DIGITAL SYSTEMS

Aim: At the end of the course the students will be able to analyze, design, and evaluate digital circuits, of medium complexity, that are based on SSIs, MSIs, and programmable logic devices.

Module 1: Number Systems and Codes (3)
Number systems: Binary, octal, and hexa-decimal number systems, binary arithmetic. Codes: Binary code, excess-3 code, gray code, error detection and correction codes.

Module 2: Boolean Algebra and Logic Functions (5)
Boolean algebra: Postulates and theorems. Logic functions, minimization of Boolean functions using algebraic, Karnaugh map and Quine – Mc Clausky methods, realization using logic gates.

Module 3: Logic Families (4)
Logic families: TTL, CMOS, Tri state logic, electrical characteristics.

Module 4: Combinational Functions (8)
Realizing logical expressions using different logic gates and comparing their performance. Hardware aspects logic gates and combinational ICs: delays and hazards. Design of combinational circuits using combinational ICs: Combinational functions: code conversion, decoding, comparison, multiplexing, demultiplexing, addition, subtraction, and multiplication.

Module 5: Analysis of Sequential Circuits (5)

Module 6: Designing with Sequential MSIs (5)
Realization of sequential functions using sequential MSIs: counting, shifting sequence generation, and sequence detection.
Module 7: Memories and PLDs (4)


Module 8: Design of Digital Systems (6)

## Lecture Plan

<table>
<thead>
<tr>
<th>Modules</th>
<th>Learning Units</th>
<th>Hours per topic</th>
<th>Total Hours</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1. Number Systems and Codes</strong></td>
<td>1. Binary, octal and hexadecimal number systems, and conversion of number with one radix to another</td>
<td>1.5</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>2. Different binary codes</td>
<td>1.5</td>
<td></td>
</tr>
<tr>
<td><strong>2. Logic Functions</strong></td>
<td>3. Boolean algebra and Boolean operators</td>
<td>1.5</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4. Boolean Functions and Logic Functions</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>5. Minimization of logic functions, minimization using K-map and realization using logic gates</td>
<td>1.5</td>
<td></td>
</tr>
<tr>
<td></td>
<td>6. Minimization using Quine-McClausky</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td><strong>3. Logic Families</strong></td>
<td>7. Introduction to Logic families</td>
<td>0.5</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>8. Features of TTL family</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>9. Features of CMOS family</td>
<td>1.5</td>
<td></td>
</tr>
<tr>
<td></td>
<td>10. Electrical characteristics</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td><strong>4. Combinational Circuits</strong></td>
<td>11. Realization of simple combinational functions using gates</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>12. Implications of delay and hazard</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>13. Realization of adders and subtractors</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>14. Design of code converters, comparators, and decoders</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>15. Design of multiplexers, demultiplexers,</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td><strong>5. Analysis of Sequential Circuits</strong></td>
<td>16. Introduction to sequential circuits: Moore and Mealy machines</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>17. Introduction to flip-flops like SR, JK, D &amp; T with truth tables, logic diagrams, and timing relationships</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>18. Conversion of Flip-Flops, Excitation table</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>19. State tables, and realization of state stables</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td><strong>6. Design with Sequential MSIs</strong></td>
<td>20. Design of shift registers and counters</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>21. Design of counters</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>22. Design of sequence generators and detectors</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td><strong>7. Memories and PLDs</strong></td>
<td>23. Introduction to different types of memories</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>24. ROM, RAM, EPROM, CDROM and Flash Memory</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>25. Introduction to Programmable Devices</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>26. Architecture of PLDs</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>28. Design flow</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>29. Design of digital systems using PLDs</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>
Learning Objectives of the Course

1. Recall

1.1 List different criteria that could be used for optimization of a digital circuit.

1.2 List and describe different problems of digital circuits introduced by the hardware limitations.

2. Comprehension

2.1 Describe the significance of different criteria for design of digital circuits.

2.2 Describe the significance of different hardware related problems encountered in digital circuits.

2.3 Draw the timing diagrams for identified signals in a digital circuit.

3. Application

3.1 Determine the output and performance of given combinational and sequential circuits.

3.2 Determine the performance of a given digital circuit with regard to an identified optimization criterion.

4. Analysis

4.1 Compare the performances of combinational and sequential circuits implemented with SSIs/MSIs and PLDs.

4.2 Determine the function and performance of a given digital circuit.

4.3 Identify the faults in a given circuit and determine the consequences of the same on the circuit performance.

4.4 Draw conclusions on the behavior of a given digital circuit with regard to hazards, asynchronous inputs, and output races.

4.5 Determine the appropriateness of the choice of the ICs used in a given digital circuit.

4.6 Determine the transition sequence of a given state in a state diagram for a given input sequence.
5. **Synthesis**

5.1 Generate multiple digital solutions to a verbally described problem.

5.2 Modify a given digital circuit to change its performance as per specifications.

6. **Evaluation**

6.1 Evaluate the performance of a given digital circuit.

6.2 Assess the performance of a given digital circuit with Moore and Melay configurations.

6.3 Compare the performance of given digital circuits with respect to their speed, power consumption, number of ICs, and cost.
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 of 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 configuration, and industrial instrumentation and control systems, electro medical 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 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. This allowed the circuit designers to provide more and more complex functions in a single unit. However, with 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 programmable devices like microprocessors coexisted for considerable time, the need to make the equipment small and portable lead to replacement of fixed function devices with programmable devices. With the all pervasive presence of 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 programmes 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 the 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.
Learning Objectives

Recall

1. Describe the format of numbers of different radices?
2. What is parity of a given number?

Comprehension

1. Explain how a number with one radix is converted into a number with another radix.
2. Summarize the advantages of using different number systems.
3. Interpret the arithmetic operations of binary numbers.
4. Explain the usefulness of different coding schemes.
5. Explain how errors are detected and/or corrected using different codes.

Application

1. Convert a given number from one system to an equivalent number in another system.
2. Illustrate the construction of a weighted code.

Analysis: Nil

Synthesis: Nil

Evaluation: Nil
Highlights

We are mostly used to working with decimal numbers. However, it is not very convenient to design electronic devices with ten well-defined states and build circuits that can work with data represented in decimal form. Devices with two well-defined states are available and circuits can be built to manipulate information presented in binary form. Consequently, the interest in working with binary digits, 0s and 1s has grown tremendously over the last few decades. But very few real-life problems use binary numbers, and sometimes it does not make good sense to use any numbers. Therefore, a designer working with circuits that manipulate binary digits must establish some correspondence between the data represented in binary numbers and numbers used in real life, events and conditions. The purpose of this Module is to show you how to represent familiar decimal numbers in binary numbers. We also find it convenient to combine three or four binary digits into groups, leading to octal and hexadecimal number systems. When we have many number systems it also becomes necessary to convert from one system to another. The purpose of representing real-life numbers in binary form is to realise hardware systems that can perform a variety of operations on these numbers in a convenient manner. We find that representing the negative numbers by a negative sign is not always the best, and there are more than one method to represent negative numbers in binary form, through 0s and 1s. Learning Unit 1 presents the basics of number systems.

Straight binary representation of real-life quantities is not always the best procedure. When information is sent over a noisy channel or stored in some place to be retrieved later, the information can be corrupted by factors not in the control of designer. In such cases it becomes necessary to encode information to protect it from errors. As we are likely to come across a wide variety of situations we need many varieties of coding of information. While coding is a major subject on its own, Learning Unit 2 presents a very simple introduction to coding.

At the end of this Module the student should be able to

- represent decimal numbers in binary form
- convert numbers in one form into numbers in another form
- represent negative numbers in one’s and two’s complement forms
- explain the usefulness of representing binary information in different codes
- acquire familiarity with simple codes that are commonly used
- explain the significance of alphanumeric codes
- explain the role of parity bits for error detection and correction
- determine odd and even parity bits both for error detection and error correction
Numbers

We use numbers
- to communicate
- to perform a task
- to quantify
- to measure

Numbers have become symbols of the present era.
Many consider what is not expressible in terms of numbers is not worth knowing
Number Systems In Use

Symbolic number system
- uses Roman numerals ($I = 1$, $V = 5$, $X = 10$, $L = 50$, $C = 100$, $D = 500$ and $M = 1000$)
- still used in some watches
- Weighted position system
- Decimal number system is the most commonly used
- Decimal numbers are based on Indian numerals
- Radix used is 10
Other Weighted Position Systems

- Advent of electronic devices with two states created a possibility of working with binary numbers.
- Binary numbers are most extensively used.
- Binary system uses radix 2.
- Sometimes octal (radix 8) and hexa-decimal (radix 16) are used.
Weighted Position Number System

Value associated with a digit is dependent on its position.
The value of a number is weighted sum of its digits.

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

A decimal point allows negative and positive powers of 10.

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

10 is called the base or radix of the number system.
General Positional Number System

Any integer $\geq 2$ can serve as the radix

Digit position $i$ has weight $r_i$.

The general form of a number is

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

$p$ digits to the left of the point (radix point) and $n$ digits to the right of the point

The value of the number is

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

Leading and trailing zeros have no values

The values $d_i$ can take are limited by the radix value

A number like $(357)_5$, is incorrect
Binary Number System

Uses 2 as its radix

Has only two numerals, 0 and 1

Example:

\((N)_2 = (11100110)_2\)

It is an eight digit binary number

The binary digits are also known as \textit{bits}.

\((N)_2\) is an 8-bit number
Binary Numbers And Their Decimal Value

\((N)_2 = (11100110)_2\)

Its decimal value is given by,
\[
(N)_2 = 1 \times 2^7 + 1 \times 2^6 + 1 \times 2^5 + 0 \times 2^4 \\
+ 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 \\
= 128 + 64 + 32 + 0 + 0 + 4 + 2 + 0 \\
= (230)_{10}
\]
Binary Numbers And Their Decimal Value (Contd.)

A binary fractional number \((N)_2 = 101.101\)

Its decimal value is given by

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

\[
= 4 + 0 + 1 + 0.5 + 0.125 = 5.625_{10}
\]

\[
= 5 + 0.5 + 0.125 = (5.625)_{10}
\]
Some Features Of Binary Numbers

- Require very long strings of 1s and 0s
- Some simplification can be done through grouping
- 3-bit groupings - Octal (radix 8) to group three binary digits
- Digits will have one of the eight values 0, 1, 2, 3, 4, 5, 6 and 7
- 4-digit groupings – Hexadecimal (radix 16)
- Digits will have one of the sixteen values 0 through 15.
- Decimal values from 10 to 15 are designated as A (=10), B (=11), C (=12), D (=13), E (=14) and F (=15)
Conversion Of Binary Numbers

- Conversion to an octal number
  - Group the binary digits into groups of three
  - $(11011001)_2 = (011) (011) (001) = (331)_8$
- Conversion to an hexa-decimal number
  - Group the binary digits into groups of four
  - $(11011001)_2 = (1101) (1001) = (D9)_{16}$
General Positional Number System
Conversions

Conversion requires, sometimes, arithmetic operations

The decimal equivalent value of a number in any radix

\[ D = \sum_{i=0}^{p-1} d_i r^i \]

Examples

\[ (331)_8 = 3 \times 8^2 + 3 \times 8^1 + 1 \times 8^0 = 192 + 24 + 1 = (217)_{10} \]
\[ (D9)_{16} = 13 \times 16^1 + 9 \times 16^0 = 208 + 9 = (217)_{10} \]
\[ (33.56)_8 = 3 \times 8^1 + 3 \times 8^0 + 5 \times 8^{-1} + 6 \times 8^{-2} = (27.69875)_{10} \]
\[ (E5.A)_{16} = 14 \times 16^1 + 5 \times 16^0 + 10 \times 16^{-1} = (304.625)_{10} \]
The conversion formula from radix $r$ number to decimal number

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

Divide the right hand side by $r$,

Remainder: $d_0$

Quotient: $Q = (((d_{n-1}) \cdot r + d_{n-2}) \cdot r + \ldots) \cdot r + d_1$

Division of $Q$ by $r$ will give $d_1$ as the remainder and so on
Conversion Of Decimal Numbers To Numbers With Radix R (contd.)

<table>
<thead>
<tr>
<th>Quotient</th>
<th>Remainder</th>
</tr>
</thead>
<tbody>
<tr>
<td>156</td>
<td>2</td>
</tr>
<tr>
<td>78</td>
<td>2</td>
</tr>
<tr>
<td>39</td>
<td>2</td>
</tr>
<tr>
<td>19</td>
<td>2</td>
</tr>
<tr>
<td>9</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

\[ (156)_{10} = (10011100)_{2} \]
More Examples Of Conversion

<table>
<thead>
<tr>
<th>Quotient</th>
<th>Remainder</th>
</tr>
</thead>
<tbody>
<tr>
<td>678</td>
<td>84</td>
</tr>
<tr>
<td>84</td>
<td>10</td>
</tr>
<tr>
<td>10</td>
<td>8</td>
</tr>
<tr>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

$(678)_{10} = (1246)_{8}$

<table>
<thead>
<tr>
<th>Quotient</th>
<th>Remainder</th>
</tr>
</thead>
<tbody>
<tr>
<td>678</td>
<td>16</td>
</tr>
<tr>
<td>42</td>
<td>16</td>
</tr>
<tr>
<td>2</td>
<td>16</td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>42</td>
</tr>
<tr>
<td>16</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>

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

Sign-Magnitude representation

“+” sign before a number indicates it as a positive number

“-” sign before a number indicates it as a negative number

Not very convenient on computers

Replace “+” sign by “0” and “-” by “1”

\[(+1100101)_2 \rightarrow (01100101)_2\]

\[(+101.001)_2 \rightarrow (0101.001)_2\]

\[(-10010)_2 \rightarrow (110010)_2\]

\[(-110.101)_2 \rightarrow (1110.101)_2\]
Negative Numbers (contd.)

Treat the first digit separately

Example:

\[1000101.11 = -101.11\]

Appears natural but not convenient for representation on the computer
Methods Of Representing Signed Numbers

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

Binary numbers
DRC is known as “one’s-complement”
RXC is known as “two’s-complement”

Decimal numbers
DRC is known as “nine’s-complement”
RXC is known as “ten’s-complement”
One’s Complement Representation

The most significant bit (MSD) represents the sign
If MSD is a “0”
The number is positive
Remaining (n-1) bits directly indicate the magnitude
If the MSD is “1”
The number is negative
complement of all the remaining (n-1) bits gives the magnitude
Example: 1111001 (1)(111001)
First (sign) bit is 1: The number is negative
Complement of 111001 000110 (6)₁₀
Range Of N-bit Numbers

One’s complement numbers:

01111111 + 63
0000110 --> + 6
00000000 --> + 0
11111111 --> + 0
11110001 --> - 6
10000000 --> - 63

“0” is represented by 000.....0 and 111.....1
7- bit number covers the range from +63 to -63.
n-bit number has a range from +\((2^{n-1} - 1)\) to -\((2^{n-1} - 1)\)
One’s Complement Of A Number

Complement all the digits
If A is an integer in one’s complement form, then
one’s complement of $A = -A$
This applies to fractions as well.
$A = 0.101 (+0.625)_{10}$
One’s complement of $A = 1.010, (-0.625)_{10}$.
Mixed number
$B = 010011.0101 (+19.3125)_{10}$
One’s complement of $B = 101100.1010 (-19.3125)_{10}$
Two’s Complement Representation

The most significant bit (MSD) represents the sign
If MSD is a “0”
The number is positive
Remaining (n-1) bits directly indicate the magnitude
If the MSD is “1”
The number is negative
Magnitude is obtained by complementing all the remaining (n-1) bits and adding a 1

Example: 1111010   (1)(111010)
First (sign) bit is 1: The number is negative
Complement 111010 and add 1  \(000101 + 1 = 000110 = (6)_{10}\)
Range Of N-bit Numbers

Two’s complement numbers:

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

“0” is represented by 000.....0

7-bit number covers the range from +63 to -64.

n-bit number has a range from +(2^{n-1} - 1) to -(2^{n-1})
Two’s Complement Of A Number

Complement all the digits and add ‘1’ to the LSB

If A is an integer in one’s complement form, then

Two’s complement of A = -A

This applies to fractions as well.

A = 0.101 (+0.625)_{10}

Two’s complement of A = 1.011, (-0.625)_{10}.

Mixed number

B = 010011.0101 (+19.3125)_{10}

Two’s complement of B = 101100.1011 (- 19.3125)_{10}
Number Systems

Introduction

Using numbers either to communicate or to perform a task has become essential to most of the human activities. Quantification and measurement, both of which require the use of numbers, have become symbols of the present era. Many consider what is not expressible 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. As the usage of decimal numbers starts very early in one’s life, 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 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 Arabic (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 radix 16). The usage of binary numbers at various levels has become so essential that it is also necessary to have good understanding of all the binary arithmetic operations.

This Learning Unit presents the weighted-position number systems and conversion from one system to the other.

Weighted-Position Number System

In a weighted-position numbering system using Arabic (Indian) numerals the value associated with a digit is dependent on its position. The value of a number is a 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 \textit{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 \textit{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 (obviously 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 (.), though in some countries a comma is used, for example 7,6 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 \textit{bits}. Consequently the above number would be referred to as an 8-bit number. Its decimal value is given by,

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

\[
= 128 + 64 + 32 + 0 + 0 + 4 + 2 + 0 = (230)_{10}
\]

Similarly 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 digits may be eased by grouping them into three digits or four digits. This kind of grouping leads to the usage of Octal (radix 8 to group three binary digits) and Hexadecimal (radix 16 to group four binary digits) representations. 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 in the following:

\[
(11011011)_2 = (011) (011) (001) = (331)_8
\]

\[
= (1101) (1001) = (D9)_{16}
\]

N. J. Rao/IISc, Bangalore
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.

General Positional Number System Conversions

In general, conversion between two radices cannot be done by simple substitutions; arithmetic operations are required. In this Learning Unit we 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

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

where $r$ is the radix of the number and there are $\mu$ digits to the left of the radix point and $n$ digits to the right. Thus, the value of the number can be found by converting each digit of the number to its radix-10 equivalent and expanding the formula using radix-10 arithmetic. Some examples are given below:

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

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

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

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

The above conversion formula can be rewritten as,

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

While this formula does not simplify the conversion of a number in some radix to a decimal number, it forms the basis for converting a decimal number $D$ to a radix $r$. If we divide the right hand side of the above formula by $r$, the remainder will be $d_0$, and the quotient will be

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

Thus, $d_0$ 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 example.

<table>
<thead>
<tr>
<th>Quotient</th>
<th>Remainder</th>
</tr>
</thead>
<tbody>
<tr>
<td>156 ÷ 2</td>
<td>78</td>
</tr>
<tr>
<td>78 ÷ 2</td>
<td>39</td>
</tr>
<tr>
<td>39 ÷ 2</td>
<td>19</td>
</tr>
<tr>
<td>19 ÷ 2</td>
<td>9</td>
</tr>
<tr>
<td>9 ÷ 2</td>
<td>4</td>
</tr>
<tr>
<td>4 ÷ 2</td>
<td>2</td>
</tr>
<tr>
<td>2 ÷ 2</td>
<td>1</td>
</tr>
<tr>
<td>1 ÷ 2</td>
<td>0</td>
</tr>
</tbody>
</table>

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

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>678</td>
<td>84</td>
<td>6</td>
</tr>
<tr>
<td>84</td>
<td>10</td>
<td>4</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

\((678)_{10} = (1246)_{8}\)

Quotient Remainder

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>678</td>
<td>42</td>
<td>6</td>
</tr>
<tr>
<td>42</td>
<td>2</td>
<td>A</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

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

**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. To make things simple when a number is positive, the sign before the number is omitted. 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. The coding that is followed is to replace “+” sign by “0” and “-” by “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 is not that very convenient, either for implementation on the computer or for hardware implementation. There are two other methods of representing signed numbers. These are Diminished Radix Complement (DRC) or \((r-1)\)-complement and Radix Complement (RX) or \(r\)-complement. With the numbers in binary form, the Diminished Radix Complement will be known as “one’s-complement” and Radix complement will be known as “two’s-complement”. Similarly if this representation is extended to the decimal numbers they will be known as nine’s-complement and ten’s-complement respectively.

**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:

| 011111 1 | + 63   |
| 000011 0 | +  6   |
| 000000 0 | +  0   |
| 111111 1 | +  0   |
| 111100 1 | -  6   |
| 100000 0 | -  63  |

From these illustrations it may be observed that 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. We have to take the complement of 111001, which becomes 000110, to determine the magnitude. As a consequence of this there are two representations of “0”, namely 000.....0 and 111.....1. 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)\text{10}, then the one’s complement of A is 1.010, which is one’s complement representation of (-0.625)\text{10}. Similarly consider the case of a mixed number.

\[
A = 010011.0101 \quad (+19.3125)_{10} \\
One’s complement of A = 101100.1010 \quad (-19.3125)_{10}
\]

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

**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:

| 011111 1 | + 63   |
| 000011 0 | +  6   |
| 000000 0 | +  0   |
| 000000 0 | +  0   |
From these illustrations it may be observed that 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. Leaving the sign bit, the remaining bits, as indicated above, are 111010. These have to be complemented (that is 000101) and a 1 has to be added (that is 000101 + 1 = 000110 = 6). There is only one representation of “0”, namely 000...0. 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. For example 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
CODES

Need for Coding

Information sent over a noisy channel is likely to be distorted

Information is coded to facilitate

• Efficient transmission
• Error detection
• Error correction
Coding

- Coding is the process of assigning a group of binary digits to multivalued items of information

Coding schemes depend on

- Security requirements
- Complexity of the medium of transmission
- Levels of error tolerated
- Need for standardization
Decoding

- Decoding is the process of reconstructing source information from the received encoded information.
- Decoding can be more complex than coding if there is no prior knowledge of coding schemes.
Bit combinations

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;
(Sometimes used to designate 32 bit or 64 bit groups of bits)
Binary coding

Assign each item of information a unique combination of 1s and 0s

n is the number of bits in the code word

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 \( x = 2^j \)
Number of bits in a code word

x: number of elements to be coded binary coded format

\[ x \leq 2^j \]

or

\[ j \geq \log_2 x \]

\[ \geq 3.32 \log_{10} x \]

j is the number of bits in a code word.
Example: Coding of alphanumeric information

(26 alphabetic characters + 10 decimals digits = 36 elements of information)

\[ j \geq 3.32 \log_{10} 36 \]

\[ j \geq 5.16 \text{ bits} \]

Number of bits required for coding = 6

Only 36 code words are used out of the 64 possible code words
Codes for consideration

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

Simple scheme
- Convert decimal number inputs into binary form
- Manipulate these binary numbers
- Convert resultant binary numbers into decimal numbers

However, it
  - requires more hardware
  - slows down the system
Binary coded decimal codes

- Encode each decimal symbol in a unique string of 0s and 1s
- Ten symbols require at least four bits to encode
- There are sixteen four-bit groups to select ten groups.
- Gives $30 \times 10^{10} \binom{16}{10} \cdot 10!$ possible codes
- Most of these codes will not have any special properties
Example of BCD code

Natural Binary Coded Decimal code (NBCD)
Consider the number \((16.85)_{10}\)

\[(16.85)_{10} = (0001 \ 0110 . 1000 \ 0101)_{NBCD}\]

\[1 \ 6 \ 8 \ 5\]

NBCD code is used in calculators
How do we select a coding scheme?

It should have some desirable properties

• ease of coding
• ease in arithmetic operations
• minimum use of hardware
• error detection property
• ability to prevent wrong output during transitions
Weighted Binary Coding

Decimal number \((A)_{10}\)

Encoded in the binary form as \(a_3 a_2 a_1 a_0\)

\(w_3, w_2, w_1\) and \(w_0\) are the weights selected for a given code

\((A)_{10} = w_3 a_3 + w_2 a_2 + w_1 a_1 + w_0 a_0\)
Weighted Binary Coding

The more popularly used codes have the weights as

\[
\begin{array}{cccc}
  w_3 & w_2 & w_1 & w_0 \\
  8 & 4 & 2 & 1 \\
  2 & 4 & 2 & 1 \\
  8 & 4 & -2 & -1 \\
\end{array}
\]
# Binary codes for decimal numbers

<table>
<thead>
<tr>
<th>Decimal digit</th>
<th>Weight 8 4 2 1</th>
<th>Weights 2 4 2 1</th>
<th>Weights 8 4 -2 -1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0 0 0 0</td>
<td>0 0 0 0</td>
<td>0 0 0 0</td>
</tr>
<tr>
<td>1</td>
<td>0 0 0 1</td>
<td>0 0 0 1</td>
<td>0 1 1 1</td>
</tr>
<tr>
<td>2</td>
<td>0 0 1 0</td>
<td>0 0 1 0</td>
<td>0 1 1 0</td>
</tr>
<tr>
<td>3</td>
<td>0 0 1 1</td>
<td>0 0 1 1</td>
<td>0 1 0 1</td>
</tr>
<tr>
<td>4</td>
<td>0 1 0 0</td>
<td>0 1 0 0</td>
<td>0 1 0 0</td>
</tr>
<tr>
<td>5</td>
<td>0 1 0 1</td>
<td>1 0 1 1</td>
<td>1 0 1 1</td>
</tr>
<tr>
<td>6</td>
<td>0 1 1 0</td>
<td>1 1 0 0</td>
<td>1 0 1 0</td>
</tr>
<tr>
<td>7</td>
<td>0 1 1 1</td>
<td>1 1 0 1</td>
<td>1 0 0 1</td>
</tr>
<tr>
<td>8</td>
<td>1 0 0 0</td>
<td>1 1 1 0</td>
<td>1 0 0 0</td>
</tr>
<tr>
<td>9</td>
<td>1 0 0 1</td>
<td>1 1 1 1</td>
<td>1 1 1 1</td>
</tr>
</tbody>
</table>
Binary coded decimal numbers

The unused six combinations are illegal
They may be utilised for error detection purposes.
Choice of weights in a BCD codes
1. Self-complementing codes
2. Reflective codes
Self complementing codes

- Logical complement of a coded number is also its arithmetic complement
- 2421 code.

Example:
- Nine’s complement of \((4)_{10} = (5)_{10}\)
- 2421 code of \((4)_{10} = 0100\)
- Complement of 0100 = 1011 = 2421 code for \((5)_{10} = (9 - 4)_{10}\).

A necessary condition: Sum of its weights should be 9.
Other self complementing codes

Excess-3 code (not weighted)

- Add 0011 (3) to all the 8421 coded numbers

631-1 weighted code
Examples of self-complementary codes

<table>
<thead>
<tr>
<th>Decimal Digit</th>
<th>Excess-3 Code</th>
<th>631-1 Code</th>
<th>2421 Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0011</td>
<td>0011</td>
<td>0000</td>
</tr>
<tr>
<td>1</td>
<td>0100</td>
<td>0010</td>
<td>0001</td>
</tr>
<tr>
<td>2</td>
<td>0101</td>
<td>0101</td>
<td>0010</td>
</tr>
<tr>
<td>3</td>
<td>0110</td>
<td>0111</td>
<td>0011</td>
</tr>
<tr>
<td>4</td>
<td>0111</td>
<td>0110</td>
<td>0100</td>
</tr>
<tr>
<td>5</td>
<td>1000</td>
<td>1001</td>
<td>1011</td>
</tr>
<tr>
<td>6</td>
<td>1001</td>
<td>1000</td>
<td>1100</td>
</tr>
<tr>
<td>7</td>
<td>1010</td>
<td>1010</td>
<td>1101</td>
</tr>
<tr>
<td>8</td>
<td>1011</td>
<td>1101</td>
<td>1110</td>
</tr>
<tr>
<td>9</td>
<td>1100</td>
<td>1100</td>
<td>1111</td>
</tr>
</tbody>
</table>
Reflective code

Imaged about the centre entries with one bit changed

Example

• 9’s complement of a reflected BCD code word is formed by changing only one of its bits
### Examples of reflective BCD codes

<table>
<thead>
<tr>
<th>Decimal Digit</th>
<th>Code-A</th>
<th>Code-B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>0100</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>1010</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>1000</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
<td>1110</td>
</tr>
<tr>
<td>4</td>
<td>0100</td>
<td>0000</td>
</tr>
<tr>
<td>5</td>
<td>1100</td>
<td>0001</td>
</tr>
<tr>
<td>6</td>
<td>1011</td>
<td>1111</td>
</tr>
<tr>
<td>7</td>
<td>1010</td>
<td>1001</td>
</tr>
<tr>
<td>8</td>
<td>1001</td>
<td>1011</td>
</tr>
<tr>
<td>9</td>
<td>1000</td>
<td>0101</td>
</tr>
</tbody>
</table>
Unit Distance Codes

Adjacent codes differ only in one bit, “Gray code” is the most popular example
3-bit and 4-bit Gray codes

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

These Gray codes have also the reflective properties
More examples of Unit Distance Codes

<table>
<thead>
<tr>
<th>Decimal Digit</th>
<th>UDC-1</th>
<th>UDC-2</th>
<th>UDC-3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
</tr>
<tr>
<td>1</td>
<td>0100</td>
<td>0001</td>
<td>1000</td>
</tr>
<tr>
<td>2</td>
<td>1100</td>
<td>0011</td>
<td>1001</td>
</tr>
<tr>
<td>3</td>
<td>1000</td>
<td>0010</td>
<td>0001</td>
</tr>
<tr>
<td>4</td>
<td>1001</td>
<td>0110</td>
<td>0011</td>
</tr>
<tr>
<td>5</td>
<td>1011</td>
<td>1110</td>
<td>0111</td>
</tr>
<tr>
<td>6</td>
<td>1111</td>
<td>1111</td>
<td>1111</td>
</tr>
<tr>
<td>7</td>
<td>0111</td>
<td>1101</td>
<td>1011</td>
</tr>
<tr>
<td>8</td>
<td>0011</td>
<td>1100</td>
<td>1010</td>
</tr>
<tr>
<td>9</td>
<td>0001</td>
<td>0100</td>
<td>0010</td>
</tr>
</tbody>
</table>
3-bit simple binary coded shaft encoder

Can lead to errors (001 $\rightarrow$ 011 $\rightarrow$ 010)
Shaft encoder disk using a 3-bit Gray code
Constructing Gray Code

1. The bits of Gray code words are numbered from right to left, from 0 to n-1.
2. Bit i is 0 if bits i and i+1 of the corresponding binary code word are the same, else bit i is 1
3. 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
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.
Convert a Gray coded number to a straight binary number

Examples:

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

Gray code : 1 0 0 0 1 0 1 1
Binary code: 1 1 1 1 0 0 1 0
Alphanumeric Codes

Keyboards use ASCII (American Standard Code for Information Interchange) code

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

Other alphanumeric codes

EBCDIC (Extended Binary Coded Decimal Interchange Code)

12-bit Hollerith code were in use for some applications
Error Detection and Correction

- Error rate cannot be reduced to zero
- Need a mechanism of correcting the errors that occur
- It is not always possible or may prove to be expensive
- It is necessary to know if an error occurred
- If an occurrence of error is known data may be retransmitted
- Data integrity is improved by encoding
- Encoding may be done for error correction or merely for error detection.
Encoding for data integrity

• Add a special code bit to a data word
• It is called the ‘Parity Bit”
• Parity bit can be added on an odd or even basis
Odd parity

The number of 1’s, including the parity bit, should be odd

Example: 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

Even Parity

The number of 1’s, including the parity bit, should be even

When S is encoded for even parity

\[(S) = (01010011)_{\text{ASCII}}\] with even parity.
Error detection with parity bits

If odd number of 1’s occur in the received data word coded for even parity then an error occurred.

Single or odd number bit errors can be detected.

Two or even number bit errors will not be detected.
Error Correction

Parity bit allows us only to detect the presence of one bit error in a group of bits. It does not enable us to exactly locate the bit that change.

Parity bit scheme can be extended to locate the faulty bit in a block of information.
Single error detecting and single error correcting coding scheme

The bits are conceptually arranged in a two-dimensional array, and parity bits are provided to check both the rows and the columns.

![Diagram showing a two-dimensional array with parity bits for rows and columns.](image)
Parity-check block codes

Detect and correct more than one-bit errors

These are known as \((n, k)\) codes.

They have \(r = n-k\) parity check bits, formed by linear operations on the \(k\) data bits.

\(R\) bits are appended to each block of \(k\) bits to generate and \(n\)-bit code word.
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.

Long codes with a relatively large number of parity check bits should provide better performance.
Single-error correcting code

(7, 3) code

<table>
<thead>
<tr>
<th>Data bits</th>
<th>Code words</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0</td>
<td>0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>0 0 1</td>
<td>0 0 1 1 1 1 1</td>
</tr>
<tr>
<td>0 1 0</td>
<td>0 1 0 0 1 1 0</td>
</tr>
<tr>
<td>0 1 1</td>
<td>0 1 1 1 0 0 1</td>
</tr>
<tr>
<td>1 0 0</td>
<td>1 0 0 1 1 0 0</td>
</tr>
<tr>
<td>1 0 1</td>
<td>1 0 1 0 0 1 1</td>
</tr>
<tr>
<td>1 1 0</td>
<td>1 1 0 1 0 1 0</td>
</tr>
<tr>
<td>1 1 1</td>
<td>1 1 1 0 1 0 1</td>
</tr>
</tbody>
</table>

Code words differ in at least three positions. Any one error is correctable since the resultant code word will still be closer to the correct one.
Hamming distance

Difference in the number of positions between any two code words

For two errors to be correctable, the Hamming distance $d$ should be at least 5

For $t$ errors correctable, $d \geq 2t+1$ or $t = [(d-1)/2]$

$[x]$ refers to the integer less than or equal to $x$. 
Codes with different properties

Codes exit for

• correcting independently occurring errors
• correcting burst errors,
• providing relatively error-free synchronization of binary data
• etc.

Coding Theory is very important to communication systems

It is a discipline by itself
Codes

Introduction

The need to send information through numbers over long distances unambiguously required further modifications to the simple weighted-positional numbering systems. These modifications 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 or encoding is the process of assigning a group of binary digits to represent multivalued items of information. By assigning each item of information a unique combination of 1s and 0s (commonly referred to as bits), 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. When information is transmitted from one point to another (short or long distances), we encode information 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. We encode information for security reasons to limit the accessibility of information. Coding may also be used to standardise a universal code that can be used by all. Depending on the security requirements and the complexity of the medium over which information is transmitted the coding schemes have to be designed. 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 this chapter. 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 
\[ n = 1, \text{ then } x = 2 \ (0, 1) \]
\[ n = 2, \text{ then } x = 4 \ (00, 01, 10, 11) \]
\[ n = 3, \text{ then } x = 8 \ (000,001,010 \ldots 111) \]
\[ \ldots \]
\[ n = j, \text{ then } x = 2^j \]

From this we can conclude that if we are given elements of information to code into binary coded format,
\[
x \leq 2^j \\
\text{or} \quad j \geq \log_2 x \\
\geq 3.32 \log_{10} x
\]

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

\[
j \geq 3.32 \log_{10} 36 \\
j \geq 5.16 \text{ bits}
\]

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 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

**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. Therefore, any hardware that would manipulate binary numbers requires devices that have two well defined stable states. While it is advantageous to perform all computations 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. Unit 1 presented methods for converting binary numbers to decimal numbers and vice-versa. 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. One method of solving this problem is to encode decimal numbers as binary strings, and use these binary forms for subsequent manipulations.

There are ten different symbols in the decimal number system: 0, 1, 2 \ldots 9. Encoding is the procedure of representing each one of these ten symbols by a unique string of 0s and 1s. As there are ten symbols we require at least four bits to represent them in the binary form. Such a representation of decimal number 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 \( 30 \times 10^{10} \binom{16}{10} \) possible codes. However, most of them will not have any special properties that would be useful in hardware design. The coding that can be used should have some desirable properties.
like ease of coding, ease in arithmetic operations, minimum use of hardware, error
detection property, or 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

\[
\begin{array}{cccc}
\text{Weights} & 8 & 4 & 2 & 1 \\
\hline
\text{Weights} & 2 & 4 & 2 & 1 \\
\hline
\text{Weights} & 8 & 4 & -2 & -1 \\
\end{array}
\]

The decimal numbers in these three codes are listed in the Table 1

<table>
<thead>
<tr>
<th>Decimal digit</th>
<th>Weight 8 4 2 1</th>
<th>Weights 2 4 2 1</th>
<th>Weights 8 4 -2 -1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0 0 0 0</td>
<td>0 0 0 0</td>
<td>0 0 0 0</td>
</tr>
<tr>
<td>1</td>
<td>0 0 0 1</td>
<td>0 0 0 1</td>
<td>0 1 1 1</td>
</tr>
<tr>
<td>2</td>
<td>0 0 1 0</td>
<td>0 0 1 0</td>
<td>0 1 1 0</td>
</tr>
<tr>
<td>3</td>
<td>0 0 1 1</td>
<td>0 0 1 1</td>
<td>0 1 0 1</td>
</tr>
<tr>
<td>4</td>
<td>0 1 0 0</td>
<td>0 1 0 0</td>
<td>0 1 0 0</td>
</tr>
<tr>
<td>5</td>
<td>0 1 0 1</td>
<td>1 0 1 1</td>
<td>1 0 1 1</td>
</tr>
<tr>
<td>6</td>
<td>0 1 1 0</td>
<td>1 1 0 0</td>
<td>1 0 1 0</td>
</tr>
<tr>
<td>7</td>
<td>0 1 1 1</td>
<td>1 1 0 1</td>
<td>1 0 0 1</td>
</tr>
<tr>
<td>8</td>
<td>1 0 0 0</td>
<td>1 1 1 0</td>
<td>1 0 0 0</td>
</tr>
<tr>
<td>9</td>
<td>1 0 0 1</td>
<td>1 1 1 1</td>
<td>1 1 1 1</td>
</tr>
</tbody>
</table>

In all the cases only ten combinations are utilised to represent the decimal digits. The
remaining six combinations are illegal. However, they may be utilised for error
detection purposes.

Consider, for example, the representation of the decimal number 16.85 in Natural
Binary Coded Decimal code (NBCD)

\[
(16.85)_{10} = (0001 \ 0110 \ . \ 1000 \ 0101)_{NBCD}
\]

As illustrated in Table 1, there are many possible weights to write a number in BCD
code. There are some desirable properties for some of the codes, 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 \((9 - A)_{10}\). For 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 given in the Table 2.

<table>
<thead>
<tr>
<th>Decimal Digit</th>
<th>Excess-3 Code</th>
<th>631-1 Code</th>
<th>2421 Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0011</td>
<td>0011</td>
<td>0000</td>
</tr>
<tr>
<td>1</td>
<td>0100</td>
<td>0010</td>
<td>0001</td>
</tr>
<tr>
<td>2</td>
<td>0101</td>
<td>0101</td>
<td>0010</td>
</tr>
<tr>
<td>3</td>
<td>0110</td>
<td>0111</td>
<td>0011</td>
</tr>
<tr>
<td>4</td>
<td>0111</td>
<td>0110</td>
<td>0100</td>
</tr>
<tr>
<td>5</td>
<td>1000</td>
<td>1001</td>
<td>1011</td>
</tr>
<tr>
<td>6</td>
<td>1001</td>
<td>1000</td>
<td>1100</td>
</tr>
<tr>
<td>7</td>
<td>1010</td>
<td>1010</td>
<td>1101</td>
</tr>
<tr>
<td>8</td>
<td>1011</td>
<td>1101</td>
<td>1110</td>
</tr>
<tr>
<td>9</td>
<td>1100</td>
<td>1100</td>
<td>1111</td>
</tr>
</tbody>
</table>

A reflective code is characterised 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 given in the Table 3.

<table>
<thead>
<tr>
<th>Decimal Digit</th>
<th>Code-A</th>
<th>Code-B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>0100</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>1010</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>1000</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
<td>1110</td>
</tr>
<tr>
<td>4</td>
<td>0100</td>
<td>0000</td>
</tr>
<tr>
<td>5</td>
<td>1100</td>
<td>0001</td>
</tr>
<tr>
<td>6</td>
<td>1011</td>
<td>1111</td>
</tr>
<tr>
<td>7</td>
<td>1010</td>
<td>1001</td>
</tr>
<tr>
<td>8</td>
<td>1001</td>
<td>1011</td>
</tr>
<tr>
<td>9</td>
<td>1000</td>
<td>0101</td>
</tr>
</tbody>
</table>

The BCD codes are widely used and the reader should become familiar with reasons for using them and their application. The most common application NBCD codes is in the calculator.
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-bit Gray codes are given in the Table 4.

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

These Gray codes listed in Table 4 have also the reflective properties. Some additional examples of unit distance codes are given in the Table 2.5. 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.

<table>
<thead>
<tr>
<th>Decimal Digit</th>
<th>UDC-1</th>
<th>UDC-2</th>
<th>UDC-3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
</tr>
<tr>
<td>1</td>
<td>0100</td>
<td>0011</td>
<td>1000</td>
</tr>
<tr>
<td>2</td>
<td>1100</td>
<td>0011</td>
<td>1001</td>
</tr>
<tr>
<td>3</td>
<td>1000</td>
<td>0010</td>
<td>0001</td>
</tr>
<tr>
<td>4</td>
<td>1001</td>
<td>0110</td>
<td>0011</td>
</tr>
<tr>
<td>5</td>
<td>1011</td>
<td>1110</td>
<td>0111</td>
</tr>
<tr>
<td>6</td>
<td>1111</td>
<td>1111</td>
<td>1111</td>
</tr>
<tr>
<td>7</td>
<td>0111</td>
<td>1101</td>
<td>1011</td>
</tr>
<tr>
<td>8</td>
<td>0011</td>
<td>1100</td>
<td>1010</td>
</tr>
<tr>
<td>9</td>
<td>0001</td>
<td>0100</td>
<td>0010</td>
</tr>
</tbody>
</table>
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:

1. A one-bit Gray code had code words, 0 and 1
2. The first $2^n$ 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.
3. The last $2^n$ 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:

1. The bits of an n-bit binary code or Gray code words are numbered from right to left, from 0 to n-1.
2. 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:

1. 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.

2. 1’s are written until the next 1 is encountered, in which case a 0 is written.

3. 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

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 bit to encode all the characters. As there is need for alphanumeric codes in a wide variety of applications, 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 popularisation of personal computers and notebook computers. These keyboards use ASCII (American Standard Code for Information Interchange) code, given in the Table 6.
Alphanumeric codes like EBCDIC (Extended Binary Coded Decimal Interchange Code) and 12-bit Hollerith code were 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.

**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. As error rate cannot be reduced to zero, 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 the 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}} \text{ 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}} \text{ 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 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, as indicated in the figure 3.

![FIG. 3: Two dimensional coding for error correction](image)

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 generalised for detecting and correcting more than one-bit errors. 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

<table>
<thead>
<tr>
<th>Data bits</th>
<th>Code words</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0</td>
<td>0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>0 0 1</td>
<td>0 0 1 1 1 1 1</td>
</tr>
<tr>
<td>0 1 0</td>
<td>0 1 0 0 1 1 0</td>
</tr>
<tr>
<td>0 1 1</td>
<td>0 1 1 1 0 0 1</td>
</tr>
<tr>
<td>1 0 0</td>
<td>1 0 0 1 1 0 0</td>
</tr>
<tr>
<td>1 0 1</td>
<td>1 0 1 0 0 1 1</td>
</tr>
<tr>
<td>1 1 0</td>
<td>1 1 0 1 0 1 0</td>
</tr>
<tr>
<td>1 1 1</td>
<td>1 1 1 0 1 0 1</td>
</tr>
</tbody>
</table>

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 correctable, \( d \geq 2t+1 \) or \( t = [(d-1)/2] \), 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.
Module 2: Logic Functions (4)

Logic functions, minimization of Boolean functions using algebraic, Karnaugh map and Quine – McClausky methods. Realization using logic functions

**Learning Objectives**

**Recall**

1. Explain the method of minimization of logic functions using K-map.
2. What is a prime implicant?
3. What is an essential prime implicant?
4. How is a logic function minimized using Karnaugh Map?
5. How is a logic function minimized using Quine-McClusky method?

**Comprehension**

1. Write logic expressions for the outputs identified in a truth table either as min terms or a max term.
2. Simplify a given logic expression using the postulates of Boolean algebra.
3. Expand a given algebraic expression in terms of min terms or max terms.
4. Create an alternate logic expression from a given logic expression using De Morgan’s theorem
5. What is logic adjacency in a K-Map?
6. How are logic adjacency and logic minimization are related?
7. Identify the essential prime implicants of a function from a map.
8. Explain the relation between operations performed using the map and the corresponding operations performed using the postulates of Boolean algebra.

**Application**

1. Create K-maps of logical expressions with 3 to 6 variables.
2. Minimize a given logical expression using Quine-McClusky method.

**Analysis**

1. Identify the differences among different realizations of a given logical expression
2. Write logical expression for a given realization in terms of different logic functions.

**Synthesis**

1. Create a logical expression for the output of a digital system whose verbal description is given.

**Evaluation**

Nil
Boolean Algebra And Boolean Operators

• We encounter situations where the choice is binary
  Move/Stop
  On/Off
  Yes/No
• An intended action takes place or does not take place
• Signals with two possible states are called “switching signals”
• We need to work with a large number of switching signals
• There arises a need for formal methods of handling such signals
Examples of Switching Signals

A control circuit for an electric bulb

Four switches control the operation of the bulb
“the bulb is switched on if the switches S1 and S2 are closed, and S3 or S4 is also closed, otherwise the bulb will not be switched on”.

Relay operations in telephone exchanges
George Boole

English mathematician (1854)
“An Investigation of the Laws of Thought”
Examined the truth or falsehood of language statements
Used special algebra of logic - Boole's Algebra (Boolean Algebra)
    assigned a value 1 to statements that are completely correct
    assigned a value 0 is assigned to statements that are completely false
Statements are referred to digital variables
We consider logical or digital variables to be synonymous
Claude Shannon

Master’s Thesis at Massachusetts Institute of Technology in 1938 “A Symbolic Analysis of Relay and Switching Circuits” Applied Boolean algebra to the analysis and design of electrical switching circuits
Realisation of Switching Circuits

ICs built with bipolar and MOS transistors are used necessary to understand the electrical aspects of these Circuits. They include:

- voltage levels,
- current capacities,
- switching time delays,
- noise margins etc.

Progress in semiconductor technology gives us better and cheaper ICs.
Learning Objectives

To know the basic axioms of Boolean algebra

To simplify logic functions (Boolean functions) using the basic properties of Boolean Algebra
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 ".", such that for every element \( x \) and \( y \in BS \), the operations (not \( x \)), \( x + y \) and \( x \cdot y \) are uniquely defined.

The unary operator ‘not’ is defined by the relation \( x = 0 \); \( x = 1 \).

The \textit{not} operator is also called the complement \( \overline{1} \)

\( \overline{0} \) is the complement of \( x \)

\( \overline{x} \)
Binary Operators “and” and “or”

The \textit{and} operator is defined by
\begin{align*}
0 \cdot 0 &= 0 \\
0 \cdot 1 &= 0 \\
1 \cdot 0 &= 0 \\
1 \cdot 1 &= 1
\end{align*}

The ‘or’ operator is defined by
\begin{align*}
0 + 0 &= 0 \\
0 + 1 &= 1 \\
1 + 0 &= 1 \\
1 + 1 &= 1
\end{align*}
Huntington's (1909) Postulates

P1. The operations are closed.
   For all $x$ and $y \in BS$,
   
   $x + y \in BS$
   $x \cdot y \in BS$

P2. For each operation there exists an identity element.
   There exists an element $0 \in BS$ such that for all
   $x \in BS, x + 0 = x$
   There exists an element $1 \in BS$ such that for all
   $x \in BS, x \cdot 1 = x$
Huntington's Postulates (Contd…)

P3. The operations are commutative.
For all $x$ and $y \in BS$,

\[ x + y = y + x \]
\[ x \cdot y = y \cdot x \]

P4. The operations are distributive.
For all $x$, $y$ and $z \in BS$,

\[ x + (y \cdot z) = (x + y) \cdot (x + z) \]
\[ x \cdot (y + z) = (x \cdot y) + (x \cdot z) \]
Huntington's Postulates (Contd...) 

P5. For every element $x \in BS$ there exists an element $\overline{x} \in BS$ (called the complement of $x$) such that 

$x + \overline{x} = 1$

$x \cdot \overline{x} = 0$

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

$x \neq y.$
Boolean Expression

A constant (0, 1)
1 or 0
A single Boolean variable or its complement (x, \(\overline{x}\))
several constants and/or Boolean variables and/or their
complements used in combination with one or more binary
operators (x + y)
If A and B are Boolean expressions, then, \(\overline{A}\), \(\overline{B}\), A+B and
A.B are also Boolean expressions.
Many of postulates are given as pairs. They differ only by the simultaneous interchange of operators "+" and "." and elements "0" and "1".

**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.

For each Boolean property the dual property is also valid without needing additional proof.
Useful Properties

Property 1: Special law of 0 and 1.

For all \( x \in BS \),
\[
x \cdot 0 = 0 \\
x + 1 = 1
\]

Proof: \( x \cdot 0 = (x \cdot 0) + 0 \) (postulate 2a)
\[
= (x \cdot 0) + (x \cdot \bar{x}) \tag{postulate 5b}
\]
\[
= x \cdot (0 + \bar{x}) \tag{postulate 4b}
\]
\[
= x \cdot \bar{x} \tag{postulate 2a}
\]
\[
= 0 \tag{postulate 5b}
\]

Part b can be proved by applying the law of duality.
Useful Properties (Contd…)

Property 2:
The element 0 is unique.
The element 1 is unique.
Proof for Part b by contradiction:
Assume that there are two 1s denoted 11 and 12.
\[ x \cdot 11 = x \] and \[ y \cdot 12 = y \] (Postulate 2b)
\[ x \cdot 11 = x \] and \[ 12 \cdot y = y \] (Postulate 3b)
Letting \( x = 12 \) and \( y = 11 \)
\[ 12 \cdot 11 = 12 \] and \[ 12 \cdot 11 = 11 \]
\[ 11 = 12 \] (transitivity property)
which becomes a contradiction of initial assumption
Property ‘a’ can be established by applying the principle of duality.
Useful Properties (Contd...)

Property 3: 
The complement of 0 is \( \overline{0} = 1 \).
The complement of 1 is \( \overline{1} = 0 \).
Proof: \( x + 0 = x \) (postulate 2a)
\[
\begin{align*}
\overline{0} + 0 &= \overline{0} \\
\overline{0} + 0 &= 1 \quad \text{(postulate 5a)}
\end{align*}
\]
\[
\overline{0} = 1
\]
Part b is valid by the application of principle of duality.
Useful Properties (Contd...)

**Property 4**: Idempotency law.

For all $x \in BS$,

$x + x = x$

$x \cdot x = x$

**Proof**: $x + x = (x + x) \cdot 1$ \hspace{1cm} (postulate 2b)

$= (x + x) \cdot (x + \bar{x})$ \hspace{1cm} (postulate 5a)

$= x + (x \cdot x)$ \hspace{1cm} (postulate 4a)

$= x + 0$ \hspace{1cm} (postulate 5b)

$= x$ \hspace{1cm} (postulate 2a)

$x \cdot x = x$ \hspace{1cm} (by duality)
Useful Properties (Contd…)

**Property 5:** Adjacency law.

For all \( x \) and \( y \in BS \),

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

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

**Proof:**

\[
\begin{align*}
x \cdot y + x \cdot \bar{y} &= x \cdot (y + \bar{y}) \quad \text{(postulate 4b)} \\
&= x \cdot 1 \quad \text{(postulate 5a)} \\
&= x \quad \text{(postulate 2b)} \\
\end{align*}
\]

\[
\begin{align*}
(x + y) \cdot (x + \bar{y}) &= x \quad \text{(by duality)}
\end{align*}
\]

Very useful in simplifying logical expressions
Useful Properties (Contd…)

Property 6: First law of absorption.
For all \( x \) and \( y \in BS \),

a. \( x + (x \cdot y) = x \)

b. \( x \cdot (x + y) = x \)

Proof: \( x \cdot (x + y) = (x + 0) \cdot (x + y) \) (postulate 2a)

\[
= x + (0 \cdot y) \quad \text{(postulate 4a)}
\]

\[
= x + 0 \quad \text{(property 2.1a)}
\]

\[
= x \quad \text{(postulate 2a)}
\]

\( x + (x \cdot y) = x \) (by duality)
Useful Properties (Contd…)

Property 7: Second law of absorption.
For all \( x \) and \( y \in BS \),

\[
\begin{align*}
\text{a.} & & x + (\bar{x} \cdot y) &= x + y \\
\text{b.} & & x \cdot (\bar{x} + y) &= x \cdot y
\end{align*}
\]

Proof: \( x + (\bar{x} \cdot y) = (x + x) \cdot (x + y) \) (postulate 4a)
\[
= 1 \cdot (x + y) \quad \text{(postulate 5a)}
\]
\[
= x + y \quad \text{(postulate 2b)}
\]
\[
x \cdot (\bar{x} + y) = x \cdot y \quad \text{(by duality)}
\]
Useful Properties (Contd...) 

Property 8: Consensus law.
For all \( x, y \) and \( z \in BS \),

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

Proof: \( x \cdot y + x \cdot z + y \cdot z \)

= \( x \cdot y + x \cdot z + l \cdot y \cdot z \)  \( \operatorname{(postulate} 2b) \)

= \( x \cdot y + x \cdot z + (x + x) \cdot y \cdot z \)  \( \operatorname{postulate} 5a \)
Useful Properties (Contd...)

Property 8 (contd...)

\[ x \cdot y + \bar{x} \cdot z + x \cdot y \cdot z + \bar{x} \cdot y \cdot z \]  (postulate 4b)

\[ = x \cdot y + x \cdot y \cdot z + \bar{x} \cdot z + \bar{x} \cdot y \cdot z \]  (postulate 3a)

\[ = x \cdot y \cdot (1 + z) + \bar{x} \cdot z \cdot (1 + y) \]  (postulate 4b)

\[ = x \cdot y \cdot 1 + \bar{x} \cdot z \cdot 1 \]  (property 2.1b)

\[ = x \cdot y + \bar{x} \cdot z \]  (postulate 2b)

\[(x + y) \cdot (\bar{x} + z) \cdot (y + z) = (x + y) \cdot (\bar{x} + z) \]  (by duality)
Useful Properties (Contd...) 

Property 9: Law of identity
For all $x$ and $y \in BS$

If $x + y = y$ and $x \cdot y = y$, then $x = y$

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

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

$$x \cdot (x + y) = x \quad (\text{property 6})$$

Therefore, by transitivity $x = y$
Useful Properties (Contd…)

Property 10: The law of involution. For all \( x \in BS \), \( x = x \)

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

\[
(x + x) = x \quad \text{and} \quad x \cdot x = x
\]

\[
x = x + 0 \quad \text{(postulate 2a)}
\]

\[
x = x + (x \cdot x) \quad \text{(postulate 5b)}
\]

\[
= (x + x) \cdot (x + x) \quad \text{(postulate 4a)}
\]

\[
= (x + x) \cdot 1 \quad \text{(postulate 5a)}
\]
Useful Properties (Contd...) 

Property 10 (contd.)

Thus \( x = x + x \)

Also \( x = x \cdot 1 \) (postulate 2b)

\( = x(x+x) \) (postulate 5a)

\( = x \cdot x + x \cdot x \) (postulate 4b)

\( = x \cdot x + 0 \) (postulate 5b)

\( = x \cdot x \) (postulate 2a)

Therefore by the law of identity, we have \( x = x \)
Property 11: DeMorgan's Law.
For all \( x, y \in BS, \)

\[ a. \quad x' + y' = (xy)' \]

\[ b. \quad x y' = (x+y)' \]

**Proof:**

\[ (x+y)(xy) = (xx + x) + (yxy) \]  \hspace{1cm} \text{(postulate 4b)}

\[ = 0 + 0 \]

\[ = 0 \]  \hspace{1cm} \text{(postulate 2a)}
Useful Properties (Contd...) 

Property 11 (contd.)

\[(x + y) + (\overline{x.y}) = (x + \overline{x.y}) + y\]  
\[= x + \overline{y} + y\]  
\[= x + 1\]  
\[= 1\]  
(postulate 3a)  
(property 2.7a)  
(postulate 5a)  
(property 2.16)  

Therefore, \((\overline{x} \cdot \overline{y})\) is the complement of \((x + y)\).

\[\overline{x.y} = \overline{x} + \overline{y}\]  
(by duality)
DeMorgan's Law

Bridges the AND and OR operations
Establishes a method for converting one form of a boolean function into another
Allows the formation of complements of expressions with more than one variable
Can be extended to expressions of any number of variables through substitution
Example

\[
x + y + z = x \cdot y \cdot z
\]

Let \( y + z = w \), then \( x + y + z = x + w \).

Since \( x + w = x \cdot w \) (by DeMorgan's law)

Therefore, \( x + w = x + y + z \) (by substitution)

\[
= x \cdot y + z \quad \text{(by DeMorgan's law)}
\]
Boolean Operators

BS = \{0, 1\}

Resulting Boolean algebra is more suited to working with switching circuits.

Variables associated with electronic switching circuits take only one of the two possible values.

The operations "+" and "." also need to be given appropriate meaning.
Binary Variables

Definition: A binary variable is one that can assume one of the two values 0 and 1.

These two values are meant to express two exactly opposite states.

If $A = 0$, then $A = 1$.

If $A = 1$, then $A = 0$.
Binary Variables (Contd…)

Examples:

if switch A is not open then it is closed
if switch A is not closed then it is open

Statement like

"0 is less than 1" or " 1 is greater than 0“ are invalid in Boolean algebra
NOT Operator

• The Boolean operator NOT, also known as complement operator.

• NOT operator is represented by “-” (overbar) on the variable, or " / " (a superscript slash) after the variable.

• Definition: Not operator is defined by

<table>
<thead>
<tr>
<th>A</th>
<th>A'</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
NOT Operator (Contd…)

- " / " symbol is preferred for convenience in typing and writing programs.

- Circuit representation:
OR Operator

• Definition: The Boolean operator "+" known as OR operator is defined by

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A+B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

• The circuit symbol for logical OR operation.
And Operator

• Definition: The Boolean operator "." known as AND operator is defined by

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A.B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

• Circuit symbol for the logical AND operation
Relationship of Operators to Electrical Switching Circuits

### AND Operator

- **A**
- **\( \bar{A} \)**

### OR Operator

- **A**
- **B**

<table>
<thead>
<tr>
<th>( \bar{A} )</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>Open</td>
<td>closed</td>
</tr>
<tr>
<td>closed</td>
<td>open</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A+B</th>
<th>A.B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Open</td>
<td>Open</td>
<td>Open</td>
<td>Open</td>
</tr>
<tr>
<td>Open</td>
<td>Closed</td>
<td>Closed</td>
<td>Open</td>
</tr>
<tr>
<td>Closed</td>
<td>Open</td>
<td>Closed</td>
<td>Open</td>
</tr>
<tr>
<td>Closed</td>
<td>Closed</td>
<td>Closed</td>
<td>Closed</td>
</tr>
</tbody>
</table>
Additional Boolean Operators

NAND,
NOR,
Exclusive-OR (Ex-OR)
Exclusive-NOR (Ex-NOR)

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

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>(A.B)'</th>
<th>(A+B)'</th>
<th>A ⊕ B</th>
<th>A ⊻ B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Additional Operations (Contd…)

NAND operation is just the complement of AND operation.
NOR operation is the complement of OR operation.
Exclusive-NOR is the complement of Exclusive-OR operation.

Circuit Symbols:

\[ A \quad B \quad (A \land B)' \quad A \oplus B \]

\[ A \quad B \quad (A \lor B)' \quad A \oplus B \]
Functionally Complete Sets of Operations

OR, AND and NOT
OR and NOT
AND and NOT
NAND
NOR
Completeness of AND, OR and NOT
Completeness of OR and NOT Operations
Completeness Of AND And NOT

A \lor \overline{B} \quad \overline{A} \lor B \quad \overline{A} \lor \overline{B} \quad \overline{A} \lor \overline{B} \lor \overline{B} \lor \overline{B} \lor \overline{B}
Completeness of NAND

A

B

OR

A

B

NOT

A

B

NOR

A

B

EX-OR

A

B

AND

A

B

EX-NOR
Completeness of NOR

A
A
B
B
NAND
A
A
B
B
NOT
A
A
B
B
AND
A
A
B
B
OR
A
A
B
B
EX-OR
A
A
B
B
EX-NOR
**Boolean Algebra and Boolean Operators**

**Introduction**

In electrical and electronic circuits one encounters a large variety of applications in which one is interested in signals that assume one of the two possible values. Examples of such signals are presence or absence of voltage, current being less than or greater than some stated level, a mechanical switch in either on or off condition etc. Some of the signals could be in the form of commands like Move, Shift, Stop, Start and Hold. In all these cases the intended action takes place or does not take place, indicating two possible values or states of the signals. The signals that have only two possible states may be called switching signals. The choice of the term 'switching' obviously is related to the mechanical devices that every one is familiar with, which have two well-defined states. When one encounters a large number of switching signals in a given electronic or electrical system, there arises a need for some formal methods of handling such signals.

Consider a simple example of an electrical circuit that controls the lighting of a bulb, shown in the figure 1. There are four switches that control the operation of the bulb. The relationship between the states of the switches and the bulb can be stated as 'the bulb is switched on if the switches S1 and S2 are closed, and S3 or S4 is also closed, otherwise the bulb will not be switched on'.

![FIG.1: A control circuit for an electric bulb](image)

The telephone exchanges of earlier era are full of relays that can operate either in two possible states or finite number of states.

Several of the semiconductor devices have certain advantages in operating in fully on state or fully off state. One distinct advantage is that in these two states the devices are likely to dissipate less power. The major advantage is that these two states can be maintained in spite of significant variations in the parameters of the associated circuits. In other words the reliability of the circuit can significantly be enhanced.
In this course on the Design of Digital Systems we are mainly concerned with two valued switching circuits. The mathematics of two valued switching circuits deals more with the logical relationships between the input signals and output signals, rather than with the manner in which these signals are actually realised in practice. While electronic circuits based on switching signals are relatively recent, the mathematical foundation for this was laid in 19th century. 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 this course, switching, logical or digital variables are considered to be synonymous.

The usage of Boole's algebra, henceforth referred to as 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, through 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”. What Shannon has been able to establish was that 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 and aerospace equipment. The switching circuitry used in these electronic systems is built from integrated circuits based on bipolar transistors, metal oxide semiconductor (MOS) transistors, or gallium arsenide devices. It is not necessary to design switching circuits at functional level to understand how the actual switching circuits based on semiconductor devices work. However, it becomes necessary to fully understand the hardware aspects of the circuits used to build a usable digital system. These hardware aspects include voltage levels, current capacities, switching time delays, noise margins etc. The main differences between the switching
devices of earlier era and the present day systems are in terms of great reductions in switching time delays, physical sizes and cost. The whole evolution of digital system is really powered by the tremendous strides made in the semiconductor technology.

This chapter presents Boolean algebra at the axiomatic level without referring to any issues related to their physical realisation. The concepts presented here are very fundamental in nature and the reader is urged to completely internalise them. It is not an exaggeration to state that the entire edifice of digital systems is built up on these basic concepts.

**Huntington Postulates and Boolean Algebra**

When a new mathematics is introduced, it is generally done through a set of self-evident statements known as postulates, axioms or maxims. These are stated without any proof. Several new propositions can be proven using the basic postulates to explore the structures and implications of this new mathematics. Such propositions are called theorems. A theorem, in this sense, is like a postulate, giving a relationship between the variables that belong to this new mathematics. Boolean algebra is a specific instance of Algebra of Propositional Logic.

Huntington postulates, presented by E.V.Huntington in his paper “Sets of Independent Postulates for the Algebra of Logic” in 1904, define a multi-valued Boolean algebra on a set of finite number of elements. In Boolean algebra as applied to the switching circuits and commonly understood in electronics, all variables and relations are two-valued. Because this algebra was first developed to deal with logical propositions that could be true or false, those are the two values a variable or relation can have in Boolean algebra. They 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, and
- $x = 0$ means $x$ is false

However, in our application of Boolean algebra to digital circuits we will find that asserted and not asserted are better names for the two values a logic-variable can have.
The reader is expected to be familiar with the concept of a set, and the meaning of equivalence relation and the principle of substitution to understand the material in this section.

**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), and two binary operators "." and "+", and for every element $x$ and $y \in BS$ the operations $\overline{x}$ (not $x$), $x.y$ and $x+y$ are uniquely defined.

The unary operator ‘not’ is defined by the relation

$$\overline{1} = 0; \quad \overline{0} = 1$$

The *not* operator is also called the complement, and consequently $\overline{x}$ 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 symbolized by a plus sign. The ‘or’ operator is defined by the relations

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

The following six (Huntington's) postulates apply to the Boolean operations.

**P1. The operations are closed.**

For all $x$ and $y \in BS$,

- a. $x + y \in BS$
- b. $x . y \in BS$

**P2. For each operation there exists an identity element.**

- a. There exists an element $0 \in BS$ such that for all $x \in BS$, $x + 0 = x$
- b. There exists an element $1 \in BS$ such that for all $x \in BS$, $x . 1 = x$

**P3. The operations are commutative.**
For all $x$ and $y \in \text{BS}$,

a. $x + y = y + x$

b. $x \cdot y = y \cdot x$

**P4. The operations are distributive.**

For all $x$, $y$ and $z \in \text{BS}$,

a. $x + (y \cdot z) = (x + y) \cdot (x + z)$

b. $x \cdot (y + z) = (x \cdot y) + (x \cdot z)$

**P5. For every element $x \in \text{BS}$ there exists an element $\overline{x} \in \text{BS}$ (called the complement of $x$) such that $x + \overline{x} = 1$ and $x \cdot \overline{x} = 0$**

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

**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 $\overline{x}$ are Boolean expressions. If $A$ and $B$ are Boolean expressions, then $\overline{A}$, $\overline{B}$, $A+B$ and $A \cdot B$ are also Boolean expressions.

An examination of the basic six postulates reveals that many of them 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. This property of duality can be utilized effectively to establish many useful properties of Boolean algebra. The duality principle states that 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:

\[
(x + y).(x\bar{y}) = (x\bar{x}.y) + (y\bar{x}.\bar{y}) \quad \text{(Postulate 4b)}
\]

\[
= 0 + 0 \quad \text{(Postulate 2a)}
\]

\[
= 0 \quad \text{(Postulate 2a)}
\]

Proof:

\[
x.0 = (x.0) + 0 \quad \text{(Postulate 2a)}
\]

\[
= (x.0) + (x\bar{x}) \quad \text{(Postulate 5b)}
\]

\[
= x(0 + \bar{x}) \quad \text{(Postulate 4b)}
\]

\[
= x\bar{x} \quad \text{(Postulate 2a)}
\]

\[
= 0 \quad \text{(Postulate 5b)}
\]

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 \( l_1 \) and \( l_2 \). Postulate 2b states that

\[
x.l_1 = x \text{ and } y.l_2 = y
\]

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

\[
x.l_1 = x \text{ and } y.l_2 = y
\]

Letting \( x = l_2 \) and \( y = l_1 \), we obtain

\[
l_2.l_1 = l_2 \text{ and } l_2.l_1 = l_1
\]

Using the transitivity property of any equivalence relationship we obtain \( l_1 = l_2 \), 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:

\[
x + 0 = x \quad \text{(Postulate 2a)}
\]
\[
0 + \bar{0} = \bar{0}
\]
\[
0 + \bar{0} = 1 \quad \text{(Postulate 5a)}
\]
\[
\bar{0} = 1
\]

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: \[
x + x = (x + x).1 \quad \text{(Postulate 2b)}
\]
\[
= (x + x).(x + \bar{x}) \quad \text{(Postulate 5a)}
\]
\[
= x + (x\bar{x}) \quad \text{(Postulate 4a)}
\]
\[
= x + 0 \quad \text{(Postulate 5b)}
\]
\[
= x \quad \text{(Postulate 2a)}
\]
\[
x.x = x \quad \text{(by duality)}
\]
Property 5: Adjacency law
For all \( x \) and \( y \in BS \),

a. \( x \cdot y + x \cdot \overline{y} = x \)
b. \( (x+y) \cdot (x + \overline{y}) = x \)

Proof:
\[
\begin{align*}
x \cdot y + x \cdot \overline{y} &= \overline{(x \cdot y)} + x \cdot \overline{y} \\
&= \overline{(x \cdot y)} + x \cdot \overline{y} \\
&= \overline{(x \cdot y)} + x \cdot \overline{y} \\
&= x \\
(x+y) \cdot (x+\overline{y}) &= x
\end{align*}
\]

(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 \( y \in BS \),

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

Proof:
\[
\begin{align*}
x \cdot (x + y) &= (x + 0) \cdot (x + y) \\
&= x \cdot \overline{(x + y)} \\
&= x \cdot \overline{y} \\
&= x + 0 \\
x + (x \cdot y) &= x
\end{align*}
\]

(by duality)
Property 7: Second law of absorption

For all \( x \) and \( y \in BS \),

a. \( x + (\overline{x} \cdot y) = x + y \)
b. \( x \cdot (\overline{x} + y) = x \cdot y \)

Proof: \( x + (\overline{x} \cdot y) = (x + \overline{x}) \cdot (x + y) \quad \text{(Postulate 4a)} \)
\approx 1 \cdot (x + y) \quad \text{(Postulate 5a)}
\approx x + y \quad \text{(Postulate 2b)}
\approx x \cdot (\overline{x} + y) = x \cdot y \quad \text{(by duality)}

Property 8: Consensus law

For all \( x \) and \( y \in BS \),

a. \( x \cdot y + (\overline{x} \cdot z) + y \cdot z = x \cdot y + \overline{x} \cdot z \)
b. \((x + y) \cdot (\overline{x} + z) \cdot (y + z) = (x + y) \cdot (\overline{x} + z) \)

Proof: \( x \cdot y + (\overline{x} \cdot z) + y \cdot z = x \cdot y + \overline{x} \cdot z + y \cdot z \quad \text{(Postulate 2b)} \)
\approx x \cdot y + \overline{x} \cdot z \cdot (x + \overline{x}) \cdot y \cdot z \quad \text{(Postulate 5a)}
\approx x \cdot y + x \cdot y \cdot z + x \cdot y \cdot z \quad \text{(Postulate 4b)}
\approx x \cdot y + x \cdot y \cdot z + x \cdot z \cdot x \cdot y \cdot z \quad \text{(Postulate 5a)}
\approx x \cdot y \cdot (1 + z) + \overline{x} \cdot z \cdot (1 + y) \quad \text{(Postulate 4b)}
\approx x \cdot y \cdot 1 + \overline{x} \cdot z \cdot 1 \quad \text{(Postulate 2.1b)}
\approx x \cdot y + \overline{x} \cdot z \quad \text{(Postulate 2b)}
\approx (x + y) \cdot (\overline{x} + z) = (x + y) \cdot (\overline{x} + z) \quad \text{(by duality)}

**Property 9**: Law of identity

For all \( x \text{ and } y \in BS \),

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

b. \(x \cdot y = y \Rightarrow x = y\)

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

\[ x(x + y) = y \]

However by the first law of absorption

\[ x(x + y) = x \] \hspace{1cm} \text{(Property 6)}

Therefore, by transitivity \( x = y \)

**Property 10**: The law of involution

For all \( x \in BS, x \equiv x \)

Proof: We need to show that the law of identity (Property 2.9)

holds, that is,

\[ (x + x) = x \text{ and } x \cdot x = x \]

\[ x = x + 0 \] \hspace{1cm} \text{(Postulate 2a)}

\[ = x + (x \cdot \overline{x}) \] \hspace{1cm} \text{(Postulate 5b)}

\[ = (x + x) \cdot (x + \overline{x}) \] \hspace{1cm} \text{(Postulate 4a)}

\[ = (x + x) \cdot 1 \] \hspace{1cm} \text{(Postulate 5a)}

Thus \( x = x + x \)

Also \( x = x \cdot 1 \) \hspace{1cm} \text{(Postulate 2b)}

\[ = x \cdot (x + \overline{x}) \] \hspace{1cm} \text{(Postulate 5a)}

\[ = x \cdot x + x \cdot \overline{x} \] \hspace{1cm} \text{(Postulate 4b)}

\[ = x \cdot x + 0 \] \hspace{1cm} \text{(Postulate 5b)}

\[ = x \cdot x \] \hspace{1cm} \text{(Postulate 2a)}

Therefore by the law of identity, we have \( x \equiv x \)
**Property 11: DeMorgan's Law**

For all \( x \) and \( y \in \text{BS} \),

\begin{align*}
\text{a. } x + y &= \overline{x} \cdot \overline{y} \\
\text{b. } x \cdot y &= \overline{x} + \overline{y}
\end{align*}

\[
(x + y)(\overline{x} \cdot \overline{y}) = (x \overline{x} \cdot \overline{y}) + (y \overline{x} \cdot y) \\
= 0 + 0 \\
= 0
\]

(Postulate 4b)

\[
(x + y) + (\overline{x} \cdot \overline{y}) = (x + \overline{x} \cdot \overline{y}) + y \\
= x + \overline{y} + y \\
= x + 1 \\
= 1
\]

(Postulate 2.7a)

Therefore, \( \overline{x} \cdot \overline{y} \) is the complement of \( x + y \).

\[
x \cdot y = \overline{x} + \overline{y}
\]

(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:

\[
x + y + z = \overline{x} \cdot \overline{y} \cdot \overline{z}
\]

Let \( y + z = w \), then \( x + y + z = x + w \).

\[
\begin{align*}
\overline{x + w} &= \overline{x} \cdot \overline{w} \\
\overline{x + w} &= x + y + z \\
&= \overline{x} \cdot \overline{y} + z \\
&= \overline{x} \cdot \overline{y} \cdot \overline{z}
\end{align*}
\]

(By DeMorgan's law)

(By substitution)

(By DeMorgan's law)

(By DeMorgan's law)

In this section we familiarised ourselves with Huntington's postulates and derived a variety of properties from the basic postulates. Several more properties can be derived, and some of those are given as problems at the end of the Module to be worked out by the reader. 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 in this Module and other Modules associated with the subject of Design of Digital systems. The properties of Boolean algebra when the set BS has two elements, namely 0 and 1, will be explored in the next section.

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

<table>
<thead>
<tr>
<th>Complementation</th>
<th>(x \overline{x} = 0)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(x + \overline{x} = 1)</td>
</tr>
<tr>
<td>0-1 law</td>
<td>(x.0 = 0)</td>
</tr>
<tr>
<td></td>
<td>(x + 1 = 1)</td>
</tr>
<tr>
<td></td>
<td>(x + 0 = x)</td>
</tr>
<tr>
<td></td>
<td>(x.1 = x)</td>
</tr>
<tr>
<td>Idempotency</td>
<td>(x.x = x)</td>
</tr>
<tr>
<td></td>
<td>(x + x = x)</td>
</tr>
<tr>
<td>Involution</td>
<td>(x = x)</td>
</tr>
<tr>
<td>Commutative law</td>
<td>(x.y = y.x)</td>
</tr>
<tr>
<td></td>
<td>(x + y = y + x)</td>
</tr>
<tr>
<td>Associative law</td>
<td>((x.y).z = x.(y.z))</td>
</tr>
<tr>
<td></td>
<td>((x + y) + z = x + (y + z))</td>
</tr>
<tr>
<td>Distributive law</td>
<td>(x + (y.z) = (x + y).(x + z))</td>
</tr>
<tr>
<td></td>
<td>(x.(y + z) = x.y + x.z)</td>
</tr>
<tr>
<td>Adjacency law</td>
<td>(x.y + x.\overline{y} = x)</td>
</tr>
<tr>
<td></td>
<td>((x + y)(x + \overline{y}) = x)</td>
</tr>
<tr>
<td>Absorption law</td>
<td>(x + x.y = x)</td>
</tr>
<tr>
<td></td>
<td>(x.(x + y) = x)</td>
</tr>
<tr>
<td></td>
<td>(x + \overline{x}.y = x + y)</td>
</tr>
<tr>
<td></td>
<td>(x(\overline{x} + y) = x.y)</td>
</tr>
</tbody>
</table>
Consensus law
\[ x.y + \overline{x}.z + y.z = x.z + \overline{x}.z \]
\[ (x + y).(\overline{x} + z).(y + z) = (x + y).(\overline{x} + z) \]

DeMorgan's law
\[ x + y = \overline{\overline{x}.\overline{y}} \]
\[ x.y = \overline{x} + \overline{y} \]

**Boolean Operators**

In the preceding section no restriction was placed on the number of elements in the set BS. However, when the set BS is restricted to two elements, that is, BS = \{0, 1\}, then the resulting Boolean algebra is more suited to working with switching circuits. It may be noted that in all the switching circuits encountered in electronics, the variables take only one of the two possible values. The operations "+" and "." also need to be given appropriate meaning when working with two valued variables.

**Definition**: A binary variable is one that can assume one of the two values 0 and 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 "\( \overline{\cdot} \)" (overbar) on the variable, or "\( / \)" (a superscript slash) after the variable, is defined by the following table.

<table>
<thead>
<tr>
<th>( A )</th>
<th>( A' )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Though it is more popular to use the symbol "\( \overline{\cdot} \)" (overbar) in most of the text-books, we will adopt the "\( / \)" to represent the complement of a variable, in view of some of the
software programmes that we would later use to design digital circuits. Besides it is also convenient for typing.

The circuit representation of the NOT operator is shown in the following:

![Circuit Diagram]

**Definition 5**: The Boolean operator "+" known as OR operator is defined by the table given in the following.

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A+B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

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

![Circuit Diagram]

**Definition 6**: The Boolean operator "." known as AND operator is defined by the table given below.

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A.B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

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

![Circuit Diagram]

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

![Circuit Diagram]

Besides these three basic logic operations, several other logic operations can also be defined. These include NAND, NOR, Exclusive-OR (Ex-OR for short) and Exclusive-NOR (Ex-NOR). These are defined in terms of different combinations of values the variables assume, as indicated in the following table:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A + B</th>
<th>A.B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Open</td>
<td>Open</td>
<td>Open</td>
<td>Open</td>
</tr>
<tr>
<td>Open</td>
<td>Closed</td>
<td>Closed</td>
<td>Open</td>
</tr>
<tr>
<td>Closed</td>
<td>Open</td>
<td>Closed</td>
<td>Open</td>
</tr>
<tr>
<td>Closed</td>
<td>Closed</td>
<td>Closed</td>
<td>Closed</td>
</tr>
</tbody>
</table>

As it can be noticed from the table, NAND operation is just the complement of AND operation, and 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 2.
FIG. 2: Circuit symbols for NAND, NOR, EX-OR and EX-NOR

Of all the Boolean operators defined so far AND, OR and NOT are considered to be the basic operators. However, it is possible to have several combinations of these operations, as listed in the following as functionally complete sets. All Boolean expressions can be expressed in terms of these functionally complete sets of operations.

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

The completeness of these combinations is shown in the figures 3, 4, 5 and 6 and 7.

FIG. 3: All Boolean functions through AND, OR and NOT operation
FIG. 4: All Boolean functions through OR and NOT operations

FIG. 5: All Boolean functions through AND and NOT operations
FIG. 6: All Boolean functions through NAND function

FIG. 7: All Boolean functions through NOR function
LOGIC FUNCTIONS

Introduction

Boolean algebra and Boolean operators are defined over a set of finite number of elements. In digital systems we are interested in the Boolean algebra wherein the number of elements in the set is restricted to two (0 and 1). All the Boolean variables which can take only either of two values may be called as Logic variables or Switching variables. All the Boolean operators, when applied to a set of two elements can be called Logic operators. We will find that it is possible to model a wide variety of situations and problems using logic variables and logic operators. This is done through defining a “logic function” of logic variables.

There are several ways in which logic functions are expressed. These include algebraic, truth-table, logic circuit, hardware description language, and map forms. All these forms of expressing logic functions are used in working with digital circuits. However, we will be mainly concerned with the algebraic, truth-table and logic circuit representation of logic functions in this Learning Unit.

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 with minterm or maxterm form.
7. Explain the nature and role of “don’t care” terms
Logic Functions in Algebraic Form

A logic function of n variables, A1, A2, ... An, defined on the set BS = {0,1}, associates a value 0 or 1 to every one of the possible 2^n combinations of the n variables.

The logic functions have the following properties:

1. If F1(A1, A2, ... An) is a logic function, then (F1(A1, A2, ... An))' 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 logic function.

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. A "literal" is complemented or uncomplemented version of a variable. A "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. A "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 the 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.

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 the 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 constitute 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,

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

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 intimately with digital electronics. Boolean functions were originally used to establish truth or falsehood of a statement. 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". As it is evident, these variables are suggestive of actions. Therefore, it is more 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 Modules associated with 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 in an exponential manner. For example, a five variable expression would require 32 entries and 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 artifact 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 smaller number of terms. Consider the Boolean function

\[ F = A.B.C.D'.E' + A.B'.C.D'.E + A'.B'.C.D.E + A.B'.C'.D.E \]

The truth-table representation of this function is given as,

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Truth-table is a very effective tool in working digital circuits, especially when the number of variables in a function is small, less than or equal to five. We will have many occasions to use this tool in all the Modules associated with Digital Systems.

**Conversion of English Sentences to Logic Functions**

Digital design problems are often expressed through one or more sentences. 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 and directly represent
actions. Each action related phrase in the sentence will be marked separately before associating a logic variable with it. The following sentence 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.

\[
F = 1 \text{ if } \text{“Anil freaks out with his friends”}; \text{ otherwise } F = 0
\]

\[
A = 1 \text{ if } \text{“it is Saturday”}; \text{ otherwise } A = 0
\]

\[
B = 1 \text{ if } \text{“he completed his assignments”}; \text{ otherwise } B = 0
\]

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

\[
F = A \cdot B
\]

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

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

\[ F = AB + C'D \]

**Minterms and Maxterms**

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 Boolean 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 1 which lists are the minterms and maxterms of three variables. The minterms are designated as \( m_0, m_1, \ldots m_7 \), and maxterms are designated as \( M_0, M_1, \ldots M_7 \).

**Table 1: Minterms and Maxterms of three variables**

<table>
<thead>
<tr>
<th>Term No.</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>Minterms</th>
<th>Maxterms</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>( A'B'C' = m_0 )</td>
<td>( A + B + C = M_0 )</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>( A'B'C = m_1 )</td>
<td>( A + B + C' = M_1 )</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>( A'BC' = m_2 )</td>
<td>( A + B' + C = M_2 )</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>( A'BC = m_3 )</td>
<td>( A + B' + C' = M_3 )</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>( ABC' = m_4 )</td>
<td>( A + B + C = M_4 )</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>( ABC = m_5 )</td>
<td>( A + B + C' = M_5 )</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>( ABC' = m_6 )</td>
<td>( A' + B + C = M_6 )</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>( ABC = m_7 )</td>
<td>( A' + B' + C' = M_7 )</td>
</tr>
</tbody>
</table>

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

\[ F = m_0 + m_3 + m_5 + m_6 \]  \( (1) \)
This is equivalent to
\[ F = A'B'C' + A'BC + A'BC' + ABC' \]  
(2)

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:

\[ F_1 = A.B + A.B'.C + A'.B.C \]  
(SOP form)  
(2)

\[ F_2 = (A+B+C') . (A+B'+C') . (A'+B'+C) \]  
(POS form)  
(3)

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 Eqn.(1) is not in canonical form. However it can be converted into canonical form by expanding the term A.B as given below:

\[ A.B = A.B.1 \]  
(postulate 2b)

\[ = A.B.(C + C') \]  
(postulate 5a)

\[ = A.B.C + A.B.C' \]  
(postulate 4b)

The canonical version of F1 is,

\[ F_1 = A.B.C + A.B.C' + A.B'.C + A'.B.C \]  
(4)

The Boolean function F2 given in the equation (2) 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 products 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. The NOT (') operation has the highest priority, followed by AND (.) and then by OR (+). In the expression for F1, performing NOT operation on B and A has the highest priority, then creation of three AND terms,
followed by 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.

\[ F_1 = A.(B+C') + A'.(C+D) \]  

By applying the distributive law, these expressions can be brought into the SOP form. More detailed manipulation of algebraic form of logic functions will be explored in the next Learning Unit.

**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

\[ F_1 = A.B + A.B' \]  

Its circuit form is shown in the figure 8.

![FIG.8: Circuit representation of the function F1 (eqn. 6)](image)

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

\[ F_2 = (A+B+C) . (A+B'+C') \]  

Its circuit form is given in the figure 9.

![FIG.9: Circuit representation of the function F2 (eqn.6)](image)
As there are different combinations of functionally complete sets of logic operations the circuit representation of a Boolean function can also be done in a number of ways. Consider the function $F_1$ given by the equation 5. Its NAND representation is shown in the figure 10.

\[
\begin{array}{c}
A \\
B
\end{array}
\quad
\begin{array}{c}
\quad \quad \\

\quad \\
\end{array}
\quad
\begin{array}{c}
\quad \quad \\
\quad \\
F_1
\end{array}
\]

**FIG.10: NAND representation of the function $F_1$ (eqn.6)**

Similarly, NOR representation of the same function $F_1$ is shown in the figure 11.

\[
\begin{array}{c}
A \\
B
\end{array}
\quad
\begin{array}{c}
\quad \quad \\

\quad \\
\end{array}
\quad
\begin{array}{c}
\quad \quad \\
\quad \\
F_1
\end{array}
\]

**FIG.11: NOR representation of the function $F_1$ (eqn.6)**
KARNAUGH-MAP METHOD OF SIMPLIFICATION

Introduction

While the algebraic minimisation process allows us to simplify or minimise any logical expression, it is necessary to be able to identify the patterns among the terms. These patterns should conform to one of the four laws of Boolean algebra listed in the last chapter. However, it is not always very convenient to identify different patterns that may be present in the expressions. If a logic function can be represented graphically that would enable us to identify the inherent patterns, it would be much more convenient to perform the simplification. One such graphic or pictorial representation of a Boolean function in the form of a map, known as 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 a pictorial form of the truth-table. The inherent structure of the map enables us to minimise a given logic expression more systematically. This advantage of the K-map comes from the fact that it implicitly uses the ability of human perception to identify patterns and relationships when the information is presented graphically. This chapter is devoted to the Karnaugh map and its method of simplification of Boolean functions.

Karnaugh Map

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

\[ F = A'B + AB' \]  

(1)

Any two variable function has \(2^2 = 4\) minterms. The truth table of this function is given in Table 1.

Table 1: Truth-table for the function F of Eqn. (1)

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
As it can be seen that while the values of the variables are arranged in the ascending order (in the numerical decimal sense), the positionally adjacent entries are not logically adjacent. For example $A'B (01)$ and $AB' (10)$ 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 1. There are two popular ways of representing the map, both of which are shown in the figure 1. The representation, the variable above the column or on the left side of the row, in which it is asserted, will be followed in this and the associated Modules.

![FIG.1: Karnaugh map of a two variable function](image)

There are four cells (squares) in this map. The cells are labelled as $0, 1, 2$ and $3$ representing the four minterms $m_0, m_1, m_2$ and $m_3$. The numbering of the cells is such that the logically adjacent minterms are positionally adjacent. For example, cell 1 is adjacent to cell 0 and cell 3, indicating the minterm $m_1 (01)$ is logically adjacent to the minterm $m_0 (00)$ and the minterm $m_3 (11)$. It may also be noticed that 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. The name of the function is written at the left hand top corner. If there is no ambiguity with regard to which function the particular map refers, one may omit this method of labelling.

Before proceeding further, it is necessary to properly define the concept of positional adjacency. Position adjacency means two adjacent cells sharing one side, which 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, 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 adjacency, which become relevant when the number
of variables is more than 3. These would be presented later in this Chapter. 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 $F$, given in the Eqn.(1), can now be incorporated into the K-map, by entering "1" in the cell represented by the minterm for which the function is asserted. A "0" is entered in all other cells. K-map for the function $F$ is shown in the figure 2.

![K-map of the function $F = A'B + AB'$](image)

**FIG. 2: K-map of the function $F = A'B + AB'$**

As it can easily be seen that the two cells in which "1" is entered are not positionally adjacent and hence are not logically adjacent. Consider another function of two variables.

$$F = A'B + AB$$

(2)

The K-map for this function, drawn as per the procedure given above is shown in the figure 3.

![K-map of the function $A'B + AB$](image)

**FIG. 3: K-map of the function $A'B + AB$**

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

**Three-Variable Karnaugh Map:** Figure 4 shows a three-variable (A, B and C) K-map corresponding to three-variable truth-table, which will have $2^3 = 8$ cells. These cells are labelled 0,1,2,...,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.
The equivalence of logic adjacency and position adjacency in the map can be verified. For example, cell 7 (111) is adjacent to the cells 3 (011), 5 (101) and 6 (110). Similarly, 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:

\[
F = \Sigma(1, 3, 5, 7) \tag{3}
\]

\[
= m1 + m3 + m5 + m7
\]

\[
= A'B'C + A'BC + AB'C + ABC
\]

\[
= A'C(B'+B) + AC(B'+B)
\]

\[
= A'C + AC
\]

\[
= (A'+A)C
\]

\[
= C
\]

The K-map of the function F is shown in the figure 5. 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 and 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 $1 \rightarrow 3 \rightarrow 7 \rightarrow 5 \rightarrow 1$ ("$\rightarrow$" indicating "adjacent to"). When a group
of cells, always $2^i$ ($i \leq n$) in number, are adjacent to one another in a sequential manner is called cyclic adjacency. In a three-variable map other groups of cells that are cyclically adjacent are 0, 1, 3 and 2; 2, 3, 7 and 6; 6, 7, 5 and 4; 4, 5, 1 and 0; 0, 2, 6, 4.

![K-map of the function F given by eqn.(3)](image)

So far we noticed two kinds of positional adjacencies: simple adjacency and cyclic adjacency. The latter has two cases, one is between two cells, and the other among a group of $2^i$ cells.

**Four-variable Karnaugh Map:** Figure 6 shows a four-variable (A, B, C and D) K-map corresponding to four-variable truth-table, which will have $2^4 = 16$ cells. These cells are labelled 0, 1, 2..., 15, which stand for combinations 0000, 0001, ...1111 respectively. Notice two columns each are associated with assertion of A and B, and two rows each are associated with the assertion of C and D.

![K-map for four variable functions](image)

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
Consider a function of four variables.

\[
F = \Sigma (2, 3, 8, 9, 11, 12)
\]

(4)

The K-map of this function is shown in the figure 7.

![K-map of the function given by the eqn. (4)](image)

**FIG.7: K-map of the function given by the eqn. (4)**

Five-variable Karnaugh Map: Figure 8 shows a Karnaugh map for five variables. It has \(2^5 = 32\) cells labelled as 0,1, 2 ...,31, corresponding to the 32 combinations from 00000 to 11111. As it may be noticed the map is divided vertically into two symmetrical parts. Variable A is not-asserted 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.

![K-map for five variable functions](image)

**FIG.8: K-map for five variable functions**

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 exists because of the division of the map into two symmetrical sections. Taking the assertion and non-assertion 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, 7-23, 15-31, 11-27, 2-18, 6-22, 14-30, and 10-26).

An example of five-variable map is shown in the figure 9.

![K-map of function F = A'B'C'D'E + A'B'CDE' + A'BC'D'E + ABCDE + A'B'C'D'E + ABC'D'E + ABCD'E'](image)

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

1. Karnaugh Map's main feature is to convert logic adjacency into positional adjacency.
2. Every Karnaugh map has $2^n$ cells corresponding to $2^n$ minterms.
3. Combinations are arranged in a special order so as to keep the equivalence of logic adjacency to positional adjacency.
4. 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 given in the Eqn. (5)

$$F = A'B + A'B'C' + ABC'D + ABC'D'$$  \hspace{1cm} (5)
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 shown in the figure 10.

![K-map of the function given by the eqn.(5)](image1)

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.

Boolean functions, sometimes, are also available in POS form. Let us assume that the function is available in the canonical POS form. An example of such a function is given in the Eqn. (6).

\[
F = \Pi (0, 4, 6, 7, 11, 12, 14, 15)
\]  

(6)

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 shown in the figure 11.

![K-map of the function given by the eqn.(6)](image2)

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 shown in the Eqn.(4.8).

\[ F = (A+B+C+D').(A+B+C'+D')(A'+B+C'+D).(A+B'+C+D). \]
\[ (A'+B'+C+D).(A'+B'+C+D').(A'+B'+C+D') \]

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 \((M_1)\) and 0011 \((M_3)\). The second term is in its canonical form and the maxterm associated with is 1010 \((M_{10})\). Similarly the maxterms associated with the third sum term are 010 \((M_4)\), 100 \((M_{12})\), 0101 \((M_5)\) and 1101 \((M_{13})\). The resultant K-map is shown in the figure 12.

![Figure 12: K-map of the function given by the eqn. (7)](image)

**Essential, Prime and Redundant 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 shown in the figure 13.

![Fig. 13: A K-map with different implicants identified](image)

Once a map representation is given, all the implicants can be determined by grouping all the 1-entered cells based on positional adjacency, including the case of single cell implicants. A single cell implicant is a 1-entered cell that is not positionally adjacent to any of the other cells in map. The figure 13 indicates four implicants numbered as 1, 2, 3 and 4. Among these four implicants all the groupings of 1-entered cells are accounted for, which also means that the four implicants describe the Boolean function completely. It may be noticed that the implicant 4 is a single cell implicant.

An implicant represents a product term, with the number of variables appearing in the term as inversely proportional to the number of 1-entered cells it represents. For example, implicant 1 in the figure 13 represents the product term $AC'$, implicant 2 represents $ABD$, implicant 3 represents $BCD$, and 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 in which the implicants can be identified. Some alternatives are given in the figure 14.
Five implicants are identified in the figure 14 (a) and three implicants are identified in the figure 14 (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. Towards establishing such a procedure we introduce three new terms known as "prime implicant", "essential implicant" and "redundant implicant".

**Definition 2**: A prime implicant is one that is not a subset of any one of the other implicants.

**Definition 3**: A prime implicant which includes a 1-entered cell that is not included in any other prime implicant is called an essential prime implicant.

**Definition 4**: A redundant implicant is one in which all the 1-entered cells are covered by other implicants.

For example, 1, 2, 3 and 4 in the figure 13, 2, 3, 4 and 5 in the figure 14 (a), and 1, 2 and 3 in the figure 14 (b) are prime implicants. Implicants 1, 3 and 4 in the figure 13, 2, 4 and 5 in the figure 14(a), and 1, 2 and 3 in the figure 14 (b) are essential prime implicants. Implicant 2 in the figure 13, and implicants 1 and 3 in the figure 14 (a) are redundant implicants. Figure 14 (b) does not have any redundant implicants. A redundant implicant represents a redundant term in an expression.

Now the method of K-map minimisation can be stated as "finding the smallest set of prime implicants that includes all the essential prime implicants that account for all the 1-entered cells of the K-map". If there is a choice, the simpler prime implicant should be chosen. The minimisation procedure is best understood through examples.
**Example 1**: Find the minimised expression for the function given by the K-map in the figure 15.

Fifteen implicants corresponding the K-map shown in the figure 15 are:

- $X_1 = C'D'$
- $X_2 = B'C'$
- $X_3 = BD'$
- $X_4 = ACD$
- $X_5 = AB'C'$
- $X_6 = BCD'$
- $X_7 = A'B'C'$
- $X_8 = BC'D'$
- $X_9 = B'C'D'$
- $X_{10} = A'C'D'$
- $X_{11} = AC'D'$
- $X_{12} = AB'D$
- $X_{13} = ABC$
- $X_{14} = A'BD'$
- $X_{15} = ABD'$
- $X_{16} = B'C'D$

![K-map](image)

FIG.15: K-map corresponding to the Example 1

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.

- $F_1 = X_1 + X_4 + X_6 + X_{16}$
  - $= X_4 + X_5 + X_6 + X_7 + X_8$  \( (9) \)
  - $= X_2 + X_3 + X_4$  \( (10) \)
  - $= X_{10} + X_{11} + X_8 + X_4 + X_6$  \( (11) \)

The k-maps with these four combinations are shown in the figure 4.16

Among the prime implicants listed in the figure there are three implicants $X_1$, $X_2$ and $X_3$ which group four 1-entered cells. Selecting the smallest number of implicants we obtain the simplified expression as:

- $F = X_2 + X_3 + X_4$
  - $= B'C' + BD' + ACD$  \( (12) \)

It may be noticed that $X_2$, $X_3$ and $X_4$ are essential prime implicants.
Example 2: Minimise the Boolean function represented by the K-map shown in the figure 17.

Three sets of prime implicants are shown in the figure 4.18:

(a) \( X_1 = B'D' \quad X_2 = A'B \quad X_3 = BD \quad X_4 = ACD \)
(b) \( X_4 = ACD \quad X_5 = AB'D' \quad X_6 = A'B'D' \quad X_7 = ABD \quad X_8 = A'BC \quad X_9 = A'BC' \)
(c) \( X_7 = ABD \quad X_{10} = B'C'D' \quad X_{11} = A'C'D' \quad X_{12} = A'BD \quad X_{13} = A'C'D' \quad X_{14} = AB'C \)

FIG.17: K-map for the example 2
FIG. 18: Three possibilities of identifying prime implicants for the K-map show in the figure 17

Some of the simplified expressions are shown in the following:

\[
F = X_1 + X_2 + X_3 + X_4 \quad (14)
\]
\[
= X_4 + X_6 + X_7 + X_8 + X_9 \quad (15)
\]
\[
= X_7 + X_{10} + X_{11} + X_{12} + X_{13} + X_{14} \quad (16)
\]

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 than 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 uncomplemented 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 19.

Four implicants are identified in the figure 19. B is asserted and A is not-asserted in all the cells of implicant 1, where as the variable C and D change their values from 0 to 1. Hence this implicant is represented by the sum term \(A + B'\). Similarly, the implicant 2 is represented by the sum term \((B' + D')\), implicant 3 by \((B + D)\), and the implicant 4 by \((A' + C' + D')\).

![FIG.19: K-map for the example 3](image)

The simplified expression in the POS form is given by;

\[
F = (A + B') \cdot (B' + D') \cdot (B + D) \cdot (A' + C' + D')
\]  

(17)

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

\[
F = (A + B') \cdot (B' + D') \cdot (B + D) \cdot (A' + B' + C')
\]  

(18)

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 all the 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 all sum terms.

**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. A typical example is the BCD decoders, where ten outputs are decoded from sixteen possible input combinations produced by four inputs representing BCD codes. Whatever encoding scheme is used, there are six combinations of the inputs that would be considered as invalid codes with respect to the other ten that were chosen by the encoding scheme. 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 "dont-care" situations. These dont-care situations can be used advantageously in generating a simpler Boolean
Such don't-care combinations of the variables are represented by an "X" in the appropriate cell of the K-map.

Example 4: 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

<table>
<thead>
<tr>
<th>Mode No</th>
<th>Input Code</th>
<th>Output</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0 0 1</td>
<td>0</td>
<td>Input from keyboard</td>
</tr>
<tr>
<td>2</td>
<td>0 1 0</td>
<td>0</td>
<td>Input from mouse</td>
</tr>
<tr>
<td>3</td>
<td>0 1 1</td>
<td>0</td>
<td>Input from light-pen</td>
</tr>
<tr>
<td>4</td>
<td>1 0 0</td>
<td>1</td>
<td>Output to printer</td>
</tr>
<tr>
<td>5</td>
<td>1 0 1</td>
<td>1</td>
<td>Output to plotter</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>X</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>X</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>X</td>
</tr>
</tbody>
</table>

The K-map for this function is given in the figure 20
The function in the SOP and POS forms may be written as in the following:

\[ F = \Sigma (4, 5) + d (0, 6, 7) \]  \hspace{1cm} (19)

\[ F = \Pi (1, 2, 3) \cdot d (0, 6, 7) \]  \hspace{1cm} (20)

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 map shown in the figure 4.20 can be redrawn in two different ways as shown in the figure 21.

The simplification can be done, as seen from the figure 20, in two different ways. The resulting expressions for the function \( F \) are:

\[ F = A \] \hspace{1cm} (21)

\[ F = AB' \] \hspace{1cm} (22)

By utilizing some of the don’t-care conditions as 1s it was possible to generate a simpler expression for the function.

**Example 5:** Simplify \( F = \Sigma (0,1,4,8,10,11,12) + d(2,3,6,9,15) \)
The K-map of this function is given in the figure 22.

The simplified expression taking the full advantage of the don't care is,

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

Let us consider simplification of several functions which are defined in terms 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 6:** Consider the following set of functions defined on the same set of variables:

- $$F_1(A, B, C) = \Sigma(0, 3, 4, 5, 6)$$
- $$F_2(A, B, C) = \Sigma(1, 2, 4, 6, 7)$$
- $$F_3(A, B, C) = \Sigma(1, 3, 4, 5, 6)$$

Let us first consider the simplification process independently for each of the functions. The K-maps for the three functions and the groupings are shown in the figure 23.

The resultant minimal expressions are:

- $$F_1 = B'C' + AC' + AB' + A'BC$$
- $$F_2 = BC' + AC' + AB + A'B'C$$
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 24.

The resultant minimal expressions are:

F1 = B'C' + AC' + AB' + A'B'C
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.

FIG. 24: Better groupings of the individual K-maps of example 6
Principle Of Quine-Mcclusky Method

Quine-McClusky method is a two stage simplification process

Step 1: Prime implicants are generated by a special tabulation process

Step 2: A minimal set of implicants is determined

Step 1: Tabulation

List the specified minter's for the 1s of a function and dont-cares

Generate all the prime implicants using logical adjacency

\[(AB' + AB = A)\]
Principle Of Quine-Mcclusky Method (Contd…) 

Work with the equivalent binary number of the product terms.

Example: A'BCD and A'BC'D' are entered as 0110 and 0100

Combined to form a term “01-0”

Step 2: Prime implicant table.

Selected prime implicants are combined
Example 1 (Contd…)

Compare every binary number in each section with every binary number in the next section

Identify the combinations where the two numbers differ from each other with respect to only one bit.

Combinations cannot occur among the numbers belonging to the same section

Example: 0001 (1) in section 1 can be combined with 0101 (5) in section 2 to result in 0-01 (1, 5).
Quine - McCluskey Method of Logic Function Minimisation

Motivation
Map methods unsuitable if the number of variables is more than six
Quine formulated the concept of tabular minimisation in 1952 improved by McClusky in 1956
Quine-McClusky method
• Can be performed by hand, but tedious, time consuming and subject to error
• Better suited to implementation on a digital computer
Example 1 (Contd…)

The results of such combinations are entered into another column.
The paired entries are checked off.
The entries of one section in the second column can again be
combined together with entries in the next section.
Continue this process.
Example 1 (Contd...)  

<table>
<thead>
<tr>
<th>Section</th>
<th>Column 1</th>
<th>Column 2</th>
<th>Column 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No.of 1s</td>
<td>Binary</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Decimal</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0001 ✓</td>
<td>1-01 (1,5)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>-001 (1,9)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0-10 (2,6) ✓</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>-010 (2,10) ✓</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>0101 ✓</td>
<td>01-1 (5,7)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>011- (6,7)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>-110 (6,14) ✓</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>10-1 (9,11)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>101- (10,11)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1-10 (10,14) ✓</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>0111 ✓</td>
<td>0-1111 (7)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1011 ✓</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1110 ✓</td>
</tr>
</tbody>
</table>

Note: Combination of entries in column 2 can only take place if the corresponding entries have the dashes at the same place.
Example 1 (Contd…)

All those terms which are not checked off constitute the set of prime implicants

The repeated terms should be eliminated

(--10 in the column 3)

The seven prime implicants:

(1,5), (1,9), (5,7), (6,7), (9,11), (10,11), (2,6,10,14)

This is not a minimal set of prime implicants

Minimal The next stage is to determine the minimal set of prime implicants
Prime Implicant Table

Each column represents a decimal equivalent of the minterm.

A row 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.
Prime Implicant Table

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

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A^CD$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$B^CD$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$ABD$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$ABC$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$A'B'C$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$A'^CD$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$CD$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(a)
Selection of Minimal Set of Implicants

Determine essential implicants

- It is the minterm not covered by any other prime implicant

- Identified by columns that have only one asterisk columns
  2 and 14 have only one asterisk each

- The associated row, CD', is an essential prime implicant.
Selection of Minimal Set of Implicants (Contd…)

CD' is selected as a member of the minimal set
(mark it by an asterisk)

Remove the corresponding columns, 2, 6, 10, 14, from the prime implicant table

A new table is prepared.
Selection of Minimal Set of Implicants
(Contd…)

(b)
Dominating Prime Implicants

Identified by the rows that have more asterisks than others

Choose 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 $A'BC$

$A'BC$ can be eliminated

Mark $A'BD$ by an asterisk

Check off the columns 5 and 7
Dominating Prime Implicants (Contd…)

Choose AB'D

Dominates over the row AB'C

Mark the row AB'D by an asterisk

Eliminate the row AB'C

Check off columns 9 and 11
Dominating Prime Implicants (Contd…)

Select \( A'C'D \)

Dominates over \( B'C'D \).

\( B'C'D \) also dominates over \( A'C'D \)

Either \( B'C'D \) or \( A'C'D \) can be chosen as the dominant prime implicant
Minimal SOP Expression

If A'C'D is retained as the dominant prime implicant

\[ F = CD' + A'C'D + A'BD + AB'D \]

If B'C'D is chosen as the dominant prime implicant

\[ F = CD' + B'C'D + A'BD + AB'D \]

The minimal expression is not unique
Types of Implicant Tables

Cyclic prime implicant table

Semi-cyclic prime implicant table

A prime implicant table is cyclic if

It does not have any essential implicants which implies (at least two asterisks in every column)

There are no dominating implicants (same number of asterisks in every row)
Example: Cyclic Prime Implicants

$$F = S \{0, 1, 3, 4, 7, 12, 14, 15\}$$
Example: Possible Prime Implicants

\[ \begin{align*}
  a &= A'B'C' \ (0,1) & e &= ABC \ (14,15) \\
  b &= A'B'D \ (1,3) & f &= ABD' \ (12,14) \\
  c &= A'CD \ (3,7) & g &= BC'D' \ (4,12) \\
  d &= BCD \ (7,15) & h &= A'C'D' \ (0,4)
\end{align*} \]
Example: Prime Implicant Table

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>3</th>
<th>4</th>
<th>7</th>
<th>1</th>
<th>14</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>e</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>f</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>g</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>h</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

- (0,1)
- (1,3)
- (3,7)
- (7,15)
- (14,15)
- (12,14)
- (4,12)
- (0,4)
Process of Simplification

All columns have two asterisks
There are no essential prime implicants.
Choose any one of the prime implicants to start with
Start with prime implicant a (mark with asterisk)
Delete corresponding columns, 0 and 1
Process of Simplification (Contd…) 

Row c becomes dominant over row b, delete row b

Delete columns 3 and 7

Row e dominates row d, and row d can be eliminated

Delete columns 14 and 15

Choose row g it covers the remaining asterisks associated with rows h and f.
Example: Reduced Prime Implicants

Table

<table>
<thead>
<tr>
<th></th>
<th>3</th>
<th>4</th>
<th>7</th>
<th>12</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(3,7)</td>
</tr>
<tr>
<td>d</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(7,15)</td>
</tr>
<tr>
<td>e</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(14,15)</td>
</tr>
<tr>
<td>f</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(12,14)</td>
</tr>
<tr>
<td>g</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(4,12)</td>
</tr>
<tr>
<td>h</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(0,4)</td>
</tr>
</tbody>
</table>

(a)

<table>
<thead>
<tr>
<th></th>
<th>4</th>
<th>12</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>f</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>g</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>h</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(b)
Example: Simplified Expression

\[ F = a + c + e + g \]

\[ = A'B'C' + A'CD + ABC + BC'D' \]
Example: Simplified Expression

The K-map of the simplified function is shown below.
Semi-Cyclic Prime Implicant Table

The number of minterms covered by each prime implicant is identical in cyclic function

Not necessarily true in a semi-cyclic function
Semi-Cyclic Prime Implicant Table: Example

Function of 5 variables

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>2</th>
<th>8</th>
<th>10</th>
<th>11</th>
<th>15</th>
<th>16</th>
<th>18</th>
<th>19</th>
<th>23</th>
<th>25</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td></td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
<td></td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td></td>
<td></td>
<td></td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>f</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>g</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>h</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>j</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>k</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(0,2,8,10)  (0,2,16,18)  (8,9,10,11)  (16,17,18,19)  (11,15)  (15,31)  (23,31)  (19,23)  (17,25)  (25,9)
Minimised Function: Example

\[ F = a + c + d + e + h + j \]
\[ \text{or } F = a + c + d + g + h + j \]
\[ \text{or } F = a + c + d + g + j + i \]
\[ \text{or } F = a + c + d + g + i + k \]
Simplification of Incompletely Specified functions

Do the initial tabulation including the don’t-cares

Construct the prime implicant table

Columns associated with don’t-cares need not be included

Further simplification is similar to that for completely specified functions
Example

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

Pay attention to the don’t-care terms

Mark the combinations among themselves (d)
## Primary Implicant Table

<table>
<thead>
<tr>
<th>Primary Implicant</th>
<th>Implicant</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000 (d)</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>00001</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>00100</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>10000 (d)</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>00110</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>01010</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>10100</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>11000</td>
<td>24</td>
<td></td>
</tr>
<tr>
<td>10110</td>
<td>22</td>
<td></td>
</tr>
<tr>
<td>11010</td>
<td>26</td>
<td></td>
</tr>
<tr>
<td>01011 (d)</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>11011 (d)</td>
<td>27</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Primary Implicant</th>
<th>Implicant</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000 - (0,1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-0000 - (0,16) (d)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-0010 - (4,6,20)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-1001 - (10,11,27)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0101 - (10,26)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-0110 - (6,22)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-1010 - (10,26)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0101 - (10,11)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1010 - (20,22)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1100 - (24,26)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1101 - (26,26)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10111(11,27)(d)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Prime Implicant Table

Don’t-cares are not included

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>4</th>
<th>6</th>
<th>10</th>
<th>20</th>
<th>22</th>
<th>24</th>
<th>26</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>f</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(0,1) (16,24) (24,26) (0,4,6,23) (4,6,20,22) (10,11,26,27)
The minimal expression is given by:

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

\[ = A'B'C'D' + ABC'E' + B'CE' + BC'D \]
QUINE-McCLUSKEY METHOD OF MINIMISATION

Introduction

K-Map based minimisation is a powerful design tool. While it is certainly more convenient and systematic than pure algebraic method, it is still not a completely systematic method. Both K-map and VEM methods still depend on our ability to observe appropriate patterns and identify the necessary implicants. It is not very difficult, though no systematic method exists, to identify the appropriate implicants for maps with up to five variables. However, the minimised expression generated for a given function is neither necessarily unique nor optimal. The selection of the optimal may depend on other criteria like the cost, the number of literals and the number of product terms. But once the number of variables is more than five or six, the map based methods are no longer convenient. We require 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 method refers to the tabular method of minimisation.

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.

Principle of Quine-McClusky Method

The Quine-McClusky method is a two stage simplification process. Prime implicants of the given Boolean function are generated by a special tabulation process in the first stage. A minimal set of implicants is determined, in the second stage, 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 dont-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 $A'B'CD'$ 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.

**Generation of Prime Implicants**

The process of generating prime implicants is best presented through an example. Consider the following Example:

**Example 1**

$$F = \Sigma (1,2,5,6,7,9,10,11,14)$$

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 1.

<table>
<thead>
<tr>
<th>Section</th>
<th>Column 1 No.of 1s</th>
<th>Column 1 Binary</th>
<th>Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0001 0010</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>0101 0110 1001</td>
<td>5 6 9</td>
</tr>
</tbody>
</table>
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 2

Table 2: Prime implicant generation of Example 1

<table>
<thead>
<tr>
<th>Section</th>
<th>No. of 1s</th>
<th>Column 1</th>
<th>Column 2</th>
<th>Column 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Binary</td>
<td>Decimal</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0001 ✓</td>
<td>1</td>
<td>1-01</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0010 ✓</td>
<td>2</td>
<td>-001</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>0101 ✓</td>
<td>5</td>
<td>0-10</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0110 ✓</td>
<td>6</td>
<td>-010</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1001 ✓</td>
<td>9</td>
<td>(2,6) ✓</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1010 ✓</td>
<td>10</td>
<td>(2,10) ✓</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>0111 ✓</td>
<td>7</td>
<td>01-1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1011 ✓</td>
<td>11</td>
<td>011-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1110 ✓</td>
<td>14</td>
<td>-110</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>10-1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>101-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1-10</td>
</tr>
</tbody>
</table>

---

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 2.
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 to 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.

**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 Eqn.1 is shown in the figure 1(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 which covers certain, 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 as shown in the figure 1(b) is prepared. We then select dominating prime implicants, which are the rows that have more asterisks than others. For example, the row $A'B'D$ 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 $A'B'D$ by an asterisk, and check off the columns 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 $\{CD', A'C'D, A'BD, AB'D\}$. 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

\[ F = \Sigma (0,1,3,4,7,12,14,15) \]

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 3 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.
If, from the reduced prime implicant table shown in the figure 3, 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:

\[
F = a + c + e + g
\]
= A'B'C' + A'CD + ABC + BC'D'

The K-map of the simplified function is shown in the figure 5.

FIG.5: Simplified K-map of the function of example 2

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 6.
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 semi-cyclic.
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.

**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 don't-cares. However, when the prime implicant table is constructed, columns associated with don't-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:

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

Tabulation is shown in the figure 6. Pay attention to the dont-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:

\[
\begin{align*}
  a &= A'B'C'D' \\
  d &= B'D'E' \\
  e &= B'CE' \\
  c &= ABC'E' \\
  g &= BC'D
\end{align*}
\]

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>(d)</td>
<td>0 ✓</td>
<td>0000- (0,1)</td>
<td>-00-00 (0,4) ✓</td>
</tr>
<tr>
<td>00001</td>
<td>1 ✓</td>
<td>-0000 (0,16) (d)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00100</td>
<td>4 ✓</td>
<td>0010 (4,6) ✓</td>
<td>-0100 (4,20) ✓</td>
<td></td>
</tr>
<tr>
<td>10000</td>
<td>(d)</td>
<td>16 ✓</td>
<td>0010 (4,6) ✓</td>
<td>-0100 (4,20) ✓</td>
</tr>
<tr>
<td>00110</td>
<td>6 ✓</td>
<td>1000 (16,20) ✓</td>
<td>-101- (10,11,26,27)</td>
<td></td>
</tr>
<tr>
<td>01010</td>
<td>10 ✓</td>
<td>0100 (16,24)</td>
<td>-0110 (6,22) ✓</td>
<td></td>
</tr>
<tr>
<td>10100</td>
<td>20 ✓</td>
<td>-1010 (10,26) ✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11000</td>
<td>24 ✓</td>
<td>-1010 (10,26) ✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10110</td>
<td>22 ✓</td>
<td>0101- (10,11) ✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11010</td>
<td>26 ✓</td>
<td>101-0 (20,22) ✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01011</td>
<td>(d)</td>
<td>11 ✓</td>
<td>110-0 (24,26)</td>
<td></td>
</tr>
<tr>
<td>11011</td>
<td>(d)</td>
<td>27 ✓</td>
<td>1101- (26,26) ✓</td>
<td></td>
</tr>
</tbody>
</table>

**FIG.7:** Tabulation of implicants for the example 4
The prime implicant table is plotted as shown in the figure 8. It may be noted that the
dont-cares are not included.

\[
\begin{align*}
&\text{FIG.8: Prime implicant table for the example 4} \\
&\begin{array}{cccccc}
1 & 4 & 6 & 10 & 22 & 24 & 26 \\
a & x & x & x & x & x & (0,1) \\
b & x & x & x & x & x & (16,24) \\
c & x & x & x & x & x & (24,26) \\
d & x & x & x & x & x & (0,4,6,23) \\
e & x & x & x & x & x & (4,6,20,22) \\
g & x & x & x & x & x & (10,11,26,27)
\end{array}
\end{align*}
\]

The minimal expression is given by:

\[
F(A,B,C,D,E) = a + c + e + g \\
= A'B'C'D' + ABC'E' + B'CE' + BC'D
\]
REFERENCES

   Chapter 1, 3 and 4 serve as references to the material presented in this Module. Chapter 2 deals with the number systems. The treatment is very elaborate and a very large number of problems are given for working out.

   Chapter 3 serves as the reference. The treatment of Boolean algebra is very formal.

   Chapter 1 is dedicated to switching algebra and switching function. The treatment in this book does not assume the knowledge of number systems. Quine-McClusky method and Iterative Consensus methods of minimisation are presented.


5. Shannon, C.E., A Symbolic Analysis of Relay and Switching Circuits, Trans. AIEE, Vol. 57, pp. 713-23,