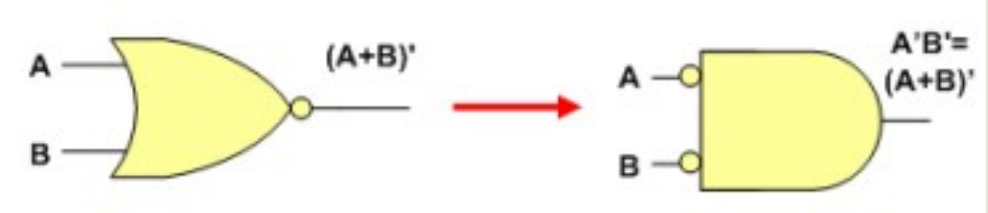


- An AND gate is equivalent to an inverted-input NOR gate.



❑ Universal Gates

- A NOR gate is equivalent to an inverted-input AND gate.

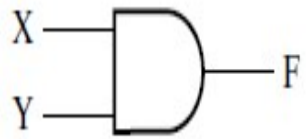


- An OR gate is equivalent to an inverted-input NAND gate.

- Two NOT gates in series are same as a buffer because they cancel each other as A'' = A.

❑ OTHER GATE TYPES

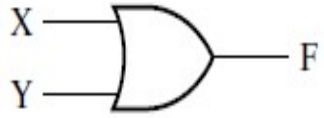
AND



$F = XY$

X	Y	F
0	0	0
0	1	0
1	0	0
1	1	1

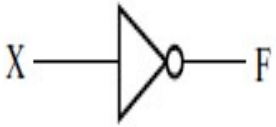
OR



$F = X + Y$

X	Y	F
0	0	0
0	1	1
1	0	1
1	1	1

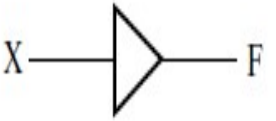
NOT
(inverter)



$F = \overline{X}$

X	F
0	1
1	0

Buffer



$F = X$

X	F
0	0
1	1

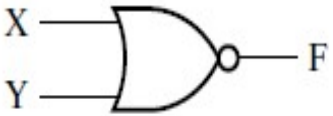
❑ OTHER GATE TYPES

NAND

$F = \overline{X \cdot Y}$

X	Y	F
0	0	1
0	1	1
1	0	1
1	1	0

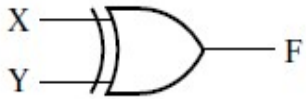
NOR



$F = \overline{X + Y}$

X	Y	F
0	0	1
0	1	0
1	0	0
1	1	0

Exclusive-OR (XOR)

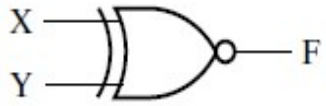


$F = X\overline{Y} + \overline{X}Y$
$= X \oplus Y$

X	Y	F
0	0	0
0	1	1
1	0	1
1	1	0

❑ OTHER GATE TYPES

Exclusive-NOR
(XNOR)

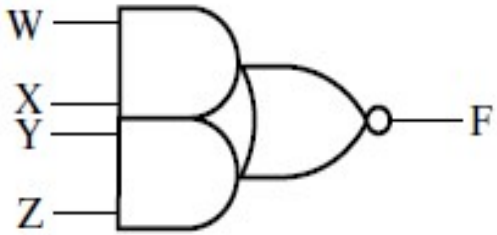


$$F = XY + \overline{X}\,\overline{Y}$$

$$= X \oplus Y$$

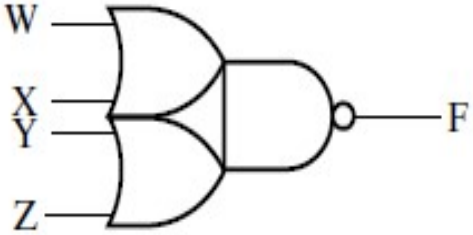
X	Y	F
0	0	1
0	1	0
1	0	0
1	1	1

AND-OR-INVERT
(AOI)



$$F = \overline{WX + YZ}$$

OR-AND -INVERT
(OAI)



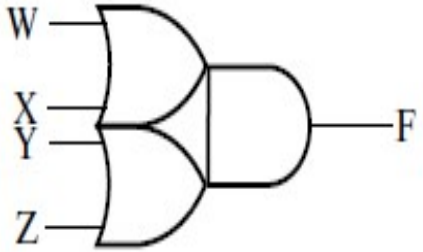
$$F = \overline{(W + X)(Y + Z)}$$

❑ OTHER GATE TYPES

AND-OR
(AO)

$F = WX + YZ$

OR-AND
(OA)



$F = (W + X)(Y + Z)$