UNIT 3 LOGIC CIRCUITS - AN INTRODUCTION

Structure **Page Nos.**

3.0 INTRODUCTION

In the previous units, we have discussed the basic configuration of computer system, simple instruction execution, data representation and different computer organisations. In addition, ISA and micro-architecture was discussed in unit 1 of this block. It may be noted that, as stated in the unit1, gates and logic circuits are the building blocks of a computer system. This unit introduces you to some of the basic components of computer system that are essential for learning the logic of binary computing devices. In this unit, you will be introduced to the concepts of logic gates, binary adders, logic circuits and combinational circuits. These circuits form the backbone of any computer system and knowing them will be useful in lower level programming.

3.1 OBJECTIVES

After going through this unit you will be able to:

- explain the basic functions of logic gates;
- describe role of Boolean algebra in digital circuits;
- perform the minimization the number of gates for a Boolean expression;
- identify and explain the basic circuits in a computer system
- design simple combinational circuits.

3.2 LOGIC GATES

A computer system is a binary device that uses electronic signals to perform basic computation on the digital data, which is also stored electronically. The

basis of such computation, called digital logic, are the electronic circuits fabricated on the semi-conductor chips that are used to formulate a set of operations. These basic sets of operations are then used to create complex circuitry, which is able to perform arithmetic, logical and control operations in a computer system. Thus, the simplest form of binary logic is: how a set of inputs can be used to create a typical output sequence, which is achieved using electronic gates.

A logic gate is an electronic circuit made of transistors, which produces a characteristic output signal for a typical input. In general, the input accepts one to several input values and produces a specific output. This output can be 0 or 1. Logic gates are used for implementing basic Boolean operations, which is explained in the subsequent sections. A logic gate is represented using a graphics symbol and performs a simple binary function, which can be represented with the help of a truth table. Figure 3.1 shows the basic logic gates. Please note that in Figure 3.1, the character *I* represent input values and *F* represent output values.

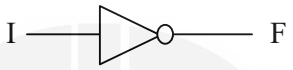
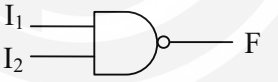
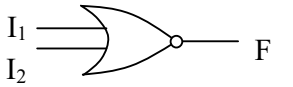
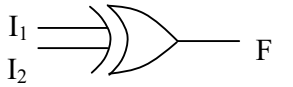
Gate	Graphical Symbol	Truth Table	Description
NOT		I F 0 1 1 0	It simply inverts the input value, i.e. input 0 will be converted to 1 and vice-versa.
OR		I1 I2 F 0 0 0 0 1 1 1 0 1 1 1 1	This gates output a value 1, if at least one of its input is 1.
AND		I1 I2 F 0 0 0 0 1 0 1 0 0 1 1 1	This gate output is 1, if both the input values are 1 else output is 0.
NAND		I1 I2 F 0 0 1 0 1 1 1 0 1 1 1 0	This is NOT AND, so where ever AND gate produce 0, this gate outputs 1.
NOR		I1 I2 F 0 0 1 0 1 0 1 0 0 1 1 0	This is NOT OR.
XOR		I1 I2 F 0 0 0 0 1 1 1 0 1 1 1 0	Exclusive OR produces output 1 when the two input are dissimilar.

Figure 3.1: Logic Gates

In the next few sections, we explain how these simple logic gates can be used to construct logic circuits. The next section explains the mathematics of logic circuits.

3.3 BOOLEAN ALGEBRA

Boolean algebra was designed by George Boole in the 19th century. It presents mathematical foundation for performing various functions on binary variables. Please recall that binary variables can have only two values 0 or 1. The value 0 by convention is taken as False and 1 as True. Please also refer to Figure 3.1, which shows the truth table for various gates. These truth tables can also be represented using the Boolean function. Figure 3.2 shows the Boolean algebraic representation of logic gates of Figure 3.1.

Gate	Boolean Representation	Explanation
NOT	$F = I'$	The symbol '′' in Boolean expression represents negation operator. Thus, output F is complement or negation of the value of I.
OR	$F = I_1 + I_2$	The OR is represented by Boolean operator '+', which represents that the value of F be zero only if both I_1 and I_2 are zero else 1.
AND	$F = I_1 . I_2$	The Boolean operator '.' represents AND operation. The value of F be 1 if both I_1 and I_2 are 1, else F will be 0.
NAND	$F = (I_1 . I_2)'$	NOT of AND
NOR	$F = (I_1 + I_2)'$	NOT of OR
XOR	$F = I_1 \oplus I_2$	$\oplus$ is an exclusive – OR operator.

Figure 3.2: Gates and related Boolean algebraic expression.

The Boolean algebra is very useful for mathematically representing a binary operation. For example, addition of two binary digits can be represented in truth table form as:

I_1	I_2	Carry (C)	Sum (S)
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

∴ It can be represented using two Boolean functions, one for each output, viz. Carry and Sum, as:

$C = I_1 . I_2$ and

$S = I_1 \oplus I_2$ (Please refer to Fig. 3.1 & Fig 3.2)

The Boolean algebra is used to simplify logic circuits that are made of logic gates. However, before we demonstrate this process of simplification, first you may go through the basic rules of Boolean algebra. Figure 3.3 shows these rules. Please note that some of the rules are shown with proof using truth table. You can make truth table yourself for the cases for which the proof is not shown.

(i)

Input	Identities			
I	$I + 0 = I$	$I + 1 = 1$	$I \cdot 0 = 0$	$I \cdot 1 = I$
0 1	0 + 0 = 0 1 + 0 = 1	0 + 1 = 1 1 + 1 = 1	0 . 0 = 0 1 . 0 = 0	0 . 1 = 0 1 . 1 = 1

(ii)

Input	Identities			
I	$I + I = I$	$I + I' = 1$	$I \cdot I = I$	$I \cdot I' = 0$
0 1	0 + 0 = 0 1 + 1 = 1	0 + 1 = 1 1 + 0 = 1	0 . 0 = 0 1 . 1 = 1	0 . 1 = 0 1 . 0 = 0

(Please note $0' = 1$ and $1' = 0$)

(iii) The rules (given without proof)

$I_1+I_2=I_2+I_1$;

$I_1 . I_2 = I_2 . I_1$;

$I_1+(I_2+I_3)=(I_1+I_2)+I_3$;

$I_1.(I_2.I_3)=(I_1.I_2).I_3$

(iv) The rules (given without proof)

$I_1 . (I_2+I_3) = (I_1 . I_2 + I_1.I_3)$;

$I_1+I_2.I_3=(I_1+I_2) . (I_1+I_3)$

(v) Demorgan's Laws:

$(I_1+I_2)' = I_1'.I_2'$

$(I_1.I_2)' = I_1' + I_2'$

(Very important laws for algebraic simplification.)

(vi) Complement of complement of a number is the Number itself

I	I'	$(I')'$
0	1	0
1	0	1

so $(I')' = I$

Figure 3.3: The Rules of Boolean algebra

All the rules and identities as given in Figure 3.3 can be used for simplification of Boolean function. This is explained with the help of following example.

Example: Simplify the Boolean function:

$F = ((A'+B')' + (A.B)')'$

Solution:

$F = ((A'+B')' + (A.B)')'$

$F = ((A'+B')')' . ((A.B)')'$ (Using Demorgan's Law)

$= (A'+B') . (A.B)$ Using Rule (vi)

$= (A.B) . (A'+B')$ Reversing the terms - Rule (iii)

$= ((A.B).A') + ((A.B) . B')$ Using Rule (vi) taking (A.B) as I_1

$= ((A.A').B) + (A.(B.B')$ Using Rule (iii)

$= 0.B + A.0$ Using Rule (ii)

$= 0+0$ Using Rule (i)

$= 0$

$F = 0$

You can check the above using the following Truth Table

A	B	A′	B′	(A′+B′)	(A.B)	(A′+B′)′	(A.B)′	(A′+B′)′ + (A.B)′	((A′+B′)′+(A.B)′)′
0	0	1	1	1	0	0	1	1	0
0	1	1	0	1	0	0	1	1	0
1	0	0	1	1	0	0	1	1	0
1	1	0	0	0	1	1	0	1	0

The Boolean algebra is very useful in simplification of logic circuits. It is explained in the next section.

3.4 LOGIC CIRCUITS

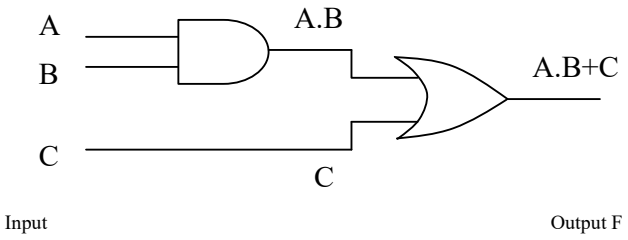
A logic circuit performs the basic operation on binary data. The operation of a logic circuit can be represented using a Boolean function, or in other words a Boolean function can be implemented as a logic circuit. The three basic gates that can be used for drawing logic circuits are - AND, OR and NOT gates. Consider, for example, the Boolean function: -

F (A,B,C) = A.B+C

The relationship between this function and its binary variables A, B, C can be represented in a truth table as shown in Figure 3.4(a). Figure 3.4(b) shows the corresponding logic circuit.

Inputs			Output
A	B	C	F= A.B+C
0	0	0	F = 0 . 0 + 0 = 0
0	0	1	F = 0 . 0 + 1 = 1
0	1	0	F = 0 . 1 + 0 = 0
0	1	1	F = 0 . 1 + 1 = 1
1	0	0	F = 1 . 0 + 0 = 0
1	0	1	F = 1 . 0 + 1 = 1
1	1	0	F = 1 . 1 + 0 = 1
1	1	1	F = 1 . 1 + 1 = 1

(a) Truth Table



(b) Logic Circuit

Figure 3.4: Truth table & logic diagram for F = A . B + C

While fabricating these logic circuits, it is expected that fewer gate types are used; however, these gate types should be able to create all kinds of circuits. Therefore, functionally complete set of gates, which are a set of gates by which *any* Boolean function may be implemented, are used to fabricate the logic circuits. Examples of functionally complete sets are: [AND, OR, NOT]; [NOR]; [NAND] etc. NAND gate, also called universal gate, is a special gate and can be used for fabrication of all kinds of circuits. You may refer to further readings for more details on Universal gates.

Check Your Progress 1

1) What is a logic gate? What is the meaning of term Universal gate?

.....

.....

2) Prove the identity $I_1+I_2.I_3 = (I_1+I_2).(I_1+I_3)$ using Truth Table

…….

…….

3) Simplify the function $F = ((A'+B)'+(A.B')')'$

…….

…….

…….

…….

…….

4) Draw the logic diagram of the function before simplification.

.....

.....

.....

……..

5) Draw the logic diagram of the simplified function.

.....

.....

…….

3.5 COMBINATIONAL CIRCUITS

Combinational circuit is an interconnection of gates, which produces one or more output based on some Boolean function for which it has been designed. A good combinational circuit does not include feedback loops. A combinational circuit is also represented using equivalent truth table or a Boolean function.

The output of the combinational circuit changes instantaneously with respect of input, though some delay is introduced due to transfer of signal from the circuit. This delay is dependent on the *depth* which is computed as number of gates in the longest path from input to output. For example, the depth of the combinational circuit of Figure 3.5 is 2.

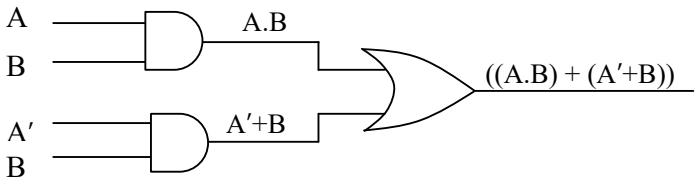


Figure 3.5: A two level AND-OR combinational circuit

Combinational circuits are primarily used to create the computational circuits of computer system logic; therefore, efficient design of combinational circuit may enhance the performance of a computer. Thus, one of the design goals of combinational circuits design is to minimize the number of gates in a combinational circuit. The constraints for combinational circuit design are:

- a combinational circuit should have limited depth.
- The number of input lines to a gate and number of gates to which the output of gate is fed, should be limited.

How can it be achieved? The following sub-sections explains the basic issues for combinational circuit design.

3.5.1 Canonical and Standard Forms of a Boolean expression.

An algebraic expression can exist in two standard forms:

i) Sum of Products (SOP) form
ii) Product of Sums (POS) form

Sum of product form

A SOP expression consists of terms consisting of operator (AND) which are joined by + operator (OR). For example, the expression $A'.B.C + A.B$ is an expression consisting of three variables A, B and C. This expression is in SOP from with two terms $A'.B.C$ and $A.B$. Each of these terms consists of product of variables using AND (.) operator, and the two terms are joined by an OR(+) operator, that is why the name Sum of Products form. In a SOP expression, a term which includes every variable in normal or complement form is called a minterm or standard product term. For example, in the above expression the term A′.B.C is a minterm, but A.B is not a minterm. However, if needed the term A.B can be converted to two minterms as:

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

In addition, please note that the value of minterm be ONE for exactly one possible combination of input values of A, B and C variables. For example, the minterm A′.B.C will have a value 1 if A′=1 and B=1 and C=1. i.e., A=0, B=1 and C=1; for any other combination of values of A, B, C the minterm will have a ZERO value.

Interestingly, the number of minterms depends on number of variables. Given, n variables, the number of minterms will be 2^n. For example, for two variables,

$n=2 \Rightarrow 2^n=2^2=4$. The possible minterms for two variables are shown in the Figure 3.6.

Variables		Minterm	
A	B		
0	0	A′B′	m_0
0	1	A′B	m_1
1	0	AB′	m_2
1	1	AB	m_3

Figure 3.6: Minterms for two variables

A function can be represented as a sum of minterms, for example a function F in two variables using minterms A′B + AB can be represented as:

F(A,B)= A′B + AB
which can be represented as:
F(A,B) = ∑ (1,3)
(Please note that A′B is minterm m_1 or 1 and AB is minterm m_3 or 3,

Product of Sum form

In this form an expression is written as the product (AND) of the terms, which use OR(+) as the basic operation, e.g. a three variables expression (A+B′+C).(A′+B′) is in POS form having the two terms (A+B′+C) and (A′+B′). Both the terms uses + operator and are joined by the AND operator, thus, the name Product of Sum form. In POS form a term, which include all the variables either in normal or complemented form is called a maxterm. For example, the expression (A+B′+C).(A′+B′) has a maxterm (A+B′+C). A maxterm can have a value 0 for exactly one combination of input. For example the maxterm A+B′+C will have value 0 if A=0, B′=0 and C=0, which is A=0, B=1 and C=0.
For any other combination of values of A, B and C, it will have a value 1.
The following Figure shows the maxterms. Please note that the output of maxterm is 0, only for the given combination of input.

A	B	Maxterm	
0	0	A+B	M_0
0	1	A+B′	M_1
1	0	A′+B	M_2
1	1	A′+B′	M_3

Figure 3.7: Maxterms for two variables

Example: Represent the function, whose SOP form is given below into an equivalent function in POS form.

F(A,B) = A′.B + A.B or F(A,B) = ∑ (1,3) or the truth table representation is:

A	B	F(A,B)
0	0	0
0	1	1
1	0	0
1	1	1

Solution:
The complement of this function in SOP form is represented as (the minterms that has 0 as function output).

$$F'(A,B) = A'.B' + A.B' \quad \text{--------(1)}$$

Taking complement of equation (1) , you will get the function F in POS form.

$$(F'(A,B))' = (A'.B'+A.B')'$$

$$\Rightarrow \quad F(A,B) = (A'.B')'+(A.B')'$$

$$= ((A')'+(B')').(A'+(B')')$$

$$= (A+B) . (A'+B)$$

Form table you can determine that function in POS form is:

$F(A,B) = \prod (0,2)$ as the terms are M_0 and M_2

Thus, you can see:

$$F(A,B) = \sum (1,3) = \prod (0,2)$$

(SOP form) (POS form)

With this background of minterm and maxterm, you now are ready to perform the process of grouping of minterms, which will result in minimization of gates needed for a digital circuit. This is discussed in the next section.

3.5.2 Minimization of Gates

The simplification of Boolean expression is useful for the design of a good combinational circuit. There are several methods of doing so, however in this unit only the following two methods are discussed in details.

- Algebraic Simplification
- Karnaugh Maps

Algebraic Simplification

The following example explains the process of algebra simplification

Example : Simplify the function: $F(A,B,C) = \sum (0,1,4,5,6,7)$

Solution: Expanding the Minterms of the functions as:

$$F(A,B,C) = A'.B'.C'+A'.B'.C + A.B'.C' + A.B'.C + A.B.C' + A.B.C$$

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

$$= A'.B'.1 +A.B'.1+A.B.1 \quad (\text{as } C'+C = 1)$$

$$= (A'.B'+A.B') + A.B$$

$$= B' (A'+A) + A.B$$

$$= B'+A.B$$

Please note that C input has no effect on the function.

The truth table for the function and the equivalent expression is:

A	B	C	F(A,B,C) = ∑ (0,1,4,5,6,7)	B′+A.B
0	0	0	1	1
0	0	1	1	1
0	1	0	0	0
0	1	1	0	0
1	0	0	1	1
1	0	1	1	1
1	1	0	1	1
1	1	1	1	1

Thus, the logic circuit for the simplified equation F(A,B,C) = AB+B′

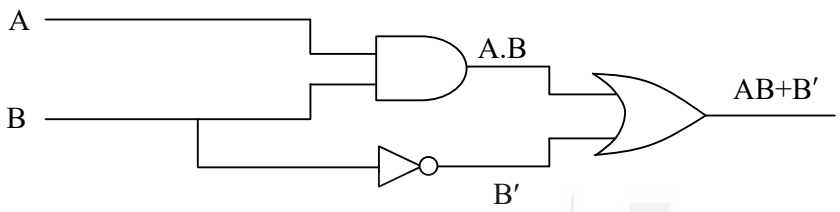


Figure 3.8: Simplified logic function using algebraic Simplifications

The logic diagram of the simplified expression is drawn using one NOT, OR and AND gate each.

The algebraic simplification becomes cumbersome because it is not clear which simplification should be applied next. The Karnaugh map is a simplified process of design of logic circuit using graphical approach. This is discussed next.

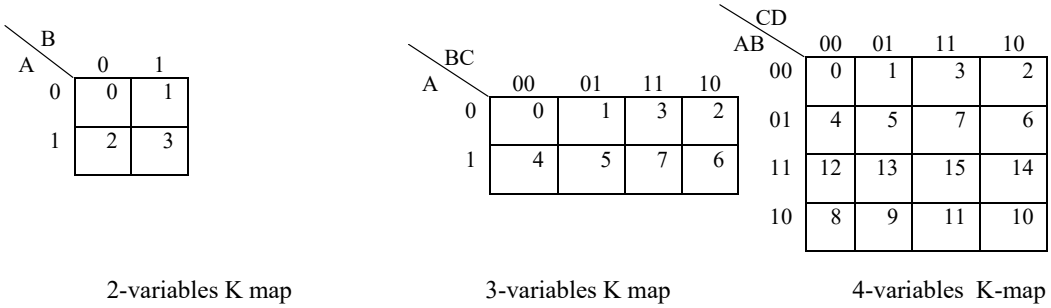
Karnaugh Maps

Karnaugh map is graphical way of representing and simplifying a Boolean function. They are useful for design of circuits involving 2 to 6 variables. The following is the process for simplification of logic circuit using Karnaugh map (K map).

Step 1: Create a Rectangular K-map and Assign binary and decimal equivalent values to each cell

Create a rectangular grid of variables in a function. Figure 3.9 shows the map of two, three and four variables. A map of 2 variables consists of a grid $2^2 = 4$ elements or cells, while a map of 3 variables has $2^3 = 8$ cells and 4 variables has $2^4 = 16$ cells. Please note that the number of cells are same as the maximum possible number of minterms for those number of variables.
Each cell corresponds to a set of variable values, shown on the top or left of the K-map. For example, the values 00, 01, 11, 10 are written on the top of the cells of K-maps of 3 and 4 variables. These represent the values of the variables. For example, for the 3-variable k-map values written on BC side for the first cell 00 indicate B=0 and C=0. Please note that variable values are assigned such that any two adjacent cells (horizontal or vertical) differ only in one variable. For example, cell values 01 and 11 differ in 1 bit only, so are the values 11 and 10. The decimal equivalent values are shown inside the cells. For example, for a 3-varaible K map cell having A=1 and BC=11, which is ABC as 111 is 7. Please note that the sequence of the number is not sequential in 3 variable and 4 variable K maps. This is because of the condition of change in only one variable between two adjacent cells. The decimal equivalent of minterm varaible values are marked inside the cells. For example, decimal equivalent (or minterm equivalent) number placed in the cell having ABCD values as 1111 in the 4 variable k-map is 15.

Please note that bottom row is adjacent to top row; and last column is adjacent to first column as they differ in only one variable respectively.



2-variables K map 3-variables K map 4-variables K-map

Fig. 3.9: K-map of 2, 3 and 4 variables

Step 2: MAP the Boolean function or truth table of Boolean function into K-map. Put a value 1 for every minterm for which the function output is 1.

Step 3: Simplify algebraic expression: Find adjacency of 1's in the K-map. You must find maximum adjacency in a sequence of …, 8, 4, 2. A cell having 1 can appear in more than one adjacencies. Find the maximal adjacencies till all 1's are part of at least one adjacency.

Step 4: Write the Boolean term for each adjacency and join these terms using OR operator.

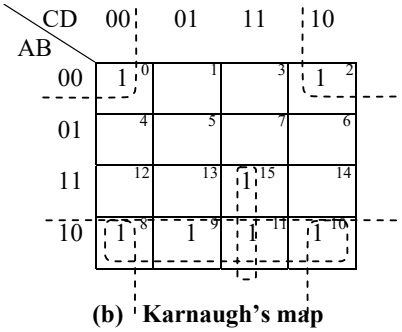
Resultant function is the simplified Boolean expression.

Example: Use K-map for finding the simplified Boolean function for the function $F(A,B,C,D) = \sum (0,2,8,9,10,11,15)$

Solution: The Truth table for the function is given below.

Decimal	A	B	C	D	F
0	0	0	0	0	1
1	0	0	0	1	0
2	0	0	1	0	1
3	0	0	1	1	0
4	0	1	0	0	0
5	0	1	0	1	0
6	0	1	1	0	0
7	0	1	1	1	0
8	1	0	0	0	1
9	1	0	0	1	1
10	1	0	1	0	1
11	1	0	1	1	1
12	1	1	0	0	0
13	1	1	0	1	0
14	1	1	1	0	0
15	1	1	1	1	1

(a) Truth table



(b) Karnaugh's map

(i) Adjacency 1: Four Corners (cells Numbered 0, 2, 8, 10)

(ii) Adjacency 2: The bottom Row (cells Numbered 8, 9, 11, 10)

(iii) Adjacency 3: Cell 11 and Cell 15

Figure 3.10: Truth table & K-Map of Function $F = \sum$ (0, 2, 8, 9, 10, 11, 15)

The three adjacencies of the K-map are shown in the Figure 3.10. You can write the Boolean expression for each adjacency.

1) The adjacency 1 of four corners (cells Numbered 0, 2, 8, 10) can be written algebrically as:

A′.B′.C′.D′ + A′.B′.C.D′ + A.B′.C′.D′ + A.B′.C.D′

= A′.B′.D′ .(C′+C) + A.B′.D′.(C′+C)

= A′.B′.D′ + A.B′.D′ (as C′+C=1)

= (A′+A).B′.D′

= B′.D′ (as A′+A=1)

Please note that an adjacency of 8/4/2 reduces the variables by 3/2/1 respectively.

A direct way of doing so is to identify the variables values of the adjacent cells which does not change, e.g. for this adjacency cell variable ABCD are 0000, 0010, 1000 and 1010. Thus the variable values of B and D does not change in all these 4 cells. In addition, since B and D have zero values among all these four cells, therefore, the expression is B′D′.

2) The four 1's in the bottom row (cells Numbered 8, 9, 11, 10)

The values of variable AB does not change and is 10 for the entire row, therefore, the expression for this adjacency would be A.B′

3) The two 1's in cell 11 and 15

A.B.C.D + A.B′.C.D

= A.C.D.(B+B′)

= A.C.D

Also please infer it directly from values 1111 1011

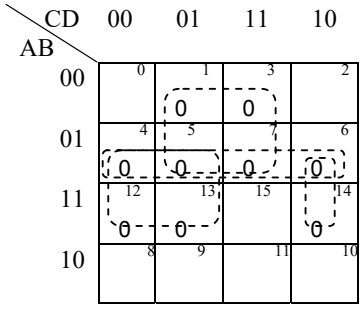
Thus, the simplified Boolean expression using K-Map is

F(A,B,C,D) = B′.D′+A.B′+A.C.D

The simplified expression using K-map is in SOP from. In order to obtain expression in POS form the K-map is created for 0 values and adjacency identified. The following example explains these steps.

Example: Use K-map to find the simplified Boolean function for the function F(A,B,C,D) = ∑ (0,2,8,9,10,11,15) in POS form.

Solution: The truth table is shown in previous example. It can be used to draw K-map for 0 values, which will be for the complement of the function, i.e. F′(A,B,C,D), as:



K-map for F′

Four Adjacencies:

(i) Cells (1,3,5,7) : A′ and D does not change, so the term is A′.D
(ii) 2nd Row (cells 4,5,7,6) : A′ B does not change A′.B
(iii) Cells 4,5,12,13 : B and C′ does not change B.C′
(iv) Cells 6 & 14 : B,C,D′ does not change B.C.D′

Since, you have found the adjacencies of 0's, therefore

F′(A,B,C,D) = A′.D+A′.B+B.C′+B.C.D′

or F(A,B,C,D) = (A′.D+A′.B+B.C′+B.C.D′)′
= (A′.D+A′.B)′ . (B.C′+B.C.D′)′
= (A′.D)′ . (A′.B)′ . (B.C′)′ . (B.C.D′)′
= ((A′)′+D′) . ((A′)′+B′) . (B′+(C′)′) . [(B.C)′+(D′)′]

F(A,B,C,D) = (A+D′) . (A+B′) . (B′+C) . (B′+C′+D), which is in POS form.

In certain digital design situations, some of the input combination has no significance, for example, while designing the circuit for BCD, the output for the input combinations 0000 (digit 0) to 1001 (decimal digit 9) are needed. For the rest of input 1010 to 1111, the output does not matter. Such K-maps are designed using DONOT CARE condition. The output for DONOT CARE input combinations is marked as X in the K-map. The cells marked X can be used for determining the maximal dependencies, but need not be covered as the case is for all 1's output. A detailed discussion on this is beyond the scope of this unit.

What will happen if you went to design circuits for more than 6 variables? With the increase in number of variables K-Maps become more cumbersome and are not suitable. Other methods have been designed to do so, which are beyond the scope of this course.

Check Your Progress 2

1) Draw the truth table for the following Boolean functions:

(i) F(A,B,C) = A′.B.C′+A.B.C+A.B.C′+B.C+A.C

(ii) F(A,B,C) = (A+B) . (A′+C′) . (C′+B′)

……

……

2 Simplify the following using algebraic simplification. And draw the logic diagram for the function so obtained

(i) F(A,B) = (A′.B′+B′)′

(ii) F(A,B) = (A.B+A′.B′)′

……

……

3) Simplify the following Boolean functions in SOP and POS forms using K-Maps. Draw the logic diagram for the resultant function.

F (A,B,C,D) = Σ (0,2,5,7,12,13,15)

……

……

3.6 DESIGN OF COMBINATIONAL CIRCUITS

The digital circuits, are constructed with NAND or NOR gates instead of AND–OR–NOT gates as they are U*niversal Gates*. Therefore, any digital circuit can be implemented using these gates. To prove this point in the following diagram AND, OR and NOT gates are implemented using NAND and NOR gates. This is shown in figure 3.11 to 3.13 below.

NOT Operation:

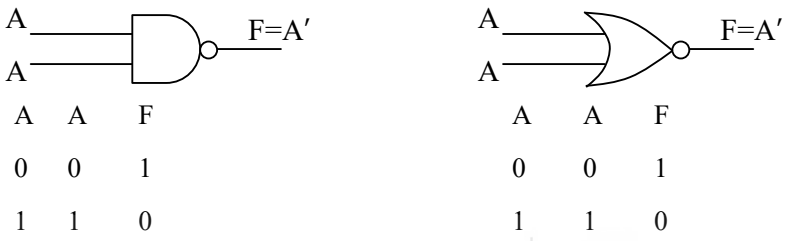


Figure 3.11: NOT Operation using NAND or NOR gates

AND Operation:

Performing AND using NAND gates can be achieved by first performing the NAND or the input followed by inverting the output as shown in Figure 3.12

$F = A.B$

$= ((A.B)')'$

$F = (A \text{ NAND } B)'$

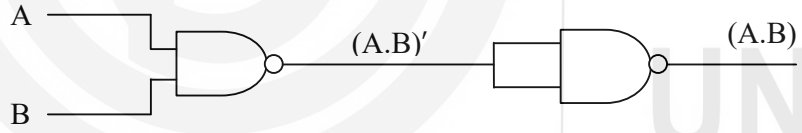


Figure 3.12: Logic circuit of AND Operation using NAND gates

AND operation can also be implemented using NOR gates. The following Boolean expression identifies that first NOR gates are used to invert the A and B input followed by taking NOR of A′ with B′

$F = A.B$

$F = ((A.B)')'$

$= (A'+B')'$

$= A' \text{ NOR } B'$

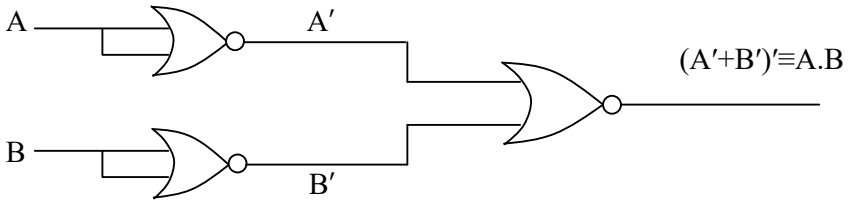


Figure 3.13: Logic circuit of AND Operation using NOR gates

OR Operation:

OR operation can be performed using NAND gate. Please refer to following Boolean expressions:

$F = A+B$

$= ((A+B)')'$

$F = (A'.B')' \Rightarrow A'$ NAND B'

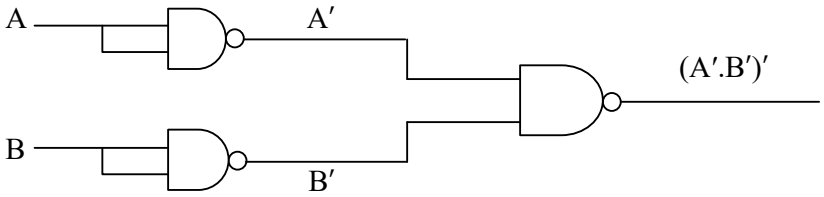


Figure 3.14: Logic circuit of OR Operation using NAND gates

$F=(A+B)$

$F= ((A+B)')'.$

$F = (A \text{ NOR } B)'$

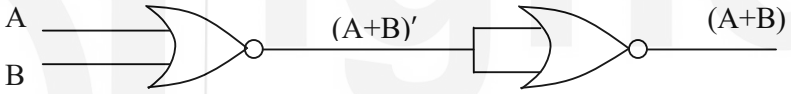
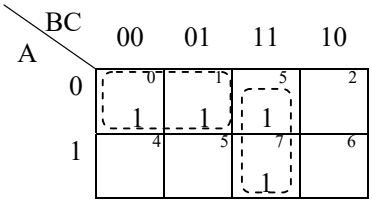


Figure 3.15: Logic circuit of OR Operation using NOR gates

A Boolean function can be implemented using the universal NAND or NOR gates by expressing the function in sum of product form as explained in the following example.

Example: Draw the circuit for $F(A,B,C) = \sum (0,1,3,7)$ using NAND gates.

Solution: Find the optimal Boolean function using K-map in SOP form:



$F(A,B,C) = A'B'+BC$

The AND – OR gate logic circuit for this is:

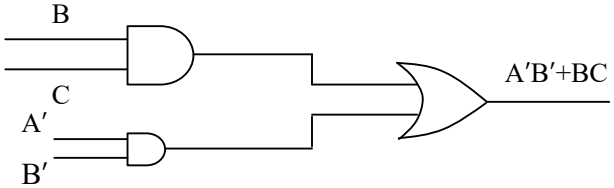


Figure 3.16 Logic Circuit Using AND-OR gate

For NAND gate logic circuit

F(A,B,C) = (A′B′+BC)
= ((A′.B′)′)′+((B.C)′)′
= ((A′. B′)′.(B.C)′)′ (Use of Demorgan's law)
= ((A′ NAND B′).(B NAND C))′
= (A′ NAND B′) NAND (B NAND C)

Thus, the circuit can be made simply by replacing two levels AND-OR circuit by NAND gates:

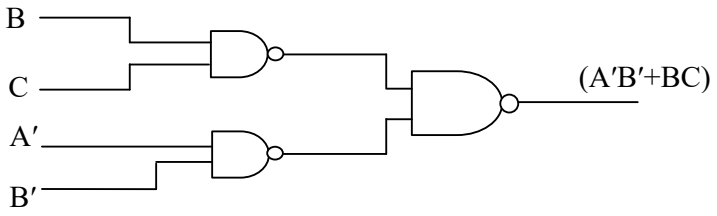


Figure 3.17: Logic Circuit by Replacing AND - OR circuit by NAND gates

A combinational circuit is required to produce a specific set of output for a given step of input. The design of a combinational circuit simply requires the following steps:

Step 1: Make the Truth table for the required design. You must draw the truth table for every output value.

Step 2: Use K-map or any other method to create optimal Boolean function that creates the desired output. One function is designed for each output.

Step 3: Draw the resultant circuit using universal gates.

The next section first design the design of half adder circuit as a combinational circuit. In addition, next section discusses some of the combinational circuits, the design of which is not detailed in this Unit..

3.7 EXAMPLES OF COMBINATIONAL CIRCUITS

In this section, first the combinational circuits design is demonstrated using basic combinational circuits like half and full adders. This is followed by discussion on combinational circuits like decoders, multiplexers etc.

3.7.1 Adders

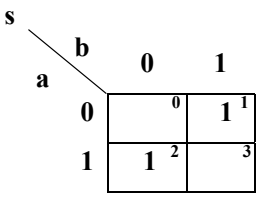
Addition is one of the most common arithmetic operations. In this section two different kinds of addition circuits are designed. The first of the two circuit adds two binary digits and is called a half adder, while the second adds three bits-two addend and one carry bit, and is called a full adder.

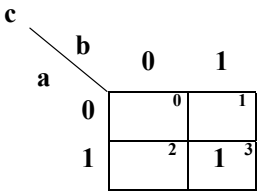
Half Adder:

Let us assume that a half adder circuit is adding two bits a and b to produce one sum bit (s) and one carry bit (c). The following truth table shows this operation. Please note one adding a =1 and b=1 you get a carry as 1 and sum bit as 0 as shown in truth table. The K-maps for the addition is shown in figure 3.18.

a	b	c	s
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

(a) Truth table





(b) K- map for sum bit **(c) K-map for carry bit**

Figure 3.18: Truth table and K-maps for half adder

The Boolean expression for them from the k-maps are:

$s = a'b+ab'$ and

$c = a.b$

The logic circuit for the half adder is based on the Boolean expressions are given are shown in Figure 3.19.

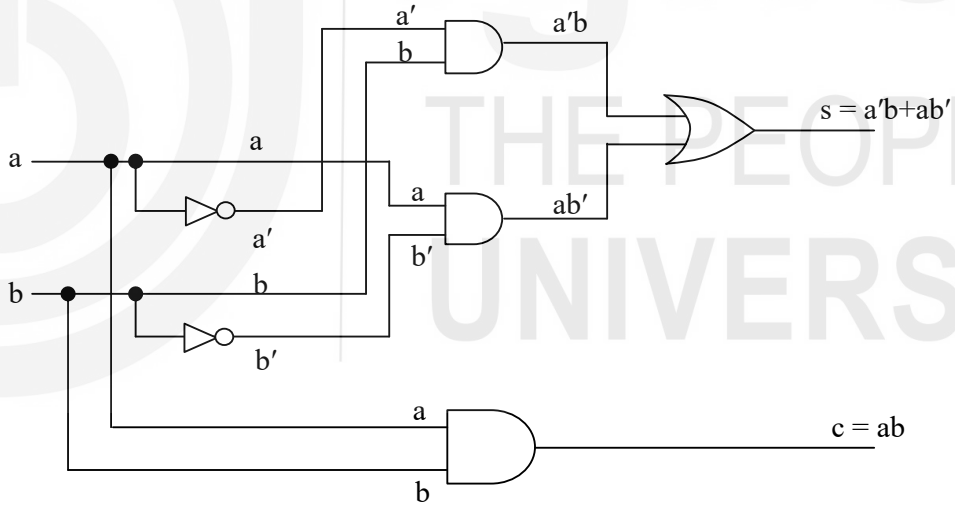


Figure 3.19: The half adder circuit-input addend bits a, b; output sum bit (s) and carry bit (c)

Full Adder:

Full adder is a circuit that adder 3 bits, viz. 2 addend bits and one carry bit. The truth table for full adder is shown in Figure 3.20. Please note that in figure 3.20, c_{in} is carry in bit and c_{out} is carry out bit.

Input					Output	
Decimal equivalent	a	b	c_{in}		Carry out (c_{out})	Sum (s)
0	0	0	0		0	0
1	0	0	1		0	1
2	0	1	0		0	1
3	0	1	1		1	0
4	1	0	0		0	1
5	1	0	1		1	0
6	1	1	0		1	0
7	1	1	1		1	1

Figure 3.20: The Truth Table for Full Adder

Please note that in the truth table, when a = 1, b = 1 and c_{in} =1 , than the output is 11 , which means sum bit (s) is 1 and carry out bit is also 1. The K-map for these are also shown in Figure 3.21

K-map for sum bit

No adjacency

K – Maps for c_{out}

Three adjacencies

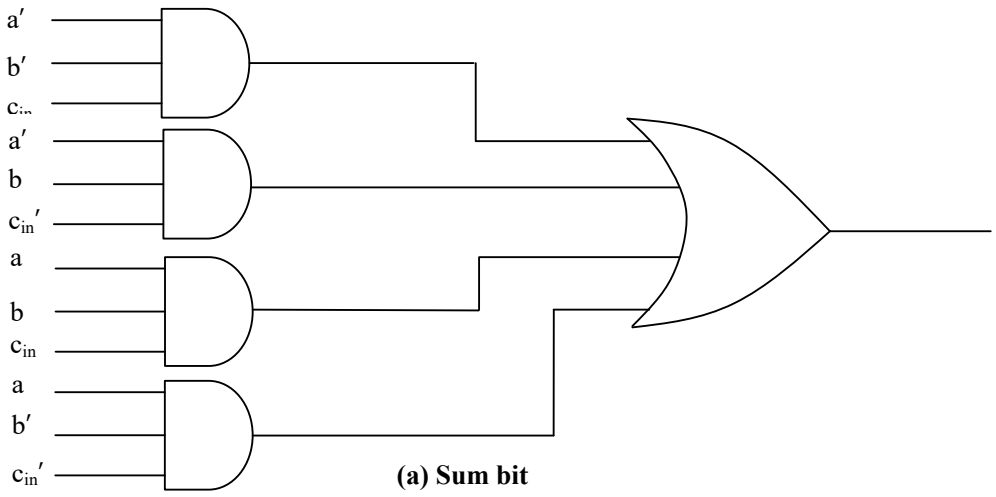
Figure 3.21: The K-maps for Full Adder

The Boolean functions based on the K-maps given in Figure 3.21 are given below:

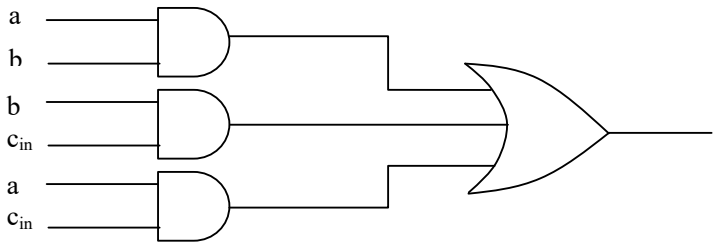
$$s = a'.b'.c_{in} + a'.b.c_{in}' + a\ b\ c_{in} + a\ b'\ c_{in}'$$

$$c_{out} = a\ b + b\ c_{in} + a\ c_{in}$$

Figure 3.22 shows the full adder circuit. Please note that for simplicity the circuits for inverting the input values are not drawn.



(a) Sum bit



(b) Carry Out bit

Figure 3.22: Full Adder

Full adder and half adder only perform bit addition of two operands without or with carry bit respectively. However, binary numbers have several bits e.g. integers can be 4 byte long. How will they be added? This is performed by creating a sequence of full adders, where carry out bit of the lower bit addition is fed as carry in bit of next higher bit addition, as shown in figure 3.23.

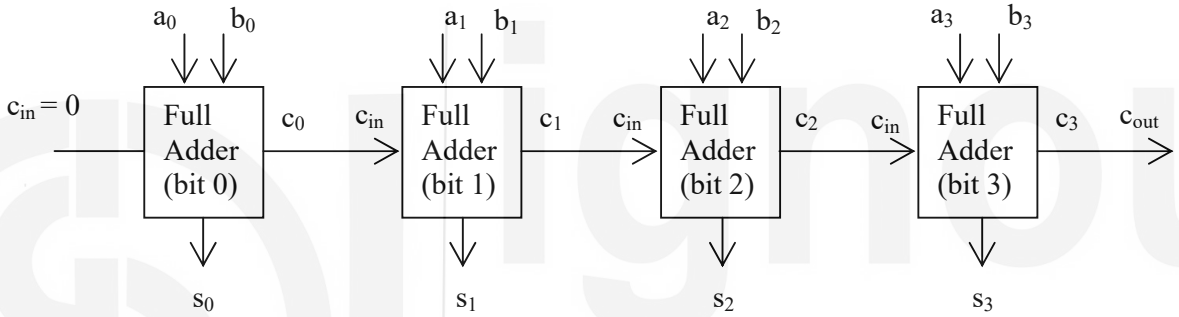


Figure 3.23: Addition of a 4 bit number using 4 full adders

Please note that in the Figure 3.23, the carry out of the previous bit addition is input as carry in bit of the next bit addition. For example, the value of c_0 bit, which is carry out of bit 0 addition, will be available if the full adder of bit 0 has been completed. Drawback of this circuit is the time taken to add the number is large as each full adder will take some signal propagation time, and the addition of the next bit cannot be performed till the c_{in} bit is available. A faster binary adder circuit would predict the carry bit. These are called look-ahead carry adders. It may be noted that carry out of the 0^{th} bit can be computed using the Boolean function $c_0 = a_0.b_0$. The value c_0 is input as c_{in} of full adder of bit 1. The truth table of Figure 3.24 has been drawn for full adder of bit 1. This truth table can be used to design circuit that computes the value of c_1 prior to actual addition.

	Input			**Output**	
Decimal equivalent	**c_0**	**a_1**	**b_1**	**c_1**	**s_1**
0	0	0	0	0	0
1	0	0	1	0	1
2	0	1	0	0	1
3	0	1	1	1	0
4	1	0	0	0	1
5	1	0	1	1	0
6	1	1	0	1	0
7	1	1	1	1	1

Figure 3.24: Truth table for Full adder (bit 1)

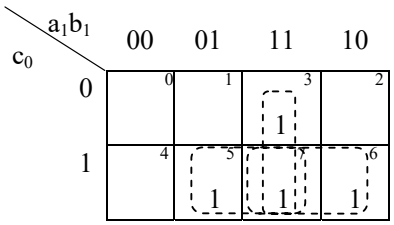


Figure 3.24: K-map for c_1 output of Full adder (bit 1)

There are three adjacencies in the K-map of Figure 3.24. The resultant Boolean function for c_1 would be:

$c_1 = a_1.b_1+c_0.a_1+c_0.b_1$

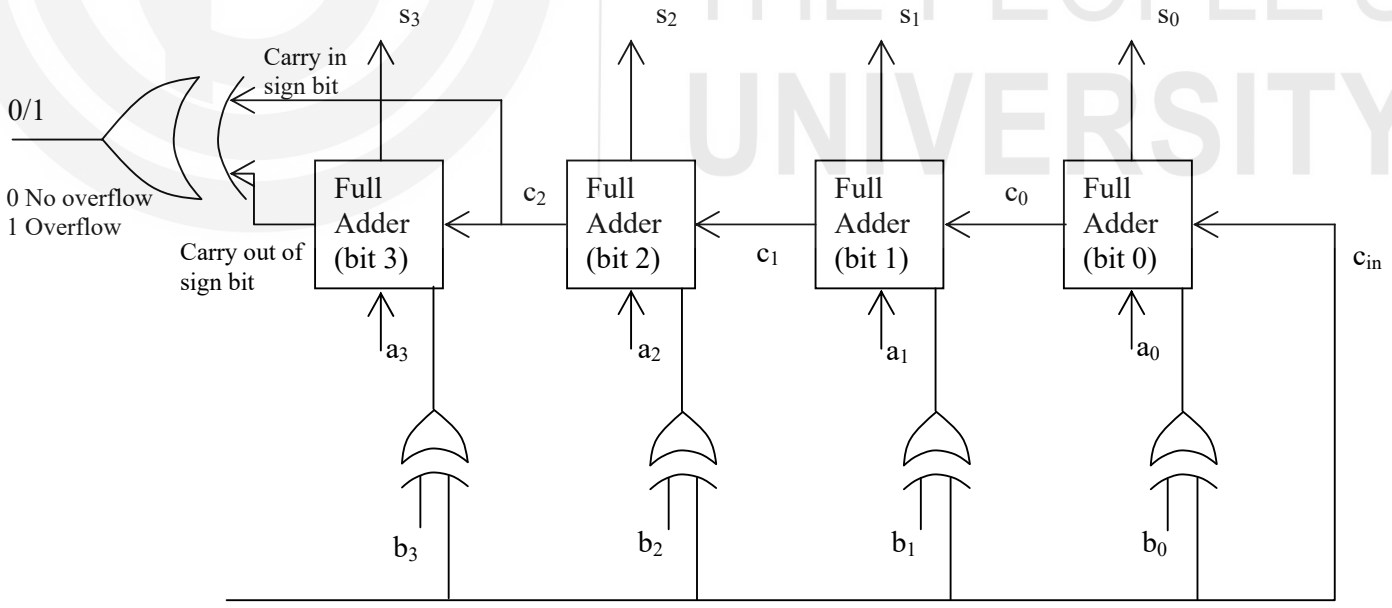
$c_1 = a_1b_1+c_0.(a_1+b_1)$ (Taking c_0 common)

$c_1 = a_1.b_1+a_0.b_0.(a_1+b_1)$ (Replacing c_0 by its equivalent)

Logic circuits can be designed for prediction of carry bits c_0, c_1, etc. and resultant circuits can be implemented along with full adder circuits. You can observe that Boolean expression for higher order carry bits like c_2, c_3 etc. will become more complex, which results in complex logic circuits. Thus, look ahead carry bit adders may be implemented for addition of binary numbers of size 4-8 bits.

Adder-Subtractor Circuit

Adder subtractor circuit is an interesting design, in which a same circuit is used for addition as well as subtraction. This example shows how with some additional logic, you may be able to perform additional operations. ALU is a fine example of extension of such logic. Figure 3.25 shows the circuit of 4 bit adder-subtractor circuit by using full adders.



Mode bit: Addition
Operation mode bit = 0
Subtraction operation mode bit = 1

Figure 3.25: Adder Subtractor circuit using full adders for addition and subtraction of 4-bit 2′s complement numbers

You may please note that the mode bit controls the b input. The following Figure shows the details of operation.

Mode bit = 0		Mode bit = 1	
Input b	Inputs the bits of b input XOR with mode bit value (0)	Input b	Input value after taking XOR of input b and mode bit
0 1	0 XOR 0 = 0 1 XOR 0 = 1	0 1	0 XOR 1 = 1 1 XOR 1 = 0
Thus, when mode bit is 0, the input to full adder is the value of input b.		Thus, when mode bit is 1, the input to adder is the value of 1′s complement of b	
c_{in} = 0 as mode bit = 0 so the circuit adds input a and input b.		c_{in} = 1, so the addition is r = a+b′+1 or r= a+2′s complement of b r = a – b ; subtraction of a and b	

Figure 3.26: Use of Mode bit to control b input in Adder subtractor circuit

Please also note that in 2's complement notation the last bit is treated as sign bit. The overflow condition is checked by finding if the carry into the sign bit and carry out of sign bit are same or not same. In case carry in to the sign bit is not the same as carry out of the sign bit, then overflow is set to 1 (by XOR gate) else overflow is set to 0 (No overflow).

3.7.2 Decoders

Decoder, as the name suggests, decodes the input to one of the output line. Figure 3.27 shows the truth table and logic circuit of a 2 × 4 decoder.

Truth table

Input		Output			
a	b	c	d	e	f
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

The Boolean functions for various output values are

c = a′ b′
d = a′ b
e = a b′
f = a b

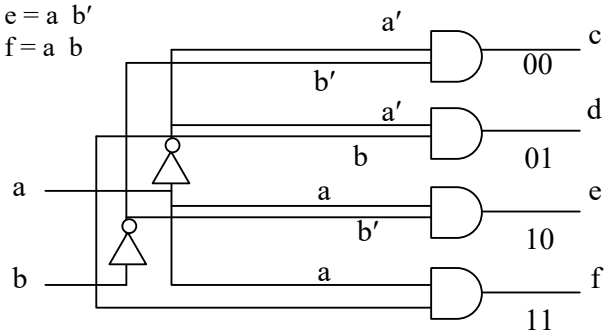


Figure 3.27: 2 × 4 decoder

A decoder line would be selected if it has an output 1. In general, decoder is a very useful circuit for selecting lines and forms the basis of Random Access Memory.

Please note that numbers of output for 2 bit decoder are $2^2 = 4$; hence the name 2×4 decoder. Similarly, the number of output for a 3 bit input would be $2^3 = 8$ and it is called 3×8 decoder.

3.7.3 Multiplexer

A multiplayer allows sharing of a line by multiple inputs. It may be very useful for serialization of data bits over a single output line. The design of a multiplexer is however, different from other combinational circuits as it is the selection lines which control the selection of input line. The following is the truth table of a 4×1 multiplexer. A 4×1 multiplexer selects one of the 4 input lines to be transmitted over a single output. Out of these 4 lines, which will be selected, will be determined by 2 selection lines. How many selection lines will be required for 8×1 multiplexer? Since $2^3 = 8$, so 3 selection lines would be required for 8×1 multiplexer.

Input			
Selection Lines		Input	Output
s_1	s_0		
0	0	I_0	I_0
0	1	I_1	I_1
1	0	I_2	I_2
1	1	I_3	I_3

Please note the values of output can be I_i, where the value of subscript *i* can vary from 0 to 3.

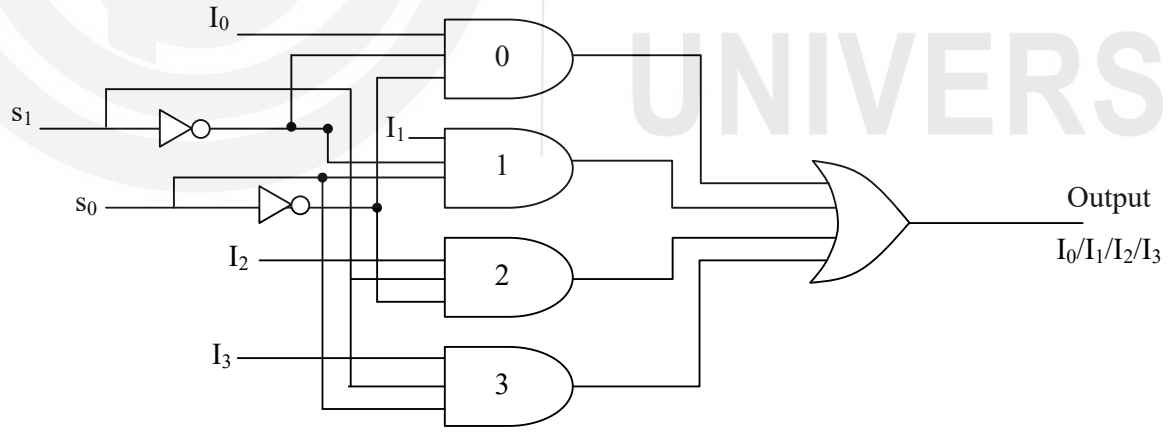


Figure 3.28: 4 × 1 Multiplexer

Please note this is a very important circuit for sharing of an output line across many sources of input.

3.7.4 Encoders

An encoder, in general, is the inverse of a decoder. Based on its input it produces a specific output. For example, the truth table of a 4×2 encoder is shown in Figure 3.29

Input				Output	
I_0	I_1	I_2	I_3	O_1	O_0
1	0	0	0	0	0
0	1	0	0	0	1
0	0	1	0	1	0
0	0	0	1	1	1

Figure 3.29: Truth Table of 4 × 2 encoder

The simple expression for various output can be

$O_1 = I_2 + I_3$

and $O_0 = I_0 + I_1$ Thus, the simple circuit for this encoder is

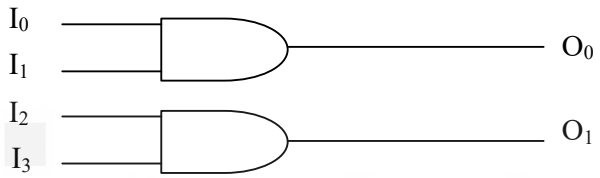


Figure 3.30 Logic Diagram of a simple encoder

3.7.5 Programmable Logic Array

The basic combinational circuits can be implemented using AND-OR-NOT gates. PLAs are circuits which are prefabricated for all possible combination of AND, OR, NOT gates. They can be used to fabricate any kind of logic circuit. They are primarily designed for SOP form of logic circuit.

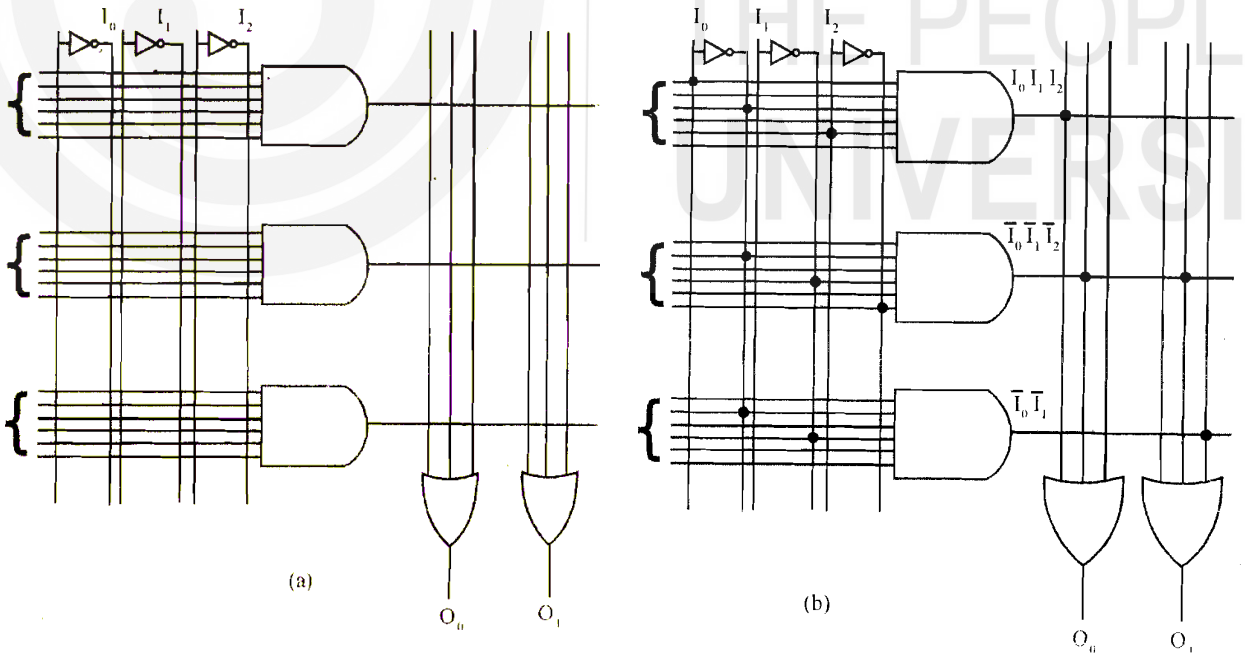


Figure 3.31: Programmable Logic Array

The figure 3.31(a) shows a PLA of 3 inputs and 2 outputs. Figure 3.31(b) shows an implementation of logic function given below using the PLA of figure 3.31 (a).:

$O_0 = I_0 . I_1 . I_2 + I_0' . I_1' . I_2'$

$O_1 = I_0' . I_1' . I_2' + I_0' . I_1'$

3.7.6 Read-only-Memory (ROM)

ROM is an example of use of Programmable Logic Devices (PLD). It stores the binary information using a combinational circuit. The RAM follows the simple sequence. Figure 3.32 shows a ROM of size 4 × 2, which has 4 lines of 2 bits each. Please note the use of 2 × 4 decoder. Also note that wherever the line will be connected an output will appear. These connections are embedded within the hardware. Thus the information of ROM is not lost even after the power failure..

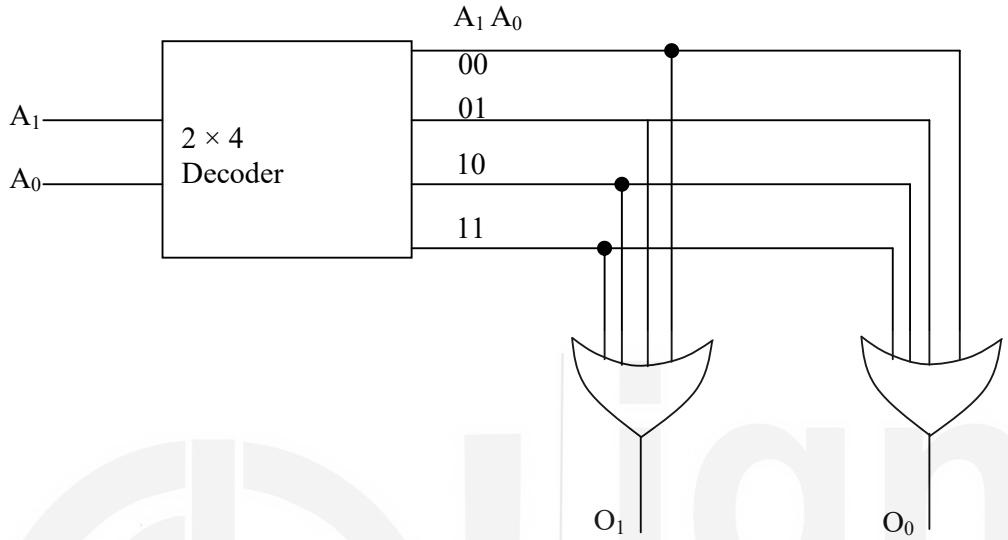


Figure 3.32: ROM Design

Please note that the ROM shown in Figure 3.32 have the following content:

Address Line selection A_1 A_0	ROM content O_1 O_0
0 0	1 0
0 1	0 0
1 0	1 0
1 1	1 0

Please note the number of words in the ROM is 4 which is $2^2 = 4$ as it has 2 address lines. A ROM with 3 address lines will have 8 words. ($2^3 = 8$).

The size of word can be chosen by the designer and, in general, it can be 8, 16, 32 or 64 bit as per the machine firmware designer. The address lines of ROM select any one line as per the address.

Check Your Progress 3

1) Design a combinational circuit, which takes four bit input and produces an output 1 if the input contains three consecutive 1 bits.

……..

……..

2) Draw the logic diagram of the function as above using

(i) AND-OR-NOT gates &

(ii) NAND gate

……..

……

……

3) Consider the circuit of Figure 3.26, what would be the output of the circuit if:

(i) Input a is 1010 and input b is 1100 and mode bit is 0

(ii) Input a is 0010 and input b is 0100 and mode bit is 1

……..

……

……

4) Why is PLA needed?

……

……

……

5) Design a full adder using two half adder circuits.

……

……

……

3.8 SUMMARY

This Unit introduces you to some of the basic concepts relating to computer logic. The Unit first introduces the concept of logic gates, the most fundamental unit of logic circuits. The Unit then explains the process of making simple logic circuits, including combinational circuit. The mathematical foundation of the logic circuit design, the Boolean algebra is also introduced. The Karnaugh's map was used to design simpler circuit. The Unit also explains the desing of different kinds of adders circuit, highlghting , how complex circuit can be desingned using K-map. Finally, the Unit explains some of the most fundamental combinational circuits like decoder, multiplexer, encodes, PLA's etc. It may be noted that the objective of this Unit is not to make you a computer hardware designer, but to introduce you to some of the basic concepts of circuit design.
You can refer to latest trends of design and development including VHDL (a hardware design language) in the further readings.

3. 9 SOLUTIONS/ANSWERS

Check Your Progress 1

1) A logic gate is most fundamental circuit that can be fabricated on a silicon chip. A logic gate operates at signal level to produce simple logic like AND, OR, NOT etc. A Universal gates can be used to implement each and every kind of logic circuit. Two examples of universal gates are NAND and NOR.

2)

I_1	I_2	I_3	$I_1+I_2.I_3$	$(I_1+I_2).(I_1+I_3)$
0	0	0	0	0
0	0	1	0	0
0	1	0	0	0
0	1	1	1	1
1	0	0	1	1
1	0	1	1	1
1	1	0	1	1
1	1	1	1	1

3) F = ((A′+B) ′+(A.B′)′)′

= ((A′+B) ′)′ . ((A.B′)′)′ (by Demorgan's Law)

= (A′+B) . (A.B′) (as (a')' = a)

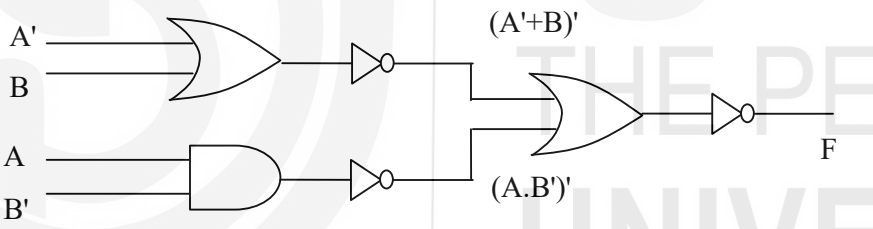
= ((A′+B).A).((A′+B).B')

= ((A'.A)+(B.A)).((A'.B')+B.B'))

= ((0+A.B).(A'.B' +0)

= (A.B.A'.B')=0

4) Draw the logic diagram of the function before simplification.



5) Just the F can be connected to logical 0 input as F=0

Check Your Progress 2

1)

A	B	C	F=A′.B.C′+A.B.C+A.B.C′+B.C+A.C	F= (A+B) . (A′+C′) . (C′+B′)
0	0	0	0	0
0	0	1	0	0
0	1	0	1	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	1
1	1	1	1	0

2 (i) F(A,B) = (A′.B′+B′)′

= (A'.B')' . (B')' (DeMorgan's Law)

= ((A')'+(B')').B (DeMorgan's Law)

= (A+B).B

= A.B + B.1=B.(A+1)= B

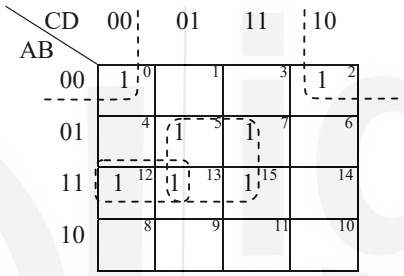
B ——————— F

(ii) F(A,B) = (A.B+A′.B′)′
= (A.B)' . (A'.B')'
= (A'+B').(A+B)
= (A'+B').A + (A'+B').B
= A'.A+B'.A+A'.B+B'.B
= A.B' + A'.B

The logic diagram is same as sum bit of half adder. It is also the A XOR B.

3) Simplify the following boolean functions in SOP and POS forms using K-Maps. Draw the logic diagram for the resultant function.

F (A,B,C,D) = Σ (0,2,5,7,12,13,15)



Three adjacencies

i) Cells 0 and 2: The variables does not change A' B' D'

ii) Cells 12 and 13; The variable does not change A B C'

iii) Cells 5,7,13,15; two variables does not change B D

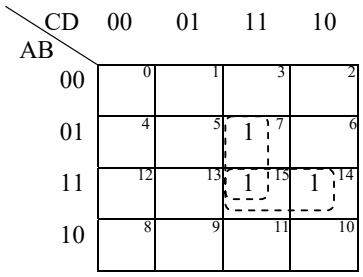
The expression is F=A'.B'.D' + A.B.C' + B.D

Check Your Progress 3

1) The Truth table:

Decimal	A	B	C	D	F
0	0	0	0	0	0
1	0	0	0	1	0
2	0	0	1	0	0
3	0	0	1	1	0
4	0	1	0	0	0
5	0	1	0	1	0
6	0	1	1	0	0
7	0	1	1	1	1
8	1	0	0	0	0
9	1	0	0	1	0
10	1	0	1	0	0
11	1	0	1	1	0
12	1	1	0	0	0
13	1	1	0	1	0
14	1	1	1	0	1
15	1	1	1	1	1

The K-map for the Truth table:



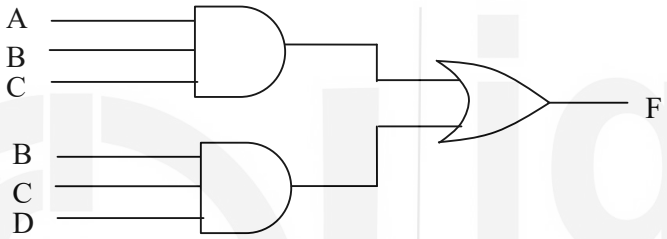
Only two adjacencies: Cells 7 and 15 B C D and

Cells 15 and14 A B C

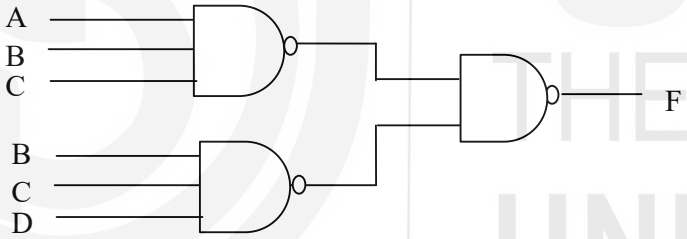
Therefore, the function is: F = A.B.C + B.C.D

2) Logic diagram of the function as above

(i)



(ii)



3) (i) Input a is 1010 and input b is 1100 and mode bit is 0

Bit wise addition will be as follows:

a	1	0	1	0
b	1	1	0	0
c	0	0	0	0
sum bit	0	1	1	0
carry out of sign bit	1			

carry in to sign bit NOT equal to carry out of sign bit,
OVERFLOW

(ii) Input a is 0010 and input b is 0100 and mode bit is 1
Bit wise addition will be as follows:

a	1	0	1	0
b (1's complement)	0	0	1	1
c	0	1	1	1
sum bit	1	1	1	0
carry out of sign bit	0			

carry in to sign bit IS EQUAL to carry out of sign bit,
NO OVERFLOW

4) PLA's can be fabricated as a chip that can be customised as per the need of the SOP logic.

5) A half adder adds two addend bits, whereas a full adder adds the two addends and previous carry bit, therefore, one half adder will be needed to add two addend bits, and second half adder will be needed to sum the sum of first half adder and previous carry bit. The output carry will be set, if any of the two half adder produce the carry out. The following block diagram shows this construction: