Title	Digital Electronics
Adaption and Typesetting	**Dr. Jeetendra Pande**, Assistant Professor- School of CS & IT, Uttarakhand Open University, Haldwani
ISBN:	

Acknowledgement

The University acknowledges with thanks to NPTEL, KKHSOU and the author for providing the study material on NPTEL and KKHSOU portals under Creative Commons Attribution-ShareAlike - CC BY-SA.

Published By: Uttarakhand Open University

DIGITAL ELECTRONICS

Index

DIGITAL SYSTEMS: MOTIVATION

A digital circuit is one that is built with devices with two well-defined states. Such circuits can process information represented in binary form. Systems based on digital circuits touch all aspects our present day lives. The present-day home products including electronic games and appliances, communication and office automation products, computers with a wide range of capabilities, and industrial instrumentation and control systems, electromedical equipment, and defense and aerospace systems are heavily dependent on digital circuits. Many fields that emerged later to digital electronics have peaked and levelled off, but the application of digital concepts appears to be still growing exponentially. This unprecedented growth is powered by the semiconductor technology, which enables the introduction of more and complex integrated circuits. The complexity of an integrated circuit is measured in terms of the number of transistors that can be integrated into a single unit. The number of transistors in a single integrated circuit has been doubling every eighteen months (Moore' Law) for several decades and reached the figure of almost one billion transistors per chip. This allowed the circuit designers to provide more and more complex functions in a single unit.

The introduction of programmable integrated circuits in the form of microprocessors in 70s completely transformed every facet of electronics. While fixed function integrated circuits and microprocessors coexisted for considerable time, the need to make the equipment smaller and portable lead to replacement of fixed function devices with programmable devices. With the all pervasive presence of the microprocessor and the increasing usage of other programmable circuits like PLDs (Programmable Logic devices), FPGAs (Field Programmable Gate Arrays) and ASICs (Application Specific Integrated Circuits), the very nature of digital systems is continuously changing.

The central role of digital circuits in all our professional and personal lives makes it imperative that every electrical and electronics engineer acquire good knowledge of relevant basic concepts and ability to work with digital circuits. At present many of the undergraduate programmers offer two to four courses in the area of digital systems, with at least two of them being core courses. The course under consideration constitutes the first course in the area of digital systems. The rate of obsolescence of knowledge, design methods, and design tools is uncomfortably high. Even the first level course in digital electronics is not exempt from this obsolescence.

Any course in electronics should enable the students to design circuits to meet some stated requirements as encountered in real life situations. However, the design approaches should be based on a sound understanding of the underlying principles. The basic feature of all design problems is that all of them admit multiple solutions. The selection of the final solution depends on a variety of criteria that could include the size and cost of the substrate on which the components are assembled, the cost of components, manufacturability, reliability, speed etc.

The course contents are designed to enable the students to design digital circuits of medium level of complexity taking the functional and hardware aspects in an integrated manner within the context of commercial and manufacturing constraints. However, no compromises are made with regard to theoretical aspects of the subject.

BLOCK I

Unit I: Number Systems

1.0 Learning Objectives

After the completion of this unit, you will be able to:

- Know Binary, octal and hexadecimal number systems.
- Perform conversion of number with one radix to another.
- Summarize the advantages of using different number systems.
- Interpret the arithmetic operations of binary numbers.
- Convert a given number from one system to an equivalent number in another system.

1.1 Introduction

We all use numbers to communicate and perform several tasks in our daily lives. Our present-day world is characterized by measurements and numbers associated with everything. In fact, many consider if we cannot express something in terms of numbers is not worth knowing. While this is an extreme view that is difficult to justify, there is no doubt that quantification and measurement, and consequently usage of numbers, are desirable whenever possible. Manipulation of numbers is one of the early skills that the present-day child is trained to acquire. The present-day technology and the way of life require the usage of several number systems. Usage of decimal numbers starts very early in one's life. Therefore, when one is confronted with number systems other than decimal, some time during the high-school years, it calls for a fundamental change in one's framework of thinking.

There have been two types of numbering systems in use throughout the world. One type is symbolic in nature. Most important example of this symbolic numbering system is the one based on Roman numerals

I = 1, V = 5, X = 10, L = 50, C = 100, D = 500 and M = 1000

While this system was in use for several centuries in Europe it is completely superseded by the weighted-position system based on Indian numerals. The Roman number system is still used in some places like watches and release dates of movies.

The weighted-positional system based on the use of radix 10 is the most commonly used numbering system in most of the transactions and activities of today's world. However, the advent of computers and the convenience of using devices that have two well defined states brought the binary system, using the radix 2, into extensive use. The use of binary number system in the field of computers and electronics also lead to the use of octal (based on radix 8) and hex-decimal system (based on radix16). The usage of binary numbers at various levels has become so essential that it is also necessary to have a good understanding of all the binary arithmetic operations. Here we explore the weighted-position number systems and conversion from one system to the other.

1.2 Weighted-Position Number System

In a weighted-position numbering system using Indian numerals the value associated with a digit is dependent on its position. The value of a number is weighted sum of its digits.

Consider the decimal number 2357. It can be expressed as

$2357 = 2 \times 10^3 + 3 \times 10^2 + 5 \times 10^1 + 7 \times 10^0$

Each weight is a power of 10 corresponding to the digit's position. A decimal point allows negative as well as positive powers of 10 to be used;

$526.47 = 5 \times 10^2 + 2 \times 10^1 + 6 \times 10^0 + 4 \times 10^{-1} + 7 \times 10^{-2}$

Here, 10 is called the *base* or *radix* of the number system. In a general positional number system, the *radix* may be any integer $r \geq 2$, and a digit position i has weight r^i. The general form of a number in such a system is

$d_{p-1}\ d_{p-2}, \ldots\ d_1, d_0\ .\ d_{-1}d_{-2}\ \ldots\ d_{-n}$

where there are *p* digits to the left of the point (called *radix point*) and *n* digits to the right of the point. The value of the number is the sum of each digit multiplied by the corresponding power of the *radix*.

$$D = \sum_{i=-n}^{p-1} d_i r^i$$

Except for possible leading and trailing zeros, the representation of a number in positional system is unique (00256.230 is the same as 256.23). Obviously, the values d_i's can take are limited by the radix value. For example, a number like $(356)_5$, where the suffix 5 represents the radix will be incorrect, as there can not be a digit like 5 or 6 in a weighted position number system with radix 5.

If the radix point is not shown in the number, then it is assumed to be located near the last right digit to its immediate right. The symbol used for the radix point is a point (.). However, a comma is used in some countries. For example, 7,6 is used, instead of 7.6, to represent a number having seven as its integer component and six as its fractional.

As much of the present-day electronic hardware is dependent on devices that work reliably in two well defined states, a numbering system using 2 as its radix has become necessary and popular. With the radix value of 2, the binary number system will have only two numerals, namely 0 and 1.

Consider the number $(N)_2 = (11100110)_2$.

It is an eight-digit binary number. The binary digits are also known as *bits*. Consequently, the above number would be referred to as an 8-bit number. Its decimal value is given by

$(N)_2 = 1 \times 2^7 + 1 \times 2^6 + 1 \times 2^5 + 0 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$

$= 128 + 64 + 32 + 0 + 0 + 4 + 2 + 0$

$= (230)_{10}$

Consider a binary fractional number $(N)_2 = 101.101$.

Its decimal value is given by

$(N)_2 = 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3}$

$= 4 + 0 + 1 + \frac{1}{2} + 0 + \frac{1}{8}$

$= 5 + 0.5 + 0.125 = (5.625)_{10}$

From here on we consider any number without its radix specifically mentioned, as a decimal number.

With the radix value of 2, the binary number system requires very long strings of 1s and 0s to represent a given number. Some of the problems associated with handling large strings of binary digits may be eased by grouping them into three digits or four digits. We can use the following groupings.

- Octal (radix 8 to group three binary digits)
- Hexadecimal (radix 16 to group four binary digits)

In the octal number system, the digits will have one of the following eight values 0, 1, 2, 3, 4, 5, 6 and 7. In the hexadecimal system we have one of the sixteen values 0 through 15. However, the decimal values from 10 to 15 will be represented by alphabet A (=10), B (=11), C (=12), D (=13), E (=14) and F (=15).

Conversion of a binary number to an octal number or a hexadecimal number is very simple, as it requires simple grouping of the binary digits into groups of three or four. Consider the binary number 11011011. It may be converted into octal or hexadecimal numbers as

$(11011001)_2 = (011)\ (011)\ (001) = (331)_8$

$= (1101)\ (1001) = (D9)_{16}$

Note that adding a leading zero does not alter the value of the number. Similarly, for grouping the digits in the fractional part of a binary number, trailing zeros may be added without changing the value of the number.

1.3 Number System Conversions

In general, conversion between numbers with different radices cannot be done by simple substitutions. Such conversions would involve arithmetic operations. Let us work out procedures for converting a number in any radix to radix 10, and vice versa. The decimal equivalent value of a number in any radix is given by the formula

$$D = \sum_{i=-n}^{p-1} d_i r^i$$

where *r* is the radix of the number and there are *p* digits to the left of the radix point and n digits to the right. Decimal value of the number is determined by converting each digit of the number to its radix-10 equivalent and expanding the formula using radix-10 arithmetic.

Some examples are:

$(331)_8 = 3 \times 8^2 + 3 \times 8^1 + 1 \times 8^0 = 192 + 24 + 1 = (217)_{10}$

$(D9)_{16} = 13 \times 16^1 + 9 \times 16^0 = 208 + 9 = (217)_{10}$

$(33.56)_8 = 3 \times 8^1 + 3 \times 8^0 + 5 \times 8^{-1} + 6 \times 8^{-2} = (27.69875)_{10}$

$(E5.A)_{16} = 14 \times 16^1 + 5 \times 16^0 + 10 \times 16^{-1} = (304.625)_{10}$

The conversion formula can be rewritten as

$D = ((\ldots((d_{n-1}).r + d_{n-2})\, r + \ldots.).r + d_1).r + d_0$

This forms the basis for converting a decimal number D to a number with radix r. If we divide the right-hand side of the above formula by r, the remainder will be d0, and the quotient will be

$Q = ((\ldots((d_{n-1}).r + d_{n-2})\, r + \ldots.).r + d_1$

Thus, d0 can be computed as the remainder of the long division of D by the radix r. As the quotient Q has the same form as D, another long division by r will give d_1 as the remainder. This process can continue to produce all the digits of the number with radix r. Consider the following examples.

Quotient	Remainder	
156 ÷ 2	78	0
78 ÷ 2	39	0
39 ÷ 2	19	1
19 ÷ 2	9	1

9 ÷ 2	4	1
4 ÷ 2	2	0
2 ÷ 2	1	0
1 ÷ 2	0	1

$(156)_{10} = (10011100)_2$

	Quotient	Remainder
678 ÷ 8	84	6
84 ÷ 8	10	4
10 ÷ 8	1	2
1 ÷ 8	0	1

$(678)_{10} = (1246)_8$

	Quotient	Remainder
678 ÷ 16	42	6
42 ÷ 16	2	A
2 ÷ 16	0	2

$(678)_{10} = (2A6)_{16}$

Check Your Progress I

Multiple Choice Questions

1. Which number system is understood easily by the computer?

(a) Binary (b) Decimal (c) Octal (d) Hexadecimal

2. How many symbols are used in the decimal number system?

(a) 2 (b) 8 (c) 10 (d) 16

3. How are number systems generally classified?

a. Conditional or non conditional

b. Positional or non positional

c. Real or imaginary

d. Literal or numerical

4. What does (10)16 represent in decimal number system?

(a) 10 (b) 0A (c) 16 (d) 15

1.4 Representation of Negative Numbers

In our traditional arithmetic we use the "+" sign before a number to indicate it as a positive number and a "-" sign to indicate it as a negative number. We usually omit the sign before the number if it is positive. This method of representation of numbers is called "sign-magnitude" representation. But using "+" and "-" signs on a computer is not convenient, and it becomes necessary to have some other convention to represent the signed numbers. We replace "+" sign with "0" and "-" with "1". These two symbols already exist in the binary system. Consider the following examples:

$(+1100101)_2 \rightarrow (01100101)_2$

$(+101.001)_2 \rightarrow (0101.001)_2$

$(-10010)_2 \rightarrow (110010)_2$

$(-110.101)_2 \rightarrow (1110.101)_2$

In the sign-magnitude representation of binary numbers the first digit is always treated separately. Therefore, in working with the signed binary numbers in sign-magnitude form the leading zeros should not be ignored. However, the leading zeros can be ignored after the sign bit is separated. For example,

1000101.11 = - 101.11

While the sign-magnitude representation of signed numbers appears to be natural extension of the traditional arithmetic, the arithmetic operations with signed numbers in this form are not that very convenient, either for implementation on the computer or for hardware implementation. There are two other methods of representing signed numbers.

- Diminished Radix Complement (DRC) or (r-1)-complement
- Radix Complement (RX) or r-complement

When the numbers are in binary form

- Diminished Radix Complement will be known as "one's-complement"

- Radix complement will be known as “two’s-complement”.

If this representation is extended to the decimal numbers they will be known as 9’scomplement and 10’s-complement respectively.

1.4.1 One’s Complement Representation

Let A be an n-bit signed binary number in one’s complement form. The most significant bit represents the sign. If it is a “0” the number is positive and if it is a “1” the number is negative. The remaining (n-1) bits represent the magnitude, but not necessarily as a simple weighted number.

Consider the following one’s complement numbers and their decimal equivalents:

0111111		+ 63
0000110	-->	+ 6
0000000	-->	+ 0
1111111	-->	+ 0
1111001	-->	- 6
1000000	-->	- 63

There are two representations of “0”, namely 000....0 and 111....1. From these illustrations we observe

- If the most significant bit (MSD) is zero the remaining (n-1) bits directly indicate the magnitude.
- If the MSD is 1, the magnitude of the number is obtained by taking the complement of all the remaining (n-1) bits.

For example, consider one’s complement representation of -6 as given above.

- Leaving the first bit ‘1’ for the sign, the remaining bits 111001 do not directly represent the magnitude of the number -6.
- Take the complement of 111001, which becomes 000110 to determine the magnitude.

In the example shown above a 7-bit number can cover the range from +63 to -63. In general, an n-bit number has a range from $+(2^{n-1} - 1)$ to $-(2^{n-1} - 1)$ with two representations for zero. The representation also suggests that if A is an integer in one's complement form, then

one's complement of A = -A

One's complement of a number is obtained by merely complementing all the digits.

This relationship can be extended to fractions as well.

For example, if A = 0.101 $(+0.625)_{10}$, then the one's complement of A is 1.010, which is one's complement representation of $(-0.625)_{10}$. Similarly consider the case of a mixed number.

A = 010011.0101 $(+19.3125)_{10}$

One's complement of A = 101100.1010 $(-19.3125)_{10}$

This relationship can be used to determine one's complement representation of negative decimal numbers.

Example 1: What is one's complement binary representation of decimal number -75?

Decimal number 75 requires 7 bits to represent its magnitude in the binary form. One additional bit is needed to represent the sign. Therefore, one's complement representation of 75 = 01001011 one's complement representation of -75 = 10110100

1.4.2 Two's Complement Representation

Let A be an n-bit signed binary number in two's complement form.

- The most significant bit represents the sign. If it is a "0", the number is positive, and if it is "1" the number is negative.
- The remaining (n-1) bits represent the magnitude, but not as a simple weighted number.

Consider the following two's complement numbers and their decimal equivalents:

0111111	→	+ 63
0000110	→	+ 6
0000000	→	+ 0
1111010	→	- 6
1000001	→	- 63
1000000	→	- 64

There is only one representation of "0", namely 000....0.

From these illustrations we observe:

If most significant bit (MSD) is zero the remaining (n-1) bits directly indicate the magnitude.

If the MSD is 1, the magnitude of the number is obtained by taking the complement of all the remaining (n-1) bits and adding a 1.

Consider the two's complement representation of -6.

- We assume we are representing it as a 7-bit number.
- Leave the sign bit.
- The remaining bits are 111010. These have to be complemented (that is 000101) and a 1 has to be added (that is 000101 + 1 = 000110 = 6).

In the example shown above a 7-bit number can cover the range from +63 to -64. In general, an n-bit number has a range from + (2^{n-1} - 1) to - (2^{n-1}) with one representation for zero. The representation also suggests that if A is an integer in two's complement form, then

Two's complement of A = -A

Two's complement of a number is obtained by complementing all the digits and adding '1' to the LSB.

This relationship can be extended to fractions as well. If A = 0.101 $(+0.625)_{10}$, then the two's complement of A is 1.011, which is two's complement representation of $(-0.625)_{10}$.

Similarly consider the case of a mixed number.

A = 010011.0101 $(+19.3125)_{10}$

Two's complement of A = 101100.1011 $(-19.3125)_{10}$

This relationship can be used to determine two's complement representation of negative decimal numbers.

Example 2: What is two's complement binary representation of decimal number -75?

Decimal number 75 requires 7 bits to represent its magnitude in the binary form. One additional bit is needed to represent the sign. Therefore,

Two's complement representation of 75 = 01001011

Two's complement representation of -75 = 10110101

Check Your progress II

1. How many bits have to be grouped together to convert the binary number to its corresponding octal number?

(a) 2 (b) 3 (c) 4 (d) 5

2. Which bit represents the sign bit in a signed number system?

a. Left most bit

b. Right most bit

c. Left centre

d. Right centre

3. The ones complement of 1010 is

(a) 1100 (b) 0101 (c) 0111 (d) 1011

4. How many bits are required to cover the numbers from +63 to -63 in one’s complement representation?

(a) 6 (b) 7 (c) 8 (d) 9

1.6 Model Question

1. Perform the following number system conversions:

(a) $10110111_2 = ?_{10}$	(b) $5674_{10} = ?_2$
(c) $10011100_2 = ?_8$	(d) $2453_8 = ?_2$
(e) $111100010_2 = ?_{16}$	(f) $68934_{10} = ?_2$
(g) $10101.001_2 = ?_{10}$	(h) $6FAB7_{16} = ?_{10}$
(i) $11101.101_2 = ?_8$	(j) $56238_{16} = ?_2$

2. Convert the following hexadecimal numbers into binary and octal numbers

(a) 78AD	(b) DA643	(c) EDC8
(d) 3245	(e) 68912	(f) AF4D

3. Convert the following octal numbers into binary and hexadecimal numbers

(a) 7643	(b) 2643	(c) 1034
(d) 3245	(e) 6712	(f) 7512

4. Convert the following numbers into binary:

(a) 1236_{10}	(b) 2349_{10}	(c) 345.275_{10}

(d) 4567_8 (e) 45.65_8 (f) 145.23_8

(g) $ADF5_{16}$ (h) $AD.F3_{16}$ (i) $12.DA_{16}$

5. What is the range of unsigned decimal values that can be represented by 8 bits?

6. What is the range of signed decimal values that can be represented by 8 bits?

7. How many bits are required to represent decimal values ranging from 75 to -75?

8. Represent each of the following values as a 6-bit signed binary number in one's complement and two's complement forms.

(a) 28 (b) -21 (c) -5 (d) -13

9. Determine the decimal equivalent of two's complement numbers given below:

(a) 1010101 (b) 0111011 (c) 11100010

UNIT II: CODES

2.0 Learning Objectives

After the completion of this unit, you will be able to:

- Know various binary codes
- Explain the usefulness of different coding schemes.
- Explain how errors are detected and/or corrected using different codes.
- Illustrate the construction of a weighted code.

2.1 Introduction

When we wish to send information over long distances unambiguously it becomes necessary to modify (encoding) the information into some form before sending, and convert (decode) at the receiving end to get back the original information. This process of encoding and decoding is necessary because the channel through which the information is sent may distort the transmitted information. Much of the information is sent as numbers. While these numbers are created using simple weighted-positional numbering systems, they need to be encoded before transmission. The modifications to numbers were based on changing the weights, but predominantly on some form of binary encoding. There are several codes in use in the context of present day information technology, and more and more new codes are being generated to meet the new demands.

Coding is the process of altering the characteristics of information to make it more suitable for intended application

By assigning each item of information a unique combination of 1s and 0s we transform some given information into binary coded form. The bit combinations are referred to as "words" or "code words". In the field of digital systems and computers different bit combinations have different designations.

Bit - a binary digit 0 or 1

Nibble - a group of four bits

Byte - a group of eight bits

Word - a group of sixteen bits;

a word has two bytes or four nibbles

Sometimes ‘word’ is used to designate a larger group of bits also, for example 32 bit or 64 bit words.

We need and use coding of information for a variety of reasons

- to increase efficiency of transmission,
- to make it error free,
- to enable us to correct it if errors occurred,
- to inform the sender if an error occurred in the received information etc.
- for security reasons to limit the accessibility of information
- to standardize a universal code that can be used by all

Coding schemes have to be designed to suit the security requirements and the complexity of the medium over which information is transmitted.

Decoding is the process of reconstructing source information from the encoded information.

Decoding process can be more complex than coding if we do not have prior knowledge of coding schemes. In view of the modern day requirements of efficient, error free and secure information transmission coding theory is an extremely important subject. However, at this stage of learning digital systems we confine ourselves to familiarising with a few commonly used codes and their properties.

We will be mainly concerned with binary codes. In binary coding we use binary digits or bits (0 and 1) to code the elements of an information set. Let n be the number of bits in the code word and x be the number of unique words.

If $n = 1$, then $x = 2$ (0, 1)

$n = 2$, then $x = 4$ (00, 01, 10, 11)

$n = 3$, then $x = 8$ (000,001,010 ...111)

$n = j$, then $x = 2j$

From this we can conclude that if we are given elements of information to code into binary coded format,

$x \leq 2j$

or $j \geq \log_2 x$

$\geq 3.32 \log_{10} x$

where j is the number of bits in a code word.

For example, if we want to code alphanumeric information (26 alphabetic characters + 10 decimals digits = 36 elements of information), we require

$j \geq 3.32 \log_{10} 36$

$j \geq 5.16$ bits

Since bits are not defined as fractional parts, we take j = 6. In other words a minimum six-bit code would be required to code 36 alphanumeric elements of information. However, with a six-bit code only 36 code words are used out of the 64 code words possible.

In this Learning Unit we consider a few commonly used codes including

1 Binary coded decimal codes
2 Unit distance codes
3 Error detection codes
4 Alphanumeric codes

2.2 Binary Coded Decimal Codes

The main motivation for binary number system is that there are only two elements in the binary set, namely 0 and 1. While it is advantageous to perform all computations on hardware in binary forms, human beings still prefer to work with decimal numbers. Any electronic system should then be able to accept decimal numbers, and make its output available in the decimal form.

One method, therefore, would be to

- convert decimal number inputs into binary form
- manipulate these binary numbers as per the required functions, and
- convert the resultant binary numbers into the decimal form

However, this kind of conversion requires more hardware, and in some cases considerably slows down the system. Faster systems can afford the additional circuitry, but the delays associated with the conversions would not be acceptable. In case of smaller systems, the speed may not be the main criterion, but the additional circuitry may make the system more expensive.

We can solve this problem by encoding decimal numbers as binary strings, and use them for subsequent manipulations. There are ten different symbols in the decimal number system: 0, 1, 2, . . ., 9. As there are ten symbols we require at least four bits to represent them in the binary form. Such a representation of decimal numbers is called **binary coding of decimal numbers**.

As four bits are required to encode one decimal digit, there are sixteen four-bit groups to select ten groups. This would lead to nearly 30×10^{10} (${}^{16}C_{10}.10!$) possible codes. However, most of them will not have any special properties that would be useful in hardware design. We wish to choose codes that have some desirable properties like

- ease of coding
- ease in arithmetic operations
- minimum use of hardware
- error detection property
- ability to prevent wrong output during transitions

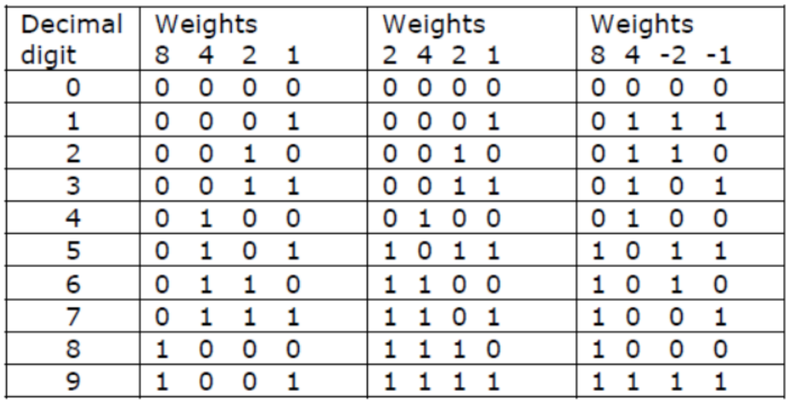
In a **weighted code** the decimal value of a code is the algebraic sum of the weights of 1s appearing in the number. Let $(A)_{10}$ be a decimal number encoded in the binary form as $a_3a_2a_1a_0$. Then

$(A)_{10} = w_3a_3 + w_2a_2 + w_1a_1 + w_0a_0$

where w_3, w_2, w_1 and w_0 are the weights selected for a given code, and a_3,a_2,$a_1$and a_0 are either 0s or 1s. The more popularly used codes have the weights as

w_3	w_2	w_1	w_0
8	4	2	1
2	4	2	1
8	4	-2	-1

The decimal numbers in these three codes are



Decimal digit	Weights 8 4 2 1	Weights 2 4 2 1	Weights 8 4 -2 -1
0	0 0 0 0	0 0 0 0	0 0 0 0
1	0 0 0 1	0 0 0 1	0 1 1 1
2	0 0 1 0	0 0 1 0	0 1 1 0
3	0 0 1 1	0 0 1 1	0 1 0 1
4	0 1 0 0	0 1 0 0	0 1 0 0
5	0 1 0 1	1 0 1 1	1 0 1 1
6	0 1 1 0	1 1 0 0	1 0 1 0
7	0 1 1 1	1 1 0 1	1 0 0 1
8	1 0 0 0	1 1 1 0	1 0 0 0
9	1 0 0 1	1 1 1 1	1 1 1 1

In all the cases only ten combinations are utilized to represent the decimal digits. The remaining six combinations are illegal. However, they may be utilized for error detection purposes. Consider, for example, the representation of the decimal number 16.85 in Natural Binary Coded Decimal code (NBCD)

$(16.85)_{10} = (\underline{0001}\ \underline{0110}\,.\,\underline{1000}\ \underline{0101})_{NBCD}$

1 6 8 5

There are many possible weights to write a number in BCD code. Some codes have desirable properties, which make them suitable for specific applications. Two such desirable properties are:

1 Self-complementing codes

2 Reflective codes

When we perform arithmetic operations, it is often required to take the “complement” of a given number. If the logical complement of a coded number is also its arithmetic complement, it will be convenient from hardware point of view. In a **self-complementing coded** decimal number, $(A)_{10}$, if the individual bits of a number are complemented it will result in $(9 - A)_{10}$.

Example: Consider the 2421 code.

- The 2421 code of $(4)_{10}$ is 0100.
- Its complement is 1011 which is 2421 code for $(5)_{10} = (9 - 4)_{10}$.

Therefore, 2421 code may be considered as a self-complementing code. A necessary condition for a self-complimenting code is that the sum of its weights should be 9. A self-complementing code, which is not weighted, is excess-3 code. It is derived from

8421 code by adding 0011 to all the 8421 coded numbers. Another self-complementing code is 631-1 weighted code. Three self-complementing codes are:

Decimal Digit	Excess-3 Code	631-1 Code	2421 Code
0	0011	0011	0000
1	0100	0010	0001
2	0101	0101	0010
3	0110	0111	0011
4	0111	0110	0100
5	1000	1001	1011
6	1001	1000	1100
7	1010	1010	1101
8	1011	1101	1110
9	1100	1100	1111

A **reflective code** is characterized by the fact that it is imaged about the centre entries with one bit changed. For example, the 9's complement of a reflected BCD code word is formed by changing only one its bits. Two such examples of reflective BCD codes are:

Decimal	Code-A	Code-B
0	0000	0100
1	0001	1010
2	0010	1000
3	0011	1110
4	0100	0000
5	1100	0001
6	1011	1111
7	1010	1001
8	1001	1011
9	1000	0101

The BCD codes are widely used and the reader should become familiar with reasons for using them and their application. The most common application of NBCD codes is in the calculator.

2.3 Unit Distance Codes

There are many applications in which it is desirable to have a code in which the adjacent codes differ only in one bit. Such codes are called Unit distance Codes.

“Gray code” is the most popular example of unit distance code. The 3-bit and 4-bitGray codes are:

Decimal	3-bit Gray	4-bit Gray
0	000	0000
1	001	0001
2	011	0011
3	010	0010
4	110	0110
5	111	0111
6	101	0101
7	100	0100
8	-	1100
9	-	1101
10	-	1111
11	-	1110
12	-	1010
13	-	1011
14	-	1001
15	-	1000

These Gray codes listed here have also the reflective properties. Some additional examples of unit distance codes are:

Decimal Digit	UDC-1	UDC-2	UDC-3
0	0000	0000	0000
1	0100	0001	1000
2	1100	0011	1001
3	1000	0010	0001
4	1001	0110	0011
5	1011	1110	0111
6	1111	1111	1111
7	0111	1101	1011
8	0011	1100	1010
9	0001	0100	0010

The most popular use of Gray codes is in the position sensing transducer known as shaft encoder. A shaft encoder consists of a disk in which concentric circles have alternate sectors with reflective surfaces while the other sectors have non-reflective surfaces. The position is sensed by the reflected light from a light emitting diode. However, there is choice in arranging the reflective and non-reflective sectors. A 3- bit binary coded disk will be as shown in the figure 1.

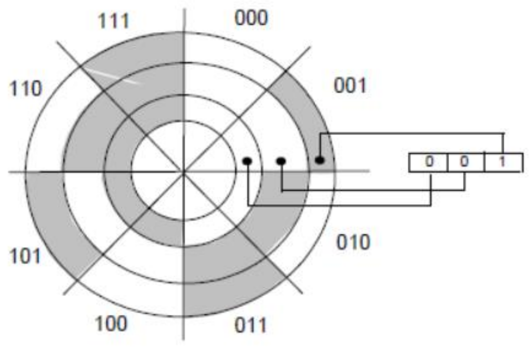


FIG.1: 3-bit binary coded shaft encoder

From this figure we see that straight binary code can lead to errors because of mechanical imperfections. When the code is transiting from 001 to 010, a slight misalignment can cause a transient code of 011 to appear. The electronic circuitry associated with the encoder will receive 001 --> 011 -> 010. If the disk is patterned to give Gray code output, the possibilities of wrong transient codes will not arise.

This is because the adjacent codes will differ in only one bit. For example the adjacent code for 001 is 011. Even if there is a mechanical imperfection, the transient code will be either 001 or 011. The shaft encoder using 3-bit Gray code is shown in the figure 2.

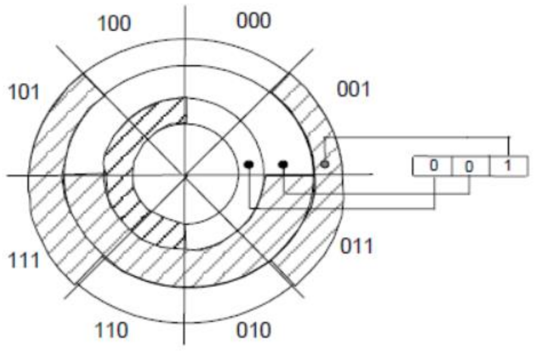


FIG. 2: Shaft encoder disk using a 3-bit Gray code

There are two convenient methods to construct Gray code with any number of desired bits. The first method is based on the fact that Gray code is also a reflective code. The following rule may be used to construct Gray code:

- A one-bit Gray code had code words, 0 and 1

- The first 2n code words of an (n+1)-bit Gray code equal the code words of an n-bit Gray code, written in order with a leading 0 appended.
- The last 2n code words of a (n+1)-bit Gray code equal the code words of an n-bit Gray code, written in reverse order with a leading 1 appended.

However, this method requires Gray codes with all bit lengths less than ‘n’ also be generated as a part of generating n-bit Gray code. The second method allows us to derive an n-bit Gray code word directly from the corresponding n-bit binary code word:

- The bits of an n-bit binary code or Gray code words are numbered from right to left, from 0 to n-1.
- Bit i of a Gray-code word is 0 if bits i and i+1 of the corresponding binary code word are the same, else bit i is 1. When i+1 = n, bit n of the binary code word is considered to be 0.

Example: Consider the decimal number 68.

$(68)_{10} = (1000100)_2$

Binary code: 1 0 0 0 1 0 0

Gray code : 1 1 0 0 1 1 0

The following rules can be followed to convert a Gray coded number to a straight binary number:

- Scan the Gray code word from left to right. All the bits of the binary code are the same as those of the Gray code until the first 1 is encountered, including the first 1.
- 1’s are written until the next 1 is encountered, in which case a 0 is written.
- 0’s are written until the next 1 is encountered, in which case a 1 is written.

Consider the following examples of Gray code numbers converted to binary numbers

Gray code :	1 1 0 1 1 0	1 0 0 0 1 0 1 1
Binary code:	1 0 0 1 0 0	1 1 1 1 0 0 1 0

2.4 Alphanumeric Codes

When information to be encoded includes entities other than numerical values, an expanded code is required. For example, alphabetic characters (A, B, ...Z) and special operation symbols like +, -, /, *, (,) and other special symbols are used in digital systems. Codes that include alphabetic characters are commonly referred to as Alphanumeric Codes. However, we require adequate number of bits to encode all the characters. As there was a need for alphanumeric codes in a wide variety of applications in the early era of computers, like teletype, punched tape and punched cards, there has always been a need for evolving a standard for these codes.

Alphanumeric keyboard has become ubiquitous with the popularization of personal computers and notebook computers. These keyboards use ASCII (American Standard Code for Information Interchange) code:

b4	b3	b2	b1	b7 b6 b5							
				000	001	010	011	100	101	110	111
0	0	0	0	NUL	DLE	SP	0	@	P	`	p
0	0	0	1	SOH	DC1	!	1	A	Q	a	q
0	0	1	0	STX	DC2	"	2	B	R	b	r
0	0	1	1	ETX	DC3	#	3	C	S	c	s
0	1	0	0	EOT	DC4	$	4	D	T	d	t
0	1	0	1	ENQ	NAK	%	5	E	U	e	u
0	1	1	0	ACK	SYN	&	6	F	V	f	v
0	1	1	1	BEL	ETB	,	7	G	W	g	w
1	0	0	0	BS	CAN	(	8	H	X	h	x
1	0	0	1	HT	EM	)	9	I	Y	i	y
1	0	1	0	LF	SUB	*	:	J	Z	j	z
1	0	1	1	VT	ESC	+	;	K	[	k	{
1	1	0	0	FF	FS	,	<	L	\	l	|
1	1	0	1	CR	GS	-	=	M	]	m	}
1	1	1	0	SO	RS	.	>	N		n	~
1	1	1	1	SI	US	/	?	O	-	o	DEL

Alphanumeric codes like EBCDIC (Extended Binary Coded Decimal Interchange Code) and 12-bit Hollerith code are in use for some applications. However, ASCII code is now the standard code for most data communication networks. Therefore, the reader is urged to become familiar with the ASCII code.

2.5 Error Detection and Correcting Codes

When data is transmitted in digital form from one place to another through a transmission channel/medium, some data bits may be lost or modified. This loss of data integrity occurs due to a variety of electrical phenomena in the transmission channel. As there are needs to transmit millions of bits per second, the data integrity should be very high. The error rate cannot be reduced to zero. Then we would like to ideally have a mechanism of correcting the errors that occur. If this is not possible or proves to be expensive, we would like to know if an error occurred.

If an occurrence of error is known, appropriate action, like retransmitting the data, can be taken. One of the methods of improving data integrity is to encode the data in a suitable manner. This encoding may be done for error correction or merely for error detection.

A simple process of adding a special code bit to a data word can improve its integrity. This extra bit will allow detection of a single error in a given code word in which it is used, and is called the 'Parity Bit'. This parity bit can be added on an odd or even basis. The odd or even designation of a code word may be determined by actual number of 1's in the data (including the added parity bit) to which the parity bit is added. For example, the S in ASCII code is

$$(S) = (1010011)_{\text{ASCII}}$$

S, when coded for odd parity, would be shown as

$$(S) = (11010011)_{\text{ASCII with odd parity}}$$

In this encoded 'S' the number of 1's is five, which is odd.

When S is encoded for even parity

$$(S) = (01010011)_{\text{ASCII with even parity}}.$$

In this case the coded word has even number (four) of ones. Thus, the parity encoding scheme is a simple one and requires only one extra bit. If the system is using even parity and we find odd number of ones in the received data word we know that an error has occurred. However, this scheme is meaningful only for single errors. If two bits in a data word were received incorrectly the parity bit scheme will not detect the faults. Then the question arises as to the level of improvement in the data integrity if occurrence of only one-bit error is detectable.

The improvement in the reliability can be mathematically determined. Adding a parity bit allows us only to detect the presence of one-bit error in a group of bits. But it does not enable us to exactly locate the bit that changed. Therefore, addition of one parity bit may be called an error detecting coding scheme. In a digital system detection of error alone is not sufficient. It has to be corrected as well. Parity bit scheme can be extended to locate the faulty bit in a block of information. The information bits are conceptually arranged in a two-dimensional array, and parity bits are provided to check both the rows and the columns.

If we can identify the code word that has an error with the parity bit, and the column in which that error occurs by a way of change in the column parity bit, we can both detect and correct the wrong bit of information. Hence such a scheme is single error detecting and single error correcting coding scheme.

This method of using parity bits can be generalized for detecting and correcting more than one-bit error. Such codes are called parity-check block codes. In this class known as (n, k) codes, r (= n-k) parity check bits, formed by linear operations on the k data bits, are appended to each block of k bits to generate an n-bit code word.

An encoder outputs a unique n-bit code word for each of the 2^k possible input k-bit blocks. For example a (15, 11) code has r = 4 parity-check bits for every 11 data bits. As r increases it should be possible to correct more and more errors. With r = 1 error correction is not possible, as such a code will only detect an odd number of errors.

It can also be established that as k increases the overall probability of error should also decrease. Long codes with a relatively large number of parity-check bits should thus provide better performance. Consider the case of (7, 3) code:

Data bits	Code words
0 0 0	0 0 0 0 0 0 0
0 0 1	0 0 1 1 1 1 1
0 1 0	0 1 0 0 1 1 0
0 1 1	0 1 1 1 0 0 1
1 0 0	1 0 0 1 1 0 0
1 0 1	1 0 1 0 0 1 1
1 1 0	1 1 0 1 0 1 0
1 1 1	1 1 1 0 1 0 1

A close look at these indicates that they differ in at least three positions. Any *one* error should then be correctable since the resultant code word will still be closer to the correct one, in the sense of the number of bit positions in which they agree, than to any other. This is an example of *single-error-correcting-code*. The difference in the number of positions between any two code words is called the *Hamming distance*, named after R.W. Hamming who, in 1950, described a general method for constructing codes with a minimum distance of 3. The Hamming distance plays a key role in assessing the error-correcting capability of codes. For two errors to be correctable, the Hamming distance d should be at least 5. In general, for **t** errors to be correctable, $\mathbf{d} \geq 2\mathbf{t}+1$ or $\mathbf{t} = [(\mathbf{d}-1)/2]$, where the [x] notation refers to the integer less than or equal to x.

Innumerable varieties of codes exist, with different properties. There are various types of codes for correcting independently occurring errors, for correcting burst errors, for providing relatively error-free synchronization of binary data etc. The theory of these codes, methods of generating the codes and decoding the coded data, is a very important subject of communication systems, and need to be studied as a separate discipline.

Check Your Progress

1. _____ is a group of four bits.
2. _____ is a group of sixteen bits.

3. ____________ is the process of reconstructing source information from the encoded information.
4. In a ___________ code the decimal value of a code is the algebraic sum of the weights of 1s appearing in the number.
5. A ______ code is characterized by the fact that it is imaged about the centre entries with one bit changed.

2.5 Model Questions

1. Write the following decimal number in Excess-3, 2421, 84-2-2 BCD codes:

(a) 563 (b) 678 (c) 1465

2. What is the use of self-complementing property? Demonstrate 631-1 BCD code is self-complementary.

3. Develop two different 4-bit unit distance codes.

4. Prove that Gray code is both a reflective and unit distance code?

5. Determine the Gray code for (a) 3710 and (b) 9710.

6. Write your address in ASCII code.

7. Write 8-bit ASCII code sequence of the name of your town/city with even parity.

8. (a) Write the following statements in ASCII

A = 4.5 x B

X = 75/Y

(b) Attach an even parity bit to each code word of the ASCII strings written for the above statements

9. Find and correct the error in the following code sequence

0 1 0 1 0

0 1 1 0 0

1 1 0 1 1

1 0 1 1 0

1 0 0 0 1

0 0 0 1 1

1 1 0 0 0

0 1 0 0 1

0 1 0 1 0 --- Parity word

|__________ Parity bit

9. Why we use coding of information? Explain.

10. What is decoding? Explain.

11. What is binary coding of decimal numbers?

12. What are the desirable properties of codes?

13. What is a refractive code?

14. Explain unit distance codes.

Answers to Check Your Progress

1. Nibble
2. Word
3. Decoding
4. weighted
5. reflective

UNIT III: BOOLEAN ALGEBRA

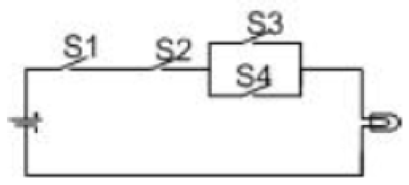
3.0 Learning Objectives

After the completion of this unit, you will be able to:

- Explain Boolean algebra and Boolean operators
- Perform Logic Functions
- Perform Minimization of logic functions using Karnaugh –map
- Implement Quine-McClausky method of minimization of logic functions

3.1 What is Boolean Algebra?

Consider the electrical circuit that controls the lighting of a bulb. Four switches control the operation of the bulb.



The manner in which the operation of the bulb is controlled can be stated as:

The bulb switches on if the switches S1 and S2 are closed, and S3 or S4 is also closed, otherwise the bulb will not switch on.

From this statement one can make the following observations:

• Any switch has two states: “closed” or “open”

• The bulb is switched on only when the switches are in some well-defined combination of states.

• The possible combinations are expressed through two types of relationships: “and” and “or”.

• The two possible combinations are

“S1 and S2 and S3 are closed”

“S1 and S2 and S4 are closed”

There are many situations of engineering interest where the variables take only a small number of possible values.
Some examples:

- Relay network used in telephone exchanges of earlier era
- Testing through multiple choice questions
- Mechanical display boards in airports and railway stations
- Choices available at road junctions.

Can you identify a situation of significance where the variables can take only a small number of distinctly defined states?
How do we implement functions similar to the example shown above? We need devices that have finite number states. It seems to be easy to create devices with two well defined states. It is more difficult and more expensive to create devices with more than two states.

Let us consider devices with two well defined states. We should also have the ability to switch the state of the device from one state to the other. We call devices having two well defined states as “two-valued switching devices”.
Some examples of devices with two states

- A bipolar transistor in either fully-off or fully-on state
- A MOS transistor in either fully off or fully on state
- Simple relays
- Electromechanical switch

If we learn to work with two-valued variables, we acquire the ability to implement functions of such variables using two-state devices. We call them “binary variables”. Very complex functions can be represented using several binary variables. As we can also build systems using millions of electronic two-state devices at very low costs, the mathematics of binary variables becomes very important.

An English mathematician, George Boole, introduced the idea of examining the truth or falsehood of language statements through a special algebra of logic. His work was published in 1854, in a book entitled “An Investigation of the Laws of Thought”. Boole's algebra was applied to statements that are either completely correct or completely false.

A value 1 is assigned to those statements that are completely correct and a value 0 is assigned to statements that are completely false. As these statements are given numerical values 1 or 0, they are referred to as digital variables. In our study of digital systems, we use the words switching variables, logical variables, and digital variables interchangeably.

Boole's algebra is referred to as Boolean algebra. Originally Boolean algebra was mainly applied to establish the **validity** or **falsehood** of logical statements. In 1938, Claude Shannon of Department of Electrical Engineering at Massachusetts Institute of Technology in (his master's thesis) provided the first applications of the principles of Boolean algebra to the design of electrical switching circuits. The title of the paper, which was an abstract of his thesis, is “A Symbolic Analysis of Relay and Switching Circuits”. Shannon established Boole's algebra to switching circuits is what ordinary algebra is to analogue circuits.

Logic designers of today use Boolean algebra to functionally design a large variety of electronic equipment such as

- hand-held calculators,
- traffic light controllers,
- personal computers,
- super computers,
- communication systems
- aerospace equipment
- etc.

We next explore Boolean algebra at the axiomatic level. However, we do not worry about the devices that would be used to implement them and their limitations.

3.2 Boolean Algebra and Huntington Postulates

Any branch of mathematics starts with a set of self-evident statements known as postulates, axioms or maxims. These are stated without any proof. Boolean algebra is a specific instance of Algebra of Propositional Logic. E.V. Huntington presented basic postulates of Boolean Algebra in 1904 in his paper "Sets of Independent Postulates for the Algebra of Logic". He defined a multi-valued Boolean algebra on a set of finite number of elements.

In Boolean algebra as applied to the switching circuits, all variables and relations are two-valued. The two values are normally chosen as 0 and 1, with 0 representing *false* and 1 representing *true*. If x is a Boolean variable, then

x = 1 means x is true

x = 0 means x is false

When we apply Boolean algebra to digital circuits we will find that the qualifications "*asserted"* and "*not-asserted"* are better names than "true" and "false". That is when x = 1 we say x is asserted, and when x = 0 we say x is not-asserted.

You are expected to be familiar with

- Concept of a set
- Meaning of equivalence relation
- The principle of substitution

Definition: A Boolean algebra consists of a finite set of elements BS subject to

- Equivalence relation "=",
- One unary operator "not" (symbolised by an over bar),
- Two binary operators "." and "+",
- For every element x and y ∈ BS the operations $\bar{x}$(not x), x.y and x +y are uniquely defined.

The unary operator '*not'* is defined by the relation

$\bar{1}$= 0; $\bar{0}$ = 1

The *not* operator is also called the complement, and consequently $\bar{x}$ is the complement of x. The binary operator '*and'* is symbolized by a dot. The '*and'* operator is defined by the relations:

0 . 0 = 0

0 . 1 = 0

1 . 0 = 0

1 . 1 = 1

The binary operator 'or' is represented by a plus (+) sign. The 'or' operator is defined by the relations

0 + 0 = 0

0 + 1 = 1

1 + 0 = 1

1 + 1 = 1

Huntington's postulates apply to the Boolean operations

P1. The operations are closed.

For all x and y ∈ BS,

a. x + y ∈ BS
b. x . y ∈ BS

P2. For each operation there exists an identity element.

a. There exists an element 0 ∈ BS such that for all x ∈ BS, x + 0 = x
b. There exists an element 1 ∈ BS such that for all x ∈ BS, x . 1 = x

P3. The operations are commutative.

For all x and y ∈ BS,

a. x + y = y + x
b. x . y = y . x

P4. The operations are distributive.

For all x, y and z ∈ BS,

a. x + (y . z) = (x + y) . (x + z)
b. x . (y + z) = (x . y) + (x . z)

P5. For every element x ∈ BS there exists an element $\bar{x}$ ∈ BS (called the complement of x) such that x + $\bar{x}$ = 1 and x . $\bar{x}$ = 0

P6. There exist at least two elements x and y $\in$ BS such that $x \neq y$.

3.2.1 Propositions from Huntington's Postulates

We derive several new propositions using the basic Huntington's postulates. Through these propositions we will be able to explore the structures and implications of that branch of mathematics. Such propositions are called theorems. A theorem gives a relationship among the variables.

Definition: A Boolean expression is a constant, 1 or 0, a single Boolean variable or its complement, or several constants and/or Boolean variables and/or their complements used in combination with one or more binary operators.

According to this definition 0, 1, x and $\bar{x}$ are Boolean expressions. If A and B are Boolean expressions, then $\bar{A}$, $\bar{B}$, A+B and A.B are also Boolean expressions.

Duality: Many of the Huntington's postulates are given as pairs, and differ only by the simultaneous interchange of operators "+" and "." and the elements "0" and "1".

This special property is called duality.

The property of duality can be utilized effectively to establish many useful properties of Boolean algebra.vThe duality principle:

"If two expressions can be proven equivalent by applying a sequence of basic postulates, then the dual expressions can be proven equivalent by simply applying the sequence of dual postulates"

This implies that for each Boolean property, which we establish, the dual property is also valid without needing additional proof.

Let us derive some useful properties:

Property 1: Special law of 0 and 1

For all x $\in$ BS,

a. $x . 0 = 0$

b. $x + 1 = 1$

Proof: $x . 0 = (x . 0) + 0$ (postulate 2a)

$= (x . 0) + (x . \bar{x})$ (postulate 5b)

$= x . (0 + \bar{x})$ (postulate 4b)

$= x . \bar{x}$ (postulate 2a)

= 0 (postulate 5b)

Property: **b** can be proved by applying the law of duality, that is, by interchanging "." and "+", and "1" and "0".

Property 2:

a. The element 0 is unique.
b. The element 1 is unique.

Proof for Part b by contradiction: Let us assume that there are two 1s denoted 1_1 and 1_2. Postulate 2b states that

$x. 1_1 = x$ and $y. 1_2 = y$

Applying the postulate 3b on commutativity to the second relationship, we get

$1_1 . x = x$ and $1_2 . y = y$

Letting $x = 1_2$ and $y = 1_1$, we obtain

$1_1 . 1_2 = 1_2$ and $1_2 . 1_1 = 1_1$

Using the transitivity property of any equivalence relationship we obtain $1_1 = 1_2$, which becomes a contradiction of our initial assumption.

Property **a** can be established by applying the principle of duality.

Property 3

a. The complement of 0 is $\bar{0} = 1$.
b. The complement of 1 is $\bar{1} = 0$.

Proof: $x + 0 = x$ (postulate 2a)

$0 + \bar{0} = 0$

$0 + \bar{0} = 1$ (postulate 5a)

$\bar{0} = 1$

Part b is valid by the application of principle of duality.

Property 4: Idempotency law

For all $x \in BS$,

a. $x + x = x$
b. $x . x = x$

Proof: $x + x = (x + x) . 1$ (postulate 2b)

$= (x + x) . (x + \bar{x})$ (postulate 5a)

$= x + (x . \bar{x})$ (postulate 4a)

= x + 0 (postulate 5b)

= x (postulate 2a)

x . x = x (by duality)

Property 5: Adjacency law

For all x and y ∈ BS,

a. x . y + x . $\bar{y}$ = x
b. (x + y) . (x + $\bar{y}$) = x

Proof: x . y + x . $\bar{y}$ = x . (y + $\bar{y}$) (postulate 4b)

= x . 1 (postulate 5a)

= x (postulate 2b)

(x + y) . (x + $\bar{y}$) = x (by duality)

The adjacency law is very useful in simplifying logical expressions encountered in the design of digital circuits. This property will be extensively used in later learning units.

Property 6: First law of absorption

For all x and y ∈ BS,

a. x + (x . y) = x
b. x . (x + y) = x

Proof x . (x + y) = (x + 0) . (x + y) (postulate 2a)

= x + (0 . y) (postulate 4a)

= x + 0 (property 2.1a)

= x (postulate 2a)

x + (x . y) = x (by duality)

Property 7: Second law of absorption

For all x and y ∈ BS,

a. x + ($\bar{x}$. y) = x + y
b. x . ($\bar{x}$ + y) = x . y

Proof: x + ($\bar{x}$. y) = (x + $\bar{x}$) . (x + y) (postulate 4a)

= 1. (x + y) (postulate 5a)

= x + y (postulate 2b)

x . ($\bar{x}$ + y) = x . y (by duality)

Property 8: Consensus law

For all x, y and z ∈ BS,

a. $x . y + \bar{x} . z + y . z = x . y + \bar{x} . z$

b. $(x + y) . (\bar{x} + z) . (y + z) = (x + y) . (\bar{x} + z)$

Proof: $x . y + \bar{x} . z + y . z$

$= x . y + \bar{x} . z + 1 . y . z$ (postulate 2b)

$= x . y + x . z + (x + \bar{x}) . y . z$ (postulate 5a)

$= x . y + \bar{x} . z + x . y . z + \bar{x} . y . z$ (postulate 4b)

$= x . y + x . y . z + \bar{x} . z + \bar{x} . y . z$ (postulate 3a)

$= x . y . (1 + z) + \bar{x} . z . (1 + y)$ (postulate 4b)

$= x . y . 1 + \bar{x} . z . 1$ (property 2.1b)

$= x . y + \bar{x} . z$ (postulate 2b)

$(x + y) . (\bar{x} + z) . (y + z) = (x + y) . (\bar{x} + z)$ (by duality)

Property 9: Law of identity

For all x and y ε BS, if

a. $x + y = y$

b. $x . y = y$, then $x = y$

Proof: Substituting (a) into the left-hand side of (b), we have

$x . (x + y) = y$

However by the first law of absorption

$x . (x + y) = x$ (property 6)

Therefore, by transitivity $x = y$

Property 10: The law of involution

For all x ∈ BS, $\bar{\bar{x}} = x$

Proof: We need to show that the law of identity (property 2.9) holds, that is,

$(\bar{\bar{x}} + x) = \bar{\bar{x}}$ and $\bar{\bar{x}} . x = \bar{\bar{x}}$

$\bar{\bar{x}} = \bar{\bar{x}} + 0$ (postulate 2a)

$= \bar{\bar{x}} + (x. \bar{x})$ (postulate 5b)

$= (\bar{\bar{x}} + x).(\bar{\bar{x}} + x)$ (postulate 4a)

$= (\bar{\bar{x}} + x).1$ (postulate 5a)

Thus $\bar{\bar{x}} = \bar{\bar{x}} + x$

Also $\bar{\bar{x}}$ = $\bar{\bar{x}}$.1 (postulate 2b)

= $\bar{\bar{x}}$.(x + $\bar{x}$) (postulate 5a)

= $\bar{\bar{x}}$.x + $\bar{\bar{x}}.\bar{x}$ (postulate 4b)

= $\bar{\bar{x}}$.x + 0 (postulate 5b)

= x.x (postulate 2a)

Therefore by the law of identity, we have $\bar{\bar{x}}$ = x

Property 11: DeMorgan's Law

For all x, y ∈ BS,

a. $\overline{x+y}$ = $\bar{x}$. $\bar{y}$

b. $\overline{x.y}$ = $\bar{x}$ + $\bar{y}$

Proof: (x + y).($\bar{x}$. $\bar{y}$) = (x. $\bar{x}$. $\bar{y}$) + (y. $\bar{x}$. $\bar{y}$) (postulate 4b)

= 0 + 0

= 0 (postulate 2a)

(x + y) + ($\bar{x}$. $\bar{y}$) = (x + $\bar{x}$. $\bar{y}$) + y (postulate 3a)

= x + $\bar{y}$ + y (property 2.7a)

= x + 1 (postulate 5a)

= 1 (property 2.16)

Therefore, ($\bar{x}$. $\bar{y}$) is the complement of (x + y).

$\overline{x.y}$ = $\bar{x}$ + $\bar{y}$ (by duality)

DeMorgan's law bridges the AND and OR operations, and establishes a method for converting one form of a Boolean function into another. More particularly it gives a method to form complements of expressions involving more than one variable. By employing the property of substitution, DeMorgan's law can be extended to expressions of any number of variables. Consider the following example:

$\overline{x+y+z}$ = $\bar{x}$. $\bar{y}$. $\bar{z}$

Let y + z = w, then x + y + z = x + w.

$\overline{x+w}$ = $\bar{x}$. $\bar{w}$ (by DeMorgan's law)

$\overline{x+w}$ = $\overline{x+y+z}$(by substitution)

= $\bar{x}$. $\overline{y+z}$ (by DeMorgan's law)

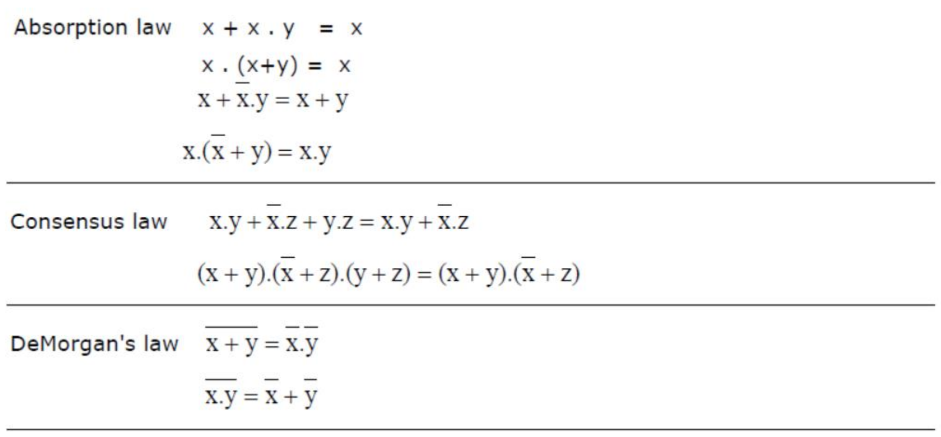
= $\bar{x}$. $\bar{y}$. z (by DeMorgan's law)

At the end of this Section the reader should remind himself that all the postulates and properties of Boolean algebra are valid when the number of elements in the BS is finite. The case of the set BS having only two elements is of more interest here and in the topics that follow in this course on Design of Digital systems.

All the identities derived in this Section are listed in the Table 1 to serve as a ready reference.

TABLE: Useful Identities of Boolean Algebra

Complementation	$x.\overline{x} = 0$ $x + \overline{x} = 1$
0 - 1 law	$x.0 = 0$ $x+1 = 1$ $x+0 = x$ $x.1 = x$
Idempotency	$x.x = x$ $x+x = x$
Involution	$\overline{\overline{x}} = x$
Commutative law	$x . y = y . x$ $X + y = y + x$
Associative law	$(x . y).z = x. (y.z)$ $(x + y) + z = x + (y+z)$
Distributive law	$x + (y.z) = (x+y).(x+z)$ $X . (y+z) = x.y + x.z$
Adjacency law	$x.y + x.\overline{y} = x$ $(x + y).(x + \overline{y}) = x$



Absorption law	$x + x \cdot y = x$
	$x \cdot (x+y) = x$
	$x + \overline{x}.y = x + y$
	$x.(\overline{x} + y) = x.y$
Consensus law	$x.y + \overline{x}.z + y.z = x.y + \overline{x}.z$
	$(x + y).(\overline{x} + z).(y + z) = (x + y).(\overline{x} + z)$
DeMorgan's law	$\overline{x + y} = \overline{x}.\overline{y}$
	$\overline{x.y} = \overline{x} + \overline{y}$

The properties of Boolean algebra when the set BS has two elements, namely 0 and 1, will be explored next.

3.3 Boolean Operators

Recall that Boolean Algebra is defined over a set (BS) with finite number of elements. If the set BS is restricted to two elements {0, 1} then the Boolean variables can take only one of the two possible values. As all switches take only two possible positions, for example ON and OFF, Boolean Algebra with two elements is more suited to working with switching circuits. In all the switching circuits encountered in electronics, the variables take only one of the two possible values.

Definition: A binary variable is one that can assume one of the two values, 0 or 1. These two values, however, are meant to express two exactly opposite states. It means, if a binary variable $A \neq 0$ then $A = 1$. Similarly if $A \neq 1$, then $A = 0$.

Note that it agrees with our intuitive understanding of electrical switches we are familiar with.

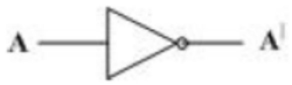
a. if switch A is not open then it is closed
b. if switch A is not closed then it is open

The values 0 and 1 should not be treated numerically, such as to say "0 is less than 1" or " 1 is greater than 0".

Definition: The Boolean operator NOT, also known as complement operator represented by "– " (overbar) on the variable, or " / " (a superscript slash) after the variable, is defined by the following table.

A	A/
0	1
1	0

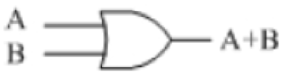
Though it is more popular to use the symbol "– " (overbar) in most of the text-books, we will adopt the " / " to represent the complement of a variable, for convenience of typing. The circuit representation of the NOT operator is shown in the following:



Definition: The Boolean operator "+" known as OR operator is defined by the table given in the following.

A	B	A+B
0	0	0
0	1	1
1	0	1
1	1	1

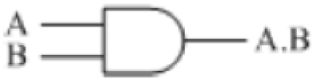
The circuit symbol for logical OR operation is given in the following.



Definition: The Boolean operator "." known as AND operator is defined by the table given below:

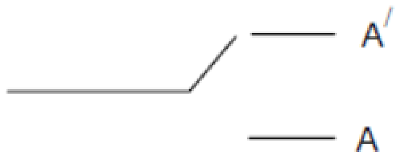
A	B	A.B
0	0	0
0	1	0
1	0	0
1	1	1

The circuit symbol for the logical AND operation is given in the following.



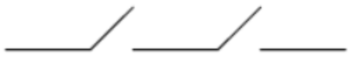
The relationship of these operators to the electrical switching circuits can be seen from the equivalent circuits given in the following.

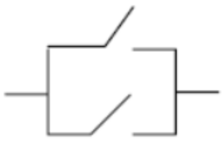
Consider the NOT operator



A'	A
open	closed
closed	open

Consider the OR and AND operators:





A	B	A + B	A.B
Open	Open	Open	Open
Open	Closed	Closed	Open
Closed	Open	Closed	Open
Closed	Closed	Closed	Closed

We can define several other logic operations besides these three basic logic operations. These include

- NAND
- NOR
- Exclusive-OR (Ex-OR for short)
- Exclusive-NOR (Ex-NOR)

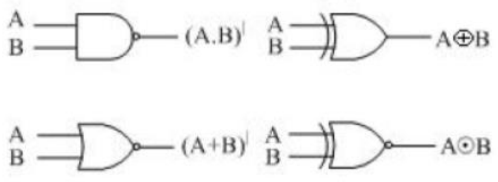
These are defined in terms of different combinations of values the variables assume, as indicated in the following table:

A	B	$(A.B)'$ NAND	$(A+B)'$ NOR	$A \oplus B$ EX-OR	$A \odot B$ EX-NOR
0	0	1	1	0	1
0	1	1	0	1	0
1	0	1	0	1	0
1	1	0	0	0	1

Observe the following:

- NAND operation is just the complement of AND operation
- NOR operation is the complement of OR operation.
- Exclusive-OR operation is similar to OR operation except that EX-OR operation leads to 0, when the two variables take the value of 1.
- Exclusive-NOR is the complement of Exclusive-OR operation.

These functions can also be represented graphically as shown in the figure.

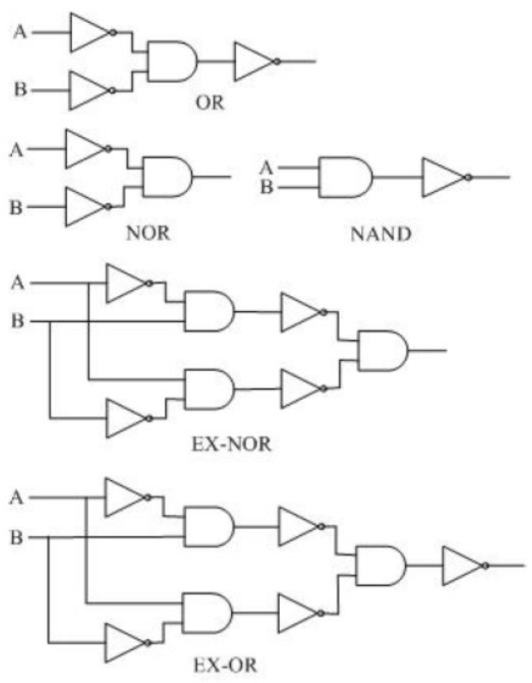


A set of Boolean operations is called functionally complete set if all Boolean expressions can be expressed by that set of operations. AND, OR and NOT constitute a functionally complete set. However, it is possible to have several combinations of Boolean operations as functionally complete sets.

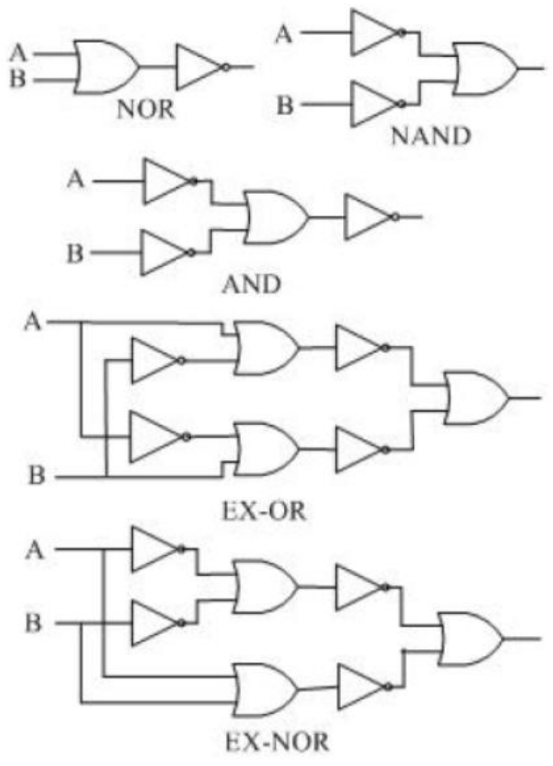
- OR, AND and NOT
- OR and NOT
- AND and NOT
- NAND
- NOR

The completeness of these combinations is shown in the following.

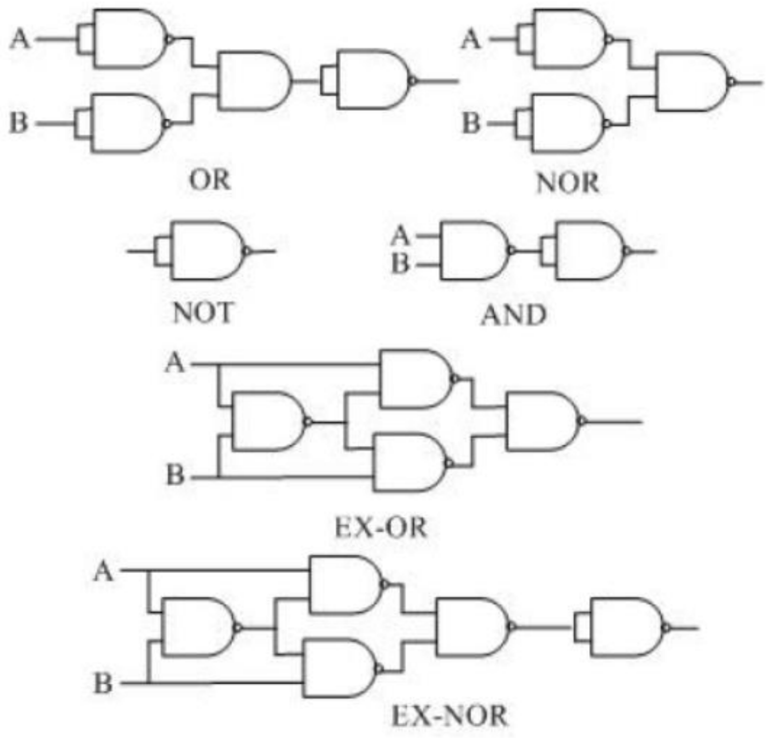
All Boolean functions through AND and NOT operations



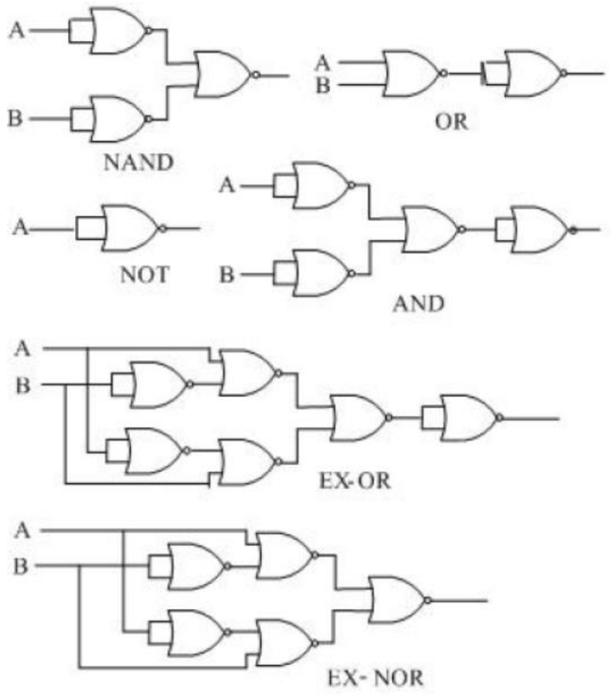
All Boolean functions through OR and NOT operations



All Boolean functions through NAND function

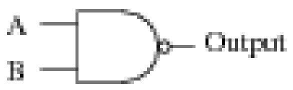


All Boolean functions through NOR function

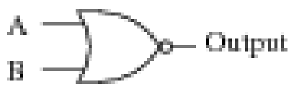


Check Your Progress

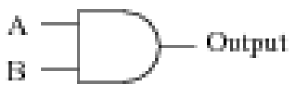
1 1. Identify each of these logic gates by name, and complete their respective truth tables:



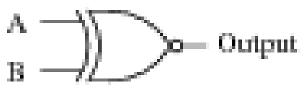
A	B	Output
0	0	
0	1	
1	0	
1	1	



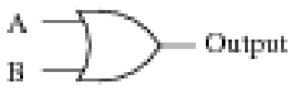
A	B	Output
0	0	
0	1	
1	0	
1	1	



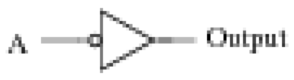
A	B	Output
0	0	
0	1	
1	0	
1	1	



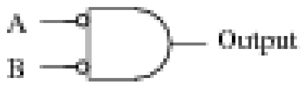
A	B	Output
0	0	
0	1	
1	0	
1	1	



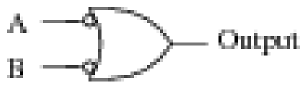
A	B	Output
0	0	
0	1	
1	0	
1	1	



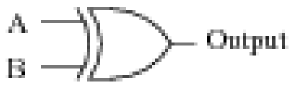
A	Output
0	
1	



A	B	Output
0	0	
0	1	
1	0	
1	1	

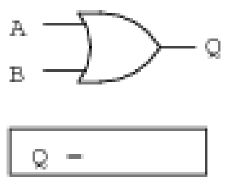


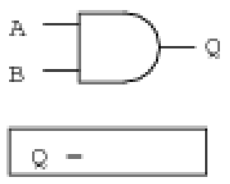
A	B	Output
0	0	
0	1	
1	0	
1	1	

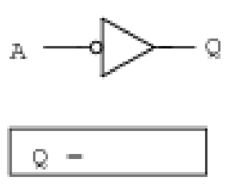


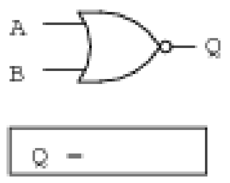
A	B	Output
0	0	
0	1	
1	0	
1	1	

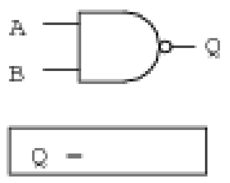
2 Write the Boolean expression for each of these logic gates, showing how the output (Q) algebraically relates to the inputs (A and B):

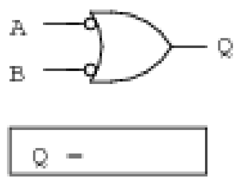


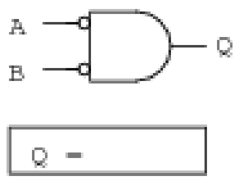




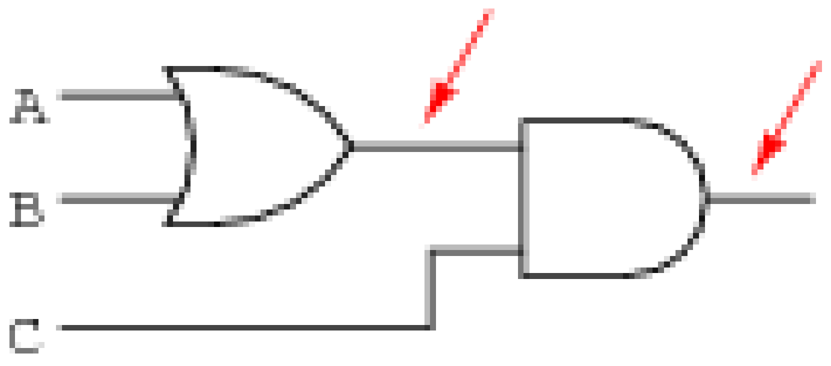








3 Convert the following logic gate circuit into a Boolean expression, writing Boolean sub-expressions next to each gate output in the diagram:



3.4 Model Questions

4 What is a postulates?

5 What is Boolean algebra?

6 What are Huntington's postulates?

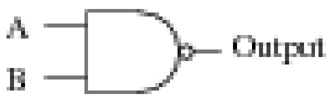
7 What are the several new propositions derived using the basic Huntington’s postulates?

8 What is DeMorgan's Law?
9 Explain different types of Boolean operators.

Answers to Check Your Progress

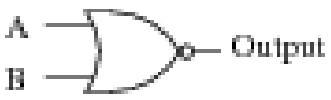
1.

NAND



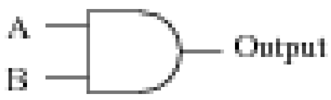
A	B	Output
0	0	1
0	1	1
1	0	1
1	1	0

NOR



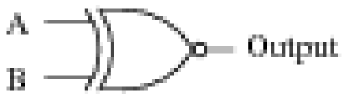
A	B	Output
0	0	1
0	1	0
1	0	0
1	1	0

AND



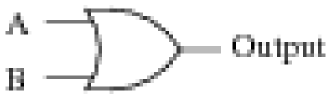
A	B	Output
0	0	0
0	1	0
1	0	0
1	1	1

XNOR



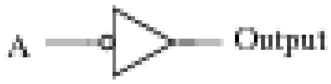
A	B	Output
0	0	1
0	1	0
1	0	0
1	1	1

OR



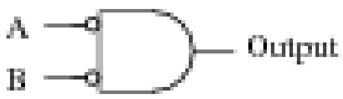
A	B	Output
0	0	0
0	1	1
1	0	1
1	1	1

NOT



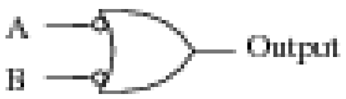
A	Output
0	1
1	0

Neg-AND



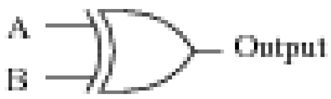
A	B	Output
0	0	1
0	1	0
1	0	0
1	1	0

Neg-OR



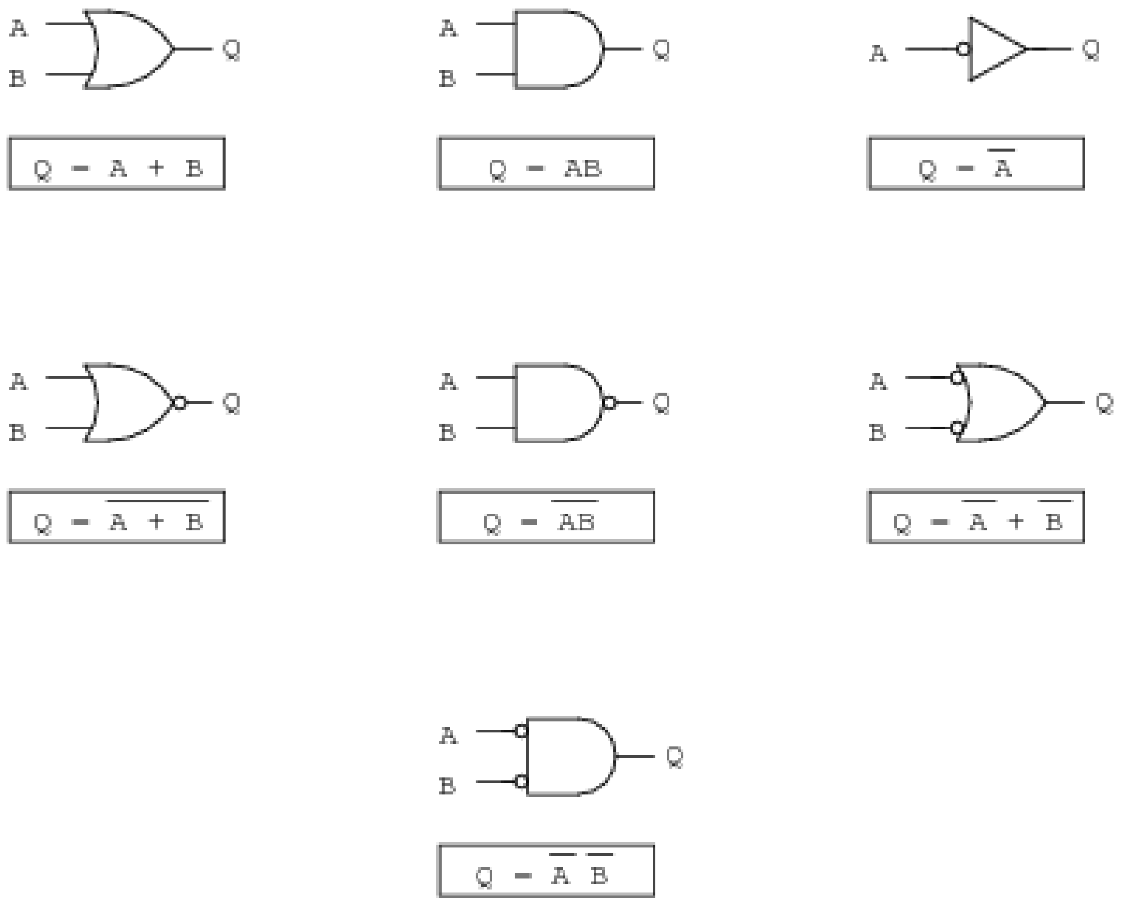
A	B	Output
0	0	1
0	1	1
1	0	1
1	1	0

XOR

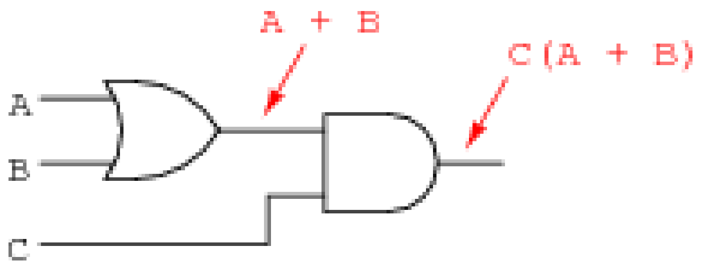


A	B	Output
0	0	0
0	1	1
1	0	1
1	1	0

2.


A
B
Q
$Q = A + B$
$Q = AB$
$Q = \overline{A}$
$Q = \overline{A + B}$
$Q = \overline{AB}$
$Q = \overline{A} + \overline{B}$
$Q = \overline{A}\ \overline{B}$

3.


A + B
C (A + B)
A
B
C

UNIT IV: LOGIC FUNCTIONS

4.0 Learning Objectives

After reading this chapter, you will be able to:

- Explain the characteristics and features of logic families.
- Know noise immunity
- Measure the noise immunity
- Know AC and DC noise margin
- Create a logic circuit if its description is given in terms of a truth-table or as a logic function.
- Convert a given truth-table into a logic function into minterm or maxterm forms.
- Explain the nature and role of “don’t care” terms

4.1 Introduction

Many types of electrical and electronic circuits can be built with devices that have two possible states. We are, therefore interested in working with variables, which can take only two values. Such two valued variables are called Logic variables or Switching variables.

We defined several Boolean operators, which can also be called Logic operators. We will find that it is possible to describe a wide variety of situations and problems using logic variables and logic operators. This is done through defining a “logic function” of logic variables. We can describe logic functions in several ways. These include:

- Algebraic
- Truth-table
- Logic circuit
- Hardware description language
- Maps

We use all these forms to express logic functions in working with digital circuits. Each form of representation is convenient in some context. Initially we will work with algebraic, truth-table, and logic circuit representation of logic functions. The objectives of this learning unit are:

1 Writing the output of a logic network, whose word description is given, as a function of the input variables either as a logic function, a truth-table, or a logic circuit.
2 Create a truth-table if the description of a logic circuit is given in terms of a logic function or as a logic circuit.
3 Write a logic function if the description of a logic circuit is given in terms of a truth-table or as a logic circuit.
4 Create a logic circuit if its description is given in terms of a truth-table or as a logic function.
5 Expand a given logic function in terms of its minterms or maxterms.
6 Convert a given truth-table into a logic function into minterm or maxterm forms.
7 Explain the nature and role of "don't care" terms

4.2 Logic Functions in Algebraic Form

Let A_1, A_2, . . . A_n be logic variables defined on the set BS = {0,1}. A logic function of n variables associates a value 0 or 1 to every one of the possible 2^n combinations of the n variables. Let us consider a few examples of such functions.

F1 = A1.A2'.A3.A4 + A1'.A2.A3'.A4 + A1.A2'.A3.A4'

F1 is a function of 4 variables. You notice that all terms in the function have all the four variables. It is not necessary to have all the variables in all the terms. Consider the following example.

F2 = A1.A2 + A1'.A2.A3' + A1'.A2.A4'.A5

F2 happens to be simplified version of a function, which has a much larger number of terms, where each term has all the variables. We will explore ways and means to generate such simplifications from a given logic expression. The logic functions have the following properties:

1 If $F1(A_1, A_2, ... A_n)$ is a logic function, then $(F1(A_1, A_2, ... A_n))'$ is also a Boolean function.

2 If F1 and F2 are two logic functions, then F1+F2 and F1.F2 are also Boolean functions.

3 Any function that is generated by the finite application of the above two rules is also a logic function

Try to understand the meaning of these properties by solving the following examples.

If F1 = A.B.C + A.B/.C + A.B.C/ what is the logic function that represents F1/ ?

If F1 = A.B + A/.C and F2 = A.B/ + B.C write the logic functions F1 + F2 and F1.F2?

As each one of the combinations can take value of 0 or 1, there are a total of 2^{2^n} distinct logic functions of n variables. It is necessary to introduce a few terms at this stage.

"**Literal**" is a not-complemented or complemented version of a variable. A and A' are literals

"Product term" or "product" refers to a series of literals related to one another through an AND operator. Examples of product terms are A.B'.D, A.B.D'.E, etc.

"**Sum term**" or "**sum**" refers to a series of literals related to one another through an OR operator. Examples of sum terms are A+B'+D, A+B+D'+E, etc.

The choice of terms "product" and "sum" is possibly due to the similarity of OR and AND operator symbols "+" and "." to the traditional arithmetic addition and multiplication operations.

4.3 Truth Table Description of Logic Functions

The truth table is a tabular representation of a logic function. It gives the value of the function for all possible combinations of values of the variables. If there are three variables in a given function, there are $2^3 = 8$ combinations of these variables. For each combination, the function takes either 1 or 0. These combinations are listed in a table, which constitutes the truth table for the given function. Consider the expression,

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

The truth table for this function is given by,

A	B	F
0	0	0
0	1	0
1	0	1
1	1	1

The information contained in the truth table and in the algebraic representation of the function are the same. The term ‘truth table’ came into usage long before Boolean algebra came to be associated with digital electronics. Boolean functions were originally used to establish truth or falsehood of statements. When statement is true the symbol "1" is associated with it, and when it is false "0" is associated. This usage got extended to the variables associated with digital circuits. However, this usage of adjectives "true" and "false" is not appropriate when associated with variables encountered in digital systems. All variables in digital systems are indicative of actions. Typical examples of such signals are "CLEAR", "LOAD", "SHIFT", "ENABLE", and "COUNT". These are suggestive of actions. Therefore, it is appropriate to state that a variable is ASSERTED or NOT ASSERTED than to say that a variable is TRUE or FALSE. When a variable is asserted, the intended action takes place, and when it is not asserted the intended action does not take place. In this context we associate "1" with the assertion of a variable, and "0" with the non-assertion of that variable. Consider the logic function,

F = A.B + A.B’

It should now be read as "F is asserted when A **and** B are asserted **or** A is asserted **and** B is not asserted". This convention of using "assertion” and “non-assertion" with the logic variables will be used in all the Learning Units of this course on Digital Systems.

The term ‘truth table’ will continue to be used for historical reasons. But we understand it as an input-output table associated with a logic function, but not as something that is concerned with the establishment of truth.

As the number of variables in a given function increases, the number of entries in the truth table increases exponentially. For example, a five variable expression would require 32 entries and a six-variable function would require 64 entries. It, therefore, becomes inconvenient to prepare the truth table if the number of variables increases beyond four. However, a simple artefact may be adopted. A truth table can have entries only for those terms for which the value of the function is "1", without loss of any information. This is particularly effective when the function has only a small number of terms. Consider the Boolean function with six variables

F = A.B.C.D’.E’ + A.B’.C.D’.E + A’.B’.C.D.E + A.B’.C’.D.E

The truth table will have only four entries rather than 64, and the representation of this function is

A	B	C	D	E	F
1	1	1	0	0	1
1	0	1	0	1	1
0	0	1	1	1	1
1	0	0	1	1	1

Truth table is a very effective tool in working with digital circuits, especially when the number of variables in a function is small, less than or equal to five.

Check Your Progress

1. ________ is a not-complemented or complemented version of a variable.
2. The ______________is a tabular representation of a logic function.

4.4 Conversion of English Sentences to Logic Functions

Some of the problems that can be solved using digital circuits are expressed through one or more sentences. For example,

- At the traffic junction the amber light should come on 60 seconds after the red light, and get witched off after 5 seconds.
- If the number of coins put into the vending machine exceed five rupees it should dispense a Thums Up bottle.
- The lift should start moving only if the doors are closed and a floor number is chosen.

These sentences should initially be translated into logic equations. This is done through breaking each sentence into phrases and associating a logic variable with each phrase.

As stated earlier many of these phrases will be indicative of actions or directly represent actions. We first mark each action related phrase in the sentence. Then we associate a logic variable with it. Consider the following sentence, which has three phrases:

Anil freaks out with his friends if it is Saturday and he completed his assignments.

We will now associate logic variables with each phrase. The words “if” and “and” are not included in any phrase and they show the relationship among the phrases.

F = 1 if “Anil freaks out with his friends”; otherwise F = 0

A = 1 if “it is Saturday”; otherwise A = 0

B = 1 if “he completed his assignments”; otherwise B = 0

F is asserted if A is asserted and B is asserted. The sentence, therefore, can be translated into a logic equation as

$$F = A.B$$

For simple problems it may be possible to directly write the logic function from the word description. In more complex cases it is necessary to properly define the variables and draw a truth-table before the logic function is prepared. Sometimes the given sentences may have some vagueness, in which case clarifications need to be sought from the source of the sentence. Let us consider another sentence with more number of phrases.

Rahul will attend the Networks class if and only if his friend Shaila is attending the class and the topic being covered in class is important from examination point of view or there

is no interesting matinee show in the city and the assignment is to be submitted. Let us associate different logic variables with different phrases.

Rahul will attend the Networks class if and only if his friend Shaila is attending the class

F A

and the topic being covered in class is important from examination point of view or

B

there is no interesting matinee show in the city and the assignment is to be submitted .

C’ D

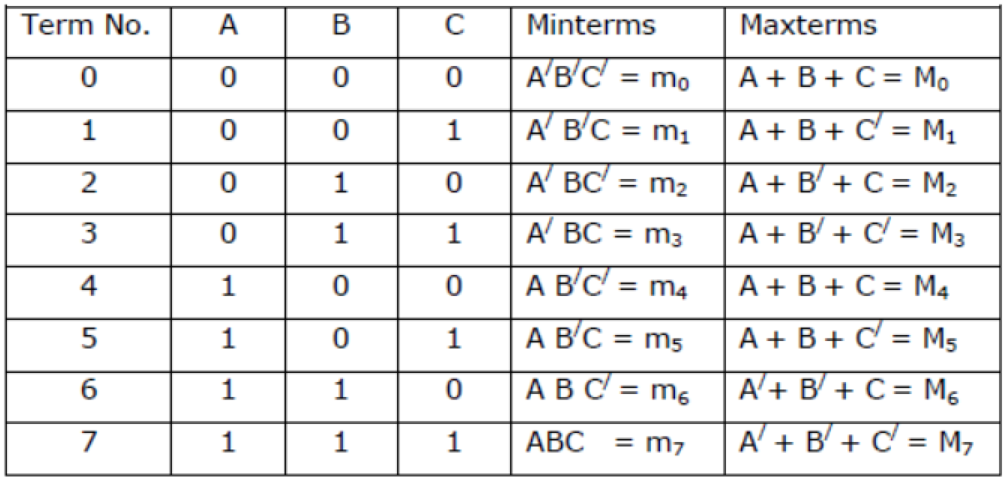
With the above assigned variables the logic function can be written as

$$F = A.B + C'.D$$

4.5 Minterms and Maxterms

A logic function has product terms. Product terms that consist of all the variables of a function are called "canonical product terms", "fundamental product terms" or "minterms". For example the logic term A.B.C' is a minterm in a three variable logic function, but will be a non-minterm in a four variable logic function. Sum terms which contain all the variables of a Boolean function are called "canonical sum terms", "fundamental sum terms" or "maxterms". (A+B'+C) is an example of a maxterm in a three variable logic function.

Consider the Table which lists all the minterms and maxterms of three variables. The minterms are designated as m_0, m_1, . . . m_7, and maxterms are designated as M_0, M_1, . .. M_7.



Term No.	A	B	C	Minterms	Maxterms
0	0	0	0	$A'B'C' = m_0$	$A + B + C = M_0$
1	0	0	1	$A' B'C = m_1$	$A + B + C' = M_1$
2	0	1	0	$A' BC' = m_2$	$A + B' + C = M_2$
3	0	1	1	$A' BC = m_3$	$A + B' + C' = M_3$
4	1	0	0	$A B'C' = m_4$	$A + B + C = M_4$
5	1	0	1	$A B'C = m_5$	$A + B + C' = M_5$
6	1	1	0	$A B C' = m_6$	$A' + B' + C = M_6$
7	1	1	1	$ABC = m_7$	$A' + B' + C' = M_7$

A logic function can be written as a sum of minterms. Consider F, which is a function of three variables.

$F = m_0 + m_3 + m_5 + m_6$

This is equivalent to

$F = A'B'C' + A'BC + AB'C + ABC'$

A logic function that is expressed as an OR of several product terms is considered to be in "sum-of-products" or SOP form. If it is expressed as an AND of several sum terms, it is considered to be in "product-of-sums" or POS form. Examples of these two forms are given in the following:

F1 = A.B + A.B’.C + A’.B.C (SOP form)

F2 = (A+B+C’) . (A+B’+C’) . (A’+B’+C) (POS form)

If all the terms in an expression or function are canonical in nature, that is, as minterms in the case of SOP form, and maxterms in the case of POS form, then it is considered to be in canonical form. For example, the function in the equation (1) is not in canonical form. However it can be converted into its canonical form by expanding the term A.B as

A.B = A . B . 1 (postulate 2b)

= A . B . (C + C’) (postulate 5a)

= A . B . C + A . B . C’ (postulate 4b)

The canonical version of F1 is,

F1 = A.B.C + A.B.C’ + A.B’.C + A’.B.C

The Boolean function F2 is in canonical form, as all the sum terms are in the form of maxterms.

The SOP and POS forms are also referred to as two-level forms. In the SOP form, AND operation is performed on the variables at the first level, and OR operation is performed at the second level on the product terms generated at the first level. Similarly, in the POS form, OR operation is performed at the first level to generate sum terms, and AND operation is performed at the second level on these sum terms. In any logical expression, the right-hand side of a logic function, there are certain priorities in performing the logical operations.

• NOT (‘) operation has the highest priority,

• AND (.) has the next priority

• OR (+) has the last priority.

In the expression for F1 the operations are to be performed in the following sequence

• NOT operation on B and A

• AND terms: A.B, A.B’.C, A’.B.C

• OR operation on AB, AB’C and A’BC

However, the order of priority can be modified through using parentheses. It is also common to express logic functions through multi-level expressions using parentheses. A simple example is shown in the following.

F1 = A.(B+C’) + A’.(C+D)

These expressions can be brought into the SOP form by applying the distributive law.

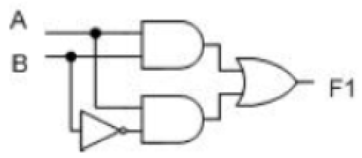
More detailed manipulation of algebraic form of logic functions will be explored in another Learning Unit.

4.6 Circuit Representation of Logic Functions

Representation of basic Boolean operators through circuits was already presented in the earlier Learning Unit. A logic function can be represented in a circuit form using these circuit symbols. Consider the logic function

F1 = A.B + A.B’

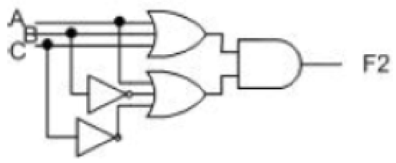
Its circuit form is



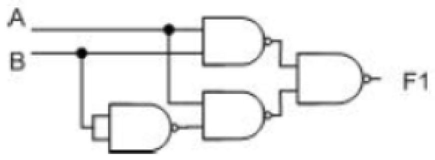
Consider another example of a Boolean function given in POS form.

F2 = (A+B+C) . (A+B’+C’)

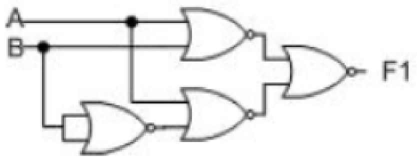
The circuit form of the logical expression F2 is



F1 can also be represented in terms of other functionally complete set of logical operations. NAND is one such functionally complete set. NAND representation of logic expression F2 is



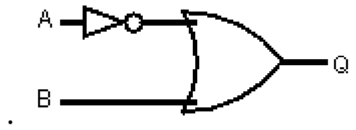
NOR is another functionally complete set. NOR representation of the same function F1 is:



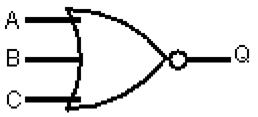
Check Your Progress II

1. Write any valid Boolean expression for Q as a function of the input variables.

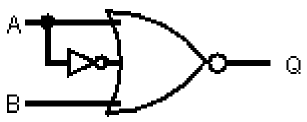
a.



b.



c.



4.7 Model Questions

1. What is Boolean algebra?
2. How can we describe logic functions?
3. What do you mean by tautology?

4. What is a binary decision?
5. How to represent logic functions in algebraic form?
6. What is a truth table? Why it is used?
7. What are canonical product terms?

Answers to Check your Progress I

1. Literal
2. truth table

Answers to Check your Progress II

a. Q = $\overline{A}$+B

b $Q = \overline{A + B + C}$

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

BLOCK II

UNIT V: KARNAUGH-MAP

5.0 Learning Objectives

After the completion of this unit, you will be able to:

- Explain K-Map
- Realize Three-variable K-Map
- Realize Four variable K-Map
- Realize Five variable K-Map
- Implement Boolean functions in POS form
- Perform Minimization with Karnaugh Map

5.1 Karnaugh-Map

The expressions for a logical function (right hand side of a function) can be very long and have many terms and each term many literals. Such logical expressions can be simplified using different properties of Boolean algebra. This method of minimization requires our ability to identify the patterns among the terms. These patterns should conform to one of the four laws of Boolean algebra. However, it is not always very convenient to identify such patterns in a given expression. If we can represent the same logic function in a graphic form that allows us to identify the inherent patterns, then the simplification can be performed more conveniently.

Karnaugh Map is one such graphic representation of a Boolean function in the form of a map. Karnaugh Map is due to M. Karnaugh, who introduced (1953) his version of the map in his paper "The Map Method for Synthesis of Combinational Logic Circuits". Karnaugh Map, abbreviated as K-map, is actually pictorial form of the truth-table. This Learning Unit is devoted to the Karnaugh map and its method of simplification of logic functions.

Karnaugh map of a Boolean function is graphical arrangement of minterms, for which the function is asserted. We can begin by considering a two-variable logic function,

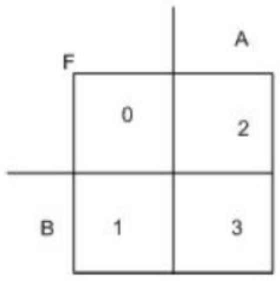
F = A’B + AB

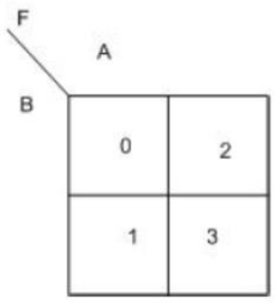
Any two-variable function has $2^2 = 4$ minterms. The truth table of this function is

A	B	F
0	0	0
0	1	1
1	0	1
1	1	0

It can be seen that the values of the variables are arranged in the ascending order (in the numerical decimal sense).

We consider that any two terms are logically adjacent if they differ only with respect any one variable. For example ABC is logically adjacent to A'BC, AB'C and ABC'. But it is not logically adjacent to A'B'C, A'BC', A'B'C', AB'C'.
The entries in the truth-table that are positionally adjacent are not logically adjacent. For example A**'**B (01) and AB**'** (10) are postionally adjacent but are not logically adjacent. The combination of 00 is logically adjacent to 01 and 10. Similarly 11 is adjacent to 10 and 01. Karnaugh map is a method of arranging the truth-table entries so that the logically adjacent terms are also physically adjacent.
The K-map of a two-variable function is shown in the figure. There are two popular ways of representing the map, both of which are shown in the figure. The representation, where the variable above the column or on the side of the row in which it is asserted, will be followed in this and the associated units.





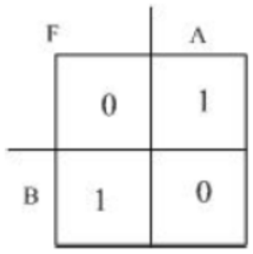
- There are four cells (squares) in this map.
- The cells labelled as 0, 1, 2 and 3 represent the four minterms m_0, m_1, m_2 and m_3.

- The numbering of the cells is chosen to ensure that the logically adjacent minterms are positionally adjacent.
- Cell 1 is adjacent to cell 0 and cell 3, indicating the minterm m1 (01) is logically adjacent to the minterm m_0 (00) and the minterm m_3 (11).
- The second column, above which the variable A is indicated, has the cells 2 and 3 representing the minterms m_2 and m_3. The variable A is asserted in these two minterms.

Let us define the concept of position adjacency. Position adjacency means two adjacent cells sharing one side. Such an adjacency is called simple adjacency. Cell 0 is positionally adjacent to cell 1 and cell 2, because cell 0 shares one side with each of them. Similarly, cell 1 is positionally adjacent to cell 0 and cell 3, as cell 2 is adjacent to cell 0 and cell 3, and cell 3 is adjacent to cell 1 and cell 2.

There are other kinds of positional adjacencies, which become relevant when the number of variables is more than 3. We will explore them at a later time. The main feature of the K-map is that by merely looking at the position of a cell, it is possible to find immediately all the logically adjacent combinations in a function.

The function F = (A/B + AB) can now be incorporated into the K-map by entering "1" in cells represented by the minterms for which the function is asserted. A "0" is entered in all other cells. K-map for the function F is

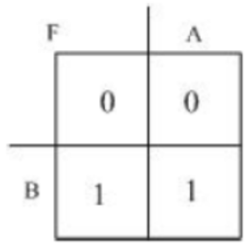


You will notice that the two cells in which "1" is entered are not positionally adjacent. Therefore, they are not logically adjacent.

Consider another function of two variables.

F = A’B + AB

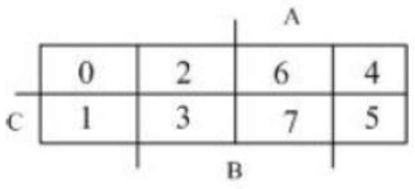
The K-map for this function is



You will notice that the cells in which "1" is entered are positionally adjacent and hence are logically adjacent.

5.1.1 Three-Variable Karnaugh Map

K-map for three variables will have $2^3 = 8$ cells as shown in the figure.



The cells are labelled 0,1,..,7, which stand for combinations 000, 001,...,111 respectively. Notice that cells in two columns are associated with assertion of A, two columns with the assertion of B and one row with the assertion of C.

Let us consider the logic adjacency and position adjacency in the map.

- Cell 7 (111) is adjacent to the cells 3 (011), 5 (101) and 6 (110).
- Cell 2 (010) is adjacent to the cell 0 (000), cell 6 (110) and cell 3 (011).
- We know from logical adjacency the cell 0 (000) and the cell 4 (100) should also be adjacent. But we do not find them positionally adjacent. Therefore, a new adjacency called "cyclic adjacency" is defined to bring the boundaries of a row or a column adjacent to each other. In a three-variable map cells 4 (100) and 0 (000), and cells 1 (001) and 5 (101) are adjacent. The boundaries on the opposite sides of a K-map are considered to be one common side for the associated two cells.

Adjacency is not merely between two cells. Consider the following function:

F = Σ (1, 3, 5, 7)

= m1 + m3 + m5 + m7

= A'B'C + A'BC + AB'C + ABC

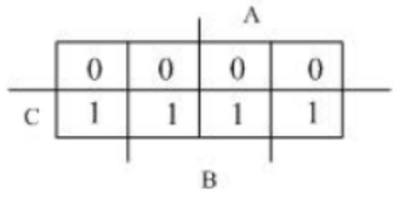
= A'C(B'+B) + AC(B'+B)

= A'C + AC

= (A'+A)C

= C

The K-map of the function F is



It is shown clearly that although there is no logic adjacency between some pairs of the terms, we are able to simplify a group of terms. For example, A’B’C, ABC, A’BC and AB’C are simplified to result in an expression "C". A cyclic relationship among the cells 1, 3, 5 and 7 can be observed on the map in the form 1 → 3 → 7 → 5 → 1
("→" indicating "adjacent to"). When a group of cells, always 2^i ($i < n$) in number, are adjacent to one another in a sequential manner those cells are considered be cyclically adjacent.

Other groups of cells with ‘cyclic adjacency’

- 0, 1, 3 and 2
- 2, 3, 7 and 6
- 6, 7, 5 and 4
- 4, 5, 1 and 0
- 0, 2, 6 and 4

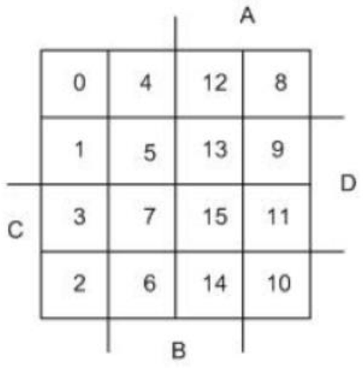
So far we noticed two kinds of positional adjacencies:

- Simple adjacency

- Cyclic adjacency (It has two cases, one is between two cells, and the other among a group of 2i cells)

5.1.2 Four-variable Karnaugh Map

A four-variable (A, B, C and D) K-map will have 24
= 16 cells.



These cells are labelled 0, 1,..., 15, which stand for combinations 0000, 0001,...1111 respectively. Notice that the two sets of columns are associated with assertion of A and B, and two sets of rows are associated with the assertion of C and D.

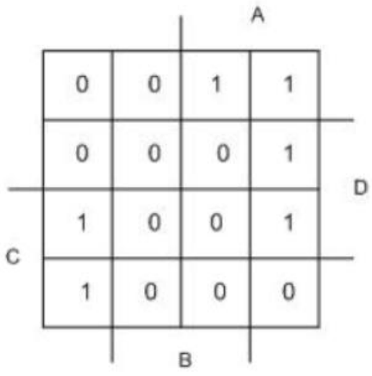
We will be able to observe both simple and cyclic adjacencies in a four-variable map also. 4, 8 and 16 cells can form groups with cyclic adjacency. Some examples of such groups are

- 0, 1, 5 and 4
- 0, 1, 3 and 2
- 10, 11, 9 and 8
- 14, 12,13 and15
- 14, 6, 7 and 15
- 3, 7, 15, 11, 10, 14, 6 and 2

Consider a function of four variables

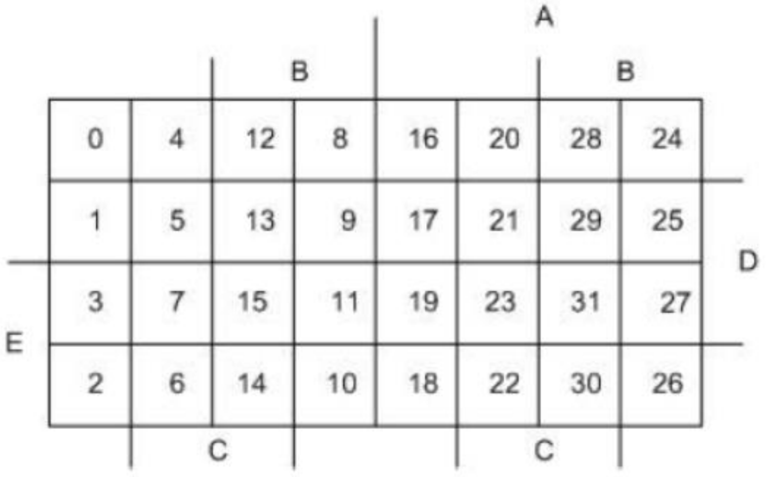
F = Σ (2, 3, 8, 9, 11, 12)

The K-map of this function is



5.1.3 Five-variable Karnaugh Map

Karnaugh map for five variables



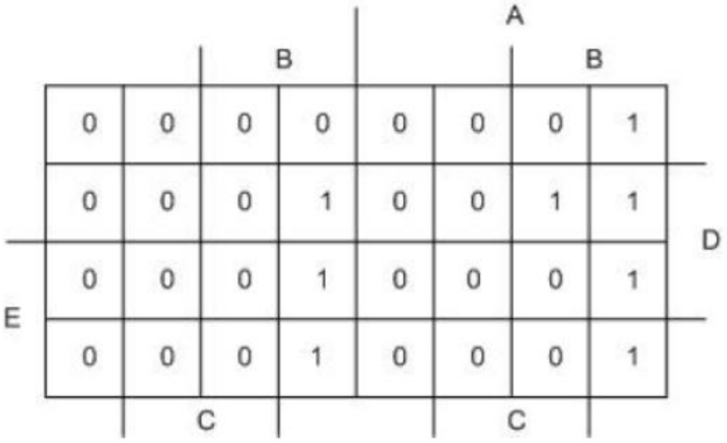
- It has 25 = 32 cells labelled as 0,1, 2 ...,31, corresponding to the 32 combinations from 00000 to 11111.
- The map is divided vertically into two symmetrical parts. Variable A is notasserted on the left side, and is asserted on the right side. The two parts of the map, except for the assertion or non-assertion of the variable A are identical with respect to the remaining four variables B, C, D and E.
- Simple and cyclic adjacencies are applicable to this map, but they need to be applied separately to the two sections of the map. For example cell 8 and cell 0 are adjacent. The largest number of cells coming under cyclic adjacency can go up to $2^5 = 32$.

- Another type of adjacency, called ‘symmetric adjacency’, exists because of the division of the map into two symmetrical sections. Taking the assertion and nonassertion of A into account, we find that cell 0 and cell 16 are adjacent. Similarly, there are 15 more adjacent cell pairs (4-20, 12-28, 8-24, 1-17, 5-21, 13-29, 9- 25, 3-19, 7-23, 15-31, 11-27, 2-18, 6-22, 14-30, and 10-26).

Consider a function of five-variable

F = A’BC’DE’ + A’BCDE’ + A’BC’DE + ABCDE + A’BC’D’E + ABC’DE’ + ABCDE’

+ ABC’DE + ABC’D’E + ABC’D’E’



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

From the study of two-, three-, four- and five-variable Karnaugh maps, we can summaries the following properties:

1 Every Karnaugh map has 2n cells corresponding to 2n minterms.

2 The main feature of a Karnaugh Map is to convert logic adjacency into positional adjacency.

3 There are three kinds of positional adjacency, namely simple, cyclic and symmetric.

We have already seen how a K-map can be prepared provided the Boolean function is available in the canonical SOP form.

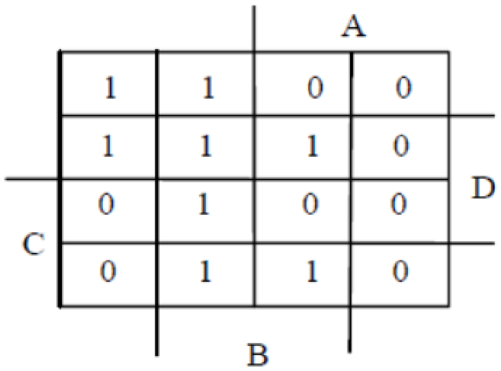
A "1" is entered in all those cells representing the minterms of the expression, and "0" in all the other cells.

However, the Boolean functions are not always available to us in the canonical form. One method is to convert the non-canonical form into canonical SOP form and prepare the K-map. The other method is to convert the function into the standard SOP form and directly prepare the K-map.

Consider the function

F = A’B + A’B’C’ + ABC’D + ABCD’

We notice that there are four variables in the expression. The first term, A’B, containing two variables actually represents four minterms, and the term A’B’C’ represents two minterms. The K-map for this function is



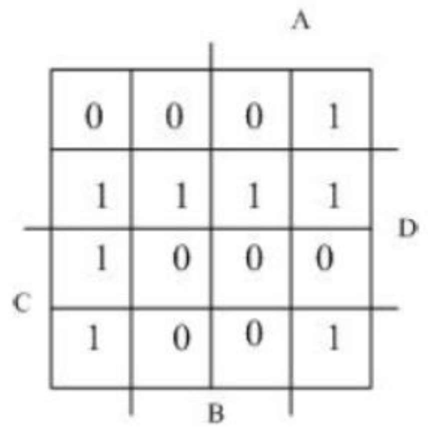
Notice that the second column represents A’B, and similarly A’B’C’ represents the two top cells in the first column. With a little practice it is always possible to fill the K-map with 1s representing a function given in the standard SOP form.

5.1.4 Boolean functions in POS form

Boolean functions, sometimes, are also available in POS form. Let us assume that the function is available in the canonical POS form. Consider an example of such a function

F = Π (0, 4, 6, 7, 11, 12, 14, 15)

In preparing the K-map for the function given in POS form, 0s are filled in the cells represented by the maxterms. The K-map of the above function is



Sometimes the function may be given in the standard POS form. In such situations we can initially convert the standard POS form of the expression into its canonical form, and enter 0s in the cells representing the maxterms. Another procedure is to enter 0s directly into the map by observing the sum terms one after the other.

Consider an example of a Boolean function given in the POS form.

F = (A+B+D’).(A’+B+C’+D).(B’+C)

This may be converted into its canonical form as

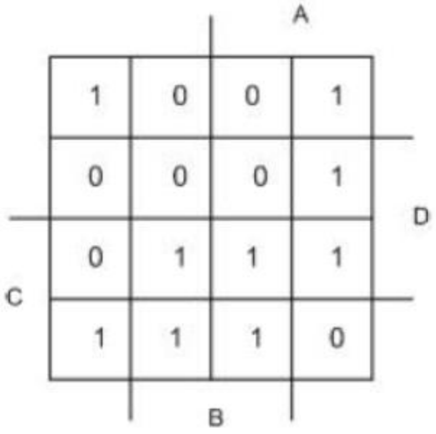
F = (A+B+C+D’).(A+B+C’+D’)(A’+B+C’+D).(A+B’+C+D). (A’+B’+C+D).

(A+B’+C+D’).(A’+B’+C+D’)

= M1 . M3 . M10 . M4 . M12 . M5 . M13

The cells 1, 3, 4, 5, 10, 12 and 13 can have 0s entered in them while the remaining cells are filled with 1s.

The second method is through direct observation. To determine the maxterms associated with a sum term we follow the procedure of associating a 0 with those variables which appear in their asserted form, and a 1 with the variables that appear in their non-asserted form. For example the first term (A+B+D’) has A and B asserted and D non-asserted. Therefore the two maxterms associated with this sum term are 0001 (M1) and 0011 (M3). The second term is in its canonical form and the maxterm associated with is 1010 (M10). Similarly the maxterms associated with the third sum term are 0100 (M4), 1100 (M12), 0101 (M5) and 1101 (M13). The resultant K-map is



We learnt in this Learning Unit

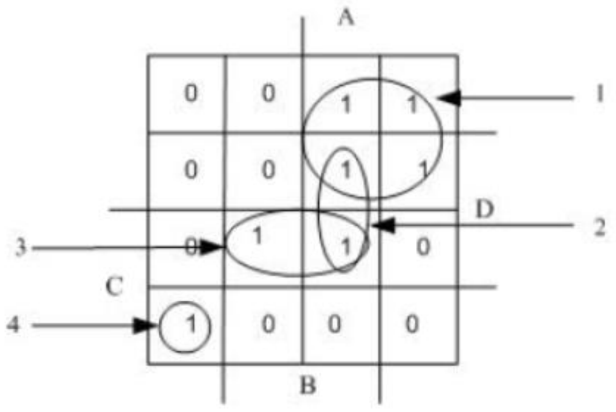
- The logic adjacency is captured as positional adjacency in a Karnaugh Map
- How to translate logic expressions given in SOP or POS forms into K-maps
- There are three types of logical adjacency, namely, simple, cyclic and symmetric adjacencies

5.2 Minimization with Karnaugh Map

Implicants: A Karnaugh map not only includes all the minterms that represent a Boolean function, but also arranges the logically adjacent terms in positionally adjacent cells. As the information is pictorial in nature, it becomes easier to identify any patterns (relations) that exist among the 1-entered cells (minterms). These patterns or relations are referred to as implicants.

Definition 1: An implicant is a group of 2^i (i = 0, 1 ....n) minterms (1-entered cells) that are logically (positionally) adjacent.

A study of implicants enables us to use the K-map effectively for simplifying a Boolean function. Consider the K-map



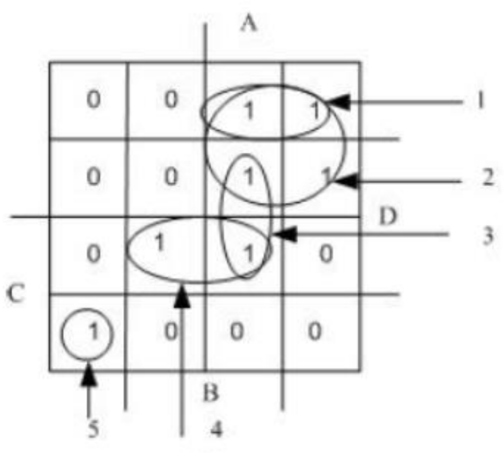
- There are four implicants: 1, 2, 3 and 4.
- The implicant 4 is a single cell implicant. A single cell implicant is a 1-entered cell that is not positionally adjacent to any of the other cells in map.
- The four implicants account for all groupings of 1-entered cells. This also means that the four implicants describe the Boolean function completely.

An implicant represents a product term, with the number of variables appearing in the term inversely proportional to the number of 1-entered cells it represents.

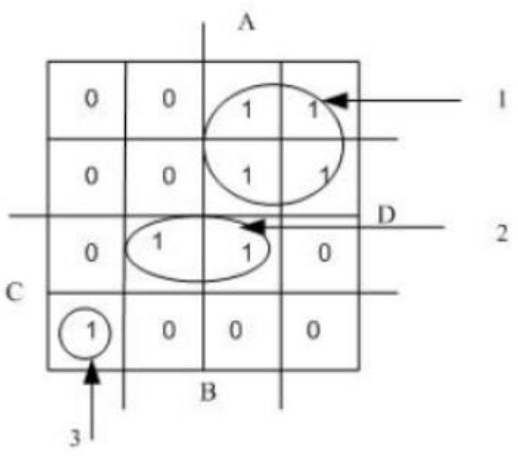
- Implicant 1 in the figure represents the product term AC’
- Implicant 2 represents ABD
- Implicant 3 represents BCD
- Implicant 4 represents A’B’CD’

The smaller the number of implicants, and the larger the number of cells that each implicant represents, the smaller the number of product terms in the simplified Boolean expression.

In this example we notice that there are different ways of identifying the implicants.



(a)



(b)

Five implicants are identified in the figure (a) and three implicants in the figure (b) for the same K-map (Boolean function). It is then necessary to have a procedure to identify the minimum number of implicants to represent a Boolean function.

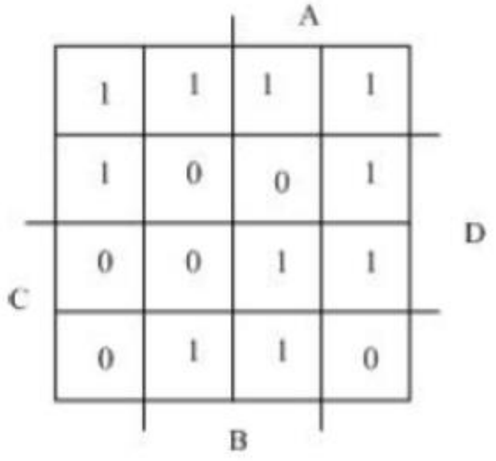
We identify three types of implicants: "prime implicant", "essential implicant" and "redundant implicant".

A **prime implicant** is one that is not a subset of any one of the other implicant. An **essential prime implicant** is a prime implicant which includes a 1-entered cell that is not included in any other prime implicant.

A **redundant implicant** is one in which all the 1-entered cells are covered by other implicants. A redundant implicant represents a redundant term in an expression. Implicants 2, 3, 4 and 5 in the figure (a), and 1, 2 and 3 in the figure (b) are prime implicants. Implicants 2, 4 and 5 in the figure (a), and 1, 2 and 3 in the figure (b) are essential prime implicants.
Implicants 1 and 3 in the figure (a) are redundant implicants. Figure (b) does not have any redundant implicants. Now the method of K-map minimisation can be stated as
"**find the smallest set of prime implicants that includes all the essential prime implicants accounting for all the 1-entered cells of the K-map**".

If there is a choice, the simpler prime implicant should be chosen. The minimization procedure is best understood through examples.

Example 1: Find the minimised expression for the function given by the K-map in the figure.



Fifteen implicants of the K-map are:

X1 = C’D’	X2 = B’C’	X3 = BD’	X4 = ACD
X5 = AB’C’	X6 = BCD’	X7 = A’B’C’	X8 = BC’D’
X9 = B’C’D’	X10 = A’C’D’	X11 = AC’D’	X12 = AB’D
X13 = ABC	X14 = A’BD’	X15 = ABD’	X16 = B’C’D

Obviously all these implicants are not prime implicants and there are several redundant implicants. Several combinations of prime implicants can be worked out to represent the function. Some of them are listed in the following.

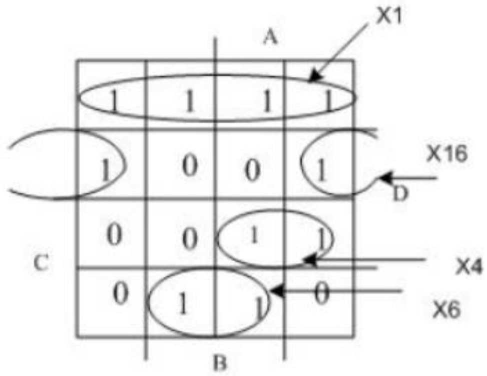
F1 = X1 + X4 + X6 + X16

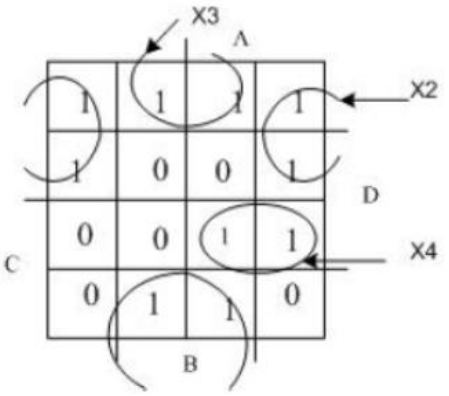
= X4 + X5 + X6 + X7 + X8

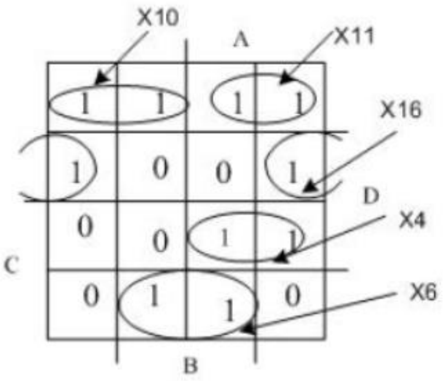
= X2 + X3 + X4

= X10 + X11 + X8 + X4 + X6

The k-maps with these four combinations are







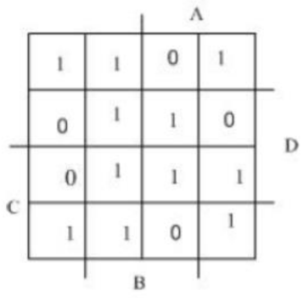
Among the prime implicants listed in the figure there are three implicants X1, X2 and X3 that group four 1-entered cells. Selecting the smallest number of implicants we obtain the simplified expression as:

F = X2 + X3 + X4

= B'C' + BD' + ACD

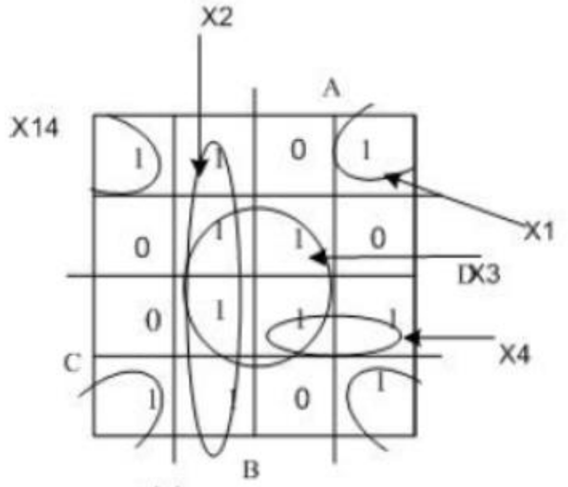
It may be noticed that X2, X3 and X4 are essential prime implicants.

Example 2: Minimise the Boolean function represented by the K-map shown in the figure.

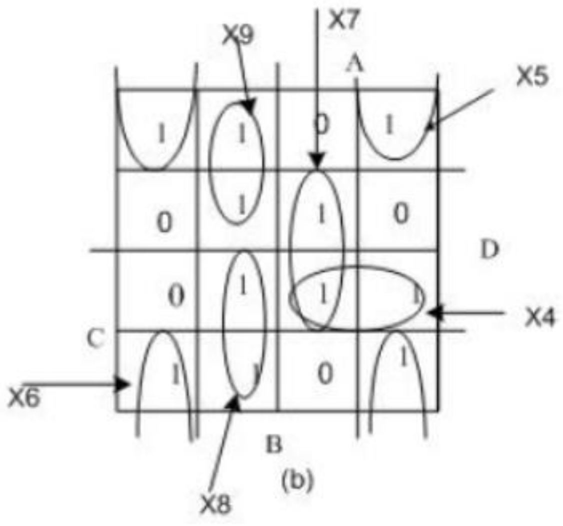


Three sets of prime implicants are:

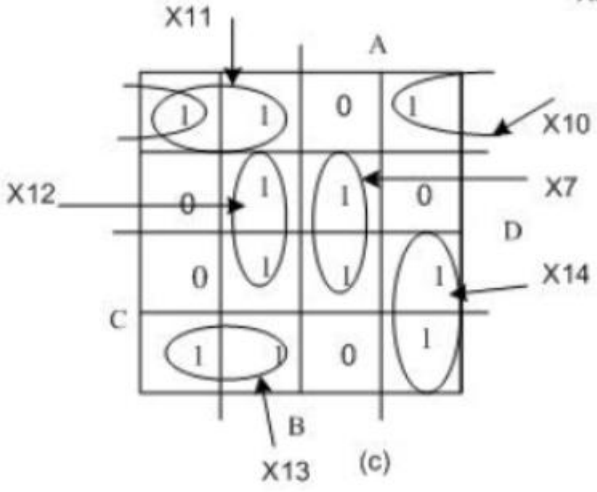
(a) X1 = B'D'	X2 = A'B	X3 = BD	X4 = ACD
(b) X4 = ACD	X5 = AB'D'	X6 = A'B'D'	X7 = ABD
X8 = A'BC	X9 = A'BC'		
(c) X7 = ABD	X10 = B'C'D'	X11 = A'C'D'	X12 = A'BD
X13 = A'C'D'	X14 = AB'C		



(a)



(b)



(c)

Some of the simplified expressions are shown in the following:

F = X1 + X2 + X3 + X4

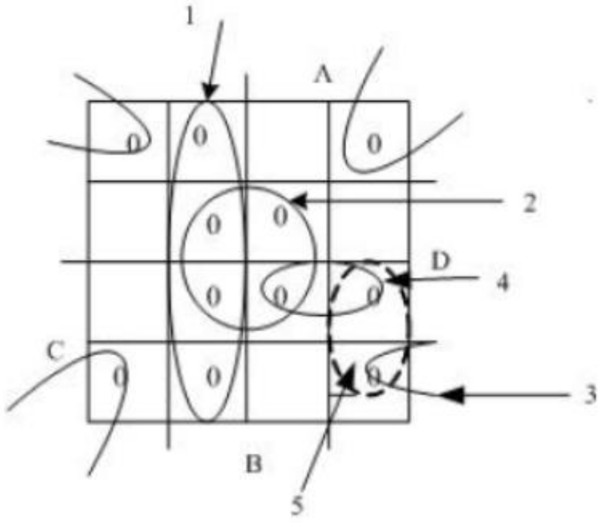
= X4 + X6 + X7 + X8 + X9

= X7 + X10 + X11 + X12 + X13 + X14

5.3 Standard POS form from Karnaugh Map

As mentioned earlier, POS form always follows some kind of duality, yet different from the principle of duality. The implicants are defined as groups of sums or maxterms which in the map representation are the positionally adjacent 0-entered cells rather then 1- entered cells as in the SOP case. When converting an implicant covering some 0-entered cells into a sum, a variable appears in complemented form in the sum if it is always 1 in value in the combinations corresponding to that implicant, a variable appears in uncomplimented form if it is always 0 in value, and the variable does not appear at all if it changes its values in the combinations corresponding to the implicant. We obtain a standard POS form of expression from the map representation by ANDing all the sums converted from implicants.

Example 3: Consider a Boolean function in the POS form represented in the K-map shown in the figure



- Initially four implicants are identified (1, 2, 3 and 4).
- Implicant 1: B is asserted and A is not-asserted in all the cells of implicant 1, where as the variables C and D change their values from 0 to 1. It is represented by the sum term (A + B').
- Implicant 2: It is represented by the sum term (B' + D').
- Implicant 3: It is represented by (B + D).
- Implicant 4: It is represented by (A' + C' + D').

The simplified expression in the POS form is given by;

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

If we choose the implicant 5 (shown by the dotted line in the figure 19) instead of 4, the simplified expression gets modified as:

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

We may summarise the procedure for minimization of a Boolean function through a K-map as follows:

1 Draw the K-map with 2^n cells, where n is the number of variables in a Boolean function.

2 Fill in the K-map with 1s and 0s as per the function given in the algebraic form (SOP or POS) or truth-table form.

3 Determine the set of prime implicants that consist of all the essential prime implicants as per the following criteria:

   - All the 1-entered or 0-entered cells are covered by the set of implicants, while

     making the number of cells covered by each implicant as large as possible.
   - Eliminate the redundant implicants.
   - Identify all the essential prime implicants.
   - Whenever there is a choice among the prime implicants select the prime implicant with the smaller number of literals.

4 If the final expression is to be generated in SOP form, the prime implicants should be identified by suitably grouping the positionally adjacent 1-entered cells, and converting each of the prime implicant into a product term. The final SOP expression is the OR of the identified product terms.

5 If the final simplified expression is to be given in the POS form, the prime implicants should be identified by suitably grouping the positionally adjacent 0-entered cells, and converting each of the prime implicant into a sum term. The final POS expression is the AND of the identified sum terms.

5.4 Simplification of Incompletely Specified Functions

So far we assumed that the Boolean functions are always completely specified, which means a given function assumes strictly a specific value, 1 or 0, for each of its 2^n input combinations. This, however, is not always the case. Consider the example is the BCD decoders

- The ten outputs are decoded from sixteen possible input combinations produced by four inputs representing BCD codes.
- An encoding scheme chooses ten valid codes.
- Irrespective of the encoding scheme there are always six combinations of the inputs that would be considered as invalid codes.
- If the input unit to the BCD decoder works in a functionally correct way, then the six invalid combinations of the inputs should never occur.

In such a case, it does not matter what the output of the decoder is for these six combinations. As we do not mind what the values of the outputs are in such situations, we call them "don’t-care" situations. These don’t-care situations can be used advantageously in generating a simpler Boolean expression than without taking that advantage. Such don’t-care combinations of the variables are represented by an "X" in the appropriate cell of the K-map.

Example: This example shows how an incompletely specified function can be represented in truth-table, Karnaugh map and canonical forms.

The decoder has three inputs A, B and C representing three bit codes and an output F. Out

of the $2^3 = 8$ possible combinations of the inputs, only five are described and hence constitute the valid codes. F is not specified for the remaining three input codes, namely, 000, 110 and 111.

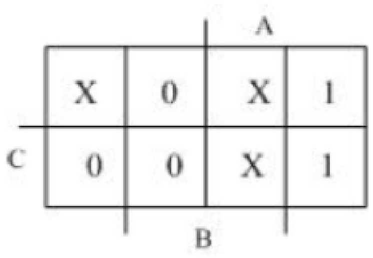
Functional description of a decoder

Mode No	Input Code	Output	Description
1	0 0 1	0	Input from keyboard
2	0 1 0	0	Input from mouse
3	0 1 1	0	Input from light-pen
4	1 0 0	1	Output to printer
5	1 0 1	1	Output to plotter

Treating these three combinations as the don't-care conditions, the truth-table may be written as:

A	B	C	F
0	0	0	X
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	X
1	1	1	X

The K-map for this function is



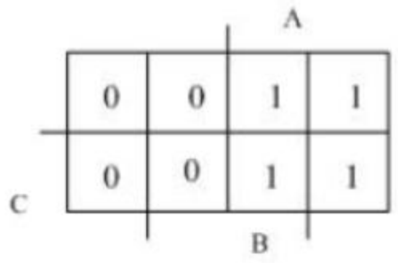
The function in the SOP and POS forms may be written as

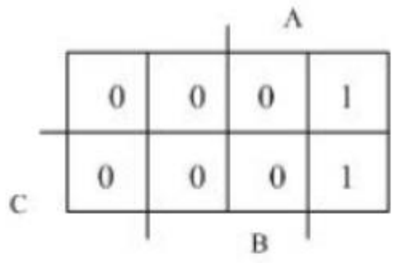
F = Σ(4, 5) + d (0, 6, 7)

F = Π (1, 2, 3) . d (0, 6, 7)

The term d (0, 6, 7) represents the collection of don't-care terms.

The don't-cares bring some advantages to the simplification of Boolean functions. The Xs can be treated either as 0s or as 1s depending on the convenience. For example the above map can be redrawn in two different ways as





The simplification can be done, therefore, in two different ways. The resulting expressions for the function F are:

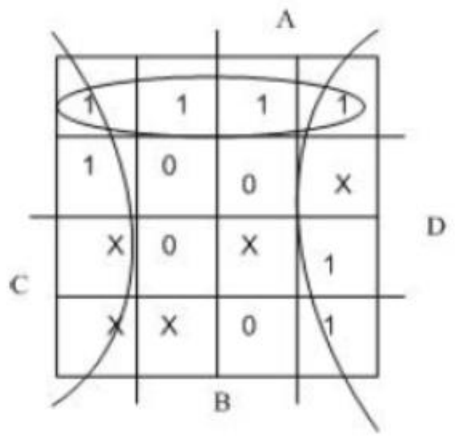
F = A

F = AB’

We can generate a simpler expression for a given function by utilizing some of the don’t care conditions as 1s.

Example: Simplify F = Σ (0,1,4,8,10,11,12) + d(2,3,6,9,15)

The K-map of this function is



The simplified expression taking the full advantage of the don’t cares is,

F = B’ + C’D’

Simplification of several functions of the same set of variables

As there could be several product terms that could be made common to more than one function, special attention needs to be paid to the simplification process.

Example: Consider the following set of functions defined on the same set of variables:

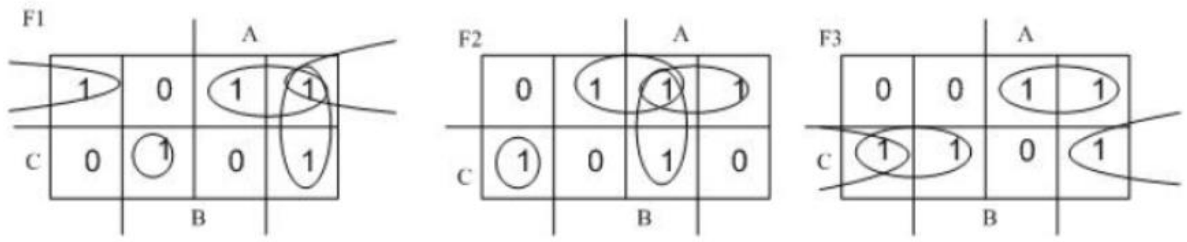
F1 (A, B, C) = Σ (0, 3, 4, 5, 6)

F2 (A, B, C) = Σ (1, 2, 4, 6, 7)

F3 (A, B, C) = Σ (1, 3, 4, 5, 6)

Let us first consider the simplification process independently for each of the functions. The

K-maps for the three functions and the groupings are



The resultant minimal expressions are:

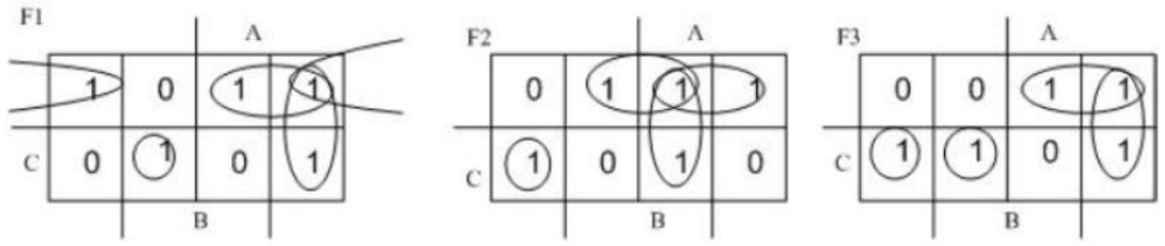
F1 = B’C’ + AC’ + AB’ + A’BC

F2 = BC’ + AC’ + AB + A’B’C

F3 = AC’ + B’C + A’C

These three functions have nine product terms and twenty one literals.

If the groupings can be done to increase the number of product terms that can be shared among the three functions, a more cost effective realisation of these functions can be achieved. One may consider, at least as a first approximation, cost of realising a function is proportional to the number of product terms and the total number of literals present in the expression. Consider the minimisation shown in the figure



The resultant minimal expressions are;

F1 = B’C’ + AC’ + AB’ + A’BC

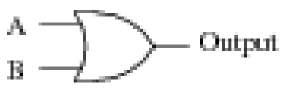
F2 = BC’ + AC’ + AB + A’B’C

F3 = AC’ + AB’ + A’BC + A’B’C

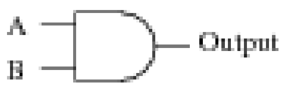
This simplification leads to seven product terms and sixteen literals.

Check Your Progress

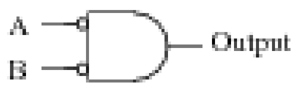
1. Identify each of these logic gates by name, and complete their respective truth tables:



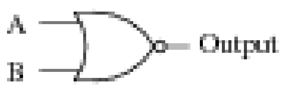
A	B	Output
0	0	
0	1	
1	0	
1	1	



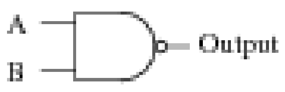
A	B	Output
0	0	
0	1	
1	0	
1	1	



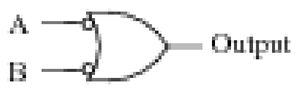
A	B	Output
0	0	
0	1	
1	0	
1	1	



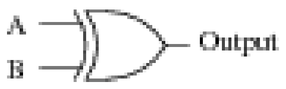
A	B	Output
0	0	
0	1	
1	0	
1	1	



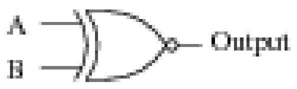
A	B	Output
0	0	
0	1	
1	0	
1	1	



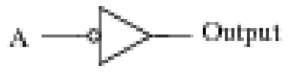
A	B	Output
0	0	
0	1	
1	0	
1	1	



A	B	Output
0	0	
0	1	
1	0	
1	1	



A	B	Output
0	0	
0	1	
1	0	
1	1	



A	Output
0	
1	

2. Minimise the following problems using the Karnaugh maps method.

Z = f(A,B,C) = $\bar{C} + \bar{A}B + AB\bar{C} + AC$

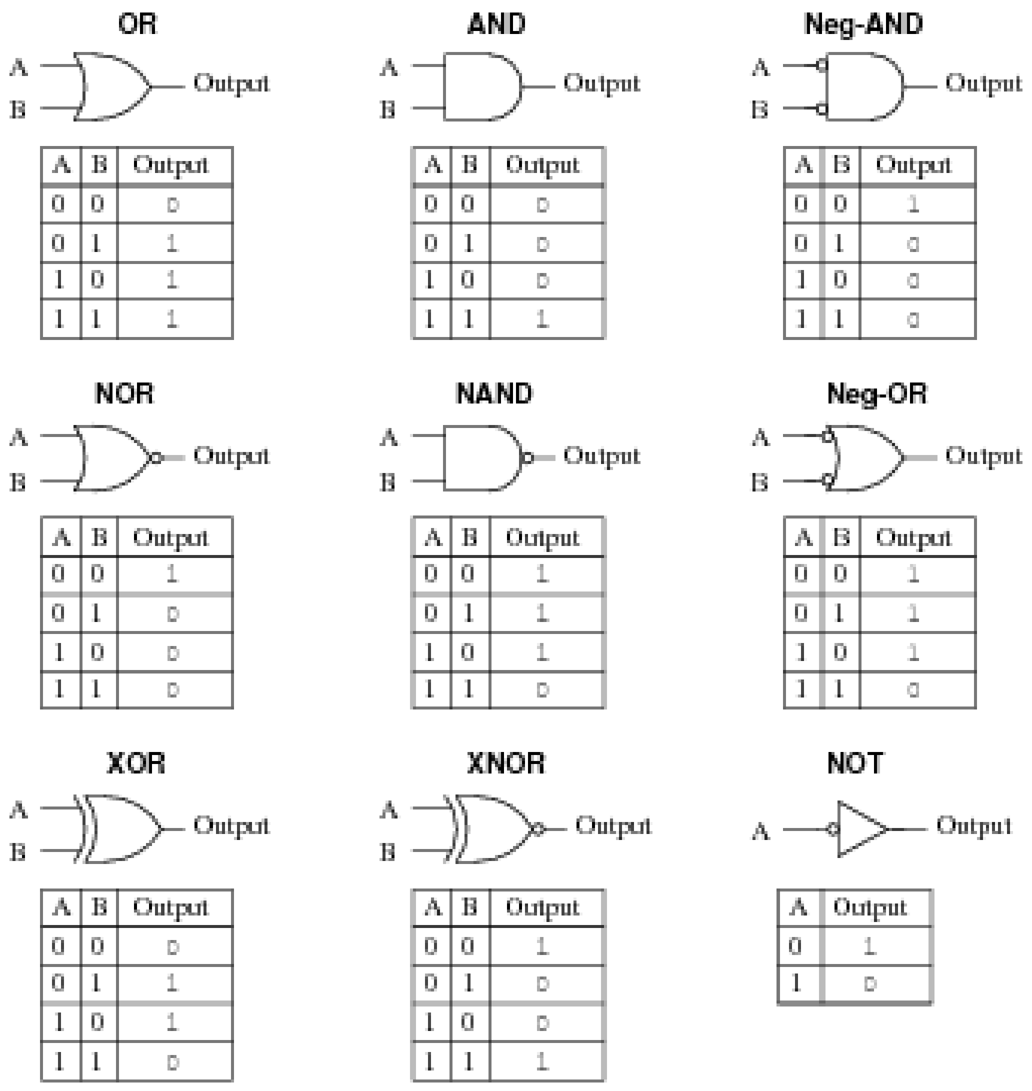
Z = f(A,B,C) = $\bar{A}B + B\bar{C} + BC + A\bar{B}\bar{C}$

Model Question

1. What is a Karnaugh-map? Why it is used?
2. Explain 3-varialble K-map with the help of an example.
3. What is POS form of representation of Boolean functions?

Answers to Check Your Progress

1.



A	B	Output
0	0	0
0	1	1
1	0	1
1	1	1

A	B	Output
0	0	0
0	1	0
1	0	0
1	1	1

A	B	Output
0	0	1
0	1	0
1	0	0
1	1	0

A	B	Output
0	0	1
0	1	0
1	0	0
1	1	0

A	B	Output
0	0	1
0	1	1
1	0	1
1	1	0

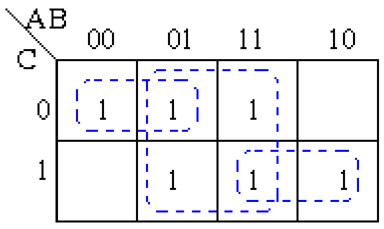
A	B	Output
0	0	1
0	1	1
1	0	1
1	1	0

A	B	Output
0	0	0
0	1	1
1	0	1
1	1	0

A	B	Output
0	0	1
0	1	0
1	0	0
1	1	1

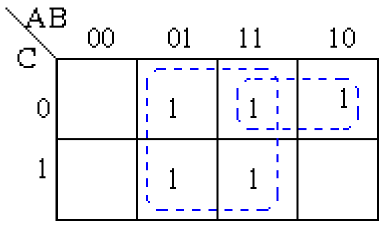
A	Output
0	1
1	0

2. $Z = f(A,B,C) = \bar{A}\bar{B}\bar{C} + \bar{A}B + AB\bar{C} + AC$



By using the rules of simplification and ringing of adjacent cells in order to make as many variables redundant, the minimised result obtained is B + AC+ $\bar{A}\bar{C}$

Z = f(A,B,C) = $\bar{A}$B + B$\bar{C}$ + BC + A$\bar{B}\bar{C}$



By using the rules of simplification and ringing of adjacent cells in order to make as many variables redundant, the minimised result obtained is B + A $\bar{C}$

UNIT VI: QUINE-MCCLUSKEY METHOD OF MINIMIZATION

6.0 Learning Objectives

After the completion of this unit, you will be able to:

- Explain the Principle of Quine-McClusky Method
- Perform Generation of Prime Implicants
- Determine of the Minimal Set of Prime Implicants
- Perform simplification of Incompletely Specified functions

6.1 Introduction

Karnaugh Map provides a good method of minimizing a logic function. However, it depends on our ability to observe appropriate patterns and identify the necessary implicants. If the number of variables increases beyond five, K-map or its variant Variable Entered Map can become very messy and there is every possibility of committing a mistake. What we require is a method that is more suitable for implementation on a computer, even if it is inconvenient for paper-and-pencil procedures. The concept of tabular minimisation was originally formulated by Quine in 1952. This method was later improved upon by McClusky in 1956, hence the name Quine-McClusky.

This Learning Unit is concerned with the Quine-McClusky method of minimisation. This method is tedious, time-consuming and subject to error when performed by hand. But it is better suited to implementation on a digital computer.

6.2 Principle of Quine-McClusky Method

The Quine-McClusky method is a two stage simplification process.

- Generate prime implicants of the given Boolean function by a special tabulation process.

- Determine the minimal set of implicants is determined from the set of implicants generated in the first stage.

The tabulation process starts with a listing of the specified minterms for the 1s (or 0s) of a function and don’t-cares (the unspecified minterms) in a particular format. All the prime implicants are generated from them using the simple logical adjacency theorem, namely, AB**’** + AB = A. The main feature of this stage is that we work with the equivalent binary number of the product terms. For example in a four variable case, the minterms A’BCD’ and A’BC’D’ are entered as 0110 and 0100. As the two logically adjacent minterms A/BCD/ and A/BC/D/ can be combined to form a product term A/BD/,the two binary terms 0110 and 0100 are combined to form a term represented as "01-0", where ‘-‘ (dash) indicates the position where the combination took place.
Stage two involves creating a prime implicant table. This table provides a means of identifying, by a special procedure, the smallest number of prime implicants that represents the original Boolean function. The selected prime implicants are combined to form the simplified expression in the SOP form. While we confine our discussion to the creation of minimal SOP expression of a Boolean function in the canonical form, it is easy to extend the procedure to functions that are given in the standard or any other forms.

6.2.1 Generation of Prime Implicants

The process of generating prime implicants is best presented through an example.

Example 1: F = Σ (1,2,5,6,7,9,10,11,14)

All the minterms are tabulated as binary numbers in sectionalised format, so that each section consists of the equivalent binary numbers containing the same number of 1s, and the number of 1s in the equivalent binary numbers of each section is always more than that in its previous section. This process is illustrated in the table as below.

Section	Column 1 No. of 1s	Binary	Decimal
1	1	0001	1
		0010	2
2	2	0101	5
		0110	6
		1001	9
		1010	10
3	3	0111	7
		1011	11
		1110	14

The next step is to look for all possible combinations between the equivalent binary numbers in the adjacent sections by comparing every binary number in each section with every binary number in the next section. The combination of two terms in the adjacent sections is possible only if the two numbers differ from each other with respect to only one bit. For example 0001 (1) in section 1 can be combined with 0101 (5) in section 2 to result in 0-01 (1, 5). Notice that combinations cannot occur among the numbers belonging to the same section. The results of such combinations are entered into another column, sequentially along with their decimal equivalents indicating the binary equivalents from which the result of combination came, like (1, 5) as mentioned above. The second column also will get sectionalised based on the number of 1s. The entries of one section in the second column can again be combined together with entries in the next section, in a similar manner. These combinations are illustrated in the Table below

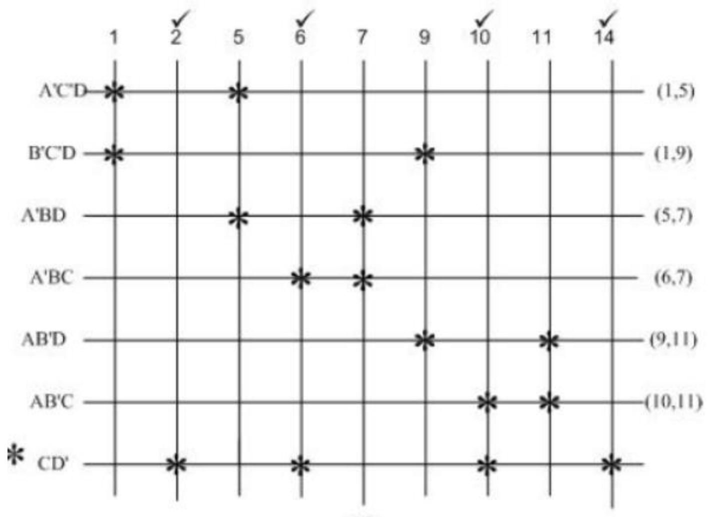
Section	Column 1 No.of 1s Decimal	Binary		Column 2		Column 3	
1	1	0001 ✓	1	1-01	(1,5)	--10	(2,6,10,14)
		0010 ✓	2	-001	(1,9)	--10	~~(2,10,6,14)~~
2	2	0101 ✓	5	0-10	(2,6)✓		
		0110 ✓	6	-010	(2,10) ✓		
		1001 ✓	9				
		1010 ✓	10				
3	3	0111 ✓	7	01-1	(5,7)		
		1011 ✓	11	011-	(6,7)		
		1110 ✓	14	-110	(6,14) ✓		
				10-1	(9,11)		
				101-	(10,11)		
				1-10	(10,14) ✓		

All the entries in the column which are paired with entries in the next section are checked off. Column 2 is again sectionalised with respect t the number of 1s. Column 3 is generated by pairing off entries in the first section of the column 2 with those items in the second section. In principle this pairing could continue until no further combinations can take place. All those entries that are paired can be checked off. It may be noted that combination of entries in column 2 can only take place if the corresponding entries have the dashes at the same place. This rule is applicable for generating all other columns as well.

After the tabulation is completed, all those terms which are not checked off constitute the set of prime implicants of the given function. The repeated terms, like --10 in the column 3, should be eliminated. Therefore, from the above tabulation procedure, we obtain seven prime implicants (denoted by their decimal equivalents) as (1,5), (1,9), (5,7), (6,7), (9,11), (10,11), (2,6,10,14). The next stage is to determine the minimal set of prime implicants.

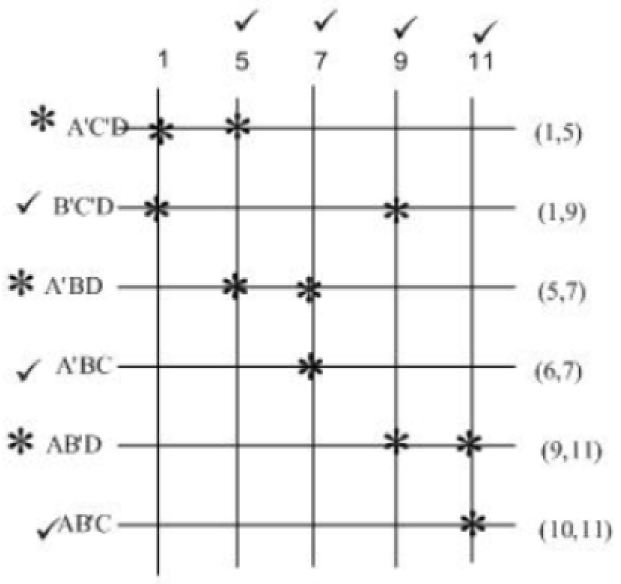
6.2.2 Determination of the Minimal Set of Prime Implicants

The prime implicants generated through the tabular method do not constitute the minimal set. The prime implicants are represented in so called "prime implicant table". Each column in the table represents a decimal equivalent of the minterm. A row is placed for each prime implicant with its corresponding product appearing to the left and the decimal group to the right side. Asterisks are entered at those intersections where the columns of binary equivalents intersect the row that covers them. The prime implicant table for the function under consideration is shown in the figure.



(a)

In the selection of minimal set of implicants, similar to that in a K-map, essential implicants should be determined first. An essential prime implicant in a prime implicant table is one that covers (at least one) minterms which are not covered by any other prime implicant. This can be done by looking for that column that has only one asterisk. For example, the columns 2 and 14 have only one asterisk each. The associated row, indicated by the prime implicant CD/, is an essential prime implicant. CD/ is selected as a member of the minimal set (mark that row by an asterisk). The corresponding columns, namely 2, 6, 10, 14, are also removed from the prime implicant table, and a new table is construction as shown in the figure.



(b)

We then select dominating prime implicants, which are the rows that have more asterisks than others. For example, the row A’BD includes the minterm 7, which is the only one included in the row represented by A’BC. A’BD is dominant implicant over A’BC, and hence A’BC can be eliminated. Mark A’BD by an asterisk and check off the column 5 and 7.

We then choose AB’D as the dominating row over the row represented by AB’C. Consequently, we mark the row AB’D by an asterisk, and eliminate the row AB’C and the columns 9 and 11 by checking them off. Similarly, we select A’C’D as the dominating one over B’C’D. However, B’C’D can also be chosen as the dominating prime implicant and eliminate the implicant A’C’D.

Retaining A’C’D as the dominant prime implicant the minimal set of prime implicants is {CD’, A’C’D, A’BD and AB’D). The corresponding minimal SOP expression for the Boolean function is:

F = CD’ + A’C/D + A’BD + AB’D

If we choose B’C’D instead of A’C’D, then the minimal SOP expression for the Boolean function is:

F = CD’ + B’C’D + A’BD + AB’D

This indicates that if the selection of the minimal set of prime implicants is not unique, then the minimal expression is also not unique. There are two types of implicant tables that have some special properties. One is referred to as cyclic prime implicant table, and the other as semi-cyclic prime implicant table. A prime implicant table is considered to be cyclic if

1 it does not have any essential implicants which implies that there are at least two asterisks in every column, and
2 There are no dominating implicants, which implies that there are same number of asterisks in every row.

Example 2: A Boolean function with a cyclic prime implicant table is shown in the figure 3.

The function is given by

F = Σ (0, 1, 3, 4, 7, 12, 14, 15)

All possible prime implicants of the function are:

a = A’B’C’ (0,1)

b = A’B’D (1,3)

c = A’CD (3,7)

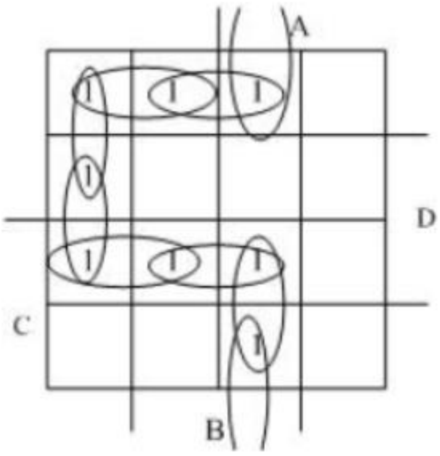
d = BCD (7,15)

e = ABC (14,15)

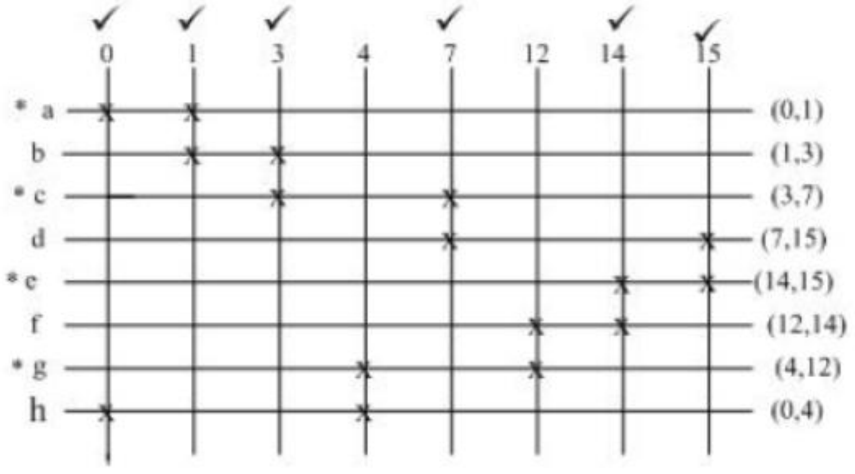
f = ABD’ (12,14)

g = BC’D’ (4,12)

h = A’C’D’ (0,4)

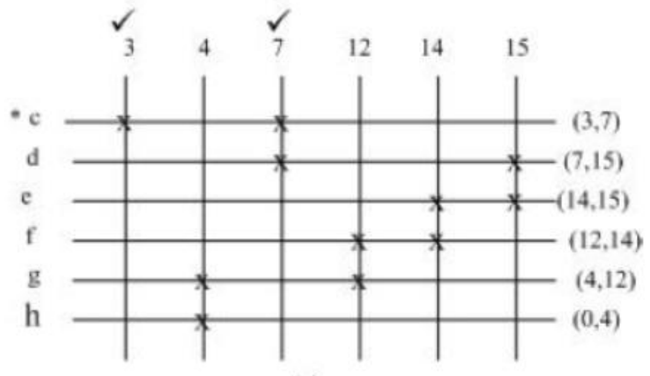


As it may be noticed from the prime implicant table in the figure that all columns have two asterisks and there are no essential prime implicants. In such a case we can choose any one of the prime implicants to start with. If we start with prime implicant a, it can be marked with asterisk and the corresponding columns, 0 and 1, can be deleted from the table. After their removal, row c becomes dominant over row b, so that row c is selected and hence row b is can be eliminated. The columns 3 and 7 can now be deleted. We observe then that the row e dominates row *d*, and row d can be eliminated. Selection of row *e* enables us to delete columns 14 and 15.

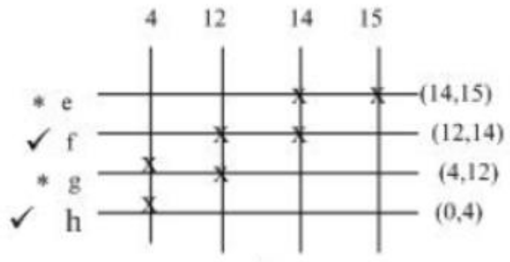


	✓	✓	✓		✓		✓	✓	
	0	1	3	4	7	12	14	15	
* a	X	X							(0,1)
b		X	X						(1,3)
* c			X		X				(3,7)
d					X			X	(7,15)
* e							X	X	(14,15)
f						X	X		(12,14)
* g				X		X			(4,12)
h	X			X					(0,4)

If, from the reduced prime implicant table shown in the figure, we choose row g it covers the remaining asterisks associated with rows h and f. That covers the entire prime implicant table. The minimal set for the Boolean function is given by:



(a)

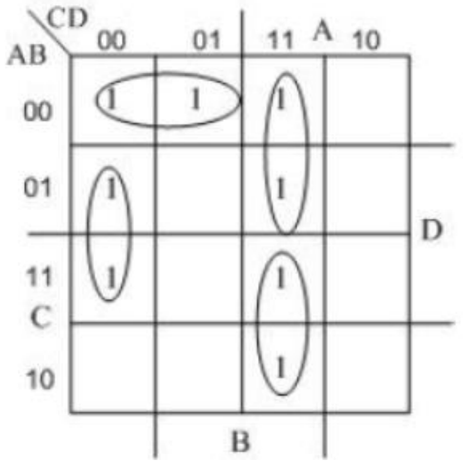


(b)

F = a + c + e + g

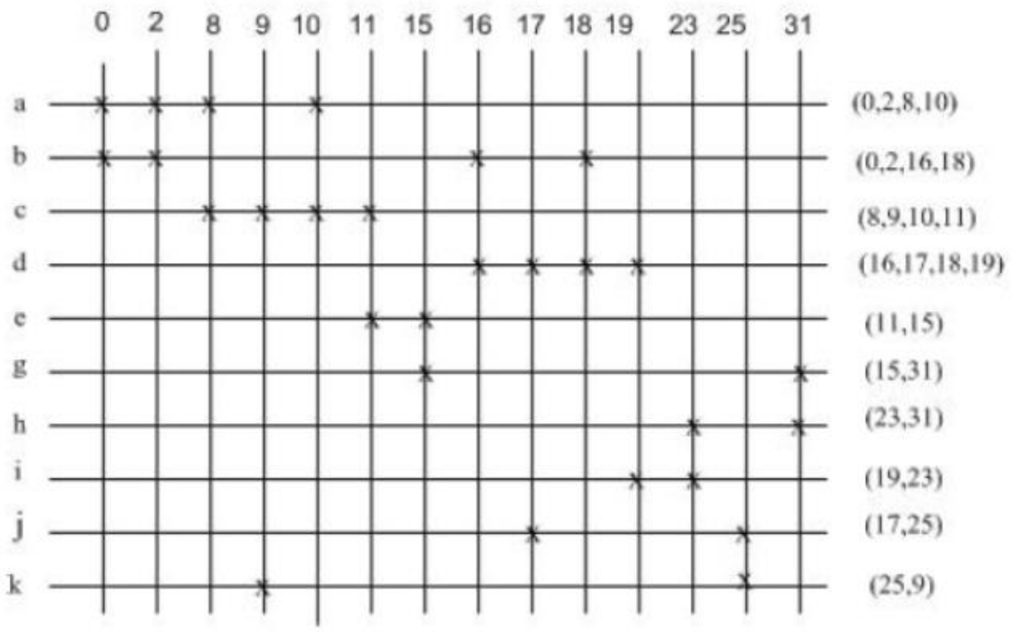
= A'B'C' + A'CD + ABC + BC'D'

The K-map of the simplified function is shown in the following figure



A semi-cyclic prime implicant table differs from a cyclic prime implicant table in one respect. In the cyclic case the number of minterms covered by each prime implicant is identical. In a semi-cyclic function this is not necessarily true.

Example 3: Consider a semi-cyclic prime implicant table of a five variable Boolean function shown in the figure.



	0	2	8	9	10	11	15	16	17	18	19	23	25	31	
a	x	x	x		x										(0,2,8,10)
b	x	x						x		x					(0,2,16,18)
c			x	x	x	x									(8,9,10,11)
d								x	x	x	x				(16,17,18,19)
e						x	x								(11,15)
g							x							x	(15,31)
h												x		x	(23,31)
i											x	x			(19,23)
j									x				x		(17,25)
k				x									x		(25,9)

Examination of the prime-implicant table reveals that rows a, b, c and d contain four minterms each. The remaining rows in the table contain two asterisks each. Several minimal sets of prime implicants can be selected. Based on the procedures presented through the earlier examples, we find the following candidates for the minimal set:

F = a + c + d + e + h + j

or F = a + c + d + g + h + j

or F = a + c + d + g + j + i

or F = a + c + d + g + i + k

Based on the examples presented we may summarise the procedure for determination of the minimal set of implicants:

1 Find, if any, all the essential prime implicants, mark them with *, and remove the corresponding rows and columns covered by them from the prime implicant table.

2 Find, if any, all the dominating prime implicants, and remove all dominated prime implicants from the table marking the dominating implicants with *s. Remove the corresponding rows and columns covered by the dominating implicants.

3 For cyclic or semi-cyclic prime implicant table, select any one prime implicant as the dominating one, and follow the procedure until the table is no longer cyclic or semicyclic.

4 After covering all the columns, collect all the * marked prime implicants together to form the minimal set, and convert them to form the minimal expression for the function.

6.2.3 Simplification of Incompletely Specified functions

The simplification procedure for completely specified functions presented in the earlier sections can easily be extended to incompletely specified functions. The initial tabulation is drawn up including the dont-cares. However, when the prime implicant table is constructed, columns associated with dont-cares need not be included because they do not necessarily have to be covered. The remaining part of the simplification is similar to that for completely specified functions.

Example 4: Simplify the following function:

F(A,B,C,D,E) =Σ(1,4,6,10,20,22,24,26) + d(0,11,16,27)

Tabulation of the implicants

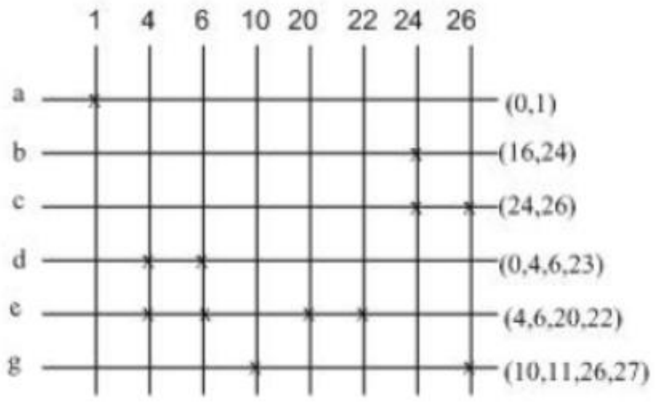
00000 (d)	0 ✓	0000- (0,1)	-0-00 (0,4,16,20)
		00-00 (0,4) ✓	-0-00 (~~0,16,4,20~~)
00001	1 ✓	-0000 (0,16) (d)	
00100	4 ✓		-01-0 (4,6,20,22)
10000 (d)	16 ✓	001-0 (4, 6) ✓	-01-0 (~~4,20,6,22~~)
		-0100 (4,20) ✓	
00110	6 ✓	10-00 (16,20) ✓	-101- (10,26,11,27)
01010	10 ✓	1-000 (16,24) ✓	-101- (~~10,11,26,27~~)
10100	20 ✓		
11000	24 ✓	-0110 (6,22) ✓	
		-1010 (10,26) ✓	
10110	22 ✓	0101- (10,11) ✓	
11010	26 ✓	101-0 (20,22) ✓	
01011 (d)	11 ✓	110-0 (24,26)	
11011 (d)	27 ✓	1101- (26,27) ✓	
		-1011 (11,27) ✓	

Pay attention to the don’t-care terms as well as to the combinations among themselves, by marking them with (d).

Six binary equivalents are obtained from the procedure. These are 0000- (0,1), 1-000 (16,24), 110-0 (24,26), -0-00 (0,4,16,20), -01-0 (4,6,20,22) and -101- (10,11,26,27) and they correspond to the following prime implicants:

a = A’B’C’D’/	b = AC’D’E’	c = ABC’E’
d = B’D’E’	e = B’CE’	g = BC’D

The prime implicant table is plotted as shown in the figure.



It may be noted that the don’t-cares are not included.

The minimal expression is given by:

F(A,B,C,D,E) = a + c + e + g

= A'B'C'D' + ABC'E' + B'CE' + BC'D

Check Your Progress

1. Minimise the function below using the tabular method of simplification:
   Z = f(A,B,C,D) = $\bar{A}\bar{B}\bar{C}\bar{D} + \bar{A}\bar{B}C\bar{D} + A\bar{B}CD + A\bar{B}C\bar{D} + \bar{A}BCD + \bar{A}BC\bar{D} + \bar{A}\bar{B}CD$
2. Using the tabular method of simplification, find all equally minimal solutions for the function below.
   Z = f(A,B,C,D) = $\sum$ (1,4,5,10,12,14

3. Minimize the function f(A, B, C, D) = $\sum$ (0,1,2,3,5,7,8,10,12,13,15), using tabular method.

6.3 Model Question

1. What is are the limitations of K-map?
2. Describe the Principle of Quine-McClusky Method.
3. Explain the process of generating prime implicants through an example.

Answers to Check Your Progress

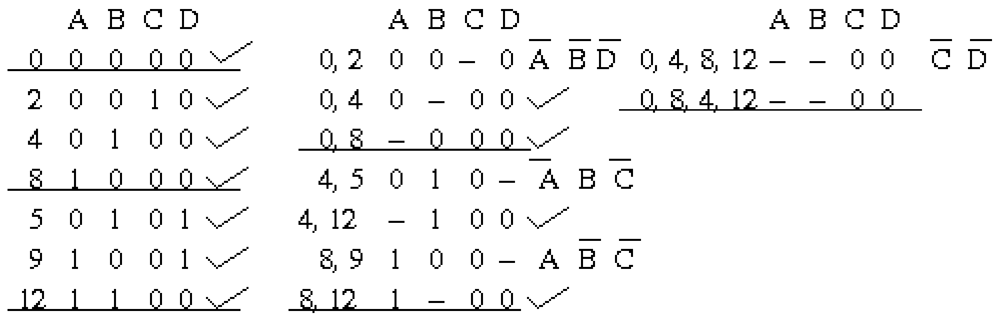
1. Consider the function: Z = f(A,B,C,D) = $\bar{A}\bar{B}\bar{C}\bar{D} + \bar{A}\bar{B}C\bar{D} + A\bar{B}CD + A\bar{B}C\bar{D} + \bar{A}BCD + \bar{A}BC\bar{D} + \bar{A}\bar{B}CD$

Convert to decimal and binary equivalents:-

Z = f(A,B,C,D) = $\sum$ (0,2,4,5,8,9,12) - decimal equivalent

Z = f(A,B,C,D) = $\sum$ (0000,0010,0100,0101,1000,1001,1100) - binary equivalent

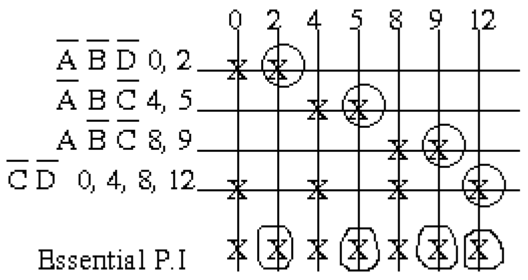
(0, 1, 1, 2, 1, 2, 2) - index values



	A	B	C	D	
0	0	0	0	0	✓
2	0	0	1	0	✓
4	0	1	0	0	✓
8	1	0	0	0	✓
5	0	1	0	1	✓
9	1	0	0	1	✓
12	1	1	0	0	✓

	A	B	C	D	
0, 2	0	0	–	0	$\bar{A}\bar{B}\bar{D}$
0, 4	0	–	0	0	✓
0, 8	–	0	0	0	✓
4, 5	0	1	0	–	$\bar{A}B\bar{C}$
4, 12	–	1	0	0	✓
8, 9	1	0	0	–	$A\bar{B}\bar{C}$
8, 12	1	–	0	0	✓

	A	B	C	D	
0, 4, 8, 12	–	–	0	0	$\bar{C}\bar{D}$
0, 8, 4, 12	–	–	0	0	



	0	2	4	5	8	9	12
$\bar{A}\bar{B}\bar{D}$ 0, 2	X	Ⓧ					
$\bar{A}B\bar{C}$ 4, 5			X	Ⓧ			
$A\bar{B}\bar{C}$ 8, 9					X	Ⓧ	
$\bar{C}\bar{D}$ 0, 4, 8, 12	X		X		X		Ⓧ
Essential P.I	X	$\boxed{X}$	X	$\boxed{X}$	X	$\boxed{X}$	$\boxed{X}$

The simplified answer is: $Z = \bar{A}\bar{B}\bar{D} + \bar{A}B\bar{C} + A\bar{B}\bar{C} + \bar{C}\bar{D}$

2

$f(A, B, C, D) = \sum(0, 1, 4, 5, 10, 12, 14) = \sum(0000, 0001, 0100, 0101, 1010, 1100, 1110)$

$= \text{Index: } 0 \quad 1 \quad 1 \quad 2 \quad 2 \quad 2 \quad 3$

	A	B	C	D	
0	0	0	0	0	✓
1	0	0	0	1	✓
4	0	1	0	0	✓
5	0	1	0	1	✓
10	1	0	1	0	✓
12	1	1	0	0	✓
14	1	1	1	0	✓

	A	B	C	D	
0, 1	0	0	0	–	✓
0, 4	0	–	0	0	✓
1, 5	0	–	0	1	✓
4, 5	0	1	0	–	✓
4, 12	–	1	0	0	$B\bar{C}\bar{D}$
10, 14	1	–	1	0	$AC\bar{D}$
12, 14	1	1	–	0	$AB\bar{D}$

	A	B	C	D	
0, 1, 4, 5	–	–	0	0	$\bar{A}\bar{C}$
0, 4, 1, 5	–	–	0	0	

	0	1	4	5	10	12	14
$\bar{A}\bar{B}\bar{D}$ 0, 2			X			X	
$\bar{A}B\bar{C}$ 4, 5					Ⓧ		X
$A\bar{B}\bar{C}$ 8, 9						X	X
$\bar{C}\bar{D}$ 0, 4, 8, 12	Ⓧ	X	X	X			
Essential P.Is	X	X	X	X	X		X

Note that the remaining term 12 is covered by $B\bar{C}\bar{D}$ and $AB\bar{D}$

One simplified answer is: $Z = \bar{A}\bar{C} + AC\bar{D} + B\bar{C}\bar{D}$
Another answer is: $Z = \bar{A}\bar{C} + AC\bar{D} + AB\bar{D}$

3 Note that this is in decimal form.

$\sum$ (0000,0001,0010,0011,0101,0111,1000,1010,1100,1101,1111) in binary form.

(0,1,1,2,2,3,1,2,2,3,4) in the index form.

First List

	A	B	C	D	
0	0	0	0	0	✓
1	0	0	0	1	✓
2	0	0	1	0	✓
8	1	0	0	0	✓
3	0	0	1	1	✓
5	0	1	0	1	✓
10	1	0	1	0	✓
12	1	1	0	0	✓
7	0	1	1	1	✓
13	1	1	1	1	✓
15	1	1	1	1	✓

Second List

	A	B	C	D	
0,1	0	0	0	–	✓
0,2	0	0	–	0	✓
0,8	–	0	0	0	✓
1,3	0	0	–	1	✓
1,5	0	–	0	1	✓
2,3	0	0	1	–	✓
2,10	–	0	1	0	✓
8,10	1	0	–	0	✓
8,12	1	–	0	0	$A\bar{C}\bar{D}$
3,7	0	–	1	1	✓
5,7	0	1	–	1	✓
5,13	–	1	0	1	✓
12,13	1	1	0	–	$AB\bar{C}$
7,15	–	1	1	1	✓
13,15	1	1	–	1	✓

Third List

	A	B	C	D	
0, 1 2 3	0	0	–	–	$\bar{A}\bar{B}$
0, 2 1 3	0	0	–	–	
0, 2, 8, 10	–	0	–	0	$\bar{B}\bar{D}$
0, 8, 2, 10	–	0	–	0	
1, 3, 5, 7	0	–	–	1	$\bar{A}D$
1, 5, 3, 7	0	–	–	1	
5, 7, 13, 15	–	1	–	1	BD
5, 13, 7, 15	–	1	–	1	

The prime implicants are: $\bar{A}\bar{B} + \bar{B}\bar{D} + \bar{A}D + BD + A\bar{C}\bar{D} + AB\bar{C}$

The chart is used to remove redundant prime implicants. A grid is prepared having all the prime implicants listed at the left and all the minterms of the function along the top. Each minterm covered by a given prime implicant is marked in the appropriate position.

	0	1	2	3	5	7	8	10	12	13	15
$\bar{A}\bar{B}$	X	X	X	X							
$\bar{B}\bar{D}$	X		X				X	Ⓧ			
$\bar{A}D$		X		X	X	X					
BD					X	X				X	Ⓧ
$A\bar{C}\bar{D}$							X		X		
$AB\bar{C}$									X	X	
Essential	X		X		X	X	X	Ⓧ		X	Ⓧ

From the above chart, BD is an essential prime implicant. It is the only prime implicant that covers the minterm decimal 15 and it also includes 5, 7 and 13. $\bar{B}\bar{D}$ is also an essential prime implicant. It is the only prime implicant that covers the minterm denoted by decimal 10 and it also includes the terms 0, 2 and 8. The other minterms of the function are 1, 3 and 12. Minterm 1 is present in $\bar{A}\bar{B}$ and $\bar{A}$D. Similarly for minterm 3. We can therefore use either of these prime implicants for these minterms. Minterm 12 is present in A$\overline{C}\,\overline{D}$ and AB$\overline{C}$, so again either can be used.

Thus, one minimal solution is: Z = $\overline{B}\,\overline{D}$ + BD + $\overline{A}\,\overline{B}$ + A$\overline{C}\,\overline{D}$

UNIT VII: LOGIC GATES

7.0 Learning Objectives

After the completion of this unit, you will be able to:

- learn about logic gates.
- draw different logic gates like AND, OR, NOT, NAND, NOR, XOR, XNOR.
- represent the output of logic gates in truth table.
- describe the DeMorgan's theorem.
- learn about the conversion of logic gates.

7.1 Introduction

This unit provides a description of logic gates and defines a few of the common logic gates found in simple digital circuits[1]. Computers work on an electrical flow where a high voltage is considered a 1 and a low voltage is considered a 0. Using these highs and lows, data are represented. Electronic circuits must be designed to manipulate these positive and negative pulses into meaningful logic. Logic gates are the building blocks of digital circuits. Combinations of logic gates form circuits designed with specific tasks in mind. For example, logic gates are combined to form circuits to add binary numbers (adders), set and reset bits of memory (flip-flops), multiplex multiple inputs, etc.

7.2 Logic Gates

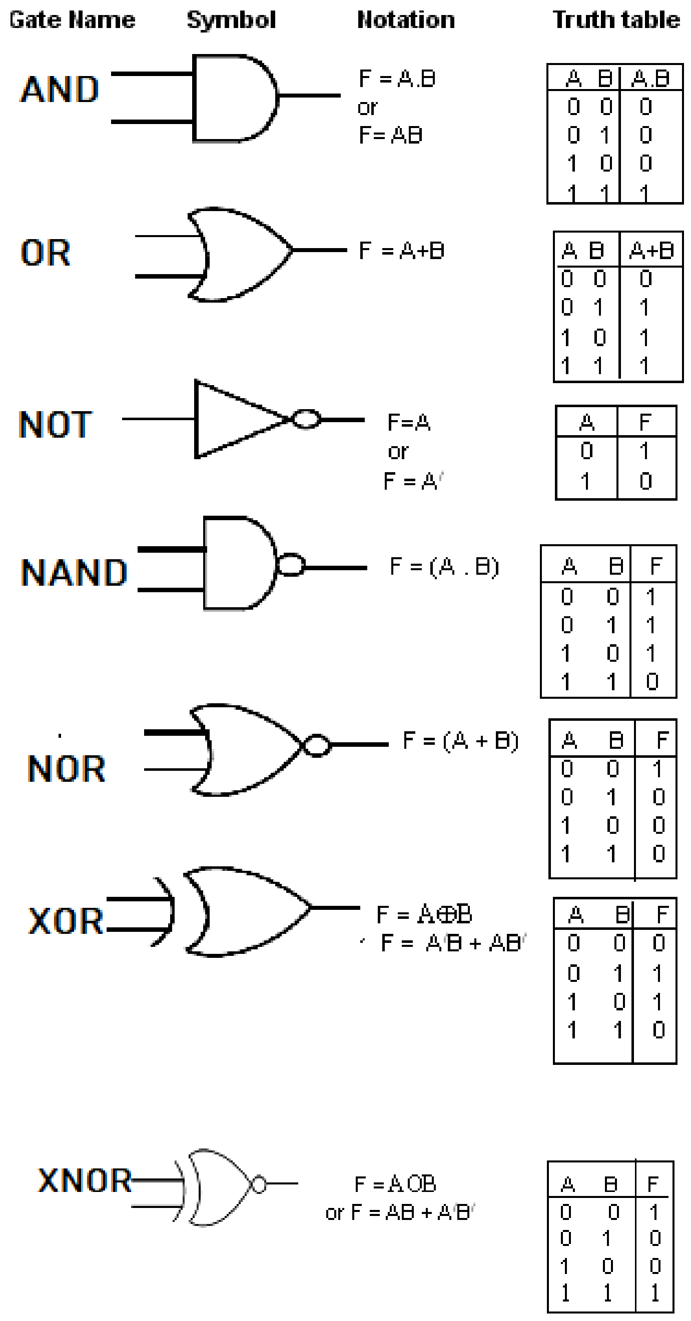
A gate is an electronic circuit which produces a typical output signal depending on its input signal. The output signal of a gate is single Boolean operation of its input signal. Gates are the basic logic elements. Any boolean function can be represented in the form of gates.

In general, we can represent gates in three ways. These are:

[1] Adopted from http://m.kkhsou.in/EBIDYA/CSC/MODIFY_logic_gates.html

- Graphical symbols
- Algebraic notation
- Truth table

Different types of logic gates with their graphical symbol, algebraic notion and their truth table are summarized below:



AND: $F = A.B$ or $F = AB$

A	B	A.B
0	0	0
0	1	0
1	0	0
1	1	1

OR: $F = A+B$

A	B	A+B
0	0	0
0	1	1
1	0	1
1	1	1

NOT: $F=A$ or $F = A'$

A	F
0	1
1	0

NAND: $F = (A . B)$

A	B	F
0	0	1
0	1	1
1	0	1
1	1	0

NOR: $F = (A + B)$

A	B	F
0	0	1
0	1	0
1	0	0
1	1	0

XOR: $F = A \oplus B$, $F = A'B + AB'$

A	B	F
0	0	0
0	1	1
1	0	1
1	1	0

XNOR: $F = A \odot B$ or $F = AB + A'B'$

A	B	F
0	0	1
0	1	0
1	0	0
1	1	1

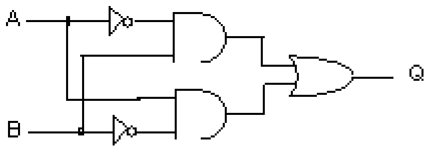
7.3 Truth Table

A truth table is a good way to show the behavior of logic gates. It shows the output states for every possible combination of input states. The symbols 0 (false) and 1 (true) are usually used in truth tables. They show how the input(s) of a logic gate relate to its output(s).

A truth table for two input of an OR gate is shown below, but it can b extended to any number of inputs. The gate input(s) are shown in the left column(s) of the table with all the different possible input combinations. This is normally done by making the inputs count up in binary. The gate output(s) are shown in the right-hand side column.

Input A	Input B	Output Q
0	0	0
0	1	1
1	0	1
1	1	1

Example: Draw the truth table for the following logic circuit and find the output expression Q.



Solution: The above logic circuit gives the following truth table and output expression

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

A	B	A'	B'	$A'.B$	$A.B'$	$A'.B+A.B'$
0	0	1	1	0	0	0
0	1	1	0	1	0	1
1	0	0	1	0	1	1
1	1	0	0	0	0	0

7.4 AND Gate

An AND gate has two or more inputs but it has only one output. An input signal applied to a gate has only two stable states, either 1 (HIGH) or 0 (LOW). There is no intermediate state. In case of a 2-input AND gate the output is 1 (high) only if both inputs are 1 (high), otherwise the output is 0 (low). The logical operation of an AND gate is called the AND operation (or AND function).

The logical AND operation can easily be explained with the help of two switches connected in series, Figure 2 The current will flow in the circuit only when both switches, A and B, are closed. A switch has two stable states, ON and OFF. The ON state is taken as logic 1, the OFF as logic 0. When both switches are ON, i.e., A and B both have logic1, the output is 1. When any one of the swithces is OFF, there is no output (no current in the circuit). The output is taken as 0.

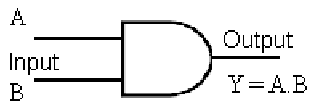


Figure 1: AND Gate

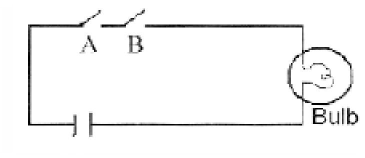


Figure 2: A circuit containing two switches in series

Figure 1 shows standard graphical symbol for a two-input AND gate. It shows only two inputs, A and B. If there are more than two inputs, they will be shown on input side. Table 1 shows the truth table for a two-input AND gate. The output of an AND gate is 1 only when all the inputs are 1. If any one of the inputs is 0, the output will be 0. Therefore, the output of an AND gate is equal to the product of inputs i.e., the output Y = A.B. The symbol ^ is also used to denote AND operation. A ^ B denotes AND operations of A and B.

Table 1: Truth table for AND gate

INPUT		OUTPUT
A	B	Y=A.B
0	0	0
0	1	0
1	0	0
1	1	1

The output of an AND gate is Y = A.B. In this case A and B (inputs) are independent variables. Y (output) is a dependent variable. The table which shows the values of dependent variable for all possible values of independent variables is called truth table.

7.5 OR-Gate

An OR gate is also known as INCLUSIVE-OR gate. It has two or more inputs, but only one output. The output of an OR gate will be high if at least any one of the inputs is high. The output will be 0 only when all inputs are 0. This logical operation will be called OR operation (or OR function). Figure 3 shows a two input OR gate. Its truth table has been shown in Table 3.2. Its output is given by Y = A + B. The logical OR operation can easily be explained taking an example of two switches connected in parallel as shown in Figure 4. The current will flow in the circuit when either switch is in ON position (i.e., logic 1). The current will not flow at all when both switches are in OFF position (i.e. logic 0). The ^(DOWN) symbol denotes OR operation. A ^(DOWN) B denotes OR operation of A and B.

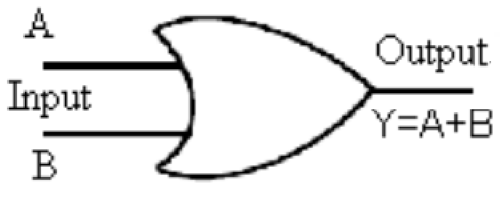


Figure 3: Graphical symbol for OR gate

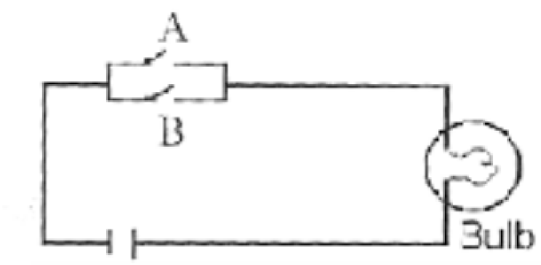


Figure 4: A circuit containing two switches in parallel

Table 2: Truth table of OR gate

INPUT		**OUTPUT**
A	B	Y=A+B
0	0	0
0	1	1
1	0	1
1	1	1

7.6 NOT Gate

A NOT gate has only one input, and only one input signal. It is also called INVERTER . Its output is the complement of the input signal. The output is 1 (HIGH) if the input is 0 (Low). The output is 0 when input is 1.Its output is given by Y = A. NOT gate are used to invert the logic levels i.e. from LOW to HIGH or HIGH to LOW when required. Figure 5 shows standard symbol for a NOT gate.

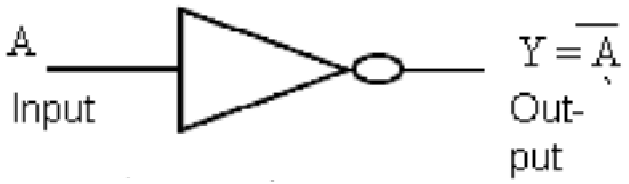


Figure 5: NOT Gate

Table 3: Truth table of NOT gate

INPUT	**OUTPUT**
A	Y=$\overline{A}$
0	1
1	0

7.7 NAND Gate

A NAND gate has two or more inputs but only one output Figure 6vshows a two-input NAND gate in two different ways. The NAND function is the complement of the AND function. An AND gate can be combined with an INVERTER to form a NAND gate. The abbreviation NAND is the short form of NOT-AND. Its output is given by **Y = A.B** . Table 4 shows the truth table for a 2-input NAND gate.

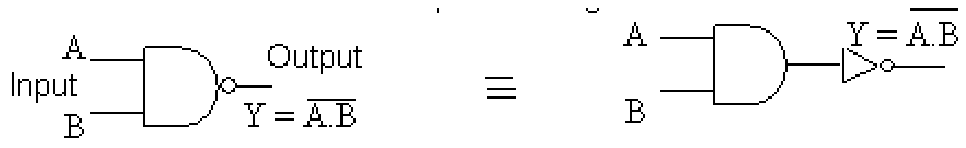


Figure 6: Graphical symbol of NAND gate

Table 4: Truth table of NAND gate

INPUT		**OUTPUT**
A	B	Y= $\overline{A}.\overline{B}$
0	0	1
0	1	1
1	0	1
1	1	0

The NAND operations is sometimes called universal operations because other logic operation can be realized by using only NAND gates. A universal operation is one that can be used to implement the three basic logic operations AND, OR and NOT. It is easier to fabricate NAND and NOR gates employing modern IC technology compared to AND and OR gates. NAND and NOR gates also consume less power. Therefore, combinational logic networks are realized by using only NAND gates . NAND gates are used as basic building blocks in fabricating a digital circuit.

7.8 NOR Gate

A NOR gate has two or more inputs but only one output. Figure 7 shows a two input NOR gate. The NOR function is the complement of OR function. An OR gate can be combined with an INVERTER to form a NOR gate as shown in the figure. The abbreviation NOR is the short form of NOT- OR. Its output is given by Y = A + B. Table 5 shows the truth table for a 2-input NOR gate.

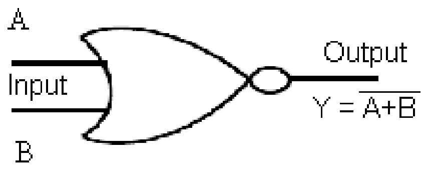


Figure 7: NOR gate

The NOR operation is also a Universal operation. Other logic gates such as AND, OR and NOT can be realized using NOR gates. Combinational logic network can be realized by using only NOR gates. NOR gates are used as building blocks in fabricating a digital network. NOR gates are available in IC forms.

Table 5: Truth Table for NOR Gate

INPUT		**OUTPUT**
A	B	$Y = \overline{A+B}$
0	0	1
0	1	0
1	0	0
1	1	0

7.9 X-OR Gate

The output of Exclusive-OR (XOR) gate is high only when its inputs are different. Its output is low when both inputs A and B are same i.e., either both are high or both are low. Its output is given by:

$$
\begin{aligned}
Y &= A \oplus B \\
&= \underline{A} \oplus \underline{B} \\
&= \underline{A}B + A\underline{B}
\end{aligned}
$$

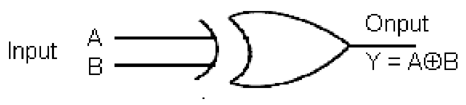


Figure 8: Exclusive-OR Gate

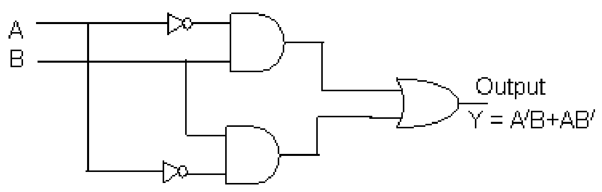


Figure 9: Logic diagram for 2 input Exclusive-OR gate

The XOR operation is also called Modulo-2-Sum operation. Figure 8 and Figure 9 shows the graphical symbol and logic diagram for a 2-input XOR gate. Table 6 shows its truth table.

Table 6: Truth table for XOR gate

INPUT		**OUTPUT**
A	B	$Y = A \oplus B$ $= \overline{AB} + A\overline{B}$
0	0	0
0	1	1
1	0	1
1	1	0

An XOR operation can be distinguished from OR functions as follows: A OR B means A or B, or both, whereas A XOR B means A or B but not both.

7.10 X-NOR Gate

The Exclusive-NOR (XNOR) operation is the complement of XOR operation. The output of XNOR is high only when the logic values of both inputs A and B are same i.e., either both are1 or both are 0. Its output is 0 when its inputs are different. Table 7 shows its truth table. The output is given by:

$$\begin{aligned} Y &= \overline{A \oplus B} \\ &= AB + \overline{A}\,\overline{B} \end{aligned}$$

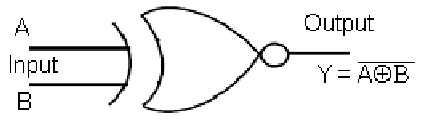


Figure 10: Graphical symbol for XNOR gate

Figure 10 shows a two-input XNOR gate. The 74135 is an XOR/XNOR gate. It has three inputs A .B and C. If C input is low it operates as an XOR gate. If C input is high it operates as XNOR gate.

Table 7: Truth Table for XNOR Gate

INPUT		**OUTPUT**
A	B	$Y = A \oplus B$ $= AB + \overline{A}\,\overline{R}$
0	0	1
0	1	0
1	0	0
1	1	1

7.11 Summary

- A logic gate is an electronic circuit that operates on one or more input signals to produce standard output signals. These logic gates are the building blocks of all electronic circuits in computer.

- To represent different logic gates, specific shapes are assigned.These are called graphical symbols.
- Logic gates are operated with binay numbers ‘0’ and ‘1’. In analog terminology, ‘0’ is termed as ‘LOW’ (ground) voltage and ‘1’ is termed as ‘HIGH’ voltage (>= 4.5 volt.)
- A truth table is a good way to show the behaviour of logic gates. It shows the output states for every possible combination of input states.
- An AND gate is the physical realization of logical multiplication (AND) operation. It is an electronic .circuit that generates an output signal of 1, only if all input signals are also 1.
- An OR gate is the physical realization of logical addition (OR) operation. It is an electronic circuit that generates an output signal of 1, if any of the l input signals is also 1.
- An NOT gate is the physical realization of complementation operation. It is an electronic circuit that generates the reverse of the input signal as output signal. It is also known as an inverter because it inverts the input.
- A NAND gate is a complemented AND gate. That is, the output of NAND gate will be 1 if any one of the inputs is a 0, and will be 0 only when all inputs are 1.
- A NOR gate is a complemented OR gate. That is, the output of NOR gate will be 1only when all inputs are 0, and it will be 0 if any input is a 1.
- Exclusive-OR and Exclusive -NOR are two important logic component.
- DeMorgan’s theorem states that the complement of a sum of variables is equal to the product of the complements of the variables.

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

  and the complement of a product of variables is equal to the sum of the complements of the variables.

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

Check Your Progress

1. A ______ is an electronic circuit which produces a typical output signal depending on its input signal.
2. An OR gate is also known as ___________ gate.
3. A NOT gate is also called ___________________.

7.12 Model Questions

1. What is logic gates? Draw the block diagram of a 3-input AND gate. Give its truth table.
2. Draw the graphical symbol and truth table for a 2-input XOR gate.
3. Draw the logic diagram and graphic symbol of a 2-input Exclusive- OR gate.
4. Define and describe DeMorgan’s Theorem.

Answers to Check Your Progress

1. gate
2. INCLUSIVE-OR
3. INVERTER

UNIT VIII: COMBINATIONAL CIRCUITS

8.0 Learning Objectives

After reading this chapter, you will be able to:

- Define a combinational circuit
- Explain the functioning of a multiplexer and a demultiplexer.
- Explain the functioning of an Encoder and a Decoder.
- Simplify a given Boolean expression using Karnaugh map
- Explain the working an a half-adder and a full-adder.

8.1 Combinational Circuit

A combinational circuit is a collection of some interconnected set of logic gates whose outputs at any time are determined directly from the present combination of inputs without regard to previous inputs[2].

A combinational circuit consists of input variables, logic gates, and output variables. The logic gates accept binary signals from the inputs and generate some binary signals to the outputs. Here binary signals mean only two possible values, one representing logic-1 and the other logic-0. A combinational circuit can be represented as a network of gates and, therefore, can be expressed by a truth table or boolean expression. A block diagram of combinational circuit is shown below:

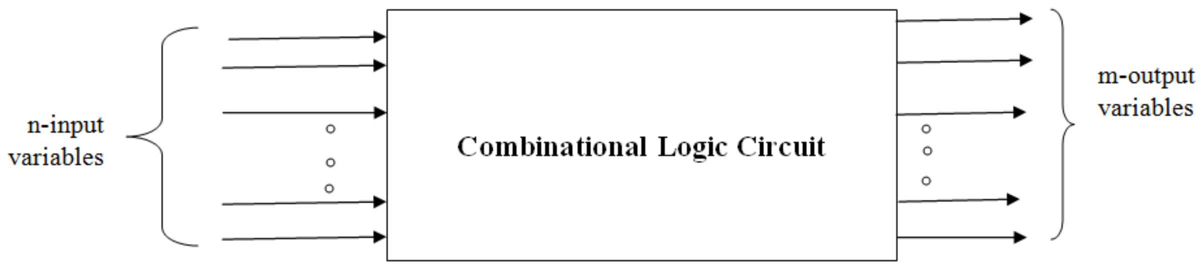


Figure 11: Block diagram of combinational circuit

[2] Adapted from http://m.kkhsou.in/EBIDYA/CSC/MODIFY_combinational_sequential.html

Here n input binary variables come from an external source and the m output variables go to an external destination. For n input variables, there are 2^n possible combinations of binary input values. For each possible input combination, there is one and only one possible output combination. A combinational circuit can be described by m Boolean functions, one for each output variable. Each output function is expressed in terms of the n input variables.

The basic design issue related to combinational circuits is: the minimization of number of gates. The normal circuit constraints for combinational circuit design are:

1. The depth of the circuit should not exceed a specific level.
2. Number of inputs to a gate and to how many gates its output can be fed.

8.2 Multiplexer and Demultiplexer

8.2.1 Multiplexer

The **multiplexer** is a circuit whose output is one of several inputs, depending on the value given by some other selection inputs. In a circuit, a multiplexer is drawn as a trapezoid as follows.

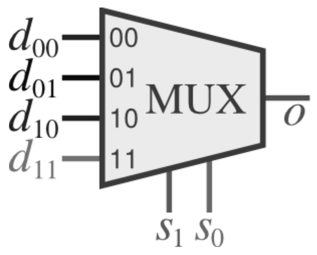


Figure 12: Multiplexer

This is a *4-to-1 multiplexer*. On the left side, you can see four data inputs, which we'll call d_{00}, d_{01}, d_{10}, and d_{11}. On the bottom, you can see two selection inputs, which we'll call s_1 and s_0. And on the right, you can see the multiplexer's output. The truth table of multiplexer is shown below:

Table 8: Truth table for multiplexer

Select Lines		Output
S_1	S_0	O
0	0	d_{00}
0	1	d_{01}
1	0	d_{10}
1	1	d_{11}

What does a multiplexer do? It routes one of the data inputs to the multiplexer's output; which of the data inputs goes out is determined based on the select inputs. For example, if s_1 is 1 and s_0 is 0, then the d_{10} input is routed to the output. If instead s_1 is 0 and s_0 is 1, then the circuit's output would be d_{01} instead. It's helpful to think of this a multiplexer as a railroad switch, connecting one input to an output; the selection inputs control the angle of the switch.

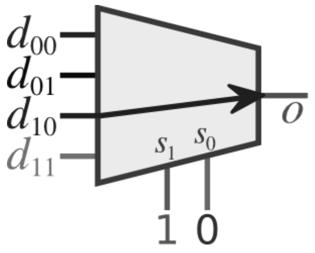


Figure 13: Path selection using line select

While a railroad switch is helpful in thinking about how a multiplexer works, circuits don't actually have moving parts. If we're going to build a multiplexer, then we can only use wires and logic gates to build it. Our solution will be to include an AND gate for each of the data inputs, and to have each of the AND gates feed into an OR gate. The AND gate corresponding to a data input will be 0 whenever the selection inputs don't correspond to the data input; that way, the AND gate corresponding to an unselected data input doesn't affect the OR gate's output, and so the OR gate's output will match the output of the AND gate whose data input is selected. We include the data input on each corresponding AND gate so that this AND gate's output will match the data input when that data input is selected. Here's the circuit diagram.

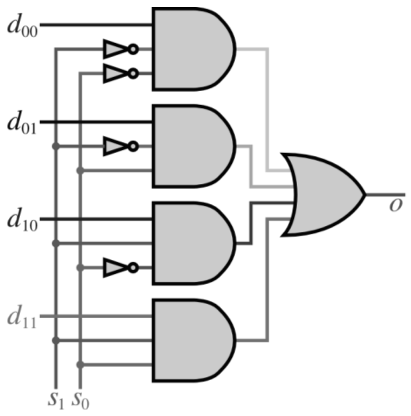


Figure 14: Circuit diagram of 4X1 multiplexer

This particular multiplexer is a *4-to-1 multiplexer*, so called because it routes one of four inputs to a single output. Other common multiplexer ratios are 2-to-1, which would have one selection bit; 8-to-1, with three selection bits; a 16-to-1, with four selection bits; and even 32-to-1, with five selection bits.

8.2.2 Demultiplexer

The **demultiplexer** reverses the process of a multiplexer: It routes a single input into one of several outputs, while the unselected outputs should emit a value of 0. We represent it in a larger circuit as follows.

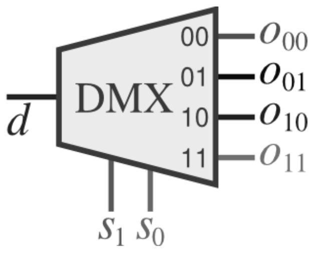


Figure 15: Demultiplexer

Internally, its circuit is similar to that of a multiplexer's, though it is simpler because each output is treated independently. The truth table for the 1X4 demultiplexer is shown below:

Table 9: Truth table for 1x4 demultiplexer

Data Input	Select Inputs		Outputs			
d	S_1	S_0	O_{11}	O_{10}	O_{01}	O_{00}
d	0	0	0	0	0	d
d	0	1	0	0	d	0
d	1	0	0	d	0	0
d	1	1	d	0	0	0

Below is an implementation diagram.

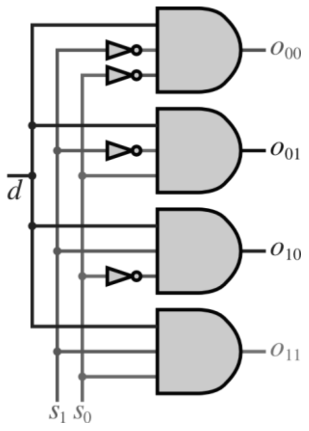


Figure 16: Circuit diagram of 1x4 demultiplexer

This is a 1-to-4 demultiplexer. Other common ratios are 1-to-2, 1-to-8, 1-to-16, and 1-to-32; the number of selection bits will vary depending on how many are required to indicate which output to use.

8.3 Encoder and Decoder

8.3.1 Encoder

An encoder is a combinational circuit that produces the reverse function from that of a decoder. An encoder consists of 2^n(or less) input lines and n output lines. The output lines generate the binary code for the 2^n input variables.

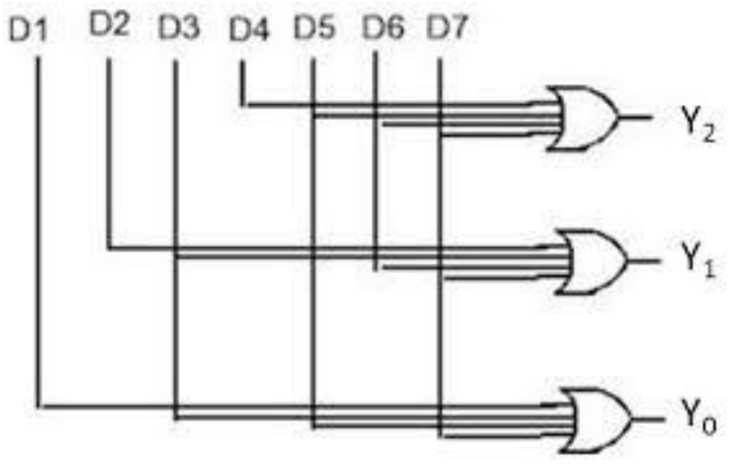


Figure 17: Logic diagram of octal to binary encoder

An octal to binary encoder logic circuit is shown in Figure 17. It has eight inputs, one for each of the eight digits, and three outputs that generate the corresponding binary number. It is constructed with OR gates and its inputs can be determined from the truth table given in Table 10.

Table 10: Truth table of octal to binary encoder

Octal Number	**Inputs**								**Outputs**		
	D0	D1	D2	D3	D4	D5	D6	D7	A	B	C
0	1	0	0	0	0	0	0	0	0	0	0
1	0	1	0	0	0	0	0	0	0	0	1
2	0	0	1	0	0	0	0	0	0	1	0
3	0	0	0	1	0	0	0	0	0	1	1
4	0	0	0	0	1	0	0	0	1	0	0
5	0	0	0	0	0	1	0	0	1	0	1
6	0	0	0	0	0	0	1	0	1	1	0
7	0	0	0	0	0	0	0	1	1	1	1

Here we assume that only one input line can be equal to 1 at any time. In this case the circuit has eight inputs and could have $2^8 = 256$ possible input combinations. Only eight of these combinations have any meaning. The other input combinations are “don’t care conditions”. The type of encoder available in Integrated Circuits form is called a priority encoder. In these types of encoders, only the highest priority input line is encoded and so an input priority is established.

8.3.2 Decoder

A decoder is a combinational logic circuit that receives coded information on n input lines and feeds them to maximum of 2^n unique output lines after conversion. But the decoder output will have less than 2^n outputs when the n-bit decoded information is unused or don't care combinations.

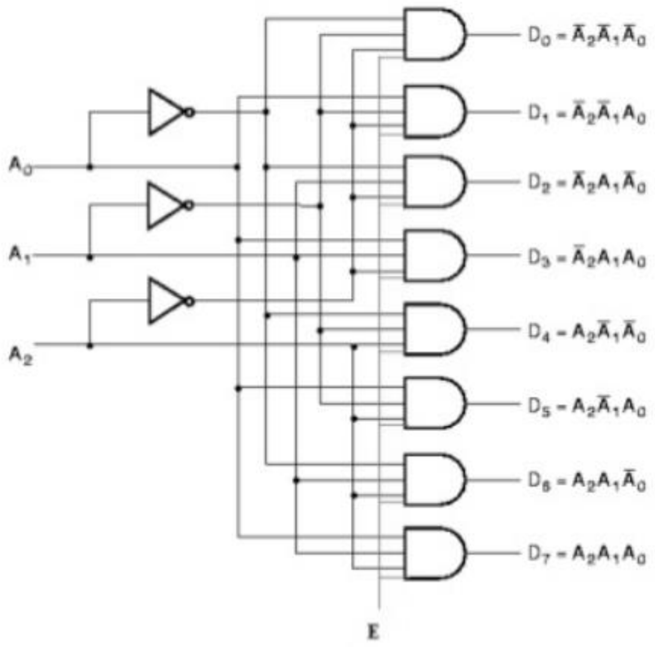

A_0
A_1
A_2
E
$D_0 = \bar{A}_2\bar{A}_1\bar{A}_0$
$D_1 = \bar{A}_2\bar{A}_1A_0$
$D_2 = \bar{A}_2A_1\bar{A}_0$
$D_3 = \bar{A}_2A_1A_0$
$D_4 = A_2\bar{A}_1\bar{A}_0$
$D_5 = A_2\bar{A}_1A_0$
$D_6 = A_2A_1\bar{A}_0$
$D_7 = A_2A_1A_0$

Figure 18: Logic diagram of 3-to-8 decoder

A 3-to-8 line decoder is shown in Figure 18. Here there are 3 input lines and 8 output lines. The 3 inputs are decoded into eight outputs, each output representing one of the minterms of the 3 input variables. The three inverters provide the complement of the inputs, and each one of the eight AND gate generates one of minterms. A 3-to-8 line decoder can be used for decoding any 3-bit code to provide eight outputs, one for each element of the code.

The decoder is used in some code converters, for example BCD- to- seven segment decoder. A 3-to-8 line decoder can be used in binary to octal conversion. Another use of a decoder is that it can be used to implement any combinational circuit.

Table 11: Truth table for 3-to-8 decoder

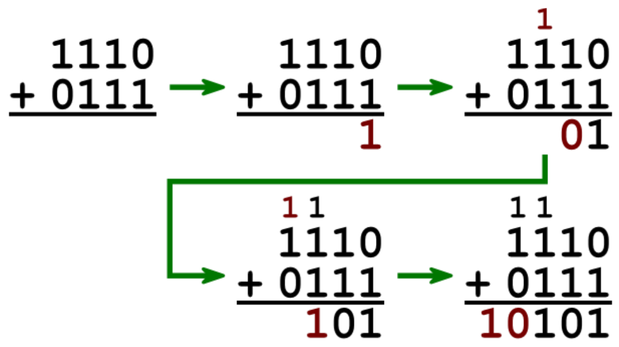
A_2	A_1	A_0	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
0	0	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	1	0	0	0	0	0	0	1	0	0
0	1	1	0	0	0	0	1	0	0	0
1	0	0	0	0	0	1	0	0	0	0
1	0	1	0	0	1	0	0	0	0	0
1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0

8.4 Half adder and Full adder

8.4.1 Half Adder

Let's begin the discussion by considering the basic adding process — how would we add two numbers? Of course, you already know how to add base-10 numbers from your studies in grade school: You start from the rightmost column (the 1's digits), add them together, write the result in the rightmost column of the result and possibly carry a 1 into the next column. You proceed column by column, leftward.

We can add binary numbers using a similar approach. The illustration below shows of the process of adding the two four-bit numbers $1110_{(2)} = 14_{(10)}$ and $0111_{(2)} = 7_{(10)}$.



We'll want to build a circuit that follows this process. For the moment, let's worry about just the first step: adding the rightmost bit from each number. What we'll want is a circuit that takes two input bits — we'll call them *a* and *b* — and computes the sum of these two bits. Since the sum could possibly be a two-bit number, this circuit will have two outputs: *sum* will be the sum's 1's-bit, and c_{out} will be the sum's 2's-bit. We'll call this circuit a **half adder**.

We can design our half adder circuit by building a truth table that enumerates each possible combination of the two inputs along with the appropriate value for c_{out} and *sum* in each case.

Table 12: Truth table for half adder

a	b	c_{out}	*sum*
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

To interpret this table, let's look at the last row, where *a* and *b* are both 1. In this case the output is based on the sum of 1 and 1, which is $10_{(2)}$. We'd place the 1's bit of this sum (which is 0) into the bottom of the rightmost column, so this row of the truth table says that *sum* should be 0. And we'd carry the two's bit of the sum (which is 1) into the next column, so the truth table says that c_{out} should be 1.

To build a circuit corresponding to this truth table, we need to determine how each output relates to its inputs. We first notice that the c_{out} output is the AND function on *a* and *b*. The *sum* output, on the other hand, is 1 if *a* is 1 **or** *b* is 1, but not both. This is the **exclusive-or** function on *a* and *b*; it is often abbreviated as the XOR of *a* and *b*. We draw a XOR gate as an OR gate with a shield on its inputs.

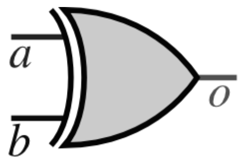
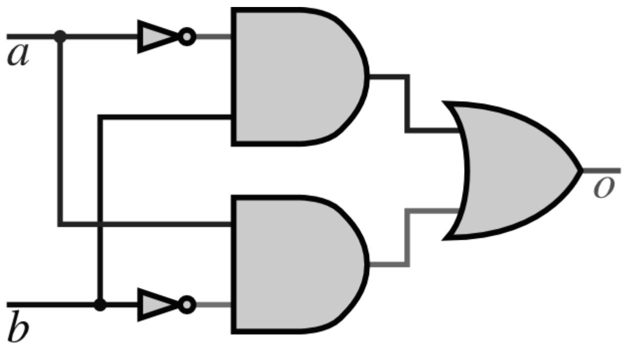
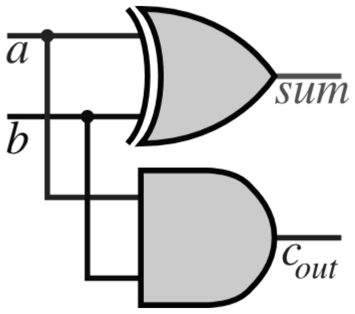


Figure 19: X-OR gate

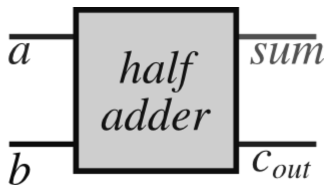
However, if we insist on drawing the circuit with just AND, OR, and NOT gates, we can derive the sum-of-products expression for *sum*: $\overline{a}\, b + a\, \overline{b}$. The resulting circuit is equivalent to the XOR gate.



With a XOR gate built, we can now put together our half adder: *sum* will be the exclusive OR of *a* and *b*, while c_{out} will be the AND of *a* and *b*.



We'll employ this circuit in other circuits that we build. Even though the diagram is rather simple, we'll prefer instead to draw an even simpler box labeled half adder to represent this circuit:



8.4.2 Full adder

Now let's consider the subsequent steps in our addition procedure, where we add later columns. In each of these later columns, there is a possibility that we have a carry coming in from the preceding column. Thus, we'll now want to build a circuit with *three* input bits and produces the sum of these inputs as its output. We'll use *a* and *b* to represent the bit from each of the numbers in the column; but there will be a third input which we'll call c_{in} corresponding to the potential carry from the preceding column. Our circuit will again have two output bits, since the sum of three 0/1 values can be a two-bit output; the bit placed as the result for the column will again be called *sum*, and the number carried into the next column will again be c_{out}.

We've already seen how we to build a circuit for adding the two input bits *a* and *b* together: We can simply use a half adder for that. But we also want to add c_{in} to this sum. To accomplish this, we can use an additional half adder, whose inputs would be the *sum* from the first half adder and c_{in}.

But we'll be left with two values of c_{out} from these two half adders. What should we do to them? We essentially want to add them together, so we could use yet another half adder to accomplish that. But in fact we can guarantee that at least one of these two c_{out} values will be 0: After all, if c_{out} happens to be 1 from the first half adder, then that half adder's *sum* output must necessarily be 0 — and with a 0 as one of the inputs into the second half adder, its c_{out} would be 0. Since at most one of these c_{out} values will be 1, we can add them together using a simple OR gate. That's a better choice than a third full adder because it requires fewer gates.

This leads to the following circuit.

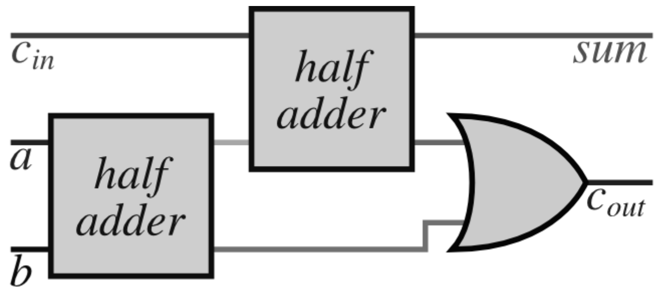


Figure 20: Full adder

The circuit diagram of full adder is shown below:

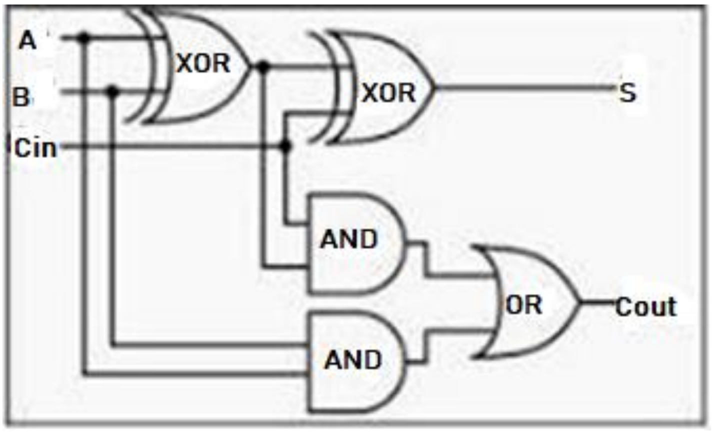


Figure 21: Full adder circuit

As with the half adder, we'll want to reuse this circuit again. We'll feel free to draw a box labeled full adder that represents this circuit.

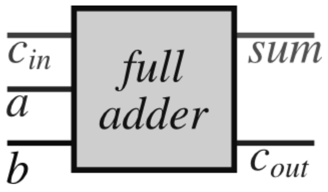


Figure 22: Block diagram of full adder

8.5 Summary

1. A Boolean algebra is an algebraic structure, supplied with two binary operators +, . and a set of elements A , such that, for all elements of the set A, the postulates formulated by E.V. Huntington are satisfied.
2. In case of two valued Boolean algebra, the set of elements contains only two elements 0 and 1.
3. Duality principle states that every algebraic expression deducible from the postulates of Boolean algebra remains valid if the operators and identity elements are interchanged.
4. Theorems of Boolean Algebra are

1.(a) $x + x = x$ (b) $x.x = x$

2.(a) $x + 1 = 1$ (b) $x . 0 = 0$

3. $(x')' = x$

4.(a) $x + (y + z) = (x + y) + z$ (b) $x .(y..z) = (x .y).z$

5.(a) $(x + y)' = x' .y'$ (b) $(x .y)' = x' + y'$

6.(a) $x + x.y = x$ (b) $x.(x+y) = x$

5. A Boolean expression is an algebraic expression formed with binary variables, two binary operators OR(+) and AND (.) , the unary operator NOT(´) and parentheses. A binary variable can take only two values 0 or 1.
6. A Boolean expression can be simplified using the theorems and postulates of boolean algebra and Karnaugh map.
7. A combinational circuit is consists of some interconnected set of logic gates with some input variables and output variables.
8. A multiplexer is a combinational circuit where binary information is selected from one of 2n input lines and transmitted to a single output line. Here n is the number of selection lines.
9. An encoder is a combinational circuit which generates the binary code for the 2n input variables.
10. A decoder is a combinational circuit that receives coded information on n input lines and transmits them to maximum of 2n unique output lines after conversion.
11. A half-adder is a combinational circuit that add two bits.
12. A full-adder is combinational circuits that add three bits.
13. A sequential circuit is the circuit whose the output is depended on the present inputs as well as past output.

Check Your Progress

1. A ___________________ is a collection of some interconnected set of logic gates whose outputs at any time are determined directly from the present combination of inputs without regard to previous inputs.
2. The _____________is a circuit whose output is one of several inputs, depending on the value given by some other selection inputs.
3. The __________________ reverses the process of a multiplexer.

8.6 Model Questions

1. Define a combinational circuit.
2. Explain the functioning of a multiplexer and a demultiplexer.
3. Explain the functioning of an Encoder and a Decoder.
4. Explain the working an a half-adder and a full-adder.
5. Simplify the following Boolean expression using Karnaugh map.
    a. $A.B.C.D + A.B'.C.D + A'.B.C.D + A'.B'.C.D$
    b. $A'.C + A'.B + A.B'.C + B.C$
6. Simplify $x.(x + y) + y.(x' + y)$ using Algebraic Method
7. Prove that
    a. $x.x = x$
    b. $x + xy = x$
8. Explain the function of a multiplexer . Draw the logic diagram of 4 -to-1 multiplexer.
9. Explain full-adder with logic diagram.
10. What do you mean by Sum-of-Product (SOP) of Boolean Expression. Explain with example.

Answers to Check Your Progress

1. combinational circuit
2. multiplexer
3. demultiplexer

BLOCK III

UNIT IX: SEQUENTIAL CIRCUITS

9.0 Learning Objectives

After the completion of this unit, you will be able to:

- Define sequential circuit
- Understand the working of a flip-flop
- Differentiate between various types of filp-flop and understand their working
- Convert D flip-flop to JK flip-flop
- Differentiate between a latch and a flip-flop
- Draw the logic diagram of a clocked RS flip-flop with four NAND gates.
- Define acre around condition
- Know the applications of JK flip-flop

9.1 Sequential Circuit

A circuit is said to be sequential circuit if its output is the function of present input and past output. The basic element used in sequential circuits is a flip-flop, which can be made using logic gates and is available in IC (Integrated Circuit) form. A block diagram of a sequential circuit is in Figure 23. It consists of a combinational circuit to which memory elements are connected to form a feedback path. The memory elements are devices capable of storing binary information within them. These circuits store and remember information. The sequential circuit receives binary information from external inputs. Classification of sequential circuits depends on the timing of their signals. They are classified as follows: (i) synchronous sequential circuits and (ii) asynchronous sequential circuits.

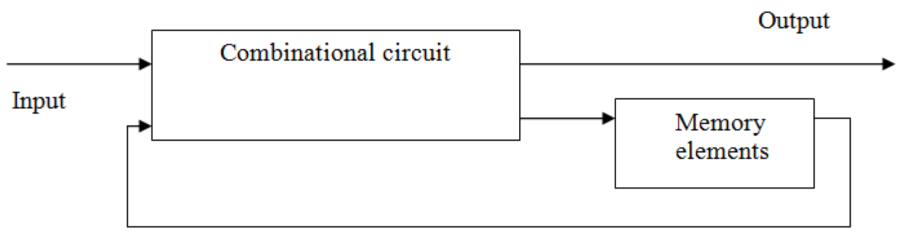


Figure 23: Block diagram of sequential circuit

Synchronous sequential circuits use flip-flops and their status can change only at discrete instants. The synchronization in sequential circuits can be achieved using a clock pulse generator. It synchronizes the effect of input over output. It presents signal of the following form (Figure 24).

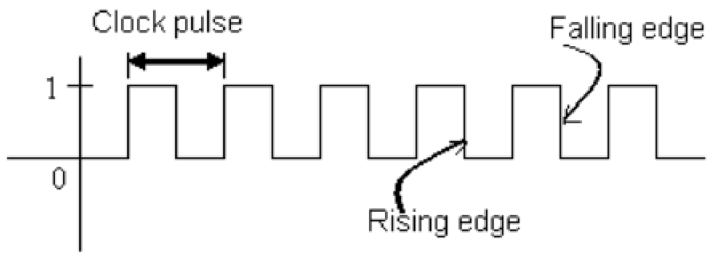


Figure 24: Clock signal of a clock pulse generator

From the diagram we can see that the clock pulse is the time between successive transitions in the same direction, that is, between two rising or two falling edges. State transitions in synchronous sequential circuits are made to take place at times when the clock is making a transition from 0 to 1 (rising edge) or from 1 to 0 (falling edge). Between successive clock pulses there is no change in the information stored in memory.

The behavior of asynchronous sequential circuits depends upon the order in which its input signals change and can be affected at any instant of time. Asynchronous sequential circuits:

1. Do not use clock pulses. The change of internal state occurs when there is a change in the input variable.
2. Their memory elements are either unclocked flip-flops(FF) or time delay-elements.
3. They often resemble combinational circuits with feedback.
4. They are used when speed of operation is important

9.2 Flip Flops

Flip-Flops are the basic building blocks of sequential circuits. A flip-flop is a binary cell which can store a bit of information. A basic function of flip-flop is storage, which means memory. A flip-flop (FF) is capable of storing 1 (one) bit of binary data. It has two stable states either '1' or '0'. A flip-flop maintains any one of the two stable states which can be treated as zero or one depending on presence and absence of output signals.

A flip-flop circuit has two outputs, one for the normal value and other for the complement value of the bit stored in it. A basic flip-flop circuit can be constructed from two NAND gates (Figure 26) or two NOR gates (Figure 25). The flip-flop in the figure have, two inputs R (reset) and S (set) and two outputs Q and Q'.

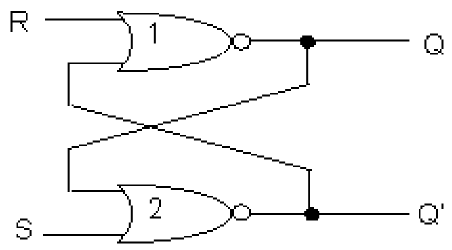


Figure 25: Basic flip-flop circuit with NOR gates

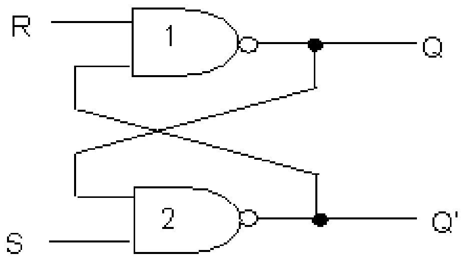


Figure 26: Basic flip-flop circuit using NAND gates

In normal mode of operation both the flip-flop inputs are at 0 i.e., S=0 and R=0. This flip-flop can show two states: either the value of Q is 1 (i.e., set state) or the value of Q is 0. We call it clear state. Let us see how the S and R input can be used to set and clear the state of the flip-flop. Suppose the flip-flop was in set state i.e., Q=1 and Q'=0 and as S=0 and R=0, the output of 'a' NOR gate will be 1 since both its input Q' and R are zero and 'b' NOR gate will show output 0 as one of its input Q is 1. Similarly if flip-flop was in clear state then Q' =1 and R=0, therefore, output of 'a' gate will be '0' and 'b' gate 1. Thus, flip-flop maintains a stable state at S=0 and R=0.

The flip-flop is taken to set if the S input momentarily goes to 1 and then goes back to 0. R remains at zero during this time. Suppose initially, the flip-flop was in state 0, i.e., the value of Q was 0. As soon as S become 1 the output of NOR gate 'b' goes to '0' i.e., Q' becomes 0 and immediately Q becomes 1 as both the input (Q' and R) to NOR gate 'a' become 0. The change in the value of S back to 0 does not change the value of Q again as the input to NOR gate 'b' now are Q=1 and S=0. Thus, the flip-flop stays in set state 1 even after S returns to zero.

If the flip-flop was in state 1 then, when S goes to 1 there is no change in value of Q' as both the input to NOR gate 'b' are 1 at this time. Thus, Q' remains in state 0 or in other words flip-flop stays in set state.

If R input goes to value 1 then flip-flop acquires the clear state. In changing momentarily the value of R to 1 the Q output changes to 0 irrespective of the sate of flip-flop and as Q is 0 and S is 0 the Q' becomes 1. Even after R comes back to value 0, Q remains 0 i.e., flip-flop comes to the clear state.

When both S and R goes to 1 at the same time, well this is the situation which may create a set or clear state depending on which of the S and R stays longer in zero state. But meanwhile both of them are 1 the value of Q and Q' becomes 1 which implies that Q and its complement both are one, an impossible situation. Therefore, the transition of both S and R to 1 simultaneously is an undesirable condition for this basic circuit. There are many types of flip-flops. They are:

1. RS Flip-Flop (Reset-Set Flip-Flop)
2. D type Flip-Flop (Data Latch)

3. JK Flip-Flop
4. T type Flip-Flop (Triggered Flip-Flop)
5. Master-Slave Flip-Flop
6. JK Master Slave Flip-Flop

9.2.1 RS Flip-Flop

The main feature in RS flip-flop is the addition of a clock pulse input. In this flip-flop, change in the value of R or S will change the state of the flip-flop only if the clock pulse at that moment is 1(one). It is denoted as:

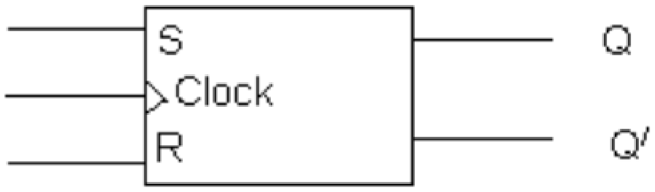


Figure 27: Graphic symbol of RS flip-flop

Here two inputs are Set (S) and Reset (R). The two outputs are Q (Normal) and Q' (Complement of Q). If Q is 1 Q' is 0 and vice versa.

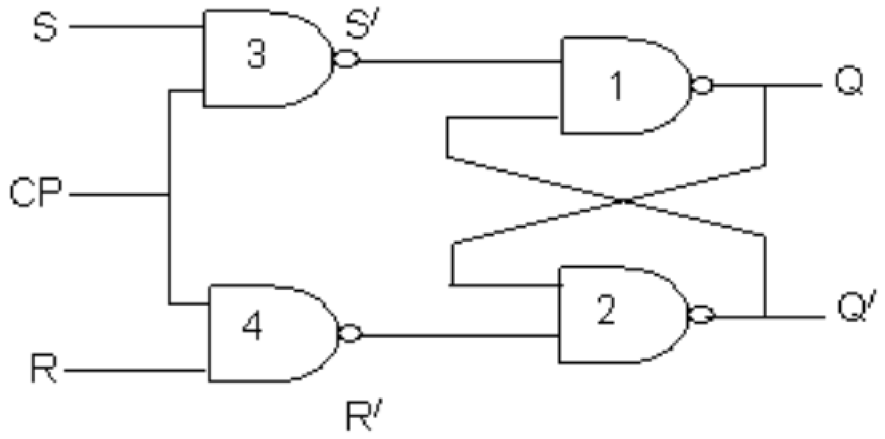


Figure 28: Logic diagram of clocked RS flip-flop using NAND gates

We can construct a flip-flop circuit from two NOR gates(Figure 29) or two NAND gates(Figure 30). The cross-coupled connection from the output of one gate to the input of other gate constitutes a feedback path. Each flip-flop has two outputs, Q and $\overline{Q}$, and two inputs, set and reset. This type of flip-flop is sometimes called a direct-coupled RS flip-flop or SR latch.

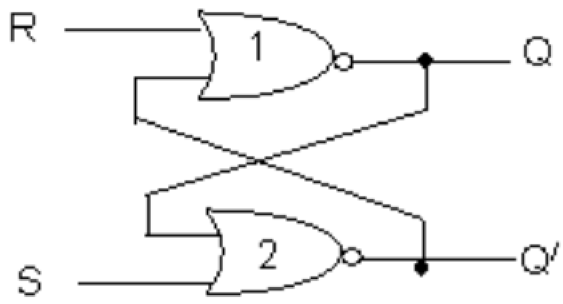


Figure 29: Basic flip-flop circuit with NOR gates

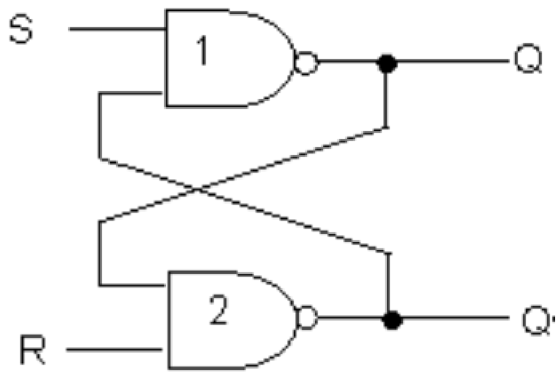


Figure 30: Basic flip-flop circuit with NAND gates

In both the circuits, the output of one gate feeds into the input of the other.

A 0(low) at set S of gate 1 gives an output 1 (high) at Q. This output 1 is fed to gate 2 along with 1 at R. The output of 2nd gate is 0 (low) at Q/. So the output at Q and Q/ are 1 and 0 for inputs of 0 and 1 at R and S respectively. This is the Set condition and remain in that until cleared. To clear the Set Condition, we operate the Reset and the Operation is reverse of Set. R is activated by 0 (low). The output of 2nd gate at Q/ is 1 (high). The 1 (high) output of 2nd gate is fed to 1st gate along with 1 at S. The two 1’s give a 0 output at Q. This state also is a stable one. Thus we see the normal output at Q is either 1 or 0, which is one bit of information in binary, Hence a flip-flop can store one bit of information.

If both R and S are 1 (high) no change can occur in the output from the previous state. This is known as hold state.

If both S and R are 0 (low), both outputs are driven to 1 (high). This is a prohibited state giving rise to racing condition.

The truth table for the RS flip flop is as follows:

Table 13: Truth table of a RS flip-flop

Mode of Operation	**Input**		**Output**	
	R	S	Q	Q’
Hold	1	1	No change	Inactive
Set	1	1	1	0
Reset	1	1	0	1
Race	1	1	1	1

9.2.2 D Flip-Flop

D flip flop is actually a slight modification of the above explained clocked SR flip-flop[3]. From the figure you can see that the D input is connected to the S input and the complement of the D input is connected to the R input. The D input is passed on to the flip flop when the value of CP is ‘1’. When CP is HIGH, the flip flop moves to the SET state. If it is ’0′, the flip flop switches to the CLEAR state. The circuit diagram and truth table is given below.

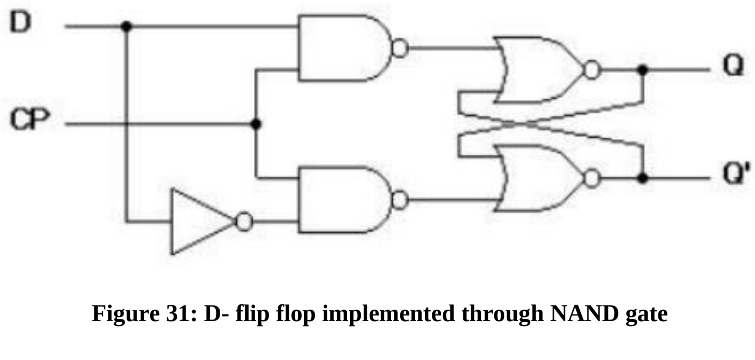


Figure 31: D- flip flop implemented through NAND gate

[3] Adapted from http://todayscircuits.blogspot.com/2011/06/flip-flops.html#.WCGARclgSpp

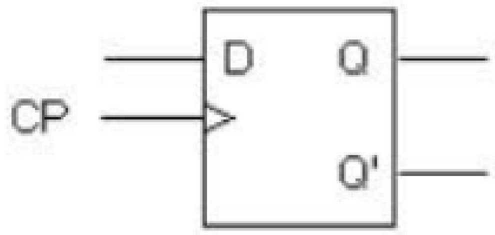


Figure 32: Graphical symbol

Table 14: Transition table

Q	D	Q(t+1)
0	0	0
0	1	1
1	0	0
1	1	1

9.2.3 JK Flip-Flop

A J-K flip flop can also be defined as a modification of the S-R flip flop. The only difference is that the intermediate state is more refined and precise than that of a S-R flip flop.

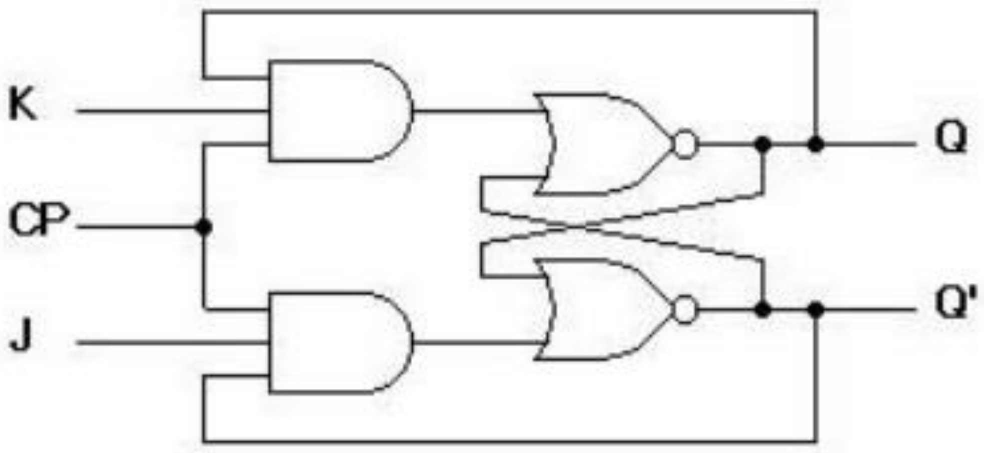


Figure 33: Logic diagram of JK flip-flop

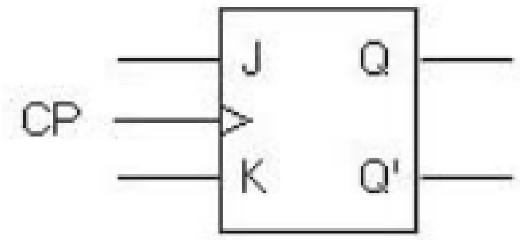


Figure 34: Graphical symbol of JK flip-flop

Table 15: Transition table of JK flip-flop

Q	J	K	Q(t+1)
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

The behavior of inputs J and K is same as the S and R inputs of the S-R flip flop. The letter J stands for SET and the letter K stands for CLEAR.

When both the inputs J and K have a HIGH state, the flip-flop switch to the complement state. So, for a value of Q = 1, it switches to Q=0 and for a value of Q = 0, it switches to Q=1.

The circuit includes two 3-input AND gates. The output Q of the flip flop is returned back as a feedback to the input of the AND along with other inputs like K and clock pulse [CP]. So, if the value of CP is ’1′, the flip flop gets a CLEAR signal and with the condition that the value of Q was earlier 1. Similarly output Q’ of the flip flop is given as a feedback to the input of the AND along with other inputs like J and clock pulse [CP]. So the output becomes SET when the value of CP is 1 only if the value of Q’ was earlier 1.

The output may be repeated in transitions once they have been complimented for J=K=1 because of the feedback connection in the JK flip-flop. This can be avoided by setting a time duration lesser than the propagation delay through the flip-flop. The restriction on the pulse width can be eliminated with a master-slave or edge-triggered construction.

9.2.4 T Flip-Flop

This is a much simpler version of the J-K flip flop. Both the J and K inputs are connected together and thus are also called a single input J-K flip flop. When clock pulse is given to the flip flop, the output begins to toggle. Here also the restriction on the pulse width can be eliminated with a master-slave or edge-triggered construction. Take a look at the circuit and truth table below.

Figure 35: Logic diagram of T flip-flop

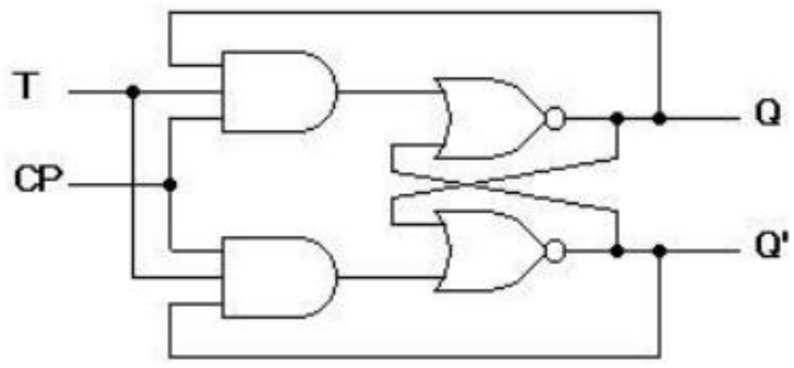


Figure 36: Graphical symbol of T flip-flop

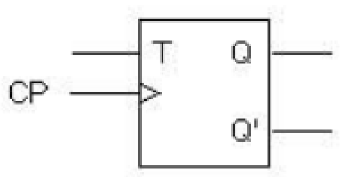


Table 16: Transition table of T flip-flop

Q	T	Q(T+1)
0	0	0
0	1	1
1	0	1
1	1	0

9.2.5 Master-Slave Flip-Flop

Consider the circuit given in Figure 37. It gives the logic diagram for master-slave *D* flip-flop[4]. It has two flip-flops (called master and slave respectively) and an inverter. The slave section is basically the same as the master section except that it is clocked on the inverted clock pulse and is controlled by the outputs of the master section rather than by the external inputs. The D input is sampled to change the output Q only at the negative edge of the clock pulse CP. The output of the NOT gate is 1 when the CP is 0. This enables slave flip-flop and its output Q is equal to the output Y of master flip-flop. The master flip-flop is disabled when CP is at logic 0 level. When the CP goes 1, the data from the external D input is given to the master. As long as the CP stays at the 1 level, the slave is disabled because the output of the inverter is equal to 0. Any input change affects the master output Y. However, it cannot affect the slave output.

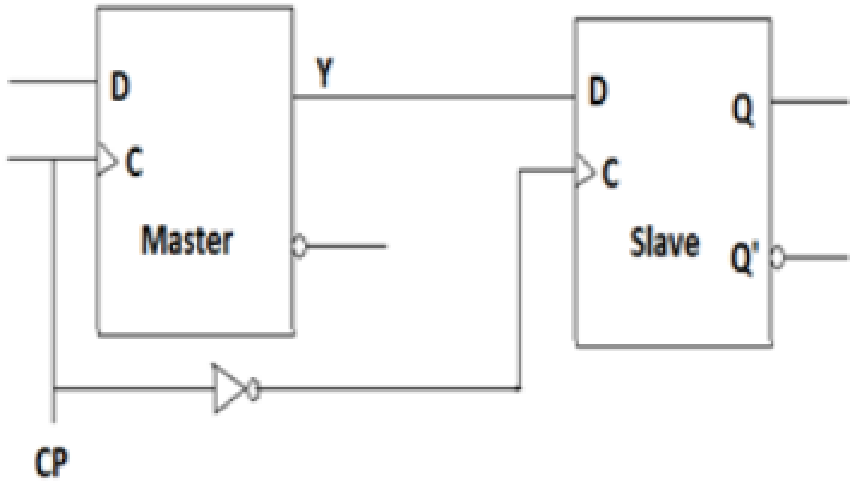


Figure 37: Master slave flip-flop

[4] Adapted from http://vle.du.ac.in/mod/book/view.php?id=11685&chapterid=22699

9.2.6 JK Master-Slave Flip-Flop

Master-slave flip flop is designed using two separate flip flops[5]. Out of these, one acts as the master and the other as a slave. The figure of a master-slave J-K flip flop is shown below.

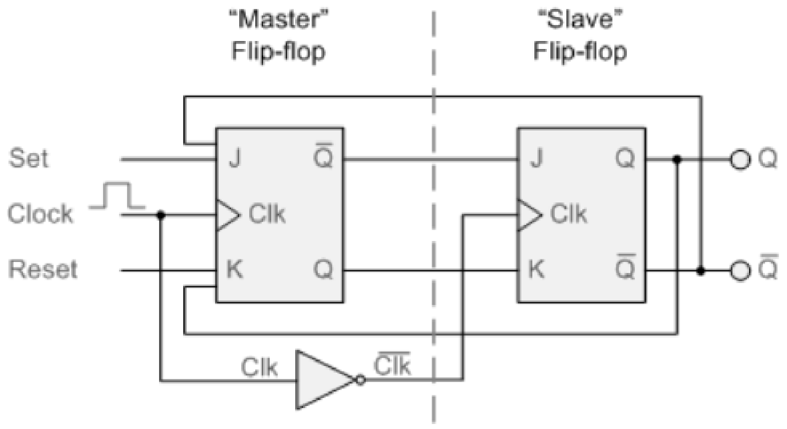


Figure 38: JK Master-Slave flip flop

From the above figure you can see that both the J-K flip flops are presented in a series connection. The output of the master J-K flip flop is fed to the input of the slave J-K flip flop. The output of the slave J-K flip flop is given as a feedback to the input of the master J-K flip flop. The clock pulse [Clk] is given to the master J-K flip flop and it is sent through a NOT Gate and thus inverted before passing it to the slave J-K flip flop.

When Clk=1, the master J-K flip flop gets disabled. The Clk input of the master input will be the opposite of the slave input. So the master flip flop output will be recognized by the slave flip flop only when the Clk value becomes 0. Thus, when the clock pulse males a transition from 1 to 0, the locked outputs of the master flip flop are fed through to the inputs of the slave flip-flop making this flip flop edge or pulse-triggered. To understand better take a look at the timing diagram illustrated below.

[5]Adapted from http://eduladder.com/viewquestions.php?questionid=766&title=working%20of%20master-slave%20JK%20flip%20flop

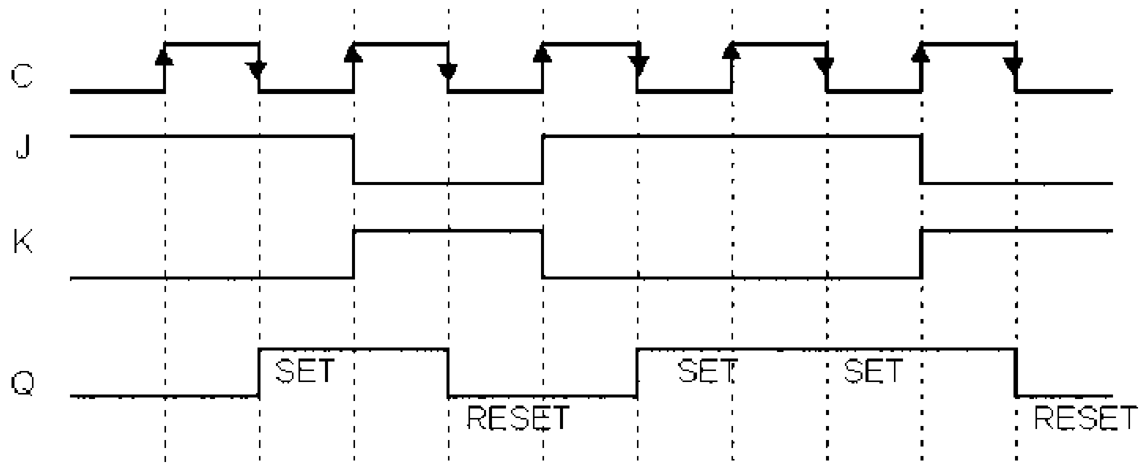


Figure 39: Timing diagram of JK Master-Slave flip flop

Thus, the circuit accepts the value in the input when the clock is HIGH, and passes the data to the output on the falling-edge of the clock signal. This makes the Master-Slave J-K flip flop a Synchronous device as it only passes data with the timing of the clock signal.

9.3 Summary

1. The output state of a "sequential logic circuit" is a function of the following three states, the "present input", the "past input" and/or the "past output".
2. Sequential logic circuits are generally termed as two state or Bistable devices which can have their output or outputs set in one of two basic states, a logic level "1" or a logic level "0" and will remain "latched" (hence the name latch) indefinitely in this current state or condition until some other input trigger pulse or signal is applied which will cause the bistable to change its state once again.
3. Sequential logic circuits can be divided into the following three main categories:
    a. Event Driven – asynchronous circuits that change state immediately when enabled.
    b. Clock Driven – synchronous circuits that are synchronised to a specific clock signal.
    c. Pulse Driven – which is a combination of the two that responds to triggering pulses.
4. The **SR flip-flop**, also known as a *SR Latch*, can be considered as one of the most basic sequential logic circuit possible.

5. It is basically S-R latch using NAND gates with an additional **enable** input. It is also called as level triggered SR-FF.
6. Master slave JK FF is a cascade of two S-R FF with feedback from the output of second to input of first. Master is a positive level triggered. But due to the presence of the inverter in the clock line, the slave will respond to the negative level.
7. Delay Flip Flop or D Flip Flop is the simple gated S-R latch with a NAND inverter connected between S and R inputs. It has only one input. The input data is appearing at the output after some time. Due to this data delay between i/p and o/p, it is called delay flip flop.
8. Toggle flip flop is basically a JK flip flop with J and K terminals permanently connected together.

Check Your Progress

A. Multiple choice questions

1. How is a J-K flip-flop made to toggle?
   a. J = 0, K = 0
   b. J = 1, K = 0
   c. J = 0, K = 1
   d. J = 1, K = 1
2. With regard to a D latch, ________.
   a. the Q output follows the D input when EN is LOW
   b. the Q output is opposite the D input when EN is LOW
   c. the Q output follows the D input when EN is HIGH
   d. the Q output is HIGH regardless of EN's input state
3. Two cross coupled NAND gates make
   a. SR Latch
   b. RS flipflop
   c. D flipflop
   d. master slave flipflop
4. D flipflop is constructed with
   a. AND gates
   b. OR gates
   c. NAND gates
   d. NOR gates
5. Mater slave flipflop can be constructed with
   a. SR Latch
   b. Adder
   c. JK flipflop
   d. multiplier

9.4 Model Questions

1. Define sequential circuit.
2. Define Flip Flop. Give the difference between Latch and Flip Flop.
3. What is sequential circuit ? How it differ from combinational circuit?
4. What is flip-flop? Explain RS flip-flop.
5. Draw the logic diagram of a clocked RS flip-flop with four NAND gates.
6. With the help of logic diagram and truth table explain the working of clocked RS Flip Flop.
7. What is “race around condition” in JK Flip flop?
8. What are the applications of JK F/F?
9. Show how a D Flip-Flop can be converted into JK-Flip Flop.

Answers to Check Your Progress

A. Multiple choice questions

1. A

2. C

3. A

4. C

5.C

CHAPTER 10: REGISTERS

10.0 Learning Outcomes

After the completion of this unit, you will be able to:

- Define and understand the functions of a register
- Demonstrate the working of a sift register
- Explain different types of shift registers
- Know the applications of shift registers
- Design 3-bit Parallel Input, Serial Output shift register

10.1 Introduction

With flip-flop we can store data bitwise but usually data does not appear as single bit. Instead it is common to store data words of n bit with typical word lengths of 4, 8, 16, 32 or 64 bits. Thus, several flip-flops are combined to form a register to store whole data words. Registers are synchronous circuits thus all flip-flops are controlled by a common clock line. A group of flip-flops constitutes a register as each flip-flop is a binary cell capable of storing one bit of information. An n - bit register has a group of n flip-flops and is capable of storing any binary information containing n bits.

Figure 40 shows a register constructed with four D flip-flops and a common clock pulse(CP) input. The CP enables all flip-flops so that information presently available at the four inputs can be transferred into the 4-bit register. The four outputs can be sampled to obtain the information presently stored in the register.

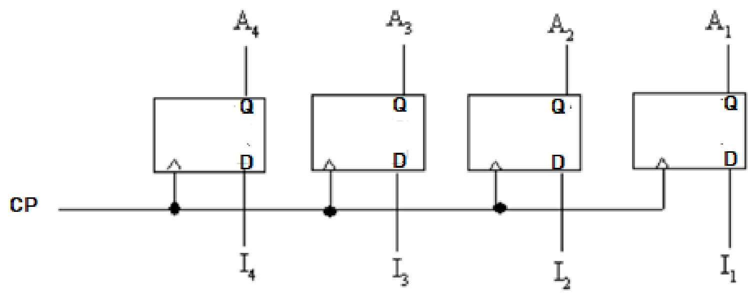


Figure 40: 4-bit register

10.2 Shift registers

Shift registers are a type of sequential logic circuit, mainly for storage of digital data. A register capable of shifting its binary information either to the right or to the left at a time, when one clock pulse is applied is called a shift register. The logical configuration of a shift register consist of a chain of flip-flops connected in cascade, with the output of one flip-flop connected to the input of the next flip-flop. All flip-flops receive a common clock pulse which causes the shift from one stage to the next.

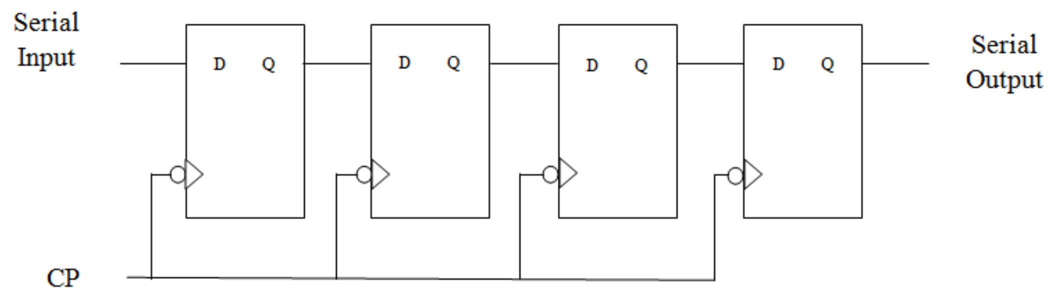


Figure 41: Shift register

The Q output of a given flip-flop is connected to the D input of flip-flop at its right. Each clock pulse shifts the contents of the register one-bit position to the right. The serial input determines what goes into the leftmost flip-flop during the shift. The serial output is taken from the output of the rightmost flip-flop prior to the application of a pulse. Although this register shifts the contents to the right, if we turn the page upside down, we find that the register shifts its content to the left. Thus, a unidirectional shift register can function either as shift-right or as a shift-left register.

10.3 Types of shift registers

A Shift Register can be used in four different configurations depending upon the way in which the data is entered into and take out of it. These four configurations are:

i. Serial input, serial output (SISO): In this type of Shift Register data can be moved in and out of the register, one bit at a time.

ii. Parallel input, serial output (PISO): Here data can be loaded simultaneously. But the data can be removed from the register one bit at a time by the clock pulse.

iii. Serial input, parallel output (SIPO): In this case data is loaded serially one bit at a time but the data stored can be read simultaneously.

iv. Parallel input, parallel output (PIPO): In this type of shift register the data can be loaded into the storage simultaneously and can also be taken out or read simultaneously.

10.3.1 Serial Input, Serial Output (SISO) Shift Register

A serial-in serial-out(SISO) shift register[6] enters and removes binary data in serial form. The diagram below shows an SISO shift register with four J-K flip-flops. The diagram below shows an SISO shift register with four J-K flip-flops. Four clock pulses are required to serially shift a 4-bit binary number. The number is transferred into the first flip-flop(FFA) of the register one bit at a time. As each new bit is entered, all of the bits that are in the register are simultaneously shifted one position to the right.

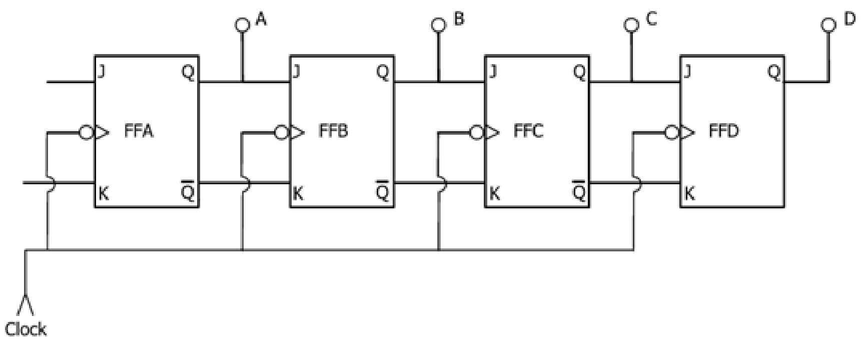


Figure 42: Serial-in serial-out(SISO) shift register

One shift occurs as the clock pulse is applied simultaneously to each flip-flop. The binary bit that is shifted into each flip-flop (a 0 or a 1) is determined by the mode that it is in before the net clock pulse arrives. The mode is determined by whether a 0 or a 1 is applied to the J and K inputs.

[6] Adopted from https://www.wisc-online.com/learn/career-clusters/manufacturing/dig3903/serial-in-serial-out-shift-register

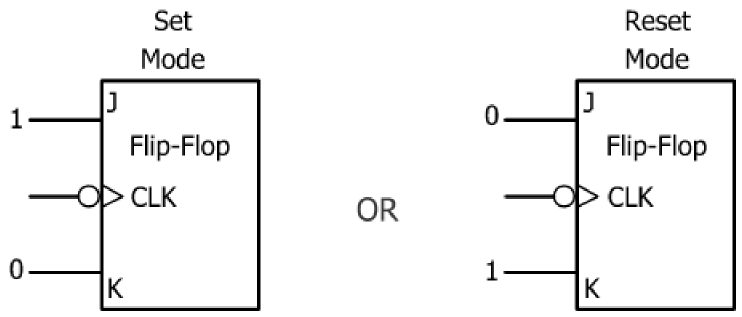
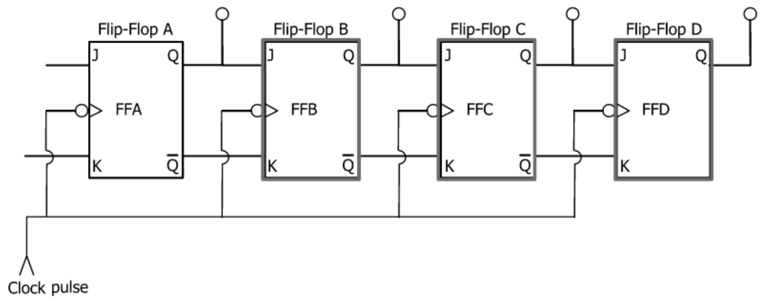
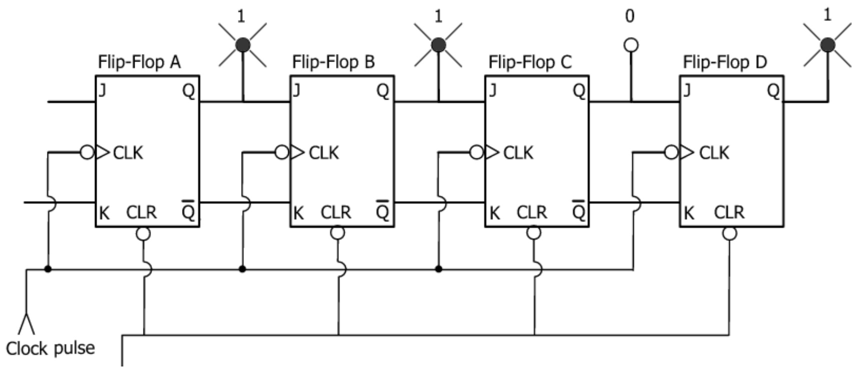


Figure 43: SET and RESET mode in JK flip-flop

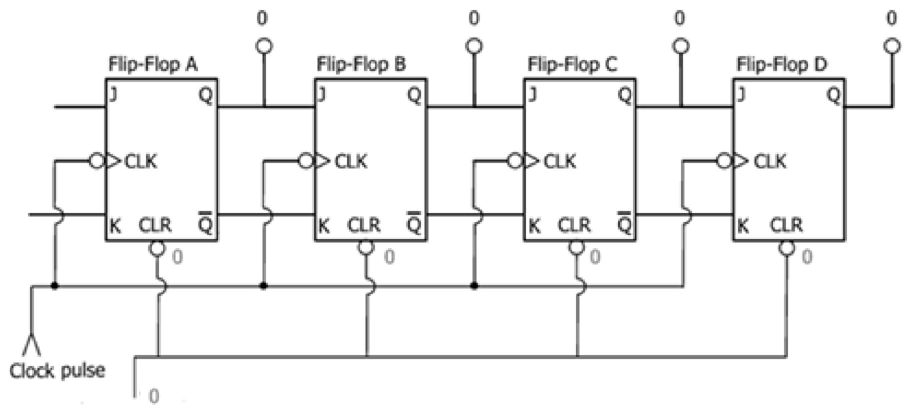
For flip-flops B,C and D, the logic state applied to the inputs is determined by the Q and $\overline{Q}$ outputs of the previous flip-flop. For flip-flop A, the mode is determined by the logic state that is applied to its J input and the opposite inverted logic state at the K input.



The diagram below shows the flip-flops with contents stored in them.



Sometimes it is desirable to clear the register. This function is achieved by applying a momentary logic 0 to all of the flip-flops simultaneously.



10.3.2 Serial Input, Parallel Output (SIPO) Shift Register

A Serial-In Parallel-Out shift register is a shift register that converts serial data to parallel data and all the internal stages of the SIPO are available as outputs. If four data bits are shifted in by four clock pulses via a single wire at data-in, below, the data becomes available simultaneously on the four Outputs out[0] to out[3] after the fourth clock pulse.

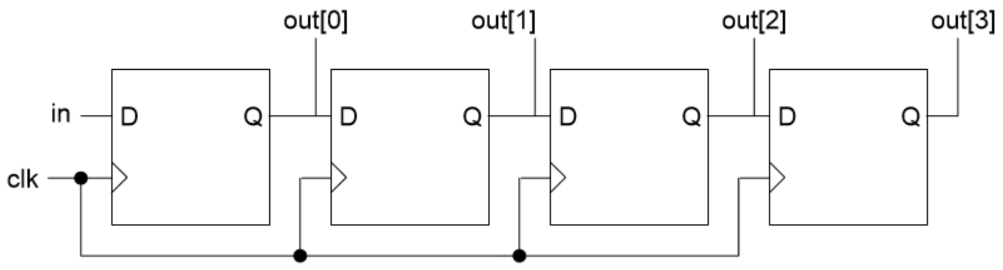


Figure 44: SIPO shift register

10.3.3 Parallel Input, Serial Output (PISO) Shift Register

Parallel-Input Serial-Output shift registers provides input data to all stages simultaneously. It stores data, shifts it on a clock by clock basis, and delays it by the number of stages times the clock period. In addition, parallel-in/ serial-out really means that we can load data in parallel into all stages before any shifting ever begins. This is a way to convert data from a *parallel* format to a *serial* format. By parallel format we mean that the data bits are present simultaneously on individual wires, one for each data bit as shown below. By serial format we mean that the data bits are presented sequentially in time on a single wire or circuit as in the case of the "data out" on the block diagram below.

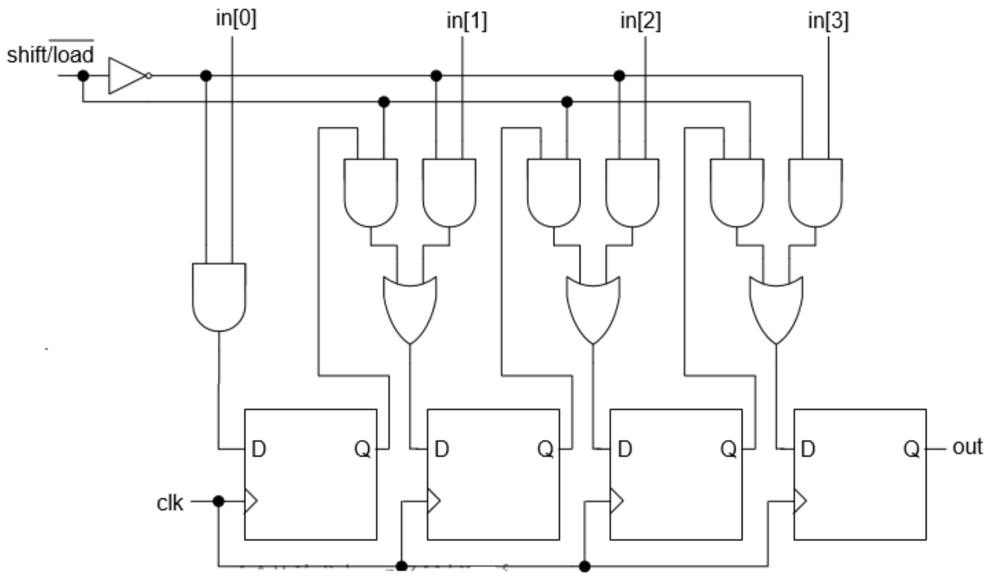


Figure 45: PISO shift register

10.3.4 Parallel Input, Parallel Output (PIPO) Shift Register

The purpose of the parallel-in/ parallel-out shift register is to take in parallel data, shift it, then output it as shown below. A universal shift register is a do-everything device in addition to the parallel-in/ parallel-out function.

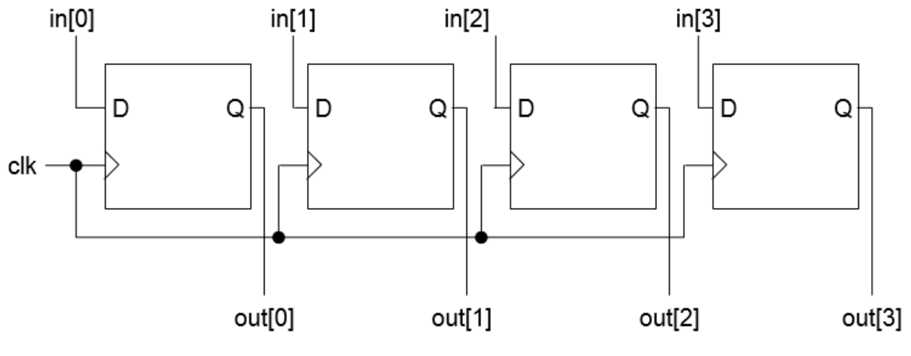
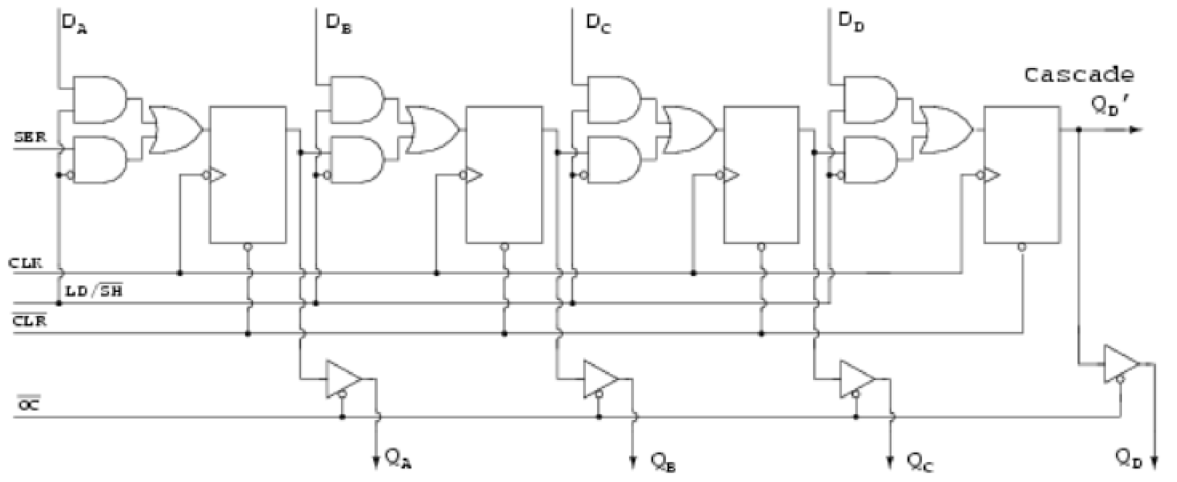


Figure 46: PIPO shift register

Above we apply four bit of data to a parallel-in/ parallel-out shift register at in[0] in[1] in[2] in[3]. The mode control, which may be multiple inputs, controls parallel loading vs shifting. The mode control may also control the direction of shifting in some real devices. The data will be shifted one bit position for each clock pulse. The shifted data is available at the outputs out[0] out[1] out[2] out[3]. The “data in” and “data out” are provided for cascading of multiple stages. Though, above, we can only cascade data for right shifting. We could accommodate cascading of left-shift data by adding a pair of left pointing signals, “data in” and “data out”, above.

The internal details of a right shifting parallel-in/ parallel-out shift register are shown below. The tri-state buffers are not strictly necessary to the parallel-in/ parallel-out shift register, but are part of the real-world device shown below.



10.4 Application of Shift register

Shift registers can be found in many applications. Here is a list of a few.

i. To produce time delay: The serial in -serial out shift register can be used as a time delay device. The amount of delay can be controlled by the number of stages in the register or by the clock frequency.

ii. To simplify combinational logic: A major problem in the realization of sequential circuits is the assignment of binary codes to the internal states of the circuit in order to reduce the complexity of circuits required. By assigning one flip-flop to one internal state, it is possible to simplify the combinational logic required to realize the complete sequential circuit. When the circuit is in a particular state, the flip-flop corresponding to that state is set to HIGH and all other flip-flops remain LOW.

iii. To convert serial data to parallel data: A computer or microprocessor-based system commonly requires incoming data to be in parallel format. But frequently, these systems must communicate with external devices that send or receive serial data. So, serial-to-parallel conversion is required.

Check Your Progress

1. _______________ are a type of sequential logic circuit, mainly for storage of digital data.
2. A ____________ shift register enters and removes binary data in serial form.

10.5 Summary

1. A register is made out of multiple flip-flops connected to each other used to store multiple bits of data.
2. A shift register is able to shift the data it contains by one bit either to the left or to the right. They are used to transform serial data to parallel, and parallel to serial data.
3. The **Shift Register** is another type of sequential logic circuit that can be used for the storage or the transfer of data in the form of binary numbers.
4. Serial-in/Serial-out Shift Register delays the input data by one clock cycle for each stage.
5. Serial-in/Parallel-out Shift Register is similar to the Serial-in/Serial-out shift registers in that it also shifts data with each clock cycle. However, the difference is that the internal stages are all made available at the output, thus converting the serial input data into parallel output data format.

10.6 Model Questions

1. What is register?
2. Define shift register?
3. Explain the four configurations of shift registers.
4. Explain the working of serial-in serial-out(SISO) shift register in details.

5. Write down the different types of shift registers and its functions.
6. What are the different applications of a shift register.
7. Design 3-bit PISO (Use D –Flip Flop).

Answers to Check Your Progress

1. Shift registers
2. serial-in serial-out(SISO)

CHAPTER 11: COUNTER

11.0 Learning Objectives

After the completion of this unit, you will be able to:

- Differentiate between synchronous and asynchronous counter
- Classify various types of counters
- Explain the working of Asynchronous (ripple) counter
- Define the working of Synchronous counter
- Define the working of an asynchronous counter
- Explain the working of Decade counter (MOD 10 Counter)
- Explain the working of Ring Counter
- Explain the working of Johnson counter

11.1 Introduction

In digital logic and computing, a counter is a device which stores (and sometimes displays) the number of times a particular event or process has occurred, often in relationship to a clock signal[7]. The most common type is a sequential digital logic circuit with an input line called the "clock" and multiple output lines. The values on the output lines represent a number in the binary or BCD number system. Each pulse applied to the clock input increments or decrements the number in the counter.

A counter circuit is usually constructed of a number of flip-flops connected in cascade. Counters are a very widely used component in digital circuits, and are manufactured as separate integrated circuits and also incorporated as parts of larger integrated circuits.

In electronics, counters can be implemented quite easily using register-type circuits such as the flip-flop, and a wide variety of classifications exist:

1. Asynchronous (ripple) counter – changing state bits are used as clocks to subsequent state flip-flops
2. Synchronous counter – all state bits change under control of a single clock

[7] Adopted from http://wikivisually.com/wiki/Counter_(digital)

3. Decade counter – counts through ten states per stage
4. Ring counter – formed by a shift register with feedback connection in a ring
5. Johnson counter – a twisted ring counter

Each is useful for different applications. Usually, counter circuits are digital in nature, and count in natural binary. Many types of counter circuits are available as digital building blocks, for example a number of chips in the 4000 series implement different counters.

Occasionally there are advantages to using a counting sequence other than the natural binary sequence—such as the binary coded decimal counter, a linear feedback shift register counter, or a Gray-code counter. Counters are useful for digital clocks and timers, and in oven timers, VCR clocks, etc.

11.2 Asynchronous (ripple) counter

An asynchronous (ripple) counter is a single d-type flip-flop, with its J (data) input fed from its own inverted output. This circuit can store one bit, and hence can count from zero to one before it overflows (starts over from 0). This counter will increment once for every clock cycle and takes two clock cycles to overflow, so every cycle it will alternate between a transition from 0 to 1 and a transition from 1 to 0.

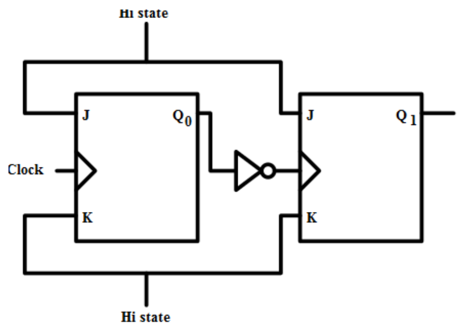


Figure 47: Asynchronous counter created from two JK flip flop

Notice that this creates a new clock with a 50% duty cycle at exactly half the frequency of the input clock. If this output is then used as the clock signal for a similarly arranged D flip-flop (remembering to invert the output to the input), one will get another 1 bit counter that counts half as fast. Putting them together yields a two-bit counter:

Cycle	Q1	Q0	(Q1:Q0)dec
0	0	0	0
1	0	1	1
2	1	0	2
3	1	1	3
4	0	0	0

You can continue to add additional flip-flops, always inverting the output to its own input, and using the output from the previous flip-flop as the clock signal. The result is called a ripple counter, which can count to 2n - 1 where n is the number of bits (flip-flop stages) in the counter. Ripple counters suffer from unstable outputs as the overflows "ripple" from stage to stage, but they do find frequent application as dividers for clock signals, where the instantaneous count is unimportant, but the division ratio overall is (to clarify this, a 1-bit counter is exactly equivalent to a divide by two circuit; the output frequency is exactly half that of the input when fed with a regular train of clock pulses). The use of flip-flop outputs as clocks leads to timing skew between the count data bits, making this ripple technique incompatible with normal synchronous circuit design styles.

11.3 Synchronous counter

The term synchronous refers to the events that have a fixed time relationship with each other. With respect to counter operation, synchronous means that all the flip-flops in the counter are clocked at the same time by a common clock pulse. In this type of counter the clock pulse are applied to the CP inputs of all flip-flops. The common pulse trigger all the flip-flops simultaneously, rather than one at a time in succession as in the ripple counter. This increases the speed of operation of the counters. The pulse to be counted are applied at the clock input terminal.

High-frequency operations require that all the flip-flops of a counter be triggered at the same time to prevent errors. We use a Synchronous counter for this type of operation. The synchronous counter is similar to a ripple counter with two exceptions: The clock pulses are applied to each flip-flop, and additional gates are added to ensure that the flip-flops toggle in the proper sequence.

11.3.1 Working of a three-stage synchronous counter

A logic diagram of a three-stage synchronous counter is shown in Figure 48. Pulse sequence is shown in **Error! Reference source not found.**. The clock input is wired to each of the flip-flops(FF) to prevent possible errors in the count. A HIGH is wired to the J and K inputs of FF1 to make the FF toggle. The output of FF1 is wired to the J and K inputs of FF2, one input of the AND gate, and indicator A. The output of FF2 is wired to the other input of the AND gate and indicator B. The AND output is connected to the J and K inputs of FF3. The C indicator is the only output of FF3.

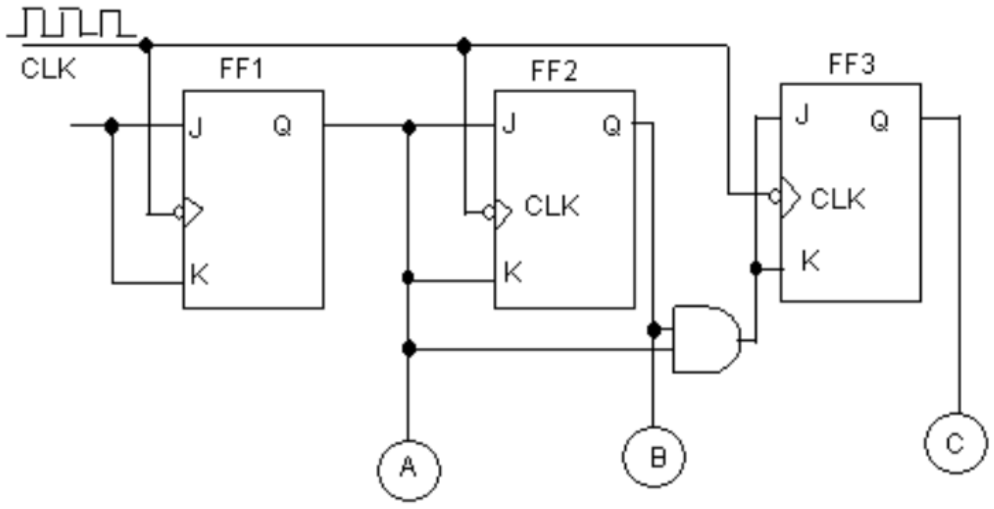


Figure 48: Logic diagram of three stage synchronous counter

Let us assume the following initial conditions: The outputs of all FFs, the clock, and the AND gate are 0; the J and K inputs to FF1 are HIGH.

Clock pulse 1 causes FF1 to set. This HIGH lights lamp A, indicating a binary count of 001. The HIGH is also applied to the J and K inputs of FF2 and one input of the AND gate. FF2 and FF3 are unaffected by the first clock pulse because the J and K inputs were LOW when the clock pulse was applied.

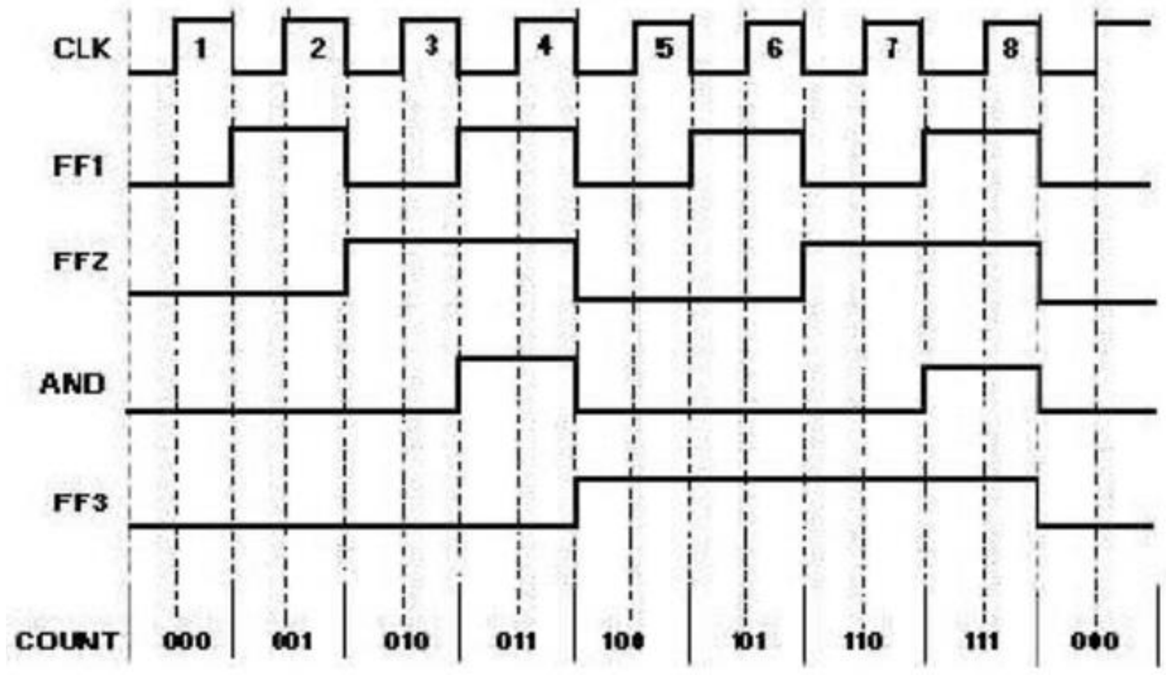


Figure 49: Timing diagram with pulse sequence

As clock pulse 2 goes LOW, FF1 resets, turning off lamp A. In turn, FF2 will set, lighting lamp B and showing a count of 010. The HIGH from FF2 is also felt by the AND gate. The AND gate is not activated at this time because the signal from FF1 is now a LOW. A LOW is present on the J and K inputs of FF3, so it is not toggled by the clock. Clock pulse 3 toggles FF1 again and lights lamp A. Since the J and K inputs to FF2 were LOW when pulse 3 occurred, FF2 does not toggle but remains set. Lamps A and B are lit, indicating a count of 011. With both FF1 and FF2 set, HIGHs are input to both inputs of the AND gate, resulting in HIGHs to J and K of FF3. No change occurred in the output of FF3 on clock pulse 3 because the J and K inputs were LOW at the time.

Just before clock pulse 4 occurs, we have the following conditions: FF1 and FF2 are set, and the AND gate is outputting a HIGH to the J and K inputs of FF3. With these conditions all of the FFs will toggle with the next clock pulse. At clock pulse 4, FF1 and FF2 are reset, and FF3 sets. The output of the AND gate goes to 0, and we have a count of 100.

It appears that the clock pulse and the AND output both go to 0 at the same time, but the clock pulse arrives at FF3 before the AND gate goes LOW because of the transit time of the signal through FF1, FF2, and the AND gate.

Between pulses 4 and 8, FF3 remains set because the J and K inputs are LOW. FF1 and FF2 toggle in the same sequence as they did on clock pulses 1, 2, and 3.

Clock pulse 7 results in all of the FFs being set and the AND gate output being HIGH.

Clock pulse 8 causes all the FFs to reset and all the lamps to turn off, indicating a count of 000 . The next clock pulse (9) will restart the count sequence.

11.4 Decade counter (MOD 10 Counter)

A Mod 10 counter has 10 possible states i.e. it counts from 0 to 9 and roll over[8]. Looking at the truth table, the counter should run from 0000_2 to 1001_2.

```
0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
```

Since the counter has to display 1001_2, the next number (1010_2) will be used to reset the counter to zero. Since the asynchronous inputs are active high, an AND gate will be used. The two flip-flops where a “1” occurs will be tied to an AND gate, and the output will be tied to the ‘clear’ input.

When the counter goes to 1010_2, the AND gate will have a 1 on its output and will activate the “clear” inputs. This will reset the counter to 0000_2.The 1010_2 will never be displayed. In essence, the counter counted/displayed from 0000_2 to 1001_2.

[8] https://www.wisc-online.com/learn/career-clusters/manufacturing/dig4603/the-mod-10-counter

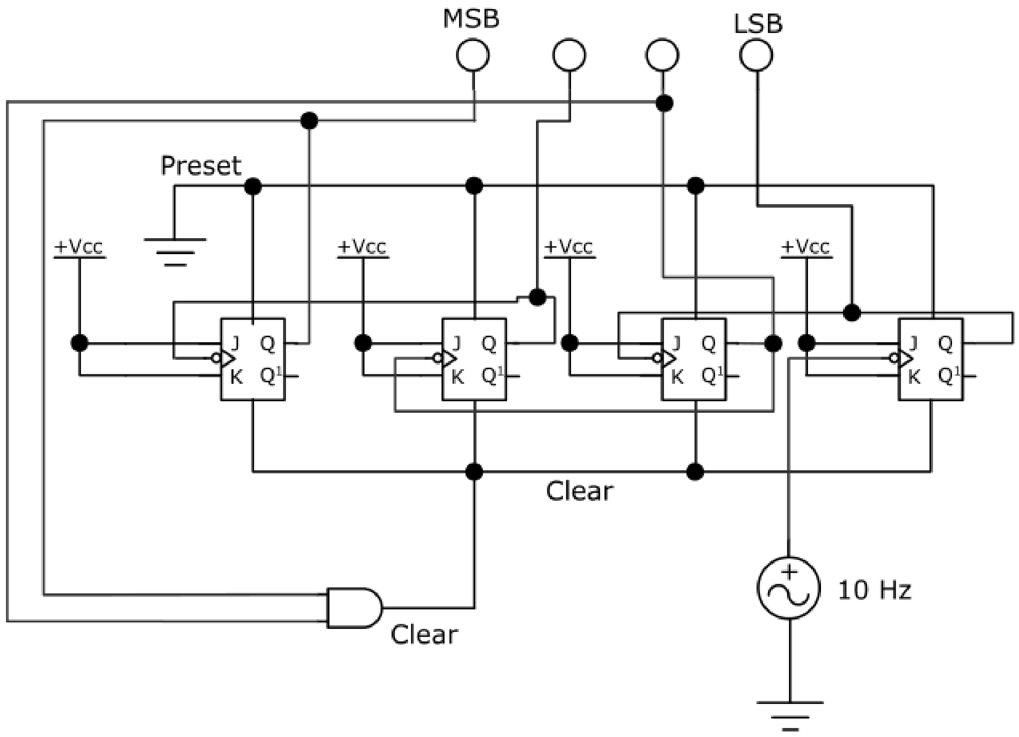


Figure 50: Mod 10 counter

11.5 Ring Counter

A ring counter is a circular shift register which is initiated such that only one of its flip-flops is the state one while others are in their zero states. A ring counter is a shift register (a cascade connection of flip-flops) with the output of the last one connected to the input of the first, that is, in a ring. Typically, a pattern consisting of a single bit is circulated so the state repeats every n clock cycles if **n** flip-flops are used.

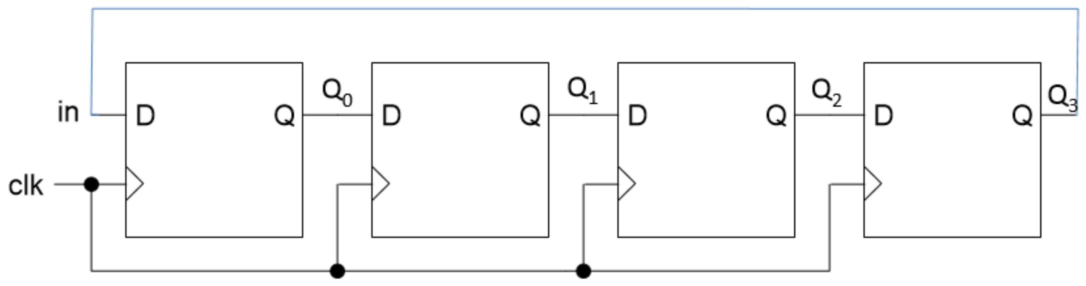


Figure 51: Ring counter

11.6 Johnson counter

A Johnson counter (or switch-tail ring counter, twisted ring counter, walking ring counter, or Möbius counter) is a modified ring counter, where the output from the last stage is inverted and fed back as input to the first stage. The register cycles through a sequence of bit-patterns, whose length is equal to twice the length of the shift register, continuing indefinitely. These counters find specialist applications, including those similar to the decade counter, digital-to-analog conversion, etc. They can be implemented easily using D- or JK-type flip-flops. It is also known as twisted ring counter.

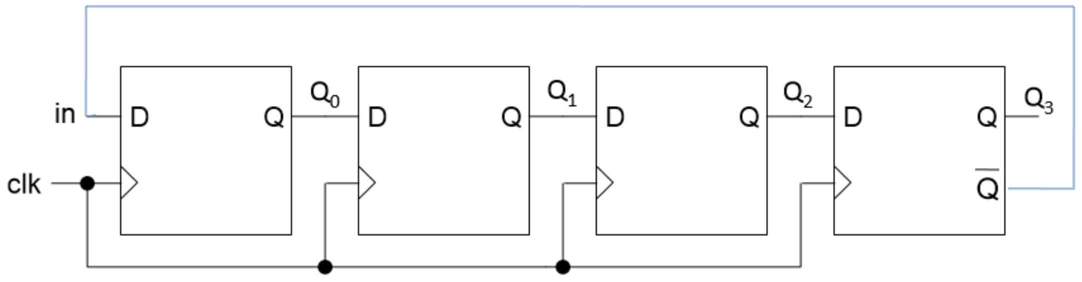


Figure 52: Johnson counter

Check Your Progress

1. A digital circuit which is used for a counting pulses is known as ____________.
2. High-frequency operations require that all the flip-flops of a counter be triggered at the ________ time to prevent errors.
3. A counter is a circular shift register which is initiated such that only one of its flip-flops is the state one while others are in their zero states.
4. A ____________ counter is a modified ring counter, where the output from the last stage is inverted and fed back as input to the first stage.

11.7 Summary

1. Counter is a sequential circuit. A digital circuit which is used for a counting pulses is known counter.
2. Counters are of two types: Asynchronous or ripple counters and Synchronous counters.

3. If the "clock" pulses are applied to all the flip-flops in a counter simultaneously, then such a counter is called as synchronous counter.
4. An asynchronous (ripple) counter is a single d-type flip-flop, with its J (data) input fed from its own inverted output.
5. A Mod 10 counter has 10 possible states i.e. it counts from 0 to 9 and roll over.
6. A ring counter is a circular shift register which is initiated such that only one of its flip-flops is the state one while others are in their zero states.
7. A Johnson counter (or switch-tail ring counter, twisted ring counter, walking ring counter, or Möbius counter) is a modified ring counter, where the output from the last stage is inverted and fed back as input to the first stage.

11.8 Model Questions

1. What is a counter?
2. What are the different types of counters?
3. Explain Asynchronous(ripple) counter.
4. Explain the working of Johnson counter.
5. Explain the working of Decade counter.
6. Explain Synchronous counter.

Answers to Check Your Progress

1. Counter
2. Same
3. Ring
4. Johnson

CHAPTER 12: MEMORY

12.0 Learning Objectives

After reading this chapter, you will be able to:

- Define semiconductor memories
- Define ROM
- Explain Basic ROM construction
- Define RAM
- Differentiate Static and Dynamic RAM
- Explain READ and WRITE operation in RAM
- Perform memory expansion.

12.1 Introduction to Semiconductor Memories

Semiconductor memories are made from silicon, unlike hard-disk drive memory which are magnetic and optics[9].

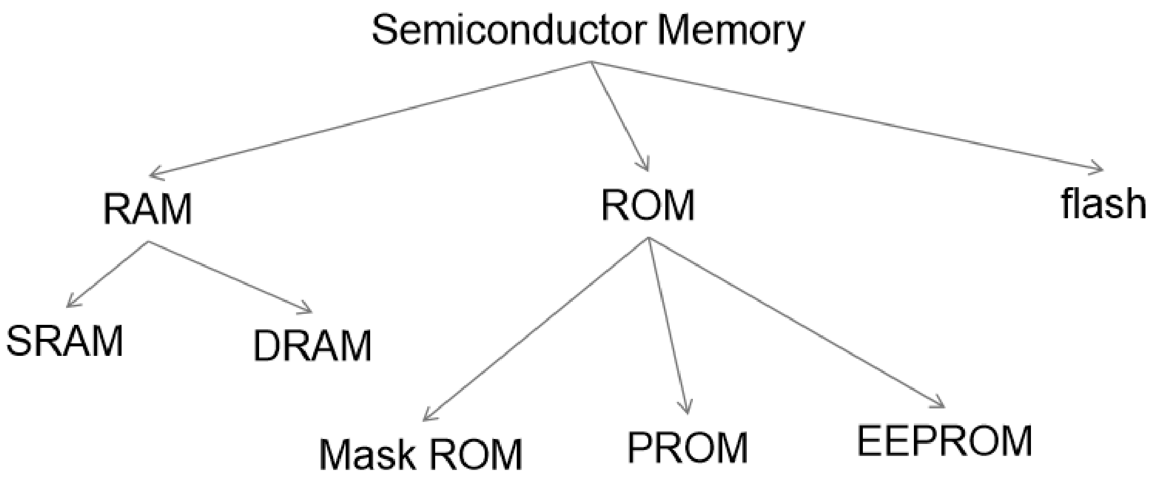


Figure 53: Semiconductor memories

Main difference between memory technologies:

- RAM: Data can be read and written but data stored is volatile, i.e. need power to retain data.

[9] Adopted from http://ocw.utm.my/course/view.php?id=48

- ROM: Data can only be read, and the data stored is not volatile, i.e. don’t need power to retain data.
- Flash: Data can be read and written, and data stored is not volatile, i.e. don’t need power to retain data.

12.2 Read Only Memory (ROM)

ROM is used to store data that never (or rarely) changed. The main advantage of ROM is data in ROMs are retained even when power is not supplied. Data in ROMs are typically pre-configured using specialized equipments. There are three commonly used ROMs:

- Mask ROM: Data is permanently stored in the memory during the manufacturing process. Once the memory array is programmed, it cannot be changed – Uses MOS transistor for memory cells.
- PROM: Uses some type of fusing process to store bits. The fusion process is irreversible, one programmed, it cannot be changed – Uses MOS transistor with fusible links for memory cells
- EEPROM: Unlike PROM, EEPROM can be reprogrammed if an existing program in the memory array is erased. EEPROM is erased and programmed using electrical pulses. Therefore, EEPROM can be rapidly programmed and erased in-circuit for reprogramming. It uses either floating gate MOS or MNOS transistors for memory cells.

12.2.1 ROM Size

Size of semiconductor memories is represented in bits. For example, an 8x4 ROM is capable of storing 32 bits. 8x4 ROM has 3 address lines ($2^3 = 8$) and 4 data lines.

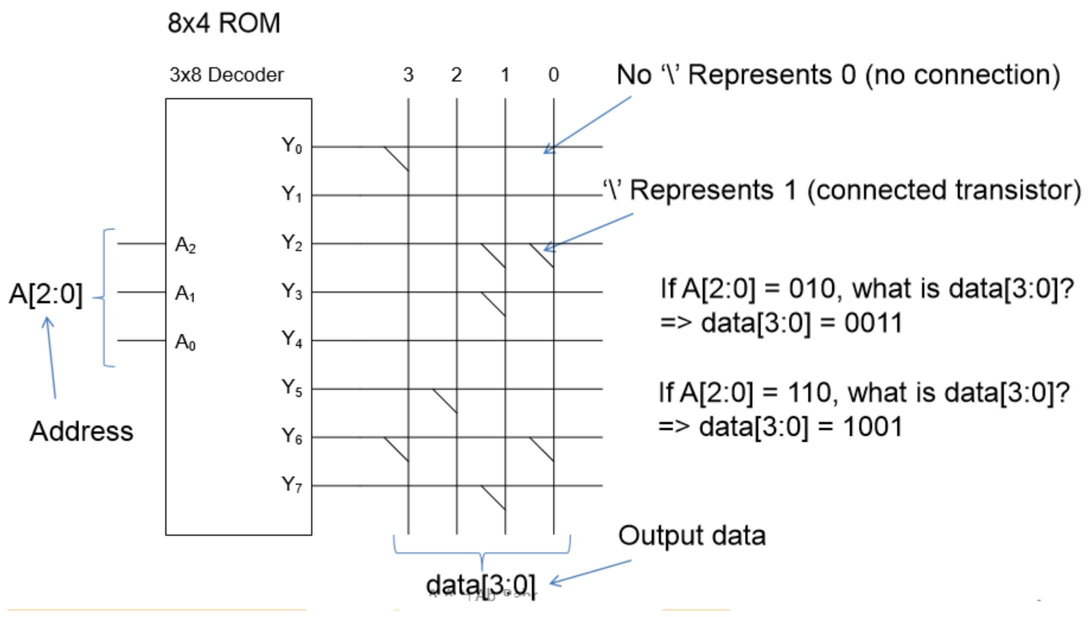
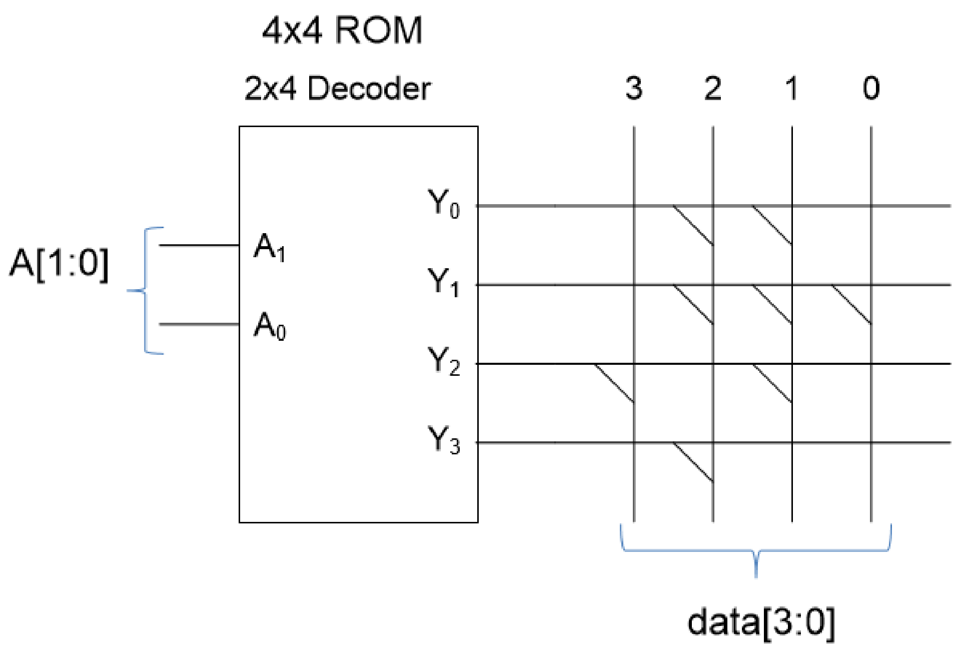

8x4 ROM
3x8 Decoder
3
2
1
0
No '\' Represents 0 (no connection)
'\' Represents 1 (connected transistor)
Y0
Y1
Y2
Y3
Y4
Y5
Y6
Y7
A2
A1
A0
A[2:0]
Address
If A[2:0] = 010, what is data[3:0]?
=> data[3:0] = 0011
If A[2:0] = 110, what is data[3:0]?
=> data[3:0] = 1001
Output data
data[3:0]

Figure 54: Basic ROM construction

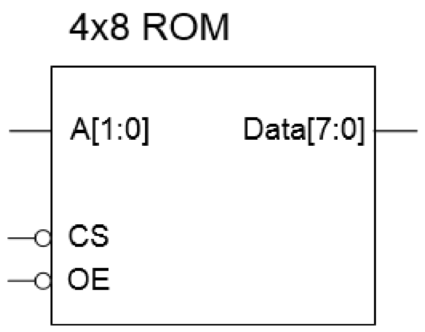
Design a ROM based on the following truth table:

Input Address		Output Data			
A[1]	A[0]	Data[3]	Data[2]	Data[1]	Data[0]
0	0	0	1	1	0
0	1	0	1	1	1
1	0	1	0	1	0
1	1	0	1	0	0

We need a 2X4 decoder for 4x4 ROM configured according to the above truth table.



ROM IC has Chip Select (CS) and Output Enable (OE) pins. Output data is valid only when CS = 0 and OE = 0, else data = z (high impedance, i.e. not ‘0’ and not ‘1’).



Assume Data[7:0] = 0011 0101 at location A[1:0] = 10. To read the data at location 10, Set A_1A_0 = 10 Set CS = 0, OE = 0 output Data[7:0] = 0011 0101.

If CS = 1 or OE = 1, data [7:0] = zzzz zzzz(high impedance), Regardless of the address input A_1A_0

12.3 Random Access Memory(RAM)

RAM is a temporary data storage and it does not retain its stored data when no power is applied. When a data unit is written into a given address in the RAM, the data unit previously stored at that address is replaced by the new unit. When a data unit is read at a given address, the data unit that is read remains there.

There are two types of RAM:

- Static RAM(SRAM)
- Dynamic RAM(DRAM)

12.3.1 SRAM vs DRAM

1. SRAM uses a latch to store 1 bit in cell while DRAM uses a capacitor to store 1 bit in a cell.
2. SRAM is more expensive to implement, i.e. requires more logic gates per cell compared to DRAM.
3. Because SRAM uses a latch, it works faster than DRAM that requires the capacitor to be periodically refreshed.
4. SRAM is typically implemented in high speed CPU cache memory, while DRAM is implemented in main memory.

12.3.2 How to read from and write to a RAM?

The first step is to provide the address and data we want to read or write. We also need a few enable signals to control when we want to enable the memory, read, and write operation. The Chip Select signal (CS) is used to enable the memory. The Write Enable signal (WE) is used to enable the write operation. The Output Enable signal (OE) is used to enable the read operation.

12.3.3 READ Operation in RAM

The figure below demonstrates the Read operation. This example explains the Read one byte at location A[1:0] = 10 from a 4x8 RAM.

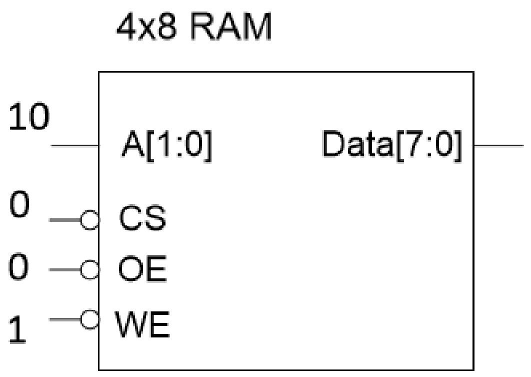


Figure 55: READ operation in RAM

Step 1: Supply address A[1:0] = 10 to Read location 10

Step 2: Set CS = 0 to enable the memory

Step 3: Set OE = 0 to enable read

Step 4: Set WE = 1 to disable write

Step 5: Get the data at location 10

12.3.4 WRITE Operation in RAM

The figure below demonstrates the Read operation. This example explains the Read one byte at location A[1:0] = 10 from a 4x8 RAM.

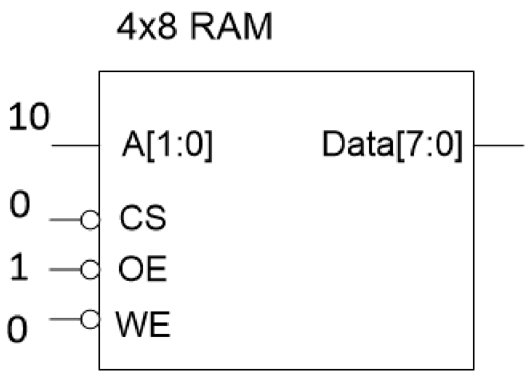


Figure 56: WRITE operation in RAM

Step 1: Supply address A[1:0] = 10 to write location 10

Step 2: Supply the Data[7:0] to write

Step 2: Set CS = 0 to enable the memory

Step 3: Set OE = 1 to disable read

Step 4: Set WE = 0 to enable write

Step 5: data[7:0] is written at location 10

12.4 Flash Memory

Flash memory is the closest to the ideal memory. It is capable of high storage capacity and retains data when power off. It has an ability to erase and reprogram at will. It also has a fast operation and is comparatively cheap. Flash memory cell is designed using stacked gate MOS transistor (floating gate transistor).

12.5 Memory Expansion

Memory can be expanded on its storage capacity or data length. First we will explain the concept of Memory storage capacity expansion with a help of an example. Suppose we have two 16x4 memory and we want to expands it to 32x4 Capacity expansion. The figure below demonstrates how two 16x4 memory are combined to obtain a 32x4 memory.

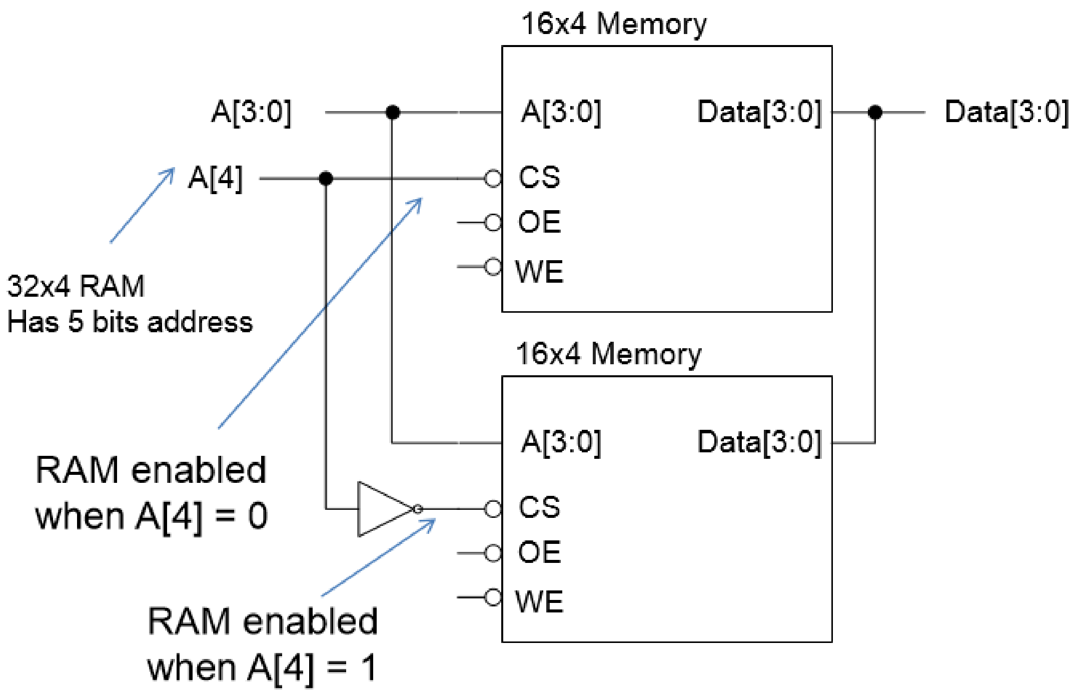


Figure 57: Memory expansion

Now we will demonstrate Memory Data length expansion with a help of an example. Suppose we have two 16x4 RAM and we want to produce a 16x8 RAM. The figure below demonstrates how two 16x4 memory are combined to obtain a 16x8 memory.

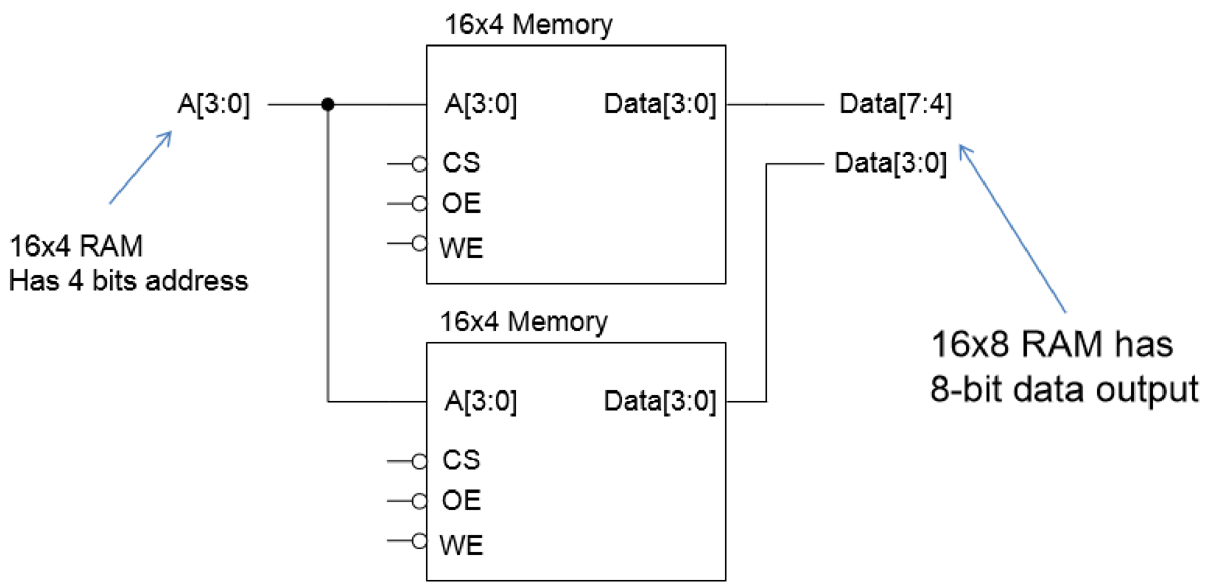


Figure 58: Data length expansion of memory

Check Your Progress

1. How many storage locations are available when a memory device has twelve address lines?
    a. 144
    b. 512
    c. 2048
    d. 4096
2. Which of the following best describes nonvolatile memory?
    a. memory that retains stored information when electrical power is removed
    b. memory that loses stored information when electrical power is removed
    c. magnetic memory
    d. nonmagnetic memory
3. Select the best description of read-only memory (ROM).
    a. nonvolatile, used to store information that changes during system operation
    b. nonvolatile, used to store information that does not change during system operation
    c. volatile, used to store information that changes during system operation
    d. volatile, used to store information that does not change during system operation
4. Memory that loses its contents when power is lost is:
    a. Nonvolatile
    b. Volatile
    c. Random
    d. Static
5. Which of the following best describes static memory devices?
    a. memory devices that are magnetic in nature and do not require constant refreshing
    b. semiconductor memory devices in which stored data is retained as long as power is applied
    c. memory devices that are magnetic in nature and require constant refreshing
    d. semiconductor memory devices in which stored data will not be retained with the power applied unless constantly refreshed

12.6 Summary

1. Semiconductor memory is a device for storing digital information that is fabricated by using integrated circuit technology.
2. Electronic semiconductor memory technology can be split into two main types or categories, according to the way in which the memory operates.
3. Random Access Memory (RAM)is the best known form of computer memory. The Read and write (R/W) memory of a computer is called RAM. The User can write information to it and read information from it.
4. The RAM is a volatile memory, it means information written to it can be accessed as long as power is on. As soon as the power is off, it can not be accessed.
5. ROM is a class of storage medium used in computers and other electronic devices. Read Only Memory (ROM), also known as firmware, is an integrated circuit programmed with specific data when it is manufactured.
6. Read only memory (ROM) is an example of nonvolatile memory.

12.7 Model Questions

1. With example define memory size, word length and memory address.
2. Specify the important characteristics of semiconductor memories.
3. What is volatile memory?
4. What is non-volatile memory?
5. List any five secondary storage devices/memory. When we use the secondary memory?
6. What is the meaning of PROM?
7. What is EEPROM?
8. What is EPROM?
9. What is Rom? Describe it's various types.
10. Describe, how information is written in memory cell.
11. Describe, how information is Read in memory cell.
12. What is the difference between a Primary memory and Secondary memory?
13. Draw a block diagram of a memory which has 4 word 3 bit per word memory and explain it.

Answers to Check Your Progress

1. a
2. a
3. b
4. b
5. b

References

Burch, C. (n.d.). *Components of digital circuits*. Retrieved Oct. 28, 2016, from http://www.toves.org/books/comps/ available under Creative Commons Attribution-Share Alike 3.0 United States License.

Eduladder. (n.d.). *Working of master-slave JK flip flop*. Retrieved Nov. 09, 2016, from http://eduladder.com/viewquestions.php?questionid=766&title=working%20of%20master-slave%20JK%20flip%20flop available under CC by sa 3.0 license.

ILLL. (n.d.). *Flip-flops and Counters*. Retrieved Nov. 09, 2016, from http://vle.du.ac.in/mod/book/view.php?id=11685&chapterid=22699 available under Creative Commons Attribution-NonCommercial-ShareAlike 2.5 India License.

Introduction to Combinational and Sequential Circuits. (n.d.). Retrieved Nov. 02, 2016, from http://m.kkhsou.in/EBIDYA/CSC/MODIFY_combinational_sequential.html avaible under creative commons license

Rao, N. (2009, Dec. 31). *Digital Electronics*. Retrieved 15 June, 2016, from NPTEL: http://nptel.ac.in/courses/106108099/1 available under Creative Commons Attribution-ShareAlike - CC BY-SA

Yewale, J. (n.d.). *Flip Flops*. Retrieved Nov. 8, 2016, from http://todayscircuits.blogspot.com/2011/06/flip-flops.html#.WCGARclgSpp available under a Creative Commons Attribution-ShareAlike 2.5 India License.

Yusof, Z. M. (2012, Feb. 18). *Digital Electronics*. Retrieved Dec. 02, 2016, from Adopted from http://ocw.utm.my/course/view.php?id=48

The University acknowledges with thanks to NPTEL and KKHSOU and the author for providing the study material on NPTEL and KKHSOU portals under Creative Commons Attribution-ShareAlike - CC BY-SA.