Course Code	CSC 324
Course Title	DIGITAL DESIGN & COMPUTER ARCHITECTURE
Course Lecturer	Dr. ONASHOGA, S. A. (Mrs.)

DEPT. OF COMPUTER SCIENCE

UNIVERSITY OF AGRICULTURE,

ABEOKUTA, OGUN STATE

NIGERIA.

MODULE 1 - Basic Logic Operations and Logic Expressions

UNIT 1: Basic Logic Operations

Contents **Pages**

1.0 Introduction ………………………………………………………………… 2

2.0 Objectives ………………………………………………………………… 2

3.0 Binary Numbers ………………………………..……………………………… 2

3.1 Binary Addition……………………………………………………………….. 6

3.2 Binary Switch……………………..…………………………………………… 6

3.3 Truth Tables…………………………………………………………………… 8

3.4 Boolean Algebra ……………………………………………………………… 9

3.5 Duality Principle……………………..……………………………………….. 12

4.0 Conclusion ………………………………………………………………..12

5.0 Summary ……………………………………………………………………… 13

6.0 Tutor Marked Assignment………………………..……………………………. 13

7.0 Further Reading and Other Resources ………………………………………… 13

1.0 Introduction

Our world is an analog world. Measurements that we make of the physical objects around us are never in discrete units, but rather in a continuous range. We talk about physical constants such as 2.718281828… or 3.141592…. To build analog devices that can process these values accurately is next to impossible. Even building a simple analog radio requires very accurate adjustments of frequencies, voltages, and currents at each part of the circuit. If we were to use voltages to represent the constant 3.14, we would have to build a component that will give us exactly 3.14 volts every time. This is again impossible; due to the imperfect manufacturing process, each component produced is slightly different from the others. Even if the manufacturing process can be made as perfect as we can get, we still would not be able to get 3.14 volts from this component every time we use it. The reason being that the physical elements used in producing the component behave differently in different environments, such as temperature, pressure, and gravitational force, just to name a few. Therefore, even if the manufacturing process is perfect, using this component in different environments will not give us exactly 3.14 volts every time.

To make things simpler, we work with a digital abstraction of our analog world. Instead of working with an infinite continuous range of values, we use just two values: 1 and 0, on and off, high and low, true and false, black and white, or however you want to call it. It is certainly much easier to control and work with two values rather than an infinite range. We call these two values a binary value for the reason that there are only two of them. A single 0 or a single 1 is then a **binary digit** or **bit**. This sounds great, but we have to remember that the underlining building block for our digital circuits is still based on an analog world. This unit explores the world of binary system with the different operations attached to it.

2.0 Objectives

Upon completion of this unit, you will be able to:

- Provide theoretical foundation for building digital logic circuits.
- Understand the binary number system and conversion
- Understand the use of Truth Tables
- Understand the Boolean algebras
- Understand the duality principle

3.0 Binary Numbers

Since digital circuits deal with binary values, we will begin with a quick introduction to binary numbers. A bit, having either the value of 0 or 1 can represent only two things or two pieces of information. It is, therefore, necessary to group many bits together to represent more pieces of information. A string of *n* bits can represent 2*n* different pieces of information. For example, a string of two bits results in the four combinations 00, 01, 10, and 11.

By using different encoding techniques, a group of bits can be used to represent different information, such as a number, a letter of the alphabet, a character symbol, or a command for the microprocessor to execute.

The use of decimal numbers is quite familiar to us. However, since the binary digit is used to represent information within the computer, we also need to be familiar with **binary numbers**. Note that the use of binary numbers is just a form of representation for a string of bits. We can

just as well use octal, decimal, Binary Coded Decimal (BCD) or hexadecimal numbers to represent the string of bits. In fact, you will find that hexadecimal numbers are often used as a shorthand notation for binary numbers.
The decimal number system is a positional system. In other words, the value of the digit is dependent on the position of the digit within the number. For example, in the decimal number 48, the decimal digit 4 has a greater value than the decimal digit 8 because it is in the tenth position, whereas the digit 8 is in the unit position. The value of the number is calculated as

$$4\times10^1 + 8\times10^0.$$

Like the decimal number system, the binary number system is also a positional system. The only difference between the two is that the binary system is a base-2 system, and so it uses only two digits, 0 and 1, instead of ten.
The binary numbers from 0 to 15 (decimal) are shown in Table 1.1(a). . The range from 0 to 15 has 16 different combinations, we need a 4-bit binary number, i.e., a string of four bits, to represent this range.
When we count in decimal, we count from 0 to 9. After 9, we go back to 0, and have a carry of a 1 to the next digit. When we count in binary, we do the same thing except that we only count from 0 to 1. After 1, we go back to 0, and have a carry of a 1 to the next bit.

The decimal value of a binary number can be found just like for a decimal number except that we raise the base number 2 to a power rather than the base number 10 to a power. For example, the value for the decimal number 658 is

$$658_{10} = 6\times10^2 + 5\times10^1 + 8\times10^0 = 600 + 50 + 8 = 658_{10}$$

Similarly, the decimal value for the binary number 1011011_2 is

$$1011011_2 = 1\times2^6 + 0\times2^5 + 1\times2^4 + 1\times2^3 + 0\times2^2 + 1\times2^1 + 1\times2^0$$
$$= 64 + 16 + 8 + 2 + 1 = 91_{10}$$

To get the decimal value, the least significant bit (in this case, the rightmost 1) is multiplied with 2^0. The next bit to the left is multiplied with 2^1, and so on. Finally, they are all added together to give the value 91_{10}.
Notice the subscript 10 in the decimal number 658_{10}, and the 2 in the binary number 1011011_2. This subscript is used to denote the base of the number whenever there might be confusion as to what base the number is in.

Table 1.1(a) - Numbers from 0 to 15 in binary, octal, and hexadecimal.

Decimal	Binary	Octal	Hexadecimal
0	0000	0	0
1	0001	1	1
2	0010	2	2
3	0011	3	3
4	0100	4	4
5	0101	5	5
6	0110	6	6
7	0111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F

Converting a decimal number to its binary equivalent can be done by successively dividing the decimal number by 2 and keeping track of the remainder at each step. Combining the remainders together (starting with the last one) forms the equivalent binary number. For example, using the decimal number 91, we divide it by 2 to get 45 with a remainder of 1. Then we divide 45 by 2 to get 22 with a remainder of 1. We continue in this fashion until the end as shown below.

2 | 91 1 least significant bit
2 | 45 1
2 | 22 0
2 | 11 1 = 1011011
2 | 5 1
2 | 2 0
1 most significant bit

Concatenating the remainders together starting with the last one results in the binary number 1011011_2.
Binary numbers usually consist of a long string of bits. A shorthand notation for writing out this lengthy string of bits is to use either the octal or hexadecimal numbers. Since octal is base-8 and hexadecimal is base-16, both of which are a power of 2, a binary number can be easily converted to an octal or hexadecimal number, or vice versa.

Octal numbers only use the digits from 0 to 7 for the eight different combinations. When counting in octal, the number after 7 is 10 as shown in Table 1.1(a). . To convert a binary number to octal, we simply group the bits into groups of threes starting from the right. The

reason for this is because $8 = 2^3$. For each group of three bits, we write the equivalent octal digit for it. For example, the conversion of the binary number $1\ 110\ 011_2$ to the octal number 163_8 is shown below:

$$\frac{001}{1}\ \frac{110}{6}\ \frac{011}{3}$$

Since the original binary number has seven bits, we need to extend it with two leading zeros to get three bits for the leftmost group. Note that when we are dealing with negative numbers, we may require extending the number with leading ones instead of zeros.

Converting an octal number to its binary equivalent is just as easy. For each octal number, we write down the equivalent three bits. These groups of three bits are concatenated together to form the final binary number. For example, the conversion of the octal number 5724_8 to the binary number $101\ 111\ 010\ 100_2$ is shown below.

5 7 2 4
101 111 010 100

The decimal value of an octal number can be found just like for a binary or decimal number except that we raise the base number 8 to a power instead. For example, the octal number 5724_8 has the value

$$5724_8 = 5\times8^3 + 7\times8^2 + 2\times8^1 + 4\times8^0 = 2560 + 448 + 16 + 4 = 3028_{10}$$

Hexadecimal numbers are treated basically the same way as octal numbers except with the appropriate changes to the base. Hexadecimal (or hex for short) numbers use base-16, and thus require 16 different digit symbols as shown in Table 1.1(a). Converting binary numbers to hexadecimal numbers involve grouping the bits into groups of fours since $16 = 2^4$. For example, the conversion of the binary number $110\ 1101\ 1011_2$ to the hexadecimal number $6DB_{16}$ is shown below. Again, we need to extend it with a leading zero to get four bits for the leftmost group.

$$\frac{0110}{6}\ \frac{1101}{D}\ \frac{1011}{B}$$

To convert a hex number to a binary number, we write down the equivalent four bits for each hex digit, and then concatenate them together to form the final binary number. For example, the conversion of the hexadecimal number $5C4A_{16}$ to the binary number $0101\ 1100\ 0100\ 1010_2$ is shown below.

5 C 4 A
0101 1100 0100 1010

The following example shows how the decimal value of the hexadecimal number $C4A_{16}$ is evaluated.

$$C4A_{16} = C\times16^2 + 4\times16^1 + A\times16^0 = 12\times16^2 + 4\times16^1 + 10\times16^0 = 3072 + 64 + 10 = 3146_{10}$$

Binary Coded Decimal (BCD) is a means of encoding decimal numbers. If each digit of a decimal number is represented by its binary equivalent, this produces a code called BCD. Since a decimal digit can be as large as 9, 4 bits are also required to code each digit. To illustrate the BCD code, take a decimal number such as 874. Each digit is changed to its binary equivalent as follows

8	7	4	(decimal)
1000	0111	0100	(BCD)

The BCD code, then represents each digit of the decimal number by a 4-bit binary number. Clearly, only the 4-bit binary numbers from 0000 through 1001 are used. The BCD does not use the numbers 1010, 1011, 1100, 1101, 1110 and 1111. In other words, only 10 of the 16 possible 4-bit binary code groups are used. If any of these "forbidden" 4-bit numbers ever occur in a machine using the BCD code, it is usually an indication that an error has occurred.
Note: It is important to realize that BCD is not another number system like binary, octal, decimal and hexadecimal. It is in fact, the decimal system with each digit encoded in its binary equivalent.

3.1 Binary Addition

The addition of two binary numbers is performed in exactly the same manner as the addition of decimal numbers. In fact, binary addition is simpler, since there are fewer cases to learn. The general steps in performing decimal addition are also followed in binary addition. However, only four cases can occur in adding the two binary digits (bits) in any position. They are:

0 + 0 = 0
1 + 0 = 1
1 + 1 = 10 = 0 + carry of 1 into next position
1 + 1 + = 11 = 1 + carry of 1 into next position

The last case occurs when the two bits in a certain position are 1 and there is a carry from the previous position. Here are several examples of the addition of two binary numbers:

```
  011 (3)        1001 (9)        11.011 (3.375)
 +110 (6)       +1111 (15)      +10.110  (2.750)
 --------       ----------      ----------------
 1001 (9)       11000 (24)      110.001 (6.125)
```

It is not necessary to consider the addition of more than two binary numbers at a time, because in all digital systems the circuitry that actually performs the addition can handle only two numbers at a time. When more than two numbers are added, the first two are added together and then their sum is added to the third number; and so on. This is not a serious drawback, since modern digital computers can typically perform an addition operation in a few microseconds.

Try the following:

Add the following pairs of binary numbers: (a) 10110 + 00111 (b) 011.101 + 010.010

3.2 Binary Switch

Besides the fact that we are working only with binary values, digital circuits are easy to understand because they are based on one simple idea of turning a switch on or off to obtain either one of the two binary values. Since the switch can be in either one of two states (on or off), we call it a **binary switch**, or just a **switch** for short. The switch has three connections: an input, an output, and a control for turning the switch on or off as shown in Figure 1.1(a). When the switch is opened as in (a), it is turned off and nothing gets through from the input to the output. When the switch is closed as in (b), it is turned on, and whatever is presented at the input is allowed to pass through to the output.

Figure 1.1(a): Binary switch: (a) opened or off; (b) closed or on.
Uses of the binary switch idea can be found in many real world devices. For example, the switch can be an electrical switch with the input connected to a power source and the output connected to a siren *S* as shown in Figure 1.1(b)

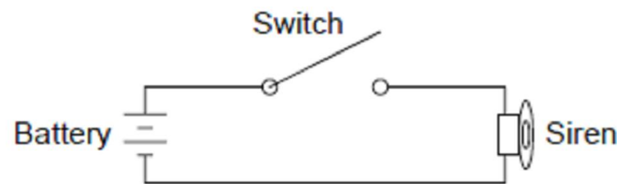


Figure 1.1(b): A siren controlled by a switch.
When the switch is closed, the siren turns on. The usual convention is to use a 1 to mean "on" and a 0 to mean "off." Therefore, when the switch is closed, the output is a 1 and the siren will turn on. We can also use a variable, x, to denote the state of the switch. We can let $x = 1$ to mean the switch is closed and $x = 0$ to mean the switch is opened. Using this convention, we can describe the state of the siren S in terms of the variable x using a simple logic expression. Since

$S = 1$ if $x = 1$ and $S = 0$ if $x = 0$,

we can write $S = x$

This logic expression describes the output S in terms of the input variable x.

Two binary switches can be connected together either in series or in parallel as shown in Figure 1.1(c)
.

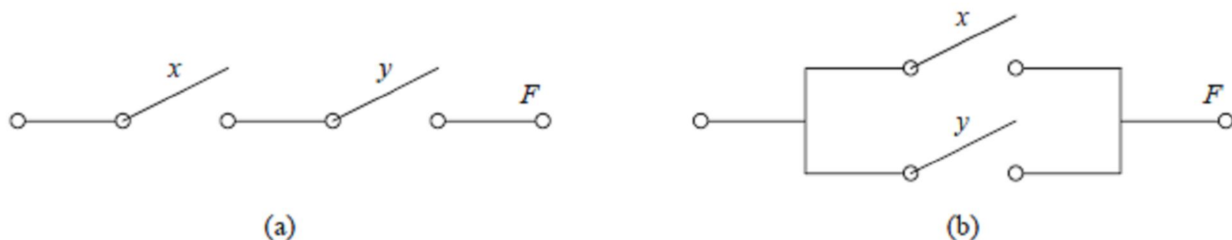


Figure 1.1(c): Connection of two binary switches: (a) in series; (b) in parallel.
If two switches are connected in series as in (a), then both switches have to be on in order for the output F to be a 1. In other words,

$F = 1$ if $x = 1$ AND $y = 1$. If either x or y is off, or both are off, then $F = 0$.

Translating this into a logic expression, we get

$F = x$ AND y

Hence, two switches connected in series give rise to the logical **AND** operator. In a Boolean function the AND operator is either denoted with a dot (**.**) or no symbol at all. Thus we can rewrite the above expression as

$$F = x \bullet y$$

Or simply as

$$F=xy$$

If we connect two switches in parallel as in (b), then only one switch needs to be on in order for the output F to be a 1. In other words, $F = 1$ if either $x = 1$, or $y = 1$, or both x and y are 1's. This means that $F = 0$ only if both x and y are 0's. Translating this into a logic expression, we get

$$F = x \text{ OR } y$$

and this gives rise to the logical **OR** operator. In a Boolean function, the OR operator is denoted with a plus symbol (+).
Thus we can rewrite the above expression as

$$F = x + y$$

In addition to the AND and OR operators, there is another basic logic operator ¯ the **NOT** operator, also known as the **INVERTER**. Whereas, the AND and OR operators have multiple inputs, the NOT operator has only one input and one output. The NOT operator simply inverts its input, so a 0 input will produce a 1 output, and a 1 becomes a 0.
In a Boolean function, the NOT operator is either denoted with an apostrophe symbol (′) or a bar on top (¯) as in

$$F = x'$$

When several operators are used in the same expression, the precedence given to the operators are, from highest to lowest, NOT, AND, and OR. The order of evaluation can be changed by means of using parenthesis. For example, the expression

$$F = xy + z'$$

means (x and y) or (not z), and the expression

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

3.3 Truth Tables

The operation of the AND, OR, and NOT logic operators can be formally described by using a **truth table** as shown in Table 1.1(b). A truth table is a two-dimensional array where there is one column for each input and one column for each output (a circuit may have more than one output). Since we are dealing with binary values, each input can be either a 0 or a 1. We simply enumerate all possible combinations of 0's and 1's for all the inputs.
Usually, we want to write these input values in the normal binary counting order. With two inputs, there are 2^2 combinations giving us the four rows in the table. The values in the output column are determined from applying the corresponding input values to the functional operator. For the AND truth table in (a) of Table 1.1(b), $F = 1$ only when x and y are both 1, otherwise, $F = 0$. For the OR truth table (b), $F = 1$ when either x or y or both is a 1, otherwise $F = 0$.
For the NOT truth table, the output F is just the inverted value of the input x.

Table 1.1(b): Truth tables for the three basic logical operators: (a) AND; (b) OR; (c) NOT.

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

(a)

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

(b)

x	F
0	1
1	0

(c)

Using a truth table is one method to formally describe the operation of a circuit or function. The truth table for any given logic expression (no matter how complex it is) can always be derived. Examples on the use of truth tables to describe digital circuits are given in the following sections. Another method to formally describe the operation of a circuit is by using Boolean expressions or Boolean functions.

3.4 Boolean Algebra

George Boole, in 1854, developed a system of mathematical logic, which we now call *Boolean algebra*. Based on Boole's idea, Claude Shannon, in 1938, showed that circuits built with binary switches can easily be described using Boolean algebra. The abstraction from switches being on and off to the use of Boolean algebra is as follows.

Let $B = \{0, 1\}$ be the Boolean algebra whose elements are one of the two values, 0 and 1. We define the operations AND (•), OR (+), and NOT (′) for the elements of B by the axioms in (a) of Table 1.1(c). These axioms are simply the definitions for the AND, OR, and NOT operators. A variable x is called a *Boolean variable* if x takes on only values in B, i.e. either 0 or 1. Consequently, we obtain the theorems in (b) of Table 1.1(c) for single variable and (c) for two and three variables.

Theorems in (b) of Table 1.1(c) can be proved easily by substituting the binary values into the expressions and using the axioms. For example, to show that Theorem 6a is true, we substitute 0 into x to get axiom 3a, and substitute 1 into x to get axiom 2a.

To prove the theorems in Table 1.1(c), we can use either one of two methods: 1) use a truth table, or 2) use axioms and theorems that have already been proven. We show these two methods in the following two examples.

Table 1.1(c): Boolean algebra axioms and theorems: (a) Axioms; (b) Single variable theorems; (c) two and three variable theorems.

1a.	$0 \bullet 0 = 0$	1b.	$1 + 1 = 1$
2a.	$1 \bullet 1 = 1$	2b.	$0 + 0 = 0$
3a.	$0 \bullet 1 = 1 \bullet 0 = 0$	3b.	$1 + 0 = 0 + 1 = 1$
4a.	$0' = 1$	4b.	$1' = 0$

(a)

5a.	$x \bullet 0 = 0$	5b.	$x + 1 = 1$	Null element
6a.	$x \bullet 1 = 1 \bullet x = x$	6b.	$x + 0 = 0 + x = x$	Identity
7a.	$x \bullet x = x$	7b.	$x + x = x$	Idempotent
8a.	$(x')' = x$			Double complement
9a.	$x \bullet x' = 0$	9b.	$x + x' = 1$	Inverse

(b)

10a.	$x \bullet y = y \bullet x$	10b.	$x + y = y + x$	Commutative
11a.	$(x \bullet y) \bullet z = x \bullet (y \bullet z)$	11b.	$(x + y) + z = x + (y + z)$	Associative
12a.	$x \bullet (y + z) = (x \bullet y) + (x \bullet z)$	12b.	$x + (y \bullet z) = (x + y) \bullet (x + z)$	Distributive
13a.	$x \bullet (x + y) = x$	13b.	$x + (x \bullet y) = x$	Absorption
14a.	$(x \bullet y) + (x \bullet y') = x$	14b.	$(x + y) \bullet (x + y') = x$	Combining
15a.	$(x \bullet y)' = x' + y'$	15b.	$(x + y)' = x' \bullet y'$	DeMorgan's

(c)

Example 1.1(a): Proof of theorem using a truth table.

Theorem 12a states that $x \bullet (y + z) = (x \bullet y) + (x \bullet z)$. To prove that Theorem 12a is true using a truth table, we need to show that for every combination of values for the three variables *x*, *y*, and *z*, the left-hand side of the expression is equal to the right-hand side. The truth table below is constructed as follows:

x	y	z	$(y + z)$	$(x \bullet y)$	$(x \bullet z)$	$x \bullet (y + z)$	$(x \bullet y) + (x \bullet z)$
0	0	0	0	0	0	0	0
0	0	1	1	0	0	0	0
0	1	0	1	0	0	0	0
0	1	1	1	0	0	0	0
1	0	0	0	0	0	0	0
1	0	1	1	0	1	1	1
1	1	0	1	1	0	1	1
1	1	1	1	1	1	1	1

We start with the first three columns labeled *x*, *y*, and *z*, and enumerate all possible combinations of values for these three variables. For each combination (row), we evaluate the intermediate expressions $y+z$, $x \bullet y$, and $x \bullet z$ by substituting the values of *x*, *y*, and *z* into the expression. Finally, we obtain the values for the last two columns, which correspond to the left-hand side and right-hand side of Theorem 12a. The values in these two columns are identical for every combination of *x*, *y*, and *z*, therefore, we can say that Theorem 12a is true.

Example 1.1(b): Proof of theorem using axioms and theorems.
Theorem 13b states that $x + (x \bullet y) = x$
To prove that Theorem 13b is true using axioms and theorems, we can argue as follows:

$$
\begin{aligned}
x + (x \bullet y) &= (x \bullet 1) + (x \bullet y) && \text{by Identity Theorem 6a} \\
&= x \bullet (1 + y) && \text{by Distributive Theorem 12a} \\
&= x \bullet (1) && \text{by Null element Theorem 5b} \\
&= x && \text{by Identity Theorem 6a}
\end{aligned}
$$

Example 1.1(b) shows that some theorems can be derived from others that have already been proven with the truth table. Full treatment of Boolean algebra is beyond the scope of this book and can be found in the references. For our purposes, we simply assume that all the theorems are true and will just use them to show that two circuits are equivalent as depicted in the next two examples.

Example 1.1(c): Use Boolean algebra to reduce the equation $F(x,y,z) = (x' + y' + x'y' + xy)\ (x' + yz)$ as much as possible.

$$
\begin{aligned}
F &= (x' + y' + x'y' + xy)\ (x' + yz) \\
&= (x' \bullet 1 + y' \bullet 1 + x'y' + xy)\ (x' + yz) \\
&= (x'\ (y + y') + y'\ (x + x') + x'y' + xy)\ (x' + yz) \\
&= (x'y + x'y' + y'x + y'x' + x'y' + xy)\ (x' + yz) \\
&= (x'y + x'y' + y'x + \cancel{y'x'} + \cancel{x'y'} + xy)\ (x' + yz) \\
&= (x'\ (y + y') + x\ (y + y'))\ (x' + yz) \\
&= (x' \bullet 1 + x \bullet 1)\ (x' + yz) \\
&= (x' + x)\ (x' + yz) \\
&= 1\ (x' + yz) \\
&= (x' + yz)
\end{aligned}
$$

Since the expression $(x' + y' + x'y' + xy)\ (x' + yz)$ reduces down to $(x' + yz)$, therefore, we do want to implement the circuit for the latter expression rather than the former because the circuit size for the latter is much smaller.

Example 1.1(d): Show, using Boolean algebra, that the two equations $F_1 = (xy' + x'y + x' + y' + z')\ (x + y' + z)$ and $F_2 = y' + x'z + xz'$ are equivalent.

$$
\begin{aligned}
F_1 &= (xy' + x'y + x' + y' + z')\ (x + y' + z) \\
&= xy'x + xy'y' + xy'z + x'yx + x'yy' + x'yz + x'x + x'y' + x'z + y'x + y'y' + y'z + z'x + z'y' + z'z \\
&= xy' + xy' + xy'z + 0 + 0 + x'yz + 0 + x'y' + x'z + xy' + y' + y'z + xz' + y'z' + 0 \\
&= xy' + xy'z + x'yz + x'y' + x'z + y' + y'z + xz' + y'z' \\
&= y'(x + xz + x' + 1 + z + z') + x'z(y + 1) + xz' \\
&= y' + x'z + xz' \\
&= F_2
\end{aligned}
$$

3.5 Duality Principle

Notice in Table 1.1(c) that we have listed the axioms and theorems in pairs. Specifically, we define the **dual** of a logic expression as one that is obtained by changing all + operators with • operators, and vice versa, and by changing all 0's with 1's, and vice versa. For example, the dual of the logic expression

$(x \bullet y' \bullet z) + (x \bullet y \bullet z') + (y \bullet z) + 0$

is

$(x+y'+z) \bullet (x+y+z') \bullet (y+z) \bullet 1$

The **duality principle** states that if a Boolean expression is true, then its dual is also true. Be careful in that it does not say that a Boolean expression is equivalent to its dual. For example, Theorem 5a in Table 1.1(c) says that $x \bullet 0 = 0$ is true, thus by the duality principle, its dual, $x + 1 = 1$ is also true.

However, $x \bullet 0 = 0$ is not equal to $x + 1 = 1$, since 0 is definitely not equal to 1.

We will see later that the inverse of a Boolean expression can be obtained by first taking the dual of that expression, and then complementing each Boolean variable in the resulting dual expression. In this respect, the duality principle is often used in digital logic design. Whereas an expression might be complex to implement, its inverse might be simpler, thus resulting in a smaller circuit, and inverting the final output of this circuit will produce the same result as from the original expression.

4.0 Conclusion

Boolean Expressions are equivalent expressions of the logic state of gates. Truth tables are tables which are set to list the possible inputs and find their corresponding outputs. By looking at a truth table, one is able to know the output of **any** possible combination of inputs. The **NOT** gate, the **OR** gate and the **AND** gate are three main types of logic gates. The rules for these operations may be summarized as follows:

OR	**AND**	**NOT**
0 + 0 = 0	0 . 0 = 0	0′ = 1
0 + 1 = 1	0 . 1 = 0	1′ = 0
1 + 0 = 1	1 . 0 = 0	
1 + 0 = 1	1 . 1 = 1	

Self Assessment Exercises

1. Convert 100011011011_2 to its decimal and octal equivalent

2. Convert 614_8 to decimal

3. Convert $24CE_{16}$ to decimal

5.0 Summary

In this unit, you learnt about

- The theoretical foundation for building digital logic circuits.
- The binary number system and conversion, also its addition
- How Truth Tables are used to show how circuits output responds to the various combination of logic levels at the inputs.
- Boolean algebras differ in a way from ordinary algebra in that Boolean constants and variables are allowed to have only two possible values, 0 or 1.
- The duality principle

6.0 Tutor Marked Assignment

1) Add this pair of binary numbers 10001111 + 000000001

2) For the function $F = AB'C' + AB$, find the logic value of F under the conditions—
   (*a*) A = 1, B = 0, C = 1; (*b*) A = 0, B = 1, C = 1;
   (*c*) A = 0, B = 0, C = 0

3) Draw truth tables for the following expressions:
   (*a*) $F = AC + AB$ (*b*) $F = AB (B + C + D')$
   (*c*) $Y= A (B' + C')$ (*d*) $Y = (A + B + C) AB'$
   (*e*) $F = ABC (C + D')$ (*f*) $F = AB + BA + C (A + B)$

4) Use a truth table to show that the following expressions are true:
   a) $w'z' + w'xy + wx'z + wxyz = w'z' + xyz + wx'y'z + wyz$
   b) $z + y' + yz' = 1$
   c) $xy'z' + x' + xyz' = x' + z'$
   d) $xy + x'z + yz = xy + x'z$

5) Simplify the following Boolean expressions using Boolean technique:
   (*a*) $AB + A (B + C) + B (B + C)$ (*b*) $AB(C + BD') (AB)'$
   (*c*) $A + AB + AB'C$ (*d*) $(A' + B)C + ABC$
   (*e*) $AB'C (BD + CDE) + AC'$ (*f*) $BD + B (D + E) + D' (D + F)$

7.0 Further Reading and Other Resources

1. Ronald J. Tocci (1988). "Digital Systems: Priciples and Applications", 4th Edition Prentice-Hall International edition.
2. http://en.wikipedia.org/wiki/Logic_gate
3. http://www.discovercircuits.com/D/digital.htm
4. http://www.encyclopedia.com/doc/1G1-168332407.html
5. http://www.logiccircuit.org/

MODULE 1 - Basic Logic Operators and Logic Expressions
UNIT 2: BOOLEAN ALGEBRA AND FUNCTIONS

Contents **Pages**

1.0 Introduction …………………………………………………………………. 15

2.0 Objectives …………………………………………………………………. 15

3.0 Boolean Theorem ……………………………..…………………………….. 15

3.1 DeMorgan's Theorem ……………………………………………………….. 15

3.2 Boolean Function and the Inverse …...……………………………………….. 16

3.3 Minterms and Maxterms……………………………………………………... 20

3.3.1 Minterms ……………………………………………………….……… 20

3.3.2 Maxterms ……………………………………………………………… 22

3.4 Canonical, Standard, and Non-Standard Forms …..……………………………… 24

4.0 Conclusion ………………………………………………………………... 25

5.0 Summary ………………………………………………………………….. 25

6.0 Tutor Marked Assignment………………………..…………………………….. 25

7.0 Further Reading and Other Resources ………………………………………… 25

1.0 Introduction

Boolean algebra is a tool for the analysis and design of digital system. Boolean algebra differs in a way from ordinary algebra in that Boolean constants and variables are allowed to have only two possible values, 0 or 1. A Boolean variable is a quantity that may, at different times, be equal to either 0 or 1.

2.0 Objectives

Upon completion of this unit, you will be able to:

- Simplify complex logic expressions by applying the various Boolean algebra laws and rules.
- Simplify intricate Boolean equations by applying the DeMorgan's theorem.
- Understand minterms and maxterms.
- Understand the canonical, standard and non-standard forms of Boolean functions.

3.0 Boolean Theorem

Boolean theorem are rules that can help us to simplify logic expressions and logic circuits as shown in Table 1.1(c). When this is done, the reduced expression will produce a circuit that is less complex than the one which the original expression would have produced. The next section discusses in detail the DeMorgan's theorem

3.1 DeMorgan's Theorem

Two of the most important theorems of Boolean algebra were contributed by a great mathematician named DeMorgan. DeMorgan's theorems are extremely useful in simplifying expressios in which a product or sum of variables is inverted. The two theorems as in Theorem 15a and 15b are:

(i) $(x + y)' = x' \cdot y'$

(ii) $(x \cdot y)' = x' + y'$

The theorem in (i) says that when the OR sum of two variables is inverted, this is the same as inverting each variable individually and then ANDing these inverted variables. The (ii) says that when the AND product of two variables is inverted, this is the same as inverting each variable individually and then ORing them. Each of DeMorgan's theorems can be readily proven by checking for all possible combinations of x and y. Although these theorems have been stated in terms of single variables x and y, they are equally valid for situations where x and/or y are expressions that contain more than one variable. For example, let's apply them to the expression

$$(AB + C)' = (AB)' \cdot C'$$

Note that here we treated (AB)′ as *x* and C as *y*. The result can be further simplified since we have a product (AB)′ that is inverted.

Using the theorem in (ii), the expression becomes

$$(AB')' \,.\, C' = (A' + (B')') \,.\, C'$$

Notice that we can replace $(B')'$ by B, so we finally have

$$(A' + B) \,.\, C' = A'C' \,.\, BC'$$

The final result contains only inverter signs that invert a single variable.

Class work:

Simplify the expression $z = ((A' + C) \,.\, (B + D'))'$

Using (ii), we can write this as

$$z = ((A' + C)' + (B + D')'$$

We can think of this as breaking the large inverter sign down the middle and changing the AND sign (.) to an OR sign (+). Now the term $((A' + C)'$ can be simplified by applying the theorem in (i). Likewise, $(B + D')'$ can be simplified.

$$z = ((A' + C)' + (B + D')'$$

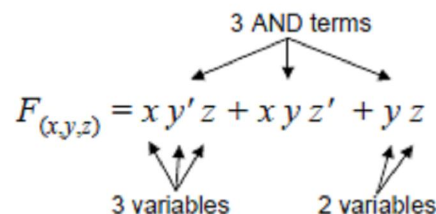
$$= ((A')' \,.\, C') + (B' \,.\, (D')')$$

here, we have broken the larger inverter signs down the middle and replaced the (+) with a (.). Canceling out the double inversions, we have finally

$$z = AC' + B'D$$

3.2 Boolean Function and the Inverse

As we have seen, any digital circuit can be described by a logical expression, also known as a *Boolean function*. Any Boolean functions can be formed from binary variables and the Boolean operators •, +, and ′ (for AND, OR, and NOT respectively). For example, the following Boolean function uses the three variables or literals *x*, *y*, and *z*. It has three **AND terms** (also referred to as **product terms**), and these AND terms are ORed (summed) together. The first two AND terms contain all three variables each, while the last AND term contains only two variables. By definition, an AND (or product) term is either a single variable, or two or more variables ANDed together. Quite often, we refer to functions that are in this format as a **sum-of-products** or **or-of-ands**.



The value of a function evaluates to either a 0 or a 1 depending on the given set of values for the variables. For example, the function above evaluates to a 1 when any one of the three AND terms evaluate to a 1, since 1 OR x is 1. The first AND term, $xy'z$, equals to a 1 if

$x = 1$, $y = 0$, and $z = 1$

Since if we substitute these values for x, y, and z into the first AND term $xy'z$, we get a 1. Similarly, the second AND term, xyz', equals to a 1 if

$x = 1$, $y = 1$, and $z = 0$.

The last AND term, yz, has only two variables. What this means is that the value of this term is not dependent on the missing variable x. In other words x can be either a 0 or a 1, but as long as $y = 1$ and $z = 1$, this term will equal to a 1. Thus, we can summarize by saying that F evaluates to a 1 if

$x = 1$, $y = 0$, and $z = 1$

or

$x = 1$, $y = 1$, and $z = 0$

or

$x = 0$, $y = 1$, and $z = 1$

or

$x = 1$, $y = 1$, and $z = 1$.

Otherwise, F evaluates to a 0.

It is often more convenient to summarize the above verbal description of a function with a truth table as shown in Table 1.2(a) under the column labeled F. Notice that the four rows in the table where $F = 1$ match the four cases in the description above.

x	y	z	F	F'
0	0	0	0	1
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	0

Table 1.2(a): Truth table for the function $F = xy'z + xyz' + yz$

The inverse of a function, denoted by F', can be easily obtained from the truth table for F by simply changing all the 0's to 1's and 1's to 0's as shown in the truth table in Table 1.2(a) under the column labeled F'. Therefore, we can write the Boolean function for F' in the sum-of-products format, where the AND terms are obtained from those rows where $F' = 1$. Thus, we get

$$F' = x'y'z' + x'y'z + x'yz' + xy'z'$$

To deduce F' algebraically from F requires the use of DeMorgan's Theorem twice. For example, using the same function

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

we obtain F' as follows

$$
\begin{aligned}
F' &= (xy'z + xyz' + yz)' \\
&= (xy'z)' \bullet (xyz')' \bullet (yz)' \\
&= (x'+y+z') \bullet (x'+y'+z) \bullet (y'+z')
\end{aligned}
$$

There are three things to notice about this equation for F'. First, F' is just the dual of F and then having all the variables inverted. Second, instead of being in a sum-of-products format, it is in a **product-of-sums** (**and-of-ors**) format where three OR terms (also referred to as sum terms) are ANDed together.

Third, from the same original function F, we obtained two different equations for F'. From the truth table, we obtained

$$F' = x'y'z' + x'y'z + x'yz' + xy'z'$$

and from applying DeMorgan's Theorem to F, we obtained

$$F' = (x'+y+z') \bullet (x'+y'+z) \bullet (y'+z')$$

Hence, we must conclude that these two expressions for F', where one is in the sum-of-products format, and the other is in the product-of-sums format, are equivalent. In general, all functions can be expressed in either the sum-of-products or product-of-sums format.

Thus, we should also be able to express the same function $F = xy'z + xyz' + yz$ in the product-of-sums format.

We can derive it using one of two methods. For method one, we can start with F' and apply DeMorgan's Theorem to it just like how we obtained F' from F.

$$
\begin{aligned}
F &= F'' \\
&= (x'y'z' + x'y'z + x'yz' + xy'z')' \\
&= (x'y'z')' \bullet (x'y'z)' \bullet (x'yz')' \bullet (xy'z')' \\
&= (x+y+z) \bullet (x+y+z') \bullet (x+y'+z) \bullet (x'+y+z)
\end{aligned}
$$

For the second method, we start with the original F and convert it to the product-of-sums format using the Boolean theorems.

$$
\begin{aligned}
F &= xy'z + xyz' + yz \\
&= (x+x+y) \bullet (x+x+z) \bullet (x+y+y) \bullet (x+y+z) \bullet (x+z'+y) \bullet \cancel{(x+z'+z)} \bullet && \text{(step 1)} \\
&\quad \cancel{(y'+x+y)} \bullet (y'+x+z) \bullet \cancel{(y'+y+y)} \bullet \cancel{(y'+y+z)} \bullet \cancel{(y'+z'+y)} \bullet \cancel{(y'+z'+z)} \bullet \\
&\quad (z+x+y) \bullet (z+x+z) \bullet (z+y+y) \bullet (z+y+z) \bullet \cancel{(z+z'+y)} \bullet \cancel{(z+z'+z)} \\
&= (x+y) \bullet (x+z) \bullet \cancel{(x+y)} \bullet (x+y+z) \bullet (x+z'+y) \bullet (y'+x+z) \bullet \cancel{(z+x+y)} \bullet \cancel{(z+x)} \bullet (z+y) \bullet \cancel{(z+y)} && \text{(step 2)} \\
&= (x+y) \bullet (x+z) \bullet (x+y+z) \bullet (x+y+z') \bullet (x+y'+z) \bullet (z+y) && \text{(step 3)} \\
&= (x+y+zz') \bullet (x+yy'+z) \bullet (x+y+z) \cdot (x+y+z') \bullet (x+y'+z) \bullet (xx'+y+z) && \text{(step 4)} \\
&= (x+y+z) \bullet (x+y+z') \bullet \cancel{(x+y+z)} \bullet (x+y'+z) \bullet \cancel{(x+y+z)} \bullet \cancel{(x+y+z')} \bullet \cancel{(x+y'+z)} \bullet \cancel{(x+y+z)} \bullet (x'+y+z) && \text{(step 5)} \\
&= (x+y+z) \bullet (x+y+z') \bullet (x+y'+z) \bullet (x'+y+z)
\end{aligned}
$$

In the first step, we apply Theorem 12b (Distributive) in Unit 1 to get every possible combination of sum terms. For example, the first sum term $(x+x+y)$ is obtained from getting the first x from $xy'z$, the second x from xyz', and the y from yz. The second sum term $(x+x+z)$ is obtained from getting the first x from $xy'z$, the second x from xyz', and the z from yz. This is repeated for all combinations. In this step, the sum terms, such as $(x+z'+z)$, where it contains variables of the form $v + v'$ can be eliminated since $v + v' = 1$, and $1 \bullet x = x$.

In the second and third steps, duplicate variables and terms are eliminated. For example, the term $(x+x+y)$ is equal to just $(x+y+y)$, which is just $(x+y)$. The term $(x+z'+z)$ is equal to $(x+1)$, which is equal to just 1, and therefore, can be eliminated completely from the expression.

In the fourth step, every sum term with a missing variable will have that variable added back in by using Theorems 6b and 9a, which says that $x + 0 = x$ and $yy' = 0$, therefore, $x + yy' = x$.

Step five uses the Distributive Theorem, and the resulting duplicate terms are again eliminated to give us the format that we want.

Functions that are in the product-of-sums format (such as the one shown below) are more difficult to deduce when they evaluate to a 1. For example, using

$$F' = (x'+y+z') \bullet (x'+y'+z) \bullet (y'+z')$$

F' evaluates to a 1 when all three terms evaluate to a 1. For the first term to evaluate to a 1, x can be 0, or y can be 1, or z can be 0. For the second term to evaluate to a 1, x can be 0, or y can be 0, or z can be 1. Finally, for the last term, y can be 0, or z can be 0, or x can be either a 0 or a 1. As a result, we end up with many more combinations to consider, even though many of the combinations are duplicates.

However, it is easier to determine when a product-of-sums format expression evaluates to a 0. For example, using the same expression

$$F' = (x'+y+z') \bullet (x'+y'+z) \bullet (y'+z')$$

F' evaluates to 0 when any one of the three OR terms is 0, since 0 AND x is 0; and this happens when $x = 1$, $y = 0$, and $z = 1$ for the first OR term,

or

$x = 1$, $y = 1$, and $z = 0$ for the second OR term,

or

$y = 1$, $z = 1$, and x can be either 0 or 1 for the last or term.

Similarly, for a sum-of-products format expression, it is easy to evaluate when it is a 1, but difficult to evaluate when it is a 0.

These four conditions in which F' evaluates to a 0 match exactly those rows in the table where $F' = 0$. Therefore, we see that in general, the unique algebraic expression for any Boolean function can be specified by either (1) selecting the rows from the truth table where the function is a 1 and use the sum-of-products format, or (2) selecting the rows from the truth table where the function is a 0 and use the product-of-sums format.

Whatever format we decide to use, the one thing to remember is that we are always interested in only when the function (or its inverse) is equal to a 1.

Figure 1.2(a) summarizes these two formats for the function $F = xy'z + xyz' + yz$ and its inverse F'. Notice that the sum-of-products format for F is the dual with its variables inverted of the product-of-sums format for F'. Similarly, the product-of-sums format for F is the dual with its variables inverted of the sum-of-products format for F'.

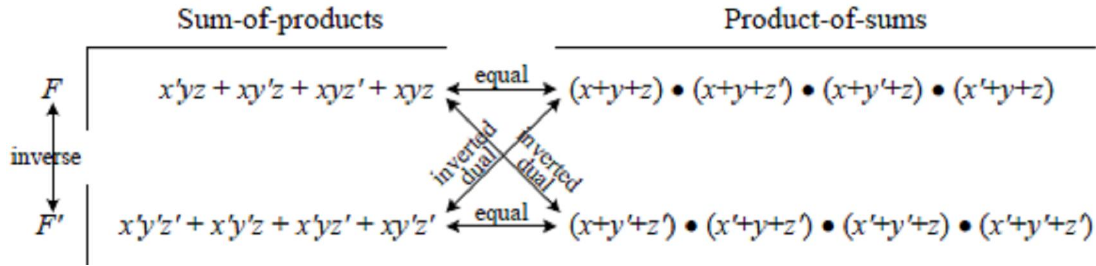


Figure 1.2(a): Relationships between the function $F = xy'z + xyz' + yz$ and its inverse F', and the sum-of-products and product-of-sums formats. The label "inverted dual" means applying the duality principle and then inverting the variables.

3.3 Minterms and Maxterms

As you recall, a product term is a term with either a single variable, or two or more variables ANDed together, and a sum term is a term with either a single variable, or two or more variables ORed together. To differentiate between a term that contains any number of variables with a term that contains *all* the variables used in the function, we use the words minterm and maxterm. We are not introducing new ideas here, rather, we are just introducing two new words and notations for defining what we have already learned.

3.3.1 Minterms

A **minterm** is a product term that contains all the variables used in a function. For a function with n variables, the notation m_i where $0 \leq i < 2n$, is used to denote the minterm whose index i is the binary value of the n variables such that the variable is complemented if the value assigned to it is a 0, and uncomplemented if it is a 1.

For example, for a function with three variables x, y, and z, the notation m_3 is used to represent the term in which the values for the variables xyz are 011 (for the subscript 3). Since we want to complement the variable whose value is a 0, and uncomplement it if it is a 1. Hence m_3 is for the minterm $x'yz$. The (a) of **Table 1.2(b)** shows the eight minterms and their notations for $n = 3$ using the three variables x, y, and z.

When specifying a function, we usually start with product terms that contain all the variables used in the function. In other words, we want the **sum of minterms**, and more specifically the sum of the one-minterms, that is the minterms for which the function is a 1 (as opposed to the zero-minterms, that is the minterms for which the function is a 0). We use the notation **1-minterm** to denote one-minterm, and **0-minterm** to denote zero-minterm.

Table 1.2(b): (a) Minterms for three variables. (b) Maxterms for three variables.

x	y	z	Minterm	Notation
0	0	0	$x'y'z'$	m_0
0	0	1	$x'y'z$	m_1
0	1	0	$x'yz'$	m_2
0	1	1	$x'yz$	m_3
1	0	0	$xy'z'$	m_4
1	0	1	$xy'z$	m_5
1	1	0	xyz'	m_6
1	1	1	xyz	m_7

(a)

x	y	z	Maxterm	Notation
0	0	0	$x+y+z$	M_0
0	0	1	$x+y+z'$	M_1
0	1	0	$x+y'+z$	M_2
0	1	1	$x+y'+z'$	M_3
1	0	0	$x'+y+z$	M_4
1	0	1	$x'+y+z'$	M_5
1	1	0	$x'+y'+z$	M_6
1	1	1	$X'+y'+z'$	M_7

(b)

The function from the previous section

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

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

and repeated in the following truth table has the 1-minterms m_3, m_5, m_6, and m_7.

x	y	z	F	F'	Minterm	Notation
0	0	0	0	1	$x'y'z'$	m_0
0	0	1	0	1	$x'y'z$	m_1
0	1	0	0	1	$x'yz'$	m_2
0	1	1	1	0	$x'yz$	m_3
1	0	0	0	1	$xy'z'$	m_4
1	0	1	1	0	$xy'z$	m_5
1	1	0	1	0	xyz'	m_6
1	1	1	1	0	xyz	m_7

Thus, a shorthand notation for the function is

$F(x, y, z) = m_3 + m_5 + m_6 + m_7$

By just using the minterm notations, we do not know how many variables are in the original function. Consequently, we need to explicitly specify the variables used by the function as in $F(x, y, z)$. We can further simplify the notation by using the standard algebraic symbol Σ for summation. Therefore, we have

$F(x, y, z) = \Sigma(3, 5, 6, 7)$

These are just different ways of expressing the same function. Since a function is obtained from the sum of the 1-minterms, the inverse of the function, therefore, must be the sum of the 0-minterms. This can be easily obtained by replacing the set of indices with those that were excluded from the original set.

Example 1.2(a): Given the Boolean function $F(x, y, z) = y + x'z$, use Boolean algebra to convert the function to the sum-of-minterms format.

This function has three variables. In a sum-of-minterms format, all product terms must have all variables. To do so, we need to expand each product term by ANDing it with $(v + v')$ for every

missing variable v in that term. Since $(v + v') = 1$, therefore, ANDing a product term with $(v + v')$ does not change the value of the term.

$F = y + x'z$
$= y(x+x')(z+z') + x'z(y+y')$ expand 1st term by ANDing it with $(x+x')(z+z')$, and 2nd term with $(y+y')$
$= xyz + xyz' + x'yz + x'yz' +$ ~~$x'yz$~~ $+ x'y'z$
$= m_7 + m_6 + m_3 + m_2 + m_1$
$= \Sigma(1, 2, 3, 6, 7)$ sum of 1-minterms

Example 1.2(b): Given the Boolean function $F(x, y, z) = y + x'z$, use Boolean algebra to convert the inverse of the function to the sum-of-minterms format.

$F' = (y + x'z)'$ inverse
$= y' \bullet (x'z)'$ use DeMorgan
$= y' \bullet (x+z')$ use DeMorgan
$= y'x + y'z'$ use Distributive Theorem to change to sum of products format
$= y'x(z+z') + y'z'(x+x')$ expand 1st term by ANDing it with $(z+z')$, and 2nd term with $(x+x')$
$= xy'z + xy'z' +$ ~~$xy'z'$~~ $+ x'y'z'$
$= m_5 + m_4 + m_0$
$= \Sigma(0, 4, 5)$ sum of 0-minterms

3.3.2 Maxterms

Similar to a minterm, a **maxterm** is a sum term that contains all the variables used in the function. For a function with n variables, the notation M_i where $0 \le i < 2n$, is used to denote the maxterm whose index i is the binary value of the n variables such that the variable is complemented if the value assigned to it is a 1, and uncomplemented if it is a 0.
For example, for a function with three variables x, y, and z, the notation M_3 is used to represent the term in which the values for the variables xyz are 011. For maxterms, we want to complement the variable whose value is a 1, and uncomplement it if it is a 0. Hence M_3 is for the maxterm $x + y' + z'$. (b) of Figure 1.2(c) shows the eight maxterms and their notations for $n = 3$ using the three variables x, y, and z. We have seen that a function can also be specified as a product of sums, or more specifically, a **product of 0-maxterms**, that is, the maxterms for which the function is a 0. Just like the minterms, we use the notation **1-maxterm** to denote one-maxterm, and **0-maxterm** to denote zero-maxterm. Thus, the function

$$F(x, y, z) = xy'z + xyz' + yz$$
$$= (x + y + z) \bullet (x + y + z') \bullet (x + y' + z) \bullet (x' + y + z)$$

which is shown in the following table

x	y	z	F	F'	Maxterm	Notation
0	0	0	0	1	$x+y+z$	M_0
0	0	1	0	1	$x+y+z'$	M_1
0	1	0	0	1	$x+y'+z$	M_2
0	1	1	1	0	$x+y'+z'$	M_3
1	0	0	0	1	$x'+y+z$	M_4
1	0	1	1	0	$x'+y+z'$	M_5
1	1	0	1	0	$x'+y'+z$	M_6
1	1	1	1	0	$x'+y'+z'$	M_7

can be specified as the product of the 0-maxterms M_0, M_1, M_2, and M_4. The shorthand notation for the function is

$$F(x, y, z) = M_0 \bullet M_1 \bullet M_2 \bullet M_4$$

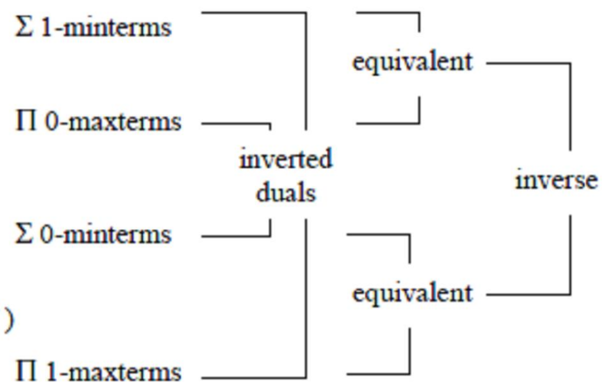
Again, by using the standard algebraic symbol Π for product, the notation is further simplified to

$$F(x, y, z) = \Pi\,(0, 1, 2, 4)$$

The following summarizes these relationships for the function $F = xy'z + xyz' + yz$ and its inverse. Comparing these equations with those in table 1.2(a), we see that they are identical.

$$\begin{aligned} F(x, y, z) &= x'yz + xy'z + xyz' + xyz \\ &= m_3 + m_5 + m_6 + m_7 \\ &= \Sigma(3, 5, 6, 7) \\ &= (x+y+z) \bullet (x+y+z') \bullet (x+y'+z) \bullet (x'+y+z) \\ &= M_0 \bullet M_1 \bullet M_2 \bullet M_4 \\ &= \Pi(0, 1, 2, 4) \end{aligned}$$

$$\begin{aligned} F'(x, y, z) &= x'y'z' + x'y'z + x'yz' + xy'z' \\ &= m_0 + m_1 + m_2 + m_4 \\ &= \Sigma(0, 1, 2, 4) \\ &= (x+y'+z') \bullet (x'+y+z') \bullet (x'+y'+z) \bullet (x'+y'+z') \\ &= M_3 \bullet M_5 \bullet M_6 \bullet M_7 \\ &= \Pi(3, 5, 6, 7) \end{aligned}$$



Notice that it is always the Σ of minterms and Π of maxterms; you never have Σ of maxterms or Π of minterms.

Example 1.2(c): Given the Boolean function $F(x, y, z) = y + x'z$, use Boolean algebra to convert the function to the product-of-maxterms format.

To change a sum term to a maxterm, we expand each term by ORing it with (vv') for every missing variable v in that term. Since $(vv') = 0$, therefore, ORing a sum term with (vv') does not change the value of the term.

$$
\begin{aligned}
F &= y + x'z \\
&= y + (x'z) \\
&= (y+x')(y+z) && \text{use Distributive Theorem to change to product of sums format} \\
&= (y+x'+zz')(y+z+xx') && \text{expand 1st term by ORing it with } zz'\text{, and 2nd term with } xx' \\
&= (x'+y+z)\,(x'+y+z')\,(x+y+z)\,\cancel{(x'+y+z)} \\
&= M_4 \bullet M_5 \bullet M_0 \\
&= \Pi(0, 4, 5) && \text{product of 0-maxterms}
\end{aligned}
$$

Example 1.2(d): Given the Boolean function $F(x, y, z) = y + x'z$, use Boolean algebra to convert the inverse of the function to the product-of-maxterms format.

$$
\begin{aligned}
F' &= (y + x'z)' && \text{inverse} \\
&= y' \bullet (x'z)' && \text{use DeMorgan} \\
&= y' \bullet (x+z') && \text{use DeMorgan} \\
&= (y'+xx'+zz') \bullet (x+z'+yy') && \text{expand 1st term by ORing it with } xx'+zz'\text{, and 2nd term with } yy' \\
&= (x+y'+z)\,(x+y'+z')\,(x'+y'+z)\,(x'+y'+z')\,(x+y+z')\,\cancel{(x+y'+z')} \\
&= M_2 \bullet M_3 \bullet M_6 \bullet M_7 \bullet M_1 \\
&= \Pi(1, 2, 3, 6, 7) && \text{product of 1-maxterms}
\end{aligned}
$$

3.4 Canonical, Standard, and non-Standard Forms

Any Boolean function that is expressed as a sum of minterms, or as a product of maxterms is said to be in its **canonical form**. For example, the following two expressions are in their canonical forms

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

$F' = (x+y'+z') \bullet (x'+y+z') \bullet (x'+y'+z) \bullet (x'+y'+z')$

As noted from the previous section, to convert a Boolean function from one canonical form to its other equivalent canonical form, simply interchange the symbols Σ with Π, and list the index numbers that were excluded from the original form. For example, the following two expressions are equivalent

$F1(x, y, z) = \sum 3, 5, 6, 7)$

$F2(x, y, z) = \boldsymbol{\pi}\,(0, 1, 2, 4)$

To convert a Boolean function from one canonical form to its inverse, simply interchange the symbols ∑ with **π**, and list the same index numbers from the original form. For example, the following two expressions are inverses

$F1(x, y, z) = \sum 3, 5, 6, 7)$

$F2(x, y, z) = \boldsymbol{\pi}\,(3, 5, 6, 7)$

A Boolean function is said to be in a **standard form** if a sum-of-products expression or a product-of-sums expression has at least one term that is not a minterm or a maxterms respectively. In other words, at least one term in the expression is missing at least one variable. For example, the following expression is in a standard form because the last term is missing the variable x.

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

Sometimes, common variables in a standard form expression can be factored out. The resulting expression is no longer in a sum-of-products or product-of-sums format. These expressions are in a **non-standard form**. For example, starting with the previous expression, if we factor out the

common variable x from the first two terms, we get the following expression, which is in a non-standard form.

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

4.0 Conclusion

The switching functions can be expressed with Boolean equations. Complex Boolean equations can be simplified by a new kind of algebra, which is popularly called Switching Algebra or Boolean algebra. When a Boolean expression is implemented with logic gates, each literal in the function is designated as input to the gate. The literal may be a primed or unprimed variable. Minimization of the number of literals and the number of terms leads to less complex circuits as well as less number of gates, which should be a designer's aim. There are several methods to minimize the Boolean function.

Self Assessment Exercise

Use DeMorgan's theorems to convert the expression $z = (A + B) \cdot (C')'$ to one that has only single-variable inversions.

5.0 Summary

In this unit, you learnt about:

- How to simplify Boolean equations by applying the DeMorgan's theorem.
- The minterms and maxterms.
- The canonical, standard and non-standard forms of Boolean functions.

6.0 Tutor Marked Assignment

1. What is meant by duality in Boolean algebra?
2. State DeMorgan's theorem.
3. Prove the following using Boolean theorems:
   (*a*) $(A + C)(A + D)(B + C)(B + D) = AB + CD$
   (*b*) $(A' + B' + D')(A' + B + D')(B + C + D)(A + C')(A + C' + D) = A'C'D + ACD' + BC'D'$
4. Find the canonical sum of products and product of sums expression for the function
   $F = X1X2X3 + X1X3X4 + X1X2X4$.
5. (*a*) Convert $Y = ABCD + A'BC + B'C'$ into a sum of minterms by algebraic method.
   (*b*) Convert $Y = AB + B'CD$ into a product of maxterms by algebraic method.

7.0 Further Reading and Other Resources

1. Ronald j. Tocci (1988). "Digital Systems: Priciples and Applications", 4th Edition Prentice-Hall International edition.
2. http://en.wikipedia.org/wiki/Logic_gate
3. http://www.discovercircuits.com/D/digital.htm
4. http://www.encyclopedia.com/doc/1G1-168332407.html
5. http://www.logiccircuit.org/

MODULE 1 - Basic Logic Operators and Logic Expressions

UNIT 3 - Logic Gates and Circuit Diagrams

Contents **Pages**

1.0 Introduction ………………………………………………………………. 27

2.0 Objectives ……………………………………………………………. 27

3.0 Logic gates ………………………………..……………………………….. 27

3.1 Types of Logic gates ………………………………………………...…. 27

3.1.1 AND gate……………………..……………………………………… 27

3.1.2 OR gate ……………………………………………………….…... 27

3.1.3 NOT circuit (INVERTER) ……………………………………….…… 28

3.2 Describing Logic Circuits Algebraically ……………………………….…... 30

3.2.1 Circuits containing INVERTERS …………………………..………. 31

3.3 Constructing Circuits from Boolean Expression…..……………………….……….. 31

4.0 Conclusion …………………………………………………….………… 32

5.0 Summary ………………………………………………………………… 33

6.0 Tutor Marked Assignment……………………..………………………… 33

7.0 Further Reading and Other Resources ……………………………………….. 33

1.0 Introduction

As Boolean functions are expressed in terms of AND, OR, and NOT operations, it is easier to implement the Boolean functions with some basic types of gates. However, for all practical purposes, it is possible to construct other types of logic gates. **Logic gates** are the actual physical implementations of the logical operators discussed in the previous units.

2.0 Objectives

Upon completion of this unit, you will be able to:

- Describe the operation of and construct the truth tables for the AND, NAND, OR and NOR gates.
- Discuss the implementation of logical operators with Logic gates
- Understand the construction of Circuit diagrams

3.0 Logic Gates

Logic gates are the actual physical implementations of the logical operators discussed in the previous units. Transistors, acting as tiny electronic binary switches are connected together to form these gates.

3.1 Types of Logic Gates

There are basically the AND gate, the OR gate, and the NOT gate (also called the INVERTER) for the corresponding AND, OR, and NOT logical operators. These gates form the basic building blocks for all digital logic circuits. The name "gate" comes from the fact that these devices operate like a door or gate to let or not to let things (in our case, current) through.
In drawing digital circuit diagrams, also called **schematic diagrams**, or just **schematics**, we use special **logic symbols** to denote these gates as shown in Figure 1.3(a).

3.1.1 AND gate

The AND gate, or specifically, the 2-input AND gate, in (a) of Figure 1.3(a) has two input connections coming in from the left and one output connection going out on the right. The AND gate output is equal to the AND product of the logic inputs. In other words, the AND gate is a circuit that operates such that its output is high only when all its inputs are high. For all other cases the AND gate output is low.

3.1.2 OR gate

Similarly, the 2-input OR gate in (b) of Figure 1.3(a) has two input connections and one output connection. In digital circuitry, an OR gate is a circuit that has two or more inputs and whose output is equal to the OR sum of the inputs. The OR gate operates such thatits output is high (logic 1), if either input or both are at alogic 1 level. The OR gate output will be low (logic 0) only if all its inputs are at logic 0. This same idea can be extended to more than two inputs.

3.1.3 NOT circuit (INVERTER)

The INVERTER in (c) of Figure 1.3(a) has one input coming from the left and one output going to the right. In other words, it always has only a single input, and its output logic level is always opposite to the logic level of this input.

The outputs from these gates, of course, are dependent on their inputs, and are defined by their logical functions.

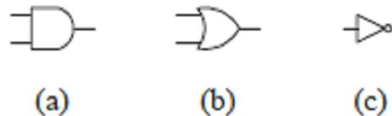


Figure 1.3(a) Logic symbols for the three basic logic gates: (a) 2-input AND; (b) 2-input OR; (c) NOT.

Sometimes, an AND gate or an OR gate with more than two inputs are needed. Hence, in addition to the 2-input AND and OR gates, there are 3-input, 4-input, or as many inputs as are needed of the AND and OR gates. In practice, however, the number of inputs is limited to a small number, such as five. The logic symbols for some of these gates are shown in (a) to (d) of Figure 1.3(b).

There are several other gates that are variants of the three basic gates that are also often used in digital circuits.

They are the NAND gate, the NOR gate, the XOR gate, and the XNOR gate. The NAND gate is derived from an AND gate and the INVERTER connected in series so that the output of the AND gate is inverted. The name "NAND" comes from the description "Not AND." Similarly, the NOR gate is the OR gate with its output inverted. The XOR, or eXclusive OR gate is like the OR gate except that when both inputs are 1, the output is a 0 instead. The XNOR, or eXclusive NOR gate is just the inverse of the XOR gate for when there are an even number of inputs (like 2 inputs). When there are an odd number of inputs (like 3 inputs), the XOR is the same as the XNOR. The logic symbols and their truth tables for some of these gates are shown in Figure 1.3(b) and Table 1.3(a) respectively.

Notice, in Table 1.3(a), the use of the little circle or bubble at the output of some of the logic symbols. This bubble is used to denote the inverted value of a signal. For example, the NAND gate is the inverse of the AND gate, thus the NAND gate logic symbol is the same as the AND gate logic symbol except that it has the extra bubble at the output.

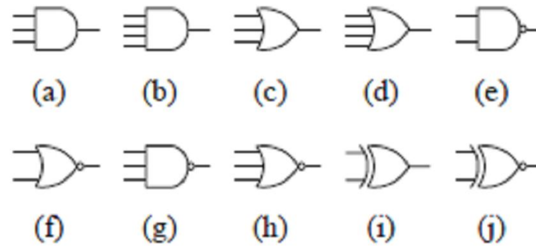


Figure 1.3(b) Logic symbols for: (a) 3-input AND; (b) 4-input AND; (c) 3-input OR; (d) 4-input OR; (e) 2-input NAND; (f) 2-input NOR; (g) 3-input NAND; (h) 3-input NOR; (i) 2-input XOR; (j) 2-input XNOR.

Table 1.3(a): Truth tables for: 2-input NAND; 2-input NOR; 2-input XOR; 2-input XNOR; 3-input AND; 3-input OR; 3-input NAND; 3-input NOR; 3-input XOR; 3-input XNOR.

x	y	2-NAND $(x \bullet y)'$	2-NOR $(x+y)'$	2-XOR $x \oplus y$	2-XNOR $x \odot y$
0	0	1	1	0	1
0	1	1	0	1	0
1	0	1	0	1	0
1	1	0	0	0	1

x	y	z	3-AND $x \bullet y \bullet z$	3-OR $x+y+z$	3-NAND $(x \bullet y \bullet z)'$	3-NOR $(x+y+z)'$	3-XOR $x \oplus y \oplus z$	3-XNOR $x \odot y \odot z$
0	0	0	0	0	1	1	0	0
0	0	1	0	1	1	0	1	1
0	1	0	0	1	1	0	1	1
0	1	1	0	1	1	0	0	0
1	0	0	0	1	1	0	1	1
1	0	1	0	1	1	0	0	0
1	1	0	0	1	1	0	0	0
1	1	1	1	1	0	0	1	1

The notations used for these gates in a logical expression are $(xy)'$ for the 2-input NAND gate, $(x+y)'$ for the 2-input NOR gate, $x \oplus y$ for the XOR gate, and $x \bigcirc y$ for the XNOR gate.

Looking at the truth table for the 2-input XOR gate, we can derive the equation for the 2-XOR gate as

$$x \oplus y = x'y + xy'$$

Similarly, the equation for the 2-input XNOR gate as derived from the 2-XNOR truth table is

$$x \odot y = x'y' + xy$$