NATIONAL OPEN UNIVERSITY OF NIGERIA

SCHOOL OF SCIENCE AND TECHNOLOGY

COURSE CODE: CIT344

COURSE TITLE:
INTRODUCTION TO COMPUTER DESIGN

COURSE GUIDE

CIT344
INTRODUCTION TO COMPUTER DESIGN

Course Team Adaora Obayi (Developer/Writer) - NOUN
Dr. Oyebanji (Programme Leader) - NOUN
Vivian Nwaocha (Coordinator) -NOUN

NATIONAL OPEN UNIVERSITY OF NIGERIA

National Open University of Nigeria
Headquarters
14/16 Ahmadu Bello Way
Victoria Island
Lagos

Abuja Office
5, Dar es Salaam Street
Off Aminu Kano Crescent
Wuse II, Abuja
Nigeria

e-mail: centralinfo@nou.edu.ng
URL: www.nou.edu.ng

Published By:
National Open University of Nigeria

First Printed 2012

ISBN: 978-058-047-6

CONTENTS **PAGE**

Introduction

CIT344: Introduction to Computer Design is a 3-credit unit course for students studying towards acquiring the Bachelor of Science in Information Technology and related disciplines.

The course is divided into 6 modules and 21 study units. It introduces you to concepts in Computer Design and their implementations in our everyday lives.

This course also provides information on numbers and codes in computer design, different logic designs, memory devices, microprocessors and finally, a type of programming called Assembly Language Programming.

At the end of this course, it is expected that you should be able to understand, explain and be adequately equipped with comprehensive knowledge of logic designs and can try your hands in some designs of your own.

This course guide therefore gives you an overview of what the course is all about, the textbooks and other course materials to be used, what you are expected to know in each unit, and how to work through the course material.

Furthermore, it suggests the general strategy to be adopted and also emphasises the need for self-assessment and tutor-marked assignment. There are also tutorial classes that are linked to this course and you are advised to attend them.

What You Will Learn in This Course

The overall aim of this course is to boost your knowledge of logic designs, microprocessors and assembly language programming. In the course of your studies, you will be equipped with definitions of common terms, characteristics and applications of logic designs. You will also learn number systems and codes, memory devices, microprocessors and finally, loops and subroutines in assembly language.

Course Aim

This course aims to give you an in-depth understanding of computer designs. It is hoped that the knowledge would enhance your expertise in logic designs.

Course Objectives

It is relevant to note that each unit has its precise objectives. You should learn them carefully before proceeding to subsequent units. Therefore, it is useful to refer to these objectives in the course of your study of the unit to assess your progress. You should always look at the unit objectives after completing a unit. In this way, you can be sure that you have done what is required of you by the end of the unit. However, below are overall objectives of this course. On successful completion of this course, you should be able to:

- explain the term number system and its various types
- state the various conversion from one number system to the other
- explain the various types of codes
- analyse and design a combinational logic circuit
- describe what a sequential logic circuit is
- state the differences between combinational and sequential logic circuit
- list the types of sequential logic circuit
- describe what a latch and flip-flop is
- describe what shift register is
- discuss about finite state machines
- describe memory and the basic operations performed on it
- state the types of memory we have
- describe microprocessors
- write a program using assembly language.

Working through This Course

To complete this course, you are required to study all the units, the recommended text books, and other relevant materials. Each unit contains tutor-marked assignments, and at some point in this course, you are required to submit the tutor-marked assignments. There is also a final examination at the end of this course. Stated below are the components of this course and what you have to do.

Course Materials

The major components of the course are:

1. Course Guide
2. Study Units
3. Text Books
4. Assignment File
5. Presentation Schedule

Study Units

There are 6 modules and 21 study units in this course. They are:

Module 1 Introduction to Numbers and Codes

Unit 1 Types of Number Systems I
Unit 2 Types of Number Systems II
Unit 3 Codes

Module 2 Combinational Logic Design and Application

Unit 1 Analysis and Design of a Combinational Logic Circuit
Unit 2 Typical Combinational Logic Circuit I
Unit 3 Typical Combinational Logic Circuit II
Unit 4 Typical Combinational Logic Circuit III

Module 3 Sequential Logic Design and Applications

Unit 1 Sequential Logic Circuits
Unit 2 Latches and Flip-Flops
Unit 3 Registers
Unit 4 Finite State Machines

Module 4 Memory Devices

Unit 1 Memory Organisation
Unit 2 Memory Types
Unit 3 Memory Expansion
Unit 4 Memory Summary

Module 5 Introduction to Microprocessors

Unit 1 Microprocessors
Unit 2 Central Processing Unit and Arithmetic and Logical Unit
Unit 3 Addressing Mode

Module 6 Assembly Language Programming

Unit 1 Learning to Program with Assembly Language
Unit 2 Branching Loops and Subroutines
Unit 3 Sample Programs in Assembly Language

Textbooks and References

These texts listed below will be of enormous benefit to you in learning this course:

Baase, Sara (1983). *VAX-11 Assembly Language Programming*. San Diego: Prentice-Hall, Inc.

Brookshear, J. G. (1997). *Computer Science: An Overview*. Addison - Wesley.

Comer, J. David (1994). *Digital Logic & State Machine Design*. Oxford: Oxford University Press.

Dandamudi, S. (2003). *Fundamentals of Computer Organization and Design*. Springer.

Emerson, W. Pugh *et al* (1991). *IBM's 360 and Early 370 Systems*. MIT Press.

Ford, William & Topp, William (1996). *Assembly Language and Systems Programming for the M68000 Family*. Jones & Bartlett Pub.

Ford, William & Topp, William (1992). *Macintosh Assembly System, Version 2.0*. Jones & Bartlett Pub.

Hayes, P. John (1993). *Introduction to Digital Logic Design*. Prentice Hall.

Hennessy, L. John & Patterson, A. David (2008). *Computer Organization & Design*. Morgan Kaufmann.

Holdsworth, Brian & Woods, Clive (2002). *Digital Logic Design*. Newnes.

Hummel, Robert L. (1992). *Programmer's Technical Reference: The Processor and Coprocessor*. Ziff-Davis Press.

Mano, M. Morris & Kime, Charles R. (2007). *Logic and Computer Design Fundamentals*. Prentice Hall.

Marcovitz, Alan (2009). *Introduction to Logic Design*. McGraw-Hill.

Neamen, Donald (2009). *Microelectronics: Circuit Analysis & Design*. McGraw-Hill.

Nelson, P. *et al* (1995). *Digital Logic Circuit Analysis & Design*. Prentice Hall.

Palmer, James & Periman, David (1993). *Schaum's Outline of Introduction to Digital Systems*. McGraw-Hill.

Patterson, David & Hennessy, John (1993). *Computer Organization & Design: The Hardware/Software Interface*. Morgan Kaufmann Publishers.

Patterson, David & Hennessy, John (1990). *Computer Architecture, A Quantitative Approach*. Morgan Kaufmann Publishers.

Rafiquzzaman, M. (2005). *Fundamentals of Digital Logic & Microcomputer Design*. Wiley – Interscience.

Roth, H. Charles Jr & Kinney, L. Larry (2009). *Fundamentals of Logic Design*.

Sanchez, Julio & Maria, P. Canton (1990). *IBM Microcomputers: A Programmer's Handbook*. McGraw-Hill.

Struble, George W. (1975). *Assembler Language Programming The IBM System/360 and 370.* Addison-Wesley Publishing Company.

Tanenbaum, Andrew S. (1998). *Structured Computer Organization.* Prentice Hall. Tocci, J. Ronald & Widmer, S. Neal (1994). *Digital Systems: Principles and Applications.*

Tokheium, Roger (1994). *Schaum's Outline of Digital Principles*. McGraw-Hill.

Wakerly, F. John (2005). *Digital Design: Principles & Practices Package*. Prentice Hall.

Wear, Larry *et al.* (1991). *Computers: An Introduction to Hardware and Software Design*. McGraw-Hill.

www.cs.siu.edu

www.educypedia.be/electronics

www.books.google.com

Assignment File

The assignment file will be given to you in due course. In this file, you will find all the details of the work you must submit to your tutor for marking. The marks you obtain for these assignments will count towards the final mark for the course. Altogether, there are 21 tutor-marked assignments for this course.

Presentation Schedule

The presentation schedule included in this course guide provides you with important dates for completion of each tutor-marked assignment. You should therefore endeavour to meet the deadlines.

Assessment

There are two aspects to the assessment of this course. First, there are tutor-marked assignments; and second, the written examination.

You are expected to take note of the facts, information and problem solving gathered during the course. The tutor-marked assignments must be submitted to your tutor for formal assessment, in accordance to the deadline given. The work submitted will count for 40% of your total course mark. At the end of the course, you will need to sit for a final written examination. This examination will account for 60% of your total score.

Tutor-Marked Assignments (TMAs)

There are 21 TMAs in this course. You need to submit all the TMAs. When you have completed each assignment, send them to your tutor as soon as possible and make certain that it gets to your tutor on or before the stipulated deadline. If for any reason you cannot complete your assignment on time, contact your tutor before the assignment is due to discuss the possibility of extension. Extension will not be granted after the deadline, unless in extraordinary cases.

Final Examination and Grading

The final examination for CIT344 will be of last for a period of 3 hours and have a value of 60% of the total course grade. The examination will consist of questions which reflect the self-assessment exercise and tutor-marked assignments that you have previously encountered. Furthermore, all areas of the course will be examined. It would be better to use the time between finishing the last unit and sitting for the examination, to revise the entire course. You might find it useful to review your TMAs

and comment on them before the examination. The final examination covers information from all parts of the course.

Course Marking Scheme

The following table includes the course marking scheme

Table 1: Course Marking Scheme

Assessment	**Marks**
Assignments 1-21	21 assignments, 40% for the best 4 Total = 10% X 4 = 40%
Final Examination	60% of overall course marks
Total	100% of Course Marks

Course Overview

This indicates the units, the number of weeks required to complete them and the assignments.

Table 2: Course Organiser

Unit	**Title of Work**	**Weeks Activity**	**Assessment (End of Unit)**
	Course Guide	Week 1	
Module 1	**Introduction to Numbers and Codes**		
1	Types of Number Systems I	Week 1	Assignment 1
2	Types of Number Systems II	Week 2	Assignment 2
3	Codes	Week 3	Assignment 3
Module 2	**Combinational Logic Design and Applications**		
1	Analysis & Design of a Combinational Logic Circuit	Week 3	Assignment 4
2	Typical Combinational Logic Circuit I	Week 4	Assignment 5
3	Typical Combinational Logic Circuit II	Week 4	Assignment 6
4	Typical Combinational Logic Circuit III	Week 5	Assignment 7
Module 3	**Sequential Logic Design and Applications**		
1	Sequential Logic Circuits	Week 5	Assignment 8
2	Latches and Flip-Flops	Week 6	Assignment 9
3	Registers	Week 6	Assignment 10
4	Finite State Machines	Week 7	Assignment 11

Module 4	**Memory Devices**		
1	Memory Organisation	Week 7	Assignment 12
2	Memory Types	Week 8	Assignment 13
3	Memory Expansion	Week 9	Assignment 14
4	Memory Summary	Week 10	Assignment 15
Module 5	**Introduction To Microprocessors**		
1	Microprocessors	Week 10	Assignment 16
2	Central Processing Unit & Arithmetic & Logical Unit	Week 11	Assignment 17
3	Addressing Mode	Week 12	Assignment 18
Module 6	**Assembly Language Programming**		
Unit 1	Learning to Program with Assembly Language	Week 13	Assignment 19
Unit 2	Branching Loops and Subroutine	Week 14	Assignment 20
Unit 3	Sample Programs in Assembly Language	Week 14	Assignment 21

How to Get the Most Out of This Course

In distance learning, the study units replace the university lecturer. This is one of the huge advantages of distance learning mode; you can read and work through specially designed study materials at your own pace and at a time and place that is most convenient. Think of it as reading from the teacher, the study guide indicates what you ought to study, how to study it and the relevant texts to consult. You are provided with exercises at appropriate points, just as a lecturer might give you an in-class exercise.

Each of the study units follows a common format. The first item is an introduction to the subject matter of the unit and how a particular unit is integrated with the other units and the course as a whole. Next to this is a set of learning objectives. These learning objectives are meant to guide your studies. The moment a unit is finished, you must go back and check whether you have achieved the objectives. If this is made a habit, then you will increase your chances of passing the course.

The main body of the units also guides you through the required readings from other sources. This will usually be either from a set book or from other sources. Self assessment exercises are provided throughout the unit, to aid personal studies and answers are provided at the end of the unit. Working through these self tests will help you to achieve the objectives of the unit and also prepare you for tutor marked assignments and examinations. You should attempt each self test as you encounter them in the units.

Read the course guide thoroughly and organise a study schedule. Refer to the course overview for more details. Note the time you are expected to spend on each unit and how the assignment relates to the units. Important details, e.g. details of your tutorials and the date of the first day of the semester are available. You need to gather together all these information in one place such as a diary, a wall chart calendar or an organiser. Whatever method you choose, you should decide on and write in your own dates for working on each unit.

Once you have created your own study schedule, do everything you can to stick to it. The major reason that students fail is that they get behind with their course works. If you get into difficulties with your schedule, please let your tutor know before it is too late for help.

Turn to unit 1 and read the introduction and the objectives for the unit.

Assemble the study materials. Information about what you need for a unit is given in the table of content at the beginning of each unit. You will almost always need both the study unit you are working on and one of the materials recommended for further readings, on your desk at the same time.

Work through the unit, the content of the unit itself has been arranged to provide a sequence for you to follow. As you work through the unit, you will be encouraged to read from your set books.

Keep in mind that you will learn a lot by doing all your assignments carefully. They have been designed to help you meet the objectives of the course and will help you pass the examination.

Review the objectives of each study unit to confirm that you have achieved them. If you are not certain about any of the objectives, review the study material and consult your tutor.

When you are confident that you have achieved a unit's objectives, you can start on the next unit. Proceed unit by unit through the course and try to pace your study so that you can keep yourself on schedule.

When you have submitted an assignment to your tutor for marking, do not wait for its return before starting on the next unit. Keep to your schedule. Pay particular attention to your tutor's comments on the tutor-marked assignment form and also written on the assignment when the assignment is returned to you. Consult you tutor as soon as possible if you have any questions or problems.

After completing the last unit, review the course and prepare yourself for the final examination. Check that you have achieved the unit objectives (listed at the beginning of each unit) and the course objectives (listed in this course guide).

Facilitators/Tutors and Tutorials

There are 8 hours of tutorial provided in support of this course. You will be notified of the dates, time and location together with the name and phone number of your tutor as soon as you are allocated a tutorial group.

Your tutor will mark and comment on your assignments, keep a close watch on your progress and on any difficulties you might encounter and provide assistance to you during the course. You must mail your tutor marked assignment to your tutor well before the due date. At least two working days are required for this purpose. They will be marked by your tutor and returned to you as soon as possible. Do not hesitate to contact your tutor by telephone, e-mail or discussion board if you need help. The following might be circumstances in which you would find help necessary:

- you do not understand any part of the study units or the assigned readings
- you have difficulty with the self test or exercise
- you have questions or problems with an assignment, with your tutor's comments on an assignment or with the grading of an assignment.

You should try your best to attend the tutorials. This is the only chance to have face-to-face contact with your tutor and ask questions which are answered instantly. You can raise any problem encountered in the course of your study. To gain the maximum benefit from the course tutorials, prepare a question list before attending them. You will learn a lot from participating actively in tutorial discussions.

Course Code CIT344
Course Title Introduction to Computer Design

Course Team Adaora Obayi (Developer/Writer) - NOUN
Dr. Oyebanji (Programme Leader) - NOUN
Vivian Nwaocha (Coordinator) -NOUN

NATIONAL OPEN UNIVERSITY OF NIGERIA

National Open University of Nigeria
Headquarters
14/16 Ahmadu Bello Way
Victoria Island
Lagos

Abuja Office
5, Dar es Salaam Street
Off Aminu Kano Crescent
Wuse II, Abuja
Nigeria

e-mail: centralinfo@nou.edu.ng
URL: www.nou.edu.ng

Published By:
National Open University of Nigeria

First Printed 2012

ISBN: 978-058-047-6

CONTENTS PAGE

MODULE 1 INTRODUCTION TO NUMBERS AND CODES

UNIT 1 TYPES OF NUMBER SYSTEMS I

CONTENTS

1.0 INTRODUCTION

The number system is the basis of computing. It is a very important foundation for understanding the way the computer system works. In this unit, we will talk about decimal and binary number system. Endeavour to assimilate as much as possible from this unit – especially, the conversion from one number system to another.

2.0 OBJECTIVES

At the end of this unit, you should be able to:

- explain the term decimal number system
- manipulate fractions of decimal numbers
- explain the term binary number system
- manipulate binary arithmetic
- convert binary to decimal
- convert decimal to binary.

3.0 MAIN CONTENT

3.1 Decimal Number System

The decimal number system has ten unique digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. Using these single digits, ten different values can be represented. Values greater than ten can be represented by using the same digits in different combinations. Thus, ten is represented by the number 10, two hundred seventy five is represented by 275, etc. Thus, same set of numbers 0, 1, 2, … 9 are repeated in a specific order to represent larger numbers. The decimal number system is a positional number system as the position of a digit represents its true magnitude. For example, 2 is less than 7, however 2 in 275 represents 200, whereas 7 represents 70. The left most digit has the highest weight and the right most digit has the lowest weight. 275 can be written in the form of an expression in terms of the base value of the number system and weights.

2 x 102 + 7 x 101 + 5 x 100 = 200 + 70 + 5 = 275

where, 10 represents the base or radix,102, 101, 100 represent the weights 100, 10 and 1 of the numbers 2, 7 and 5.

Fractions in Decimal Number System

In a Decimal Number System the fraction part is separated from the integer part by a decimal point. The integer part of a number is written on the left hand side of the decimal point. The fraction part is written on the right side of the decimal point. The digits of the integer part on the left hand side of the decimal point have weights 100, 101, 102 etc. respectively starting from the digit to the immediate left of the decimal point and moving away from the decimal point towards the most significant digit on the left hand side. Fractions in decimal number system are also represented in terms of the base value of the number system and weights. The weights of the fraction part are represented by 10-1, 10-2, 10-3, etc. The weights decrease by a factor of 10 moving right of the decimal point. The number 382.91 in terms of the base number and weights is represented as

$3 \times 102 + 8 \times 101 + 2 \times 100 + 9 \times 10^{-1} + 1 \times 10^{-2} = 300 + 80 + 2 + 0.9 + 0.01 = 382.91$

3.2 Binary Number System

Binary as the name indicates is a base-2 number system having only two numbers 0 and 1. The binary digit 0 or 1 is known as a 'Bit'. Below is the decimal equivalent of the binary number system.

Table 1: Decimal Equivalents of Binary Number System

Decimal Number	**Binary Number**	**Decimal Number**	**Binary Number**
0	0	10	1010
1	1	11	1011
2	10	12	1100
3	11	13	1101
4	100	14	1110
5	101	15	1111
6	110	16	10000
7	111	17	10001
8	1000	18	10010
9	1001	19	10011
		20	10100

Counting in binary number system is similar to counting in decimal number systems. In a decimal number system a value larger than 9 has to be represented by 2, 3, 4, or more digits. Similarly, in the binary number system a binary number larger than 1 has to be represented by 2, 3, 4, or more binary digits.

Any binary number comprising of binary 0 and 1 can be easily represented in terms of its decimal equivalent by writing the binary number in the form of an expression using the base value 2 and weights 20, 21, 22, etc.

The number 10011_2 (the subscript 2 indicates that the number is a binary number and not a decimal number ten thousand and eleven) can be rewritten in terms of the expression:

$10011_2 = (1 \times 24) + (0 \times 23) + (0 \times 22) + (1 \times 21) + (1 \times 20)$
$= (1 \times 16) + (0 \times 8) + (0 \times 4) + (1 \times 2) + (1 \times 1)$
$= 16 + 0 + 0 + 2 + 1$
$= 19$

3.2.1 Fractions in Binary Number System

In a decimal number system the integer part and the fraction part of a number are separated by a decimal point. In a binary number system the integer part and the Fraction part of a binary number can be similarly represented separated by a decimal point. The binary number 1011.101_2 has an integer part represented by 1011 and a fraction part 101 separated by a decimal point. The subscript 2 indicates that the number is a binary number and not a decimal number. The binary number 1011.101_2 can be written in terms of an expression using the base value 2 and weights 23, 22, 21, 20, 2^{-1}, 2^{-2} and 2^{-3}.

$1011.101_2 = (1 \times 23) + (0 \times 22) + (1 \times 21) + (1 \times 20) + (1 \times 2^{-1}) + (0 \times 2^{-2}) + (1 \times 2^{-3})$
$= (1 \times 8) + (0 \times 4) + (1 \times 2) + (1 \times 1) + (1 \times 1/2) + (0 \times 1/4) + (1 \times 1/8)$
$= 8 + 0 + 2 + 1 + 0.5 + 0 + 0.125$
$= 11.625$

Computers do handle numbers such as 11.625 that have an integer part and a fraction part. However, it does not use the binary representation 1011.101. Such numbers are represented and used in floating-point numbers notation.

3.2.2 Binary Arithmetic

Digital systems use the binary number system to represent numbers. Therefore these systems should be capable of performing standard arithmetic operations on binary numbers.

Binary Addition

Binary addition is identical to decimal addition. By adding two binary bits, a sum bit and a carry bit are generated. The only difference between the two additions is the range of numbers used. In binary addition, four possibilities exist when two single bits are added together. The four possible input combinations of two single bit binary numbers and their corresponding sum and carry outputs are specified in Table 2.

Table 2: Addition of Two Single Bit Binary Numbers

First Number	Second Number	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

The first three additions give a result 0, 1 and 1 respectively which can be represented by a single binary digit (bit). The fourth addition results in the number 2, which can be represented in binary as 102. Thus, two digits (bits) are required. This is similar to the addition of 9 + 3 in decimal. The answer is 12 which cannot be represented by a single digit; thus, two digits are required. The number 2 is the sum part and 1 is the carry part.

Any number of binary numbers having any number of digits can be added together.

Binary Subtraction

Binary subtraction is identical to decimal subtraction. The only difference between the two is the range of numbers. Subtracting two single bit binary numbers results in a difference bit and a borrow bit. The four possible input combinations of two single bit binary numbers and their corresponding difference and borrow outputs are specified in Table 3. It is assumed that the second number is subtracted from the first number.

Table 3: Subtraction of Two Single Bit Binary Numbers

First Number	Second Number	Difference	Borrow
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

The second subtraction subtracts 1 from 0 for which a borrow is required to make the first digit equal to 2. The difference is 1. This is similar to decimal subtraction when 17 is subtracted from 21. The first digit 7 cannot be subtracted from 1, therefore 10 is borrowed from the next significant digit. Borrowing a 10 allows subtraction of 7 from 11 resulting in a difference of 4.

Binary Multiplication

Binary multiplication is similar to the decimal multiplication except for the range of numbers. Four possible combinations of two single bit binary numbers and their products are listed in table 4.

Table 4: Multiplication of two Single Bit Binary Numbers

First Number	Second Number	Product
0	0	0
0	1	0
1	0	0
1	1	1

Binary Division

Division in binary follows the same procedure as in the division of decimal numbers. Fig 1 illustrates the division of binary numbers.

```
      10
101 | 1101
      101
      ---
       011
       000
       ---
        11
```

Fig. 1 : Binary Division

3.2.3 Binary to Decimal Conversion

Most real world quantities are represented in decimal number system. Digital systems on the other hand are based on the binary number system. Therefore, when converting from the digital domain to the real-world, binary numbers have to be represented in terms of their decimal equivalents. The method used to convert from binary to decimal is the sum-of-weights method.

Sum-of-Weights Method

Sum-of-weights as the name indicates sums the weights of the binary digits (bits) of a binary number which is to be represented in decimal. The sum-of-weights method can be used to convert a binary number of any magnitude to its equivalent decimal representation.

In the sum-of-weights method an extended expression is written in terms of the binary base number 2 and the weights of the binary number to be converted. The weights correspond to each of the binary bits which are multiplied by the corresponding binary value.

Binary bits having the value 0 do not contribute any value towards the final sum expression. The binary number 10110_2 is therefore written in the form of an expression having weights 2^0, 2^1, 2^2, 2^3 and 2^4

corresponding to the bits 0, 1, 1, 0 and 1 respectively. Weights 2^0 and 2^3 do not contribute in the final sum as the binary bits corresponding to these weights have the value 0.

$10110_2 = 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$
$= 16 + 0 + 4 + 2 + 0$
$= 22$

Sum-of-Non-Zero Terms

In the sum-of-weights method, the binary bits 0 do not contribute towards the final sum representing the decimal equivalent. Secondly, the weight of each binary bit increases by a factor of 2 starting with a weight of 1 for the least significant bit. For example, the binary number 101102 has weights 2^0=1, 2^1=2, 2^2=4, 2^3=8 and 2^4=16 corresponding to the bits 0, 1, 1, 0 and 1 respectively.

The sum-of-non-zero terms method is a quicker method to determine decimal equivalents of binary numbers without resorting to writing an expression. In the sum-of-non-zero terms method, the weights of non-zero binary bits are summed, as the weights of zero binary bits do not contribute towards the final sum representing the decimal equivalent.

The weights of binary bits starting from the right most least significant bit is 1, The next significant bit on the left has the weight 2, followed by 4, 8, 16, 32, etc. corresponding to higher significant bits. In binary number system the weights of successive bits increase by an order of 2 towards the left side and decrease by an order of 2 towards the right side. Thus, a binary number can be quickly converted into its decimal equivalent by adding weights of non-zero terms which increase by a factor of 2. Binary numbers having an integer and a fraction part can similarly be converted into their decimal equivalents by applying the same method.

A quicker method is to add the weights of non-zero terms. Thus, for the numbers:
$10011_2 = 16 + 2 + 1 = 19$
$1011.101_2 = 8 + 2 + 1 + ½ + 1/8 = 11 + 5/8 = 11.625$

3.2.4 Decimal to Binary Conversion

Conversion from decimal to binary number system is also essential to represent real-world quantities in terms of binary values. The sum-of-weights and repeated division by 2 methods are used to convert a decimal number to equivalent binary.

Sum-of-Weights

The sum-of-weights method used to convert binary numbers into their decimal equivalent is based on adding binary weights of the binary number bits. Converting back from the decimal number to the original binary number requires finding the highest weight included in the sum representing the decimal equivalent. A binary 1 is marked to represent the bit which contributed its weight in the sum representing the decimal equivalent. The weight is subtracted from the sum decimal equivalent. The next highest weight included in the sum term is found. A binary 1 is marked to represent the bit which contributed its weight in the sum term and the weight is subtracted from the sum term. This process is repeated until the sum term becomes equal to zero. The binary 1s and 0s represent the binary bits that contributed their weight and bits that did not contribute any weight respectively.

The process of determining binary equivalent of a decimal number 392 and 411 is illustrated in a tabular form.

Table 5: Converting Decimal to Binary using Sum-of-Weights Method

Sum Term	Highest Weight	Binary Number	Sum Term = Sum Term – Highest Weight
392	256	100000000	136
136	128	110000000	8
8	8	110001000	0

The sum-of-weights method requires mental arithmetic and is a quick way of converting small decimal numbers into binary. With practice large decimal numbers can be converted into binary equivalents.

Repeated Division-by-2

Repeated division-by-2 method allows decimal numbers of any magnitude to be converted into binary. In this method, the decimal number to be converted into its binary equivalent is repeatedly divided by 2. The divisor is selected as 2 because the decimal number is being converted into binary a base-2 number system. Repeated division method can be used to convert decimal number into any number system by repeated division by the base-number.

In the repeated-division method the decimal number to be converted is divided by the base number, in this particular case 2. A quotient value and a remainder value is generated, both values are noted done. The

remainder value in all subsequent divisions would be either a 0 or a 1. The quotient value obtained as a result of division by 2 is divided again by 2. The new quotient and remainder values are again noted down. In each step of the repeated division method the remainder values are noted down and the quotient values are repeatedly divided by the base number. The process of repeated division stops when the quotient value becomes zero. The remainders that have been noted in consecutive steps are written out to indicate the binary equivalent of the original decimal number.

Table 6: Converting Decimal to Binary using Repeated Division by 2 Method

Number	Quotient after division	Remainder after division
392	196	0
196	98	0
98	49	0
49	24	1
24	12	0
12	6	0
6	3	0
3	1	1
1	0	1

The process of determining the binary equivalent of a decimal number 392 is illustrated in a tabular form above. Reading the numbers in the remainder column from bottom to top 110001000 gives the binary equivalent of the decimal number 392.

SELF-ASSESSMENT EXERCISE

Explain with the aid of good examples, the different methods of converting binary numbers to decimal numbers.

4.0 CONCLUSION

The decimal number system has ten unique digits 0, 1, 2, 3… 9. Using these single digits, ten different values can be represented. Values greater than ten can be represented by using the same digits in different combinations. Binary indicates a base-2 number system having only two numbers 0 and 1. The binary digit 0 or 1 is known as a ‘Bit’.

5.0 SUMMARY

In this unit, we discussed decimal and binary number systems, manipulation of their fractions, binary arithmetic and conversion of

decimal to binary and vice versa. Hoping that you understood the topics discussed, you may now attempt the questions below.

6.0 TUTOR-MARKED ASSIGNMENT

1. Briefly explain these terms: decimal number system, binary number system.
2. Convert these decimal numbers to binary
   (a) 105 (b) 345 (c) 55
3. Convert these binary numbers to decimal
   (a) 10110.101 (b) 1111.111 (c) 110100100

7.0 REFERENCES/FURTHER READING

Comer, J. David (1994). *Digital Logic & State Machine Design*. Oxford University Press.

Dandamudi, S. (2003). *Fundamentals of Computer Organization and Design*. Springer.

Hayes, P. John (1993). *Introduction to Digital Logic Design*. Prentice Hall.

Hennessy, L. John & Patterson, A. David (2008). *Computer Organization & Design*. Morgan Kaufmann.

Holdsworth, Brian & Woods, Clive (2002). *Digital Logic Design*. Newnes.

Mano, M. Morris & Kime, Charles R. (2007). *Logic and Computer Design Fundamentals*. Prentice Hall. Pg 283.

Marcovitz Alan (2009). *Introduction to Logic Design*. McGraw-Hill.

Nelson, P Victor, *et al* (1995). *Digital Logic Circuit Analysis & Design*. Prentice Hall.

Palmer James & Periman David (1993). *Schaum's Outline of Introduction to Digital Systems*. McGraw-Hill.

Rafiquzzaman M. (2005). *Fundamentals of Digital Logic & Microcomputer Design*. Wiley – Interscience.

www.cs.siu.edu

www.educypedia.be/electronics

www.books.google.com

UNIT 2 TYPES OF NUMBER SYSTEMS II

CONTENTS

1.0 INTRODUCTION

In this unit we shall conclude with hexadecimal and octal number systems.

2.0 OBJECTIVES

At the end of this unit, you should be able to:

- explain the term hexadecimal number system
- count in hexadecimal
- convert binary to hexadecimal
- convert hexadecimal to binary
- convert decimal to hexadecimal
- convert hexadecimal to decimal
- explain hexadecimal addition and subtraction
- explain the term octal number system
- explain hexadecimal addition and subtraction.

3.0 MAIN CONTENT

3.1 Hexadecimal Number System

Representing even small number such as 6918 requires a long binary string (1101100000110) of 0s and 1s. Larger decimal numbers would require lengthier binary strings. Writing such long string is tedious and prone to errors.

The hexadecimal number system is a base 16 number system and therefore has 16 digits and is used primarily to represent binary strings in a compact manner. Hexadecimal number system is not used by a digital system. The hexadecimal number system is for our convenience to write binary strings in a short and concise form. Each hexadecimal number digit can represent a 4-bit binary number. The binary numbers and the hexadecimal equivalents are listed below:

Table 1: Hexadecimal Equivalents of Decimal and Binary Numbers

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

3.1.1 Counting in Hexadecimal Number System

Counting in hexadecimal is similar to the other number systems already discussed. The maximum value represented by a single hexadecimal digit is F which is equivalent to decimal 15. The next higher value decimal 16 is represented by a combination of two hexadecimal digits 10_{16} or 10 H. The subscript 16 indicates that the number is hexadecimal 10 and not decimal 10. Hexadecimal numbers are also identified by appending the character H after the number. The hexadecimal numbers for decimal numbers 16 to 39 are listed below in table 2:

Table 2: Counting using Hexadecimal Numbers

Decimal	Hexadecimal	Decimal	Hexadecimal	Decimal	Hexadecimal
16	10	24	18	32	20
17	11	25	19	33	21
18	12	26	1A	34	22
19	13	27	1B	35	23
20	14	28	1C	36	24
21	15	29	1D	37	25
22	16	30	1E	38	26
23	17	31	1F	39	27

3.1.2 Binary to Hexadecimal Conversion

Converting binary to hexadecimal is a very simple operation. The binary string is divided into small groups of 4-bits starting from the least significant bit. Each 4-bit binary group is replaced by its hexadecimal equivalent.

1101011010111001 0110 binary number 1101 0110 1011 1001 0110 Dividing into groups of 4-bits

D 6 B 9 6 Replacing each group by its hexadecimal equivalent

Thus, 1101011010111001 0110 is represented in hexadecimal by D6B96

Binary strings which cannot be exactly divided into a whole number of 4-bit groups are assumed to have 0's appended in the most significant bits to complete a group.

1101100000110 Binary Number
1 1011 0000 0110 Dividing into groups of 4-bits
0001 1011 0000 0110 Appending three 0s to complete the group
1 B 0 6 Replacing each group by its hexadecimal equivalent

3.1.3 Hexadecimal to Binary Conversion

Converting from Hexadecimal back to binary is also very simple. Each digit of the hexadecimal number is replaced by an equivalent binary string of 4-bits.

F D 1 3 hexadecimal number

1111 1101 0001 0011 Replacing each hexadecimal digit by its 4-bit binary equivalent.

3.1.4 Decimal to Hexadecimal Conversion

There are two methods to convert from decimal to hexadecimal. The first method is the indirect method and the second method is the repeated division method.

Indirect Method

A decimal number can be converted into its hexadecimal equivalent indirectly by first converting the decimal number into its binary equivalent and then converting the binary to Hexadecimal.

Repeated Division-by-16 Method

The repeated division method has been discussed earlier and used to convert decimal numbers to binary by repeatedly dividing the decimal number by 2. A decimal number can be directly converted into hexadecimal by using repeated division. The decimal number is continuously divided by 16 (base value of the hexadecimal number system).

The conversion of decimal 2096 to hexadecimal using the repeated division-by-16 method is illustrated in Table 3. The hexadecimal equivalent of 2096_{10} is 830_{16}.

Table 3: Hexadecimal Equivalent of Decimal Numbers Using Repeated Division

Number	Quotient after division	Remainder after division
2096	131	0
131	8	3
8	0	8

3.1.5 Hexadecimal to Decimal Conversion

Converting hexadecimal numbers to decimal is done using two methods. The first method is the indirect method and the second method is the sum-of-weights method.

Indirect Method

The indirect method of converting hexadecimal number to decimal number is to first convert hexadecimal number to binary and then binary to decimal.

Sum-of-Weights Method

A hexadecimal number can be directly converted into decimal by using the sum of weights method. The conversion steps using the sum-of-weights method are shown.

CA02 hexadecimal number
$C \times 16^3 + A \times 16^2 + 0 \times 16^1 + 2 \times 16^0$ Writing the number in an expression
$(C \times 4096) + (A \times 256) + (0 \times 16) + (2 \times 1)$
$(12 \times 4096) + (10 \times 256) + (0 \times 16) + (2 \times 1)$ Replacing hexadecimal values with
decimal equivalents
49152 + 2560 + 0 + 2 Summing the weights
51714 Decimal equivalent

3.1.6 Hexadecimal Addition and Subtraction

Numbers represented in hexadecimal can be added and subtracted directly without having to convert them into decimal or binary equivalents. The rules of addition and subtraction that are used to add and subtract numbers in decimal or binary number systems apply to hexadecimal addition and subtraction. Hexadecimal addition and subtractions allows large binary numbers to be quickly added and subtracted.

Hexadecimal Addition

Carry	1
Number 1	2 AC 6
Number 2	9 2 B 5
Sum	B D 7 B

Hexadecimal Subtraction

Borrow	1 1 1
Number 1	9 2 B 5
Number 2	2 A C 6
Difference	6 7 E F

3.2 Octal Number System

Octal number system also provides a convenient way to represent long string of binary numbers. The octal number is a base 8 number system with digits ranging from 0 to 7. Octal number system was prevalent in earlier digital systems and is not used in modern digital systems especially when the hexadecimal number is available. Each octal number digit can represent a 3-bit binary number. The binary numbers and the octal equivalents are listed below

Table 4: Octal Equivalents of Decimal and Binary Numbers

Decimal	Binary	Octal
0	000	0
1	001	1
2	010	2
3	011	3
4	100	4
5	101	5
6	110	6
7	111	7

3.2.5 Counting in Octal Number System

Counting in octal is similar to counting in any other number system. The maximum value represented by a single octal digit is 7. For representing larger values a combination of two or more octal digits has to be used. Thus, decimal 8 is represented by a combination of10_8. The subscript 8 indicates the number is octal 10 and not decimal ten. The octal numbers for decimal numbers 8 to 30 are listed below:

Table 5: Counting using Octal Numbers

Decimal	Octal	Decimal	Octal	Decimal	Octal
8	10	16	20	24	30
9	11	17	21	25	31
10	12	18	22	26	32
11	13	19	23	27	33
12	14	20	24	28	34
13	15	21	25	29	35
14	16	22	26	30	36
15	17	23	27	31	37

3.2.6 Binary to Octal Conversion

Converting binary to octal is a very simple. The binary string is divided into small groups of 3-bits starting from the least significant bit. Each 3-bit binary group is replaced by its octal equivalent.

11101011010111001010 Binary number
111 010 110 101 110 010 110 Dividing into groups of 3-bits
7 2 6 5 6 2 6 Replacing each group by its octal equivalent

Thus, 11101011010111001010 is represented in octal by 7265626

Binary strings which cannot be exactly divided into a whole number of 3-bit groups are assumed to have 0's appended in the most significant bits to complete a group.

1101100000110 Binary number
1 101 100 000 110 Dividing into groups of 3-bits
001 101 100 000 110 Appending three 0s to complete the group
1 5 4 0 6 Replacing each group by its octal equivalent

3.2.7 Octal to Binary Conversion

Converting from octal back to binary is also very simple. Each digit of the octal number is replaced by an equivalent binary string of 3-bits.
1 7 2 6 Octal number
001 111 010 110 Replacing each octal digit by its 3-bit binary equivalent.

3.2.8 Decimal to Octal Conversion

There are two methods to convert from decimal to octal. The first method is the
Indirect Method and the second method is the repeated division method.

Indirect Method

A decimal number can be converted into its octal equivalent indirectly by first converting the decimal number into its binary equivalent and then converting the binary to octal.

Repeated Division-by-8 Method

The repeated division method has been discussed earlier and used to convert decimal numbers to binary and hexadecimal by repeatedly dividing the decimal number by 2 and 16 respectively. A decimal number can be directly converted into octal by using repeated division. The decimal number is continuously divided by 8 (base value of the Octal number system).

The conversion of decimal 2075 to octal using the repeated division-by-8 method is illustrated in Table 6. The octal equivalent of 2075_{10} is 4033_8.

Table 6: Octal Equivalent of Decimal Numbers Using Repeated Division

Number	Quotient after Division	Remainder after Division
2075	259	3
259	32	3
32	4	0
4	0	4

3.2.5 Octal to Decimal Conversion

Converting octal numbers to decimal is done using two methods. The first method is the indirect method and the second method is the sum-of-weights method.

Indirect Method

The indirect method of converting octal number to decimal number is to first convert octal number to binary and then binary to decimal.

Sum-of-Weights Method

An octal number can be directly converted into decimal by using the sum of weights method. The conversion steps using the sum-of-weights method are shown.

4033 octal number
$4 \times 8^3 + 0 \times 8^2 + 3 \times 8^1 + 3 \times 8^0$ Writing the number in an expression
$(4 \times 512) + (0 \times 64) + (3 \times 8) + (3 \times 1)$
$2048 + 0 + 24 + 3$ Summing the weights
2075 Decimal equivalent

3.2.6 Octal Addition and Subtraction

Numbers represented in octal can be added and subtracted directly without having to convert them into decimal or binary equivalents. The rules of addition and subtraction that are used to add and subtract numbers in decimal or binary number systems apply to octal addition and subtraction. Octal addition and subtractions allows large binary numbers to be quickly added and subtracted.

1. Octal Addition

Carry 1
Number 1 7 6 0 2
Number 2 5 7 7 1
Sum 1 5 5 7 3

2. Octal Subtraction

Borrow 1 1
Number 1 7 6 0 2
Number 2 5 7 7 1
Difference 1 6 1 1

SELF-ASSESSMENT EXERCISE

Explain how you can convert hexadecimal and octal numbers to binary and decimal numbers and vice versa.

4.0 CONCLUSION

In this unit we talked about hexadecimal and octal number systems, counting in hexadecimal and octal, hexadecimal & octal arithmetic, conversion of hexadecimal to binary and vice versa, hexadecimal to decimal and vice versa.

5.0 SUMMARY

In this unit we talked about hexadecimal and octal number systems, counting in hexadecimal and octal, hexadecimal and octal arithmetic, conversion of hexadecimal to binary and vice versa, hexadecimal to decimal and vice versa.

6.0 TUTOR-MARKED ASSIGNMENT

1. Briefly explain these terms: hexadecimal number system, octal number system.
2. Convert these decimal numbers to hexadecimal and octal and back to decimal 105 (b) 345 (c) 55.
3. Convert these binary numbers to hexadecimal and octal and back to binary
   (b) 10110.101 (b) 1111.111 (c) 110100100.

7.0 REFERENCES/FURTHER READING

Comer, J. David (1994). *Digital Logic & State Machine Design*. Oxford University Press.

Dandamudi, S. (2003). *Fundamentals of Computer Organization and Design*. Springer.

Hayes, P. John (1993). *Introduction to Digital Logic Design*. Prentice Hall.

Hennessy, L. John & Patterson, A. David (2008). *Computer Organization & Design*. Morgan Kaufmann.

Holdsworth, Brian & Woods, Clive (2002). *Digital Logic Design*. Newnes.

Mano, M. Morris; Kime, Charles R. (2007). *Logic and Computer Design Fundamentals*. Prentice Hall. Pg 283.

Marcovitz, Alan (2009). *Introduction to Logic Design*. McGraw-Hill.

Nelson, P. Victor *et al* (1995). *Digital Logic Circuit Analysis & Design*. Prentice Hall.

Palmer, James & Periman, David (1993). *Schaum's Outline of Introduction to Digital Systems*. McGraw-Hill.

Rafiquzzaman, M. (2005). *Fundamentals of Digital Logic & Microcomputer Design*. Wiley – Interscience.

www.cs.siu.edu

www.educypedia.be/electronics

www.books.google.com

UNIT 3 CODES

CONTENTS

1.0 INTRODUCTION

We have different types of code a few are briefly discussed below they include: Excess code, BCD code, Gray code, alphanumeric code, ASCII code, Extended ASCII code.

2.0 OBJECTIVES

At the end of this unit, you should be able to:

- define code
- explain the various types of codes.

3.0 MAIN CONTENT

3.1 Codes

A code in computer parlance is a generic term for program instructions, used in two general senses. The first sense refers to human-readable source code, which include the instructions written by the programmer in a programming language. The second refers to executable machine code, which include the instructions of a program that were converted from source code to instructions that the computer can understand.

3.1.1 The Excess Code

Consider the decimal number range +7 to -8. These positive and negative decimal numbers can be represented by the 2's complement representation. The magnitude of positive and negative numbers cannot be easily compared as the positive and negative numbers represented in 2's complement form are not represented on a uniformly increasing scale.

The decimal number range +7 to -8 is represented using an excess-8 code that assigns 0000 to -8 the lowest number in the range and 1111 to +7 the highest number in the range. Excess-8 code is obtained by adding a number to the lowest number -8 in the range such that the result is zero. The number is 8. The number 8 is added to all the remaining decimal numbers from -7 up to the highest number +7. The excess-8 represented is presented below.

Table 1: Excess-8 Code Representation of Decimal Numbers in the Range 7 to 8

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

3.1.2 BCD Code

Binary Coded Decimal (BCD) code is used to represent decimal digits in binary. BCD code is a 4-bit binary code; the first 10 combinations represent the decimal digits 0 to 9. The remaining six 4-bit combinations 1010, 1011, 1100, 1101, 1110 and 1111 are considered to be invalid and do not exist.

The BCD code representing the decimal digits 0 to 9 is shown in the Table below:

Table 2: BCD Representation of Decimal Digits 0 to 9

Decimal	BCD	Decimal	BCD
0	0000	5	0101
1	0001	6	0110
2	0010	7	0111
3	0011	8	1000
4	0100	9	1001

To write 17, two BCD code for 1 and 7 are used 0001 and 0111. The two digits are considered to be separate. The conventional method of representing decimal 17 using unsigned binary is 10001. A telephone keypad having the digits 0 to 9 generates BCD codes for the keys pressed.

Most digital systems display a count value or the time in decimal on 7-segment LED display panels. Since the numbers displayed are in decimal, therefore the BCD Code is used to display the decimal numbers. Consider a 2-digit 7-segment display that can display a count value from 0 to 99. To display the two decimal digits two separate BCD codes are applied at the two 7-segment display circuit inputs.

BCD Addition

Multi-digit BCD numbers can be added together.

```
23      0010 0011
45      0100 0101
-----------------
68      0110 1000
```

The two 2-digit BCD numbers are added and generate a result in BCD. In the example, the least significant digits 3 and 5 add up to 8 which is a valid BCD representation. Similarly, the most significant digits 2 and 4 add up to 6 which also is a valid BCD representation.

Consider the next example where the least significant numbers add up to a number greater than 9 for which there is no valid BCD code

```
23      0010 0011
48      0100 1000
-----------------
71      0110  1011
```

For BCD numbers that add up to an invalid BCD number or generate a carry the number 6(0110) is added to the invalid number. If a carry results, it is added to the next most significant digit. Thus:

0011
1000
1011 11 is generated which is an invalid BCD number
0110 6 is added
1 0001

A carry is generated which is added to the result of the next most significant digits
1
0110
0111

The answer is 0111 0001

3.1.3 Gray Code

The Gray code does not have any weights assigned to its bit positions. The Gray code is not a positional code. The Gray code is different from the unsigned binary code as successive values of Gray code differ by only one bit. Table 3 shows the Gray code representation of decimal numbers 0 to 9.

Table 3: Gray Code Representation of Decimal Values

Decimal	Gray	Binary
0	0000	0000
1	000**1**	0001
2	00**1**1	0010
3	001**0**	0011
4	0**1**10	0100
5	011**1**	0101
6	01**0**1	0110
7	010**0**	0111
8	**1**100	1000
9	110**1**	1001

The bits in **bold** change in successive values of Gray code representation.

3.1.4 Alphanumeric Code

All the representation studied so far allow decimal numbers to be represented in binary. Digital systems also process text information as in editing of documents. Thus, each letter of the alphabet, upper case and lower case, along with the punctuation marks should have a representation. Numbers are also written in textual form such as 2nd

June 2003. The ASCII code is a universally accepted code that allows 128 characters and symbols to be represented.

3.1.5 ASCII Code

The ASCII code (American Standard Code for Information Interchange) is a 7-bit code representing 128 unique codes which represent the alphabet characters A to Z in lower case and upper case, the decimal numbers 0 to 9, punctuation marks and control characters.

ASCII codes 011 0000 (30h) to 011 1001 (39h) represents numbers 0 to 9

ASCII codes 1100001 (61h) to 1111010(7Ah) represent lower case alphabets a to z

ASCII codes 100 0001 (41h) to 101 1010 (5Ah) represent upper case alphabets A to Z

ASCII codes 000 0000 (0h) to 001 1111 (1Fh) represent the 32 Control characters.

3.1.6 Extended ASCII Code

The 7-bit ASCII code only has 128 unique codes which are not enough to represent some graphical characters displayed on computer screens. An 8-bit code extended ASCII code gives 256 unique codes. The extended 128 unique codes represent graphic symbols which have become an unofficial standard as vendors use their own interpretation of these graphic codes.

4.0 CONCLUSION

Excess-8 code is obtained by adding a number to the lowest number -8 in the range such that the result is zero. Binary Coded Decimal (BCD) code is used to represent decimal digits in binary. BCD code is a 4-bit binary code; the first 10 combinations represent the decimal digits 0 to 9.

The Gray code does not have any weights assigned to its bit positions, is not a positional code and is different from the unsigned binary code as successive values of Gray code differ by only one bit.

The ASCII Code (American Standard Code for Information Interchange) is a 7-bit code representing 128 unique codes which

represent the alphabet characters A to Z in lower case and upper case, the decimal numbers 0 to 9, punctuation marks and control characters.

5.0 SUMMARY

In this unit, we talked about various codes, their characteristics and how they can be used.

6.0 TUTOR-MARKED ASSIGNMENT

Write short notes on the following:

(a) Excess code
(b) BCD code
(c) Gray code
(d) Alphanumeric code
(e) ASCII code
(f) Extended ASCII code.

7.0 REFERENCES/FURTHER READING

Comer, J. David (1994). *Digital Logic & State Machine Design*. Oxford University Press.

Dandamudi, S. (2003). *Fundamentals of Computer Organization and Design*. Springer.

Hayes, P. John (1993). *Introduction to Digital Logic Design*. Prentice Hall.

Hennessy, L. John & Patterson A. David (2008). *Computer Organization & Design*. Morgan Kaufmann.

Holdsworth, Brian & Woods Clive (2002). *Digital Logic Design*. Newnes.

Mano, M. Morris; Kime, Charles R. (2007). *Logic and Computer Design Fundamentals*. Prentice Hall. Pg 283.

Marcovitz, Alan (2009). *Introduction to Logic Design*. McGraw-Hill.

Nelson, P. Victor; *et al.* (1995). *Digital Logic Circuit Analysis & Design*. Prentice Hall.

Palmer, James & Periman, David (1993). *Schaum's Outline of Introduction to Digital Systems*. McGraw-Hill.

Rafiquzzaman, M. (2005). *Fundamentals of Digital Logic & Microcomputer Design*. Wiley – Interscience.

www.cs.siu.edu

www.educypedia.be/electronics

www.books.google.com

MODULE 2 COMBINATIONAL LOGIC DESIGN & APPLICATIONS

UNIT 1 ANALYSIS AND DESIGN OF A COMBINATIONAL LOGIC CIRCUIT

CONTENTS

1.0 INTRODUCTION

Digital logic circuits can be classified into two types: Combinational and Sequential logic circuits.

Combinational logic circuit is designed using logic gates in which applications of inputs generate the outputs at any time. A combinational circuit does not require memory so the output depends only on the current inputs. Combinational circuits help in reducing design complexity and reduce the chip count in a circuit.

2.0 OBJECTIVES

At the end of this unit, you should be able to:

- analyse a combinational logic circuit
- design a combinational logic circuit.

3.0 MAIN CONTENT

3.1 Analysis of a Combinational Logic Circuit

A combinational logic circuit can be analysed by:

- identifying the number of inputs and outputs
- expressing the output functions in terms of the inputs, and
- determining the truth table for the logic diagram.

3.2 Design of a Combinational Logic Circuit

A combinational circuit can be designed using three steps as follows:

- determine the inputs and the outputs from the problem definition and then derive the truth table
- use k-maps to minimise the number of inputs in order to express the outputs - this reduces the number of gates and thus the implementation cost
- draw the logic diagrams.

4.0 CONCLUSION

Combinational logic circuit is designed using logic gates in which applications of inputs generate the outputs at any time. It does not require memory so the output depends only on the current inputs.

5.0 SUMMARY

In this unit, we discussed combinational logic circuit, its analysis and how it can be designed.

6.0 TUTOR-MARKED ASSIGNMENT

Briefly explain the term combinational logic circuit, how it is analysed and designed.

7.0 REFERENCES/FURTHER READING

Comer, J. David (1994). *Digital Logic & State Machine Design*. Oxford University Press.

Dandamudi, S. (2003). *Fundamentals of Computer Organization and Design*. Springer.

Hayes, P. John (1993). *Introduction to Digital Logic Design*. Prentice Hall.

Hennessy, L. John & Patterson, A. David (2008). *Computer Organization & Design*. Morgan Kaufmann.

Holdsworth, Brian & Woods, Clive (2002). *Digital Logic Design*. Newnes.

Mano, M. Morris & Kime, Charles, R. (2007). *Logic and Computer Design Fundamentals*. Prentice Hall. Pg 283.

Marcovitz, Alan (2009). *Introduction to Logic Design*. McGraw-Hill.

Neamen, Donald (2009). *Microelectronics: Circuit Analysis & Design*. McGraw-Hill.

Nelson, P. Victor; *et al.* (1995). *Digital Logic Circuit Analysis & Design*. Prentice Hall.

Palmer, James & Periman, David (1993). *Schaum's Outline of Introduction to Digital Systems*. McGraw-Hill.

Patterson, David & Hennessy, John (1993). *Computer Organization & Design: The Hardware/Software Interface*. Morgan Kaufmann Publishers.

Rafiquzzaman, M. (2005). *Fundamentals of Digital Logic & Microcomputer Design*. Wiley – Interscience.

Roth, H. Charles Jr & Kinney, L. Larry (2009). *Fundamentals of Logic Design*.

Tanenbaum, Andrew S. (1998). *Structured Computer Organization.* Prentice Hall.

Tocci, J. Ronald & Widmer S. Neal (1994). *Digital Systems: Principles and Applications.*

Tokheium, Roger (1994). *Schaum's Outline of Digital Principles.* McGraw-Hill.

Wakerly, F. John (2005). *Digital Design: Principles & Practices Package*. Prentice Hall.

UNIT 2 TYPICAL COMBINATIONAL LOGIC CIRCUIT I

CONTENTS

1.0 INTRODUCTION

In this unit we shall discuss Adders, types and their implementation.

2.0 OBJECTIVES

At the end of this unit, you should be able to:

- name the two types of adders we have
- discuss the half-adder, its function table, sum and carry out Boolean expression
- discuss the full-adder, its function table, sum and carry out Boolean expression
- form a full-adder using half-adders
- discuss about the parallel binary adder.

3.0 MAIN CONTENT

3.1 Typical Combinational Logic Circuit I

3.1.1 Adders

We have two types of adder: Half-adder and Full-adder

3.1.1.1 Half-adder

A single bit binary adder circuit basically adds two bits. The output of the single bit adder circuit generates a sum bit. An adder circuit that only has two bit input representing the two single bit numbers A and B and does not have the carry bit input from the least significant digit is regarded as a half-adder. The block diagram below represents a half-adder.

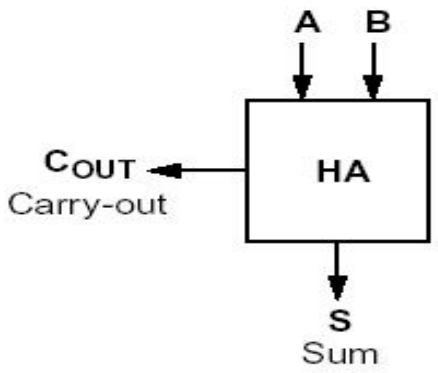


Fig. 1: Block Representation of Half-Adder

A half-adder can be fully described in terms of its function table and the circuit implementation.

3.1.1.2 Half-Adder Function Table

The half-adder has a 2-bit input and a 2-bit output. The function table of the half-adder has two input columns representing the two single bit numbers A and B. The function table also has two output columns representing the sum bit and carry out bit.

A	B	Sum	C_{out}
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

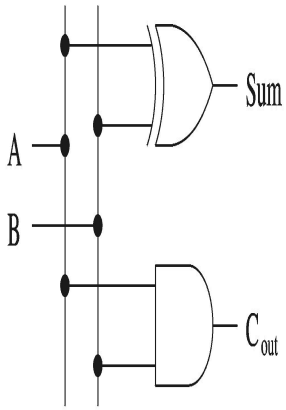


Fig. 2: Half-Adder Function Table and Circuit Implementation

3.1.1.3 Half-adder Sum & Carry Out Boolean Expression

The sum and carry out expressions of the half-adder can be determined from the function table. The half-adder sum and carry out outputs are defined by the expressions:

Sum = $\overline{A}B + A\overline{B} = A \oplus B$
Carry out = AB

3.1.1.4 Full-Adder

An adder circuit which has three inputs, one representing single bit number A, the other representing the single bit number B and the third bit represents the single bit carry is referred to as a full-adder. The single bit binary adder has two bit output. One bit represents the Sum between numbers A and B. The other bit represents the carry bit generated due to addition. The diagram below represents the block diagram of a full-adder.

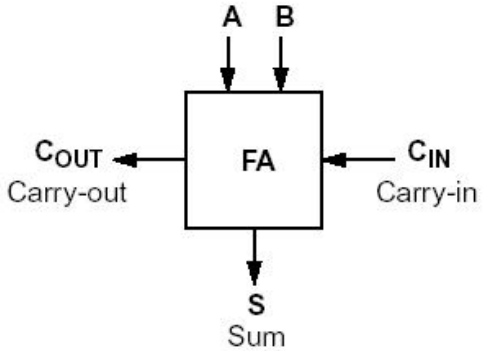


Fig. 3: Block Representation of a Full-Adder

3.1.1.5 Full-adder Function Table

The full-adder has a 3-bit input and a 2-bit output. The function table of the full-adder has three input columns representing the two single bit numbers A, B and the carry in bit. The function table also has two output columns representing the sum bit and carry out bit.

A	B	C_{in}	Sum	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

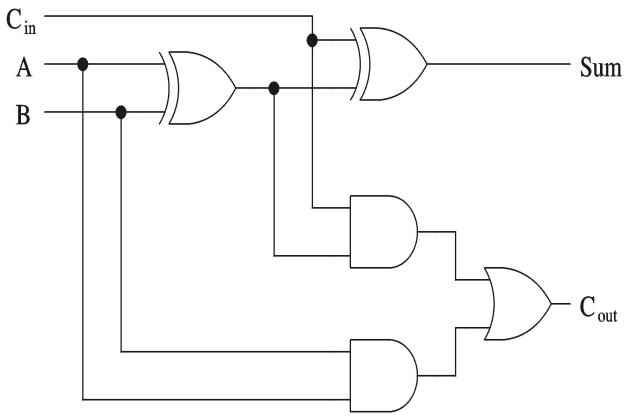


Fig. 4: Full-adder Function Table and Circuit Implementation

3.1.1.6 Full-adder Sum & Carry out Boolean Expression

The sum and carry out expressions of the full-adder can be determined from the function table. The full-adder sum and carry out outputs are defined by the expressions:

$Sum = \overline{A}\,\overline{B}\,C + \overline{A}B\overline{C} + A\overline{B}\overline{C} + ABC$
$Sum = \overline{A}(B\overline{C} + \overline{B}C) + A(\overline{B}\overline{C} + BC)$
$Sum = A(B \oplus C) + A(\overline{B \oplus C})$
$Sum = A \oplus B \oplus C$
$CarryOut = \overline{A}BC + A\overline{B}C + AB\overline{C} + ABC$
$CarryOut = C(\overline{A}B + A\overline{B}) + AB(\overline{C} + C)$
$CarryOut = C(A \oplus B) + AB$

3.1.1.7 Forming a Full-Adder using Half-adders

A 1-bit full-adder can be implemented by combining together two half-adders.

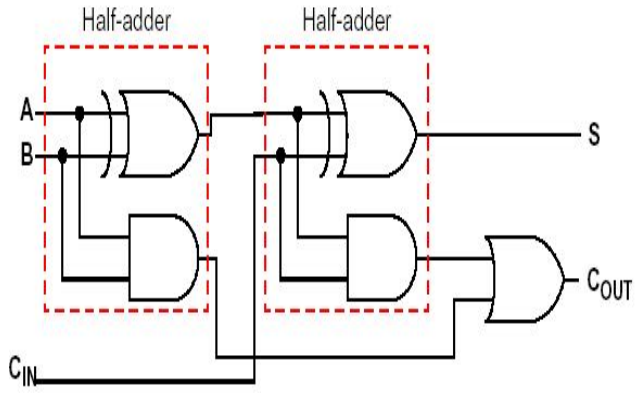


Fig. 5: Implementing a Full-Adder using Two Half-Adders

The sum output of the first half-adder is (A ⊕ B)
The carry out of the first half-adder is AB
The sum output of the second half-adder is (A ⊕ B) ⊕ C_{in} = (A ⊕ B ⊕ C_{in})
The carry out of the second half-adder is (A ⊕ B)C_{in}
The output of the OR gate is in AB + (A ⊕ B)C_{in}

3.1.1.8 Parallel Binary Adders

Single bit full or half-adders do not perform any useful function. To add two 4-bit numbers a 4-bit adder is required. Four single bit full-adders are connected together to form a 4-bit parallel adder capable of adding two 4-bit binary numbers. A 4-bit binary adder can be formed with four full-adders as follows:

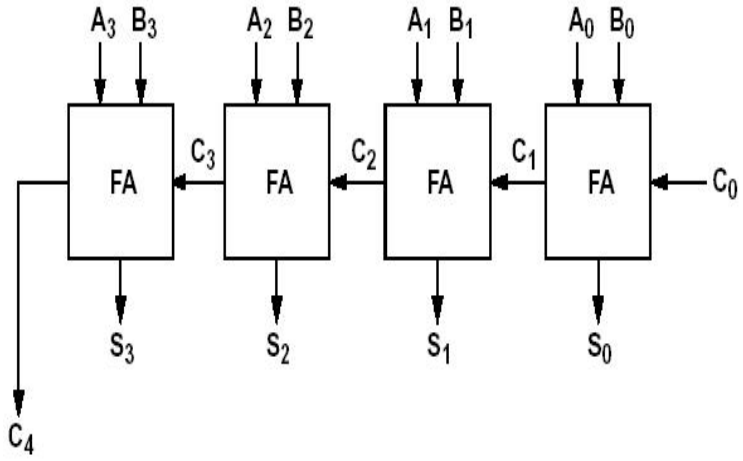
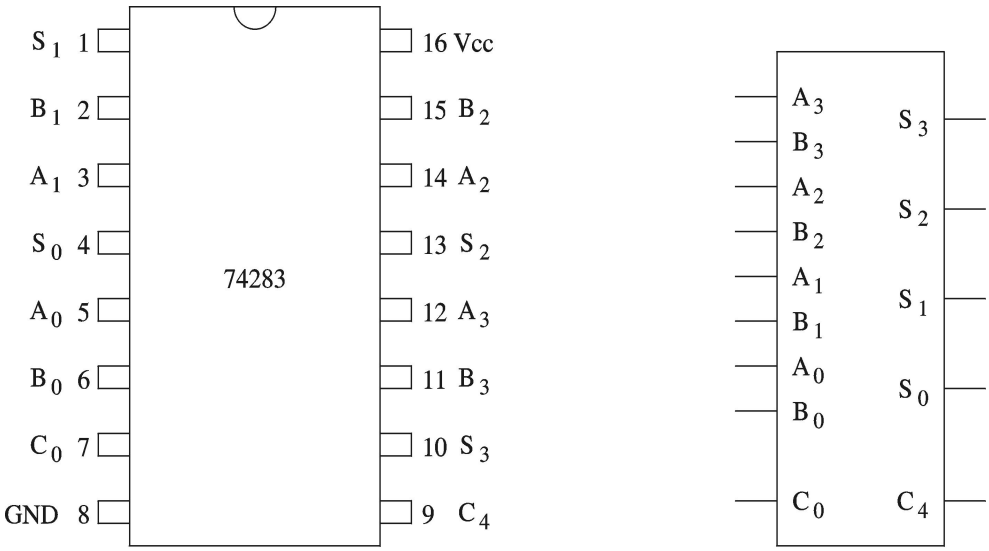


Fig. 6: A 4-Bit Binary Adder

The connection diagram and logic symbols are shown below:



Connection Diagram **Logic Symbol**

Fig.7: Connection Diagram and Logic Symbol of a 4-Bit Binary Adder

SELF-ASSESSMENT EXERCISE

Explain with the aid of diagrams how you can form a full-adder from half-adders.

4.0 CONCLUSION

An adder circuit that only has two bit input representing the two single bit numbers A and B and does not have the carry bit input from the least significant digit is regarded as a half-adder. An adder circuit which has three inputs, one representing single bit number A, the other representing the single bit number B and the third bit represents the single bit carry is referred to as a full-adder.

5.0 SUMMARY

In this unit we talked about types of adder, half-adder, half-adder function table, half-adder sum and carry out Boolean expression, full-adder, full-adder function table, full-adder sum and carry out Boolean expression.

6.0 TUTOR-MARKED ASSIGNMENT

Write short notes with diagrams where necessary on the following:

1. half-adder, its function table and sum & carry out Boolean expression.
2. full-adder, its function table and sum & carry out Boolean expression.

7.0 REFERENCES/FURTHER READING

Comer, J. David (1994). *Digital Logic & State Machine Design*. Oxford University Press.

Dandamudi, S. (2003). *Fundamentals of Computer Organization and Design*. Springer.

Hayes, P. John (1993). *Introduction to Digital Logic Design*. Prentice Hall.

Hennessy, L. John & Patterson, A. David (2008). *Computer Organization & Design*. Morgan Kaufmann.

Holdsworth, Brian & Woods, Clive (2002). *Digital Logic Design*. Newnes.

Mano, M. Morris & Kime, Charles, R. (2007). *Logic and Computer Design Fundamentals*. Prentice Hall. Pg 283.

Marcovitz, Alan (2009). *Introduction to Logic Design*. McGraw-Hill.

Neamen, Donald (2009). *Microelectronics: Circuit Analysis & Design*. McGraw-Hill.

Nelson, P. Victor *et al.* (1995). *Digital Logic Circuit Analysis & Design*. Prentice Hall.

Palmer, James & Periman, David (1993). *Schaum's Outline of Introduction to Digital Systems*. McGraw-Hill.

Patterson, David & Hennessy, John (1993). *Computer Organization & Design: The Hardware/Software Interface*. Morgan Kaufmann Publishers.

Rafiquzzaman, M. (2005). *Fundamentals of Digital Logic & Microcomputer Design*. Wiley – Interscience.

Roth, H. Charles Jr & Kinney, L. Larry (2009). *Fundamentals of Logic Design*.

Tanenbaum, Andrew S. (1998). *Structured Computer Organization.* Prentice Hall.

Tocci, J. Ronald & Widmer, S. Neal (1994). *Digital Systems: Principles and Applications.*

Tokheium, Roger (1994). *Schaum's Outline of Digital Principles*. McGraw-Hill.

Wakerly, F. John (2005). *Digital Design: Principles & Practices Package*. Prentice Hall.

UNIT 3 TYPICAL COMBINATIONAL LOGIC CIRCUIT II

CONTENTS

1.0 INTRODUCTION

In this unit we shall discuss multiplexers, demultiplexers and their applications in our everyday live.

2.0 OBJECTIVES

At the end of this unit, you should be able to:

- state what a multiplexer is
- state how a multiplexer is designed
- discuss the applications of multiplexers
- state what a demultiplexer is
- discuss the applications of demultiplexers.

3.0 MAIN CONTENT

3.1 Typical Combinational Logic Circuit II

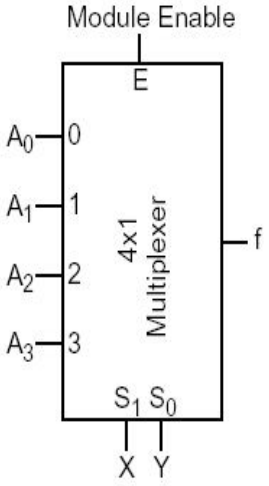
3.1.1 Multiplexers

Multiplexer is a digital switch that has several inputs and a single output. The Multiplexer also has select inputs that allow any one of the multiple inputs can be selected to be connected to the output. Multiplexers are also known as Data Selectors. The main use of the

Multiplexer is to select data from multiple sources and to route it to a single destination. In a computer, the ALU combinational circuit has two inputs to allow arithmetic operations to be performed on two quantities. The two quantities are usually stored in different set of registers.

The inputs of the two multiplexers are connected to the output of each of the multiple registers.

The outputs of the two multiplexers are connected to the two inputs of the ALUs. The multiplexers are used to route the contents of any two registers to the ALU inputs.



		Inputs				Output
X	Y	A_3	A_2	A_1	A_0	F
0	0	X	X	X	0	A_0=0
0	1	X	X	0	X	A_1=0
1	0	X	0	X	X	A_2=0
1	1	0	X	X	X	A_3=0
0	0	X	X	X	1	A_0=1
0	1	X	X	1	X	A_1=1
1	0	X	1	X	X	A_2=1
1	1	1	X	X	X	A_3=1

Fig. 1: A 4 X 1 Multiplexers and its Truth Table

4-data input multiplexer selects one of many inputs to be directed to an output.

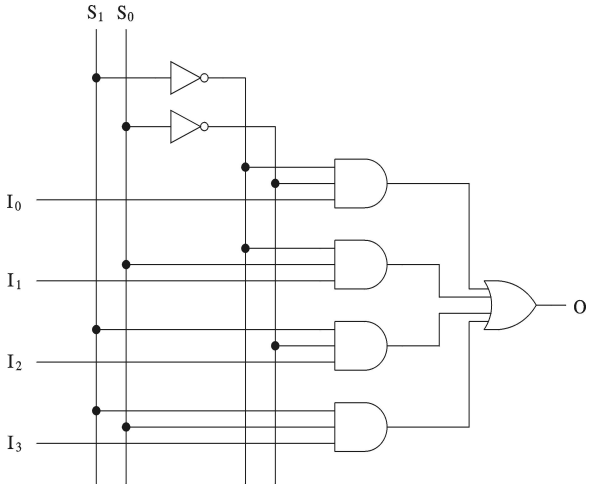


Fig. 2: 4-Data Input MUX Implementation

3.1.1.1 Using Pass Gate

The 4 x 1 MUX can be implemented with pass gates as follows:

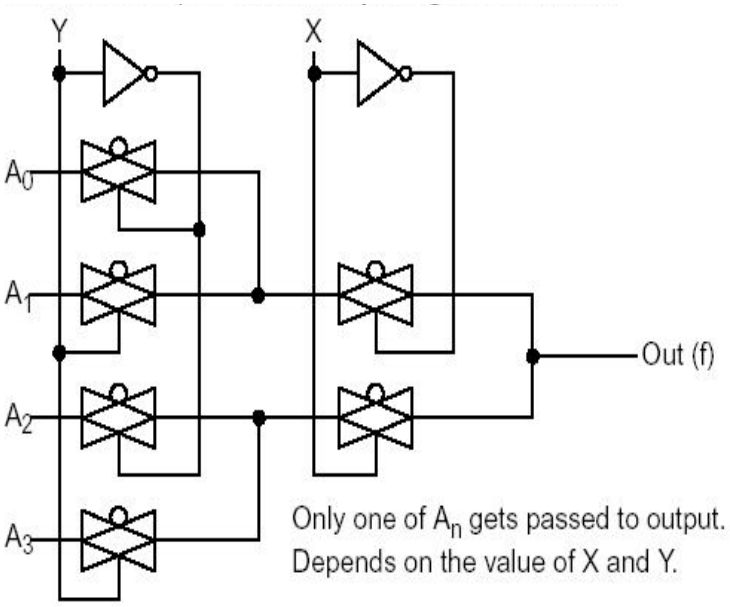


Fig. 3: Implementation of the 4x1 MUX with Pass Gates

Multiplexers can be efficiently implemented using the majority function and even- parity function.

Table 1: Majority Function

Original truth table

A	B	C	F_1
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

New truth table

A	B	F_1
0	0	0
0	1	C
1	0	C
1	1	1

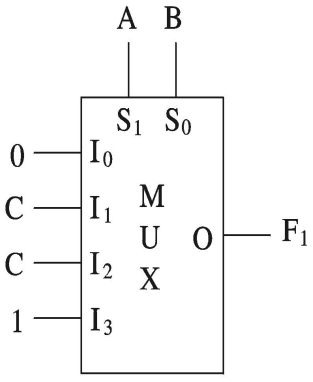


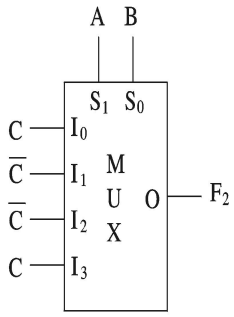
Table 2: Even-Parity Function

Original truth table

A	B	C	F_1
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

New truth table

A	B	F_1
0	0	C
0	1	$\overline{C}$
1	0	$\overline{C}$
1	1	C



3.1.1.2 Design with Multiplexers

Any Boolean function can be implemented by setting the inputs corresponding to the function and the selectors as the variables as shown below:

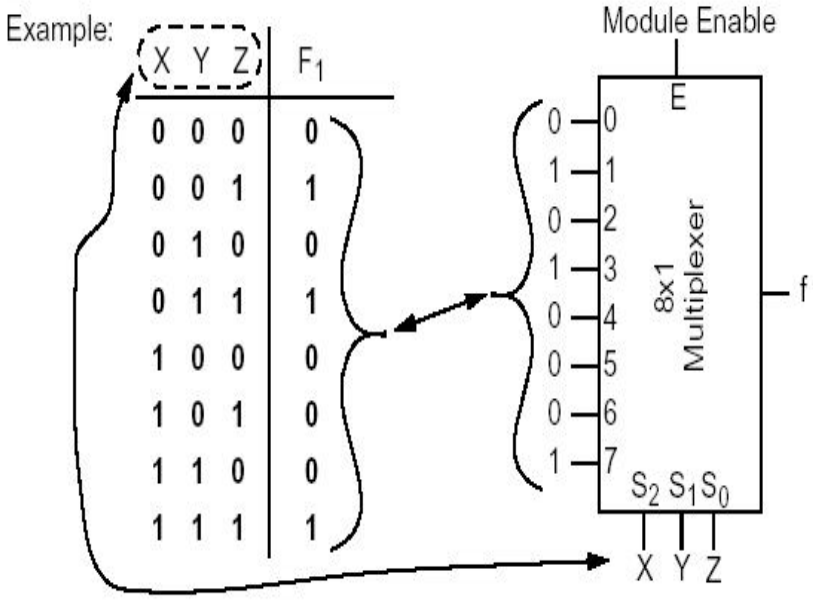


Fig. 4: Design with Multiplexers

3.1.1.4 Applications of Multiplexers

Multiplexers are used in a wide variety of applications. Their primary use is to route data from multiple sources to a single destination. Other than its use as a data router, they are used as a parallel to serial converter, logic function generator and also for operation sequencing.

Data Routing

A two - digit 7-Segment display uses two 7-Segments Display digits connected to two BCD to 7-Segment display circuits. To display the number 29 the BCD number 0010 representing the MSD is applied at the inputs of the BCD to 7-Segment display circuit connected to the MSD 7-Segment Display Digit. Similarly, the BCD input 1001 representing the numbers 9 is applied at the inputs of the LSD display circuit. The circuit uses two BCD to 7-Segment decoder circuits to decode each of the two BCD inputs to the respective 7-Segment display outputs. The display circuit can be implemented using a single CD to 7-Segment IC and a multiplexer.

Parallel to Series Conversion

In a digital system, binary data is used and represented in parallel. Parallel data is a set of multiple bits. For example, a nibble is a parallel set of 4-bits, a byte is a parallel set of 8 bits. When two binary numbers are added, the two numbers are represented in parallel and the parallel adder works and generates a sum term which is also in parallel.

Transmission of information to remote locations through a piece of wire requires that the parallel information (data) be converted into serial form. In a serial data representation, data is represented by a sequence of single bits. An 8-bit parallel data can be transmitted through a single piece of wire 1-bit at a time. Transmitting 8-bits simultaneously (in parallel form) requires 8 separate wires for the 8-bits. Laying of 8 wires across two remote locations for data transfer is expensive and is therefore not practical. All communication systems set up across remote locations use serial transmission.

An 8-bit parallel data can be converted into serial data by using an 8-to-1 multiplexer such as 74X151 which has 8 inputs and a single output. The 8-bit data which is to be transmitted serially is applied at the 8 inputs I0-7 of the multiplexer. A three bit counter which counts from 0 to 7 is connected to the three select inputs S0, S1 and S2. The counter is connected to a clock which sends a clock pulse to the counter every 1 millisecond. Initially, the counter is reset to 000, the I0 input is selected and the data at input I0 is routed to the output of the multiplexer. On receiving the clock signal after 1 millisecond the counter increments its count from 000 to 001 which selects I1 input of the multiplexer and routes the data present at the input to the output. Similarly, at the next clock pulse the counter increments to 010, selecting I2 input and routing the data to the output. Thus, after 8 milliseconds the parallel data is routed to the output 1-bit at a time. The output of the multiplexer is connected to the wire through which the serial data is transmitted.

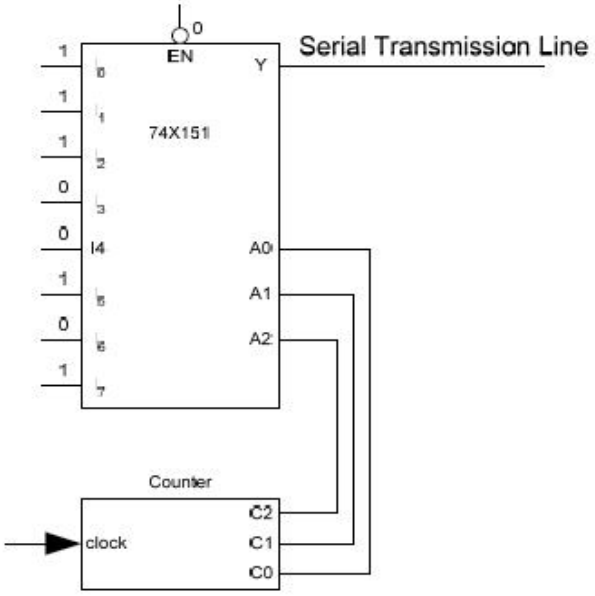
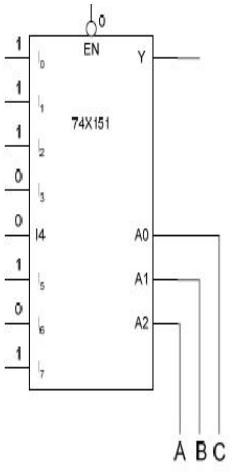


Fig. 5: Serial to Parallel Conversion

Logic Function Generator

Multiplexers can be used to implement a logic function directly from the function table without the need for simplification. The select inputs of the multiplexer are used as the function variables. The inputs of the multiplexer are connected to logic 1 and 0 to represent the missing and available terms. The three variable function table and its 8-to-1 multiplexer based function implementation is shown in the figure below:



Input			Output
A	B	C	Y
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Fig. 6: Logic Function Generator Based on 3-Variable Logic Function Table

Operation Sequencing

Many industrial applications have processes that run in a sequence. A paint manufacturing plant might have a four step process to manufacture paint. Each of the four steps runs in a sequence one after the other. The second step cannot start before the first step has completed. Similarly, the third and fourth steps of the paint manufacturing process cannot proceed unless steps two and three have completed. It is not necessary that each of the manufacturing steps is of the same duration. Each manufacturing step can have different time duration and can be variable depending upon the quantity of paint manufactured or other parameters. Normally, the end of each step in the manufacturing process is indicated by a signal which is actuated by some machine which has completed its part of the manufacturing process. On receiving the signal, the next step of the manufacturing process is initiated. The entire sequence of operations is controlled by a multiplexer and a decoder circuit.

The manufacturing processes are started by resetting the 2-bit counter to 00. The counter output is connected to the select input of the multiplexer and the inputs of the decoder which selects the multiplexer input I0 is and activates the Decoder output Y0. The decoder output is connected to initiate the first process. When the process completes it indicates the completion of the process by setting its output to logic 1. The output of Process 1 is connected to I0 input of the Multiplexer. When Process 1 sets its output to 1 to indicate its completion, the logic 1 is routed by the Multiplexer to the clock input of the 2-it counter. The counter on receiving logic 1 increments its count to 01, which selects I1 input of the Multiplexer and the Y1 output of the Decoder. The input to Process 1 is deactivated and Process 2 is activated by Y1. On completion of Process 2 its output is set to logic 1, which is routed by the multiplexer to the clock input of the 2-bit counter which increments to the next count. This continues until Process 4 signals its completion after which the Decoder and the Multiplexer is deselected completing the manufacturing process.

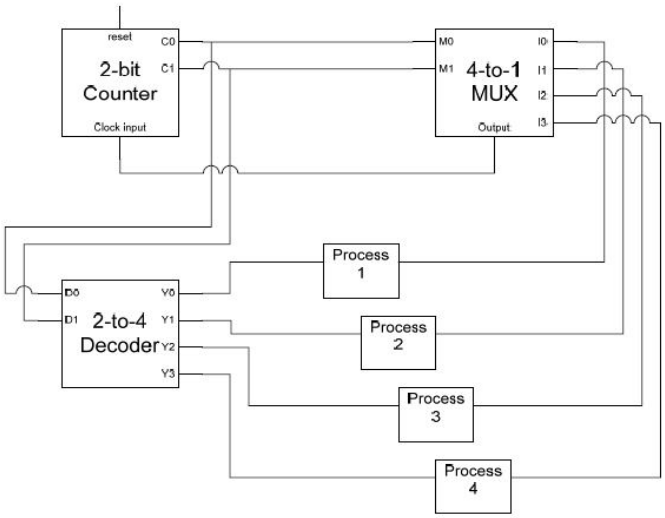
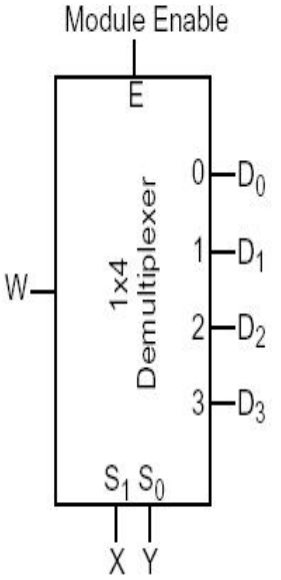


Fig. 7: Control of Manufacturing Process through Operation Sequencing

3.1.2 Demultiplexer

A multiplexer has several inputs. It selects one of the inputs and routes the data at the selected input to the single output. Demultiplexer has an opposite function to that of the multiplexer. It has a single input and several outputs. The demultiplexer selects one of the several outputs and routes the data at the single input to the selected output. A demultiplexer is also known as a data distributor.



Inputs			Outputs			
X	Y	W	D_3	D_2	D_1	D_0
0	0	0	0	0	0	W=0
0	1	0	0	0	W=0	0
1	0	0	0	W=0	0	0
1	1	0	W=0	0	0	0
0	0	1	0	0	0	W=1
0	1	1	0	0	W=1	0
1	0	1	0	W=1	0	0
1	1	1	W=1	0	0	0

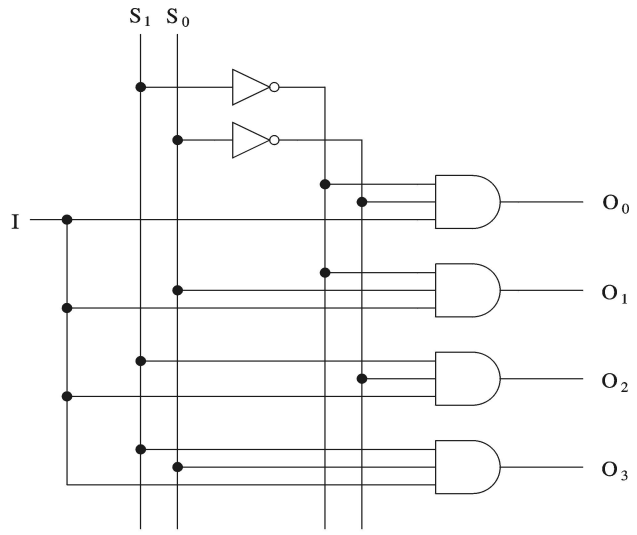


Fig. 8: A 1 x 4 Demultiplexer

The circuit if compared to that of the 2-to-4 decoder. The decoder enable input is used as the demultiplexer data input. A demultiplexer is not available commercially. A demultiplexer is available as a decoder/demultiplexer chip which can be configured to operate as a demultiplexer or a decoder.

The circuit of the 1-to-4 demultiplexer is similar to the 2-to-4 binary decoder. The only difference between the two is the addition of the data input line, which is used as enable line in the 2-to-4 decoder circuit.

3.1.2.1 Applications of Demultiplexer

Demultiplexer is used to connect a single source to multiple destinations as shown in the figure below:

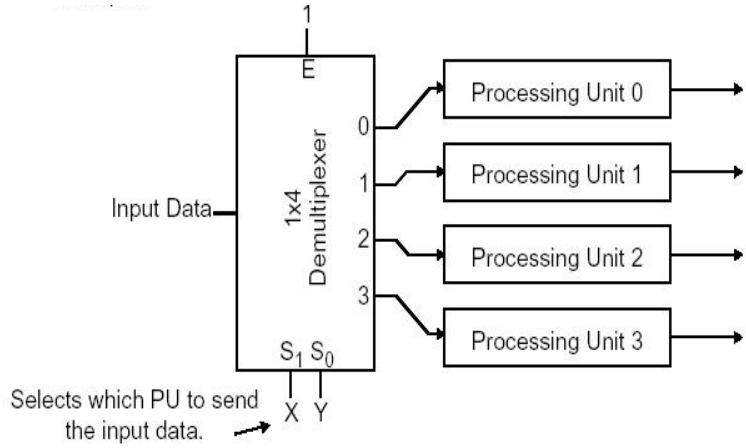


Fig. 9: Demultiplexer Used to Connect a Single Source to Multiple Destinations

It is used at the output of the ALU circuit. The output of the ALU has to be stored in one of the multiple registers or storage units. The Data input of the demultiplexer is connected to the output of the ALU. Each output of the demultiplexer is connected to each of the multiple registers. By selecting the appropriate output data from the ALU is routed to the appropriate register for storage.

The second use of the demultiplexer is the reconstruction of parallel data from the incoming serial data stream. Serial data arrives at the data input of the demultiplexer at fixed time intervals. A counter attached to the Select inputs of the demultiplexer routes the incoming serial bits to successive outputs where each bit is stored. When all the bits have been stored, data can be read out in parallel.

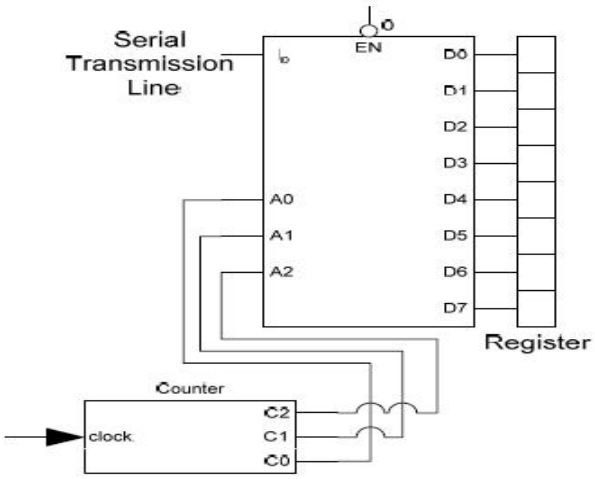


Fig. 10: DeMUX as a Serial to Parallel Converter

4.0 CONCLUSION

Multiplexer is a digital switch that has several inputs and a single output. It also has select inputs that allow any one of the multiple inputs can be selected to be connected to the output. They are also known as data selectors. Multiplexers are used in a wide variety of applications. Their primary use is to route data from multiple sources to a single destination. Other than its use as a data router, they are used as a parallel to serial converter, logic function generator and also for operation sequencing.

Demultiplexer has an opposite function to that of the Multiplexer. It has a single input and several outputs. It selects one of the several outputs and routes the data at the single input to the selected output. A demultiplexer is also known as a data distributor. It is used to connect a

single source to multiple destinations. It is also used for the reconstruction of parallel data from the incoming serial data stream.

5.0 SUMMARY

In this unit, we explained about multiplexers and demultiplexers, how they are designed and their applications.

6.0 TUTOR-MARKED ASSIGNMENT

Discuss briefly on the following:

i. MUX
ii. DeMUX
iii. 4 Differences b/w MUX and DeMUX
iv. Designing with multiplexers and demultiplexers
v. Applications of MUX and DeMUX.

7.0 REFERENCES/FURTHER READING

Comer, J. David (1994). *Digital Logic & State Machine Design*. Oxford University Press.

Dandamudi, S. (2003). *Fundamentals of Computer Organization & Design*. Springer.

Hayes, P. John (1993). *Introduction to Digital Logic Design*. Prentice Hall.

Hennessy, L. John & Patterson A. David (2008). *Computer Organization & Design*. M,organ Kaufmann.

Holdsworth, Brian & Woods Clive (2002). *Digital Logic Design*. Newnes.

Mano, M. Morris & Kime, Charles R. (2007). *Logic and Computer Design Fundamentals*. Prentice Hall. Pg 283.

Marcovitz, Alan (2009). *Introduction to Logic Design*. McGraw-Hill.

Neamen, Donald (2009). *Microelectronics: Circuit Analysis & Design*. McGraw-Hill.

Nelson, P. Victor *et al.* (1995). *Digital Logic Circuit Analysis & Design*. Prentice Hall.

Palmer, James & Periman, David (1993). *Schaum's Outline of Introduction to Digital Systems*. McGraw-Hill.

Patterson, David & Hennessy, John (1993). *Computer Organization & Design: The Hardware/Software Interface*. Morgan Kaufmann Publishers.

Rafiquzzaman, M. (2005). *Fundamentals of Digital Logic & Microcomputer Design*. Wiley – Interscience.

Roth, H. Charles Jr & Kinney, L. Larry (2009). *Fundamentals of Logic Design*.

Tanenbaum, Andrew S. (1998). *Structured Computer Organization.* Prentice Hall.

Tocci, J. Ronald & Widmer, S. Neal (1994). *Digital Systems: Principles and Applications.*

Tokheium, Roger (1994). *Schaum's Outline of Digital Principles*. McGraw-Hill.

Wakerly, F. John (2005). *Digital Design: Principles & Practices Package*. Prentice Hall.

UNIT 4 TYPICAL COMBINATIONAL LOGIC CIRCUIT III

CONTENTS

1.0 INTRODUCTION

In this unit, we shall discuss decoders and encoders, designing with them and their applications in our everyday lives.

2.0 OBJECTIVES

At the end of this unit, you should be able to:

- define a decoder
- design with decoders
- discuss the decoder networks and applications of decoders
- define an encoder
- define a priority encoder.

3.0 MAIN CONTENT

3.1 Typical Combinational Logic Circuit III

3.1.1 Decoders

A decoder has multiple inputs and multiple outputs. The decoder device accepts as an input a multi-bit code and activates one or more of its outputs to indicate the presence of the multi-bit code. A standard decoder is an m-to-n line where $m \leq 2^n$.

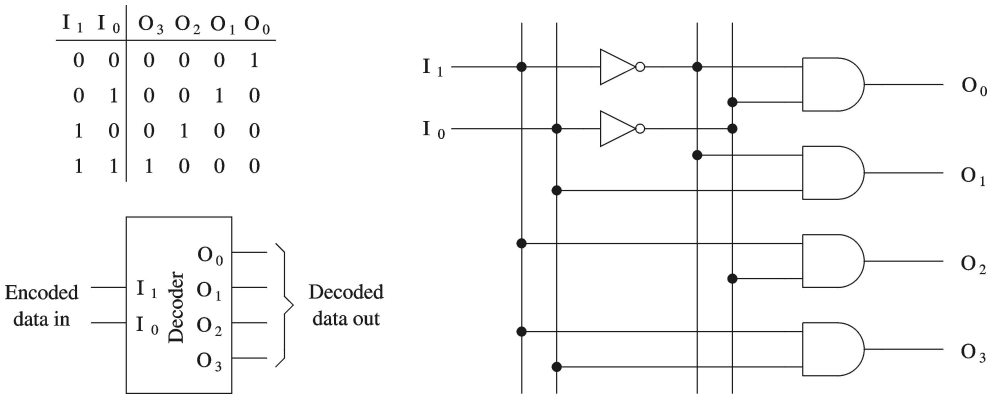
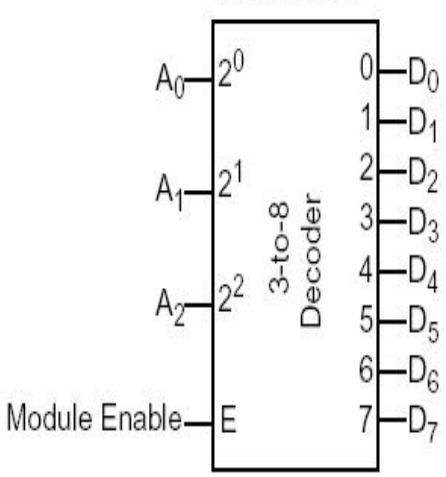


Fig. 1: Truth Table, Logic Symbol and Circuit Implementation of a 2-to-4 Decoder

3.1.1.1 Decoders with Enable Line

Often, decoders have an enable line that turns on outputs or leaves them off. The figure below shows a 3-to-8 decoder with enable and its truth table.



Inputs				Outputs							
A_2	A_1	A_0	E	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
X	X	X	0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0	0	0	1
0	0	1	1	0	0	0	0	0	0	1	0
0	1	0	1	0	0	0	0	0	1	0	0
0	1	1	1	0	0	0	0	1	0	0	0
1	0	0	1	0	0	0	1	0	0	0	0
1	0	1	1	0	0	1	0	0	0	0	0
1	1	0	1	0	1	0	0	0	0	0	0
1	1	1	1	1	0	0	0	0	0	0	0

Fig. 2: A 2-to-4 Decoder with Enable and its Truth Table

3.1.1.2 Designing with Decoders

Any Boolean function can be implemented using a decoder and OR gates as shown in the figure below:

Inputs			Outputs	
A_2	A_1	A_0	F_1	F_2
0	0	0	0	0
0	0	1	1	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	0	1
1	1	0	0	0
1	1	1	1	0

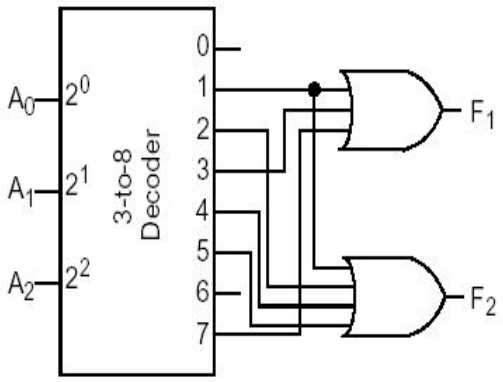


Fig. 3: Implementation of a Boolean Function

3.1.1.3 Decoder Networks

We can use multiple decoders to form a larger decoder. Below is a 3-to-8 decoder implemented with two 2-to-4 decoders.

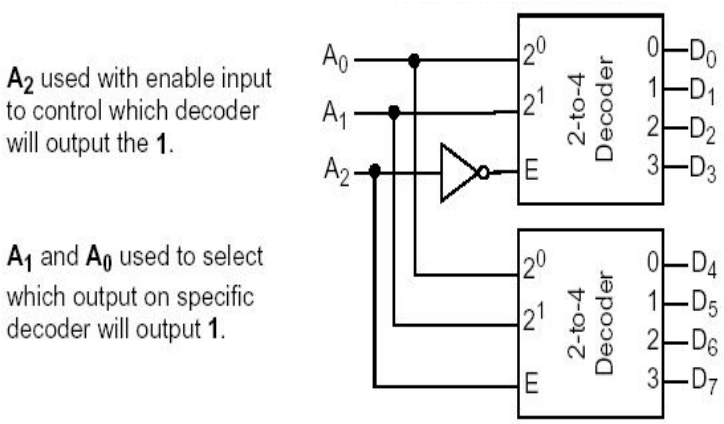


Fig. 4: A 3-to-8 Decoder Implemented with Two 2-to-4 Decoders

3.1.1.4 Applications of Decoders

Decoders have two major uses in Computer Systems.

Selection of Peripheral Devices

Computers have different internal and external devices like the Hard Disk, CD Drive, Modem, Printer, etc. Each of these different devices is selected by specifying different codes. A decoder is used to uniquely select or deselect the appropriate devices.

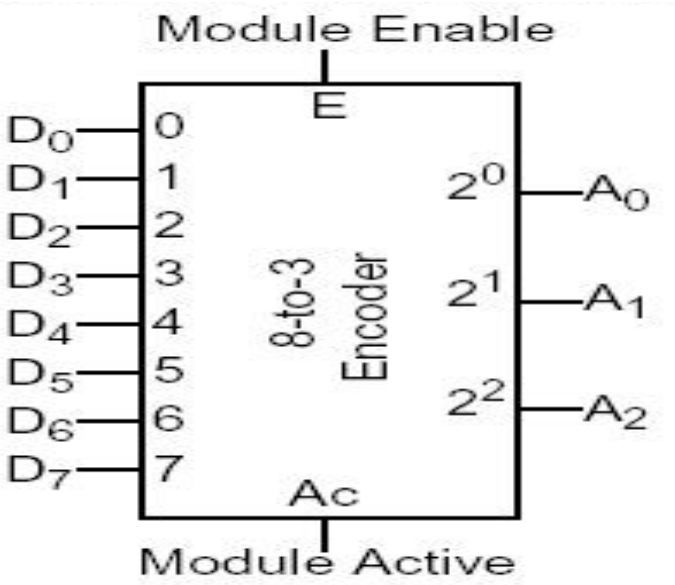
Instruction Decoder

Computer programs are based on instructions which are decoded by the computer hardware and implemented. The codes 1100010, 1100011, 1110000 and 1000101 represent - add two numbers, subtract two numbers, clear the result and store the result instructions.

These instruction codes are decoded by an instruction decoder to generate signals that control different logic circuits like the ALU and memory to perform these operations.

3.1.2 Encoders

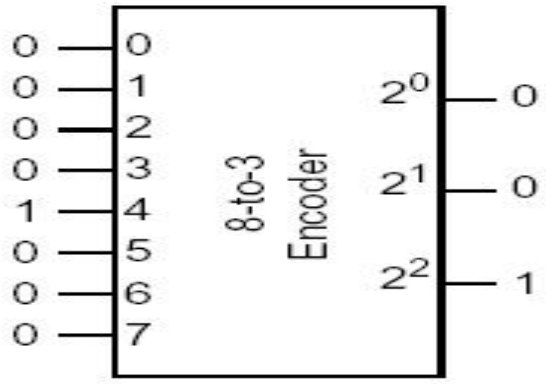
An encoder functional device performs an operation which is the opposite of the decoder function. The encoder accepts an active level at one of its inputs and at its output generates a BCD or binary output representing the selected input. A standard binary encoder is an m-to-n-line encoder, where $m \leq 2^{n.}$



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

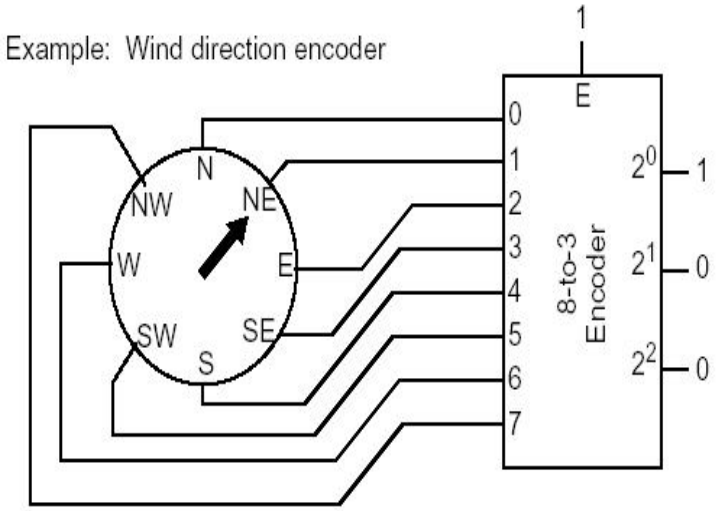
Fig. 5: An 8-to-3-Line Encoder with its Truth Table

Example: An input of 00010000 in an encoder will give



3.1.2.1 Designing with Encoders

Encoders are useful when the occurance of one of several disjoint events needs to be represented by an integer identifying the event.



3.1.2.2 Priority Encoders

A priority encoder takes the input of 1 with the highest index and translates that index to the output.

Enable input	I_3	I_2	I_1	I_0	O_1	O_0	Input active control signal
0	X	X	X	X	0	0	0
1	0	0	0	0	0	0	0
1	0	0	0	1	0	0	1
1	0	0	1	X	0	1	1
1	0	1	X	X	1	0	1
1	1	X	X	X	1	1	1

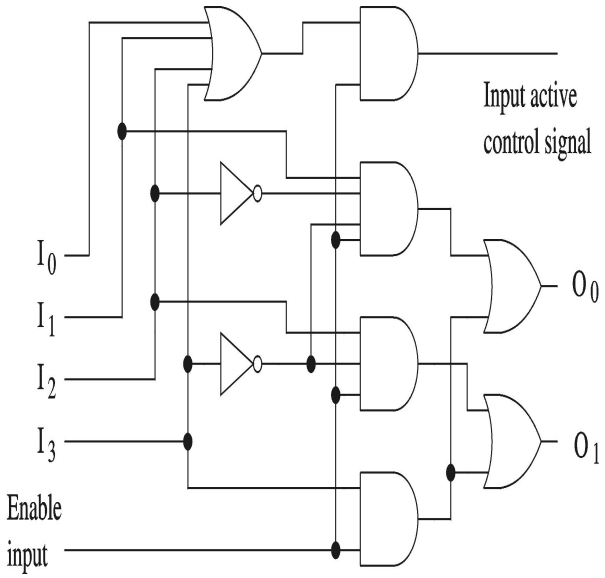


Fig. 6: Truth Table and Circuit Implementation of a Priority Encoder

3.1.2.3 Designing with P-Encoders

Priority encoders are useful when inputs have a predefined priority and we wish to select the input with the highest priority. An example is in resolving interrupt requests.

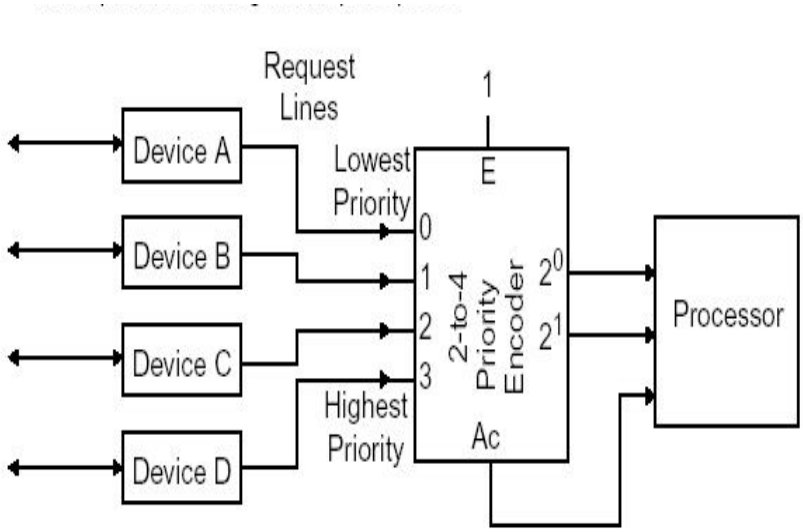


Fig. 7: Resolving Interrupt Requests

SELF-ASSESSMENT EXERCISE

Write on the applications of decoders and encoders.

4.0 CONCLUSION

A decoder has multiple inputs and multiple outputs. The decoder device accepts as an input a multi-bit code and activates one or more of its outputs to indicate the presence of the multi-bit code. Decoders have two major uses in computer systems: selection of peripheral devices and instruction decoder.

An encoder functional device performs an operation which is the opposite of the decoder function. The encoder accepts an active level at one of its inputs and at its output generates a BCD or binary output representing the selected input.

5.0 SUMMARY

In this unit, we examined decoders and encoders and also how to design with them.

6.0 TUTOR-MARKED ASSIGNMENT

Discuss briefly on the following:

i. decoders
ii. encoders
iii. differences between decoders and encoders
iv designing with decoders and encoders
v. applications of decoders and encoders.

7.0 REFERENCES/FURTHER READING

Comer, J. David (1994). *Digital Logic & State Machine Design*. Oxford University Press.

Dandamudi, S. (2003). *Fundamentals of Computer Organization and Design*. Springer.

Hayes, P. John (1993). *Introduction to Digital Logic Design*. Prentice Hall.

Hennessy, L. John & Patterson, A. David (2008). *Computer Organization & Design*. Morgan Kaufmann.

Holdsworth, Brian & Woods, Clive (2002). *Digital Logic Design*. Newnes.

Mano, M. Morris & Kime, Charles R. (2007). *Logic and Computer Design Fundamentals*. Prentice Hall.

Marcovitz, Alan (2009). *Introduction to Logic Design*. McGraw-Hill.

Neamen, Donald (2009). *Microelectronics: Circuit Analysis & Design*. McGraw-Hill.

Nelson, P. Victor; *et al*. (1995). *Digital Logic Circuit Analysis & Design*. Prentice Hall.

Palmer, James & Periman, David (1993). *Schaum's Outline of Introduction to Digital Systems*. McGraw-Hill.

Patterson, David & Hennessy, John (1993). *Computer Organization & Design: The Hardware/Software Interface*. Morgan Kaufmann Publishers.

Rafiquzzaman, M. (2005). *Fundamentals of Digital Logic & Microcomputer Design*. Wiley – Interscience.

Roth, H. Charles Jr & Kinney, L. Larry (2009). *Fundamentals of Logic Design*.

Tanenbaum Andrew S. (1998). *Structured Computer Organization.* Prentice Hall.

Tocci, J. Ronald & Widmer, S. Neal (1994). *Digital Systems: Principles and Applications.*

Tokheium, Roger (1994). *Schaum's Outline of Digital Principles*. McGraw-Hill.

Wakerly, F. John (2005). *Digital Design: Principles & Practices Package*. Prentice Hall.

MODULE 3 SEQUENTIAL LOGIC DESIGN & APPLICATIONS

UNIT 1 SEQUENTIAL LOGIC CIRCUITS

CONTENTS

1.0 INTRODUCTION

Digital logic circuits can be classified into two types: Combinational and Sequential logic circuits. We shall in this module explain the sequential logic circuit and its numerous applications.

2.0 OBJECTIVES

At the end of this unit, you should be able to:

- describe what a sequential logic circuit is
- state the differences between combinational and sequential logic circuit
- list and explain the types of sequential logic circuit.

3.0 MAIN CONTENT

3.1 Sequential Logic Circuits

3.1.1 Overview

The combinational digital circuits have no storage element; therefore combinational circuits handle only instantaneous inputs. The outputs of the combinational circuits also cannot be stored. The absence of a memory element restricts the use of digital combinational circuits to certain application areas. The use of a memory element which is capable of storing digital inputs and outputs is an important part of all practical digital circuits.

Consider an ALU which performs arithmetic and logical operations. An ALU cannot perform its operations unless it is connected to memory elements that store the inputs applied at the inputs of the ALU and outputs from the ALU. Consider an ALU that performs addition operation on a set of numbers, 2, 3, 4 and 5. The ALU can add two numbers at a time; therefore the ALU has to add the four numbers two at a time. The four numbers have to be stored temporarily; the partial results after adding two numbers also need to be stored. To add the four numbers, the first two numbers 2 and 3 are stored in two separate memory elements are added together, the result (5) has to be added to the next number 4. The result (5) is temporarily stored in one of the two memory elements used to store the numbers 2 and 3. The result (5) is added to the third number 4 to provide another partial sum result 9 which has to be stored and then added with the fourth number 5.

Digital circuits that use memory elements for their operation are known as Sequential circuits. Thus, sequential circuits are implemented by combining combinational circuits with memory elements as shown below:

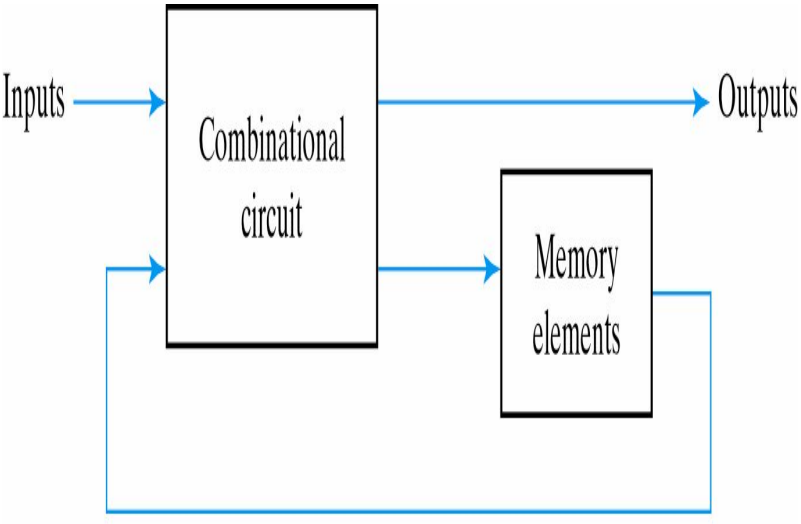


Fig. 1: Block Diagram of a Sequential Circuit

We have two types of sequential circuits namely:

Synchronous sequential circuits – Their behaviour is determined by the values of the signals at only discrete instants of time.

Asynchronous sequential circuits – Their behaviour is immediately affected by the input signal changes.

SELF-ASSESSMENT EXCERCISE

Write in details the two types of sequential circuits we have.

4.0 CONCLUSION

Digital circuits that use memory elements for their operation are known as sequential circuits. Thus, sequential circuits are implemented by combining combinational circuits with memory elements.

We have two types of sequential circuits namely: Synchronous sequential circuits and asynchronous sequential circuits.

5.0 SUMMARY

In this unit we explained sequential logic circuit and its types.

6.0 TUTOR-MARKED ASSIGNMENT

Extensively discuss sequential logic circuit and its types using diagrams and good examples.

7.0 REFERENCES/FURTHER READING

Comer, J. David (1994). *Digital Logic & State Machine Design*. Oxford University Press.

Dandamudi, S. (2003). *Fundamentals of Computer Organization and Design*. Springer.

Hayes, P. John (1993). *Introduction to Digital Logic Design*. Prentice Hall.

Hennessy, L. John & Patterson, A. David (2008). *Computer Organization & Design*. Morgan Kaufmann.

Holdsworth, Brian & Woods, Clive (2002). *Digital Logic Design*. Newnes.

Mano, M. Morris & Kime, Charles, R. (2007). *Logic and Computer Design Fundamentals*. Prentice Hall. Pg 283.

Marcovitz, Alan (2009). *Introduction to Logic Design*. McGraw-Hill.

Neamen, Donald (2009). *Microelectronics: Circuit Analysis & Design*. McGraw-Hill.

Nelson, P. Victor, *et al.* (1995). *Digital Logic Circuit Analysis & Design*. Prentice Hall.

Palmer, James & Periman, David (1993). *Schaum's Outline of Introduction to Digital Systems*. McGraw-Hill.

Patterson, David & Hennessy, John, (1993). *Computer Organization & Design: The Hardware/Software Interface*. Morgan Kaufmann Publishers.

Rafiquzzaman, M. (2005). *Fundamentals of Digital Logic & Microcomputer Design*. Wiley – Interscience.

Roth, H. Charles Jr & Kinney, L. Larry (2009). *Fundamentals of Logic Design*.

Tanenbaum, Andrew S. (1998). *Structured Computer Organization.* Prentice Hall.

Tocci, J. Ronald & Widmer, S. Neal (1994). *Digital Systems: Principles and Applications.*

Tokheium, Roger (1994). *Schaum's Outline of Digital Principles*. McGraw-Hill.

Wakerly, F. John (2005). *Digital Design: Principles & Practices Package*. Prentice Hall.

UNIT 2 LATCHES AND FLIP-FLOPS

CONTENTS

1.0 INTRODUCTION

In this unit, we shall extensively discuss latches and flip-flops.

2.0 OBJECTIVES

At the end of this unit, you should be able to:

- describe what a latch and flip-flop is
- describe how a latch and flip-flop are designed
- explain the various types of latch and flip-flop
- explain how the types of latch and flip-flop are designed
- state the various uses of latches and flip-flops
- explain the various applications of the latch and flip-flop.

3.0 MAIN CONTENT

3.1 Latches and Flip-Flops

The basic logic element that provides memory in many sequential circuits is the *flip-flop*.

3.1.1 Latches

A latch is a temporary storage device that has two stable states. A latch output can change from one state to the other by applying appropriate inputs. A latch normally has two inputs, the binary input combinations at the latch input allows the latch to change its state. A latch has two outputs Q and its complement Q. The latch is said to be in logic high state when Q=1 and Q=0 and it is in the logic low state when Q=0 and Q=1. When the latch is set to a certain state it retains its state unless the inputs are changed to set the latch to a new state.

Thus, a latch is a memory element which is able to retain the information stored in it.

3.1.1.1 The NAND Gate Based S-R (Set-Reset) Latch

An S-R Latch is implemented by connecting two NAND gates together. The output of each NAND gate is connected to the input of the other NAND gate. The unconnected inputs of the two NAND gates are the Set S and Reset R inputs. The outputs of the two NAND gates are the Q and its complement Q. The circuit diagram of the NAND based S-R latch is shown in figure 1.

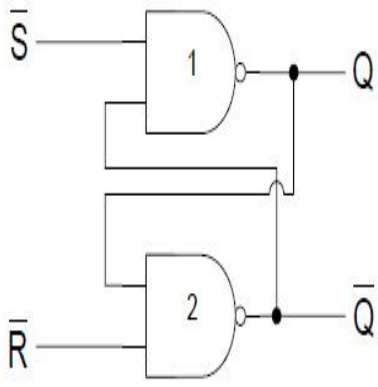


Fig. 1: NAND Based S-R Latch

A truth-table shows the operation of the S-R NAND based latch. The Output Q_{t+1} represents the Q output of NAND gate 1 at time interval t+1.When inputs are S = 1 and R = 1 the next state output Q_{t+1} remains the same as the previous state output Q_t. When inputs are S = 0 and R = 1 the output Q is set to 1. When inputs are S = 1 and R = 0 the output Q is set to 0. Inputs S = 0 and R = 0 are not applied as they place the latch in an invalid state.

The NAND gate based S-R latch has active-low inputs.

Table 1: Truth-Table of NAND based S-R Latch

Input		**Output**
S	R	Q_{t+1}
0	0	invalid
0	1	1
1	0	0
1	1	Q_t

3.1.1.2 The NOR Gate Based S-R (Set-Reset) Latch

A NOR based S-R latch is implemented using NOR gates instead of NAND gates.

Connections are identical to that of the NAND based latch. The S and R inputs have been switched.

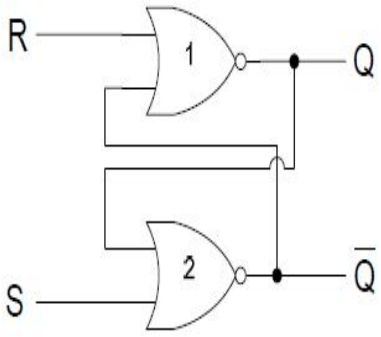


Fig. 2: NOR Based S-R Latch

The truth table of the NOR gate based latch is shown. When inputs are S =0 and R = 0 the next state output Qt+1 remains the same as the previous state output Qt. When inputs are S = 0 and R = 1 the output Q is set to 0. When inputs are S = 1 and R = 0 the output Q is set to 1. Inputs S = 1 and R = 1 are not applied as they place the latch in an invalid state. The NOR gate based S-R latch has active-high inputs.

Table 2: Truth-Table of NOR based S-R Latch

Input		Output
S	R	Qt+1
0	0	Qt
0	1	0
1	0	1
1	1	invalid

Comparing the operation of the NOR based and NAND based S-R latches.

The NAND based latch has active-low inputs, whereas NOR based latch has active-high inputs. Both the S-R latches are set to logic 1 when the set input is activated and the reset input is inactive.

Both the latches are set to logic 0 when the reset input is activated and the set input is inactive. The latches maintain the output state when both the set and reset inputs are inactive.

For both the latches both the set and reset inputs cannot be activated simultaneously as this leads to invalid output states. The logic symbols of the two latches are shown below.

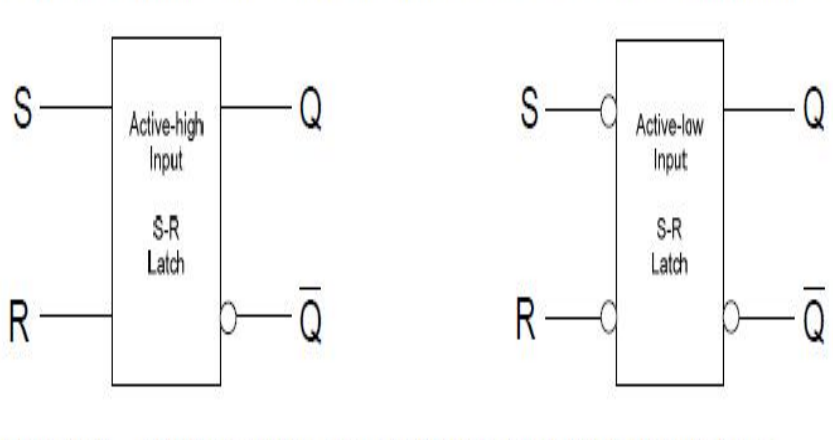


Fig. 3: NOR Based Active-High and NAND Based Active-Low S-R Latches

3.1.1.3 Application of S-R Latch

Digital systems use switches to input values and to control the output. For example, a keypad uses 10 switches to enter decimal numbers 0 to 9. When a switch is closed the switch contacts physically vibrate or 'bounce' before making a solid contact. The switch bounce causes the voltage at the output of the switch to vary between logic low and high

for a very short duration before it settles to a steady state. The variation in the voltage causes the digital circuit to operate in an erratic manner. An S-R latch connected between the switch and the digital circuit prevents the varying switch output from reaching the digital circuit.

In figure 3 above, when the switch is moved up to connect the resistor to the ground, the output voltage fluctuates between logic 1 and 0 for a very brief period of time when the switch vibrates before making a solid contact. The output voltage settles to logic 0 when a solid contact is made. The active-low input S-R latch shown in figure 4 prevents the output signal from varying between logic 1 and 0. When the switch is moved from down position to up position, the R input is set to 1 and S input is set to 0, which sets the Q output of the S-R latch to 1. The S input varies between 0 and 1 due to switch 'bounce', however the S-R latch doesn't change its output state Q when S = 1 and R = 1.

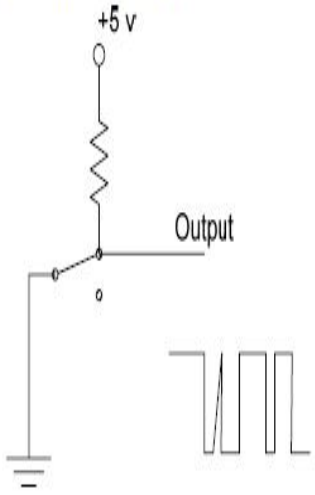


Fig. 4: The Output of a Switch Connected to Logic High

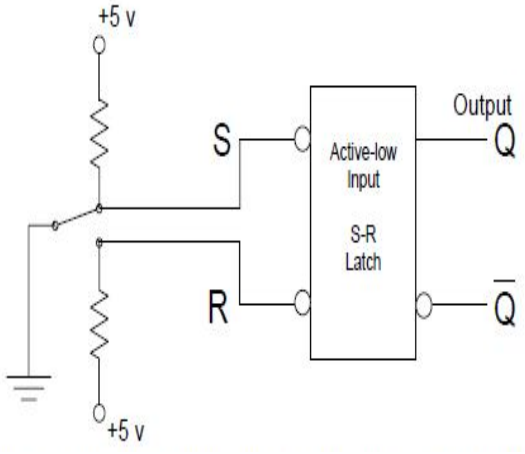


Fig. 5: The Switch Connected through an S-R Latch

The S-R NAND gate based latch is available in the form of an Integrated Circuit. The 74LS279 IC has four S-R latches which can be used independently.

3.1.1.4 The Gated S-R Latch

The gated S-R latch has an enable input which has to be activated to operate the latch. The circuit diagram of the gated S-R latch is shown in figure 6. In the gated S-R circuit, the S and R inputs are applied at the inputs of the NAND gates 1 and 2 when the enable input is set to active-high. If the enable input is disabled by setting it to logic low the output of NAND gates 3 and 4 remains logic 1, whatever the state of S and R inputs. Thus, logic 1 applied at the inputs of NAND gates 1 and 2 keeps the Q and Q outputs to the previous state. The logic symbol of a gated S-R latch is shown in figure 6.

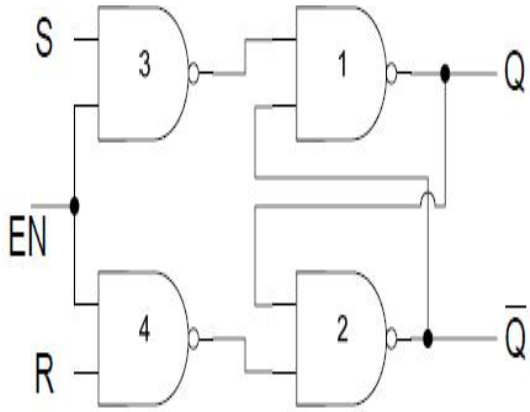


Fig. 6: Gated S-R Latch

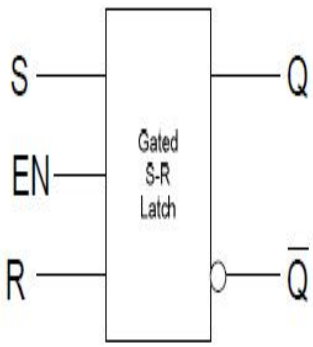


Fig. 7: Logic Symbol of a Gated S-R Latch

Table 3: Truth-Table of a Gated S-R Latch

Input			Output
EN	S	R	Qt+1
0	x	x	Qt
1	0	0	Qt
1	0	1	0
1	1	0	1
1	1	1	invalid

3.1.1.5 The Gated D Latch

One way to eliminate the undesirable condition of the indeterminate state in the *SR* latch is to ensure that inputs *S* and *R* are never equal to 1 at the same time. This is done by the *D* latch. Thus, the *D* latch has the ability to hold data in its internal storage. The output follows changes in the data input as long as the control input is gated. The circuit is often called a *transparent latch*.

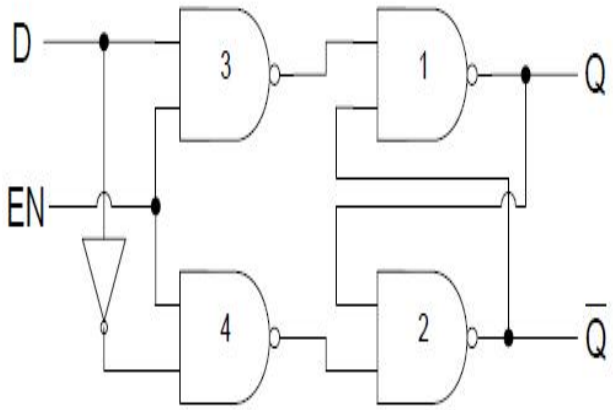


Fig. 8: Gated D Latch

Table 4: Truth Table of a Gated D Latch

Input			Output
EN	S (D)	R	Q_{t+1}
0	x	X	Q_t
1	0	0	Q_t
1	0	1	0
1	1	0	1
1	1	1	Invalid

Input		Output
EN	D	Q_{t+1}
0	x	Q_t
1	0	0
1	1	1

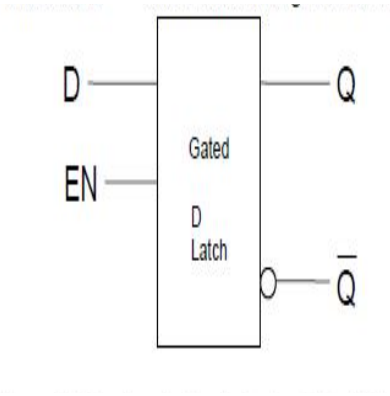


Fig. 9: Logic Symbol of a Gated D Latch

3.1.2 Flip-Flops

When latches are used for the memory elements in sequential circuits, a serious difficulty arises. Recall that latches have the property of immediate output responses (i.e., transparency).Because of this the output of a latch cannot be applied directly (or through logic) to the input of the same or another latch when all the latches are triggered by a common clock source. Flip-flops are used to overcome this difficulty.

Flip-flops are synchronous bi-stable devices, known as bi-stable multivibrators. Flip-flops have a clock input instead of a simple enable input. The output of the flip-flop can only change when appropriate inputs are applied at the inputs and a clock signal is applied at the clock input. Flip-flops with enable inputs can change their state at any instant when the enable input is active. Digital circuits that change their outputs when the enable input is active are difficult to design and debug as different parts of the digital circuit operate at different times.

In synchronous systems, the output of all the digital circuits changes when a clock signal is applied instead of the enable signal. The change in the state of the digital circuit occurs either at the low-to-high or high-to-low transition of the clock signal. Since the transition of the clock signal is for a very short and precise time intervals; thus, all digital parts of a digital system change their states simultaneously. The low to high or high to low transition of the clock is considered to be an edge. Three different types of edge-triggered flip-flops are generally used in digital logic circuits.

- S-R edge-triggered flip-flop
- D edge-triggered flip-flop
- J-K edge-triggered flip-flop

Each flip-flop has two variations, that is, it is either positive edge-triggered or negative edge triggered. A positive edge-triggered flip-flop changes its state on a low-to-high transition of the clock and a negative edge-triggered flip-flop changes its state on a high-to-low transition of the clock. The edge-detection circuit which allows a flip-flop to change its state on either the positive or the negative transition of the clock is implemented using a simple combinational circuit. The edge detection circuit that detects the positive and the negative clock transition are shown in figure 11.

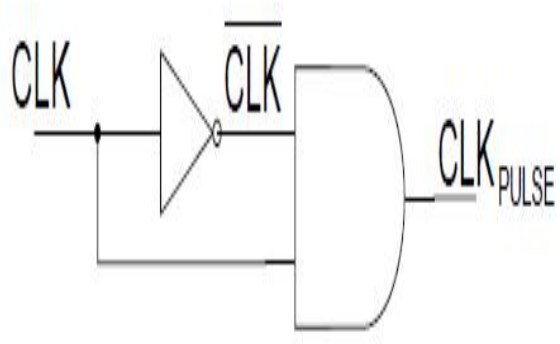


Fig. 10: Positive Clock Edge Detection Circuit

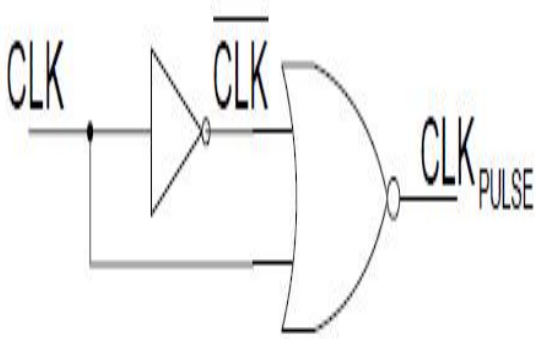


Fig. 11: Negative Clock Edge Detection Circuit

3.1.2.1 S-R Edge-Triggered Flip-Flops

The logic symbols of a positive edge and a negative edge triggered S-R flip-flops are shown in figure 12 below.

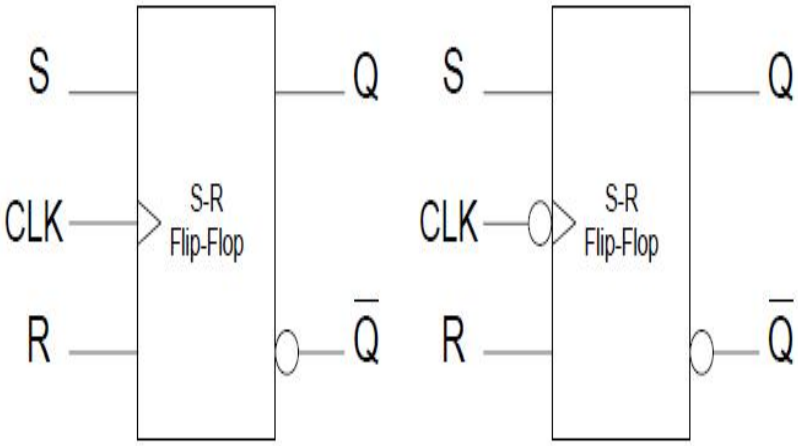


Fig. 12: Logic Symbol of Positive and Negative Edge -Triggered S-R Flip-Flops

The truth tables of the two S-R flip-flops are also shown below in table 5.

Table 5: Truth-Table of Positive and Negative Edge -Triggered S-R Flip-Flops

Input			Output
CLK	S	R	Q_{t+1}
0	X	X	Q_t
1	X	X	Q_t
↑	0	0	Q_t
↑	0	1	0
↑	1	0	1
↑	1	1	invalid

Input			Output
CLK	S	R	Q_{t+1}
0	x	x	Q_t
1	x	x	Q_t
↓	0	0	Q_t
↓	0	1	0
↓	1	0	1
↓	1	1	invalid

3.1.2.2 Edge-Triggered D Flip-Flops

The logic symbols of a positive edge and a negative edge triggered D flip-flops are shown in figure 13 below.

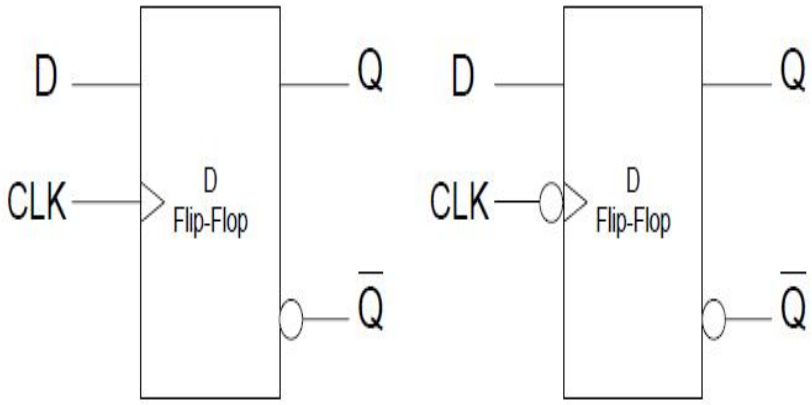


Fig. 13: Logic Symbol of Positive and Negative Edge Triggered D Flip-Flops

The truth tables of the two D flip-flops are also shown.

Table 6: Truth-Table of Positive and Negative Edge Triggered D Flip-Flops

Input		Output
CLK	D	Q_{t+1}
0	X	Q_t
1	X	Q_t
↑	0	0
↑	1	1

Input		Output
CLK	D	Q_{t+1}
0	X	Q_t
1	X	Q_t
↓	0	0
↓	1	1

3.1.2.3 Edge-Triggered J-K Flip-Flops

The J-K flip-flop is widely used in digital circuits. Its operation is similar to that of the SR flip-flop except that the J-K flip-flop doesn't have an invalid state; instead it toggles its state.

The circuit diagram of a J-K edge-triggered flip-flop is shown in figure 14.

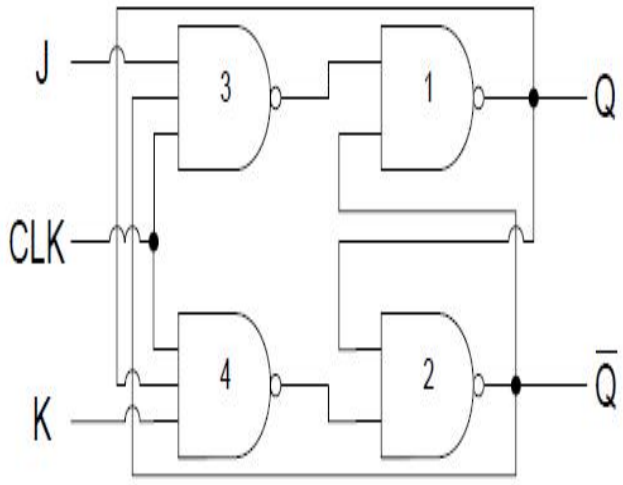


Fig. 14: Edge-Triggered J-K Flip-Flop

Consider the Q and $\overline{Q}$ output of the J-K flip-flop set to 1 and 0 respectively and 0 and 1 respectively. Four set of inputs are applied at J and K, the effect on the outputs is as follows.

J = 0 and K =0

With Q=1 and $\overline{Q}$=0, on a clock transition the outputs of NAND gates 3 and 4 are set to logic 1. With logic 1 value at the inputs of NAND gates 1 and 2 the output Q and Q remains unchanged. With Q=0 and Q=1, on a clock transition the outputs of the NAND gates 3 and 4 are set to logic 1. With logic 1 value at the inputs of NAND gates 1 and 2 the output Q and Q remains unchanged.

Thus, when J=0 and K=0 the previous state is maintained and there is no change in the output.

J = 0 and K =1

With Q=1 and $\overline{Q}$=0, on a clock transition the output of NAND gate 3 is set to logic 1.

The output of the NAND gate 4 is set to 0 as all three of its inputs are at logic 1. The logic 1 and 0 at the inputs of the NAND gates 3 and 4 respectively resets the Q output to 0 and $\overline{Q}$ to 1. With Q=0 and Q=1, on a clock transition the output of NAND gate 3 is set to logic 1. The

output of the NAND gate 4 is also set to 1 as the input of the NAND gate 4 is connected to Q=0. The logic 1 and 1 at the inputs of the NAND gates 3 and 4 respectively retains the Q and Q to 0 and 1 respectively.

Thus, when J=0 and K=1 the J-K flip-flop irrespective of its earlier state is rest to state $\overline{Q}$=0 and Q=1.

J = 1 and K =0

With Q=1 and $\overline{Q}$=0, on a clock transition the output of NAND gate 4 is set to logic 1.

The output of the NAND gate 3 is also set to 1 as its input connected to Q is at logic 0. Thus, inputs 1 and 1 at inputs of NAND gates 1 and 2 retain the Q and Q output to 1 and 0 respectively. With Q=0 and Q=1, on a clock transition the output of NAND gate 4 is set to logic 1. The output of the NAND gate 3 is set to 0 as all its input are at logic 1. Thus, inputs 0 and 1 at inputs of NAND gates 1 and 2 sets the flip-flop to Q=1 and Q=0.

Thus, when J=1 and K=0 the J-K flip-flop irrespective of its output state is set to state $\overline{Q}$=1 and Q=0.

J = 1 and K =1

With Q=1 and $\overline{Q}$=0, on a clock transition the output of the NAND gates 3 and 4 depend on the outputs Q and Q. The output of NAND gate 3 is set to 1 as Q is connected to its input. The output of NAND gate 4 is set to 0 as all its inputs including Q is at logic 1. A logic 1 and 0 at the input of gates 1 and 2 toggles the outputs Q and Q from logic 1 and 0 to 0 and 1 respectively. With Q=0 and Q=1, on a clock transition the output of NAND gate 3 is set to 0 as Q and the output of NAND gate 4 is set to 1. A logic 0 and 1 at the input toggles the outputs Q and Q from logic 0 and 1 to 1 and 0 respectively.

In summary, when J-K inputs are both set to logic 0, the output remains unchanged. At J=0 and K=1 the J-K flip-flop is reset to Q=0 and Q=1. At J=1 and K=0 the flip-flop is set to Q=1 and Q=0. With J=1 and K=1 the output toggles from the previous state.

The truth tables of the positive and negative edge triggered J-K flip-flops are shown in Table 7.

Table 7: Truth-Table of Positive and Negative Edge Triggered J-K Flip-Flops

Input			Output
CLK	J	K	Q_{t+1}
0	x	X	Q_t
1	x	X	Q_t
↑	0	0	Q_t
↑	0	1	0
↑	1	0	1
↑	1	1	$\overline{Q}_t$

Input			Output
CLK	J	K	Q_{t+1}
0	x	x	Q_t
1	x	x	Q_t
↓	0	0	Q_t
↓	0	1	0
↓	1	0	1
↓	1	1	$\overline{Q}_t$

The logic symbols of the J-K flip-flops are shown in figure 15.

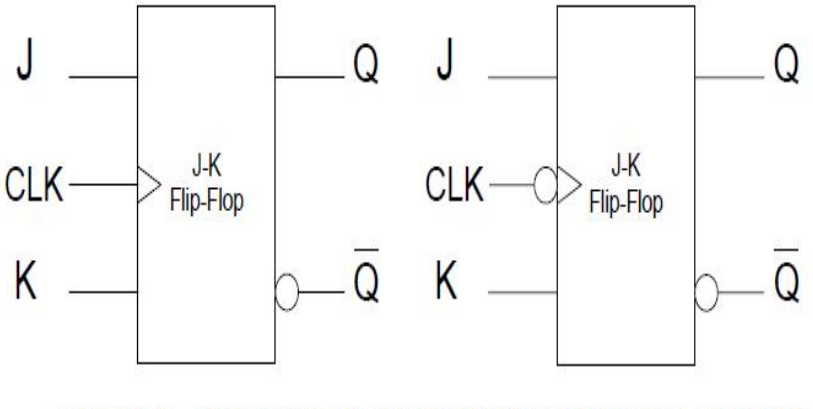


Fig. 15: Logic Symbol of Positive and Negative Edge Triggered J-K Flip-Flops

3.1.2.4 Asynchronous Preset and Clear Inputs

The S-R, J-K and D inputs are known as synchronous inputs because the outputs change when appropriate input values are applied at the inputs and a clock signal is applied at the clock input. If there is no clock transition then the inputs have no effect on the output.

Digital circuits require that the flip-flops be set or reset to some initial state before a new set of inputs is applied for changing the output. The flip-flops are set-reset to some initial state by using asynchronous inputs known as Preset and Clear inputs. Since these inputs change the output

to a known logic level independently of the clock signal therefore these inputs are known as asynchronous inputs. The circuit diagram of a J-K flip-flop with Preset and Set is shown below.

Asynchronous inputs are shown. The asynchronous inputs override the synchronous inputs. Thus, to operate the flip-flop in the synchronous mode the asynchronous inputs have to be disabled. To preset the flip-flop to Q=1 and $\overline{Q}$=0 the PRE input is set to 0 which sets the Q output to 1 and the output of NAND gate 4 to 1. The CLR input is set to 1 which sets the Q output to 0 as all three inputs of the NAND gate 2 are set to 1. The flip-flop is cleared to Q=0 and Q=1 by setting the PRE input is set to 1 and the CLR input is to 0. The CLR input set to 0 sets Q=1 it also sets the output of NAND gate 3 to 1. The PRE input set to1 sets the output Q to 0. When the PRE and the CLR inputs are used inputs J and K have no effect on the operation of the flip-flop. To use the flip-flop with synchronous inputs J-K, the PRE and the CLR inputs are set to logic 1. Setting PRE and the CLR to logic 0 is not allowed.

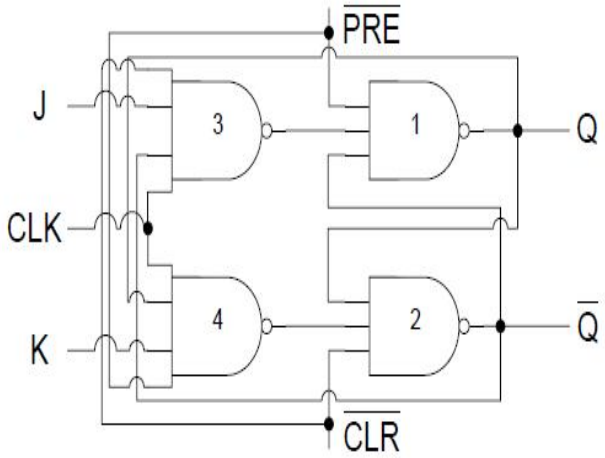


Fig. 16: J-K Flip-Flop with Asynchronous Preset and Clear Inputs

Figure 17 shows the logic symbol of a J-K edge-triggered flip-flop with synchronous and asynchronous inputs.

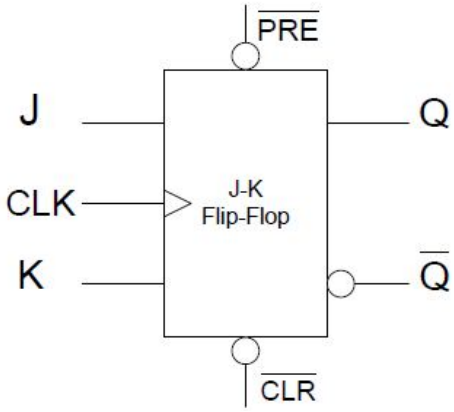


Fig. 17: Logic Symbol of a J-K Flip-Flop with Asynchronous Inputs

The truth table of a J-K flip-flop with asynchronous inputs is shown in Table 8.

Table 8: Truth Table of J-K Flip-Flop with Asynchronous Inputs

Input		Output
$\overline{PRE}$	$\overline{CLR}$	Q_{t+1}
0	0	Invalid
0	1	1
1	0	0
1	1	Clocked operation

The 74HC74 Dual Positive-Edge triggered D Flip-Flop

The edge-triggered D flip-flop with asynchronous inputs is available as an Integrated Circuit. The 74HC74 has dual D-flip-flops with independent clock inputs, synchronous and asynchronous inputs.

The 74HC112 Dual Positive-Edge triggered J-K flip-flop

The edge-triggered D flip-flop with asynchronous inputs is available as an Integrated Circuit. The 74HC112 has dual J-K-flip-flops with independent clock inputs, synchronous and asynchronous inputs.

3.1.2.5 Master-Slave Flip-Flops

Master-Slave flip-flops have become obsolete and are being replaced by edge triggered flip-flops. Master-Slave flips have two stages each stage works in one half of the clock signal. The inputs are applied in the first half of the clock signal. The outputs do not change until the second half of the clock signal. As mentioned earlier the use of edge-triggered flip-flop is to synchronise the operation of a digital circuit with a common clock signal. The master-slave setup also allows digital circuits to operate in synchronisation with a common clock signal. The circuit diagram of the master-slave J-K flip-flop is shown in figure 18 below. The Master-Slave flip-flop is composed of two parts the Master and the Slave. Both the Master and the Slave are Gated S-R flip-flops. The Master-Slave flip-flop is not synchronised with the clock positive or negative transition, rather it known as a pulse triggered flip-flop as it operates at the positive and negative clock cycles.

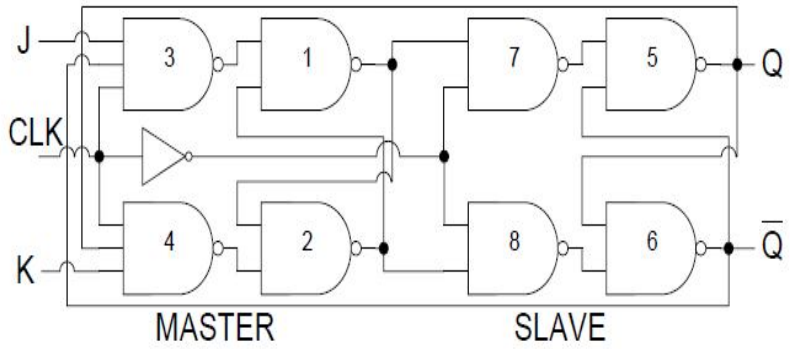


Fig. 18: Master-Slave Flip-Flop

The truth-table of the master-slave flip-flop is shown in table 9 below.

Table 9: Truth Table of the Master-Slave J-K Flip-Flop

Input			Output
CLK	J	K	Q_{t+1}
Pulse	0	0	Q_t
Pulse	0	1	0
Pulse	1	0	1
Pulse	1	1	$\overline{Q}_t$

3.1.2.6 Flip-Flop Operating Characteristics

The performance of the flip-flop is specified by several operating characteristics.

The important operating characteristics are

- Propagation Delay
- Set-up Time
- Hold Time
- Maximum Clock frequency
- Pulse width
- Power Dissipation

Propagation Delay

The propagation delay time is the interval of time when the input is applied and the output changes. Four different types of Propagation Delays are measured.

1. Propagation delay t_{PLH} measured with respect to the triggering edge of the clock to the low-to-high transition of the output. On a positive or negative clock transition the flip-flop changes its output state. The propagation delay is measured at 50% transition mark on the triggering edge of the clock and the 50% mark on the low-to-high transition of the output that occurs due to the clock transition.

2. Propagation delay t_{PHL} measured with respect to the triggering edge of the clock to the high-to-low transition of the output. On a positive or negative clock transition the flip-flop changes its output state. The propagation delay is measured at 50% transition mark on the triggering edge of the clock and the 50% mark on the high-to-low transition of the output that occurs due to the clock transition.

3. Propagation Delay t_{PLH} measured with respect to the leading edge of the preset input to the low-to-high transition of the output. On a high-to-low transition of the preset signal the flip-flop changes its output state to logic high. The propagation delay is measured at 50% transition mark on the triggering edge of the preset signal and the 50% mark on the low-to-high transition of the output that occurs due to the preset signal.

4. Propagation delay t_{PHL} measured with respect to the leading edge of the clear input to the high-to-low transition of the output. On a

high-to-low transition of the clear signal the flip-flop changes its output state to logic low. The propagation delay is measured at 50% transition mark on the triggering edge of the clear signal and the 50% mark on the high-to-low transition of the output that occurs due to the preset signal.

Set-Up Time

When a clock transition occurs at the clock input of a flip-flop the output of the flip-flop is set to a new state based on the inputs. For the flip-flop to change its output to a new state at the clock transition, the input should be stable. The minimum time required for the input logic levels to remain stable before the clock transition occurs is known as the set-up time

Hold Time

The input signal maintained at the flip-flop input has to be maintained for a minimum time after the clock transition for the flip-flop to reliably clock in the input signal. The minimum time for which the input signal has to be maintained at the input is the hold time of the flip-flop.

Maximum Clock Frequency

A flip-flop can be operated at a certain clock frequency. If the clock frequency is increased beyond a certain limit the flip-flop will be unable to respond to the fast changing clock transitions, therefore the flip-flop will be unable to function. The maximum clock frequency f_{max} is the highest rate at which the flip-flop operates reliably.

Pulse Width

A flip-flop uses the clock, preset and clear inputs for its operation. Each signal has to be of a specified duration for correct operation of the flip-flop. The manufacturer specifies the minimum pulse width t_w for each of the three signals. The clock signal is specified by minimum high time and minimum low time.

Power Dissipation

A flip-flop consumes power during its operation. The power consumed by a flip-flop is defined by $P = V_{cc} \times I_{cc}$. The flip-flop is connected to +5 volts and it draws 5 mA of current during its operation, therefore the power dissipation of the flip-flop is 25 mW.

A digital circuit is made of a number of gates, functional units and flip-flops. The total power requirement of each device should be known so that an appropriate dc power source is used to supply power to the digital circuit.

3.1.2.7 Applications of Edge-Triggered D Flip-Flops

1. Data Storage using D-Flip-Flop

A multiplexer based parallel-to-serial converter needs to have stable parallel data at its inputs as it converts it to serial data. Latches are used to maintain stable data at the input of the multiplexer. The time required to convert parallel data to serial data depends upon the number of parallel bits. A byte parallel data requires 8-bit storage and 8 clocks are required to convert it into serial data. The demerit in a gated D-latch based circuit is the extended enable time. During the time in which the D-latches are enabled data applied at the input of the latches can change. D-latch is said to work in transparent mode when the enable signal is activated. D-latch operates in the latched mode when the enable signal is inactive. The conversion should only start when the enable signal has been deactivated and the 8-bit data has been stored in the latches. A better and a precise parallel to serial converter circuit uses edge triggered D-flip-flops. The 8-bit data to be converted into serial data is stored precisely at the clock transition. Thus, if the data changes after the clock transition it has no effect on the data stored in the D flip-flop.

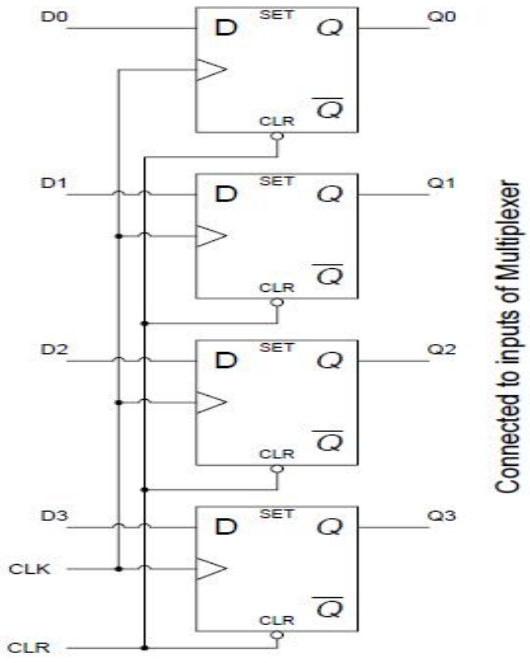


Fig. 19: D-Flip-Flops used for Parallel Data Storage

2. Synchronising Asynchronous inputs using D Flip-Flop

In synchronised digital systems all the circuits change their state with respect to a common clock and all the input and output signals are synchronised. However, external inputs that are applied to digital circuits through switches and keypads are not synchronised with the clock. The asynchronous inputs can occur at any instant of time

3. Parallel Data Transfer using D flip-flop

Microprocessor use multi-bit flip-flops to store information. These multi-bit flip-flops are known as registers. These registers for example, can store data generated at the output of the ALU. The registers can also be used to exchange or copy data, see figure 21. A register is a set of flip-flops connected in parallel to store multi-bit binary information. The clock inputs of all the flip-flops are connected together, to allow simultaneous latching of the multi-bit input data.

3.1.2.8 Applications of Edge-Triggered J-K Flip-Flops

1. J-K Flip-Flop used as Sequence Detector

Some digital applications require that the inputs be applied in a certain sequence to activate an output. This is possible with J-K flip-flops.

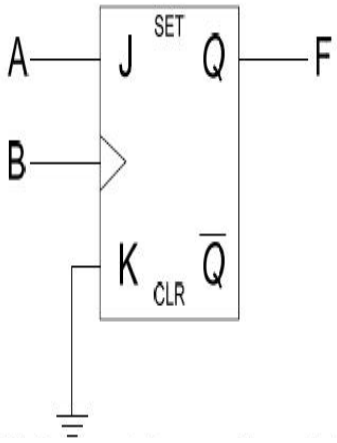


Fig. 20: J-K Flip-Flop Connected to Respond to a Particular Input Sequence

2. J-K Flip-Flop used as Frequency Divider

In digital circuit different parts of the circuit can operate at different frequencies obtained from the master clock frequency. For example, three different parts of a digital system might operate at 4 MHZ, 2 MHZ

and 1 MHZ clock frequency respectively. Same clock source should be used (instead of three separate clock sources) to maintain synchronization between the three parts. A clock frequency can be divided by 2 using a J-K flip flop. The J-K inputs of the flip-flop are connected to logic high (1). At each clock transition the output of the flip-flop toggles to the alternate state. A 4MHz clock signal can be divided into 2

MHZ and 1 MHZ signal using two J-K flip-flops connected together.

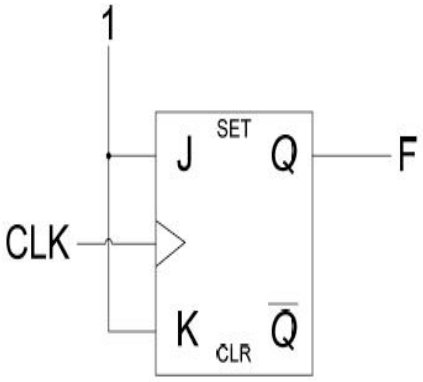


Fig. 21: J-K Flip-Flop Connected as Frequency Divider

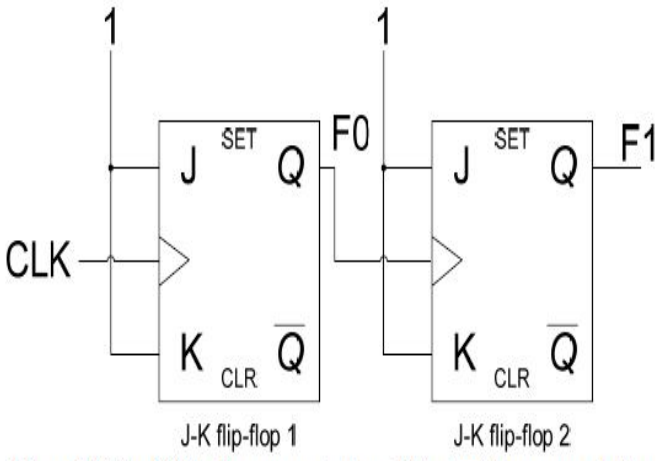


Fig. 22: J-K Flip-Flop Connected as Divide-by-4 Frequency Divider

3. J-K Flip-Flop used as a Shift Register

Binary numbers can be multiplied or divided by a constant 2 by shifting the binary numbers left or right by 1-bit respectively. Multiplication and division by a factor of 2n, (where n= 1, 2, 3, 4 ….) can be achieved by shifting the binary by n bits to the left or right respectively. Binary

numbers can be easily shifted in the left or right direction by using J-K flip-flop based shift registers.

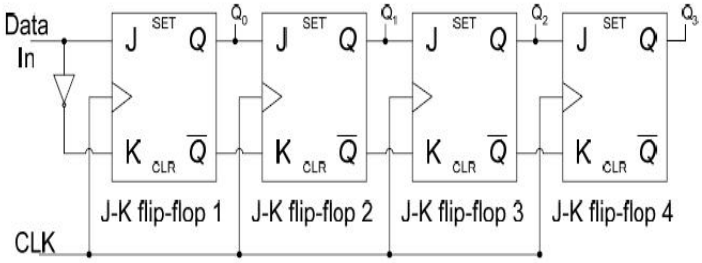


Fig. 23: 4-Bit Right Shift Register

4. J-K Flip-Flop used as a Counter

Counters which count up or countdown are commonly used in digital circuits. An up counter counts up from 0 to 10 increments to the next higher count value on the application of each clock signal. Similarly, a down-counter counts down to the next lower count value on the application of each clock pulse.

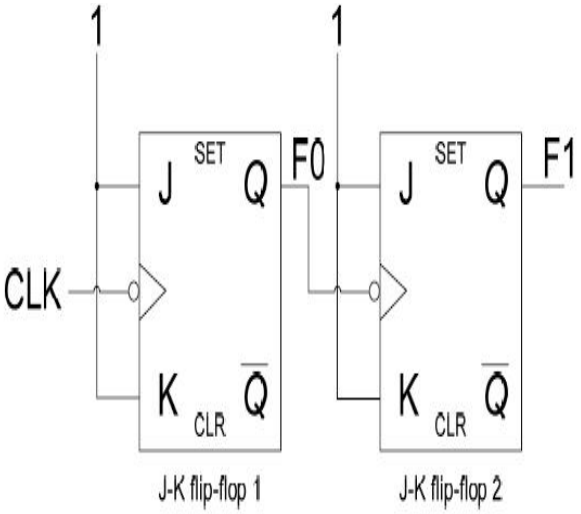


Fig. 24: 2-Bit Up-Counter

4.0 CONCLUSION

A latch is a temporary storage device that has two stable states. A latch output can change from one state to the other by applying appropriate inputs. A latch normally has two inputs, the binary input combinations at the latch input allows the latch to change its state.

The basic logic element that provides memory in many sequential circuits is the flip-flop. Flip-flops are synchronous bi-stable devices, known as bi-stable multivibrators. Flip-flops have a clock input instead of a simple enable input. The output of the flip-flop can only change when appropriate inputs are applied at the inputs and a clock signal is applied at the clock input.

Three different types of edge-triggered flip-flops generally used in digital logic circuits are:

S-R edge-triggered flip-flop
D edge-triggered flip-flop
J-K edge-triggered flip-flop.

5.0 SUMMARY

In this unit we explained about latches and flip-flops. Digital systems use switches to input values and to control the output. For example, a keypad uses 10 switches to enter decimal numbers 0 to 9. When a switch is closed the switch contacts physically vibrate or 'bounce' before making a solid contact. The switch bounce causes the voltage at the output of the switch to vary between logic low and high for a very short duration before it settles to a steady state. The variation in the voltage causes the digital circuit to operate in an erratic manner. An S-R latch connected between the switch and the digital circuit prevents the varying switch output from reaching the digital circuit.

6.0 TUTOR-MARKED ASSIGNMENT

Extensively discuss latches and flip-flops and their types using diagrams and good examples.

7.0 REFERENCES/FURTHER READING

Comer, J. David (1994). *Digital Logic & State Machine Design*. Oxford University Press.

Dandamudi, S. (2003). *Fundamentals of Computer Organization and Design*. Springer.

Hayes, P. John (1993). *Introduction to Digital Logic Design*. Prentice Hall.

Hennessy, L. John & Patterson, A. David (2008). *Computer Organization & Design*. Morgan Kaufmann.

Holdsworth, Brian & Woods, Clive (2002). *Digital Logic Design*. Newnes.

Mano, M. Morris & Kime, Charles, R. (2007). *Logic and Computer Design Fundamentals*. Prentice Hall. Pg 283.

Marcovitz, Alan (2009). *Introduction to Logic Design*. McGraw-Hill.

Neamen, Donald (2009). *Microelectronics: Circuit Analysis & Design*. McGraw-Hill.

Nelson, P. Victor, *et al* (1995). *Digital Logic Circuit Analysis & Design*. Prentice Hall.

Palmer, James & Periman, David (1993). *Schaum's Outline of Introduction to Digital Systems*. McGraw-Hill.

Patterson, David & Hennessy, John (1993). *Computer Organization & Design: The Hardware/Software Interface*. Morgan Kaufmann Publishers.

Rafiquzzaman M. (2005). *Fundamentals of Digital Logic & Microcomputer Design*. Wiley – Interscience.

Roth, H. Charles Jr & Kinney, L. Larry (2009). *Fundamentals of Logic Design*.

Tanenbaum, Andrew S. (1998). *Structured Computer Organization.* Prentice Hall.

Tocci, J. Ronald & Widmer, S. Neal (1994). *Digital Systems: Principles and Applications.*

Tokheium, Roger (1994). *Schaum's Outline of Digital Principles*. McGraw-Hill.

Wakerly, F. John (2005). *Digital Design: Principles & Practices Package*. Prentice Hall.

UNIT 3 REGISTERS

CONTENTS

1.0 INTRODUCTION

In this unit we shall discuss registers. We shall look at the shift register, shift register counters and the applications of shift registers.

2.0 OBJECTIVES

At the end of this unit, you should be able to:

- define shift register
- explain its various types
- explain its various applications.

3.0 MAIN CONTENT

3.1 Registers

Information is stored in a CPU memory location called a register. Registers can be thought of as the CPU's tiny scratchpad, temporarily storing instructions or data. When a program is running, one special register called the program counter keeps track of which program instruction comes next by maintaining the memory location of the next program instruction to be executed. The CPU's control unit coordinates and times the CPU's functions, and it uses the program counter to locate and retrieve the next instruction from memory.

3.1.1 Shift Registers

In digital circuits multi-bit data has to be stored temporarily until it is processed. A flip-flop is able to store a single binary bit of information. Multiple bits of data are stored by using multiple flip-flops which have their clock inputs connected together. Thus, by activating the clock signal multiple-bits of data are stored.

Technically, a register performs two basic functions. It stores data and it moves or shifts data. The shifting of data involves shifting of bits from one flip-flop to the other within the register or moving data in and out of the register. The shift operation of the binary data is carried out by applying clock signals. Several different kinds of shift operations can be identified. The different shift operations are described using a 4-bit shift register.

Serial In/Shift Right/Serial Out Operation

Data is shifted in the right-hand direction one bit at a time with each transition of the clock signal. The data enters the shift register serially from the left hand side and after four clock transitions the 4-bit register has 4-bits of data. The data is shifted out serially one bit at a time from the right hand side of the register if clock signals are continuously applied. Thus, after 8 clock signals the 4-bit data is completely shifted out of the shift register.

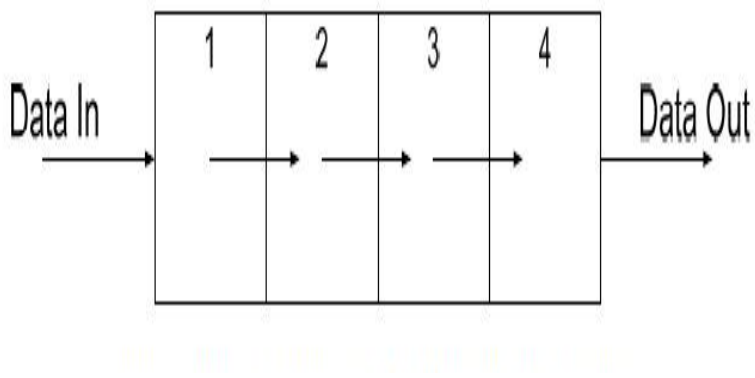


Fig. 1: Serial In/Serial Right/Serial Out Operation

Serial In/Shift Left/Serial Out Operation

Data is shifted in the left-hand direction one bit at a time with each transition of the clock signal. The data enters the shift register serially from the right hand side and after four clock transitions the 4-bit register has 4-bits of data. The data is shifted out serially one bit at a time from

the left hand side of the register if clock signals are continuously applied. Thus, after 8 clock signals the 4-bit data is completely shifted out of the shift register.

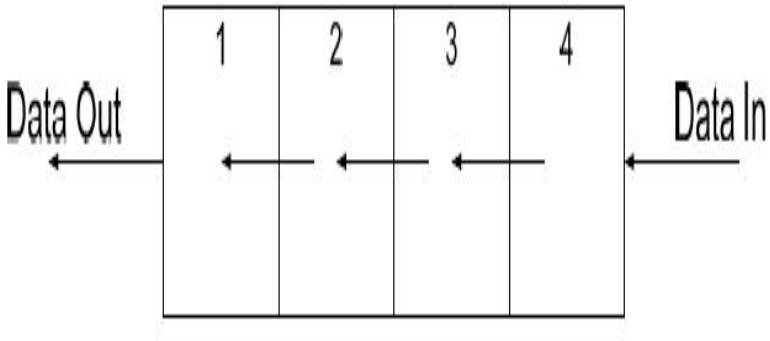


Fig. 2: Serial In/Serial Left/Serial Out Operation

Serial shift registers can be implemented using any type of flip-flops. A serial shift register implemented using D flip-flops with the serial data applied at the D input of the first flip-flop and serial data out obtained at the Q output of the last flip-flop is shown. At each clock transition 1–bit of serial data is shifted in and at the same instant 1-bit of serial data is shifted out. For a 4-bit shift register, 8 clock transitions are required to shift in 4-bit data and completely shift out the 4-bit data. As the data is shifted out 1-bit at a time, a logic 0 value is usually shifted in to fill up the vacant bits in the shift register.

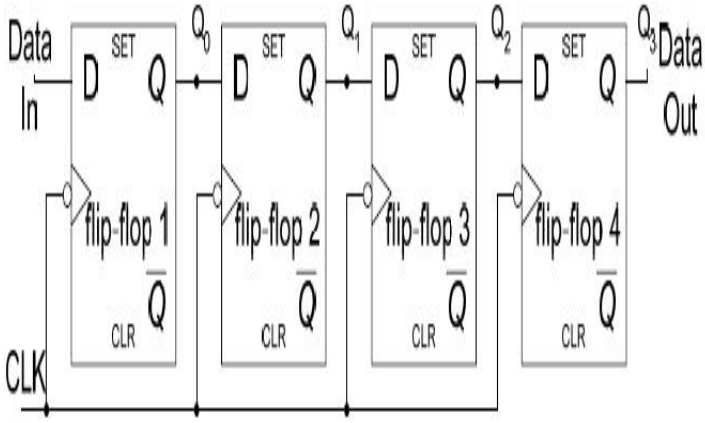


Fig. 3: Serial In/Shift Right/Serial Out Register

The shift left and shift right shift registers are identical in their working. They are connected differently for shift left and shift right operations. Bidirectional Shift Registers are available which allow data to be shifted

left or right. The 4-bit register is configured to shift left or right by setting the RIGHT /LEFT signal to logic high or low respectively. When the register is configured to shift right, the AND gates marked 1 are enabled. The input of the first flip-flop is connected to the serial Input, the inputs of the next three flip-flops are connected to the Q outputs of the previous flip-flops. Thus, on a clock transition data is shifted 1-bit towards the right. The serial data is shifted out of the register through output Q3. When the register is configured to shift left the AND gates marked 2 are enabled, connecting the Q outputs of the flip-flop on the right hand side to the D input of the flip-flop on the left hand side. Thus, on each clock transition data is shifted 1-bit towards left. Serial date out is available through the Q0 output. Serial data is input through the Serial Data in line which is connected to the fourth AND gate marked 2 on the extreme right hand side.

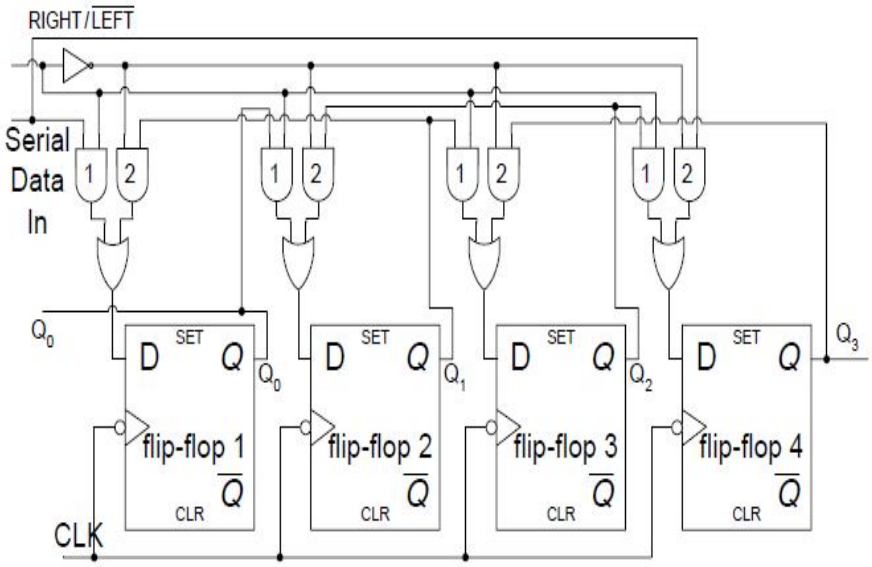


Fig. 4: Bi-Directional, 4-Bit Shift Register

Serial In/Parallel Out Operation

Data is shifted in the left-hand direction one bit at a time with each transition of the clock signal. The data enters the shift register serially from the right hand side and after four clock transitions the 4-bit register has 4-bits of data. The data is shifted out in parallel by the application of a single clock signal. The shift register has 4 parallel outputs. The circuit diagram of the Serial In/Parallel Out register is shown.

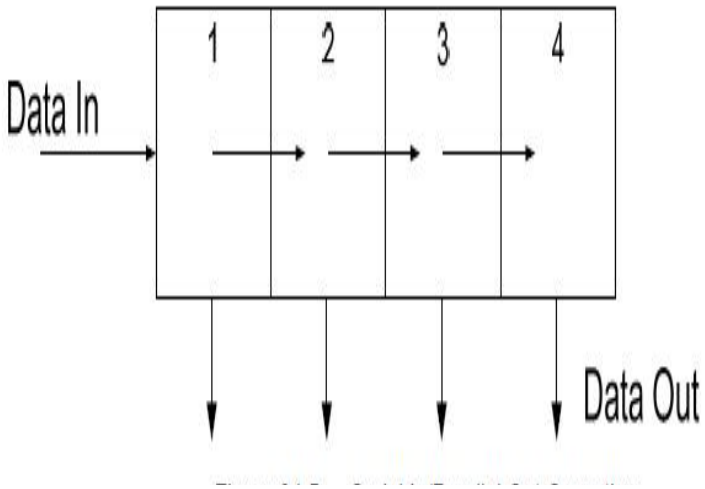


Fig. 5: Serial In/Parallel Out Operation

Parallel In/Serial Out Operation

The register has parallel inputs, data bits are loaded into the register in parallel by activating a load signal. The data is shifted out serially by application of clock signals. Thus, in a 4-bit shift register, after 4 clock signals the 4-bit data is completely shifted out of the shift register.

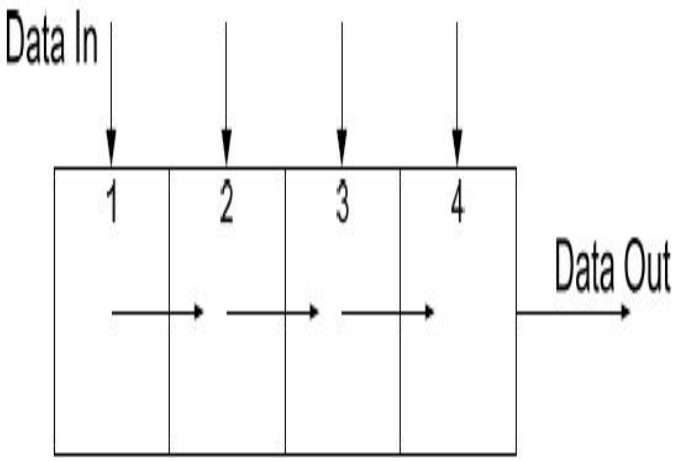


Fig. 6: Parallel In/Serial Out Operation

The internal circuit of a 4-bit Parallel In/Serial Out Shift register is shown. The 4-bit data is initially loaded in Parallel into the shift register by setting the SHIFT /LOAD input to logic low. The AND gates marked 2 are enabled allowing data to be applied at the inputs of the respective D flip-flops. On a positive clock transition the data is latched by the

respective flip-flops. To shift the data, the SHIFT /LOAD is set to logic high which enables AND gates marked 1 connecting the Q outputs of the each flip-flop connected to the D input of the next flip-flop.

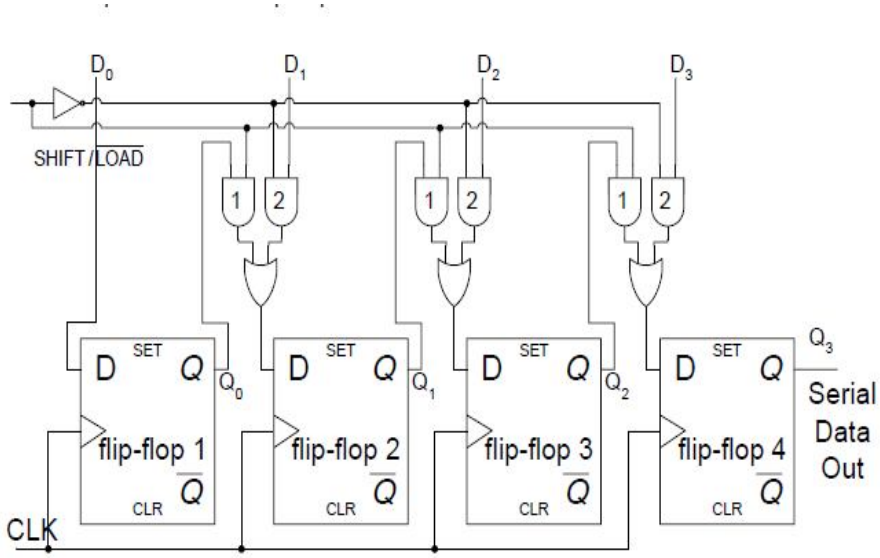


Fig. 7: 4-Bit Parallel In/Serial Out Shift Register

Parallel In/Parallel Out Operation

The register has parallel inputs and parallel outputs. Data is entered in parallel by applying a single clock pulse. Data is latched by the flip-flops on the clock transition and is available in parallel form at the flip-flop outputs.

The internal circuit of 4-bit Parallel In/Parallel Out Register is shown. The Parallel In/Parallel Out register stores Parallel data and usually does not allows any shift operations.

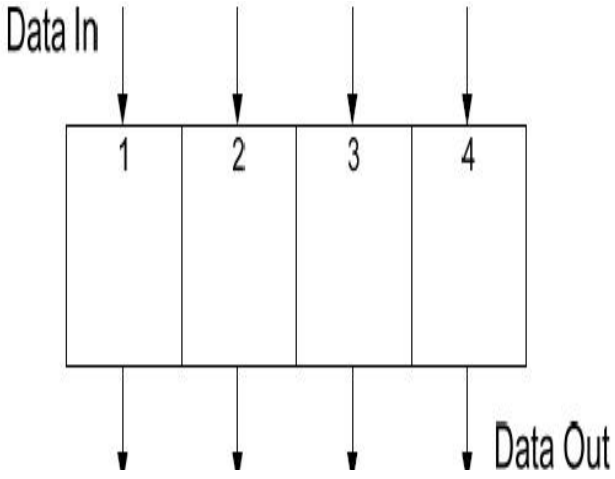


Fig. 8: Parallel In/Parallel Out Operation

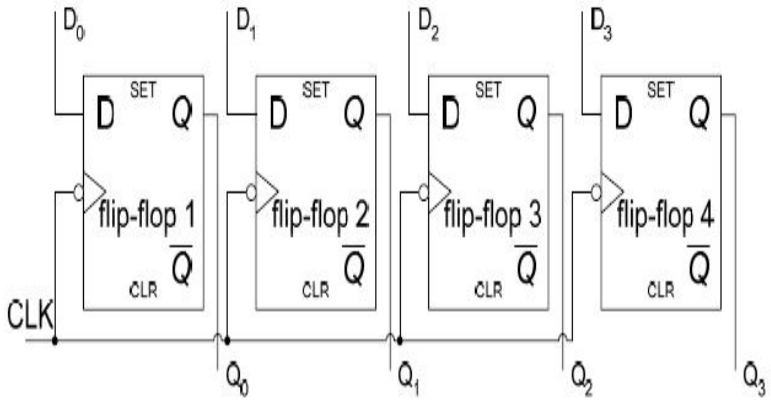


Fig. 9: A D-Flip-Flop based 4-Bit Parallel In/Parallel Out Register

Rotate Right Operation

The serial output of the register is connected to the serial input of the register. By applying clock pulses data is shifted right. The data shifted out of the serial out pin at the right hand side is re-circulated back into the shift register input at the left hand side. Thus, the data is rotated right within the register.

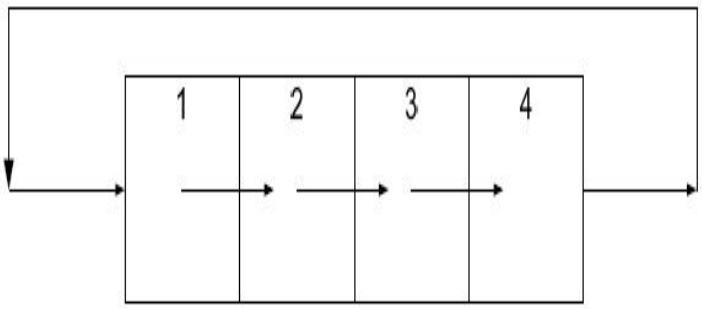


Fig. 10: Rotate Right Operation

Rotate Left Operation

The serial output of the register is connected to the serial input of the register. By applying clock pulses data is shifted left. The data shifted out of the serial out pin at the left hand side is re-circulated back into the shift register input at the right hand side. Thus, the data is rotated left within the register.

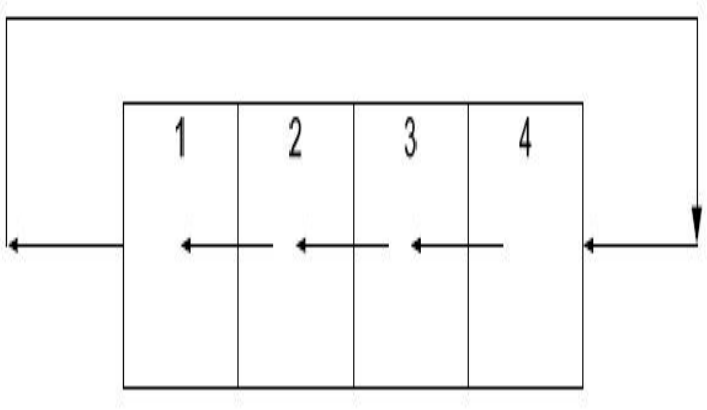


Fig. 11: Rotate Left Operation

3.1.2 Shift Register Counters

Shift register counters are basically, shift registers connected to perform rotate left and rotate right operations. When data is rotated through a register counter a specific sequence of states is repeated. Two commonly used register counters in digital logic are the Johnson Counter and the Ring Counter.

Johnson Counter

In a Johnson counter, the Q output of the last flip-flop of the shift register is connected to the data input of the first flip-flop. The circuit of a 4-bit, D flip-flop based Johnson Counter is shown in the figure below. The sequence of states that are implemented by a n-bit Johnson counter are 2n. Thus, a 4-bit Johnson counter sequences through 8 states and a 5-bit Johnson counter sequences through 10 states.

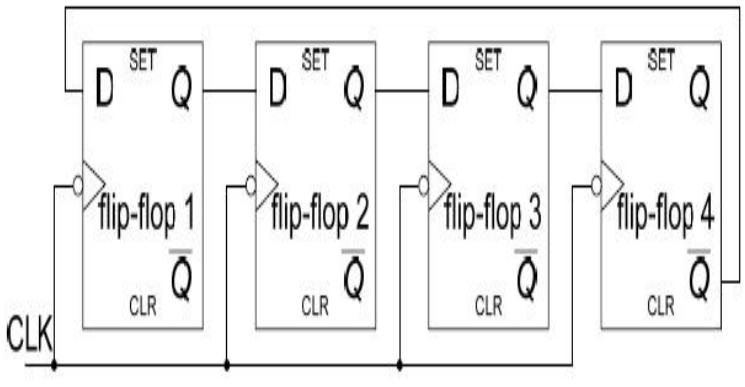


Fig. 12: 4-Bit Johnson Counter

Table 1: Sequence of States of a 4-Bit Johnson Counter

Clock Pulse	Q_0	Q_1	Q_2	Q_3
0	0	0	0	0
1	1	0	0	0
2	1	1	0	0
3	1	1	1	0
4	1	1	1	1
5	0	1	1	1
6	0	0	1	1
7	0	0	0	1

Ring Counter

The Ring Counter is similar to the Johnson Counter, except that the Q output of the last flip-flop of the shift register is connected to the data input of the first flip-flop of the shift register. All the flip-flops of the counter are cleared to logic low except for the first flip-flop which is preset to logic high.

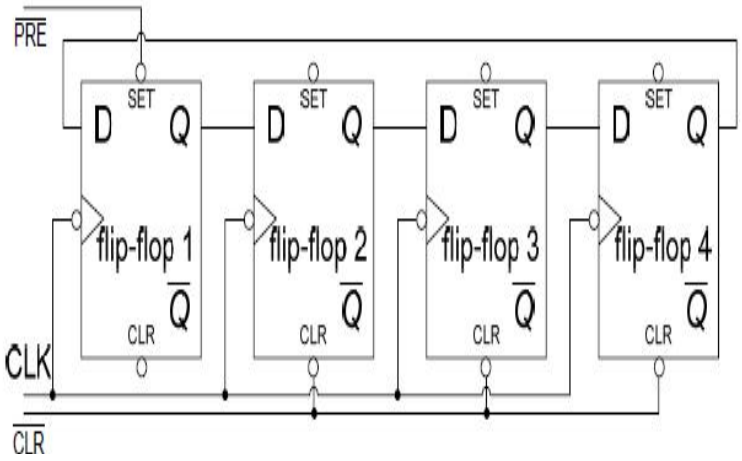


Fig. 13: 4-Bit Ring Counter

After the initialisation of the counter, the logic high set at the output of the first flip-flop is shifted right at each clock transition. With a Ring Counter circuit no decoding gates are required. Each state of the ring counter has a unique output.

Table 2: Sequence of States of a 4-Bit Ring Counter

Clock Pulse	Q_0	Q_1	Q_2	Q_3
0	1	0	0	0
1	0	1	0	0
2	0	0	1	0
3	0	0	0	1

3.1.3 Applications of Shift Registers

The major application of a shift register is to convert between parallel and serial data. Shift registers are also used as keyboard encoders. The two applications of the shift registers are discussed.

Serial-to-Parallel Converter

Earlier, Multiplexer and Demultiplexer based Parallel to Serial and Serial to Parallel converters were discussed. The Multiplexer and Demultiplexer require registers to store the parallel data that is converted into serial data and parallel data which is obtained after converting the incoming serial data. A Parallel In/Serial Out shift register offers a better solution instead of using a Multiplexer-Register combination to convert parallel data into serial data. Similarly, a Serial In/Parallel Out shift register replaces a Demultiplexer-Register combination.

In Asynchronous Serial data transmission mode, a character which is constituted of 8-bits (which can include a parity bit) is transmitted. To separate one character from another and to indicate when data is being transmitted and when the serial transmission line is idle (no data is being transmitted) a set of start bit and stop bits are appended at both ends of the 8-bit character. A character is preceded by a logic low start bit. When the line is idle it is set to logic high, when a character is about to be transmitted the start bit sets the line to logic low. The logic low start bit is an indication that 8 character bits are to follow and the transmission line is no longer in an idle state. After 8-character bits have been transmitted, the end of the character is indicated by two stop bits that are at logic high. The two logic bits indicate the end of the character and also set the transmission line to the idle state.

Therefore, a total of 11 bits are transmitted to send one character from one end to the other. The logic low start bit is also a signal for the receiver circuit to start receiving the 8 character bits that are following the start bit. The 11-bit serial character format is shown.

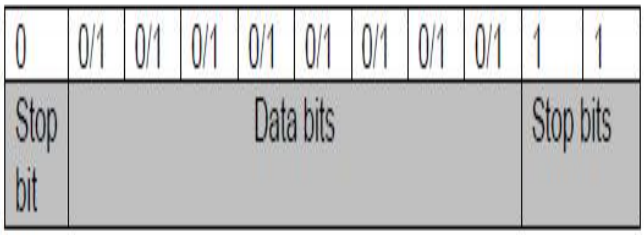


Fig. 14: 11-Bit Serial Data Format

A Serial to Parallel converter circuit based on shift registers is shown below. The serial data is preceded by a logic low start bit which triggers the J-K flip-flop. The output of the flip-flop is set to logic high which enables the clock generator. The clock pulses generated are connected to the clock input of a Serial In/Parallel Out shift register and also to the clock input of an 8-bit counter. On each clock transition, the Serial In/Parallel Out shift register shifts in one bit data. When the 8-bit counter reaches its terminal count 111, the terminal count output signal along with the clock signal trigger the One-Shot and also allow the Parallel In/Parallel Out register to latch in the Parallel data at the output of the Serial In/Parallel Out shift register. The One-shot resets the J-K flip-flop output Q to logic 0 disabling the clock generator and also clears the 8-bit counter to count 000.

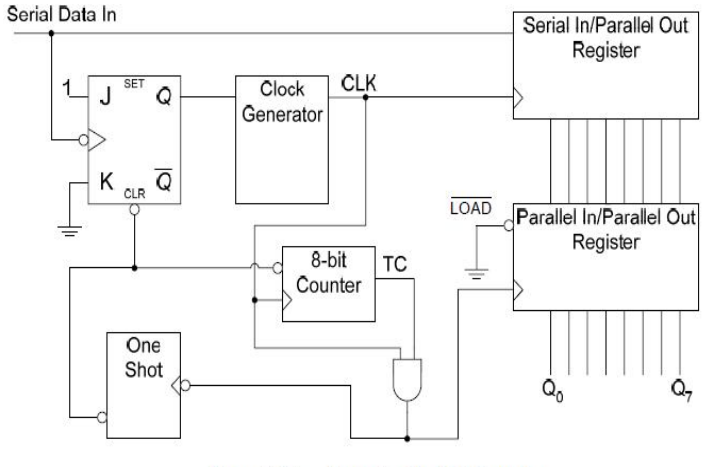


Fig. 15: Series-to-Parallel Converter

Keyboard Encoder

We have a simple keypad encoder circuit where the 0 to 9 digit keypad was connected through a decade to BCD encoder. Pressing any keypad key enables the corresponding input of the encoder circuit which encodes the input as a 4-bit BCD output.

Computer keyboards which have more keys employ a keyboard encoder circuit that regularly scans the keyboard to check for any key press. The scanning is done by organising the keys in the form of rows and columns. With the help of a shift register based ring counter one row is selected at a time. The two counters are connected as an 8-bit Ring counter which sequences through a bit pattern having all 1's and a single 0. The 8 state sequence selects one row at a time by setting it to logic 0. If a key is pressed, the corresponding column also becomes logic 0 as it connected to the selected row. The row and column which are selected are encoded by the row and column encoders. When a key is pressed, the selected column which is set to logic 0 sets the output of the NAND gate to logic 1 which triggers two One Shots. The first One Shot inhibits the clock signal to the ring counters for a short interval until the Key Code is stored. The One Shot also triggers the second One-Shot that sends a pulse to the clock input of the Key Code register. The Key Code Register stores the key ID represented as 3-bit column and 3-bit row code.

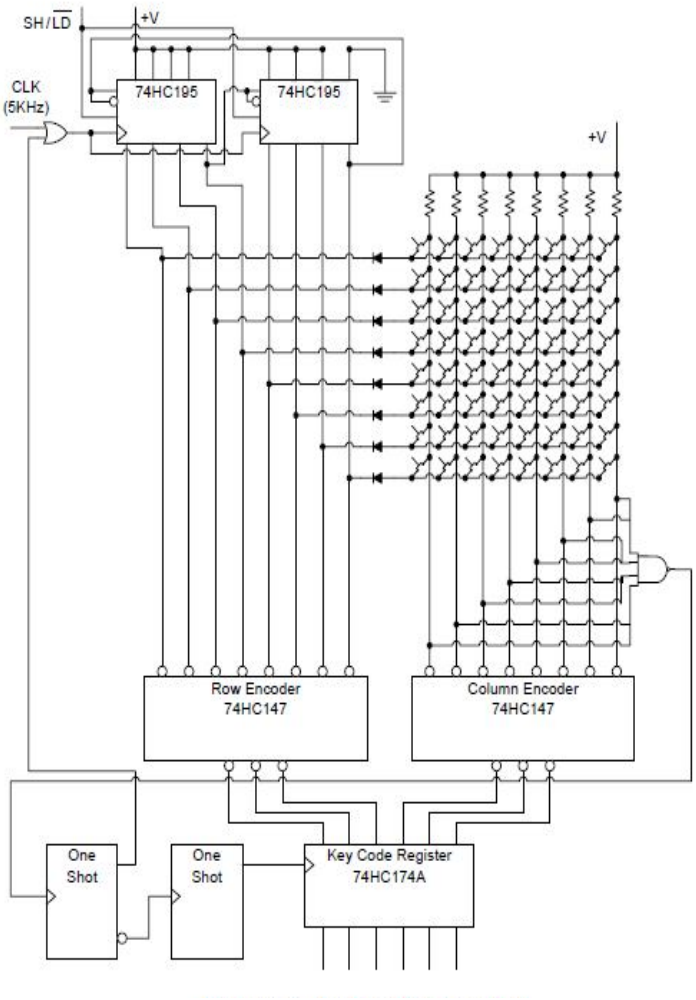


Fig. 16: Keyboard Encoder Circuit

4.0 CONCLUSION

A register performs two basic functions. It stores data and it moves or shifts data. The shifting of data involves shifting of bits from one flip-flop to the other within the register or moving data in and out of the register.

The different shift operations are:

1. Serial In/Shift Right/Serial Out Operation
2. Serial In/Shift Left/Serial Out Operation
3. Serial In/Parallel Out Operation
4. Parallel In/Serial Out Operation
5. Parallel In/Parallel Out Operation
6. Rotate Right Operation
7. Rotate Left Operation.

Shift register counters are shift registers connected to perform rotate left and rotate right operations. When data is rotated through a register counter a specific sequence of states is repeated. Two commonly used register counters in digital logic are the Johnson Counter and the Ring Counter.

The applications of a shift register are used to convert between parallel and serial data. They are also used as keyboard encoders.

SELF-ASSESSMENT EXERCISE

Give a detailed explanation of the different shift operations you know.

5.0 SUMMARY

In this unit we discussed registers. Information from an input device or from the computer's memory is communicated via the bus to the central processing unit (CPU), which is the part of the computer that translates commands and runs programs. The CPU is a microprocessor chip—that is, a single piece of silicon containing millions of tiny, microscopically wired electrical components. Information is stored in a CPU memory location called a register. Registers can be thought of as the CPU's tiny scratchpad, temporarily storing instructions or data.

6.0 TUTOR-MARKED ASSIGNMENT

State in not less than four pages what you know about registers.

7.0 REFERENCES/FURTHER READING

Comer, J. David (1994). *Digital Logic & State Machine Design*. Oxford University Press.

Dandamudi, S. (2003). *Fundamentals of Computer Organization and Design*. Springer.

Hayes, P. John (1993). *Introduction to Digital Logic Design*. Prentice Hall.

Hennessy, L. John & Patterson, A. David (2008). *Computer Organization & Design*. Morgan Kaufmann.

Holdsworth, Brian & Woods, Clive (2002). *Digital Logic Design*. Newnes.

Mano, M. Morris & Kime, Charles, R. (2007). *Logic and Computer Design Fundamentals*. Prentice Hall. Pg 283.

Marcovitz, Alan (2009). *Introduction to Logic Design*. McGraw-Hill.

Neamen, Donald (2009). *Microelectronics: Circuit Analysis & Design*. McGraw-Hill.

Nelson, P. Victor *et al.* (1995). *Digital Logic Circuit Analysis & Design*. Prentice Hall.

Palmer, James & Periman, David (1993). *Schaum's Outline of Introduction to Digital Systems*. McGraw-Hill.

Patterson, David & Hennessy, John (1993). *Computer Organization & Design: The Hardware/Software Interface*. Morgan Kaufmann Publishers.

Rafiquzzaman, M. (2005). *Fundamentals of Digital Logic & Microcomputer Design*. Wiley – Interscience.

Roth, H. Charles Jr & Kinney, L. Larry (2009). *Fundamentals of Logic Design*.

Tanenbaum, Andrew S. (1998). *Structured Computer Organization.* Prentice Hall.

Tocci, J. Ronald & Widmer, S. Neal (1994). *Digital Systems: Principles and Applications.*

Tokheium, Roger (1994). *Schaum's Outline of Digital Principles*. McGraw-Hill.

Wakerly, F. John (2005). *Digital Design: Principles & Practices Package*. Prentice Hall.

UNIT 4 FINITE STATE MACHINES

CONTENTS

1.0 INTRODUCTION

This unit introduces finite state machines, a primitive, but useful computational model for both hardware and certain types of software. We also discuss regular expressions, the correspondence between non-deterministic and deterministic machines, and more on grammars. Finally, we describe typical hardware components that are essentially physical realisations of finite-state machines.

2.0 OBJECTIVES

At the end of this unit, you should be able to:

- discuss finite state machines
- define hardware
- distinguish between a Mealy or Moore machine.

3.0 MAIN CONTENT

3.1 Finite State Machines

Finite state machines provide a simple computational model with many applications. Recall the definition of a Turing machine: a finite-state controller with a movable read/write head on an unbounded storage tape. If we restrict the head to move in only one direction, we have the

general case of a finite-state machine. The sequence of symbols being read can be thought to constitute the input, while the sequence of symbols being written could be thought to constitute the output. We can also derive output by looking at the internal state of the controller after the input has been read. Finite-state machines, also called *finite-state automata* (singular: *automaton*) or just finite *automata* are much more restrictive in their capabilities than Turing machines. The basic operation of a Finite State Machine system is this: as the system is in one of the defined states at any instant of time, it will *react* to specified (external) inputs or (internal) conditions with specified *actions*, and transition to another defined state, or remain in its current state, depending on the design. For example, we can show that it is not possible for a finite-state machine to determine whether the input consists of a prime number of symbols. Much simpler languages, such as the sequences of well-balanced parenthesis strings, also cannot be recognised by finite-state machines. Still there are the following applications:

- Simple forms of pattern matching (precisely the patterns definable by "regular expressions").
- Models for sequential logic circuits, of the kind on which every present-day computer and many device controllers is based.
- An intimate relationship with directed graphs having arcs labeled with symbols from the input alphabet.

Even though each of these models can be depicted in a different setting, they have a common mathematical basis.

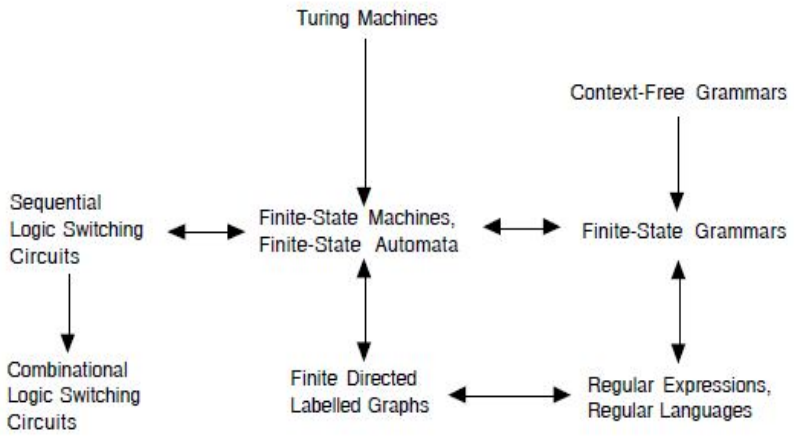


Fig. 1: The Interrelationship of Various Models with Respect to Computational or Representational Power

(The arrows move in the direction of restricting power. The bi-directional arrows show equivalences).

Finite State Machines are generally depicted as a state diagram, represented graphically with two symbols: the **state bubble** and the **transition arrow**. States are labeled or numbered and both inputs and outputs are described textually. At any instant of time, a state machine is in its *current state*. Depending on the specified input events and conditions, a transition to the *next state* will occur. Finite State Machines are considered *deterministic* if all transitions to next states are unique to a given state and its inputs. Most useful FSMs are fully deterministic, making them ideal for embedded systems software and the process of validation and verification.

It is common to distinguish Finite State Machines as either a Mealy or Moore machine. The difference between the two machines, as we shall explore later, are noteworthy, but not necessarily paramount to successful Finite State Machine design for software application architecture, but likely important in sequential circuit design.
Finite State Machines are defined as sharing the following characteristics:

- a finite set of defined states, one of which being defined as the *initial state* of the machine
- a set of defined inputs
- a set of defined outputs
- a set of transitions between selected states, and
- the machine is said to be in a single state at any instant of time

3.2 Finite State Machines as Restrictive Turing Machines

One way to view the finite-state machine model as a more restrictive Turing machine is to separate the input and output halves of the tapes, as shown below.

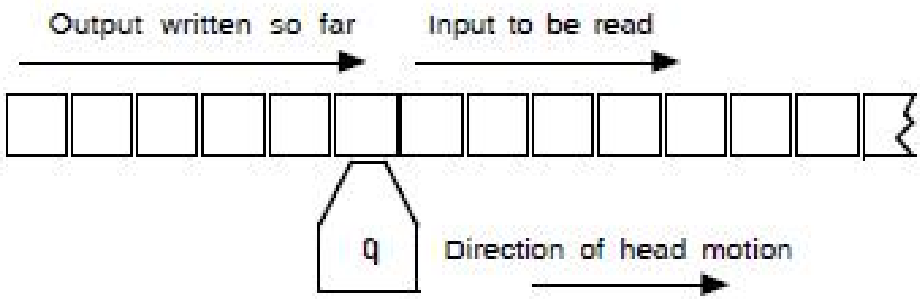


Fig. 2: Finite State Machine as a One-Way Moving Turing Machine

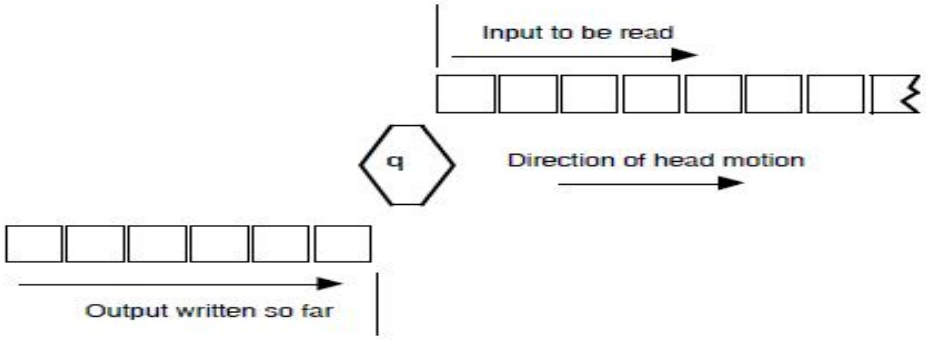


Fig. 3: Finite-State Machine as Viewed with Separate Input and Output

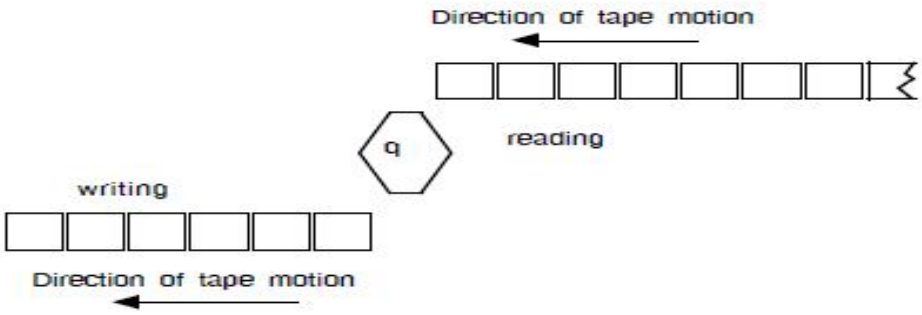


Fig. 4: Finite-State Machine Viewed as a Stationary-Head, Moving-Tape, Device

Since the motion of the head over the tape is strictly one-way, we can abstract away the idea of a tape and just refer to the input *sequence* read and the *output* sequence produced, as suggested in the next diagram. A machine of this form is called a ***transducer***, since it maps input sequences to output sequences. The term *Mealy machine*, after George H. Mealy (1965), is also often used for transducer.

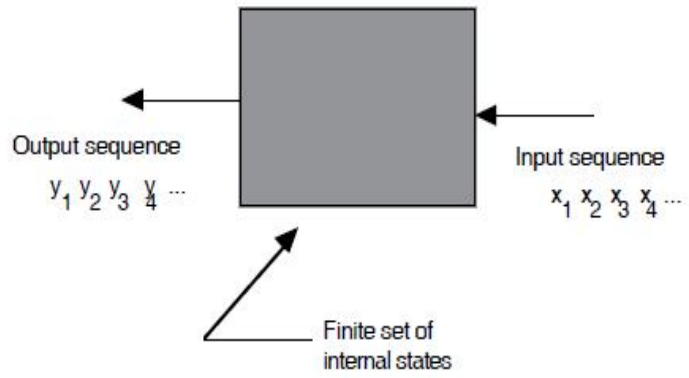


Fig. 5: A Transducer Finite-State Machine Viewed as a Tapeless "Black Box" Processing an Input Sequence to Produce an Output Sequence

On the other hand, occasionally, we are not interested in the sequence of outputs produced but just an output associated with the current state of the machine. This simpler model is called a *classifier*, or *Moore machine*, after E.F. Moore (1965).

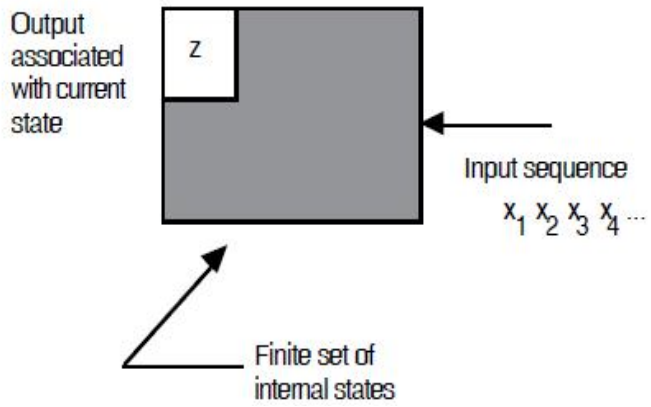


Fig. 6: Classifier Finite State Machine. Output is a function of current state, rather than being a Sequence

For a Mealy machine, the transition will be associated with an output. For a Moore machine, the output occurs within the next state. For this reason, Finite State Machines are often called *'reactive'* systems. Inputs are also called events. Typical events may be: a message received from another state machine, a simple event flag set by another state machine, or the expiration of a time interval. Likewise, outputs may be: sending a message to another state machine, setting an event flag for another state machine to respond to, or starting a timed interval. Also, multiple unique transitions are allowed from one state to other defined states. For software FSMs, each state will have its unique source code logic to process events, perform actions and output, and to effect state transitions. A complete system may be comprised of one or more Finite State Machines, as determined by the partitioning process performed during the initial design or analysis. Although the Mealy Machine model may be more flexible than Moore Machine, it is the need of the system being analysed or designed that determines which of the two is most suitable. Note that a given state machine may be comprised of both Mealy and Moore models, if such a design meets functional requirements of the system. Also, be aware that both Mealy and Moore Machines can be logically converted to the other. The point here is that the correct state logic required for efficient operation is what's important; the resulting machine archetype (Mealy or Moore) should be only a secondary observation.

3.3 Modeling the Behaviour of Finite State Machines

Concentrating initially on transducers, there are several different notations we can use to capture the behaviour of finite-state machines:

- as a functional program mapping one list into another
- as a restricted imperative program, reading input a single character at a time and producing output a single character at a time
- as a feedback system
- representation of functions as a table
- representation of functions by a directed labeled graph.

For concreteness, we shall use the sequence-to-sequence model of the machine, although the other models can be represented similarly. Let us give an example that we can use to show the different notations:

Example: An Edge-Detector

The function of an edge detector is to detect transitions between two symbols in the input sequence, say 0 and 1. It does this by outputting 0 as long as the most recent input symbol is the same as the previous one. However, when the most recent one differs from the previous one, it

outputs a 1. By convention, the edge detector always outputs 0 after reading the very first symbol. Thus, we have the following input output sequence pairs for the edge-detector, among an infinite number of possible pairs:

input	output
0	0
00	00
01	01
011	010
0111	0100
01110	01001
1	0
10	01
101	011
1010	0111
10100	01110
etc.	

3.4 Functional Program View of Finite State Machines

In this view, the behaviour of a machine is as a function from lists to lists.

Each state of the machine is identified with such a function.

The initial state is identified with the overall function of the machine.
The functions are interrelated by mutual recursion: when a function processes an input symbol, it calls another function to process the remaining input.

Each function: looks at its input by one application of *first*, produces an output by one application of *cons*, the first argument of which is determined purely by the input obtained from first, and calls another function (or itself) on rest of the *input.*

We make the assumptions that:
The result of cons, in particular the first argument, becomes partially available even before its second argument is computed.

Each function will return NIL if the input list is NIL, and we do not show this explicitly.

Functional code example for the edge-detector

We will use three functions, f, g, and h. The function f is the overall representation of the edge detector.

```
f([0 | Rest]) => [0 | g(Rest)];
f([1 | Rest]) => [0 | h(Rest)];
f([]) => [];
g([0 | Rest]) => [0 | g(Rest)];
g([1 | Rest]) => [1 | h(Rest)];
g([]) => [];
h([0 | Rest]) => [1 | g(Rest)];
h([1 | Rest]) => [0 | h(Rest)];
h([]) => [];
```

Notice that f is never called after its initial use. Its only purpose is to provide the proper output (namely 0) for the first symbol in the input.

Example of f applied to a specific input:

```
f([0, 1, 1, 1, 0]) ==> [0, 1, 0, 0, 1]
```

An alternative representation is to use a single function, say k, with an extra argument, treated as just a symbol. This argument represents the *name* of the function that would have been called in the original representation. The top-level call to k will give the initial state as this argument:

```
k("f", [0 | Rest]) => [0 | k("g", Rest)];
k("f", [1 | Rest]) => [0 | k("h", Rest)];
k("f", []) => [];
k("g", [0 | Rest]) => [0 | k("g", Rest)];
k("g", [1 | Rest]) => [1 | k("h", Rest)];
k("g", []) => [];
k("h", [0 | Rest]) => [1 | k("g", Rest)];
k("h", [1 | Rest]) => [0 | k("h", Rest)];
k("h", []) => [];
```

The top level call with input sequence x is k ("f", x) since "f" is the initial state.

3.5 Imperative Program View of Finite State Machines

In this view, the input and output are viewed as streams of characters. The program repeats the processing cycle:

read character,
select next state,
write character,
go to next state

ad infinitum. The states can be represented as separate "functions", as in the functional view, or just as the value of one or more variables. However the allowable values must be restricted to a finite set. No stacks or other extendible structures can be used, and any arithmetic must be restricted to a finite range.

The following is a transliteration of the previous program to this view. The program is started by calling f(). Here we assume that read() is a method that returns the next character in the input stream and write(c) writes character c to the output stream.

```
void f() // initial function
{
switch( read() )
{
case '0': write('0'); g(); break;
case '1': write('0'); h(); break;
}
Finite-State Machines 477
}
void g() // previous input was 0
{
switch( read() )
{
case '0': write('0'); g(); break;
case '1': write('1'); h(); break; // 0 -> 1 transition
}
}
void h() // previous input was 1
{
switch( read() )
{
case '0': write('1'); g(); break; // 1 -> 0 transition
case '1': write('0'); h(); break;
}
}
```

[Note that this is a case where all calls can be "tail recursive", i.e. could be implemented as *goto*s by a smart compiler.]

The same task could be accomplished by eliminating the functions and using a single variable to record the current state, as shown in the following program. As before, we assume read() returns the next character in the input stream and write(c) writes character c to the output stream.

```
static final char f = 'f'; // set of states
static final char g = 'g';
static final char h = 'h';
static final char initial_state = f;
main()
{
char current_state, next_state;
char c;
current_state = initial_state;
while( (c = read()) != EOF )
{
switch( current_state )
{
case f: // initial state
switch( c )
{
case '0': write('0'); next_state = g; break;
case '1': write('0'); next_state = h; break;
}
break;
case g: // last input was 0
switch( c )
{
case '0': write('0'); next_state = g; break;
case '1': write('1'); next_state = h; break; // 0 -> 1
}
break;
case h: // last input was 1
switch( c )
{
case '0': write('1'); next_state = g; break; // 1 -> 0
case '1': write('0'); next_state = h; break;
}
break;
}
current_state = next_state;
}
}
```

3.6 Feedback System View of Finite State Machines

The feedback system view abstracts the functionality of a machine into two functions, the next-state or state-transition function F, and the output function G.

F: States x Symbols States state-transition function
G: States x Symbols Symbols output function

The meaning of these functions is as follows:

F(q, σ) is the state to which the machine goes when currently in state q and σ is read

G(q, σ) is the output produced when the machine is currently in state q and σ is read

The relationship of these functions is expressible by the following diagram.

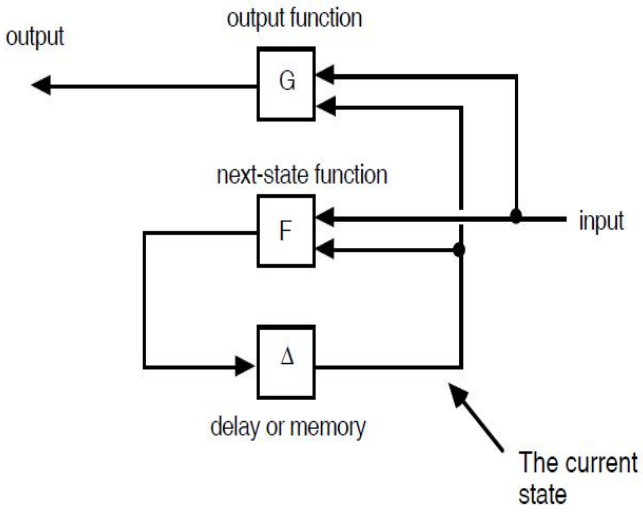


Fig. 7: Feedback Diagram of Finite State Machine Structure

From F and G, we can form two useful functions

F*: States x Symbols* States extended state-transition function
G*: States x Symbols* Symbols extended output function where Symbols* denotes the set of all *sequences* of symbols. This is done by induction:

$F^*(q, \lambda) = q$
$F^*(q, x\sigma) = F(F^*(q, x), \sigma)$
$G^*(q, \lambda) = \lambda$
$G^*(q, x\sigma) = G^*(q, x)\, G(F^*(q, x), \sigma)$

In the last equation, juxtaposition is like cons'ing on the right. In other words, $F^*(q, x)$ is the state of the machine after all symbols in the sequence x have been processed, whereas $G^*(q, x)$ is the sequence of outputs produced along the way. In essence, G^* can be regarded as the *function computed by* a transducer. These definitions could be transcribed into rex rules by representing the sequence $x\sigma$ as a list $[\sigma \mid x]$ with λ corresponding to [ ].

3.7 Tabular Description of Finite State Machines

This description is similar to the one used for Turing machines, except that the motion is left unspecified, since it is implicitly one direction. In lieu of the two functions F and G, a finite-state machine could be specified by a single function combining F and G of the form:

States x Symbols ⟶ States x Symbols analogous to the case of a Turing machine, where we included the motion:

States x Symbols ⟶ Symbols x Motions x States

These functions can also be represented succinctly by a table of 4-tuples, similar to what we used for a Turing machine, and again called a *state transition table*:

State1, *Symbol1*, *State2*, *Symbol2*
Such a 4-tuple means the following:

If the machine's control is in *State1* and reads *Symbol1*, then machine will write *Symbol2* and the next state of the controller will be *State2*.
The state-transition table for the edge-detector machine is:

Table 1: State Transition Table for the Edge-Detector Machine

	current state	input symbol	next state	output symbol
start state	f	0	g	0
	f	1	h	0
	g	0	g	0
	g	1	h	1
	h	0	g	1
	h	1	h	0

Unlike the case of Turing machines, there is **no particular halting convention**. Instead, the machine is always read to proceed from whatever current state it is in. This does not stop us from assigning our own particular meaning of a symbol to designate, for example, end-of-input.

3.8 Classifiers, Acceptors, Transducers, and Sequencers

In some problems we don't care about generating an entire sequence of output symbols as do the **transducers** discussed previously. Instead, we are only interested in categorizing each input sequence into one of a finite set of possibilities. Often these possibilities can be made to derive from the current state.

So we attach the result of the computation to the state, rather than generate a sequence. In this model, we have an output functionc:
Q ⟶ C which gives a category or class for each state. We call this type of machine a **classifier** or **controller**. In the simplest non-trivial case of classifier, there are two categories. The states are divided up into the "accepting" states (class 1, say) and the "rejecting" states (class 0). The machine in this case is called an **acceptor** or **recogniser**.

The sequences it accepts are those given by c(F*(q0, x)) = 1 that is, the sequences *x* such that, when started in state q0, after reading x, the machine is in a state q such that c(q) = 1. The set of all such x, since it is a set of strings, is a **language**. If A designates a finite-state acceptor, then L(A) = { x in Σ* | c(F*(q0, x)) = 1} is the **language accepted by** A.

The structure of a classifier is simpler than that of a transducer, since the output is only a function of the state and not of both the state and input. The structure is shown as follows:

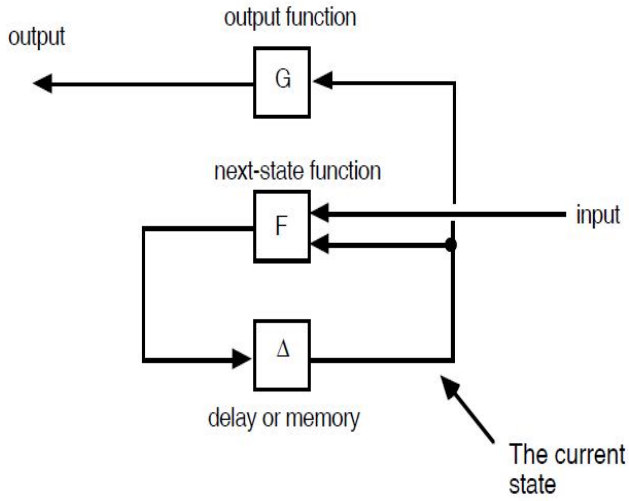


Fig. 8: Feedback Diagram of Classifier Finite State Machine Structure

A final class of machine, called a **sequencer** or **generator**, is a special case of a transducer or classifier that has a single-letter input alphabet. Since the input symbols are unchanging, this machine generates a fixed sequence, interpreted as either the output sequence of a transducer or the sequence of classifier outputs. An example of a sequencer is a MIDI (Musical Instrument Digital Interface) sequencer, used to drive electronic musical instruments. The output alphabet of a MIDI sequencer is a set of 16-bit words, each having a special interpretation as pitch, note start and stop, amplitude, etc. Although most MIDI sequencers are programmable, the program typically is of the nature of an initial setup rather than a sequential input.

3.9 Description of Finite State Machines using Graphs

Any finite state machine can be shown as a graph with a finite set of nodes. The nodes correspond to the states. There is no other memory implied other than the state shown. The start state is designated with an arrow directed into the corresponding node, but otherwise unconnected.

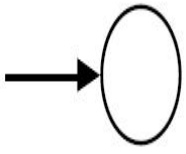


Fig. 9: An Unconnected In-Going Arc Indicates that the Node is the Start State

The arcs and nodes are labeled differently, depending on whether we are representing a transducer, a classifier, or an acceptor. In the case of a **transducer**, the arcs are labeled σ/δ as shown below, where σ is the input symbol and δ is the output symbol. The state transition is designated by virtue of the arrow going from one node to another.

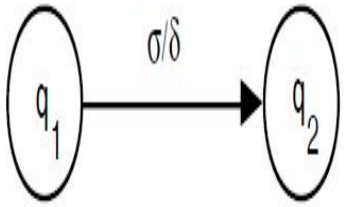


Fig. 10: Transducer Transition from q1 to q2, based on Input σ, Giving Output δ

In the case of a **classifier**, the arrow is labeled only with the input symbol. The categories are attached to the names of the states after /.

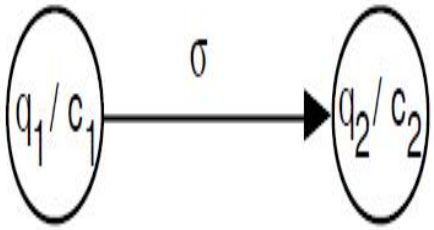


Fig. 11: Classifier Transition from q1 to q2, based on Input σ

In the case of an **acceptor**, instead of labeling the states with categories 0 and 1, we sometimes use a double-lined node for an accepting state and a single-lined node for a rejecting state.

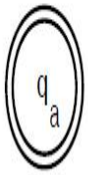


Fig. 12: Acceptor, an Accepting State

Transducer Example

The edge detector is an example of a transducer. Here is its graph:

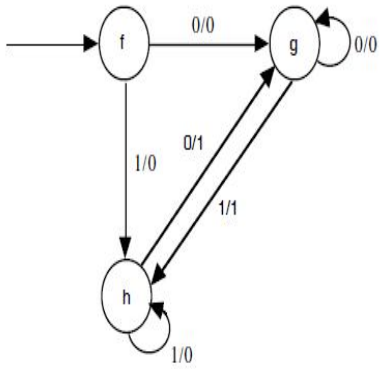


Fig. 13: Directed Labeled Graph for the Edge Detector

Let us also give examples of classifiers and acceptors, building on this example.

Classifier Example

Suppose we wish to categorise the input as to whether the input so far contains 0, 1, or more than 1 "edges" (transitions from 0 to 1, or 1 to 0). The appropriate machine type is classifier, with the outputs being in the set {0, 1, more}. The name "more" is chosen arbitrarily. We can sketch how this classifier works with the aid of a graph. The construction begins with the start state. We don't know how many states there will be initially. Let us use a, b, c, ... as the names of the states, with *a* as the start state. Each state is labeled with the corresponding class as we go. The idea is to achieve a finite closure after some number of states have been added. The result is shown below:

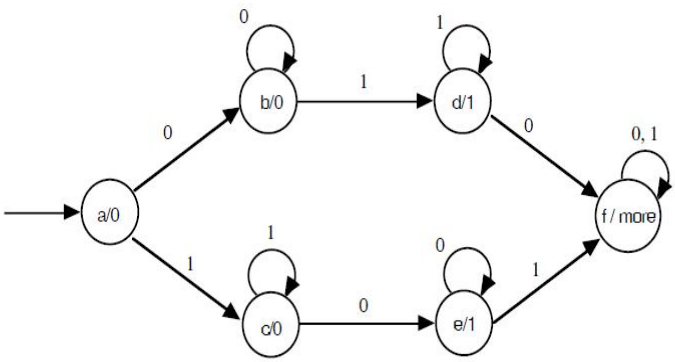


Fig. 14: Classifier for Counting 0, 1, or more than 1 Edges

Acceptor Example

Let us give an acceptor that accepts those strings with exactly one edge. We can use the state transitions from the previous classifier. We need only designate those states that categorise there being one edge as accepting states and the others as rejecting states.

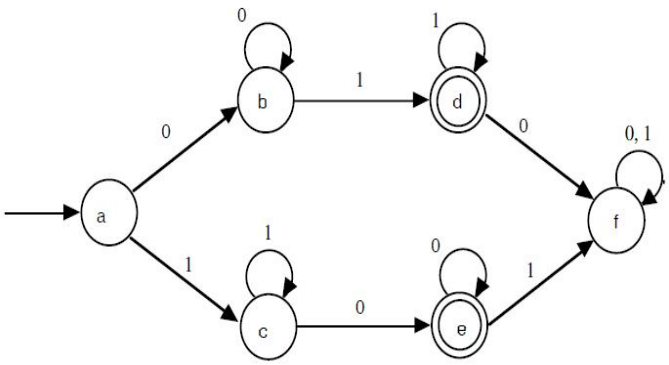


Fig. 15: Acceptor for Strings with Exactly One Edge. Accepting States are d and e

Sequencer Example

The following sequencer, where the sequence is that of the outputs associated with each state, is that for a naive traffic light:

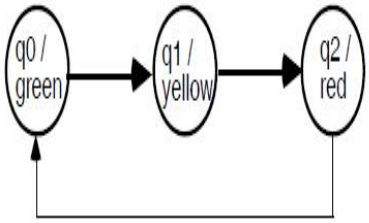


Fig. 16: A Traffic Light Sequencer

Inter-Convertibility of Transducers and Classifiers

We can describe a relationship between classifiers and transducers, so thatmost of the theory developed for one will be applicable to the other. One possible connection is, given an input sequence x, record the outputs corresponding to the states through which a classifier goes in processing x. Those outputs could be the outputs of an appropriately-defined transducer. However, classifiers are a little more general in this sense, since they give output even for the empty sequence λ, whereas the

output for a transducer with input λ is always just λ. Let us work in terms of the following equivalence:

A transducer T started in state q0 is equivalent to a classifier C started in state q0 if, for any non-empty sequence x, the sequence of outputs emitted by T is the same as the sequence of outputs of the states through which C passes.

With this definition in mind, the following would be a classifier equivalent to the edge detector transducer.

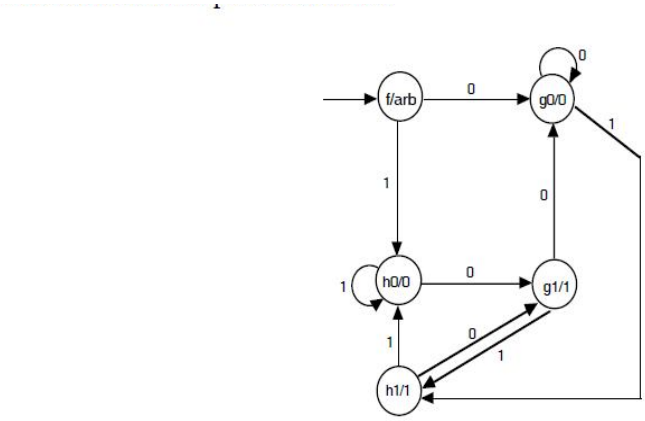


Fig. 17: A Classifier Formally Equivalent to the Edge-Detector Transducer

To see how we constructed this classifier, observe that the output emitted by a transducer in going from a state q to a state q', given an input symbol σ, should be the same as the output attached to state q' in the classifier. However, we can't be sure that all transitions into a state q' of a transducer produce the same output. For example, there are two transitions to state *g* in the edge-detector that produce 0 and one that produces 1, and similarly for state h. This makes it impossible to attach a fixed input to either g or h. Therefore we need to "split" the states g and h into two, a version with 0 output and a version with 1 output. Call these resulting states g0, g1, h0, h1. Now we can construct an output-consistent classifier from the transducer. We don't need to split f, since it has a very transient character. Its output can be assigned arbitrarily without spoiling the equivalence of the two machines. The procedure for converting a classifier to a transducer is simpler. When the classifier goes from state q to q', we assign to the output transition the state output value c(q'). The following diagram shows a transducer equivalent to the classifier that reports 0, 1, or more edges.

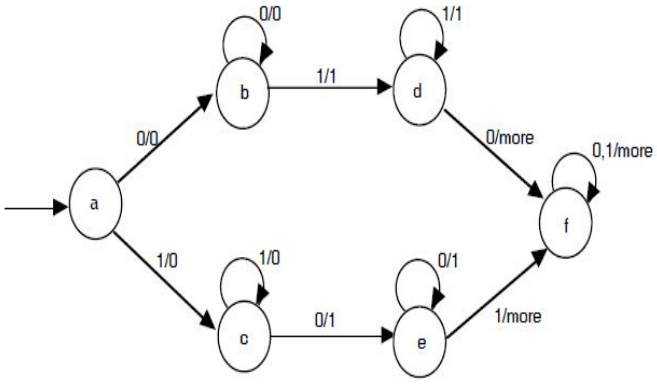


Fig. 18: A Transducer Formally Equivalent to the Edge-Counting Classifier

SELF-ASSESSMENT EXERCISE

Write short notes with the aid of diagrams on the following terms: (a) classifiers (b) acceptors (c) transducers (d) sequencers.

4.0 CONCLUSION

Finite State Machines provide a simple computational model with many applications. The basic operation of a Finite State Machine system is this: as the system is in one of the defined states at any instant of time, it will *react* to specified (external) inputs or (internal) conditions with specified *actions*, and transition to another defined state, or remain in its current state, depending on the design. State Machines are generally depicted as a State Diagram, represented graphically with two symbols: the state bubble and the transition arrow.

It is common to distinguish Finite State Machines as either a Mealy or Moore machine. Finite State Machines are defined as sharing the following characteristics:

1. a finite set of defined states, one of which being defined as the *initial state* of the machine
2. a set of defined inputs
3. a set of defined outputs
4. a set of transitions between selected states, and
5. the machine is said to be in a single state at any instant of time

5.0 SUMMARY

In this unit we discussed extensively about finite state machines.

6.0 TUTOR-MARKED ASSIGNMENT

State in not less than 800 words, all you know about finite state machines.

7.0 REFERENCES/FURTHER READING

Comer, J. David (1994). *Digital Logic & State Machine Design*. Oxford University Press.

Dandamudi, S. (2003). *Fundamentals of Computer Organization and Design*. Springer.

Hayes, P. John (1993). *Introduction to Digital Logic Design*. Prentice Hall.

Hennessy, L. John & Patterson, A. David (2008). *Computer Organization & Design*. Morgan Kaufmann.

Holdsworth, Brian & Woods, Clive (2002). *Digital Logic Design*. Newnes.

Mano, M. Morris, Kime, Charles R. (2007). *Logic and Computer Design Fundamentals*. Prentice Hall. Pg 283.

Marcovitz, Alan (2009). *Introduction to Logic Design*. McGraw-Hill.

Neamen, Donald (2009). *Microelectronics: Circuit Analysis & Design*. McGraw-Hill.

Nelson, P. Victor, *et al.* (1995). *Digital Logic Circuit Analysis & Design*. Prentice Hall.

Palmer, James & Periman, David (1993). *Schaum's Outline of Introduction to Digital Systems*. McGraw-Hill.

Patterson, David & Hennessy, John, (1993). *Computer Organization & Design: The Hardware/Software Interface*. Morgan Kaufmann Publishers.

Rafiquzzaman, M. (2005). *Fundamentals of Digital Logic & Microcomputer Design*. Wiley – Interscience.

Roth, H. Charles Jr & Kinney, L. Larry (2009). *Fundamentals of Logic Design*.

Tanenbaum, Andrew S. (1998). *Structured Computer Organization.* Prentice Hall.

Tocci, J. Ronald & Widmer, S. Neal (1994). *Digital Systems: Principles and Applications.*

Tokheium, Roger (1994). *Schaum's Outline of Digital Principles*. McGraw-Hill.

Wakerly, F. John (2005). *Digital Design: Principles & Practices Package*. Prentice Hall.

MODULE 4 MEMORY

Unit 1 Memory Organisation
Unit 2 Memory Types
Unit 3 Memory Expansion
Unit 4 Memory Summary

UNIT 1 MEMORY ORGANISATION

CONTENTS

1.0 INTRODUCTION

Memory is only one of the required hardware components of a PC. It is not more important, nor less important than the other components of your PC. But much of our modern day software will not operate efficiently, if at all, without "gobs" of memory. Sequential circuits cannot operate without a memory element. Memory elements used in Sequential circuits are relatively small and store few binary bits of information. Large memories capable of storing very large amounts of information are used in computer systems.

A computer which executes an application program has the application stored in the form of program instructions in large memories.

Memories store data in units that have one, four, eight or higher number of bits.

Smallest unit of binary data is a bit. Data is also handled in a 4-bit unit called a Nibble. In many applications the data is handled as an 8-bit unit called a byte, which is a combination of two 4-bit units that are called Nibbles. A complete unit of information is sometimes called a Word and consists of one or more bytes.

Each storage element of a memory can either store a logic 0 or a logic 1 and is called a cell. Memories are arranged in an array and each cell can be identified by specifying a row and a column number. Each square in the diagram represents a memory cell capable of storing a binary 1 or 0. The first eight bits of binary information 11001010 in the first row are stored in eight cells. The addresses of the eight consecutive cells staring from the left most cell are (1,1), (1,2), (1,3), (1,4), (1,5), (1,6), (1,7) and (1,8) representing the first row and columns 1 to 8 respectively. Individual cells at row 5 and column 3 have a binary 1 and a cell at row 6 and column 7 have a binary 0 stored.

Row								
1	1	1	0	0	1	0	1	0
2	0	1	0	0	1	0	1	0
3	0	1	0	0	1	1	0	1
4	1	0	1	0	0	1	0	1
5	1	0	1	1	1	0	1	1
6	0	1	0	1	0	0	0	0
7	1	0	0	1	0	1	1	0
8	0	0	1	1	0	0	0	0
	1	2	3	4	5	6	7	8

Column

Fig. 1: 64-Cell Memory Array

2.0 OBJECTIVES

At the end of this unit, you should be able to:

- define memory and the basic operations performed on it
- state the functions of Read/Write signals.

3.0 MAIN CONTENT

3.1 Memory Organisation

The memory array can be organised in several ways depending on the unit of data. The 64-cell array organised as 8 x 8 cell array is considered as an 8 byte memory, that is, it has eight locations and each location stores a single byte. The 64-cell array organised as 4 x 16 cell array stores 16 nibbles and if organised as 1 x 64 stores 64 single bit values. The 4 x 16 memory array allows data to be accessed in the form of 4-bit nibbles. The 1 x 64 array allows data to be accessed in units of 1 bit.

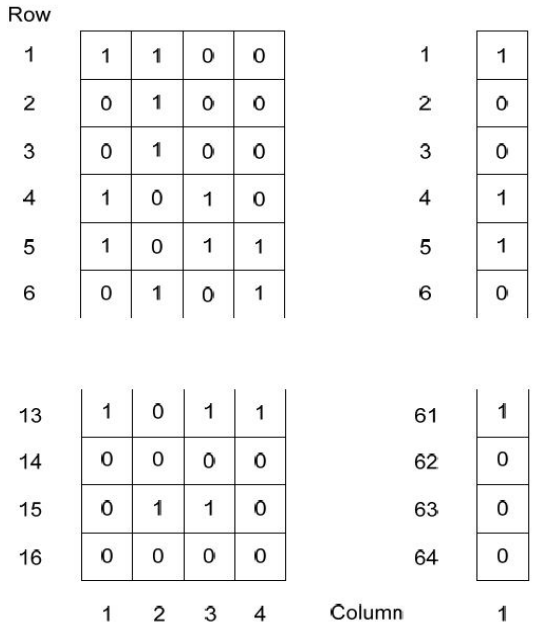


Fig. 2: Memory Organised as 4 x 16 and 1 x 64 Arrays

A memory is identified by the number of units it can store times the unit size. Thus, the 8 x 8 memory is identified as an 8 Byte memory, the 16 x 4 memory is used as a 16 Nibble memory and the 64 x 1 is known as a 64 bit memory. Practical memory chips are organised as 16 K x 8 memory, storing 16K bytes or 16 x 1024 = 16384 bytes. A 32 K x 4 memory stores 32K nibbles or 32 x 1024 = 32768 nibbles.

3.1.1 Memory Capacity and Density

Each memory array has a maximum capacity to store information in the form of bits. Thus, a 16 K x 8 memory, stores 16K bytes or 16 x 1024 = 16384 bytes or 131072 bits. A 32 K x 4 memory stores 32K nibbles or 32 x 1024 = 32768 nibbles or 131072 bits. The total number of cells in each case is 131072. Memory density on the other hand specifies the number of bits stored per unit area. More the number of bits stored in a

unit area more dense the memory, that is, more bits are stored in less space. The capacity and the density of a memory are determined by the total number of cells implemented in a unit area.

3.1.2 Memory Signals and Basic Operations on Memory

Two basic operations are performed on memories, that is, reading of information from the memory and writing of data to the memory. To support the two read and write operations memories provide several signals.

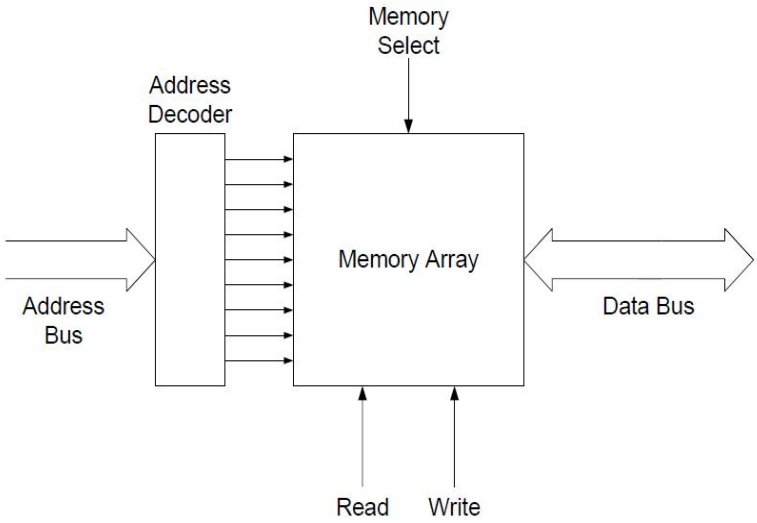


Fig. 3: Block Diagram of a Read-Write Memory

3.1.2.1 Read and Write Signals

Read/Write signals are required to configure the memory for read and write operation. Memory chips have a single Read/Write signal. When the signal is set to high it allows data to be read from the memory. When the signal is set to low data is written into the memory. Some memory chips have two separate Read and Write signals. The read and write signals are separately asserted to control the Read and Write operation.

3.1.2.2 Address Signals

Address signals are required to specify the location in the memory from which information is accessed (read or written). A set of parallel address lines known as the address bus carry the address information. The number of bits (lines) comprising the address bus depends upon the size of the memory. For example, a memory having four locations to store data has four unique addresses (00, 01, 10, 11) specified by a 2-bit address bus. The size of the address bus depends upon the total

addressable locations specified by the formula 2n, where n is the number of bits. Thus, 24=16 (n=4) specifies 4 bits to uniquely identify 16 different locations.

3.1.2.3 Data Signals

Data lines are required to retrieve the information from the memory array during a read operation and to provide the data that is to be stored in the memory during a write operation. As the memory reads or writes one data unit at a time therefore the data lines should be equal to the number of data bits stored at each addressable location in the memory. A memory organised as a byte memory reads or writes byte data values, therefore the number of data lines or the size of the data bus should be 8-bits or 1 byte. A memory organised to store nibble data values requires a 4-bit wide data bus. Generally, the wider the data bus more data can be accessed at each read or write operation.

3.1.2.4 Memory Select or Enable Signal

In a computer system there are more than one memory chips to store program information. At any particular instant a read or write operation is carried out on a single addressable location. The unique location can only be accessed in one of the several memory chips; thus, a single memory chip has to be selected before a read or write operation can be carried out. All memory chips have a chip enable or chip select signal which has to be activated before the memory can be accessed.

3.1.2.5 Memory Read Operation

Memory Read operation is carried out by first selecting the memory chip by activating the Memory Select signal. The Read signal is asserted to configure the memory circuitry for reading data from the memory. An address (100) is applied on the Address Lines. The internal address decoder of the memory decodes the address and selects one unique row from which data is read.

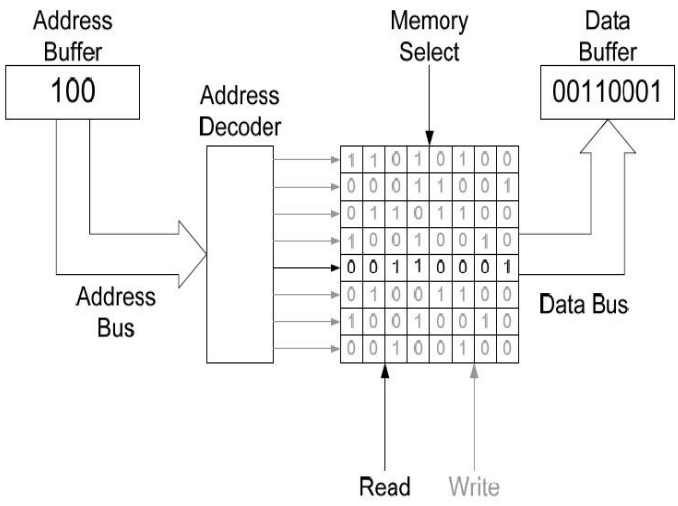


Fig. 4: Memory Read Operation

The address of the location in the memory from which data is to be read is supplied by the microprocessor. The microprocessor stores the address in its address buffer. The data read from the memory is stored in a data buffer inside the microprocessor. In the diagram shown, a microprocessor places an address 100 on its external address bus connected to the address lines of the memory. The internal address decoder of the memory decodes the address 100 and activates a row select line which selects the row location 4. The dat (00110001) at the location is read from the memory and placed on the data bus where it is latched by the microprocessor and stored in its data buffer.

3.1.2.6 Memory Write Operation

Memory Write operation is carried out by first selecting the memory chip by activating the Memory Select signal. The Write signal is asserted to configure the memory circuitry for writing data to the memory. An address (011) is placed on the Address Lines by the microprocessor. The internal address decoder of the memory decodes the address and selects one unique row select line which selects the row location 3. The data (10110010) to be written to the selected memory location is placed on the external data bus by the microprocessor which is stored in the selected location.

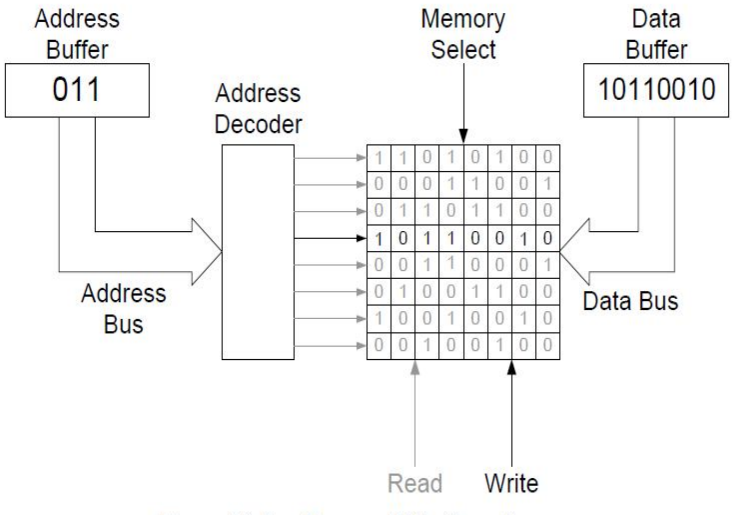


Fig. 5: Memory Write Operation

4.0 CONCLUSION

Memory is only one of the required hardware components of a PC. It is not more important, nor less important than the other components of your PC.

A computer which executes an application program has the application stored in the form of program instructions in large memories.

The memory array can be organised in several ways depending on the unit of data.
A memory is identified by the number of units it can store times the unit size.

Each memory array has a maximum capacity to store information in the form of bits. The capacity and the density of a memory are determined by the total number of cells implemented in a unit area.

Two basic operations are performed on memories, that is, reading of information from the memory and writing of data to the memory.

5.0 SUMMARY

In this unit we discussed about memory organisation. The memory array can be organised in several ways depending on the unit of data. The 64-cell array organised as 8 x 8 cell array is considered as an 8 byte memory, that is, it has eight locations and each location stores a single byte. The 64-cell array organised as 4 x 16 cell array stores 16 nibbles and if organised as 1 x 64 stores 64 single bit values. The 4 x 16

memory array allows data to be accessed in the form of 4-bit nibbles. The 1 x 64 array allows data to be accessed in units of 1 bit.

6.0 TUTOR-MARKED ASSIGNMENT

Write short notes on the following

(a) memory organisation
(b) memory signals
(c) read & write signals
(d) memory read operation
(e) memory write operation.

7.0 REFERENCES/FURTHER READING

Comer, J. David (1994). *Digital Logic & State Machine Design*. Oxford University Press.

Dandamudi, S. (2003). *Fundamentals of Computer Organization and Design*. Springer.

Hayes, P. John (1993). *Introduction to Digital Logic Design*. Prentice Hall.

Hennessy, L. John & Patterson A. David (2008). *Computer Organization & Design*. Morgan Kaufmann.

Holdsworth, Brian & Woods, Clive (2002). *Digital Logic Design*. Newnes.

Mano, M. Morris; Kime, Charles R. (2007). *Logic and Computer Design Fundamentals*. Prentice Hall. Pg 283.

Marcovitz, Alan (2009). *Introduction to Logic Design*. McGraw-Hill.

Neamen, Donald (2009). *Microelectronics: Circuit Analysis & Design*. McGraw-Hill.

Nelson, P. Victor; *et al.* (1995). *Digital Logic Circuit Analysis & Design*. Prentice Hall.

Palmer, James & Periman, David (1993). *Schaum's Outline of Introduction to Digital Systems*. McGraw-Hill.

Patterson, David & Hennessy, John, (1993). *Computer Organization & Design: The Hardware/Software Interface*. Morgan Kaufmann Publishers.

Rafiquzzaman, M. (2005). *Fundamentals of Digital Logic & Microcomputer Design*. Wiley – Interscience.

Roth, H. Charles Jr & Kinney, L. Larry (2009). *Fundamentals of Logic Design*.

Tanenbaum, Andrew S. (1998). *Structured Computer Organization.* Prentice Hall.

Tocci, J. Ronald & Widmer, S. Neal (1994). *Digital Systems: Principles and Applications.*

Tokheium, Roger (1994). *Schaum's Outline of Digital Principles*. McGraw-Hill.

Wakerly, F. John (2005). *Digital Design: Principles & Practices Package*. Prentice Hall.

UNIT 2 MEMORY TYPES

CONTENTS

1.0 INTRODUCTION

This unit introduces us to the types of memory we have. It goes further to explain them and their applications.

2.0 OBJECTIVES

At the end of this unit, you should be able to:

- state the types of memory we have
- state the sub-types of memory
- explain their operations and application.

3.0 MAIN CONTENT

3.1 Memory Types

Two major categories of memory chips are the Random Access Memory (RAM) and Read-Only Memory (ROM).

RAM allows a read or write operation to be carried out at any address. All locations are accessible in equal time. RAM memories do not store permanent data. As soon as the power supply to the memory chip is turned off, the entire data stored in the memory is lost permanently.

RAM memories are also known as volatile memories as they lose data when the power is turned off.

ROM chips retain data permanently even if the power to a ROM chip is turned off. ROM chips are also known as non-volatile memory chips due to their ability to retain data permanently. Since ROM chips are read only, therefore user cannot write any information to ROM chips. ROM chips are programmed by the manufacturer and contain important information which is required to start (Boot Up) the computer.

3.1.1 Random Access Memory (RAM)

RAM is divided into two types, Static RAM which uses flip-flops as storage elements and Dynamic RAM which uses capacitors to store binary information. In a Static RAM each cell which is capable of storing a binary 0 or 1 is made up of a flip-flop which retains information as long as power continues to be supplied to the flip-flop. Dynamic RAM on the other hand uses a capacitor to store a single bit of data. To store binary 1, the capacitor is charged and to store binary 0, the capacitor is in the uncharged state. Capacitors over a period of time lose their charge and unless the capacitors are refreshed the information stored by the capacitor is lost. Dynamic memories periodically charge their capacitors by implementing a refresh cycle. Static memories are faster than Dynamic memories therefore data access in Static memories is faster as compared to dynamic memories. Dynamic memories on the other hand have a high density and can store much more data per unit area and at a lesser cost. Dynamic memories have a high storage density, as capacitors are simpler to implement and occupy a very small semiconductor area as compared to flip-flops.

3.1.1.1 Static RAM

Each cell of a Static RAM is implemented using a flip-flop which is implemented using several MOSFET transistors. External power is required to operate the transistors. As long as the external power is applied the static memory cell retains the data. The circuit of a single flipflop based cell which can store a binary 0 or 1 is shown.

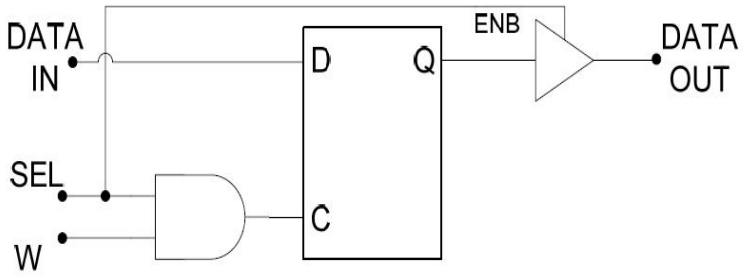


Fig. 1: Circuit Diagram of a Static Memory Cell based on a Flip-Flop

The flip-flop used to store a binary bit works like a latch. When the SEL signal is activated, the output buffer is enabled allowing data to be read out from the memory cell. When both the SEL and W(rite) signals are activated the latch is configured in the transparent mode and the data applied at the Data In line flows through the latch to the output. The Data In and Data Out lines can be connected together to form a bi-directional line which does not cause any problems with the reading or writing of data. This is possible as the read and write operations takes place at different time intervals. The flip-flop based cells are combined to form an array. Additional logic is added to select cells at appropriate locations and to read and write data. A 3 x 8 decoder decodes a 3-bit address to select any one of the eight locations comprising of a group of 4-cells. For example, when the address is 000, the first output line of the 3 x 8 decoder is activated which is connected to the SEL input of the four latches in the first row.

Similarly, address 111 activates the eighth output line of the 3 x 8 decoder which selects the four latches in the last row (location). The memory array has four Data In lines to store the 4-bit data values at the eight locations. Data In 3 and Data In 0 represents the most and least significant bits of the 4- bit data respectively. The four Data In lines connect the Data In inputs of all the latches in each column respectively. The memory array also has four Data Out lines, each data line connects the output of each latch in a column. The read and write operations are controlled through the three signals W, CS and OE. The Chip Select (CS) signal along with the Output Enable (OE) signal enable each of the four tri-state buffers connected to end of each Data Out line. When data is to be read from a memory array, the memory chip is selected and the output enabled. The Write (W) signal along with the CS signal are used to write data into any 4-bit location.

To write data 1001 at the 6th memory location, the address A2, A1 and A0 bits are set to 110 which select the 6th row of the memory array. The data 1001 is placed at the four Data In lines respectively. The CS and W

signals are activated which set the four latches in the sixth row to transparent mode allowing data 1001 applied at the four Data In lines to be available at the Q outputs of the four latches respectively. As soon as the CS and W signals are deactivated, the latches store the data value. A 16K x 8 memory is shown. The memory is capable of storing byte values in 16 x 1024 locations. To address these unique locations, fourteen address lines are required. The memory has eight bi-directional data lines through which data is read/written at selected memory locations. The three CS, WE and OE are shown to be active low.

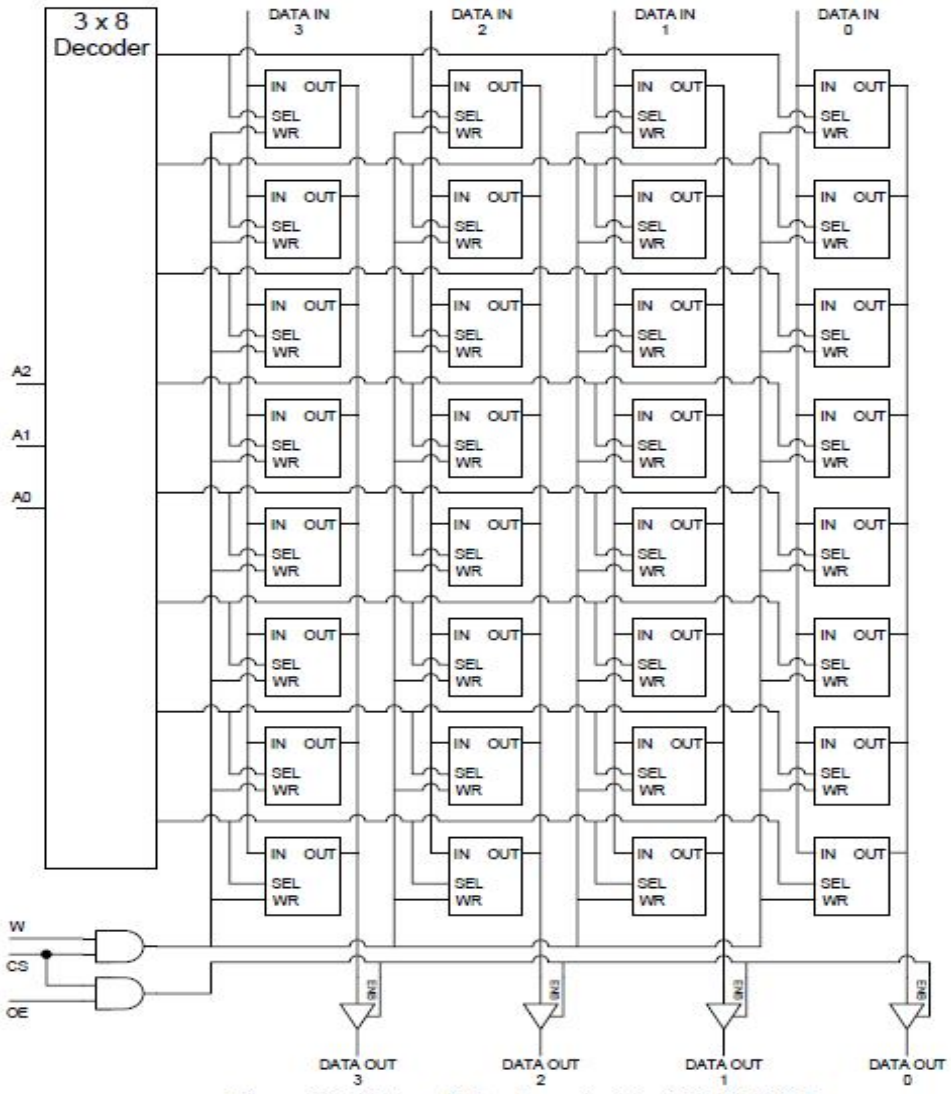


Fig. 2: Internal Structure of an 8 x 4 Static RAM

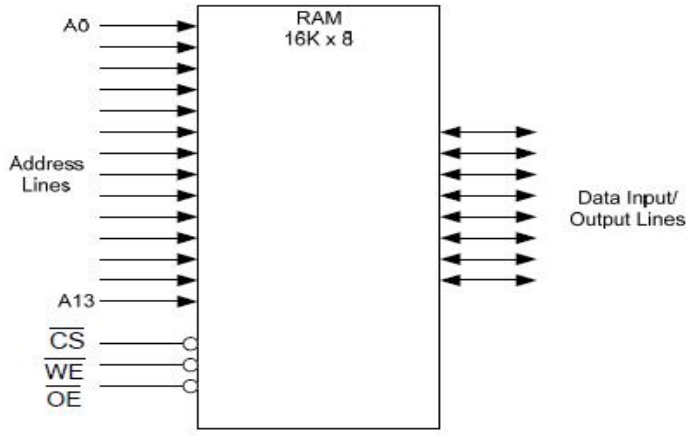


Fig. 3: 16K x 8 Static RAM

RAM chips are subdivided into Asynchronous RAM (ASRAM) and Synchronous Burst RAM (SB SRAM). The Static memory described is an Asynchronous SRAM, the operation of which does not depend upon the clock signal. The read and write operations are carried out asynchronously. Synchronous SRAM uses a clock signal which is used by the microprocessor to synchronise its activities to synchronise the read and write operations for faster operation. The block diagram of a Synchronous Burst SRAM is shown.

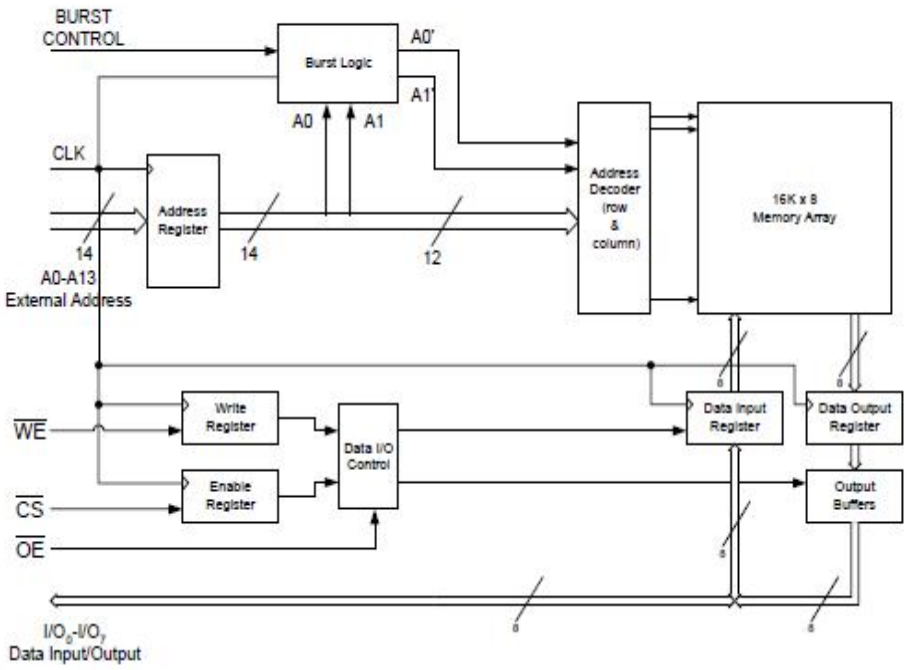


Fig. 4: Block Diagram of a Synchronous Burst RAM

Synchronous RAM is very similar to the Asynchronous RAM, in terms of the memory array, the address decoders, read/write and enable inputs. In the Asynchronous memory the various input signals are asynchronous and are not tied to the clock, whereas in the Synchronous memory all the inputs are synchronised with respect to the clock and are latched into their various registers on an active clock pulse edge. In the diagram, the external address, the WE and the CS external signals are latched in on a positive clock transition simultaneously. The data that is to be written into the memory is also latched into the Data Input Register at the same positive clock transition. For a read operation the data is latched in the Data Output register on the positive clock transition. There are two variations of the Synchronous SRAM, the Flow-through and the Pipelined SRAM. In the Flow-through SRAM there is no Data Output Register so the data is asynchronously available on the data lines during a read operation. In the Pipelined version there is a Data Output Register which latches in the data read from the memory array.

3.1.1.2 Dynamic RAM

A static RAM uses a latch to store a single bit of information. Four gates are used to implement a latch. In terms of transistors, 4 to 6 transistors are required to implement a single storage cell. In order to build memories with higher densities, a single transistor is used to store a binary value. A single transistor cannot store a binary value however it is used to charge and discharge a capacitor. A single memory cell is thus implemented using a single transistor and a capacitor which occupy lesser space as compared to the six transistors which are used to implement a single Static RAM cell. Thus, the density of the capacitor based memory is significantly increased. The capacitor based memory is known as a Dynamic RAM (DRAM). The drawback of DRAM is the discharging of the capacitor over a period of time. Unless the capacitor is periodically recharged all the information stored in terms of binary bits in a capacitor based memory array is lost. The extra circuitry required to refresh the capacitor complicates the operation of the DRAM. The circuit diagram of a single DRAM capacitor based memory cell is shown. The capacitor is connected through a MOSFET which connects or disconnects the column line at B to the capacitor at D. If the row is set at logic high the MOSFET connects the column line to the capacitor. If the row line is set to logic low the MOSFET disconnects the column line form the capacitor.

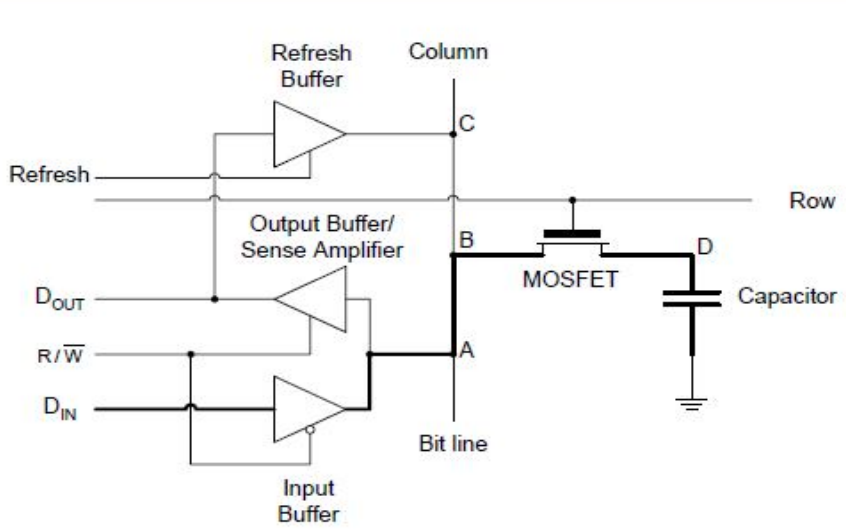


Fig. 5: Writing a 1 or 0 into the DRAM Cell

A write operation allows a logic 1 or 0 to be stored in a DRAM cell (capacitor). The appropriate cell is selected by specifying the address of the memory location which is decoded and the row connecting the desired cell is activated. The R/W signal is set to logic low indicating a write operation which enables the tri-state Input Buffer. The logic 1 which is to be stored in the memory cell is applied at the D_{IN} data line which is available at A on the column line. The row line is selected (set to logic high) which allows the MOSFET to connect column B to capacitor D. The capacitor is charged to logic 1 voltage level via ABD. A Write operation to store logic 0 in a DRAM cell is similar. The appropriate row is selected by specifying the storage location address. The R/W signal is set to logic low which enables the Input Buffer.

The logic 0 to be stored in the DRAM cell is applied at the D_{IN} which is stored on the capacitor via ABD. The thick line in the diagram indicates the data path from D_{IN} to the storage capacitor. The read operation is accomplished by specifying the address of the location from which data is to be read. The DRAM address decoder activates the appropriate row. The R/W signal is set to logic high which enables the output buffer. The logic 1 or 0 stored on the capacitor is available at D_{OUT} through path DBA.

The capacitor cannot retain the charge; therefore it has to be periodically charged through a refresh cycle. The Refresh Buffer is enabled by setting the Refresh signal to high. The input of the Refresh Buffer is connected to the output buffer/sense amplifier. The R/W signal is set to logic high during the Refresh cycle allowing the information stored on the capacitor to be available at the output of the Output Buffer/Sense

amplifier. The information is feed back to the capacitor through the Refresh Buffer via path CBD.

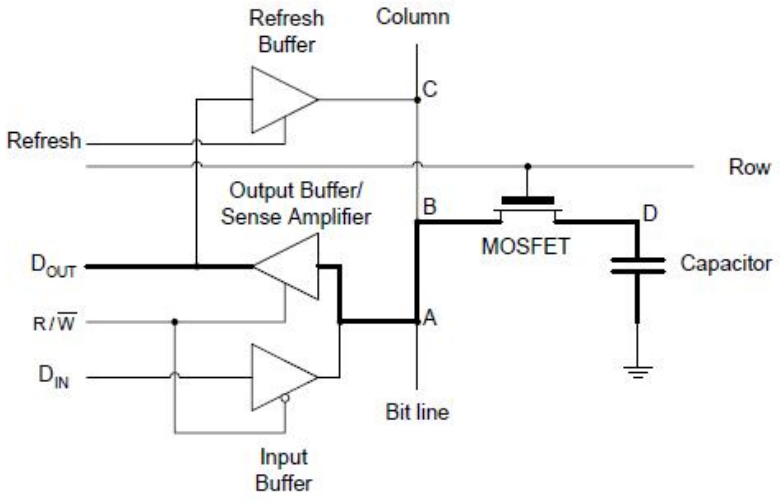


Fig. 6: Reading a 0 or 1 from a DRAM Cell

3.1.1.3 Types of DRAMS

There are several different types of DRAMS available.

- Fast Page Mode DRAM: Compared to random access read/write, FAST Page Mode is faster where successive columns on the same row are read/written in successively by asserting the CAS strobe signal. The CAS signal when de-asserted, disables the D_{OUT} data line, therefore the next column address cannot occur unless the data at the current address is latched by the external system reading data from the DRAM. The access speed of the DRAM during read operation is therefore limited by the external system latching the data available on the D_{OUT} line.

- Extended Data Output (EDO) DRAM: The memory in its operation is similar to the FPMDRAM; however the CAS signal doesn't disable the D_{OUT} when it goes to its non-asserted state. Thus, the valid data on the D_{OUT} line can be remain until the CAS signal is asserted again to access the next column. Thus, the next column address can be accessed before the external system accepts the current data.
- Synchronous DRAM: The DRAM operations are tied to a clock signal that also times the microprocessor operations. This allows the DRAM to closely synchronise with the microprocessor.

3.1.2 Read-Only Memory (ROM)

A ROM contains permanent data that cannot be changed. Thus, ROM memory does not allow write operation. A ROM stores data that are used repeatedly in system applications, such as tables, conversions, programmed instructions for system initialisation and operation. ROMs retain data when the power is turned off.

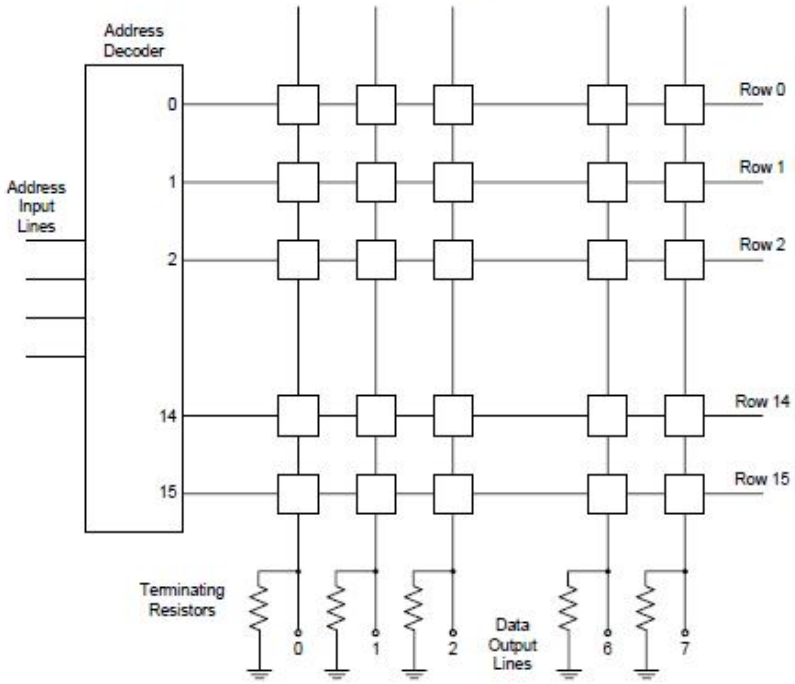


Fig. 7: General Architecture of a ROM

A 16 x 8 ROM is shown. A 4-bit address is decoded by a 4 x 16 decoder which selects the appropriate row line. The MOSFETs connected to the selected row output logic 1 on the respective column lines. The MOSFETs that are not connected output logic 0.The terminating resistor connected to the end of each column line ensures that the output line stays low when a MOSFET outputs logic 0.

Figure 41.3 ROM cell storing a logic 1 and logic 0

Fig. 8: ROM Cell Storing a Logic 1 and Logic 0

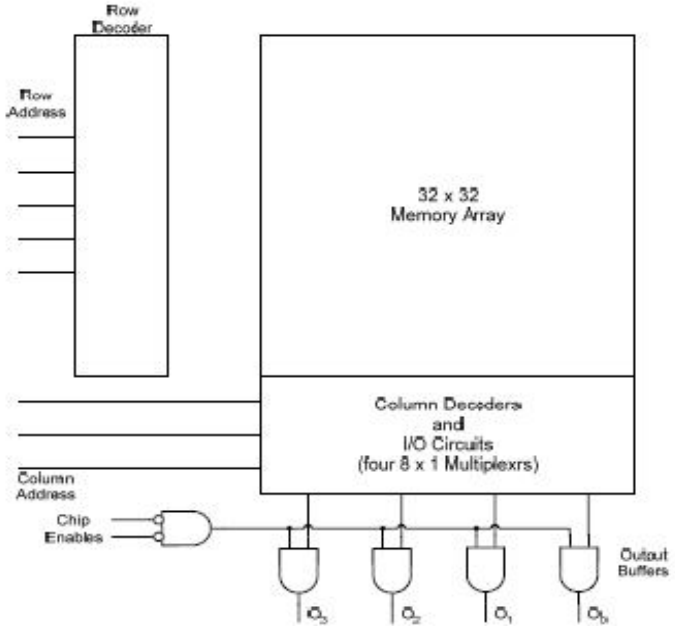


Fig. 9: Internal Structure of a 264 x 4 ROM

ROMs are of different types:

- **Mask ROM:** Data is permanently stored during the manufacturing process.
- **PROM:** Programmable ROM allows storage of data by the user using a PROM programmer. The PROM once programmed stores the data permanently.
- **EPROM:** Erasable PROM allows erasing of stored data and reprogramming.
- **UV EPROM:** Is a programmable ROM. Data is erased by exposing the PROM to Ultraviolet light.
- **EEPROM:** Electrically Erasable PROM is erased electrically. EEPROM allows in-circuit programming and doesn't need to be removed from the circuit for erasure or programming.

3.1.2.1 ROM Application

The 264 x 4 ROM can be used as conversion table to convert 4-bit binary values to 4-bit equivalent Gray Code values. The 4-bit code which is to be converted is applied as an address at the 4-bit address input of the ROM. At each of the 256 locations corresponding to the 256 addresses 256 Gray Code values are stored. The 4-bit Gray Code contents stored at the first 16 locations of the ROM are shown. ROM can also be used as a simple table. Each location in the ROM stores a value which can be accessed by specifying the location address. Look-Up tables used in computers can be implemented using ROMs.

Table 1: ROM Programmed to Convert 4-Bit Binary to 4-Bit Gray Code

Address	Data
0000	0000
0001	0001
0010	0011
0011	0010
0100	0110
0101	0111
0110	0101
0111	0100
1000	1100
1001	1101
1010	1111
1011	1110
1100	1010
1101	1011
1110	1001
1111	1000

3.1.3 Flash Memory

An ideal memory should have high density, have read/write capability, should be nonvolatile, have fast access time and should be cost effective. The ROM, PROM, EPROM, EEPROM, SRAM and DRAM all exhibit some of these characteristics; however, none of these memories have all the mentioned characteristics except for the FLASH memory.

FLASH memories have high density, that is, they store more information per unit area as more storage cells are implemented per unit area. These memories have read/write capability and are non-volatile and can store data for indefinite time period. The high density FLASH memory cell is implemented using a single floating-gate MOS transistor. A data bit is stored as a charge (logic 0) or the absence of a charge (logic 1) on the floating gate. The amount of charge present o the floating gate determines if the transistor will turn and conduct current from the drain to the source when a control voltage is applied at the Control rate during the read operation.

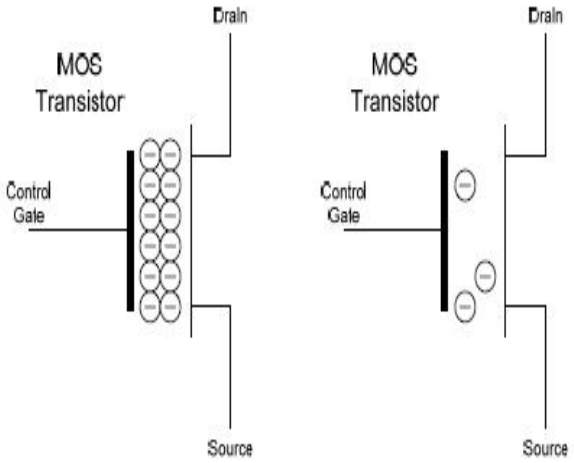


Fig. 10: MOS Transistor with Charge (logic 0) and no Charge (logic 1)

3.1.3.1 Flash Memory Operations

FLASH memory operations are classified into

- Programming Operation
- Read Operation
- Erase Operation

Programming Operation

Initially, all cells are at the logic 1 state - that is with no charge. The programming operation adds charge to the floating gate of those cells that are to store a logic 0. No charge is added to those gates that are to store logic 1. The charges are stored by applying a positive voltage at the Control Gate with respect to the Source which attracts electrons to the floating gate. Once the gate is charged it retains the charge for years.

Read Operation

During the read operation a positive voltage is applied to the MOS transistor control gate. If a negative charge is stored on the gate then the positive read voltage is not sufficient to overcome the negative charge therefore the transistor is not turned on. On the other hand if there is no or small amount of negative charge stored, the positive read voltage is sufficient to overcome the negative charge turning on the transistor. When the transistor is turned on there is a current from the drain to the source of the cell transistor. The presence of this current is sensed to indicate a 1. The absence of this current indicates a 0.

Erase Operation

During the erase operation charge is removed from the memory cell. A sufficiently large positive voltage is applied at the source with respect to the control gate. The voltage applied across the control gate and source is opposite to the voltage applied during programming. If charges are present on the gate, the positive voltage supply at the source attracts the electrons depleting the gate. A flash memory is erased prior to programming.

4.0 CONCLUSION

Two major categories of memory chips are the Random Access Memory (RAM) and Read-Only Memory (ROM). RAM allows a read or write operation to be carried out at any address. All locations are accessible in equal time. RAM memories do not store permanent data. As soon as the power supply to the memory chip is turned off, the entire data stored in the memory is lost permanently.

ROM chips retain data permanently even if the power to a ROM chip is turned off.

RAM is divided into two types, Static RAM which uses flip-flops as storage elements and Dynamic RAM which uses capacitors to store binary information.

Static Memories are faster than Dynamic memories therefore data access in Static Memories is faster as compared to Dynamic Memories. Dynamic memories on the other hand have a high density and can store much more data per unit area and at a lesser cost. ROMs are of different types: Mask ROM, PROM, EPROM, UV EPROM & EEPROM.

FLASH memories have high density, that is, they store more information per unit area as more storage cells are implemented per unit area.

FLASH Memory operations are classified into: Programming Operation, Read Operation and Erase Operation.

SELF-ASSESSMENT EXERCISE

Extensively discuss the different types of ROM we have.

5.0 SUMMARY

In this unit, we discussed about the types of memory, their applications and operations.

6.0 TUTOR-MARKED ASSIGNMENT

Discuss extensively with diagrams and examples, the two types of memory we have.

7.0 REFERENCES/FURTHER READING

Comer, J. David (1994). *Digital Logic & State Machine Design*. Oxford University Press.

Dandamudi, S. (2003). *Fundamentals of Computer Organization and Design*. Springer.

Hayes, P. John (1993). *Introduction to Digital Logic Design*. Prentice Hall.

Hennessy, L. John & Patterson, A. David (2008). *Computer Organization & Design*. Morgan Kaufmann.

Holdsworth, Brian & Woods, Clive (2002). *Digital Logic Design*. Newnes.

Mano, M. Morris; Kime, Charles R. (2007). *Logic and Computer Design Fundamentals*. Prentice Hall.

Marcovitz, Alan (2009). *Introduction to Logic Design*. McGraw-Hill.

Neamen, Donald (2009). *Microelectronics: Circuit Analysis & Design*. McGraw-Hill.

Nelson, P. Victor; *et al.* (1995). *Digital Logic Circuit Analysis & Design*. Prentice Hall.

Palmer, James & Periman, David (1993). *Schaum's Outline of Introduction to Digital Systems*. McGraw-Hill.

Patterson, David & Hennessy, John, (1993). *Computer Organization & Design: The Hardware/Software Interface*. Morgan Kaufmann Publishers.

Rafiquzzaman, M. (2005). *Fundamentals of Digital Logic & Microcomputer Design*. Wiley – Interscience.

Roth, H. Charles Jr & Kinney, L. Larry (2009). *Fundamentals of Logic Design*.

Tanenbaum, Andrew S. (1998). *Structured Computer Organization.* Prentice Hall.

Tocci, J. Ronald & Widmer, S. Neal (1994). *Digital Systems: Principles and Applications.*

Tokheium, Roger (1994). *Schaum's Outline of Digital Principles*. McGraw-Hill.

Wakerly, F. John (2005). *Digital Design: Principles & Practices Package*. Prentice Hall.

UNIT 3 MEMORY EXPANSION

CONTENTS

1.0 INTRODUCTION

This unit exposes us to the various ways and reasons memory is expanded.

2.0 OBJECTIVES

At the end of this unit, you should be able to:

- state why memory is expanded
- explain the roles of microprocessors in a computer system.

3.0 MAIN CONTENT

3.1 Memory Expansion

Digital systems require different amounts of memory in the form of RAM and ROM memory depending upon specific applications. A computer requires large amounts of RAM memory to store multiple application programs, data and the operating system.

In a computer, part of the RAM is reserved to support the Video Memory, Stack and I/O buffers. The ROM used by a computer is relatively very small as it stores few bytes of code used to Boot the Computer system on power up.

Micro-controller based digital system designed for specific applications do not have large memory requirement, in fact the total memory requirement of such micro-controller systems is met by on-board RAM and ROM having a total storage capacity of few hundred kilobytes.

Computer and digital systems have the capability to allow RAM memory to be expanded as the needed arises by inserting extra memory

in dedicated memory sockets on the computer motherboard. The total amount of memory that is supported by any digital system depends upon the size of the address bus of the microprocessor or a micro-controller. A microprocessor having an address bus of 16 bits can generate 216 or 65536 unique addresses to access 65536 locations which allows either a single 65536 location RAM or a combination of RAM and ROM totaling 65536 memory locations to be connected to the microprocessor. It is also possible to initially have a 32768 location RAM connected to the microprocessor with the remaining 32768 address locations unoccupied allowing the microprocessor to execute a program that can be stored in 32768 locations.

The remaining memory space can be utilised latter by connecting another 32768 location RAM.

Microprocessors used in computer systems have memory spaces of the order of 232 and larger. The data unit size accessed by a microprocessor when it issues an address to either read or write from or to a memory also depends upon the microprocessor architecture more specifically the number of the data lines. A microprocessor having an 8-bit data bus can access a byte of information from any unique memory location. A microprocessor having a 16- bit data bus allows two bytes to be accessed from a memory location. Practically, microprocessors used in computer systems have up to 64 bit wide data buses allowing up to 8 bytes of data to be accessed simultaneously. A microprocessor that accesses 64-bits of data simultaneously requires RAM to be organised in such a way that allows 8 bytes of data to be accessed when ever any unique address is selected. On the other hand a microprocessor having a data bus of only 8-bits requires RAM that allows only a single byte of data to be accessed when ever any single address location is selected. The total memory requirement of a computer or digital system is determined by the size of the address and data bus of a microprocessor. Microprocessors which have small address bus and a data bus have a small memory space. Microprocessors which have wide address and data buses have very large memory spaces which are rarely fully occupied by RAM and ROM devices.

Memory, both RAM and ROM are implemented in fixed data unit sizes of 1, 4 or 8 bits. Similarly, these memory devices are implemented having sizes in terms of total addressable locations which are restricted to address ranges between few hundred kilobytes to megabytes.

The memories that are available in fixed sizes have to be connected together to form larger memories having appropriate data unit sizes and total number of addressable locations to fulfill the memory space requirements of a digital or computer system.

Another important aspect of the RAM and ROM memories that are manufactured are the addresses of each memory location. For example, two 32Kbyte RAM chips have 215 locations each. The first addressable locations in both the RAM chips have an address 0. Similarly, the second and third locations in both the memory chips have addresses 1 and 2 respectively. If the two RAM chips are connected together to form a 64 Kbyte RAM then one of 32Kbyte memory chips should respond to the address between 0 and 32767 and the other 32Kbyte memory chip should respond to the address 32768 and 65535. The two memory chips have bases address 0 and 32768 respectively.

4.0 CONCLUSION

Digital systems require different amounts of memory in the form of RAM and ROM Memory depending upon specific applications. A computer requires large amounts of RAM memory to store multiple application programs, data and the operating system. The ROM used by a computer is relatively very small as it stores few bytes of code used to Boot the computer system on power up. The total memory requirement of a computer or digital system is determined by the size of the address and data bus of a microprocessor.

5.0 SUMMARY

In this unit we talked about ways and reasons why we should expand our memory.

6.0 TUTOR-MARKED ASSIGNMENT

State and explain 5 reasons we need to expand memory.

7.0 REFERENCES/FURTHER READING

Comer, J. David (1994). *Digital Logic & State Machine Design*. Oxford University Press.

Dandamudi, S. (2003). *Fundamentals of Computer Organization and Design*. Springer.

Hayes, P. John (1993). *Introduction to Digital Logic Design*. Prentice Hall.

Hennessy, L. John & Patterson, A. David (2008). *Computer Organization & Design*. Morgan Kaufmann.

Holdsworth, Brian & Woods, Clive (2002). *Digital Logic Design*. Newnes.

Mano, M. Morris; Kime, Charles R. (2007). *Logic and Computer Design Fundamentals*. Prentice Hall. Pg 283.

Marcovitz, Alan (2009). *Introduction to Logic Design*. McGraw-Hill.

Neamen, Donald (2009). *Microelectronics: Circuit Analysis & Design*. McGraw-Hill.

Nelson, P. Victor, *et al.* (1995). *Digital Logic Circuit Analysis & Design*. Prentice Hall.

Palmer, James & Periman, David (1993). *Schaum's Outline of Introduction to Digital Systems*. McGraw-Hill.

Patterson, David & Hennessy, John (1993). *Computer Organization & Design: The Hardware/Software Interface*. Morgan Kaufmann Publishers.

Rafiquzzaman, M. (2005). *Fundamentals of Digital Logic & Microcomputer Design*. Wiley – Interscience.

Roth, H. Charles Jr & Kinney, L. Larry (2009). *Fundamentals of Logic Design*.

Tanenbaum, Andrew S. (1998). *Structured Computer Organization.* Prentice Hall.

Tocci, J. Ronald & Widmer, S. Neal (1994). *Digital Systems: Principles and Applications.*

Tokheium, Roger (1994). *Schaum's Outline of Digital Principles*. McGraw-Hill.

Wakerly, F. John (2005). *Digital Design: Principles & Practices Package*. Prentice Hall.

UNIT 4 MEMORY SUMMARY

CONTENTS

1.0 INTRODUCTION

This unit summarises all that we have talked about memory. It does it in a clear and concise manner for easy understanding.

2.0 OBJECTIVES

At the end of this unit, you should be able to:

- define memory
- list the different memory types.

3.0 MAIN CONTENT

3.1 Memory Summary

A summary of memory types and their characteristics are shown. The Static Ram (SRAM) is non-volatile and is not a high density memory as a latch is required to store a single bit of information. Implementation of a latch requires almost six transistors. The Dynamic Ram is also non-volatile however it offers high density memories as each storage cell requires a single transistor and a capacitor. ROMs and PROMs retain information permanently even if the supply voltage is removed. Since a single transistor is used to store a logic 0 or 1 therefore ROMS and PROMs are high density memories. EEPROMs allow data to be read or written however the ability to change the data without having to remove the EEPROM chip from a circuit board requires extra logic. Thus, EEPROM memories are not high density memories.

Table 1: A Summary of Memory Types

Memory Type	Non-Volatile	High Density	One-Transistor Cell	In-System Write ability
SRAM	No	No	No	Yes
DRAM	No	Yes	Yes	Yes
ROM	Yes	Yes	Yes	No
EPROM	Yes	Yes	Yes	No
EEPROM	Yes	No	No	Yes
FLASH	Yes	Yes	Yes	Yes

SELF-ASSESSMENT EXERCISE

Mention five memory types you know and give five differences among them.

4.0 CONCLUSION

Memory is of various types which have various characteristics and they are summarised in this unit.

5.0 SUMMARY

This unit summarises the various memory types. The Static Ram (SRAM) is non-volatile and is not a high density memory as a latch is required to store a single bit of information.

6.0 TUTOR-MARKED ASSIGNMENT

Write a detailed essay on what you have learned in this module.

7.0 REFERENCES/FURTHER READING

Comer, J. David (1994). *Digital Logic & State Machine Design*. Oxford University Press.

Dandamudi, S. (2003). *Fundamentals of Computer Organization and Design*. Springer.

Hayes, P. John (1993). *Introduction to Digital Logic Design*. Prentice Hall.

Hennessy, L. John & Patterson, A. David (2008). *Computer Organization & Design*. Morgan Kaufmann.

Holdsworth, Brian & Woods, Clive (2002). *Digital Logic Design*. Newnes.

Mano, M. Morris; Kime, Charles R. (2007). *Logic and Computer Design Fundamentals*. Prentice Hall. Pg 283.

Marcovitz, Alan (2009). *Introduction to Logic Design*. McGraw-Hill.

Neamen, Donald (2009). *Microelectronics: Circuit Analysis & Design*. McGraw-Hill.

Nelson, P. Victor, *et al.* (1995). *Digital Logic Circuit Analysis & Design*. Prentice Hall.

Palmer, James & Periman, David (1993). *Schaum's Outline of Introduction to Digital Systems*. McGraw-Hill.

Patterson, David & Hennessy, John (1993). *Computer Organization & Design: The Hardware/Software Interface*. Morgan Kaufmann Publishers.

Rafiquzzaman, M. (2005). *Fundamentals of Digital Logic & Microcomputer Design*. Wiley – Interscience.

Roth, H. Charles Jr & Kinney, L. Larry (2009). *Fundamentals of Logic Design*.

Tanenbaum, Andrew S. (1998). *Structured Computer Organization.* Prentice Hall.

Tocci, J. Ronald & Widmer, S. Neal (1994). *Digital Systems: Principles and Applications.*

Tokheium, Roger (1994). *Schaum's Outline of Digital Principles*. McGraw-Hill.

Wakerly, F. John (2005). *Digital Design: Principles & Practices Package*. Prentice Hall.

MODULE 5 INTRODUCTION TO MICROPROCESSORS

Unit 1 Microprocessors
Unit 2 Central Processing Unit & Arithmetic and Logical Unit
Unit 3 Addressing Mode

UNIT 1 MICROPROCESSORS

CONTENTS

1.0 INTRODUCTION

In this unit, we shall be discussing about microprocessors, their history, design, speed and finally architecture.

2.0 OBJECTIVES

At the end of this unit, you should be able to:

- define microprocessors
- state the history of microprocessors
- explain their architecture, speed and design.

3.0 MAIN CONTENT

3.1 Microprocessors

3.1.1 Microprocessor History

A microprocessor (also known as a CPU or central processing unit) is a complete computation engine that is fabricated on a single chip. The first microprocessor was the Intel 4004, introduced in 1971. The 4004 was not very powerful all it could do was add and subtract, and it could only do that 4 bits at a time. But it was amazing that everything was on one chip. Prior to the 4004, engineers built computers either from collections of chips or from discrete components (transistors wired one at a time). The 4004 powered one of the first portable electronic calculators.

The first microprocessor to make it into a home computer was the Intel 8080, a complete 8-bit computer on one chip, introduced in 1974. The first microprocessor to make a real splash in the market was the Intel 8088, introduced in 1979 and incorporated into the IBM PC (which first appeared around 1982). If you are familiar with the PC market and its history, you know that the PC market moved from the 8088 to the 80286 to the 80386 to the 80486 to the Pentium to the Pentium II to the Pentium III to the Pentium 4. All of these microprocessors are made by Intel and all of them are improvements on the basic design of the 8088. The Pentium 4 can execute any piece of code that ran on the original 8088, but it does it about 5,000 times faster!

The following table helps you to understand the differences between the different processors that Intel has introduced over the years.

Table 1: Different Processors Produced over Time by Intel

Name	Date	Transistors	Microns	Clock speed	Data width
8080	1974	6,000	6	2 MHz	8 bits
8088 8-bit bus	1979	29,000	3	5 MHz	16 bits,
80286	1982	134,000	1.5	6 MHz	16 bits
80386	1985	275,000	1.5	16 MHz	32 bits
80486	1989	1,200,000	1	25 MHz	32 bits
Pentium 64-bit bus	1993	3,100,000	0.8	60 MHz	32 bits,
Pentium II 64-bit bus	1997	7,500,000	0.35	233 MHz	32 bits,
Pentium III 64-bit bus	1999	9,500,000	0.25	450 MHz	32 bits,
Pentium 4 64-bit bus	2000	42,000,000	0.18	1.5 GHz	32 bits,

3.1.2 Microprocessor Design

A microprocessor executes programs including the operating system itself and user applications all of which perform useful work. From the microprocessor's point of view, a program is simply a group of low-level *instructions* that the microprocessor executes more or less in sequence as it receives them. How efficiently and effectively the microprocessor executes instructions is determined by its internal design, also called its *architecture*. The CPU architecture, in conjunction with CPU speed, determines how fast the CPU executes instructions of various types. The external design of the microprocessor, specifically its external interfaces, determines how fast it communicates information back and forth with external cache, main memory, the chipset, and other system components.

Microprocessor Components

Modern microprocessors have the following internal components:

1. Execution unit

The core of the CPU, the execution unit processes instructions.

2. Branch predictor

The branch predictor attempts to guess where the program will jump (or branch) next, allowing the prefetch and decode unit to retrieve

instructions and data in advance so that they will already be available when the CPU requests them.

3. Floating-point unit

The floating-point unit (FPU) is a specialised logic unit optimised to perform non-integer calculations much faster than the general-purpose logic unit can perform them.

4. Primary cache

Also called Level 1 or L1 cache, primary cache is a small amount of very fast memory that allows the CPU to retrieve data immediately, rather than waiting for slower main memory to respond.

5. Bus interfaces

Bus interfaces are the pathways that connect the microprocessor to memory and other components. For example, modern microprocessors connect to the chipset Northbridge via a dedicated bus called the frontside bus (FSB) or host bus.

3.1.3 Microprocessor Speed

The microprocessor clock coordinates all CPU and memory operations by periodically generating a time reference signal called a clock cycle or tick. Clock frequency is specified in megahertz (MHz), which specifies millions of ticks per second, or gigahertz (GHz), which specifies billions of ticks per second. Clock speed determines how fast instructions execute. Some instructions require one tick, others multiple ticks, and some processors execute multiple instructions during one tick. The number of ticks per instruction varies according to micro processor architecture, its instruction set, and the specific instruction.

Complex Instruction Set Computer (CISC)

Microprocessors use complex instructions. Each requires many clock cycles to execute, but accomplishes a lot of work.

Reduced Instruction Set Computer (RISC)

Microprocessors use fewer, simpler instructions. Each takes few ticks but accomplishes relatively little work.

These differences in efficiency mean that one CPU cannot be directly compared to another purely on the basis of clock speed. For example, an

AMD Athlon XP 3000+, which actually runs at 2.167 GHz, may be faster than an Intel Pentium 4 running at 3.06 GHz, depending on the application. The comparison is complicated because different CPUs have different strengths and weaknesses. For example, the Athlon is generally faster than the Pentium 4 clock for clock on both integer and floating-point operations (that is, it does more work per CPU tick), but the Pentium 4 has an extended instruction set that may allow it to run optimised software literally twice as fast as the Athlon. The only safe use of direct clock speed comparisons is within a single family. Also, even within a family, processors with similar names may differ substantially internally.

3.1.4 Microprocessor Architecture

Clock speeds increase every year, but the laws of physics limit how fast CPUs can run. If designers depended only on faster clock speeds for better performance, CPU performance would have hit the wall years ago. Instead, designers have improved internal architectures while also increasing clock speeds. Recent CPUs run at more than 650 times the clock speed of the PC/XT's 8088 processor, but provide 6,500 or more times the performance. Here are some major architectural improvements that have allowed CPUs to continue to get faster every year:

- **Wider data busses and registers**

For a given clock speed, the amount of work done depends on the amount of data processed in one operation. Early CPUs processed data in 4-bit (*nibble*) or 8-bit (*byte*) chunks, whereas current CPUs process 32 or 64 bits per operation.

- **FPUs**

All CPUs work well with integers, but processing floating-point numbers to high precision on a general-purpose CPU requires a huge number of operations.

All modern CPUs include a dedicated FPU that handles floating-point operations efficiently.

- **Pipelining**

Early CPUs took five ticks to process an instruction—one each to load the instruction, decode it, retrieve the data, execute the instruction, and write the result. Modern CPUs use pipelining, which dedicates a separate stage to each process and allows one full instruction to be executed per clock cycle.

- **Superscalar architecture**

If one pipeline is good, more are better. Using multiple pipelines allows multiple instructions to be processed in parallel, an architecture called superscalar. A superscalar processor processes multiple instructions per tick.

SELF-ASSESSMENT EXCERCISE

Explain in details with the aid of diagrams, the components of a modern microprocessor.

4.0 CONCLUSION

A microprocessor also known as a CPU or central processing unit is a complete computation engine that is fabricated on a single chip.

A Microprocessor executes programs including the operating system itself and user applications all of which perform useful work.

How efficiently and effectively the microprocessor executes instructions is determined by its internal design, also called its architecture.

Modern microprocessors have the following internal components: execution unit, branch predictor, floating-point unit, primary cache & bus interfaces.

The microprocessor clock coordinates all CPU and memory operations by periodically generating a time reference signal called a clock cycle or tick.

5.0 SUMMARY

This unit discusses microprocessors and the basic information about them.

6.0 TUTOR-MARKED ASSIGNMENT

Write short notes with diagrams on the following terms as it relates to microprocessors:

(a) Pipelining (b) Execution Unit (c) Primary cache (d)Complex Instruction Set Computer (CISC).

7.0 REFERENCES/FURTHER READING

Comer, J David (1994). *Digital Logic & State Machine Design*. Oxford University Press.

Dandamudi, S. (2003). *Fundamentals of Computer Organization and Design*. Springer.

Hayes, P. John (1993). *Introduction to Digital Logic Design*. Prentice Hall.

Hennessy, L. John & Patterson, A. David (2008). *Computer Organization & Design*. Morgan Kaufmann.

Holdsworth, Brian & Woods, Clive (2002). *Digital Logic Design*. Newnes.

Mano, M. Morris; Kime, Charles R. (2007). *Logic and Computer Design Fundamentals*. Prentice Hall. Pg 283.

Marcovitz, Alan (2009). *Introduction to Logic Design*. McGraw-Hill.

Neamen, Donald (2009). *Microelectronics: Circuit Analysis & Design*. McGraw-Hill.

Nelson, P. Victor; *et al.* (1995). *Digital Logic Circuit Analysis & Design*. Prentice Hall.

Palmer, James & Periman, David (1993). *Schaum's Outline of Introduction to Digital Systems*. McGraw-Hill.

Patterson, David & Hennessy, John (1993). *Computer Organization & Design: The Hardware/Software Interface*. Morgan Kaufmann Publishers.

Rafiquzzaman, M. (2005). *Fundamentals of Digital Logic & Microcomputer Design*. Wiley – Interscience.

Roth, H. Charles Jr & Kinney, L. Larry (2009). *Fundamentals of Logic Design*.

Tanenbaum, Andrew S. (1998). *Structured Computer Organization.* Prentice Hall.

Tocci, J. Ronald & Widmer, S. Neal (1994). *Digital Systems: Principles and Applications.*

Tokheium, Roger (1994). *Schaum's Outline of Digital Principles*. McGraw-Hill.

Wakerly, F. John (2005). *Digital Design: Principles & Practices Package*. Prentice Hall.

UNIT 2 CENTRAL PROCESSING UNIT & ARITHMETIC AND LOGIC UNIT

CONTENTS

1.0 INTRODUCTION

This unit exposes us to the Central Processing Unit and the Arithmetic & Logical Unit.

2.0 OBJECTIVES

At the end of this unit, you should be able to:

- explain what the CPU is
- state the various terms associated with the CPU
- explain what the ALU is.

3.0 MAIN CONTENT

3.1 Central Processing Unit & Arithmetic & Logical Unit

3.1.1 Central Processing Unit

A computer is a complex system consisting of many different components. But at the heart or the brain of the computer is a single component that does the actual computing. This is the Central Processing Unit (CPU). In a modern desktop computer, the CPU is a single "chip" on the order of one square inch in size. The job of the CPU is to execute programs.

A program is simply a list of unambiguous instructions meant to be followed mechanically by a computer. A computer is built to carry out instructions that are written in a very simple type of language called machine language. Each type of computer has its own machine language, and it can directly execute a program only if it is expressed in that language. (It can execute programs written in other languages if they are first translated into machine language.)

When the CPU executes a program, that program is stored in the computer's main memory (also called the RAM or random access memory). In addition to the program, memory can also hold data that is being used or processed by the program. Main memory consists of a sequence of locations. These locations are numbered, and the sequence number of a location is called its address. An address provides a way of picking out one particular piece of information from among the millions stored in memory. When the CPU needs to access the program instruction or data in a particular location, it sends the address of that information as a signal to the memory; the memory responds by sending back the data contained in the specified location. The CPU can also store information in memory by specifying the information to be stored and the address of the location where it is to be stored.

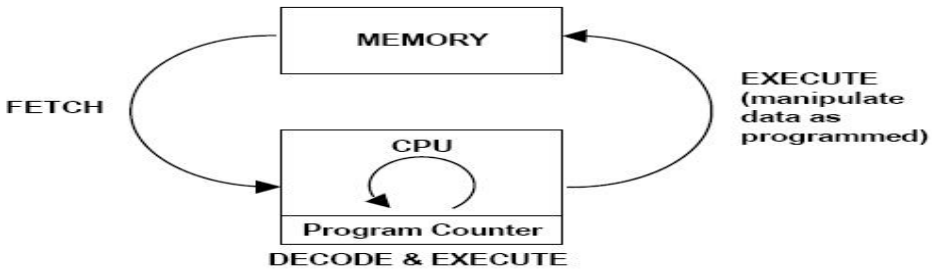


Fig. 1: CPU “Fetch-Execute” Cycle

The CPU “fetch-execute” cycle include:

- fetch instruction from memory
- decode instruction
- perform operations required by the instruction.

A simple program for the above cycle is shown below:

```
function fetch- execute {
pc=init_pc();
while (not_done) {
opcode = fetch_instr(memory[pc]);
execute(opcode);
pc=pc+1;
}
}
function execute(opcode) {
decop = decode(opcode);
if need_data(decop) {
data = get_data(decop);
}
result = compute(decop, data);
if save_result(decop) {
save_result(decop, result);
}
}
```

3.1.2 CPU Registers

CPU registers are *very* special memory locations constructed from flip-flops. They are not part of main memory; the CPU implements them on-chip. Various members of the 80 x 86 family have different register sizes. The 886, 8286, 8486, and 8686 (x86 from now on) CPUs have exactly four registers, all 16 bits wide. All arithmetic and location operations occur in the CPU registers.

Because the x86 processor has so few registers, we'll give each register its own name and refer to it by that name rather than its address. The names for the x86 registers are:

AX	-	The accumulator register
BX	-	The base address register
CX	-	The count register
DX	-	The data register.

Besides the above registers, which are visible to the programmer, the x86 processors also have an instruction pointer register which contains the address of the next instruction to execute. There is also a flags register that holds the result of a comparison. The flags register remembers if one value was less than, equal to, or greater than another value.

Because registers are on-chip and handled specially by the CPU, they are much faster than memory. Accessing a memory location requires one or more clock cycles. Accessing data in a register usually takes zero clock cycles. Therefore, you should try to keep variables in the registers. Register sets are very small and most registers have special purposes which limit their use as variables, but they are still an excellent place to store temporary data.

SELF-ASSESSMENT EXCERCISE

Write short notes on the following registers:

(a) the accumulator register
(b) the base address register
(c) the count register
(d) the data register.

3.1.3 Polling Loops and Interrupts

The CPU spends almost all its time fetching instructions from memory and executing them. However, the CPU and main memory are only two out of many components in a real computer system. A complete system contains other devices such as:

- A hard disk for storing programs and data files. (Note that main memory holds only a comparatively small amount of information, and holds it only as long as the power is turned on. A hard disk is necessary for permanent storage of larger amounts of information, but programs have to be loaded from disk into main memory before they can actually be executed.)
- A keyboard and mouse for user input.
- A monitor and printer which can be used to display the computer's output.
- A network interface that allows the computer to communicate with other computers that are connected to it on a network.
- A scanner that converts images into coded binary numbers that can be stored and manipulated on the computer.

The list of devices is entirely open ended, and computer systems are built so that they can easily be expanded by adding new devices. Somehow the CPU has to communicate with and control all these devices. The CPU can only do this by executing machine language instructions (which is all it can do, period). So, for each device in a system, there is a device driver, which consists of software that the CPU executes when it has to deal with the device. Installing a new device on a system generally has two steps: plugging the device physically into the

computer, and installing the device driver software. Without the device driver, the actual physical device would be useless, since the CPU would not be able to communicate with it.

A computer system consisting of many devices is typically organised by connecting those devices to one or more busses. A bus is a set of wires that carry various sorts of information between the devices connected to those wires. The wires carry data, addresses, and control signals. An address directs the data to a particular device and perhaps to a particular register or location within that device. Control signals can be used, for example, by one device to alert another that data is available for it on the data bus.

Now, devices such as keyboard, mouse, and network interface can produce input that needs to be processed by the CPU. How does the CPU know that the data is there? One simple idea, which turns out to be not very satisfactory, is for the CPU to keep checking for incoming data over and over. Whenever it finds data, it processes it. This method is called polling, since the CPU polls the input devices continually to see whether they have any input data to report. Unfortunately, although polling is very simple, it is also very inefficient. The CPU can waste an awful lot of time just waiting for input.

To avoid this inefficiency, interrupts are often used instead of polling. An interrupt is a signal sent by another device to the CPU. The CPU responds to an interrupt signal by putting aside whatever it is doing in order to respond to the interrupt. Once it has handled the interrupt, it returns to what it was doing before the interrupt occurred. For example, when you press a key on your computer keyboard, a keyboard interrupt is sent to the CPU. The CPU responds to this signal by interrupting what is doing, reading the key that you pressed, processing it, and then returning to the task it was performing before you pressed the key.

Again, you should understand that this is purely mechanical process: A device signals an interrupt simply by turning on a wire. The CPU is built so that when that wire is turned on, it saves enough information about what it is currently doing so that it can return to the same state later. This information consists of the contents of important internal registers such as the program counter. Then the CPU jumps to some predetermined memory location and begins executing the instructions stored there. Those instructions make up an interrupt handler that does the processing necessary to respond to the interrupt. (This interrupt handler is part of the device driver software for the device that signalled the interrupt.) At the end of the interrupt handler is an instruction that tells the CPU to jump back to what it was doing; it does that by restoring its previously saved state.

Interrupts allow the CPU to deal with asynchronous events. In the regular fetch-and-execute cycle, things happen in a predetermined order; everything that happens is "synchronised" with everything else. Interrupts make it possible for the CPU to deal efficiently with events that happen "asynchronously", that is, at unpredictable times.

As another example of how interrupts are used, consider what happens when the CPU needs to access data that is stored on the hard disk. The CPU can only access data directly if it is in main memory. Data on the disk has to be copied into memory before it can be accessed. Unfortunately, on the scale of speed at which the CPU operates, the disk drive is extremely slow. When the CPU needs data from the disk, it sends a signal to the disk drive telling it to locate the data and get it ready. (This signal is sent synchronously, under the control of a regular program.) Then, instead of just waiting the long and unpredictable amount of time the disk drive will take to do this, the CPU goes on with some other task. When the disk drive has the data ready, it sends an interrupt signal to the CPU. The interrupt handler can then read the requested data.

All modern computers use multitasking to perform several tasks at once. Some computers can be used by several people at once. Since the CPU is so fast, it can quickly switch its attention from one user to another, devoting a fraction of a second to each user in turn. This application of multitasking is called timesharing. But even modern personal computers with a single user use multitasking. For example, the user might be typing a paper while a clock is continuously displaying the time and a file is being downloaded over the network.

Each of the individual tasks that the CPU is working on is called a thread. (Or a process; there are technical differences between threads and processes, but they are not important here.) At any given time, only one thread can actually be executed by a CPU. The CPU will continue running the same thread until one of several things happens:

- the thread might voluntarily yield control, to give other threads a chance to run
- the thread might have to wait for some asynchronous event to occur. For example, the thread might request some data from the disk drive, or it might wait for the user to press a key. While it is waiting, the thread is said to be blocked, and other threads have a chance to run. When the event occurs, an interrupt will "wake up" the thread so that it can continue running
- the thread might use up its allotted slice of time and be suspended to allow other threads to run.

Not all computers can "forcibly" suspend a thread in this way; those that can are said to use preemptive multitasking. To do preemptive multitasking, a computer needs a special timer device that generates an interrupt at regular intervals, such as 100 times per second. When a timer interrupt occurs, the CPU has a chance to switch from one thread to another, whether the thread that is currently running likes it or not.

Ordinary users, and indeed ordinary programmers, have no need to deal with interrupts and interrupt handlers. They can concentrate on the different tasks or threads that they want the computer to perform; the details of how the computer manages to get all those tasks done are not relevant to them. In fact, most users, and many programmers, can ignore threads and multitasking altogether. However, threads have become increasingly important as computers have become more powerful and as they have begun to make more use of multitasking. Indeed, threads are built into the Java programming language as a fundamental programming concept.

Just as important in Java and in modern programming in general is the basic concept of asynchronous events. While programmers don't actually deal with interrupts directly, they do often find themselves writing event handlers, which, like interrupt handlers, are called asynchronously when specified events occur. Such "event-driven programming" has a very different feel from the more traditional straight-though, synchronous programming.

By the way, the software that does all the interrupt handling and the communication with the user and with hardware devices is called the operating system. The operating system is the basic, essential software without which a computer would not be able to function. Other programs, such as word processors and World Wide Web browsers, are dependent upon the operating system. Common operating systems include UNIX, DOS, Windows, and the Macintosh OS.

3.1.4 Arithmetic and Logical Unit (ALU)

The Arithmetic and Logical Unit (ALU) is where most of the action takes place inside the CPU. Microprocessors have Arithmetic and Logic Units, a combinational circuit that can perform any of the arithmetic operations and logic operations on two input values. The operation to be performed is selected by set of inputs known as function select inputs. There are different MSI ALUs available that have two 4-bit inputs a 4-bit output and three to five function select inputs that allows up to 32 different functions to be performed.

Three commercially available 4-bit ALUS are:

- 74XX181: The 4-bit ALU has five function select inputs allowing it to perform 32 different Arithmetic and Logic operations.
- 74XX381: The 4-bit ALU only has three function select inputs allowing only 8 different arithmetic and logic functions.
- 74XX382: The 4-bit ALU is similar to the 74XX381, the only difference is that 74XX 381 provides group-carry look-ahead outputs and 74XX382 provides ripple carry and overflow outputs

Table 1: Function Table of 74XX381 4-Bit ALU

Input			
S2	S1	S0	Function
0	0	0	F=0000
0	0	1	$F=B-A-1+C_{in}$
0	1	0	$F=A-B-1+C_{in}$
0	1	1	$F=A+B+C_{in}$
1	0	0	$F = A \oplus B$
1	0	1	F=A+B
1	1	0	F=A.B
1	1	1	F=1111

4.0 CONCLUSION

The CPU "fetch-execute" cycle include:

fetch instruction from memory,
decode instruction and
perform operations required by the instruction.

The Arithmetic and Logical Unit (ALU) is where most of the action takes place inside the CPU. Microprocessors have Arithmetic and Logic Units, a combinational circuit that can perform any of the arithmetic operations and logic operations on two input values.

5.0 SUMMARY

This unit talks extensively on Central Processing Unit and Arithmetic & Logical Unit of a computer system.

6.0 TUTOR-MARKED ASSIGNMENT

Define and explain these terms: ALU, CPU.

7.0 REFERENCES/FURTHER READING

Comer, J David (1994). *Digital Logic & State Machine Design*. Oxford University Press.

Dandamudi, S. (2003). *Fundamentals of Computer Organization and Design*. Springer.

Hayes, P. John (1993). *Introduction to Digital Logic Design*. Prentice Hall.

Hennessy, L. John & Patterson, A. David (2008). *Computer Organization & Design*. Morgan Kaufmann.

Holdsworth, Brian & Woods, Clive (2002). *Digital Logic Design*. Newnes.

Mano, M. Morris; Kime, Charles R. (2007). *Logic and Computer Design Fundamentals*. Prentice Hall. Pg 283.

Marcovitz, Alan (2009). *Introduction to Logic Design*. McGraw-Hill.

Neamen, Donald (2009). *Microelectronics: Circuit Analysis & Design*. McGraw-Hill.

Nelson, P. Victor; *et al.* (1995). *Digital Logic Circuit Analysis & Design*. Prentice Hall.

Palmer, James & Periman, David (1993). *Schaum's Outline of Introduction to Digital Systems*. McGraw-Hill.

Patterson, David & Hennessy, John (1993). *Computer Organization & Design: The Hardware/Software Interface*. Morgan Kaufmann Publishers.

Rafiquzzaman, M. (2005). *Fundamentals of Digital Logic & Microcomputer Design*. Wiley – Interscience.

Roth, H. Charles Jr & Kinney, L. Larry (2009). *Fundamentals of Logic Design*.

Tanenbaum, Andrew S. (1998). *Structured Computer Organization.* Prentice Hall.

Tocci, J. Ronald & Widmer, S. Neal (1994). *Digital Systems: Principles and Applications.*

Tokheium, Roger (1994). *Schaum's Outline of Digital Principles*. McGraw-Hill.

Wakerly, F. John (2005). *Digital Design: Principles & Practices Package*. Prentice Hall.

UNIT 3 ADDRESSING MODES

CONTENTS

1.0 INTRODUCTION

Processors let you access memory in many different ways. Memory addressing modes provide flexible access to memory, allowing you to easily access variables, arrays, records, pointers, and other complex data types. Mastery of the addressing modes is the first step towards mastering assembly language.

2.0 OBJECTIVES

At the end of this unit, you should be able to:

- explain what addressing modes are
- state the types of addressing modes we have
- outline the operations and applications of modes.

3.0 MAIN CONTENT

3.1 Addressing Modes

3.1.1 Register Addressing Mode

Most 80386 instructions can operate on the general purpose register set. By specifying the name of the register as an operand to the instruction,

you may access the contents of that register. Consider the mov (move) instruction:

```
mov    destination, source
```

This instruction copies the data from the source operand to the destination operand. The 32- bit registers are certainly valid operands for this instruction. The only restriction is that both operands must be the same size. Now let's look at some actual mov instructions:

```
mov     eax, ebx; Copies the value from EBX into EAX
mov     edl, eal  ; Copies the value from EAL into EDL
mov     esi,edx  ; Copies the value from EDX into ESI
mov     esp, ebp; Copies the value from EBP into ESP
mov     edh, ecx; Copies the value from ECX into EDH
mov     eax, eax ;Yes, this is legal!
```

Remember, the registers are the best place to keep often used variables. Instructions using the registers are shorter and faster than those that access memory.

In addition to the general purpose registers, many instructions (including the mov instruction) allow you to specify one of the segment registers as an operand. There are two restrictions on the use of the segment registers with the mov instruction. First of all, you may not specify ecs as the destination operand; second, only one of the operands can be a segment register. You cannot move data from one segment register to another with a single mov instruction. To copy the value of ecx to edx, you'd have to use some sequence like:

```
mov     eax, ecx
mov     edx, eax
```

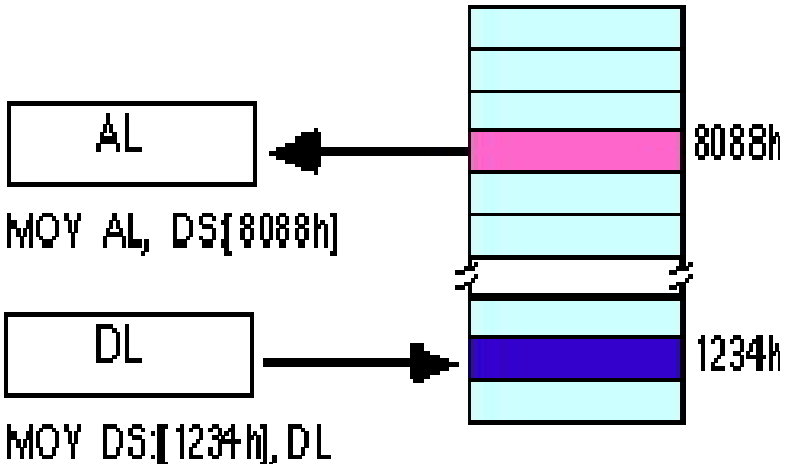
You should never use the segment registers as data registers to hold arbitrary values. They should only contain segment addresses.

3.1.2 Memory Addressing Mode

The key to good assembly language programming is the proper use of memory addressing modes. The 80386 processor generalised the memory addressing modes. Whereas the 8086 only allowed you to use bx or bp as base registers and si or di as index registers, the 80386 lets you use almost any general purpose 32 bit register as a base or index register. Furthermore, the 80386 introduced new scaled indexed addressing modes that simplify accessing elements of arrays. Beyond the increase to 32 bits, the new addressing modes on the 80386 are probably the biggest improvement to the chip over earlier processors.

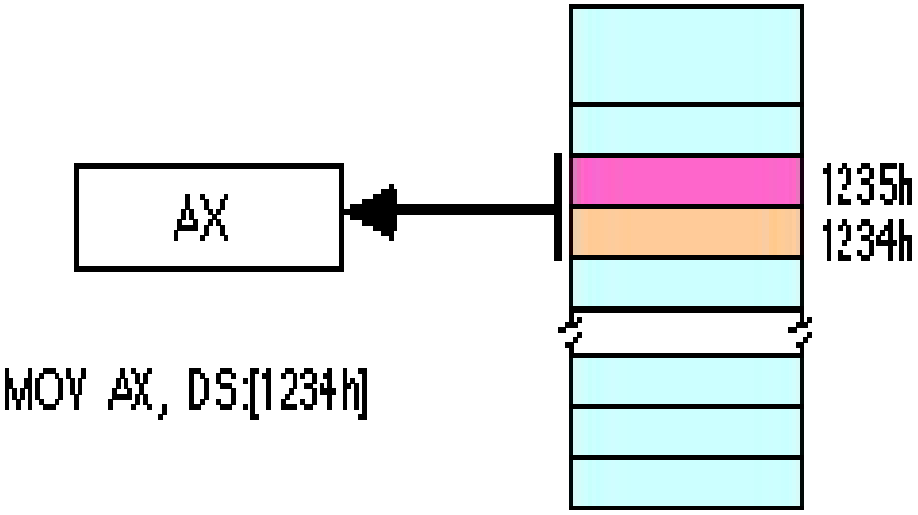
3.1.3 The Displacement Only Addressing Mode

The most common addressing mode, and the one that's easiest to understand, is the displacement-only (or direct) addressing mode. The displacement-only addressing mode consists of a 16 bit constant that specifies the address of the target location. The instruction mov al,ds:[8088h] loads the al register with a copy of the byte at memory location 8088h. Likewise, the instruction mov ds:[1234h],dl stores the value in the dl register to memory location 1234h:



The displacement-only addressing mode is perfect for accessing simple variables. Of course, you'd probably prefer using names like "I" or "J" rather than "DS:[1234h]" or "DS:[8088h]". Well, fear not, you'll soon see it's possible to do just that.

Intel named this the displacement-only addressing mode because a 16 bit constant (displacement) follows the mov opcode in memory. In that respect it is quite similar to the direct addressing mode on the x86 processors. There are some minor differences, however. First of all, a displacement is exactly that - some distance from some other point. On the x86, a direct address can be thought of as a displacement from address zero. On the 80x86 processors, this displacement is an offset from the beginning of a segment (the data segment in this example). For now, you can think of the displacement-only addressing mode as a direct addressing mode. The examples in this unit will typically access bytes in memory. Don't forget, however, that you can also access words on the 8086 processors:



By default, all displacement-only values provide offsets into the data segment. If you want to provide an offset into a different segment, you must use a segment override prefix before your address. For example, to access location 1234h in the extra segment (es) you would use an instruction of the form mov ax,es:[1234h]. Likewise, to access this location in the code segment you would use the instruction mov ax, cs:[1234h]. The ds: prefix in the previous examples is not a segment override. The CPU uses the data segment register by default. These specific examples require ds: because of MASM's syntactical limitations.

3.1.4 Register Indirect Addressing Modes

The 80x86 CPUs let you access memory indirectly through a register using the register indirect addressing modes. There are four forms of this addressing mode on the 8086, best demonstrated by the following instructions:

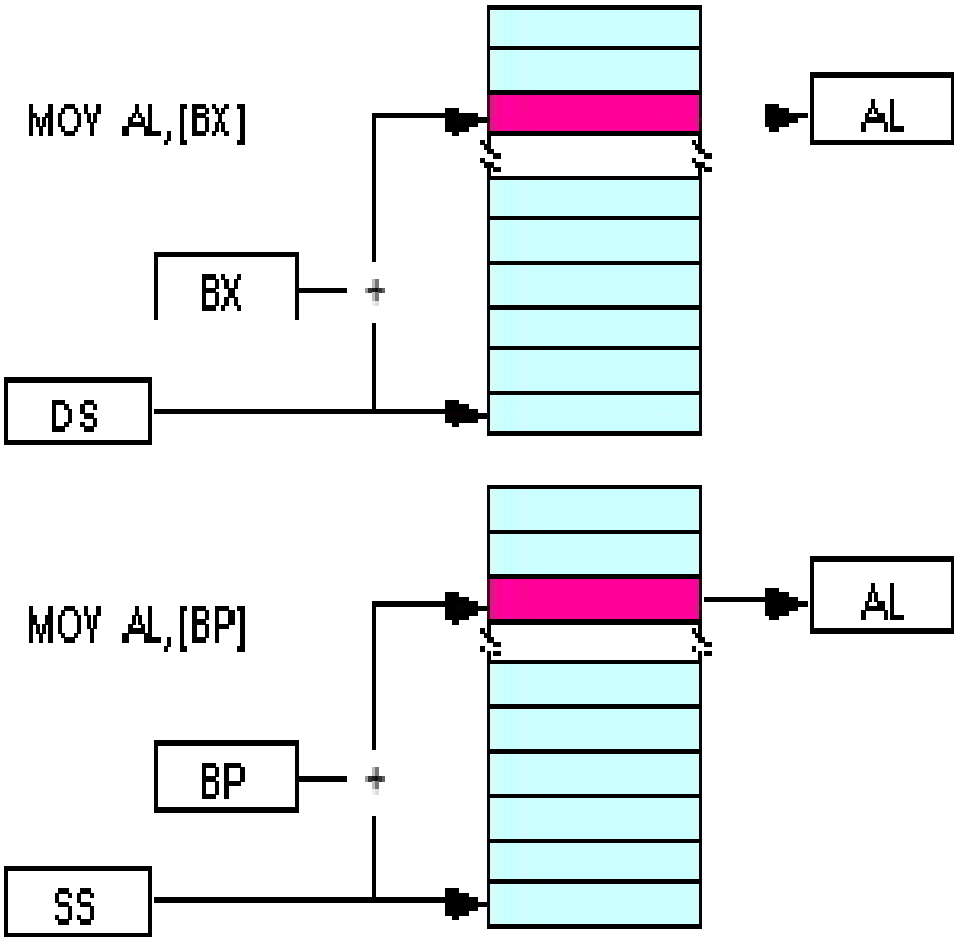
```
mov     al, [bx]
mov     al, [bp]
mov     al, [si]
mov     al, [di]
```

As with the x86 [bx] addressing mode, these four addressing modes reference the byte at the offset found in the bx, bp, si, or di register, respectively. The [bx], [si], and [di] modes use the ds segment by default. The [bp] addressing mode uses the stack segment (ss) by default.

You can use the segment override prefix symbols if you wish to access data in different segments. The following instructions demonstrate the use of these overrides:

```
mov    al, cs:[bx]
mov    al, ds:[bp]
mov    al, ss:[si]
mov    al, es:[di]
```

Intel refers to [bx] and [bp] as base addressing modes and bx and bp as base registers (in fact, bp stands for base pointer). Intel refers to the [si] and [di] addressing modes as indexed addressing modes (si stands for source index, di stands for destination index). However, these addressing modes are functionally equivalent. This text will call these forms register indirect modes to be consistent.



Note: the [si] and [di] addressing modes work exactly the same way, just substitute si and di for bx above.

On the 80386 you may specify any general purpose 32 bit register when using the register indirect addressing mode. [eax], [ebx], [ecx], [edx],

[esi], and [edi] all provide offsets, by default, into the data segment. The [ebp] and [esp] addressing modes use the stack segment by default.

Note that while running in 16 bit real mode on the 80386, offsets in these 32 bit registers must still be in the range 0...0FFFFh. You cannot use values larger than this to access more than 64K in a segment. Also note that you must use the 32 bit names of the registers. You cannot use the 16 bit names. The following instructions demonstrate all the legal forms:

```
mov     al, [eax]
mov     al, [ebx]
mov     al, [ecx]
mov     al, [edx]
mov     al, [esi]
mov     al, [edi]
mov     al, [ebp]       ;Uses SS by default.
mov     al, [esp]       ;Uses SS by default.
```

3.1.5 Indexed Addressing Mode

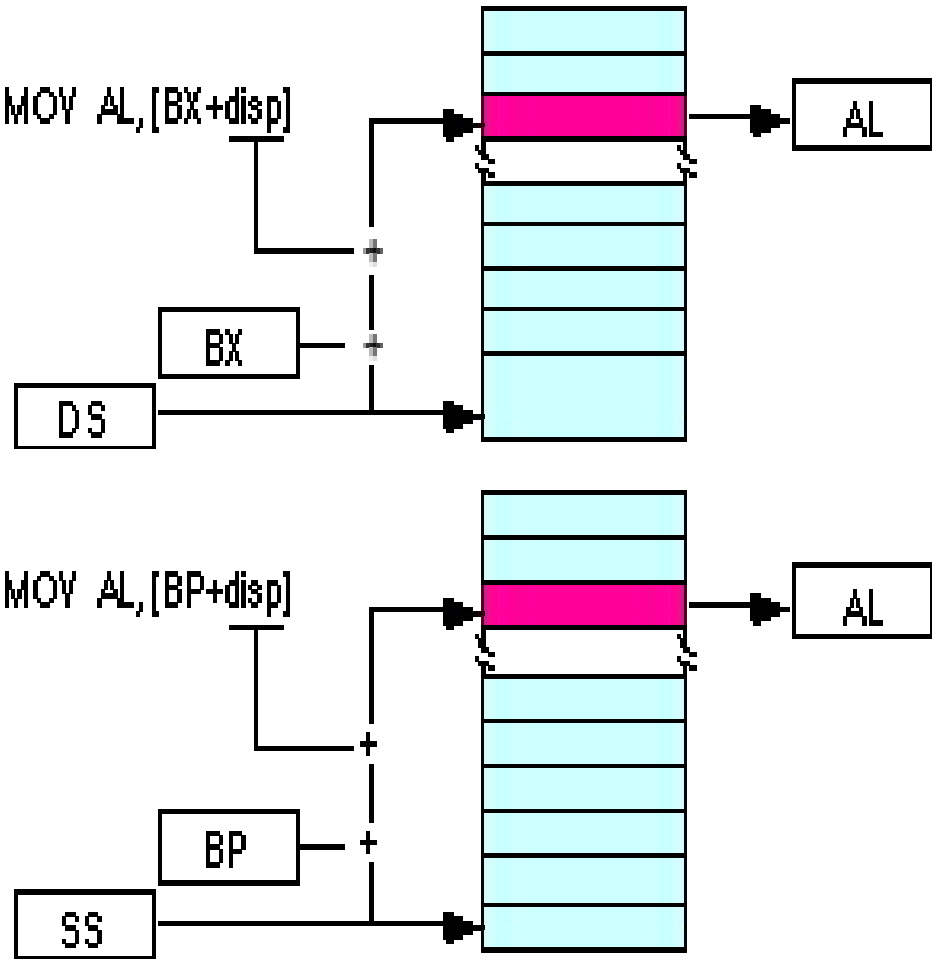
The indexed addressing modes use the following syntax:

```
mov     al, disp[bx]
mov     al, disp[bp]
mov     al, disp[si]
mov     al, disp[di]
```

If bx contains 1000h, then the instruction mov cl,20h[bx] will load cl from memory location ds:1020h. Likewise, if bp contains 2020h, mov dh,1000h[bp] will load dh from location ss:3020.

The offsets generated by these addressing modes are the sum of the constant and the specified register. The addressing modes involving bx, si, and di all use the data segment, the disp[bp] addressing mode uses the stack segment by default. As with the register indirect addressing modes, you can use the segment override prefixes to specify a different segment:

```
mov     al, ss:disp[bx]
mov     al, es:disp[bp]
mov     al, cs:disp[si]
mov     al, ss:disp[di]
```



You may substitute si or di in the figure above to obtain the [si+disp] and [di+disp] addressing modes.

Note that Intel still refers to these addressing modes as based addressing and indexed addressing. Intel's literature does not differentiate between these modes with or without the constant. If you look at how the hardware works, this is a reasonable definition. From the programmer's point of view, however, these addressing modes are useful for entirely different things.

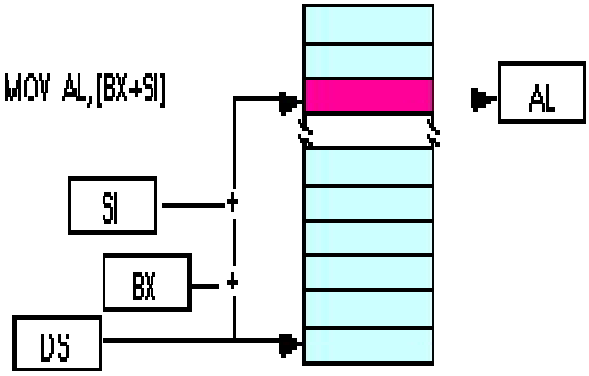
3.1.6 Based Indexed Addressing Modes

The based indexed addressing modes are simply combinations of the register indirect addressing modes. These addressing modes form the offset by adding together a base register (bx or bp) and an index register (si or di). The allowable forms for these addressing modes are:

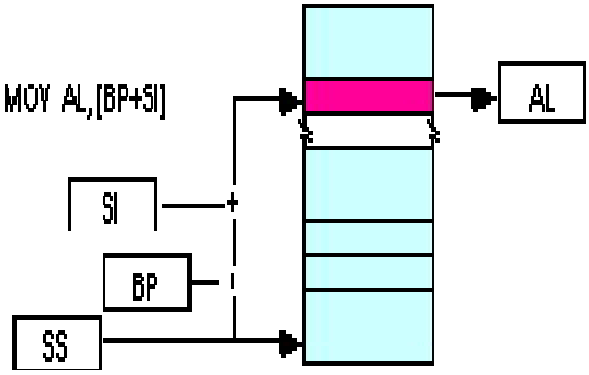
```
mov     al, [bx][si]
mov     al, [bx][di]
mov     al, [bp][si]
mov     al, [bp][di]
```

Suppose that bx contains 1000h and si contains 880h. Then the instruction moval, [bx][si] would load al from location DS:1880h. Likewise, if bp contains 1598h and di contains 1004, mov ax,[bp+di] will load the 16 bits in ax from locations SS:259C and SS:259D.

The addressing modes that do not involve bp use the data segment by default. Those that have bp as an operand use the stack segment by default.



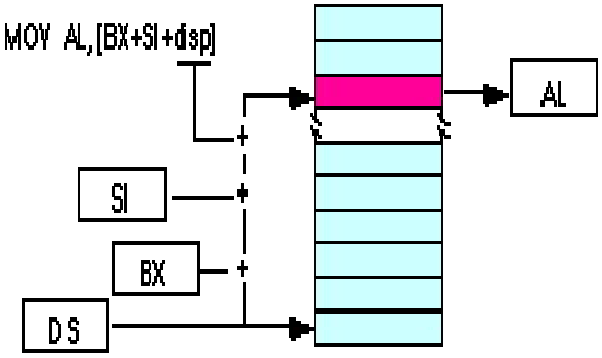
You substitute di in the figure above to obtain the [bx+di] addressing mode.



You substitute di in the figure above for the [bp+di] addressing mode.

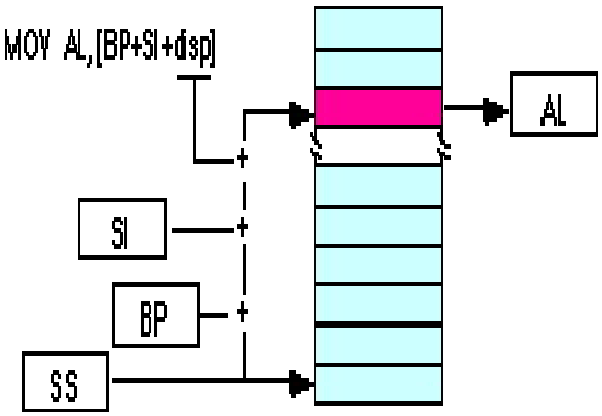
Based Indexed Plus Displacement Addressing Mode

These addressing modes are a slight modification of the base/indexed addressing modes with the addition of an eight bit or sixteen bit constant. The following are some examples of these addressing modes:




```
mov     al, disp[bx][si]
mov     al, disp[bx+di]
mov     al, [bp+si+disp]
mov     al, [bp][di][disp]
```

You may substitute di in the figure above to produce the [bx+di+disp] addressing mode.



You may substitute di in the figure above to produce the [bp+di+disp] addressing mode.

Suppose bp contains 1000h, bx contains 2000h, si contains 120h, and di contains 5. Then mov al,10h[bx+si] loads al from address DS:2130; mov ch,125h[bp+di] loads ch from location SS:112A; and mov bx,cs:2[bx][di] loads bx from location CS:2007.

80386 Indexed, Base/Indexed, and Base/Indexed/Disp Addressing Modes

The indexed addressing modes (register indirect plus a displacement) allow you to mix a 32 bit register with a constant. The base/indexed addressing modes let you pair up two 32 bit registers. Finally, the base/indexed/displacement addressing modes let you combine a constant and two registers to form the effective address. Keep in mind that the offset produced by the effective address computation must still be 16

bits long when operating in real mode. On the 80386 the terms base register and index register actually take on some meaning. When combining two 32 bit registers in an addressing mode, the first register is the base register and the second register is the index register. This is true regardless of the register names. Note that the 80386 allows you to use the same register as both a base and index register, which is actually useful on occasion. The following instructions provide representative samples of the various base and indexed address modes along with syntactical variations:

```
mov    al, disp[eax]        ;Indexed addressing
mov    al, [ebx+disp]       ;modes.
mov    al, [ecx][disp]
mov    al, disp[edx]
mov    al, disp[esi]
mov    al, disp[edi]
mov    al, disp[ebp]        ;Uses SS by default.
mov    al, disp[esp]        ;Uses SS by default.
```

The following instructions all use the base+indexed addressing mode. The first register in the second operand is the base register, the second is the index register. If the base register is esp or ebp the effective address is relative to the stack segment. Otherwise the effective address is relative to the data segment. Note that the choice of index register does not affect the choice of the default segment.

```
mov    al, [eax][ebx]       ;Base+indexed addressing
mov    al, [ebx+ebx]        ;modes.
mov    al, [ecx][edx]
mov    al, [edx][ebp]       ;Uses DS by default.
mov    al, [esi][edi]
mov    al, [edi][esi]
mov    al, [ebp+ebx]        ;Uses SS by default.
mov    al, [esp][ecx]       ;Uses SS by default.
```

Naturally, you can add a displacement to the above addressing modes to produce the base+indexed+displacement addressing mode. The following instructions provide a representative sample of the possible addressing modes:

```
mov    al, disp[eax][ebx]   ;Base+indexed addressing
mov    al, disp[ebx+ebx]    ;modes.
mov    al, [ecx+edx+disp]
mov    al, disp[edx+ebp]    ;Uses DS by default.
mov    al, [esi][edi][disp]
mov    al, [edi][disp][esi]
```

```
mov     al, disp[ebp+ebx]       ;Uses SS by default.
mov     al, [esp+ecx][disp]     ;Uses SS by default.
```

There is one restriction the 80386 places on the index register. You cannot use the esp register as an index register. It's okay to use esp as the base register, but not as the index register.

3.1.7 80386 Scaled Indexed Addressing Modes

The indexed, base/indexed, and base/indexed/disp addressing modes described above are really special instances of the 80386 scaled indexed addressing modes. These addressing modes are particularly useful for accessing elements of arrays, though they are not limited to such purposes. These modes let you multiply the index register in the addressing mode by one, two, four, or eight. The general syntax for these addressing modes is

```
disp[index*n]
[base][index*n]
or
disp[base][index*n]
```

where "base" and "index" represent any 80386 32 bit general purpose registers and "n" is the value one, two, four, or eight.

The 80386 computes the effective address by adding disp, base, and index*n together. For example, if ebx contains 1000h and esi contains 4, then

```
mov al,8[ebx][esi*4]            ;Loads AL from location 1018h
mov al,1000h[ebx][ebx*2]        ;Loads AL from location 4000h
mov al,1000h[esi*8]             ;Loads AL from location 1020h
```

Note that the 80386 extended indexed, base/indexed, and base/indexed/displacement addressing modes really are special cases of this scaled indexed addressing mode with "n" equal to one. That is, the following pairs of instructions are absolutely identical to the 80386:

```
mov     al, 2[ebx][esi*1]       mov     al, 2[ebx][esi]
mov     al, [ebx][esi*1]        mov     al, [ebx][esi]
mov     al, 2[esi*1]            mov     al, 2[esi]
```

4.0 CONCLUSION

The effective address is the final offset produced by an addressing mode computation. There is even a special instruction load effective address (lea) that computes effective addresses.

Not all addressing modes are created equal! Different addressing modes may take differing amounts of time to compute the effective address. The exact difference varies from processor to processor. Generally, though, the more complex an addressing mode is, the longer it takes to compute the effective address. Complexity of an addressing mode is directly related to the number of terms in the addressing mode.

The displacement field in all addressing modes except displacement-only can be a signed eight bit constant or a signed 16 bit constant. If your offset is in the range -128...+127 the instruction will be shorter (and therefore faster) than an instruction with a displacement outside that range. The size of the value in the register does not affect the execution time or size. So if you can arrange to put a large number in the register(s) and use a small displacement that is preferable over large constant and small values in the register(s). If the effective address calculation produces a value greater than 0FFFFh, the CPU ignores the overflow and the result wraps around back to zero.

5.0 SUMMARY

In this unit, we discussed addressing modes, their various types and how they are used.

6.0 TUTOR-MARKED ASSIGNMENT

Mention and explain the different types of addressing modes we have.

7.0 REFERENCES/FURTHER READING

Comer, J. David (1994). *Digital Logic & State Machine Design*. Oxford University Press.

Dandamudi, S. (2003). *Fundamentals of Computer Organization and Design*. Springer.

Hayes, P. John (1993). *Introduction to Digital Logic Design*. Prentice Hall.

Hennessy, L. John & Patterson, A. David (2008). *Computer Organization & Design*. Morgan Kaufmann.

Holdsworth, Brian & Woods, Clive (2002). *Digital Logic Design*. Newnes.

Mano, M. Morris; Kime, Charles, R. (2007). *Logic and Computer Design Fundamentals*. Prentice Hall. Pg 283.

Marcovitz, Alan (2009). *Introduction to Logic Design*. McGraw-Hill.

Neamen, Donald (2009). *Microelectronics: Circuit Analysis & Design*. McGraw-Hill.

Nelson, P. Victor; *et al.* (1995). *Digital Logic Circuit Analysis & Design*. Prentice Hall.

Palmer, James & Periman, David (1993). *Schaum's Outline of Introduction to Digital Systems*. McGraw-Hill.

Patterson, David & Hennessy, John, (1993). *Computer Organization & Design: The Hardware/Software Interface*. Morgan Kaufmann Publishers.

Rafiquzzaman, M. (2005). *Fundamentals of Digital Logic & Microcomputer Design*. Wiley – Interscience.

Roth, H. Charles Jr & Kinney, L. Larry (2009). *Fundamentals of Logic Design*.

Tanenbaum, Andrew S. (1998). *Structured Computer Organization.* Prentice Hall.

Tocci, J. Ronald & Widmer, S. Neal (1994). *Digital Systems: Principles and Applications.*

Tokheium, Roger (1994). *Schaum's Outline of Digital Principles*. McGraw-Hill.

Wakerly, F. John (2005). *Digital Design: Principles & Practices Package*. Prentice Hall.

MODULE 6 ASSEMBLY LANGUAGE PROGRAMMING

Unit 1 Learning to Program with Assembly Language
Unit 2 Branching Loops and Subroutines
Unit 3 Sample Programs in Assembly Language

UNIT 1 LEARNING TO PROGRAM WITH ASSEMBLY LANGUAGE

CONTENTS

1.0 INTRODUCTION

Unlike the other programming languages, assembly language is not a single language, but rather a group of languages. Each processor family (and sometimes individual processors within a processor family) has its own assembly language.

In contrast to high level languages, data structures and program structures in assembly language are created by directly implementing them on the underlying hardware. So, instead of cataloguing the data structures and program structures that can be built (in assembly language you can build any structures you so desire, including new structures nobody else has ever created), we will compare and contrast the hardware capabilities of various processor families.

Assembly languages have the same structure and set of commands as machine languages, but they enable a programmer to use names instead of numbers.

Each type of CPU has its own machine language and assembly language, so an assembly language program written for one type of CPU won't run on another. In the early days of programming, all programs were written in assembly language. Now, most programs are written in high-level language. Programmers still use assembly language when speed is essential or when they need to perform an operation that isn't possible in a high-level language.

Assembly language teaches how a computer works at the machine level (i.e. registers).The foundation of many abstract issues in software lies in assembly language. Assembly language is not used just to illustrate algorithms, but to demonstrate what is actually happening inside the computer.

2.0 OBJECTIVES

At the end of this unit, you should be able to:

- define assembly language
- write programs using assembly language.

3.0 MAIN CONTENT

3.1 Learning to Program with Assembly Language

3.1.1 Availability

Assemblers are available for just about every processor ever made. **Native** assemblers produce object code on the same hardware that the object code will run on. **Cross** assemblers produce object code on different hardware that the object code will run on.

3.1.2 Why Learn Assembly Language

The first reason to work with assembler is that it provides the opportunity of knowing more the operation of your PC, which allows the development of software in a more consistent manner.

The second reason is the total control of the PC which you can have with the use of the assembler.

Another reason is that the assembly programs are quicker, smaller, and have larger capacities than ones created with other languages.

Lastly, the assembler allows an ideal optimization in programs, be it on their size or on their execution.

3.1.3 Assembly Language Statements

Assembly language statements in a source file use the following format:

{Label} {Mnemonic} {Operand}} {;Comment}

Each entity above is a field. The four fields above are the label field, the mnemonic field, the operand field, and the comment field.

The label field is (usually) an optional field containing a symbolic label for the current statement. Labels are used in assembly language, just as in HLLs, to mark lines as the targets of GOTOs (jumps). You can also specify variable names, procedure names, and other entities using symbolic labels. Most of the time the label field is optional, meaning a label need be present only if you want a label on that particular line. Some mnemonics, however, require a label, others do not allow one. In general, you should always begin your labels in column one as it makes your programs easier to read.

A mnemonic is an instruction name (e.g., mov, add, etc.). The word mnemonic means memory aid. mov is much easier to remember than the binary equivalent of the mov instruction! The braces denote that this item is optional. Note, however, that you cannot have an operand without a mnemonic.

The mnemonic field contains an assembler instruction. Instructions are divided into three classes: 80x86 machine instructions, assembler directives, and pseudo opcodes. 80x86 instructions, of course, are assembler mnemonics that correspond to the actual 80x86 instructions.
Assembler directives are special instructions that provide information to the assembler but do not generate any code. Examples include the segment directive, equ, assume, and end. These mnemonics are not valid 80x86 instructions. They are messages to the assembler, nothing else.

A pseudo-opcode is a message to the assembler, just like an assembler directive, however a pseudo-opcode will emit object code bytes. Examples of pseudo-opcodes include byte, word, dword, qword, and tbyte. These instructions emit the bytes of data specified by their operands but they are not true 80X86 machine instructions.

The operand field contains the operands, or parameters, for the instruction specified in the mnemonic field. Operands never appear on lines by themselves. The type and number of operands (zero, one, two, or more) depend entirely on the specific instruction.

The comment field allows you to annotate each line of source code in your program. Note that the comment field always begins with a semicolon. When the assembler is processing a line of text, it completely ignores everything on the source line following a semicolon.

Each assembly language statement appears on its own line in the source file. You cannot have multiple assembly language statements on a single line. On the other hand, since all the fields in an assembly language statement are optional, blank lines are fine. You can use blank lines anywhere in your source file. Blank lines are useful for spacing out certain sections of code, making them easier to read.

The Microsoft Macro Assembler is a free form assembler. The various fields of an assembly language statement may appear in any column (as long as they appear in the proper order). Any number of spaces or tabs can separate the various fields in the statement. To the assembler, the following two code sequences are identical:

```
mov     ax, 0
mov     bx, ax
add     ax, dx
mov     cx, ax
mov      ax, 0
mov      bx, ax
add       ax, dx
mov      cx, ax
```

The first code sequence is much easier to read than the second. With respect to readability, the judicial use of spacing within your program can make all the difference in the world.

Assembly language programs are hard enough to read as it is. Formatting your listings to help make them easier to read will make them much easier to maintain.

You may have a comment on the line by itself. In such a case, place the semicolon in column one and use the entire line for the comment, examples:

```
; The following section of code positions the cursor to the upper
; left hand position on the screen:
```

```
        mov     X, 0
        mov     Y, 0

; Now clear from the current cursor position to the end of the
; screen to clear the video display:

;               etc.
```

3.1.4 Assembly Language Structure

In assembly language code lines have two parts, the first one is the name of the instruction which is to be executed, and the second one are the parameters of the command. For example: add ah bh

Here "add" is the command to be executed; in this case an addition, and "ah" as well as "bh" are the parameters.

For example: mov al, 25

In the above example, we are using the instruction mov, it means move the value 25 to al register.

The name of the instructions in this language is made of two, three or four letters. These instructions are also called mnemonic names or operation codes, since they represent a function the processor will perform.

Sometimes instructions are used as follows:
add al,[170]

The brackets in the second parameter indicate to us that we are going to work with the content of the memory cell number 170 and not with the 170 value; this is known as direct addressing.

3.1.5 Using Debug Program

Program Creation Process

For the creation of a program it is necessary to follow five steps:

Design of the algorithm, stage the problem to be solved is established and the best solution is proposed, creating squematic diagrams used for the better solution proposal. Coding the algorithm, consists in writing the program in some programming language; assembly language in this specific case, taking as a base the proposed solution on the prior step. Translation to machine language is the creation of the object program, in

other words, the written program as a sequence of zeros and ones that can be interpreted by the processor. Test the program, after the translation the program into machine language, execute the program in the computer machine. The last stage is the elimination of detected faults on the program on the test stage. The correction of a fault normally requires the repetition of all the steps from the first or second.

Debug Program

To create a program in assembler two options exist, the first one is to use the TASM or Turbo Assembler, of Borland, and the second one is to use the debugger - on this first section we will use this last one since it is found in any PC with the MS-DOS, which makes it available to any user who has access to a machine with these characteristics.

Debug can only create files with a .COM extension, and because of the characteristics of these kinds of programs they cannot be larger than 64 kb, and they also must start with displacement, offset, or 0100H memory direction inside the specific segment.

Debug provides a set of commands that lets you perform a number of useful operations:

A	Assemble symbolic instructions into machine code
D	Display the contents of an area of memory
E	Enter data into memory, beginning at a specific location
G	Run the executable program in memory
N	Name a program
P	Proceed, or execute a set of related instructions
Q	Quit the debug program
R	Display the contents of one or more registers
T	Trace the contents of one instruction
U	Unassembled machine code into symbolic code
W	Write a program onto disk

It is possible to visualise the values of the internal registers of the CPU using the Debug program. To begin working with Debug, type the following prompt in your computer:

C:/>Debug [Enter]

On the next line a dash will appear, this is the indicator of Debug, at this moment the instructions of Debug can be introduced using the following command:

-r[Enter]

AX=0000 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0D62 ES=0D62 SS=0D62 CS=0D62 IP=0100 NV EI PL NZ NA PO NC
0D62:0100 2E CS:

0D62:0101 803ED3DF00 CMP BYTE PTR [DFD3],00 CS:DFD3=03

All the contents of the internal registers of the CPU are displayed; an alternative of viewing them is to use the "r" command using as a parameter the name of the register whose value wants to be seen. For example:

-rbx
BX 0000
:
This instruction will only display the content of the BX register and the Debug indicator changes from "-" to ":"

When the prompt is like this, it is possible to change the value of the register which was seen by typing the new value and [Enter], or the old value can be left by pressing [Enter] without typing any other value.

3.1.6 Creating Basic Assembly Language Program

The first step is to initiate the Debug, this step only consists of typing debug [Enter] on the operative system prompt.

To assemble a program on the Debug, the "a" (assemble) command is used; when this command is used, the address where you want the assembling to begin can be given as a parameter, if the parameter is omitted the assembling will be initiated at the locality specified by CS:IP, usually 0100h, which is the locality where programs with .COM extension must be initiated. And it will be the place we will use since only Debug can create this specific type of programs.

Even though at this moment it is not necessary to give the "a" command a parameter, it is recommendable to do so to avoid problems once the CS:IP registers are used, therefore we type:

a 100[enter]
mov ax,0002[enter]
mov bx,0004[enter]
add ax,bx[enter]
nop[enter][enter]

What does the program do?, move the value 0002 to the ax register, move the value 0004 to the bx register, add the contents of the ax and bx registers, the instruction, no operation, to finish the program.

In the debug program.

After to do this, appear on the screen some like the follow lines:

```
C:\>debug
-a 100
0D62:0100 mov ax,0002
0D62:0103 mov bx,0004
0D62:0106 add ax,bx
0D62:0108 nop
0D62:0109
```

Type the command "t" (trace), to execute each instruction of this program, example:

```
-t
AX=0002 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000
DI=0000

DS=0D62 ES=0D62 SS=0D62 CS=0D62 IP=0103 NV EI PL NZ NA
PO NC

0D62:0103 BB0400 MOV BX,0004
```

You see that the value 2 move to AX register. Type the command "t" (trace), again, and you see the second instruction is executed.

```
-t
AX=0002 BX=0004 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000
DI=0000

DS=0D62 ES=0D62 SS=0D62 CS=0D62 IP=0106 NV EI PL NZ NA
PO
 NC
0D62:0106 01D8 ADD AX,BX
```

Type the command "t" (trace) to see if the instruction add is executed, you will see the follow lines:

```
-t
AX=0006 BX=0004 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000
DI=0000
```

DS=0D62 ES=0D62 SS=0D62 CS=0D62 IP=0108 NV EI PL NZ NA PE NC

0D62:0108 90 NOP

The possibility that the registers contain different values exists, but AX and BX must be the same, since they are the ones we just modified.

To exit Debug use the "q" (quit) command.

Storing and Loading the Programs

It would not seem practical to type an entire program each time it is needed, and to avoid this it is possible to store a program on the disk, with the enormous advantage that by being already assembled it will not be necessary to run Debug again to execute it.

The steps to save a program that it is already stored on memory are:

Obtain the length of the program subtracting the final address from the initial address, naturally in hexadecimal system. Give the program a name and extension. Put the length of the program on the CX register. Order Debug to write the program on the disk.

By using as an example the following program, we will have a clearer idea of how to take these steps:

When the program is finally assembled it would look like this:

0C1B:0100 mov ax,0002
0C1B:0103 mov bx,0004
0C1B:0106 add ax,bx
0C1B:0108 int 20
0C1B:010A

To obtain the length of a program the "h" command is used, since it will show us the addition and subtraction of two numbers in hexadecimal. To obtain the length of ours, we give it as parameters the value of our program's final address (10A), and the program's initial address (100). The first result the command shows us is the addition of the parameters and the second is the subtraction.

-h 10a 100
020a 000a

The "n" command allows us to name the program.

-n test.com

The "rcx" command allows us to change the content of the CX register to the value we obtained from the size of the file with "h", in this case 000a, since the result of the subtraction of the final address from the initial address.

-rcx
CX 0000
:000a

Lastly, the "w" command writes our program on the disk, indicating how many bytes it wrote.

-w
Writing 000A bytes

To save an already loaded file two steps are necessary:

Give the name of the file to be loaded.

Load it using the "l" (load) command.

To obtain the correct result of the following steps, it is necessary that the above program be already created.

Inside Debug we write the following:
-n test.com
-l
-u 100 109

0C3D:0100 B80200 MOV AX,0002
0C3D:0103 BB0400 MOV BX,0004
0C3D:0106 01D8 ADD AX,BX
0C3D:0108 CD20 INT 20

The last "u" command is used to verify that the program was loaded on memory. What it does is that it disassembles the code and shows it disassembled. The parameters indicate to Debug from where and to where to disassemble.

Debug always loads the programs on memory on the address 100H, otherwise indicated.

3.1.7 Building Assembly Language Programs

In order to be able to create a program, several tools are needed:

First, an editor to create the source program; second, a compiler, which is nothing more than a program that "translates" the source program into an object program. And third, a linker that generates the executable program from the object program.

The editor can be any text editor at hand, and as a compiler we will use the TASM macro assembler from Borland, and as a linker we will use the Tlink program.

The extension used so that TASM recognises the source programs in assembler is .ASM; once translated the source program, the TASM creates a file with the .OBJ extension, this file contains an "intermediate format" of the program, called like this because it is not executable yet but it is not a program in source language either anymore. The linker generates, from a.OBJ or a combination of several of these files, an executable program, whose extension usually is .EXE though it can also be .COM, depending of the form it was assembled.

3.1.8 Assembly Language Programming

To build assembler programs using TASM programs is a different program structure than from using debug program.

It's important to include the following assembler directives:

.MODEL SMALL
Assembler directive that defines the memory model to use in the program

.CODE
Assembler directive that defines the program instructions
.STACK
Assembler directive that reserves a memory space for program instructions
in the stack
END
Assembler directive that finishes the assembler program
Let's program

First Step

use any editor program to create the source file. Type the following lines:

```
first example
; use ; to put comments in the assembler program
.MODEL SMALL; memory model
.STACK; memory space for program instructions in the stack
.CODE; the following lines are program instructions
mov ah,1h; moves the value 1h to register ah
mov cx,07h;moves the value 07h to register cx
int 10h;10h interruption
mov ah,4ch;moves the value 4 ch to register ah
int 21h;21h interruption
END; finishes the program code
```

This assembler program changes the size of the computer cursor.

Second Step

Save the file with the following name: examp1.asm Don't forget to save this in ASCII format.

Third Step

Use the TASM program to build the object program.
Example:

```
C:\>tasm exam1.asm
Turbo Assembler Version 2.0 Copyright (c) 1988, 1990 Borland International

Assembling file: exam1.asm
Error messages: None
Warning messages: None
Passes: 1
Remaining memory: 471k
```

The TASM can only create programs in .OBJ format, which are not executable by themselves, but rather it is necessary to have a linker which generates the executable code.

Fourth Step

Use the TLINK program to build the executable program example:
C:\>tlink exam1.obj

Turbo Link Version 3.0 Copyright (c) 1987, 1990 Borland International
C:\>

Where exam1.obj is the name of the intermediate program, .OBJ. This generates a file directly with the name of the intermediate program and the .EXE extension.

Fifth Step

Execute the executable program
C:\>exam1[enter]

Remember, this assembler program changes the size of the cursor.

Another example

First Step

use any editor program to create the source file. Type the following lines:

```
;example11
.model small
.stack
.code
mov ah,2h ;moves the value 2h to register ah
mov dl,2ah ;moves de value 2ah to register dl
;(Its the asterisk value in ASCII format)
int 21h ;21h interruption
mov ah,4ch ;4ch function, goes to operating system
int 21h ;21h interruption
end ;finishes the program code
```

Second Step

Save the file with the following name: exam2.asm
Don't forget to save this in ASCII format.

Third Step

Use the TASM program to build the object program.
C:\>tasm exam2.asm
Turbo Assembler Version 2.0 Copyright (c) 1988, 1990 Borland International
Assembling file: exam2.asm
Error messages: None
Warning messages: None
Passes: 1
Remaining memory: 471k

Fourth Step

Use the TLINK program to build the executable program
C:\>tlink exam2.obj
Turbo Link Version 3.0 Copyright (c) 1987, 1990 Borland International
C:\>

Fifth Step

Execute the executable program
C:\>ejem11[enter]
*
C:\>

This assembler program shows the asterisk character on the computer screen

3.1.9 Assembly Process

Segments

The architecture of the x86 processors forces to the use of memory segments to manage the information, the size of these segments is of 64kb.

The reason of being of these segments is that, considering that the maximum size of a number that the processor can manage is given by a word of 16 bits or register, it would not be possible to access more than 65536 localities of memory using only one of these registers, but now, if the PC's memory is divided into groups or segments, each one of 65536 localities, and we use an address on an exclusive register to find each segment, and then we make each address of a specific slot with two registers, it is possible for us to access a quantity of 4294967296 bytes

of memory, which is, in the present day, more memory than what we will see installed in a PC.

In order for the assembler to be able to manage the data, it is necessary that each piece of information or instruction be found in the area that corresponds to its respective segments. The assembler accesses this information taking into account the localisation of the segment, given by the DS, ES, SS and CS registers and inside the register the address of the specified piece of information. It is because of this that when we create a program using the Debug on each line that we assemble, something like this appears:

1CB0:0102 MOV AX, BX

Where the first number, 1CB0, corresponds to the memory segment being used, the second one refers to the address inside this segment, and the instructions which will be stored from that address follow. The way to indicate to the assembler with which of the segments we will work with is with the .CODE, .DATA and .STACK directives.

The assembler adjusts the size of the segments taking as a base the number of bytes each assembled instruction needs, since it would be a waste of memory to use the whole segments. For example, if a program only needs 10kb to store data, the data segment will only be of 10kb and not the 64kb it can handle.

Symbols Chart

Each one of the parts on code line in assembler is known as token, for example on the code line:

MOV AX,Var

We have three tokens, the MOV instruction, the AX operator, and the VAR operator. What the assembler does to generate the OBJ code is to read each one of the tokens and look for it on an internal "equivalence" chart known as the reserved words chart, which is where all the mnemonic meanings we use as instructions are found. Following this process, the assembler reads MOV, looks for it on its chart and identifies it as a processor instruction. Likewise it reads AX and recognises it as a register of the processor, but when it looks for the Var token on the reserved words chart, it does not find it, so then it looks for it on the symbols chart which is a table where the names of the variables, constants and labels used in the program where their addresses on memory are included and the sort of data it contains, are found. Sometimes the assembler comes on a token which is not defined on the

program, therefore what it does in these cased is to pass a second time by the source program to verify all references to that symbol and place it on the symbols chart. There are symbols which the assembler will not find since they do not belong to that segment and the program does not know in what part of the memory it will find that segment, and at this time the linker comes into action, which will create the structure necessary for the loader so that the segment and the token be defined when the program is loaded and before it is executed.

SELF-ASSESSMENT EXERCISE

Explain extensively in your own words the processes/steps to take in writing a program in assembly language.

4.0 CONCLUSION

Assemblers are available for just about every processor ever made. Native assemblers produce object code on the same hardware that the object code will run on. Cross assemblers produce object code on different hardware that the object code will run on.

The first reason to work with assembler is that it provides the opportunity of knowing more the operation of your PC, which allows the development of software in a more consistent manner.

The second reason is the total control of the PC which you can have with the use of the assembler. Another reason is that the assembly programs are quicker, smaller, and have larger capacities than ones created with other languages. Lastly, the assembler allows an ideal optimisation in programs, be it on their size or on their execution.

Assembly language statements in a source file use the following format:

{Label} {Mnemonic} {Operand}} {; Comment}

Each assembly language statement appears on its own line in the source file. You cannot have multiple assembly language statements on a single line.

In assembly language, code lines have two parts, the first one is the name of the instruction which is to be executed, and the second one are the parameters of the command.

In order to be able to create a program, several tools are needed: First an editor to create the source program. Second a compiler, which is nothing more than a program that "translates" the source program into an object

program. And third, a linker that generates the executable program from the object program.

5.0 SUMMARY

In this unit we introduced assembly languages, what they are and how to write programs using assembly language.

6.0 TUTOR-MARKED ASSIGNMENT

1. Explain in your own words what the program below does:

a 100[enter]
mov ax,0002[enter]
mov bx,0004[enter]
add ax,bx[enter]
nop[enter][enter].

Then write a similar program and explain what the program does.

2. What provides a set of commands that lets you perform a number of useful operations? Name the operations and explain what they do.

7.0 REFERENCES/FURTHER READING

Baase, Sara (1983). *VAX-11 Assembly Language Programming*. San Diego State University. Prentice-Hall, Inc.

Emerson, W. Pugh, Lyle, R. Johnson & John, H. Palmer (1991). *IBM's 360 and Early 370 systems*. MIT Press. p. 10.

Ford, William & Topp, William (1996). *Assembly Language and Systems Programming for the M68000 Family*. Jones & Bartlett Pub.

Ford, William & Topp, William, (1992). *Macintosh Assembly System, Version 2.0*. Jones & Bartlett Pub.

Sanchez, Julio & Maria, P. Canton (1990). *IBM Microcomputers: A Programmer's Handbook*. McGraw Hill.

Struble, George W. (1975). *Assembler Language Programming The IBM System/360 and 370.* Addison-Wesley Publishing Company.

www.cs.siu.edu

www.educypedia.be/electronics

www.books.google.com

UNIT 2 BRANCHING LOOPS & SUBROUTINES

CONTENTS

1.0 INTRODUCTION

This unit discusses program flow control instructions in assembly language.

2.0 OBJECTIVES

At the end of this unit, you should be able to:

- describe the program flow control instructions of assembly language
- state the program flow control instructions.

3.0 MAIN CONTENT

3.1 Branching Loops & Subroutines

3.1.1 Types of Instruction

Data Movement

In any program it is necessary to move the data in the memory and in the CPU registers; there are several ways to do this: it can copy data in the memory to some register, from register to register, from a register to a stack, from a stack to a register, to transmit data to external devices as well as vice versa.

This movement of data is subject to rules and restrictions. The following are some of them:

It is not possible to move data from a memory locality to another directly; it is necessary to first move the data of the origin locality to a register and then from the register to the destiny locality.

It is not possible to move a constant directly to a segment register; it first must be moved to a register in the CPU.

It is possible to move data blocks by means of the movs instructions, which copies a chain of bytes or words; movsb which copies n bytes from a locality to another; and movsw copies n words from a locality to another. The last two instructions take the values from the defined addresses by DS:SI as a group of data to move and ES:DI as the new localisation of the data.

To move data there are also structures called batteries, where the data is introduced with the push instruction and are extracted with the pop instruction.
In a stack the first data to be introduced is the last one we can take, this is, if in our program we use these instructions:

```
PUSH AX
PUSH BX
PUSH CX
```

To return the correct values to each register at the moment of taking them from the stack it is necessary to do it in the following order:

```
POP CX
POP BX
POP AX
```

For the communication with external devices the out command is used to send information to a port and the in command to read the information received from a port.

The syntax of the out command is:

```
OUT DX,AX
```

Where DX contains the value of the port which will be used for the communication and AX contains the information which will be sent.

The syntax of the in command is:

```
IN AX,DX
```

Where AX is the register where the incoming information will be kept and DX contains the address of the port by which the information will arrive.

3.1.2 Jumps, Loops and Procedure

The unconditional jumps in a written program in assembler language are given by the jmp instruction; a jump is to moves the flow of the execution of a program by sending the control to the indicated address.

A loop, known also as iteration, is the repetition of a process a certain number of times until a condition is fulfilled.

A procedure is a collection of instructions to which we can direct the flow of our program, and once the execution of these instructions is over control is given back to the next line to process of the code which called on the procedure.

3.1.3 Program Flow Control Instructions

The instructions discussed thus far execute sequentially; that is, the CPU executes each instruction in the sequence it appears in your program. To write real programs requires several control structures, not just the sequence. Examples include the if statement, loops, and subroutine invocation (a call). Since compilers reduce all other languages to assembly language, it should come as no surprise that assembly language supports the instructions necessary to implement these control structures. 80x86 program control instructions belong to three groups: unconditional transfers, conditional transfers, and subroutine call and return instructions. The following sections describe these instructions:

1. Unconditional Jumps

The jmp (jump) instruction unconditionally transfers control to another point in the program. There are six forms of this instruction: an intersegment/direct jump, two intrasegment/direct jumps, an intersegment/indirect jump, and two intrasegment/indirect jumps. Intrasegment jumps are always between statements in the same code segment. Intersegment jumps can transfer control to a statement in a different code segment.

These instructions generally use the same syntax, it is jmp target.

For example, the following short little loop continuously reads the parallel printer data port and inverts the L.O. bit. This produces a square wave electrical signal on one of the printer port output lines:

```
mov dx,378h   ;Parallel printer port address.
LoopForever: in al, dx ;Read character from input port.
xor  al,1 ;Invert the L.O. bit.
out dx, al ;Output data back to port.
jmp LoopForever    ;Repeat forever.
```

2. The CALL and RET Instructions

The call and ret instructions handle subroutine calls and returns. There are five different call instructions and six different forms of the return instruction:

```
call    disp16  ;direct intrasegment, 16 bit relative.
call    adrs32  ;direct intersegment, 32 bit segmented address.
call    mem16 ;indirect intrasegment, 16 bit memory pointer.
call    reg16    ;indirect intrasegment, 16 bit register pointer.
call    mem32 ;indirect intersegment, 32 bit memory pointer.

ret             ;near or far return
retn            ;near return
retf            ;far return
ret     disp    ;near or far return and pop
retn    disp    ;near return and pop
retf    disp    ;far return and pop
```

The call instructions take the same forms as the jmp instructions except there is no short (two byte) intrasegment call.

The far call instruction does the following:

- It pushes the cs register onto the stack.
- It pushes the 16 bit offset of the next instruction following the call onto the stack.
- It copies the 32 bit effective address into the cs:ip registers. Since the call instruction allows the same addressing modes as jmp, call can obtain the target address using a relative, memory, or register addressing mode.
- Execution continues at the first instruction of the subroutine. This first instruction is the opcode at the target address computed in the previous step.

The near call instruction does the following:

- It pushes the 16 bit offset of the next instruction following the call onto the stack.
- It copies the 16 bit effective address into the ip register. Since the call instruction allows the same addressing modes as jmp, call can obtain the target address using a relative, memory, or register addressing mode.
- Execution continues at the first instruction of the subroutine. This first instruction is the opcode at the target address computed in the previous step.

The ret (return) instruction returns control to the caller of a subroutine. It does so by popping the return address off the stack and transferring control to the instruction at this return address. Intrasegment (near) returns pop a 16 bit return address off the stack into the ip register. An intersegment (far) return pops a 16 bit offset into the ip register and then a 16 bit segment value into the cs register. These instructions are effectively equal to the following:

```
retn: pop ip
retf: popd cs:ip
```

Clearly, you must match a near subroutine call with a near return and a far subroutine call with a corresponding far return. If you mix near calls with far returns or vice versa, you will leave the stack in an inconsistent state and you probably will not return to the proper instruction after the call. Of course, another important issue when using the call and ret instructions is that you must make sure your subroutine doesn't push something onto the stack and then fail to pop it off before trying to return to the caller. Stack problems are a major cause of errors in assembly language subroutines. Consider the following code:

```
Subroutine: push    ax
push    bx
.
.
.
pop bx
ret
.
.
.
call    Subroutine
```

The call instruction pushes the return address onto the stack and then transfers control to the first instruction of subroutine. The first two push instructions push the ax and bx registers onto the stack, presumably in order to preserve their value because subroutine modifies them. Unfortunately, a programming error exists in the code above, subroutine only pops bx from the stack, it fails to pop ax as well. This means that when subroutine tries to return to the caller, the value of ax rather than the return address is sitting on the top of the stack. Therefore, this subroutine returns control to the address specified by the initial value of the ax register rather than to the true return address. Like the call instruction, it is very easy to simulate the ret instruction using two 80x86 instructions. All you need to do is pop the return address off the stack and then copy it into the ip register. For near returns, this is a very simple operation, just pop the near return address off the stack and then jump indirectly through that register:

```
pop     ax
jmp     ax
```

Simulating a far return is a little more difficult because you must load cs:ip in a single operation. The only instruction that does this (other than a far return) is the jmp mem32 instruction.

3. The Conditional Jump Instructions

Although the jmp, call, and ret instructions provide transfer of control, they do not allow you to make any serious decisions. The 80x86's conditional jump instructions handle this task. The conditional jump instructions are the basic tool for creating loops and other conditionally executable statements like the “if … then” statement.

The conditional jumps test one or more flags in the flags register to see if they match some particular pattern (just like the setcc instructions). If the pattern matches, control transfers to the target location. If the match fails, the CPU ignores the conditional jump and execution continues with the next instruction. Some instructions, for example, test the conditions of the sign, carry, overflow, and zero flags.

One thing nice about conditional jumps is that you do not flush the pipeline or the prefetch queue if you do not take the branch. If one condition is true far more often than the other, you might want to use the conditional jump to transfer control to a jmp nearby, so you can continue to fall through as before. For example, if you have a je target instruction and target is out of range, you could convert it to the following code:

```
je      GotoTarget
```

```
.
.
          .
GotoTarget:     jmp     Target
```

Although a branch to target now requires executing two jumps, this is much more efficient than the standard conversion if the zero flag is normally clear when executing the je instruction.

The 80386 and later processor provide an extended form of the conditional jump that is four bytes long, with the last two bytes containing a 16 bit displacement. These conditional jumps can transfer control anywhere within the current code segment. Therefore, there is no need to worry about manually extending the range of the jump. If you've told MASM you're using an 80386 or later processor, it will automatically choose the two byte or four byte form, as necessary. The 80x86 conditional jump instruction give you the ability to split program flow into one of two paths depending upon some logical condition. Suppose you want to increment the ax register if bx is or equal to cx. You can accomplish this with the following code:

```
cmp   bx, cx
jne     SkipStmts
inc     ax
SkipStmts:
```

The trick is to use the opposite branch to skip over the instructions you want to execute if the condition is true. Always use the "opposite branch (N/no N)" rule given earlier to select the opposite branch. You can make the same mistake choosing an opposite branch here as you could when extending out of range jumps.

You can also use the conditional jump instructions to synthesise loops. For example, the following code sequence reads a sequence of characters from the user and stores each character in successive elements of an array until the user presses the Enter key (carriage return):

```
mov      di, 0
ReadLnLoop:     mov      ah, 0           ;INT 16h read key opcode.
int         16h
mov       Input[di], al
inc         di
cmp       al, 0dh          ;Carriage return ASCII code.
jne      ReadLnLoop
mov     Input[di-1],0   ;Replace carriage return with zero.
```

1. The LOOP Instruction

This instruction decrements the cx register and then branches to the target location if the cx register does not contain zero. Since this instruction decrements cx then checks for zero, if cx originally contained zero, any loop you create using the loop instruction will repeat 65,536 times. If you do not want to execute the loop when cx contains zero, use jcxz to skip over the loop.

There is no "opposite" form of the loop instruction, and like the jcxz/jecxz instructions the range is limited to ±128 bytes on all processors. If you want to extend the range of this instruction, you will need to break it down into discrete components:

; "loop lbl" becomes:

```
dec     cx
jne     lbl
```

You can easily extend this jne to any distance.

There is no eloop instruction that decrements ecx and branches if not zero (there is a loope instruction, but it does something else entirely). The reason is quite simple. As of the 80386, Intel's designers stopped wholeheartedly supporting the loop instruction. Oh, it's there to ensure compatibility with older code, but it turns out that the dec/jne instructions are actually faster on the 32 bit processors. Problems in the decoding of the instruction and the operation of the pipeline are responsible for this strange turn of events.

Although the loop instruction's name suggests that you would normally create loops with it, keep in mind that all it is really doing is decrementing cx and branching to the target address if cx does not contain zero after the decrement. You can use this instruction anywhere you want to decrement cx and then check for a zero result, not just when creating loops. Nonetheless, it is a very convenient instruction to use if you simply want to repeat a sequence of instructions some number of times. For example, the following loop initialises a 256 element array of bytes to the values 1, 2, 3, ...

```
mov     ecx, 255
ArrayLp:        mov     Array[ecx], cl
loop    ArrayLp
mov     Array[0], 0
```

The last instruction is necessary because the loop does not repeat when cx is zero. Therefore, the last element of the array that this loop processes is Array[1], hence the last instruction.

The loop instruction does not affect any flags.

2. The LOOPE/LOOPZ Instruction

Loope/loopz (loop while equal/zero, they are synonyms for one another) will branch to the target address if cx is not zero and the zero flag is set. This instruction is quite useful after cmp or cmps instruction, and is marginally faster than the comparable 80386/486 instructions if you use all the features of this instruction. However, this instruction plays havoc with the pipeline and superscalar operation of the Pentium so you're probably better off sticking with discrete instructions rather than using this instruction. This instruction does the following:

cx := cx - 1
if ZeroFlag = 1 and cx 0, goto target

The loope instruction falls through on one of two conditions. Either the zero flag is clear or the instruction decremented cx to zero. By testing the zero flag after the loop instruction (with a je or jne instruction, for example), you can determine the cause of termination.

This instruction is useful if you need to repeat a loop while some value is equal to another, but there is a maximum number of iterations you want to allow. For example, the following loop scans through an array looking for the first non-zero byte, but it does not scan beyond the end of the array:

```
mov     cx, 16              ;Max 16 array elements.
mov      bx, -1             ;Index into the array (note next inc).
SearchLp: inc        bx     ; Move on to next array element.
cmp      Array[bx], 0       ;See if this element is zero.
loope    SearchLp           ;Repeat if it is.
je       AllZero            ;Jump if all elements were zero.
```

Note that this instruction is not the opposite of loopnz/loopne. If you need to extend this jump beyond ±128 bytes, you will need to synthesise this instruction using discrete instructions. For example, if loope target is out of range, you would need to use an instruction sequence like the following:

```
jne     quit
dec     cx
```

```
je      Quit2
jmp    Target
quit:       dec     cx              ;loope decrements cx, even if ZF=0.
quit2:
```

The loope/loopz instruction does not affect any flags.

3. The LOOPNE/LOOPNZ Instruction

This instruction is just like the loope/loopz instruction except loopne/loopnz (loop while not equal/not zero) repeats while cx is not zero and the zero flag is clear. The algorithm is

```
cx := cx - 1
if ZeroFlag = 0 and cx  0, goto target
```

You can determine if the loopne instruction terminated because cx was zero or if the zero flag was set by testing the zero flag immediately after the loopne instruction. If the zero flag is clear at that point, the loopne instruction fell through because it decremented cx to zero. Otherwise it fell through because the zero flag was set.

This instruction is not the opposite of loope/loopz. If the target address is out of range, you will need to use an instruction sequence like the following:

```
je      quit
dec     cx
je      Quit2
jmp    Target
quit:       dec      cx              ;loopne decrements cx, even if ZF=1.
quit2:
```

You can use the loopne instruction to repeat some maximum number of times while waiting for some other condition to be true. For example, you could scan through an array until you exhaust the number of array elements or until you find a certain byte using a loop like the following:

```
mov     cx, 16  ;Maximum # of array elements.
mov     bx, -1  ;Index into array.
LoopWhlNot0:    inc   bx  ;Move on to next array element.
cmp     Array[bx],0     ;Does this element contain zero?
loopne  LoopWhlNot0  ;Quit if it does, or more than 16 bytes.
```

Although the loope/loopz and loopne/loopnz instructions are slower than the individual instruction from which they could be synthesised, there is

one main use for these instruction forms where speed is rarely important; indeed, being faster would make them less useful - timeout loops during I/O operations. Suppose bit #7 of input port 379h contains a one if the device is busy and contains a zero if the device is not busy. If you want to output data to the port, you could use code like the following:

```
mov  dx, 379h
WaitNotBusy:      in     al, dx       ;Get port
test   al, 80h                 ;See if bit #7 is one
jne    WaitNotBusy         ;Wait for "not busy"
```

The only problem with this loop is that it is conceivable that it would loop forever. In a real system, a cable could come unplugged, someone could shut off the peripheral device, and any number of other things could go wrong that would hang up the system. Robust programs usually apply a timeout to a loop like this. If the device fails to become busy within some specified amount of time, then the loop exits and raises an error condition. The following code will accomplish this:

```
mov      dx, 379h            ;Input port address
mov      cx, 0               ;Loop 65,536 times and then quit.
WaitNotBusy:      in        al, dx              ;Get data at port.
test       al, 80h           ;See if busy
loopne  WaitNotBusy      ;Repeat if busy and no time out.
jne TimedOut                 ;Branch if CX=0 becos we timed out.
```

You could use the loope/loopz instruction if the bit were zero rather than one.

The loopne/loopnz instruction does not affect any flags.

Program loops consist of three components: an optional initialisation component, a loop termination test, and the body of the loop. The order with which these components are assembled can dramatically change the way the loop operates. Three permutations of these components appear over and over again. Because of their frequency, these loop structures are given special names in HLLs: `while` loops, `repeat..until` loops (`do..while` in C/C++), and `loop..endloop` loops.

4. While Loops

The most general loop is the while loop. It takes the following form:
WHILE boolean expression DO statement;

There are two important points to note about the while loop. First, the test for termination appears at the beginning of the loop. Second as a direct consequence of the position of the termination test, the body of the loop may never execute. If the termination condition always exists, the loop body will always be skipped over.

Consider the following Pascal while loop:

```
I := 0;

WHILE (I<100) do I := I + 1;
```

`I := 0;` is the initialisation code for this loop. I is a loop control variable, because it controls the execution of the body of the loop. `(I<100)` is the loop termination condition. That is, the loop will not terminate as long as I is less than 100. `I:=I+1;` is the loop body. This is the code that executes on each pass of the loop. You can convert this to 80x86 assembly language as follows:

```
                mov     I, 0
WhileLp:        cmp     I, 100
                jge     WhileDone
                inc     I
                jmp     WhileLp

WhileDone:
```

Note that a Pascal while loop can be easily synthesised using an if and a goto statement. For example, the Pascal while loop presented above can be replaced by:

```
I := 0;
1:      IF (I<100) THEN BEGIN
I := I + 1;
GOTO 1;
END;
```

More generally, any while loop can be built up from the following: optional initialisation code
1: IF not termination condition THEN BEGIN
loop body
GOTO 1;
END;

5. Repeat..Until Loops

The repeat..until (do..while) loop tests for the termination condition at the end of the loop rather than at the beginning. In Pascal, the repeat..until loop takes the following form:

optional initialisation code
REPEAT
loop body
UNTIL termination condition

This sequence executes the initialisation code, the loop body, then tests some condition to see if the loop should be repeated. If the boolean expression evaluates to false, the loop repeats; otherwise the loop terminates. The two things to note about the `repeat..until` loop is that the termination test appears at the end of the loop and, as a direct consequence of this, the loop body executes at least once.

Like the `while` loop, the `repeat..until` loop can be synthesised with an `if` statement and a `goto` . You would use the following:
initialisation code
1: loop body
IF NOT termination condition THEN GOTO 1
Based on the example above, you can easily synthesise `repeat..until` loops in assembly language.

6. LOOP..ENDLOOP Loops

If `while` loops test for termination at the beginning of the loop and `repeat..until` loops check for termination at the end of the loop, the only place left to test for termination is in the middle of the loop. Although Pascal and C/C++ don't directly support such a loop, the `loop..endloop` structure can be found in HLL languages like Ada. The `loop..endloop` loop takes the following form:

LOOP
loop body
ENDLOOP;

Note that there is no explicit termination condition. Unless otherwise provided for, the `loop..endloop` construct simply forms an infinite loop. Loop termination is handled by an `if` and `goto` statement. Consider the following (pseudo) Pascal code which employs a `loop..endloop` construct:

```
LOOP
READ(ch)
IF ch = '.' THEN BREAK;
WRITE(ch);
ENDLOOP;
```

In real Pascal, you'd use the following code to accomplish this:

```
1:
READ(ch);
IF ch = '.' THEN GOTO 2; (* Turbo Pascal supports BREAK! *)
WRITE(ch);
GOTO 1

2:
```

In assembly language you'd end up with something like:

```
LOOP1: getc
cmp     al, '.'
je      EndLoop
putc
jmp     LOOP1
EndLoop:
```

7. FOR Loops

The `for` loop is a special form of the `while` loop which repeats the loop body a specific number of times. In Pascal, the `for` loop looks something like the following:

```
FOR var := initial TO final DO stmt
or
FOR var := initial DOWNTO final DO stmt
```

Traditionally, the `for` loop in Pascal has been used to process arrays and other objects accessed in sequential numeric order. These loops can be converted directly into assembly language as follows:

In Pascal:

```
FOR var := start TO stop DO stmt;
```

In Assembly:

```
mov     var, start
FL:         mov     ax, var
cmp     ax, stop
jg          EndFor
```

```
; code corresponding to stmt goes here.

        inc     var
        jmp     FL

EndFor:
```

Fortunately, most for loops repeat some statement(s) a fixed number of times. For example,

FOR I := 0 to 7 do write(ch);

In situations like this, it's better to use the 80x86 loop instruction (or corresponding dec cx/jne sequence) rather than simulate a for loop:

```
mov     cx, 7
LP:         mov     al, ch
call      putc
loop    LP
```

Keep in mind that the loop instruction normally appears at the end of a loop whereas the for loop tests for termination at the beginning of the loop. Therefore, you should take precautions to prevent a runaway loop in the event cx is zero (which would cause the loop instruction to repeat the loop 65,536 times) or the stop value is less than the start value. In the case of

FOR var := start TO stop DO stmt;

assuming you don't use the value of var within the loop, you'd probably want to use the assembly code:

```
mov     cx, stop
sub     cx, start
jl        SkipFor
inc      cx
LP:         stmt
loop    LP
SkipFor:
```

Remember, the sub and cmp instructions set the flags in an identical fashion. Therefore, this loop will be skipped if stop is less than start. It will be repeated (stop-start)+1 times otherwise. If you need to reference the value of var within the loop, you could use the following code:

```
mov     ax, start
mov     var, ax
mov     cx, stop
sub      cx, ax
jl       SkipFor
inc      cx
LP:        stmt
inc      var
loop     LP
SkipFor:
```

SELF-ASSESSMENT EXCERCISE

Mention all the program flow instructions you know and write short notes on each.

4.0 CONCLUSION

In any program, it is necessary to move the data in the memory and in the CPU registers; there are several ways to do this: it can copy data in the memory to some register, from register to register, from a register to a stack, from a stack to a register, to transmit data to external devices as well as vice versa.

The unconditional jumps in a written program in assembler language are given by the jmp instruction; a jump is to moves the flow of the execution of a program by sending the control to the indicated address.

A loop, known also as iteration, is the repetition of a process a certain number of times until a condition is fulfilled.

A procedure is a collection of instructions to which we can direct the flow of our program, and once the execution of these instructions is over control is given back to the next line to process of the code which called on the procedure.

Program control instructions belong to three groups: unconditional transfers, conditional transfers, and subroutine call and return instructions.

5.0 SUMMARY

In this unit, we learnt about program flow control instructions in assembly language. The movement of data is subject to rules and restrictions. It is not possible to move data from a memory locality to another directly; it is necessary to first move the data of the origin

locality to a register and then from the register to the destiny locality. It is not possible to move a constant directly to a segment register; it first must be moved to a register in the CPU. It is possible to move data blocks by means of the movs instructions, which copies a chain of bytes or words; movsb which copies n bytes from a locality to another; and movsw copies n words from a locality to another.

6.0 TUTOR-MARKED ASSIGNMENT

Mention and explain the different types of program flow control we have.

7.0 REFERENCES/FURTHER READING

Baase, Sara (1983). *VAX-11 Assembly Language Programming*. San Diego State University. Prentice-Hall, Inc.

Emerson, W. Pugh, Lyle, R. Johnson & John, H. Palmer (1991). *IBM's 360 and Early 370 systems*. MIT Press. p. 10.

Ford, William & Topp, William (1996). *Assembly Language and Systems Programming for the M68000 Family*. Jones & Bartlett Pub.

Ford, William & Topp, William, (1992).*Macintosh Assembly System, Version 2.0*. Jones & Bartlett Pub.

Sanchez, Julio & Maria, P. Canton (1990). *IBM Microcomputers: A Programmer's Handbook*. McGraw Hill.

Struble, George W. (1975). *Assembler Language Programming the IBM System/360 and 370.*Addison-Wesley Publishing Company.

www.cs.siu.edu

www.educypedia.be/electronics

www.books.google.com

UNIT 3 SAMPLE PROGRAMS IN ASSEMBLY LANGUAGE

CONTENTS

1.0 INTRODUCTION

This unit gives some simple programs used to demonstrate the use of various instructions in assembly language. It also compares assembly language with high level languages.

2.0 OBJECTIVES

At the end of this unit, you should be able to:

- write some simple programs in assembly language
- compare assembly language and high level languages.

3.0 MAIN CONTENT

3.1 Sample Programs in Assembly Language

The following simple programs demonstrate the use of the various instructions and logical operations in assembly language.

3.1.1 Simple Arithmetic I

This program demonstrates some simple arithmetic instructions in assembly language. The comments after the semi-colon explain each line of program.

```
; Simple Arithmetic
; This program demonstrates some simple arithmetic instructions.

.386  ;So we can use extended registers
 option  segment:use16      ; and addressing modes.
dseg     segment para public 'data'
; Some type definitions for the variables we will declare:

uint          typedef word         ;Unsigned integers.
integer       typedef sword        ;Signed integers.

; Some variables we can use:

j             integer ?
k             integer ?
l             integer ?

u1            uint    ?
u2            uint    ?
u3            uint    ?
dseg          ends

cseg          segment para public 'code'
assume  cs:cseg, ds:dseg

Main      proc
mov     ax, dseg
mov     ds, ax
mov     es, ax

; Initialise our variables:

mov     j, 3
mov     k, -2

mov     u1, 254
mov     u2, 22

; Compute L := j+k and u3 := u1+u2

mov     ax, J
add     ax, K
mov     L, ax

mov     ax, u1          ;Note that we use the "ADD"
add     ax, u2          ; instruction for both signed
```

```
mov     u3, ax          ; and unsigned arithmetic.

            ; Compute L := j-k and u3 := u1-u2

mov     ax, J
sub     ax, K
mov     L, ax

mov     ax, u1          ;Note that we use the "SUB"
sub     ax, u2          ; instruction for both signed
mov     u3, ax          ; and unsigned arithmetic.

; Compute L := -L

                neg     L

; Compute L := -J

                mov     ax, J           ;Of course, you would only use the
                neg     ax              ; NEG instruction on signed values.
                mov     L, ax

; Compute K := K + 1 using the INC instruction.

                inc     K

; Compute u2 := u2 + 1 using the INC instruction.
; Note that you can use INC for signed and unsigned values.

                inc     u2

; Compute J := J - 1 using the DEC instruction.

                dec     J

; Compute u2 := u2 - 1 using the DEC instruction.
; Note that you can use DEC for signed and unsigned values.

                dec     u2

Quit:       mov     ah, 4ch         ;DOS opcode to quit program.
                int     21h             ;Call DOS.
Main        endp

cseg        ends
```

```
sseg        segment para stack 'stack'
stk         byte    1024 dup ("stack   ")
sseg        ends

zzzzzzseg segment para public 'zzzzzz'
LastBytes byte    16 dup (?)
zzzzzzseg  ends
             end     Main
```

3.1.2 Simple Arithmetic II

This program demonstrates some simple arithmetic instructions in assembly language. The comments after the semi-colon explain each line of program.

```
; Simple Arithmetic
; This program demonstrates some simple arithmetic instructions.

.386                        ; So we can use extended registers
option  segment:use16               ; and addressing modes.

dseg        segment para public 'data'

; Some type definitions for the variables we will declare:

uint        typedef word            ;Unsigned integers.
integer     typedef sword           ;Signed integers.

; Some variables we can use:

j           integer ?
k           integer ?
l           integer ?

u1          uint    ?
u2          uint    ?
u3          uint    ?
dseg        ends

cseg        segment para public 'code'
             assume  cs:cseg, ds:dseg

Main        proc
            mov     ax, dseg
            mov     ds, ax
            mov     es, ax
```

```
; Initialise our variables:

            mov     j, 3
            mov     k, -2
            mov     u1, 254
            mov     u2, 22

; Extended multiplication using 8086 instructions.
;
; Note that there are separate multiply instructions for signed and
; unsigned operands.
;
; L := J * K (ignoring overflow)

mov    ax, J
imul   K                ;Computes DX:AX := AX * K
mov    L, ax            ;Ignore overflow into DX.

; u3 := u1 * u2

mov    ax, u1
mul    u2               ;Computes DX:AX := AX * U2
mov    u3, ax           ;Ignore overflow in DX.

; Extended division using 8086 instructions.
;
; Like multiplication, there are separate instructions for signed
; and unsigned operands.
;
; It is absolutely imperative that these instruction sequences sign
; extend or zero extend their operands to 32 bits before dividing.
; Failure to do so will may produce a divide error and crash the
; program.
;
; L := J div K

mov    ax, J
cwd             ;*MUST* sign extend AX to DX:AX!
idiv   K        ;AX := DX:AX/K, DX := DX:AX mod K
mov    L, ax

; u3 := u1/u2

mov    ax, u1
mov    dx, 0        ;Must zero extend AX to DX:AX!
div    u2           ;AX := DX:AX/u2, DX := DX:AX
```

```
mod     u2
mov     u3, ax

; Special forms of the IMUL instruction available on 80286, 80386, and
; later processors.  Technically, these instructions operate on signed
; operands only, however, they do work fine for unsigned operands as
well.
; Note that these instructions produce a 16-bit result and set the overflow
; flag if overflow occurs.
;
; L := J * 10 (80286 and later only)

            imul    ax, J, 10       ;AX := J*10
            mov     L, ax

; L := J * K (80386 and later only)

            mov     ax, J
            imul    ax, K
            mov     L, ax


Quit:       mov     ah, 4ch             ;DOS opcode to quit program.
            int     21h                 ;Call DOS.
Main        endp
cseg        ends

sseg        segment para stack 'stack'
stk         byte    1024 dup ("stack   ")
sseg        ends

zzzzzzseg   segment para public 'zzzzzz'
LastBytes   byte    16 dup (?)
zzzzzzseg   ends
            end     Main
```

3.1.3 Logical Operations

This program demonstrates some logical instructions in assembly language. The comments after the semi-colon explain each line of program.

```
; Logical Operations
; This program demonstrates the AND, OR, XOR, and NOT instructions

.386   ;So we can use extended registers
```

```
option  segment:use16   ; and addressing modes.

Dseg segment para public 'data'

; Some variables we can use:

j           word    0FF00h
k           word    0FFF0h
l           word    ?

c1          byte    'A'
c2          byte    'a'

LowerMask   byte    20h

dseg        ends
cseg        segment para public 'code'
            assume  cs:cseg, ds:dseg

Main        proc
            mov     ax, dseg
            mov     ds, ax
            mov     es, ax

; Compute L := J and K (bitwise AND operation):

            mov     ax, J
            and     ax, K
            mov     L, ax

; Compute L := J or K (bitwise OR operation):

            mov     ax, J
            or      ax, K
            mov     L, ax

; Compute L := J xor K (bitwise XOR operation):

            mov     ax, J
            xor     ax, K
            mov     L, ax

; Compute L := not L (bitwise NOT operation):

            not     L
```

```
; Compute L := not J (bitwise NOT operation):

            mov     ax, J
            not     ax
            mov     L, ax

; Clear bits 0..3 in J:

            and     J, 0FFF0h

; Set bits 0..3 in K:

            or      K, 0Fh

; Invert bits 4..11 in L:

            xor     L, 0FF0h

; Convert the character in C1 to lower case:
            mov     al, c1
            or      al, LowerMask
            mov     c1, al

; Convert the character in C2 to upper case:

            mov     al, c2
            and     al, 5Fh             ;Clears bit 5.
            mov     c2, al

Quit:       mov     ah, 4ch             ;DOS opcode to quit program.
            int     21h                 ;Call DOS.
Main        endp

cseg        ends

sseg        segment para stack 'stack'
stk         byte    1024 dup ("stack   ")
sseg        ends

zzzzzzseg   segment para public 'zzzzzz'
LastBytes   byte    16 dup (?)
zzzzzzseg   ends
            end     Main
```

SELF-ASSESSMENT EXERCISE

1. Write in detail what you understood these three programs do.
2. Try to write a simple program in assembly language that can do anything you choose explaining each line of the program.

3.2 Comparison of Assembly and High Level Languages

Perhaps the most glaring difference among the three types of languages [high level, assembly, and machine] is that as we move from high-level languages to lower levels, the code gets harder to read (with understanding). The major advantages of high-level languages are that they are easy to read and are machine independent. The instructions are written in a combination of English and ordinary mathematical notation, and programs can be run with minor, if any, changes on different computers.

Assembly languages are close to a one to one correspondence between symbolic instructions and executable machine codes. Assembly languages also include directives to the assembler, directives to the linker, directives for organising data space, and macros. Macros can be used to combine several assembly language instructions into a high level language-like construct (as well as other purposes). There are cases where a symbolic instruction is translated into more than one machine instruction. But in general, symbolic assembly language instructions correspond to individual executable machine instructions.

High level languages are abstract. Typically a single high level instruction is translated into several (sometimes dozens or in rare cases even hundreds) executable machine language instructions. Some early high level languages had a close correspondence between high level instructions and machine language instructions. For example, most of the early COBOL instructions translated into a very obvious and small set of machine instructions. The trend over time has been for high level languages to increase in abstraction. Modern object oriented programming languages are highly abstract (although, interestingly, some key object oriented programming constructs do translate into a very compact set of machine instructions).

Assembly language is much harder to program than high level languages. The programmer must pay attention to far more detail and must have an intimate knowledge of the processor in use. But high quality hand crafted assembly language programs can run much faster and use much less memory and other resources than a similar program written in a high level language. Speed increases of two to 20 times faster are fairly common, and increases of hundreds of times faster are

occasionally possible. Assembly language programming also gives direct access to key machine features essential for implementing certain kinds of low level routines, such as an operating system kernel or microkernel, device drivers, and machine control.

High level programming languages are much easier for less skilled programmers to work in and for semi-technical managers to supervise. And high level languages allow faster development times than work in assembly language, even with highly skilled programmers. Development time increases of 10 to 100 times faster are fairly common. Programs written in high level languages (especially object oriented programming languages) are much easier and less expensive to maintain than similar programs written in assembly language (and for a successful software project, the vast majority of the work and expense is in maintenance, not initial development).

Assembly languages occupy a unique place in the computing world. Since most assembler-language statements are symbolic of individual machine-language instructions, the assembler-language programmer has the full power of the computer at his disposal in a way that users of other languages do not. Because of the direct relationship between assembler language and machine language, assembler language is used when high efficiency of programs is needed, and especially in areas of application that are so new and amorphous that existing program-oriented languages are ill-suited for describing the procedures to be followed."

If one has a choice between assembly language and a high-level language, why choose assembly language? The fact that the amount of programming done in assembly language is quite small compared to the amount done in high-level languages indicates that one generally doesn't choose assembly language. However, there are situations where it may not be convenient, efficient, or possible to write programs in high-level languages. Programs to control and communicate with peripheral devices (input and output devices) are usually written in assembly language because they use special instructions that are not available in high-level languages, and they must be very efficient. Some systems programs are written in assembly language for similar reasons. In general, since high-level languages are designed without the features of a particular machine in mind and a compiler must do its job in a standardised way to accommodate all valid programs, there are situations where to take advantage of special features of a machine, to program some details that are inaccessible from a high-level language, or perhaps to increase the efficiency of a program, one may reasonably choose to write in assembly language.

In situations where programming in a high-level language is not appropriate, it is clear that assembly language is to be preferred to machine language. Assembly language has a number of advantages over machine code aside from the obvious increase in readability. One is that the use of symbolic names for data and instruction labels frees the programmer from computing and recomputing the memory locations whenever a change is made in a program. Another is that assembly languages generally have a feature, called macros, that frees the [programmer] from having to repeat similar sections of code used in several places in a program. Often compilers translate into assembly language rather than machine code.

4.0 CONCLUSION

The most glaring difference among the three types of languages [high level, assembly, and machine] is that as we move from high-level languages to lower levels, the code gets harder to read (with understanding). The major advantages of high-level languages are that they are easy to read and are machine independent.

5.0 SUMMARY

In this unit we saw some simple programs in assembly language and also made some comparison between assembly and high level languages.

6.0 TUTOR-MARKED ASSIGNMENT

Write 6 differences between assembly and high level languages.

7.0 REFERENCES/FURTHER READING

Baase, Sara (1983). *VAX-11 Assembly Language Programming*. San Diego State University. Prentice-Hall, Inc.

Emerson, W. Pugh, Lyle, R. Johnson & John, H. Palmer (1991). *IBM's 360 and Early 370 Systems*. MIT Press. p. 10.

Ford, William & Topp, William (1996). *Assembly Language and Systems Programming for the M68000 Family*. Jones & Bartlett Pub.

Ford, William & Topp, William, (1992). *Macintosh Assembly System, Version 2.0*. Jones & Bartlett Pub.

Sanchez, Julio & Maria, P. Canton (1990). *IBM Microcomputers: A Programmer's Handbook.* McGraw Hill.

Struble, George W. (1975). *Assembler Language Programming the IBM System/360 and 370.* Addison-Wesley Publishing Company.

www.cs.siu.edu

www.educypedia.be/electronics

www.books.google.com