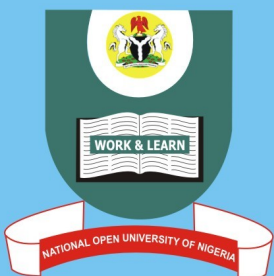


# CIT 246

## INTRODUCTION TO COMPUTER ORGANIZATION



**NATIONAL OPEN UNIVERSITY OF NIGERIA**

<b>COURSE GUIDE</b>
-------------------------

**CIT 246****INTRODUCTION TO COMPUTER ORGANISATION**

Course Adapter

A. A. Afolorunso  
National Open University of Nigeria  
14\16 Ahmadu Bello Way  
Victoria Island  
Lagos

Course Co-ordinator

A. A. Afolorunso  
National Open University of Nigeria  
14\16 Ahmadu Bello Way  
Victoria Island  
Lagos

**NATIONAL OPEN UNIVERSITY OF NIGERIA**

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

Abuja Office  
No 5 Dar es Salaam Street  
Off Aminu Kano Crescent  
Wuse II  
Abuja

e-mail: [centralinfo@nou.edu.ng](mailto:centralinfo@nou.edu.ng)  
URL: [www.nou.edu.ng](http://www.nou.edu.ng)

National Open University of Nigeria

First Printed 2008

ISBN: 978-058-557-5

All Rights Reserved

Printed by:

<b>CONTENTS</b>	<b>PAGE</b>
Introduction.....	1
What You will Learn in this Course.....	1
Course Aims.....	1
Course Objectives.....	1
Working through this Course.....	2
Course Materials.....	2
Study Units .....	3
Textbooks and References .....	3
Assignment File.....	3
Presentation Schedule.....	4
Assessment.....	4
Tutor-Marked Assignment .....	4
Final Examinations and Grading.....	5
Course Marking Scheme.....	5
Course Overview.....	6
How to Get the Best from This Course .....	6
Facilitators/Tutors and Tutorials .....	8
Summary .....	9

## **Introduction**

CIT 246 – Introduction to Computer Organisation is a three credit unit course of eight units. This course presents an overview of the structure and functioning of computer systems. It covers aspects on data representation, interconnection structures, memory system, input-output system, and the organisation of the central processing unit. In addition the concepts of microprocessors, reduced instruction set computers and parallel architecture are discussed.

This course is divided into two modules. The first module deals with the basic architecture of the computer system. it includes discussions on data representation, bus structure, digital logic circuits, random access memory, secondary storage, high-speed memories, input/output devices, and input/output techniques.

The second module focuses on the organisation of the CPU, where the discussions are about the instruction formats, addressing modes, control unit operations, and the micro-programmed control unit.

The aim of this course is to equip you with the basic knowledge you require to understand the inner workings of a computer system. By the end of the course, you should be able to confidently explain the inner features and workings of every component part of the computer hardware system.

This Course Guide gives you a brief overview of the course content, course duration, and course materials.

A course on computers can never be complete because of the existing diversities of the computer system. Therefore, you are advised to read through further readings to enhance the basic understanding you will acquire from the course material.

## **What You will Learn in this Course**

The main purpose of this course is to introduce you to concepts relating to computer organisation and provide you with the indepth knowledge of the internal components of the computer hardware components and their basic functions.. This we intend to achieve through the following:

### **Course Aims**

1. Introduce the concepts relating to Computer Organisation;
2. Expose the basic relationships that exist among the internal components of the computer system and their interactions.

3. Discuss instruction set and the characteristics, addressing schemes and formats of the instruction.
4. Discuss micro-programmed control unit in detail.
5. Expose the basics of digital logic circuits as well as memory organisation.

### **Course Objectives**

Certain objectives have been set out to ensure that the course achieves its aims. Apart from the course objectives, every unit of this course has set objectives. In the course of the study, you will need to confirm, at the end of each unit, if you have met the objectives set at the beginning of the unit. By the end of this course you should be able to:

1. Define the logical structure of the computer
2. Describe data representation in computers
3. Define the concept of Interrupt
4. Define an input/output processor
5. Discuss various types of instructions and differentiate among various types of operands.
6. Discuss the basic organisation of ALU.
7. Explain the working of a microprogrammed control unit.

### **Working through this Course**

In order to have a thorough understanding of the course units, you will need to read and understand the contents and practise the steps by designing a mini system for your department, and be committed to learning and implementing your knowledge.

This course is designed to cover approximately sixteen weeks, and it will require your devoted attention. You should do the exercises in the Tutor-Marked Assignments and submit to your tutors.

### **Course Materials**

These include:

1. The Course Guide
2. Study units
3. Recommended texts
4. A file for your assignments and for records to monitor your progress.

## Study Units

There are eight study units in this course:

### Module 1

Unit 1	Introduction and Data Representation
Unit 2	Digital Logic Circuits
Unit 3	Memory Organisation
Unit 4	Input/Output Organisation

### Module 2

Unit 1	Instruction Sets
Unit 2	Register Organisation And Micro-Operations
Unit 3	ALU And Control Unit Organisation
Unit 4	Microprogrammed Control Unit

Make use of the course materials, do the exercises to enhance your learning.

## Textbooks and References

Mano, M. Morris (1993). *Computer System Architecture* (3<sup>rd</sup> ed) Prentice Hall of India.

Hayes, John P.(1988). *Computer Architecture and Organisation* (2<sup>nd</sup> ed). McGraw-Hill International Editions.

Stallings William. *Computer Organisation and Architecture* (3<sup>rd</sup> ed). Maxwell Macmillan International Editions.

Baron, Robert J. and Higbie Lee.*Computer Architecture* Addison-Wesley Publishing Company.

Tanenbaum, Andrew S.(1993).*Structural Computer Organisation* (3<sup>rd</sup> ed).Printice Hall of India.

## Assignment File

There are two types of assignments: the Self-Assessment Exercises and the Tutor-Marked Assignments. The self-assessment exercises will enable you monitor your performance by yourself, while the tutor-marked assignments will be supervised. The assignments take a certain percentage of your total score in this course. The tutor-marked assignments will be assessed by your tutor within a specified period.

The examination at the end of this course will aim at determining your level of mastery of the subject matter. This course includes eight tutor-marked assignments and each must be done and submitted accordingly. Your best scores however, will be recorded for you. Be sure to send these assignments to your tutor before the deadline to avoid loss of marks.

### **Presentation Schedule**

The Presentation Schedule included in your course materials gives you the important dates for the completion of tutor- marked assignments and for attending tutorials. Remember, you are required to submit all your assignments by the due date. You should guard against lagging behind in your work.

### **Assessment**

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

In tackling the assignments, you are expected to apply the information and knowledge you acquired during this course. The assignments must be submitted to your tutor for formal assessment in accordance with the deadlines stated in the Assignment File. The work you submit to your tutor for assessment will count for 30% of your total course mark.

At the end of the course, you will need to sit for a final three-hour examination. This will also count for 70% of your total course mark.

### **Tutor-Marked Assignment**

There are eight tutor- marked assignments in this course. You need to submit all the assignments. The total marks for the best four (4) assignments will be 30% of your total course mark.

Assignment questions for the units in this course are contained in the Assignment File. You should be able to complete your assignments from the information and materials contained in your set textbooks and study units. However, you may wish to use other references to broaden your viewpoint and provide a deeper understanding of the subject.

When you have completed each assignment, send it together with a form to your tutor. Make sure that each assignment reaches your tutor on or before the deadline given. If however you cannot complete your work on time, contact your tutor before the assignment is done to discuss the possibility of an extension.



## Final Examinations and Grading

The final examination for the course will carry 70% percentage of the total marks available for this course. The examination will cover every aspect of the course, so you are advised to revise all your corrected assignments before the examination.

This course endows you with the status of a teacher and that of a learner. This means that you teach yourself and that you learn, as your learning capabilities would allow. It also means that you are in a better position to determine and to ascertain the what, the how, and the when of your language learning. No teacher imposes any method of learning on you.

The course units are similarly designed with the introduction following the table of contents, then a set of objectives and then the discourse and so on.

The objectives guide you as you go through the units to ascertain your knowledge of the required terms and expressions.

## Course Marking Scheme

This table shows how the actual course marking is broken down.

Assessment	Marks
Assignment 1- 4	Four assignments, best three marks of the four count at 30% of course marks
Final Examination	70% of overall course marks
Total	100% of course marks

## Course Overview

Unit	Title of Work	Weeks Activity	Assessment (End of Unit)
	Course Guide	Week 1	
	<b>Module 1</b>		
1	Introduction and Data Representation	Week 1	Assignment 1
2	Digital Logic Circuit	Week 2 - 3	Assignment 2
3	Memory Organisation	Week 4 - 5	Assignment 3
4	Input/Output Organisation	Week 6 - 7	Assignment 4

	<b>Module 2</b>		
1	Instruction Sets	Week 8 -9	Assignment 5
2	Register Organisation And Micro-Operations	Week 10 – 11	Assignment 6
3	ALU And Control Unit Organisation	Week 12 – 13	Assignment 7
4	Microprogrammed Control Unit	Week 14 - 15	Assignment 8
	Revision	Week 16	
	Examination	Week 17	
Total		17 weeks	

### **How to Get the Best from this Course**

In distance learning the study units replace the university lecturer. This is one of the great advantages of distance learning; you can read and work through specially designed study materials at your own pace, and at a time and place that suit you best. Think of it as reading the lecture instead of listening to a lecturer. In the same way that a lecturer might set you some reading to do, the study units tell you when to read your set books or other material. Just as a lecturer might give you an in-class exercise, your study units provide exercises for you to do at appropriate points.

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 is a set of learning objectives. These objectives enable you know what you should be able to do by the time you have completed the unit. You should use these objectives to guide your study. When you have finished the units you must go back and check whether you have achieved the objectives. If you make a habit of doing this you will significantly improve your chances of passing the course.

Remember that your tutor's job is to assist you. When you need help, don't hesitate to call and ask your tutor to provide it.

1. Read this Course Guide thoroughly.
2. 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 assignments relate to the units. Whatever method you chose to use, you should decide on it and write in your own dates for working on each unit.

3. 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 lag behind in their course work.
4. Turn to Unit 1 and read the introduction and the objectives for the unit.
5. Assemble the study materials. Information about what you need for a unit is given in the overview at the beginning of each unit. You will almost always need both the study unit you are working on and one of your set of books on your desk at the same time.
6. 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 instructed to read sections from your set books or other articles. Use the unit to guide your reading.
7. Review the objectives for each study unit to confirm that you have achieved them. If you feel unsure about any of the objectives, review the study material or consult your tutor.
8. When you are confident that you have achieved a unit's objectives, you can then start on the next unit. Proceed unit by unit through the course and try to pace your study so that you keep yourself on schedule.
9. 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. When the assignment is returned, pay particular attention to your tutor's comments, both on the tutor-marked assignment form and on the assignment. Consult your tutor as soon as possible if you have any questions or problems.
10. 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 15 hours of tutorials provided in support of this course. You will be notified of the dates, times and location of these tutorials, 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 for you during the course. You must mail or submit your tutor-marked assignments to your tutor well before the due date (at least two working days are required). They will be marked by your tutor and returned to you as soon as possible.

Do not hesitate to contact your tutor by telephone, or e-mail if you need help. The following might be circumstances in which you would find help necessary. Contact your tutor if:

- you do not understand any part of the study units or the assigned readings,
- you have difficulty with the self-tests or exercises,
- you have a question or problem 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 to ask questions which are answered instantly. You can raise any problem encountered in the course of your study. To gain the maximum benefit from course tutorials, prepare a question list before attending them. You will learn a lot from participating in discussions actively.

## Summary

Introduction to Computer Organisation, as the title implies, introduces you to the fundamental concepts of how the computer system operates internally to perform the basic tasks required of it by the end-users. Therefore, you should acquire the basic knowledge of the internal workings of the components of the computer system in this course. The content of the course material was planned and written to ensure that you acquire the proper knowledge and skills in order to be able to programme the computer to do your bidding. The essence is to get you to acquire the necessary knowledge and competence and equip you with the necessary tools.

We wish you success with the course and hope that you will find it interesting and useful.

<b>MAIN COURSE</b>
------------------------

Course Code	CIT 246
Course Title	Introduction to Computer Organisation
Course Adapter	A. A. Afolorunso National Open University of Nigeria 14/16, Ahmadu Bello Way Victoria Island, Lagos
Course Co-ordinator	A. A. Afolorunso National Open University of Nigeria 14/16, Ahmadu Bello Way Victoria Island, Lagos

<b>Module 1</b>	.....	<b>1</b>
Unit 1	Introduction and Data Representation .....	1
Unit 2	Digital Logic Circuits .....	47
Unit 3	Memory Organisation .....	92
Unit 4	Input/Output Organisation .....	133
<b>Module 2</b>	.....	<b>154</b>
Unit 1	Instruction Sets .....	154
Unit 2	Register Organisation and Micro-Operations .....	193
Unit 3	ALU And Control Unit Organisation.....	224
Unit 4	Microprogrammed Control Unit .....	248

## **MODULE 1**

Unit 1	Introduction and Data Representation
Unit 2	Digital Logic Circuits
Unit 3	Memory Organisation
Unit 4	Input/Output Organisation

### **UNIT 1 INTRODUCTION AND DATA REPRESENTATION**

#### **CONTENTS**

1.0	Introduction
2.0	Objectives
3.0	Main Content
3.1	The Von Neumann Architecture
3.2	Computers: Then and Now
3.2.1	Mechanical Computers
3.2.2	First Generation Computers
3.2.3	Second Generation Computers
3.2.4	Third Generation Computers
3.2.5	Later Generation
3.3	Data Representation
3.3.1	Number Systems
3.3.2	Decimal Representation in Computers
3.3.3	Alphanumeric Representation
3.3.4	Computational Data Representation
3.3.5	The Fixed Point Representation
3.3.6	The Decimal Fixed Point Representation
3.3.7	The Floating Point Representation
3.3.8	Error Detection and Correction Codes
3.4	Instruction Execution
4.0	Conclusion
5.0	Summary
6.0	Tutor-Marked Assignment
7.0	References/Further Readings

#### **1.0 INTRODUCTION**

The computer which is one of the major components of an information technology network is gaining increasing popularity. Today, computer technology has permeated every sphere of existence of modern man. From railway reservations to medical diagnosis; from TV programs to satellite launching; from matchmaking to the arrest of criminals –

everywhere we witness the elegance, sophistication and efficiency possible only with the help of computers.

In this unit we will revisit one of the important computer system structures: The von Neumann Architecture. In addition we will examine data representation, error detection and correction and a simple model of instruction execution. This model will be enhanced in the second module of this course. You can obtain more details on these terms from the references and further readings.

## **2.0 OBJECTIVES**

At the end of the unit you will be able to:

- define the logical structure of the computer;
- describe data representation in computers;
- describe the use of integer representation of data;
- identify how error detection and correction codes are used;
- define the instruction cycle; and
- define the concept of interrupt.

## **3.0 MAIN CONTENT**

### **3.1 The Von Neumann Architecture**

The basic function performed by a computer is the execution of a program. Therefore, one of the key aspects in program execution is the execution of an instruction. But what is an instruction? Obviously an instruction is a form of control code, which supplies the information about an operation and the data on which the operation is to be performed. The control unit (CU) interprets each of these instructions and generates respective control signals.

The Arithmetic Logic Unit (ALU) in special storage areas called registers performs the arithmetic and logical operations. The size of the register is one of the important considerations in determining the processing capabilities of the CPU. Register size refers to the amount of information that can be held in a register at a time for processing. The larger the register size, the faster may be the speed of processing.

An input/output system also called I/O components allows data input and reporting of the results in proper format and form. For transfer of information a computer system internally needs the system interconnections. One such interconnection structure is BUS interconnection.

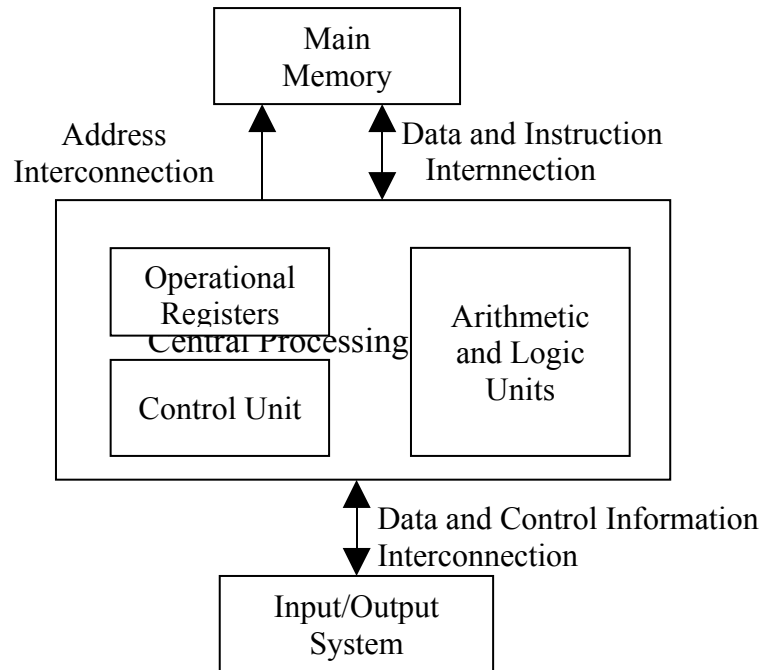


The main memory is needed in a computer to store instructions and data at the time of programme execution. It was pointed out by von-Neumann that the same memory can be used for storing data and instructions. In such cases the data can be treated as data on which processing can be performed, while instructions can be treated as data which can be used for the generation of control signals. The memory unit stores all the information in a group of memory cells, also called memory locations, as binary digits. Each memory location has a unique address and can be addressed independently. The contents of the desired memory locations are provided to the central processing unit by referring to the address of the memory location. Memory to CPU is another important data transfer path. The amount of information which can be transferred between CPU and memory depends on the size of the BUS connecting the two.

Let us summarise the key features of a von Neumann machine.

- The von Neumann machine uses a stored program concept, i.e., the program and data are stored in the same memory unit. The computers prior to this idea used to store programs and data on separate memories. Entering and modifying these programs were very difficult as they were entered manually by setting switches, plugging, and unplugging.
- Each location of the memory of the von Neumann machine can be addressed independently.
- Execution of instructions in the von Neumann machine is carried out in a sequential fashion (unless explicitly altered by the program itself) from one instruction to the next.

Figure 1 shows the basic structure of a von Neuman machine.



**Figure 1. The basic structure of a conventional von Neumann machine**

A von Neumann machine has only a single path between the main memory and the control unit (CU). This feature/constraint is referred to as the von Neumann bottleneck. Several other architectures have been suggested for modern computers. You can find about non-von Neumann architectures in the further readings.

### SELF-ASSESSMENT EXERCISE 1

State whether True or False:

1. A byte is equal to 8 bits and can represent a character internally.  
True  False
2. A word on pentium is equal to one byte. True  False
3. The von Neumann architecture specifies different memory units for data and instructions. The memory which stores data is called data memory and the memory which stores instructions is called instruction, independently. True  False
4. In von Neumann architecture each bit of memory can be accessed independently. True  False

5. A program is a sequence of instructions designed for achieving a task or goal. True  False
6. One MB is equal to 1024KB True  False

### 3.2 Computers: Then and Now

Let us discuss the history of computers because this will give the basic information about the technological development trends in the computer in the past and its projections in the future. If we want to know about computers completely then we must start from the history of computers and look into the details of various technological and intellectual breakthroughs. These are essential to give us the feel of how much work and effort has been done to get the computer in this shape.

The ancestors of the modern age computer were the mechanical and electromechanical devices. This ancestry can be traced to as far back as the 17<sup>th</sup> century, when the first machine capable of performing four mathematical operations, viz.: addition, subtraction, division and multiplication was invented. In the subsequent subsection we present a very brief account of mechanical computers.

#### 3.2.1 Mechanical Computers

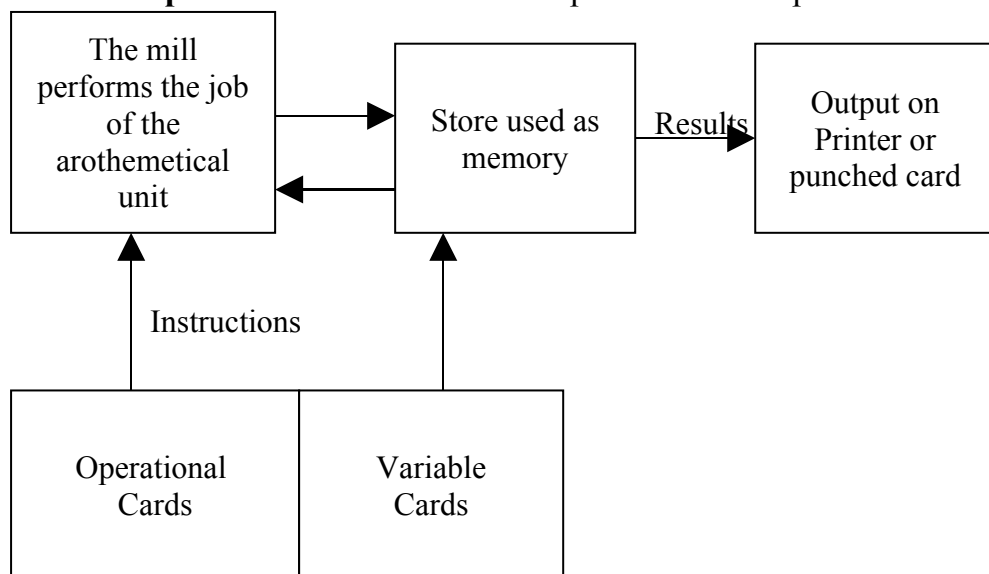
Blaise Pascal made the very first attempt towards the automatic computing. He invented a device, which consisted of lots of gears and chains and used to perform repeated addition and subtraction. This device was called pascaline. Later many attempts were made in this direction; we will not go into the details of these mechanical calculating devices. But we must examine some details about the innovation by Charles Babbage, the grandfather of the modern computer. He designed two computers:

**The Difference Engine:** It was based on the mathematical principle of finite differences and was used to solve calculations on large numbers using a formula. It was also used for polynomial and trigonometric functions.

**The Analytical Engine by Babbage:** It was a general purpose-computing device, which could be used for performing any mathematical operation automatically. It consisted of the following components:

- **The Store:** A mechanical memory unit consisting of sets of counter wheels

- **The Mill:** An arithmetical unit, which is capable of performing the four basic arithmetical operations.
- **Cards:** There are basically two types of cards:
  - *Operation Cards:* Selects one of four arithmetical operations by activating the mill to perform the selected function.
  - *Variable Cards:* Selects the memory location to be used by the mill for a particular operation (i.e., the source of the operands and the destination of the results).
- **Output:** It could be directed to a printer or a cardpunch device.



Cards make the program  
Each card contains an instruction

**Figure 2: Logical structure of Babbage's analytical engine**

The basic features of this analytical engine were as follows:

- It was a general-purpose programmable machine.
- It had the provision of an automatic sequence control, thus enabling programs to alter its sequence of operations.
- The provision of sign checking of result existed.
- A mechanism for advancing or reversing control cards was permitted, thus enabling the execution of any desired instruction. In other words, Babbage devised conditional and branching instructions. The Babbage machine is fundamentally the same as a modern computer. Unfortunately Babbage's work could not be completed. But as a tribute to Charles Babbage his analytical engine

was completed in the decade and is now on display at the Science Museum at London.

Next notable attempts towards the computer were electromechanical. Zuse used electromechanical relays that could be either opened or closed automatically. Thus, the use of binary digits, rather than decimal numbers started.

### **Harvard Mark I and the Bug**

The next significant effort towards devising an electromechanical computer was made at the Harvard University, jointly sponsored by IBM and the Department of UN Navy. Howard Aiken of Harvard University developed a system called Mark I in 1944. Mark I was a decimal machine.

You may have heard a term called “bug”. It is mainly used to indicate errors in computer programs. This term was coined when one day, a program in Mark I did not run properly due to a moth short-circuiting the computer. Since then, the moth or the bug has been linked with errors or problems in computer programming. Thus, the process of eliminating errors in a program is known as “debugging”.

The basic drawbacks of these mechanical and electromechanical computers were:

- The friction/inertia of moving components had limited the speed.
- The data movement using gears and liner was quite difficult and unreliable.
- The change was to have switching and storing mechanisms with no moving parts; and then the electronic switching technique “triode” vacuum tubes were used, hence the birth of the first electronic computer.

### **3.2.2 First Generation Computers**

It is indeed ironic that scientific inventions of great significance have often been linked with supporting a very sad and undesirable aspect of civilisation i.e. fighting wars. Nuclear energy would not have been developed as fast, if colossal efforts were not spent towards devising nuclear bombs.

Similarly, the origin of the first truly general-purpose computer was also designed to meet the requirements of World War II. The ENIAC (the Electronic Numerical Integrator and Calculator) was designed in 1945 at the University of Pennsylvania to calculate figures for thousands of

gunnery tables required by the US army for accuracy in artillery fire. The ENIAC ushered in the era of what is known as first generation computers. It could perform 5000 additions or 500 multiplications per minute. It was however, a monstrous installation. It used thousands of vacuum tubes (18000), weighed 30 tons, occupied a number of rooms, needed a great amount of electricity and emitted excessive heat. The main features of ENIAC can be summarised as:

- ENIAC was a general purpose-computing machine in which vacuum tube technology was used.
- ENIAC was based on decimal arithmetic rather than binary arithmetic.
- ENIAC needed to be programmed manually by setting switches and plugging or unplugging. Thus, to pass a set of instructions to the computer was cumbersome and time-consuming. This was considered to be the major deficiency of ENIAC.

The trends, which were encountered during the era of first generation computers, were as follows:

- Centralised control in a single CPU; all the operations required a direct intervention of the CPU.
- The use of ferrite-core main memory was started during this time.
- Concepts such as the use of virtual memory and index register (you will know more about these terms later) started.
- Punched cards were used as input devices.
- Magnetic tapes and magnetic drums were used as secondary memory.
- Binary code or machine language was used for programming.
- Towards the end, due to difficulties encountered in the use of machine language as programming language, the use of symbolic language that is now called assembly language started.
- Assembler, a program that translates assembly programs to machine was made.
- The computer was accessible to only one program at a time (single user environment).
- The advent of von Neumann architecture.

### **3.2.3 Second Generation Computers**

The second-generation computers started with the advent of transistors. A transistor is a two-state device made from silicon. It is cheaper, smaller and dissipates less heat than the vacuum tube but can be utilised in a similar way as that of vacuum tubes. Unlike vacuum tubes, a transistor does not require wires, metal glass capsule and vacuum;

therefore, it is called a solid-state device. The transistors were invented in 1947 and launched the electronic revolution in 1950.

The generation of computers is basically differentiated by a fundamental hardware technology. Each new generation of computers is characterised by greater speed, larger memory capacity and smaller size than the previous generation. Thus, second generation computers were more advanced in terms of arithmetic and logic units and the control unit than their counterparts of the first generation. By this time high-level languages were beginning to be used and the provisions for system software were starting.

One of the main computer series during this time was the IBM 700 series. Each successful number of this series showed increased performance and capacity and reduced cost. In these series two main concepts, I/O channels – an independent processor for input/output, and a multiplexor- a useful routing device, were used.

### **3.2.4 Third Generation Computers**

A single self-contained transistor is called a discrete component. In the 1960s, the electronic equipment were made from the discrete components such as transistors, capacitors, resistors and so on. These components were manufactured separately and were soldered on circuit boards, which then could be used for making computers of the electronic components. Since the computer could contain around 10,000 of these transistors, the entire mechanism was cumbersome. Then started the era of microelectronics (small electronics) with the invention of integrated circuits (ICs). The use of ICs in computer defined the third generation of computers.

Some examples of third generation computers are IBM system/360 family and DEC PDP/8 systems. Third generation computers mainly used SSI chips. The PDP/8 was a compact, cheap system from DEC. This computer established a concept of minicomputer. One of the key concepts which were brought forward during this time was the concept of the family of compatible computers. IBM mainly started this concept with its system/360 family.

A family of computers consists of several models. Each model is assigned a model number; for example, the IBM system/360 family has, Models 30, 40, 50, 65 and 75. As we go from a lower model number to higher model number in this family, the memory capacity, processing speed and cost increases. But, all these models are compatible in nature; that is, a program written on a lower model can be executed on a higher model without any change. Only the time of execution is reduced as we

go towards a higher model. The biggest advantage of this family system was the flexibility of the selection of models. For example, if you had limited budget and processing requirements you could possibly have started with a relatively moderate model such as Model 30. As your business flourished and your processing requirement increased, you could upgrade your computer with subsequent models such as 40, 50, 65 and 75 depending on your need. Here, you were not sacrificing investment on the already developed software as they could be used in these machines also.

Let us summarise the main characteristics of the family. These are:

- The instructions on a family are of similar type. Normally, the instructions set on a lower end member is a subset of higher end member, therefore, a program written on a lower end member can be executed on a higher end member, but a program written on a higher end member may or may not execute on a lower end member.
- The operating system used on family members is the same. In certain cases some features can be added in the operating system for the higher end members.
- The speed of execution of instruction increases from lower end family members to higher end members.
- The number of I/O ports or interfaces increases as we move to higher members.
- Memory size increases as we move towards higher members.
- The cost increases from lower to higher members.

But how was the family concept implemented? There were three main features of implementation. These are:

- Increased complexity of the arithmetic logic unit
- Increase in memory – CPU data paths
- Simultaneous access of data in higher ends members.

The major developments which took place in the third generation can be summarised as:

- IC circuits were starting to find their application in the computer hardware replacing the discrete transistor component circuits. This resulted in reduction in the cost and the physical size of the computer.



- Semiconductor (integrated circuit) memories were starting to augment ferrite core memory as main memory.
- The CPU design was made simple and the CPU was made more flexible using a technique called microprogramming (this will be discussed in Module 2).
- Certain new techniques were introduced to increase the effective speed of program execution. These techniques were pipelining and multiprocessing. These can be found in the further readings.
- The operating system of computers was incorporated with the efficient methods of sharing the facilities or resources such as processor and memory space, automatically.

### **3.2.5 Later Generations**

One of the major milestones in the IC technology was the very large scale integration (VLSI) where thousands of transistors could be integrated on a single chip. The main impact of VLSI was that it was possible to produce a complete CPU or main memory or other similar devices on a single IC chip. This implied that mass production of CPU; memory etc. could be done at a very low cost. The VLSI-based computer architecture is sometimes referred to as the fourth generation computer.

The fourth generation is also coupled with Parallel Computer Architectures. These computers had shared or distributed memory and specialised hardware units for floating point computation. In this era, multiprocessing operating systems, compilers and special languages and tools were developed for parallel processing and distributed computing. VAX 9000, CRAY X-MP, IBM/3090 F are some of the systems developed during this era.

The fifth generation computers are also available presently. These computers mainly emphasise Massively Parallel Processing. These computers use high-density packaging and optical technologies. Discussions on such technologies are beyond the scope of this course.

### **SELF-ASSESSMENT EXERCISE 2**

1. What is a general-purpose machine?
2. List the advantages of IC technology over discrete components.

### **3.3 Data Representation**

Having discussed the history of computers, we shall focus our attention on answering some questions such as:

How is information represented in a computer? It is in the form of a binary digit popularly called **bit**. But how are the arithmetical calculations performed through these bits? How then are words like ABCD stored in the computer? This section will try to highlight all these points. But before we try to answer these questions, let us first recapitulate the number system.

### 3.3.1 Number Systems

A number system of base (also called radix)  $r$  is a system which has  $r$  distinct symbols for  $r$  digits. A string of these symbolic digits represents a number. To determine the quantity that the number represents, we multiply the number by an integer power of  $r$  depending on the place in which it is located and then find the sum of weighted digits.

**Decimal Numbers:** The decimal number system has ten digits represented by 0,1,2,3,4,5,6,7,8 and 9. Any decimal number can be represented as a string of these digits and since there are ten decimal digits, therefore, the base or radix of this system is 10.

Thus, a string of number 234.5 can be represented in quantity as:

$$2 \times 10^2 + 3 \times 10^1 + 4 \times 10^0 + 5 \times 10^{-1}$$

**Binary Numbers:** In binary numbers we have two digits 0 and 1 and they can also be represented, as a string of these two-digits called bits. The base of the binary number system is 2.

For converting the value of binary numbers to the decimal equivalent we have to find its quantity, which is found by multiplying a digit by its place value. For example, binary number 101010 is equivalent to

$$\begin{aligned} & 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 \\ & = 1 \times 32 + 0 \times 16 + 1 \times 8 + 0 \times 4 + 1 \times 2 + 0 \times 1 \\ & = 32 + 8 + 2 \\ & = 42 \text{ in decimal.} \end{aligned}$$

**Octal Numbers:** An octal system has eight digit represented as 0, 1, 2, 3, 4, 5, 6, 7. For finding the equivalent decimal number of an octal number one has to find the quantity of the octal number, which is again calculated as:

Octal number             $(23.4)_8$

(Please note the subscript 8 indicates it is an octal number, similarly a subscript 2 will indicate binary, 10 will indicate decimal and H will

indicate a hexadecimal number; in case no subscript is specified then the number should be treated as a decimal number or else whatever number system is specified before it.)

Decimal equivalent

$$\begin{aligned}
 (23.4)_8 &= 2 \times 8^1 + 3 \times 8^0 + 4 \times 8^{-1} \\
 &= 2 \times 8 + 3 \times 1 + 4 \times 1/8 \\
 &= 16 + 3 + 0.5 \\
 &= (19.5)_{10}
 \end{aligned}$$

**Hexadecimal Numbers:** The hexadecimal system has 16 digits, which are represented as 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F. A number  $(F2)_H$  is equivalent to

$$\begin{aligned}
 &F \times 16^1 + 2 \times 16^0 \\
 &= (15 \times 16) + 2
 \end{aligned}$$

$$\begin{aligned}
 (\text{As } F \text{ is equivalent to } 15 \text{ for decimal}) \\
 &= 240 + 2 \\
 &= (242)_{10}
 \end{aligned}$$

**Conversion of Decimal Number to Binary Number:** For converting a decimal number to binary number, the integer and fractional part are handled separately. Let us explain it with the help of an example.

**Example 1:** Convert the decimal number 43.125 to binary.

**Solution:**

<b>Integer Part = 43</b>	<b>Fraction 0.125</b>
By dividing the quotient of integer part repeatedly by 2 and separating the remainder till we get 0 as the quotient	By multiplying the fraction repeatedly by 2 and separating the integer as you get it till you have all zeros in fraction

<u>Quotient</u>	<u>Remainder</u>	0.125
21	1	x2
	▲	
	↑	
	↓	
	▼	

10	1	0	0.250
5	0		<u>x2</u>
2	1	0	0.500
1	0		<u>x2</u>
0	1	1	1.000

The integer equivalent binary  
is  $(101011)_2$   
(Read the way arrow is)

The decimal equivalent to  
binary is .001  
(Read the way arrow is)

Thus the number  $(101011.001)_2$  is  $= (43.125)_{10}$

$$\begin{aligned}
 &1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 + 0 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-2} \\
 &= 32 + 0 + 8 + 0 + 2 + 1 + 0 + 0 + 1/8 \\
 &= (43.125)_{10}
 \end{aligned}$$

One easy direct method in decimal to binary conversion for integer part is to first write the place values as:

$$\begin{array}{ccccccc}
 2^6 & 2^5 & 2^4 & 2^3 & 2^2 & 2^1 & 2^0 \\
 64 & 32 & 16 & 8 & 4 & 2 & 1
 \end{array}$$

Now, take the integer part, e.g., 40, find the next lower or equal binary place value number that is 32 in this case. Place 1 at 32 and subtract 32 from 40, which is 8. Do the same for 8 till we reach 0.

These steps are shown as:

32	16	8	4	2	1	
1						$40 - 32 = 8$
1	0	1				$8 - 8 = 0$
1	0	1	0	0	0	is the number

Try it.

**Conversion of Binary to Octal and Hexadecimal:** The rules for these conversions are straightforward. For converting binary to octal the binary number is divided into groups of three, which are then combined by place value to generate equivalent octal. For example the number

1	101011	.001	01
001	101011	.001	010

(Please note the number is unchanged, we have added 0 to complete the grouping. Also note the style of grouping before and after the decimal.

We count three numbers from right to left while after the decimal, from left to right.)

$$\begin{aligned} \text{The } 011 &= 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 \\ &= 0 + 2 + 1 = 3 \\ 101 &= 1 \times 4 + 0 \times 2 + 1 \times 1 = 5 \\ 001 &= 0 \times 4 + 0 \times 2 + 1 \times 1 = 1 \\ 010 &= 0 \times 4 + 1 \times 2 + 0 \times 1 = 2 \end{aligned}$$

The number is  $(153.12)_8$

Grouping four binary digits and finding an equivalent hexadecimal digit for it can make the hexadecimal conversion. For example the same number will be equivalent to

$$\begin{aligned} &110 \quad 1011 \quad . \quad 0010 \quad 1 \\ = &0110 \quad 1011 \quad . \quad 0010 \quad 1000 \\ = &6 \quad 11 \quad . \quad 2 \quad 8 \\ = &6 \quad B \quad . \quad 2 \quad 8 \quad (\text{11 in hexadecimal is B}) \\ = &(6B.28)_H \end{aligned}$$

Conversely, we can conclude that a hexadecimal digit can be broken down into a string of binary having 4 places and an octal can be broken down into a string of binary having 3 places. Figure 3 gives the binary equivalent of octal and hexadecimal numbers.

Octal Number	Binary- coded Octal	Hexadecimal Number	Binary-coded Hexadecimal
0	000	0	0000
1	001	1	0001
2	010	2	0010
3	011	3	0011
4	100	4	0100
5	101	5	0101
6	110	6	0110
7	111	7	0111
		8	1000
		9	1001
		-Decimal-	
		A	1010
		B	1011
		C	1100
		D	1101
		E	1110
		F	1111

**Figure 3: Binary equivalent of octal and hexadecimal numbers**

### 3.3.2 Decimal Representation in Computers

The binary number system is most natural for the computer because of the two stable states of its components. But unfortunately, this is not a very natural system for us as we work with the decimal number system. Then how does the computer do the arithmetic? One of the solutions, which are followed in most computers, is to convert all input values to binary. Then the computer performs arithmetical operations and finally converts the results back to the decimal number so that we can interpret it easily. Is there any alternative to this scheme? Yes, there exists an alternative way of performing computation in decimal form but it requires that the decimal numbers be coded suitably before performing these computations. Normally, the decimal digits are coded in 6-8 bits as alphanumeric characters but for the purpose of arithmetical calculations the decimal digits are treated as four bit binary codes.

As we know 2 binary bits can represent  $2^2 = 4$  different combination, 3 bits can represent  $2^3 = 8$  combination and 4 bits can represent  $2^4 = 16$  combination. To represent decimal digits into binary form we require 10 combinations only, but we need to have a 4-digit code. One of the common representations is to use the first ten binary combinations to represent the ten decimal digits. These are popularly known as Binary Coded Decimals (BCD). Figure 4 shows the binary coded decimal numbers. Let us represent 43.125 in BCD. It is 0100 0011.0001.0010 0101. Compare it with the binary we have acquired in Example 1.

Decimal	Binary Coded Decimal	
0	0000	
1	0001	
2	0010	
3	0011	
4	0100	
5	0101	
6	0110	
7	0111	
8	1000	
9	1001	
10	0001	0000
11	0001	0001
12	0001	0010
13	0001	0011
....	.....	.....
20	0010	0000
....	.....	.....
30	0011	0000

**Figure 4: Binary Coded Decimals (BCD)**

### 3.3.3 Alphanumeric Representation

But what about alphabets and special characters like +, -, etc? How do we represent these in computer? A set containing letters of the alphabet (in both cases), the decimal digits (10 in number) and special characters (roughly 10-15 in number) consist of at least 70 – 80 elements. One such code generated for this set and is popularly used is ASCII (American National Standard Code for Information Interchange). This code uses 7 bits to represent 128 characters. Now, an extended ASCII is used having an 8-bit character representation code on microcomputers.

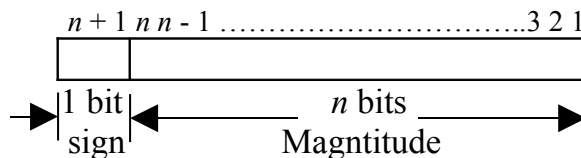
Similarly, binary codes can be formulated for any set of discrete elements, e.g. colours, the spectrum, the musical notes, chessboard positions etc. In addition these binary codes are also used to formulate instructions, which are advanced forms of data representation. We will discuss instructions in detail in the later modules.

### 3.3.4 Computational Data Representation

Till now we have discussed various number systems and BCD and alphanumeric representations but how are these codes actually used to represent data for scientific calculations? The computer is a discrete digital device and it stores information in flip-flops (see Unit 2 of this course for more details) which are two state devices, in binary form. Basic requirements of computational data representation in binary form are:

- Representation of sign;
- Representation of magnitude;
- If the number is fractional, then binary or decimal point, and exponent

The solution to sign representation is easy, because sign can be either positive or negative, therefore, one bit can be used to represent a sign. By default it should be the leftmost bit. Thus, a number of  $n$  bit can be represented as  $n+1$  bit number, where  $n+1^{\text{th}}$  bit is the sign bit and the rest  $n$  bits represent its magnitude (Please refer to Figure 5).



**Figure 5: A  $(n + 1)$  bit number**

The decimal position can be represented by a position between the flip-flops (storage cells in computer). But how can one determine this

decimal position? Well to simplify the representation aspect, two methods were suggested:

1. Fixed representation where the decimal position is assumed either at the beginning or at the end of a number; and
2. Floating point representation where a second register is used to keep the value of exponent that determines the position of the binary or decimal point in the number.

Before discussing these two representations let us first discuss the term “complement” of a number. These complements may be used to represent negative numbers in digital computers.

**Complement:** There are two types of complements for a number of base  $r$ ; these are called  $r$ 's complement and  $(r-1)$ 's complement. For example, for decimal numbers the base is 10, therefore, complements will be 10's complement and  $(10-1) = 9$ 's complements. For binary numbers we talk about 2's and 1's complements. But how are we to obtain complements and what do these complements mean? Let us discuss these issues with the help of following example:

**Example 2:** Find the 9's complement and 10's complement for the decimal number 256.

### Solution

**9's complement:** The 9's complement is obtained by subtracting each of the numbers from 9 (the highest digit value). Similarly, for obtaining 1's complement for a binary number we have to subtract each binary digit of the number from the digit 1 in the same manner as given in the example 3.

$$\begin{array}{r} 9\text{'s complement of } 256 \\ = \quad \quad \quad 9 \quad 9 \quad 9 \\ \quad \quad \quad \underline{- 2 \quad - 5 \quad - 6} \\ = \quad \quad \quad 7 \quad 4 \quad 3 \end{array}$$

**10's complement:** adding 1 in the 9's complement produces the 10's complement:

$$10\text{'s complement of } 256 = 743 + 1 = 744$$

Please note that on adding the number and its 9's complement we get 999 (for this three digit numbers) while on adding the number and its 10's complement we get 1000.

**Example 3:** Find 1's and 2's complement of 1010



**Solution**

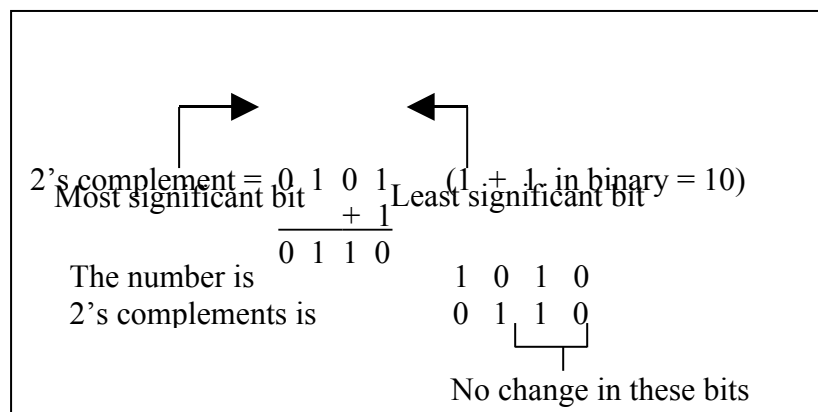
**1's complement:** The 1's complement of 1010 is

$$\begin{array}{r} 1 \quad 1 \quad 1 \quad 1 \\ - 1 \quad - 0 \quad - 1 \quad - 0 \\ \hline 0 \quad 1 \quad 0 \quad 1 \end{array}$$

The number is                    1    0    1    0  
The 1's complement is        0    1    0    1

Please note that wherever you have a digit 1 in the number, the complement contains 0 for that digit and vice versa. In other words to obtain 1's complement of a binary number, we only have to change all the 1's of the number to 0 and all the zeros to 1's. This can be done by complementing each bit.

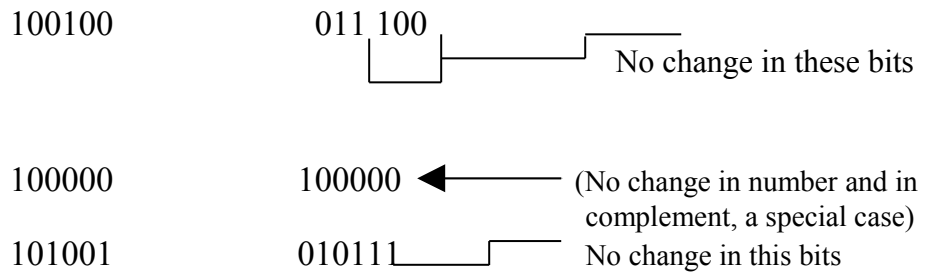
**2's complement:** adding 1 in 1's complement will generate the 2's complement.



The 2's complement can also be obtained by not complementing the least significant zeros till the first 1 is encountered. This 1 is also not complemented. After this 1, the rest of the bits on the left are complemented.

Therefore, 2's complement of the following number (using this method) should be (you can check it by finding 2's complement as we have done in the example):

Number	2's complement
--------	----------------



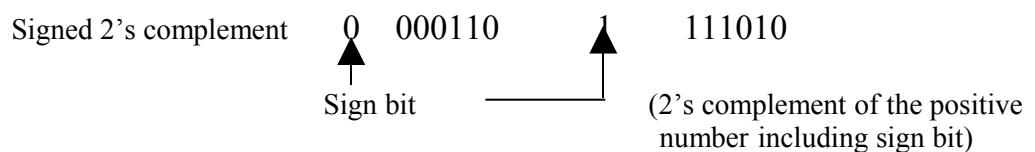
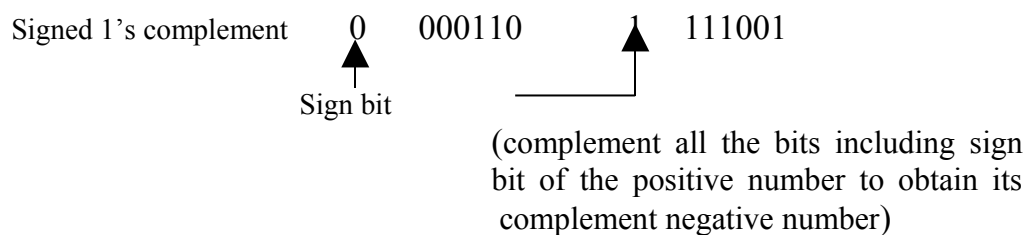
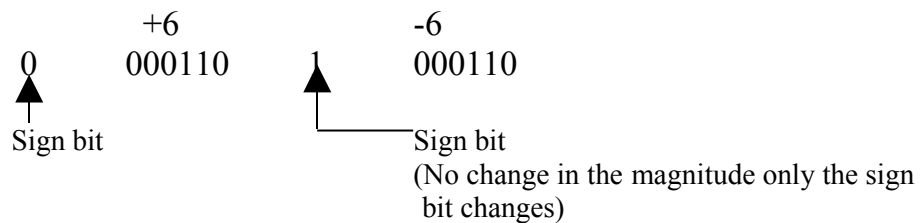
### 3.3.5 The Fixed Point Representation

The fixed-point numbers in binary use a sign bit. A positive number has a sign bit 0, while the negative number has a sign bit 1. In the fixed-point numbers we assume that the position of the binary point is at the end. It implies that all the represented numbers should be integers. A negative number can be represented in one of the following ways.

- Signed magnitude representation,
- Signed 1's complement representation, or
- Signed 2's complement representation.

(Assumption size of register = 7 bit, the 8<sup>th</sup> bit is used for error checking and correction or other purposes).

#### Signed magnitude representation



#### Arithmetical Additions

The complexity of arithmetical additions is dependent on the representation which has been followed. Let us discuss this with the help of the following example.

**Example:** Add 25 and – 30 in binary using a 7 bit register in  
signed magnitude representation  
signed 1's complement  
signed 2's complement

**Solution:** 25 or +25 is  
0 011001  
- 30 in signed magnitude representation is:  
+30 is 0011110,  
Therefore – 30 is 1 011110

To do the arithmetical addition with one negative number we have to check the magnitude of the numbers. The number which has the smaller magnitude is then subtracted from the bigger number and the sign of bigger number is selected. The implementation of such a scheme in digital hardware will require a long sequence of control decisions as well as circuits that will add, compare and subtract numbers. Is there a better alternative than this scheme? Let us first try the signed 2's complement.

-30 in signed 2's complement notation will be  
+30 is 0 011110  
-30 is 1 100010 (2's complement of 30 including a sign bit)  
+25 is 0 011001  
-25 is 1 100111

## Addition

$$\begin{array}{r}
 +25 \quad 0 \ 011 \ 001 \\
 +30 \quad \underline{0 \ 011 \ 110} \\
 +55 \quad \underline{0 \ 110 \ 111} \\
 \hline
 +25 \quad 0 \ 011 \ 001 \\
 -30 \quad \underline{1 \ 100 \ 110} \\
 -05 \quad \underline{1 \ 111 \ 011} \\
 \hline
 \end{array}
 \begin{array}{l}
 +5 \text{ is } 0 \ 000 \ 101 \\
 -5 \text{ is } 1 \ 111 \ 011 \\
 \text{in 2's complement representation} \\
 \text{(Just add the two numbers)}
 \end{array}$$

$$\begin{array}{r}
 -25 \quad 1 \ 100 \ 001 \\
 +30 \quad \underline{0 \ 011 \ 110} \\
 +05 \quad \underline{0 \ 000 \ 101} \\
 \hline
 \end{array}
 \begin{array}{l}
 \text{(Just add the two numbers)}
 \end{array}$$



Discard the carry out of the sign bit.

$$\begin{array}{r}
 -25 \quad 1 \ 100 \ 111 \\
 -30 \quad \underline{1 \ 100 \ 010} \\
 -55 \quad \underline{1 \ 001 \ 001} \\
 \hline
 \end{array}
 \begin{array}{l}
 +55 \text{ is } 0 \ 110 \ 111 \text{ in} \\
 \text{2's complement representation.} \\
 \text{Therefore in the 2's complement notation } -55 \text{ is} \\
 = 1 \ 001 \ 001
 \end{array}$$



Discard the carry bit.

Please note how easy it is to add two numbers using signed 2's complement. This procedure requires only one control decision and only one circuit for adding the two numbers. But it puts the additional condition that the negative numbers should be stored in signed 2's complement form in the registers. This can be achieved by complementing the positive number bit by bit and then incrementing the resultant by 1 to get signed 2's complement.

### Signed 1's Complement Representation

Another possibility, which is also simple, is the use of signed 1's complement. Signed 1's complement has a rule. Add the two numbers, including the sign bit. If **carry** of the most significant bit or sign bit is one, then increment the result by 1 and discard the carry over. Let us repeat all the operations with 1's complement.

$$\begin{array}{r}
 +25 \quad 0 \ 011 \ 001 \\
 +30 \quad 0 \ 011 \ 110 \\
 \hline
 +25 \quad 0 \ 011 \ 001 \\
 +30 \quad \underline{0 \ 011 \ 110} \\
 +55 \quad \underline{0 \ 110 \ 111} \\
 \hline
 \end{array}
 \begin{array}{r}
 (-25) \quad 1 \ 100 \ 110 \\
 -30 \quad 1 \ 100 \ 001 \\
 \hline
 (-25) \quad 1 \ 100 \ 110 \\
 +30 \quad \underline{0 \ 011 \ 110} \\
 +5 \quad \underline{1 \ 0 \ 000 \ 100} \\
 \hline
 \end{array}$$



Carry out is 1 so add 1 to the sum, and the carry over.

Thus sum = 0 000 101 which is number 5.

$$\begin{array}{r}
 +25 \quad 0 \ 011 \ 001 \\
 -30 \quad \underline{1 \ 100 \ 001} \\
 \hline
 \end{array}$$

$$\begin{array}{r}
 \underline{-5} \quad \underline{1 \quad 111 \quad 010} \\
 +5 \text{ is} \qquad \qquad 0 \ 000 \ 101 \\
 -5 \text{ in 1's complement} \quad 1 \ 111 \ 010 \\
 \\ 
 -25 \qquad \qquad \qquad 1 \ 100 \ 110 \\
 \underline{-30} \qquad \qquad \qquad 1 \ 100 \ 001 \\
 \underline{-55} \qquad \qquad \qquad \underline{1 \ 1 \ 000 \ 111} \\
 \qquad \qquad \qquad \uparrow \\
 \qquad \qquad \text{Carry out}
 \end{array}$$

Since, the carry out is 1, so add 1 to sum and discard the carry out.

$$\begin{array}{r}
 1 \ 000 \ 111 \\
 \underline{\qquad \qquad 1} \\
 1 \ 001 \ 000
 \end{array}$$

$$\begin{array}{r}
 +55 \text{ is} \qquad \qquad 0 \ 110 \ 111 \\
 -55 \text{ is 1's complement} \quad 1 \ 001 \ 000
 \end{array}$$

Another interesting feature about these representations is the representation of 0. In the signed magnitude and 1's complement there are two representations for zero as:

Signed magnitude	+ 0	- 0
	0 000000	1 000000
Signed 1's complement	0 000000	1 111111

But in signed 2's complement there is just one zero and there is no positive or negative zero.

$$\begin{array}{r}
 +0 \ 000000 \qquad -0 \ 2's \ complement \ of \\
 +0 = 1 \ 111111 \\
 \qquad \qquad \qquad \underline{\qquad \qquad 1} \\
 \qquad \qquad \qquad 1 \ 0 \ 000000 \\
 \qquad \qquad \qquad \blacktriangle \text{---} \text{ discard the carry}
 \end{array}$$

Thus, both +0 and -0 are the same in 2's complement notation. This is an added advantage in favour of 2's complement notation. The highest numbers, which can be accommodated in a register, also depend on the type of representation. In general, in an 8-bit register, 1 bit is used as sign. Therefore, the remaining 7 bits can be used for representing the value. The highest and the lowest number, which can be represented, are:

$$\begin{array}{r}
 \text{For a signed magnitude representation} \\
 = \qquad \qquad \qquad 2^7 - 1 \text{ to } -(2^7 - 1) \\
 \qquad \qquad \qquad = \qquad \qquad \qquad 128 - 1 \text{ to } -(128 - 1)
 \end{array}$$

$$= 127 \text{ to } -127$$

For signed 1's complement 127 to -127

But, for signed 2's complement we can represent +127 to -128. The -128 is represented in signed 2's complement notation as 10000000.

**Arithmetic Subtraction:** The subtraction can be easily done using the 2's complement by taking the 2's complement of the subtrahend (inclusive of sign bit) and then adding the two numbers.

Signed 2's complement provides a very simple way for adding and subtracting two numbers, thus, many computers (including IBM PC) adopt signed 2's complement notation. The reason why signed 2's complement is preferred over signed 1's complement is because it has only one representation for zero.

**Overflow:** An overflow is said to have occurred when the sum of two  $n$  digits number occupies  $n+1$  digit. This definition is valid for both binary as well as decimal digits. But what is the significance of an overflow for binary numbers since it is not a problem for the cases when we add two numbers? Well the answer is in the limits of representation of numbers. Every computer employs a limit for representing numbers. For instance, in our examples we are using 8-bit registers for calculating the sum. But what will happen if the sum of the two numbers can be accommodated in 9 bits? Where are we going to store the 9<sup>th</sup> bit? The problem will be cleared by the following example.

**Example:** Add the numbers 65 and 75 in 8-bit register in signed 2's complement notation.

$$\begin{array}{r} 65 \quad 0 \quad 1000001 \\ 75 \quad 0 \quad 1001011 \\ \hline 140 \quad 1 \quad 0001100 \end{array}$$

The expected result is +140 but the binary sum is a negative number and is equal to -116, which obviously is a wrong result. This has occurred because of an overflow.

How does the computer know that an overflow has occurred?

*If the carry into the sign bit is not equal to the carry out of the sign bit then an overflow must have occurred.*

For example,

-65	1	0111111		-65	1	0111111
-15	1	1110001	+	-75	1	0110101
-80	1	0110000	=	-140	1	0110100

▲ Carry into sign bit = 1  
 └ Carry out of sign bit = 1  
 No overflow

▼ Carry into sign bit = 0  
 Carry out of sign bit = 1  
 Therefore, overflow

Thus, an overflow has occurred; i.e., the arithmetical results so calculated have exceeded the capacity of the representation. This overflow also implies that the calculated results might be erroneous.

### 3.3.6 The Decimal Fixed Point Representation

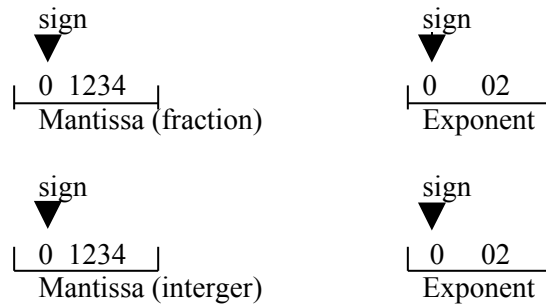
A decimal digit is represented as a combination of four bits; thus, a four digit decimal number will require 16 bits for decimal digits representation and an additional 1 bit for a sign. Normally to keep the conversion of one decimal digit to 4 bits, the sign sometimes is also assigned a 4-bit code. This code can be the bit combination which has not been used to represent decimal digits; e.g., 1100 may represent plus and 1101 can represent minus.

Although this scheme wastes considerable amount of storage space, it does not require the conversion of a decimal number to binary. Thus, it can be used at places where the amount of computer arithmetic is less than the amount of input/output of data; e.g., calculators or business data processing. The arithmetic in decimal can also be performed as in binary except that instead of signed 1's complement, signed 9's complement is used and instead of signed 2's complement signed 10's complement is used.

### 3.3.7 The Floating Point Representation

The floating-point number representation consists of two parts. The first part of the number is a signed fixed-point number, which is termed **mantissa**, and the second part specifies the decimal or binary point position and is termed an **exponent**. The mantissa can be an integer or a fraction. Please note that the position of the decimal or binary point is assumed and it is not a physical point; therefore, wherever we are representing a point it is only the assumed position.

**Example 1:** A decimal +12.34 in a typical floating point notation is:

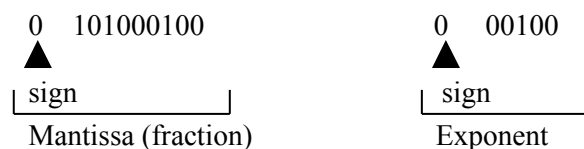


This number in any of the above form (if represented in BCD) requires 17 bits for mantissa (1 for sign and 4 for each decimal digit as BCD) and 9 bits for exponent (1 for sign and 4 for each decimal digit as BCD). Please note that the exponent indicates the correct decimal location. In the first case (where exponent is) +2 indicates that the actual position of the decimal point is two places to the right of the assumed position, while exponent – 2 indicates that the assumed position of the point is two places towards the left of assumed position. The assumption of the position of point is normally the same in a computer resulting in a consistent computational environment.

Floating-point numbers are often represented in normalised forms. A floating point number whose mantissa does not contain zero as the most significant digit of the number is considered to be in normalised form. For example, a BCD mantissa +370 which is 0 0011 0111 0000 is in normalised form because these leading zero's are not part of a zero digit. On the other hand a binary number 001100 is not in the normalised form. The normalised form of this number is:

0            1100  
(sign)

A floating binary number +1010.001 in a 16-bit register can be represented in normalised form (assuming 10 bits for mantissa and 6 bits for exponent).



A zero cannot be normalised as all the digits in mantissa in this case have to be zero.

Arithmetical operations involved with floating point numbers are more complex in nature, take longer time for execution and require complex hardware. Yet the floating-point representation is a must as it is useful in scientific calculations. Real numbers are normally represented as



floating point numbers. Floating point numbers are explained in detail in the next module of this course.

### 3.3.8 Error Detection and Correction Codes

Before we wind up the data representation in the context of today's computer, we must discuss a code which helps in the recognition and correction of errors. The computer is an electronic medium therefore, there is a possibility of errors being introduced during the transmission of data. These errors may result from some disturbance in the transmission media or from external disturbances. Most of these errors result in the change of a bit from 0 to 1 or 1 to 0. One of the simplest error detection codes that is used commonly is called parity bit.

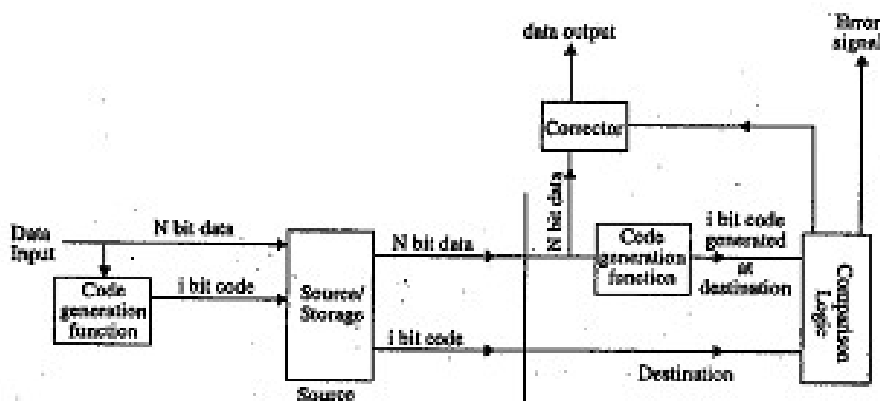
**The Parity Bit:** A parity bit is an extra bit added with binary data such that it makes the total number of 1's in the data either odd or even. For example, in a 7-bit data 0110101 let us add an 8<sup>th</sup> bit, which is a parity bit. If the added parity bit is an even parity bit then the value of this parity bit should be zero as already four 1's are there in the 7-bit number. If we are adding an odd parity bit then it will be 1, since we already have four 1 bits in the number and on adding an 8<sup>th</sup> bit (which is a parity bit) as 1 we are making the total number of 1's in the number (which now includes parity bit also) as 5, an odd number.

Similarly in 0010101

Parity bit for even parity is 1

Parity bit for odd parity is 0

But how does the parity bit detect an error? We will discuss this issue in general as an error detection and correction system (refer to Figure 6).



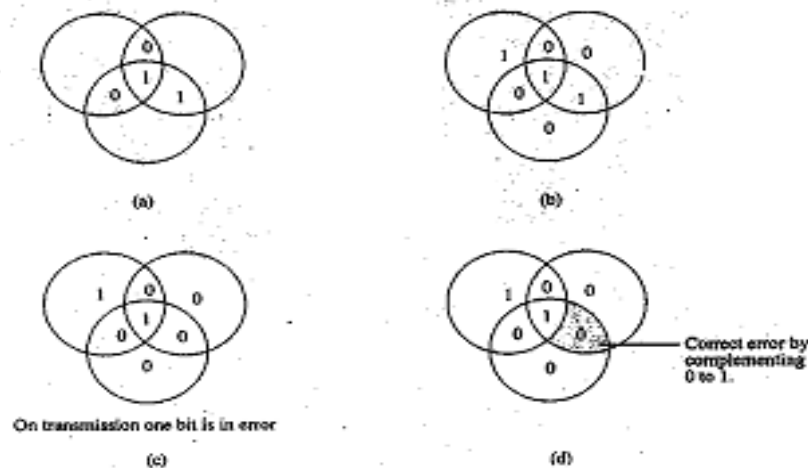
**Figure 6: Error detection and correction**

For transferring or storing N bit data with error detection and/or correction mechanism, an error detection code is generated using a function (for example, even parity bit generation) at the source or storage end. The data as well as the error detector code is passed on to

the data destination where once again the error detection code is generated using the same function on the data. The comparison of the generated and incoming error detection code determines whether the error has occurred or not. In case an error has occurred, an error signal is generated and a correction mechanism (in case the error code is an error detection and correction code, this step is not valid if parity bits are used), is applied on the data. In case, no error is detected then the data needs no correction and is sent as it is. Sometimes, it is not possible to correct the errors. Such conditions are reported.

The parity bit is only an error detection code. The concept of the parity bit has been developed and error detection and correction codes have been developed using more than one parity bit. One such code is the Hamming error correcting code.

**The Hamming Error Correcting Code:** Richard Hamming at Bell Laboratories devised this code. We will discuss this code with the help of Venn diagrams. For simplicity we will confine ourselves only to 4-bit data. Figure 7 (a) shows the Venn diagrams with filled in data bits, which are filled in the intersecting inner compartments.



**Figure 7: The Hamming error correction code**

The next step is to fill in the parity bits for these four data bits. The principle here is that we add the parity bits such that the number of 1's in a circle is even (even parity). Figure 7(b) is filled up using this rule. Please note that each circle has an even number of 1's.

After the data transfer, let us say, we encounter a situation where one of the data bit is changed from 1 to 0. Thus, an error has occurred. This condition is shown in Figure 7 (c). Please note that except for the data bit no other bit has changed. How will this error be detected and rectified? Figure 7(d) shows the detection and rectification. The parity bit of the two circles is indicating an error of one bit. Since two circles

are indicating errors, then the error is at the intersection of these two circles. So, we have not only recognised the error but also its source. Thus, in this case by changing the bit in error from 0 to 1 we will rectify the error.

Now let us discuss a scheme for error correction and detection of **single bit** errors in the context of figure 7, for 8-bit words. The first question in this respect is: what should be the length of the code? Before answering this question we have to look into the comparison logic of error detection. Error detection is done by comparing the two 'i' bit error detection and correction codes fed to the comparison logic bit by bit (refer to figure 6). Let us have the comparison logic which produces a zero if the compared bits are the same, or else it produces a one.

Therefore, if similar position bits are the same then we get zero at that bit position, but if they are different, i.e. this bit position may point to some error, then this particular bit position will be marked as one. This way a match word called syndrome word is constructed. This syndrome word is "i" bit long; therefore, it can represent  $2^i$  values or combinations. For example, a 4-bit syndrome word can represent  $2^4 = 16$  values, which range from 0 to 15 as:

0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111  
1000, 1001, 1010, 1011, 1100, 1101, 1110, 1111

The value 0000 or 0 represent no error while the other values i.e.  $2^i - 1$  (for 4 bits  $2^4 - 1 = 15$ , that is from 1 to 15) represent an error condition. Each of these  $2^i - 1$  (or 15 for 4 bits) values can be used to represent an error of a particular bit. Since the error can occur during the transmission of "N" bit data plus "i" bit error correction code, we need to have at least "N+i" error values to represent them. Therefore, the number of error correction bits should be found from the following equation:

$$2^i - 1 \geq N + i$$

If we are assuming an 8-bit word then we need to have

$$2^i - 1 \geq 8 + i$$

Say at  $i = 3$  LHS =  $2^3 - 1 = 7$ ; RHS =  $8 + 3 = 11$   
 $i = 4$  LHS =  $2^4 - 1 = 15$ ; RHS =  $8 + 4 = 12$

Therefore, for an 8-bit word we need to have at least a 4 error correction code.

Similarly for a 16-bit word we need to have  $i = 5$

$$2^5 - 1 = 31 \text{ and } 16 + i = 16 + 5 = 21$$

For a 16-bit word we need to have five error correcting bits.

The next step in this connection will be to arrange the 8-bit word and its 4-bit error correction code in such a way that a particular value of the syndrome word specifies an error in a unique bit (which may be data or an error detection code). The following arrangement of the (N+i) bits is suggested.

Bit positions	12	11	10	9	8	7	6	5	4	3	2	1
Data bits	8	7	6	5	4	3	2	1				
Correction bits					8			4	2	1		

**Figure 8: Data bits and bit positions**

The above arrangement is derived on the basis that:

- The syndrome word zero implies no error.
- If the syndrome word contains only one bit as one then it should be inferred that an error has occurred only in the parity bits, therefore, no correction is needed in the data. But how can we implement it? This can be implemented easily by assigning the check bits as 1<sup>st</sup>, 2<sup>nd</sup>, 4<sup>th</sup>, and 8<sup>th</sup> bit positions.
- In case more than one bit in the syndrome word is set as 1 then the numerical value of the syndrome word should determine the bit position which is in error.
- The arrangement shown in figure 8 has an added advantage, i.e. each bit position can be calculated as a fraction of correction bit positions. Please note that in case any one of the correction bits has changed during data transmission, that implies that any one of the 1<sup>st</sup> and 2<sup>nd</sup> or 4<sup>th</sup> or 8<sup>th</sup> bit position data have been altered. Therefore, the syndrome bit will be 0001 if the data at first bit position has changed, 0010 if 2<sup>nd</sup> bit position has changed; or 0100 if data at 4<sup>th</sup> bit position has changed, or 1000 if data at 8<sup>th</sup> bit position has changed. Thus, the proposed bit arrangement scheme of figure 8 satisfies the second assumption for the bit arrangement scheme. The next assumption in this regard is the value of syndrome word should indicate the bit position which is in error, that is, if there is an error in bit position 3 it should change correction bits of bit positions 1 and 2 and so on. Let us discuss how this can be achieved.

Let us make the table for the proposed scheme.

**Table 1**

Bit position in error	Effectuated bit position of syndrome word				Comments
1	-	-	-	1	One bit position in error
2	-	-	2	-	One bit position in error
3	-	-	2	1	$1 + 2 = 3$
4	-	4	-	-	One bit position in error
5	-	4	-	1	$4 + 1 = 5$
6	-	4	2	-	$4 + 2 = 6$
7	-	4	2	1	$4 + 2 + 1 = 7$
8	8	-	-	-	One bit position in error
9	8	-	-	1	$8 + 1 = 9$
10	8	-	2	-	$8 + 2 = 10$
11	8	-	2	1	$8 + 2 + 1 = 11$
12	8	4	-	-	$8 + 4 = 12$

We are assuming that only one bit can be in error. The table given indicates that the first bit in the syndrome word should be one if any one of the 1<sup>st</sup>, 3<sup>rd</sup>, 5<sup>th</sup>, 7<sup>th</sup>, 9<sup>th</sup>, or 11<sup>th</sup> bit position is in error. The second bit of the syndrome word should be one if any one of the 2<sup>nd</sup>, 3<sup>rd</sup>, 6<sup>th</sup>, 7<sup>th</sup>, 10<sup>th</sup>, or 11<sup>th</sup> bit position is in error, the third bit of the syndrome word should be 1 if any of the 4<sup>th</sup>, 5<sup>th</sup>, 6<sup>th</sup>, 7<sup>th</sup>, or 12<sup>th</sup> bit position is in error and the fourth bit of the syndrome word should be 1 if any of the 9<sup>th</sup>, 10<sup>th</sup>, 11<sup>th</sup>, or 12<sup>th</sup>, bit position is in error.

Since the 1<sup>st</sup>, 2<sup>nd</sup>, 4<sup>th</sup>, 8<sup>th</sup> bit position are occupied by correction bits or check bits at the source, therefore, they should not be used for calculating check bits,

Based on the above facts, the check bits should be calculated as:

Check bit 1 = Even parity of (3<sup>rd</sup>, 5<sup>th</sup>, 7<sup>th</sup>, 9<sup>th</sup>, 11<sup>th</sup> bit position)

Check bit 2 = Even parity of (3<sup>rd</sup>, 6<sup>th</sup>, 7<sup>th</sup>, 10<sup>th</sup>, 11<sup>th</sup> bit position)

Check bit 3 = Even parity of (5<sup>th</sup>, 6<sup>th</sup>, 7<sup>th</sup>, 12<sup>th</sup> bit position)

Check bit 4 = Even parity of (9<sup>th</sup>, 10<sup>th</sup>, 11<sup>th</sup>, 12<sup>th</sup> bit position)

or in other words (refer figure 8):

Check bit 1 = Even parity of (Data bits 1, 2, 4, 5, 7)

Check bit 2 = Even parity of (Data bits 1, 3, 4, 6, 7)

Check bit 3 = Even parity of (Data bits 2, 3, 4, 8)

Check bit 4 = Even parity of (Data bits 5, 6, 7, 8)

This is called a single error correcting (SEC) code. Let us see an example of error correction using this code.

**Example:** An 8-bit input word 01011011 on transmission is received as 01001011 (that is error in 5<sup>th</sup> data bit). How can the SEC code, if used, rectify this error?

**Solution:** The SEC code for an 8-bit word is of 4 bits

Check bit 1 = Even parity of (1,1,1,1,1) = 1


Check bit 2 = Even parity of (1,0,1,0,1) = 1

Check bit 3 = Even parity of (1,0,1,0) = 0

Check bit 4 = Even parity of (1,0,1,0) = 0

Therefore, the 12-bit word to be transmitted is:

Bit positions	12	11	10	9	8	7	6	5	4	3	2	1
Data bits	8	7	6	5		4	3	2		1		
Check bits						4			3		2	1
Data to be transmitted	0	1	0	1	0	1	0	1	0	1	1	1
The data is received as	0	1	0	0	0	1	0	1	0	1	1	1

  
 (Error in 5<sup>th</sup> data bit)

Calculation of check bits of data received:

Check bit 1 = Even parity of (1, 1, 1, 0, 1) = 0

Check bit 2 = Even parity of (1, 0, 1, 0, 1) = 1

Check bit 3 = Even parity of (1, 0, 1, 0) = 0

Check bit 4 = Even parity of (0, 0, 1, 0) = 1

Syndrome word = compare the received check bits to calculated check bits of received data

$$\begin{array}{r}
 = 0\ 0\ 1\ 1 \text{ received check bits} \\
 \quad \underline{1\ 0\ 1\ 0} \text{ calculated check bits} \\
 \quad 1\ 0\ 0\ 1 \text{ syndrome word}
 \end{array}$$

Please note that for syndrome word calculation if two check bits are the same then the respective bit in the syndrome word will be 0. If the two check bits are different then the bit in the syndrome word will be 1.

Thus, syndrome word = 1001, which implies that the 9<sup>th</sup> bit position in the received 12-bit information is in error. The 9<sup>th</sup> bit position corresponds to the fifth data bit. Change this bit to 1 if it is 0 or to 0 if it is 1. Since in the received data it is 0, therefore, change it to 1.

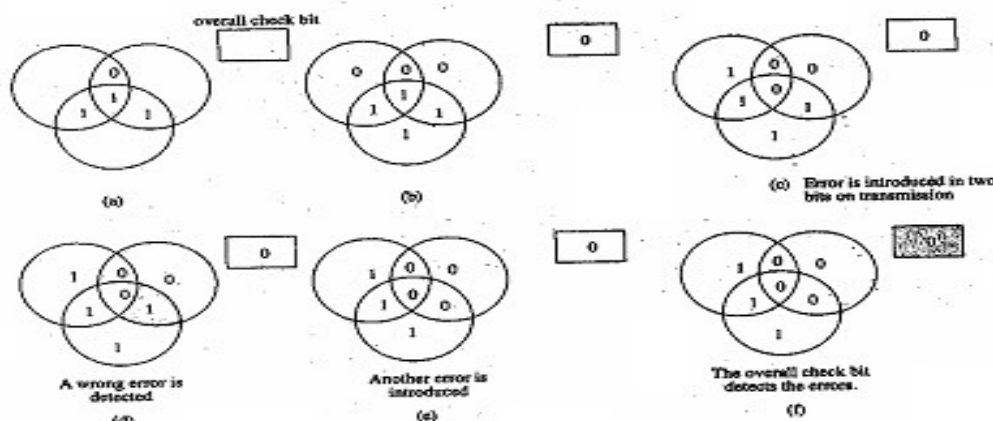
Hence, the data was (excluding check bits) received as 01001011

The corrected data is 01011011

The corrected data is the same as the transmitted data.

Normally, the SEC code may be used in semiconductor memories for correction of single bit errors. However, it is supplemented with an added feature for the detection of errors in two bits. This is called a SEC-DED (Single Error Correction – Double Error Detecting) code. This code requires an additional check bit in comparison with the SEC code. We will only illustrate the working principle of the SEC-DED code with the help of a Venn diagram for a 4-bit data word.

Basically, the SEC-DED code guards against the errors of two bits in SEC codes. Figure 9 shows the principle of the SEC-DED code. Figure 9(a) shows a 4-bit data placed in the Venn diagram. In Figure 9(b) the necessary check bits have been added. Please note a square on the top; this is an additional overall even parity bit. Figure 9(c) shows a situation of data access or transmission where two-data bits (a check bit and a data bit) get changed. Now the SEC code tries to correct the data but it results in another error (refer to figure 9(d) and 9(e)). But the error is caught on checking the extra parity bit, hence the double error detection (refer to figure 9(f)).



**Figure 9: SEC-DED code**

Thus, the error is detected a second time but it cannot be corrected; only an error signal can be issued informing that the data read contains errors.

### 3.4 Instruction Execution

Having discussed the data representation in detail, let us come back to the discussion on the instructions.

As discussed earlier, the basic function performed by a computer is the execution of a program. The program which is to be executed is a set of instructions that is stored in the memory. The central processing unit (CPU) executes the instructions of the program to complete a task.

The major responsibility of the instruction execution is with the CPU. The instruction execution takes place in the CPU registers. Let us, first discuss a few typical registers, some of which are commonly available in some of the machines.

These registers are:

**Memory Address Register (MAR):** It specifies the address of the memory location from which the data or instruction is to be accessed (for a read operation) or to which the data is to be stored (for a written operation).

**Memory Buffer Register (MBR):** It is a register which contains the data to be written in the memory (for a written operation) or it receives the data from the memory (for read operation).

**Program Counter (PC):** It keeps track of the instruction that is to be executed next, after the execution of an on-going instruction.

**Instruction Register (IR):** Here the instructions are loaded before their execution.

The simplest model of instruction processing can be a two-step process. The CPU reads (fetches) instructions (coded) from the memory one at a time, and executes or performs the operation specified by this instruction. The instruction fetch is carried out for all the instructions. An instruction fetch involves the reading of an instruction from the memory location(s) to the CPU. The execution of this instruction may involve several operations, depending on the nature of the instruction.

The processing needed for a single instruction (fetch and execution) is referred to as an instruction cycle. The instruction cycle consists of the fetch cycle and the execution cycle. Program execution terminates if the electric power supply is discontinued or some sort of unrecoverable error occurs, or by a program itself.



A program counter is used for a fetch cycle in a typical CPU. The program counter keeps track of the instruction that is to be fetched next. Normally the next instruction in sequence is fetched next as programs are executed in sequence.

The fetched instruction is in the form of a binary code and is loaded into an instruction register (IR) in the CPU. The CPU interprets the instruction and takes the required action. In general, these actions can be divided into the following categories:

**Data Transfer:** From the CPU to memory or from memory to CPU, or from CPU to I/O, or from I/O to CPU.

**Data Processing:** A logic, arithmetical or shift operation may be performed by the CPU on the data.

**Transfer of Control:** This action may require an alteration of the sequence of execution. For example, an instruction from location 100 on execution may specify that the next instruction should be fetched from location 200. On the execution of such an instruction, the program counter, which was having a location value of 101 (the next instruction to be fetched in a case in which the memory word is equal to the register size) will be modified to contain a location value of 200. The value 101 may be saved in case the transfer of control instruction is a subroutine or function call.

The execution of a program may involve any combination of these actions. Let us take an example of an instruction execution.

**Example:** Let us assume a hypothetical machine which has a 16-bit instruction and data. Each instruction of the machine consists of two components: the operation code and the address of the operand in memory.

The operation code is assumed to be of four bits; therefore, the remaining 12 bits are for the address of the operand. The memory is divided into words of 16 bits. The CPU of this machine contains an additional register called accumulator (AC) in addition to the registers given earlier. The AC stores the data temporarily for computation purposes. Figure 10 shows the instruction and data formats for this machine.



**Figure 10: Instruction and data format for an assumed machine**

The machine can have  $2^4 = 16$  possible operation codes. For example, let us assume operation codes as:

0001 as “Load the accumulator with the content of the memory”  
 0010 as “Store the current value of the accumulator in the memory”  
 0011 as “Add the value from memory to the Accumulator”

This machine can address  $2^{12} = 4096$  memory words directly.

Please note that PC can be of 12 bits.

Now let us use the hexadecimal notation to show how the execution is performed.

Let us assume that three consecutive instructions to be executed are:

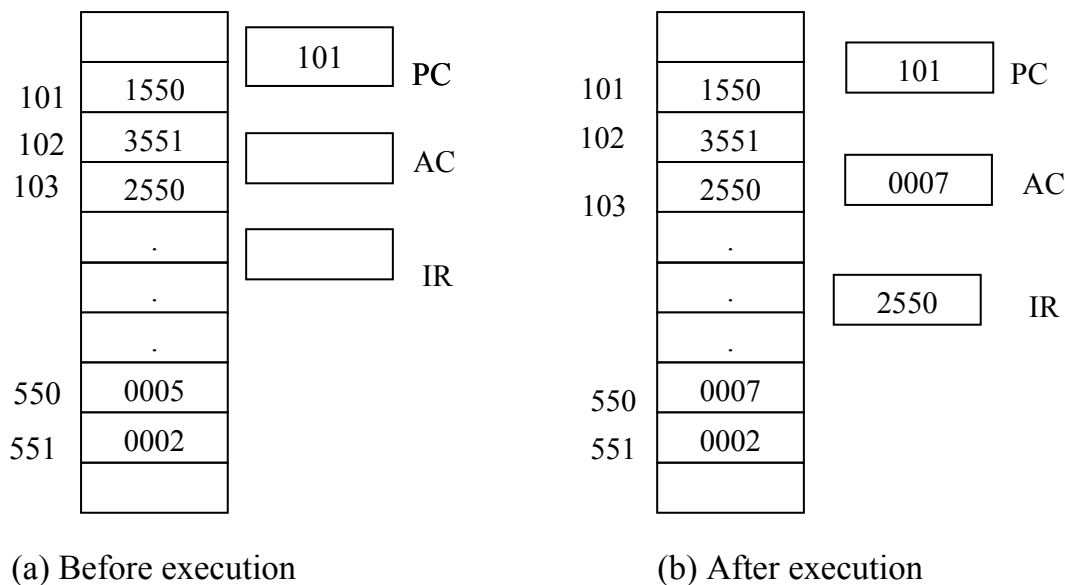
Opcode	Operand	Address	Operation desired
0001	0101	0101	0000 (Load accumulator)
0011	0101	0101	0001 (Add value from memory to accumulators)
0010	0101	0101	0000 (Store AC in the memory)

The hexadecimal notations for these instructions are:

1 5 5 0  
 3 5 5 1  
 2 5 5 0

Let us assume that these instructions are stored in three consecutive memory locations (all addresses are in hex notation) 101, 102 and 103 (we will use hexadecimal notation for this example) and the PC contains a value (101), which in turn is the address of first of these instructions.

(Please refer to Figure 11).



**Figure 11: Memory and registers content on execution of the three given consecutive instructions.**

Since the PC contains value 101, on the next instruction cycle the address stored in the PC is passed to MAR which in turn helps in accessing the memory location 101 and brings its content in MBR which in turn passes it on to IR. The PC is incremented to contain 102 now. The IR has the value 1550, which is decoded as “Load the content of address 550 in the accumulator”.

Thus, the accumulator register is loaded with the content of location 550, which is 0005. Now the instruction 101 execution is complete, and the next instruction that is 102 (indicated by PC) is fetched and PC is incremented to 103. This instruction is 3551, which instructs the CPU to add the contents of memory location 551 to the accumulator.

Therefore, the accumulator will now contain the sum of its earlier value and the value stored in memory location 551.

On the execution of the instruction at memory location 103, PC becomes 104; the accumulator results are stored in location 550 and IR still contains the third instruction. This state is shown in Figure 11(b).

Please note that the execution of the instructions in the above example requires only data transfer and data processing operations. All the instructions of the example require one memory reference during their execution. Does an instruction require more than one memory reference or operand address? Well, one such instruction is ADD A, B of PDP – 11. The execute cycle of this instruction may consist of steps such as:

- Decode that instruction which is ADD
- Read memory location A (only the contents) into the CPU.
- Read memory location B (only the contents) into the CPU. (Here, we are assuming that the CPU has at least two registers for storing memory location contents. The contents of location A and location B need to be written in two different registers).
- Add values of the two above registers.
- Write back the result from the register containing the sum (in the CPU) to the memory location.

Thus, in general, the execution cycle for a particular instruction may involve more than one stage and memory reference. In addition, an instruction may ask for an I/O operation. Considering the above situations, let us work out a more detailed view of an instruction cycle. Figure 12 gives a state diagram of an instruction cycle. A state is defined as a particular instance of instruction execution.

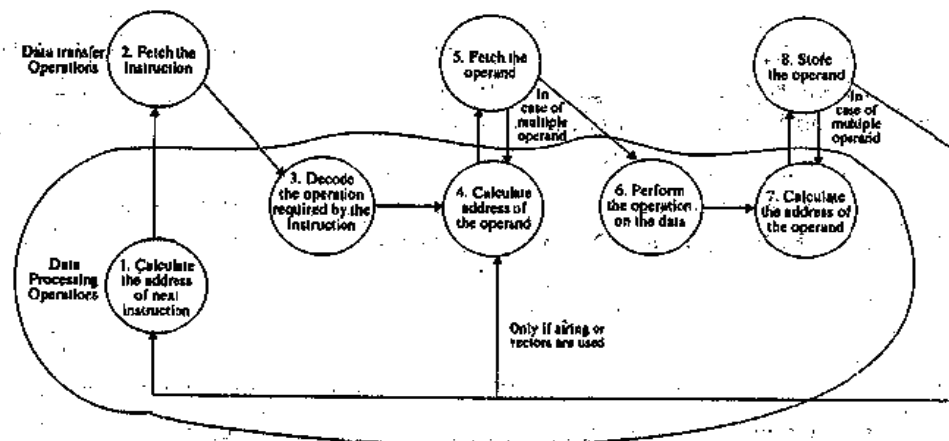


Figure 12: An instruction cycle

Please note that in the above state diagram some states may be bypassed while some may be visited more than once. The instruction cycle shown in Figure 12 consists of the following states/stages:

- First the address of the next instruction is calculated based on the width instruction and memory organisation. For example, if in a computer an instruction is of 16 bits and if the memory is organised as 16-bit words, then the address of the next instruction is evaluated by adding one to the address of the previous instruction. In case the memory is organised as bytes, which can be addressed individually, then we need to add two to the previous address to get the address of

the next instruction. In most computers this information is available in the PC register.

- An instruction is fetched from a memory location to the CPU.
- The next state decodes the instruction to determine the type of operation desired and the operands to be used.
- In case the operands need to be fetched from memory or via I/O, then the address of the memory location or I/O device is calculated.
- The operand is fetched from the memory or read from the I/O device.
- The operation asked by the instruction is performed.
- Finally, the results are written back to the memory or I/O wherever desired by first calculating the address of the operand and then transferring the value.

Please note that multiple operands and multiple results are allowed in many computers. An example of such a case is PDP-11 where an instruction ADD A, B requires operand A and B to be fetched. The sequence of states reference in this case is: State 1-2-3-4-5-6-7-8.

In certain machines a single instruction can issue an operation to be performed on an array of numbers or a string of characters. Such an operation involves repeated fetch for the operands.

Before winding up our discussion on instruction execution, let us discuss one very important mechanism, which plays an important role in input/output and other processing.

## **Interrupts**

The term interrupt is defined loosely to any exceptional event that causes the CPU to temporarily transfer its control from a currently executing program to a different program which provides service to the exceptional event. An interrupt may be generated by a number of sources, which may be either internal or external to the CPU. Figure 13 gives the list of the classes of very common interrupts along with some events in which they may occur.

Program Interrupts or traps	Generated internally by the CPU, on certain exceptional events during instruction execution. Some of these events can be: <ul style="list-style-type: none"> <li>• Division by zero</li> <li>• Arithmetic overflow</li> <li>• Attempt at executing an illegal/privileged instruction.</li> <li>• Trying to reference location other than allowed for that program.</li> </ul>
Timer Interrupts	Generated by a timer within the processor. It has a special significance for multiprogramming operating systems.
I/O Interrupts	Generated by input/output controllers. The request of starting an Input/output operation or signaling normal completion of an Input/output operation or signaling the occurrence of an error in Input/output operation.
Hardware failure	Generated on hardware failure. Some of these failures can be: <ul style="list-style-type: none"> <li>• Power failure</li> <li>• Memory parity error.</li> </ul>

**Figure 13: Various classes of interrupts**

Interrupts are a useful mechanism; they are mainly used for improving the efficiency of processing. Why? The main cause is in the fact that almost all the external devices are slower than the processor, therefore, in a typical system a processor has to continually test the status of the I/O device; in turn wasting a lot of CPU time. With the interrupt facility the CPU is freed from the task of testing the status of the input/output device and can do useful processing during this time, thus increasing the processing efficiency.

How does the CPU know that an interrupt has occurred? The CPU can be interrupted by providing a control line. This line, also known as the interrupt request line, connects the source/sources of interrupts to the CPU. The interrupt signal is then stored in a register of the CPU. This register is periodically tested by the CPU (when? we will answer it a little later) to determine the occurrence of an interrupt. Once the CPU knows that an interrupt has occurred then what? The CPU then needs to execute an interrupt servicing programme, which tries to remove/service the condition/device, which has caused the interrupt. In general, each source of interrupt requires a different interrupt servicing program to be executed. The CPU is normally assigned the address of the interrupt servicing program to be executed. Considering these requirements let us

work out the steps which CPU must perform on the occurrence of an interrupt.

1. The CPU must find out the source of the interrupt, as this will determine which interrupt service program is to be executed.
2. The CPU requires the addresses of the interrupt service routine, which are normally stored in the memory.
3. What happens to the program it was executing before the interrupt? This program needs to be interrupted till the CPU executes the interrupt service program. Do we need to do something for this program? Well, the context of this program is to be saved. We will discuss this a bit later.
4. Finally, the CPU executes the interrupt service program till the completion of the program. A RETURN statement marks the ends of this program. After that the control is passed back to the interrupted program.

Let us analyse some of the above points in greater details.

Let us first discuss saving the context of a program. The execution of a program in the CPU is done using certain sets of registers and their respective circuitry. As the CPU registers are also used by the interrupt service program it is very likely that these routines alter the content of several registers, therefore, it is the responsibility of the operating system that before an interrupt service program is executed the previous context of the CPU registers should be stored, such that the execution of the interrupted program can be restarted without any change from the point of interruption. Therefore, at the beginning of interrupt processing the essential context of the processor is saved either into a special save area in the main memory or into a stack. This context is restored when the interrupt service program is finished, thus, the interrupted program execution can be restarted from the point of interruption.

**Interrupts and the Instruction Cycle:** Let us summarise the interrupt process. On the occurrence of an interrupt, an interrupt request (in the form of a signal) is issued to the CPU. The CPU immediately suspends the operation of the currently executing program, saves the context of this program and starts executing a program which services that interrupt request. This program is also known as an interrupt handler. After the interrupting condition/device has been serviced the execution of the original program is resumed.

A user program can perceive the interrupt as: the interruption of the execution in between. The execution resumes as soon as the interrupt processing is completed. Therefore, the user program need not contain any special code for interrupt handling. This job is to be performed by the processor and the operating system, which in turn is also responsible for suspending the execution of the user program, and later interrupt handling resumes the user program from the same point.

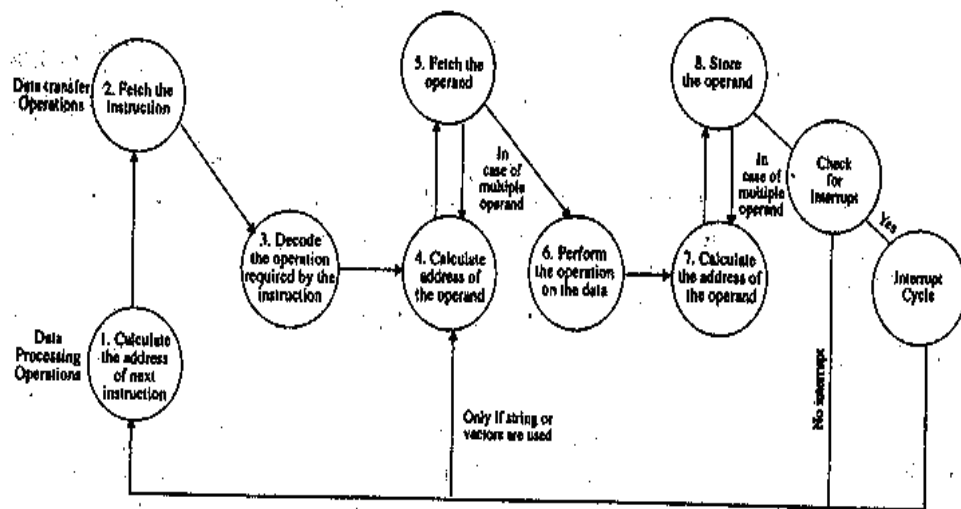


Figure 14: An instruction cycle with an interrupt cycle

The interrupts can be added in the instruction cycle (refer Figure 14) as an additional state. In the interrupt cycle, the responsibility of the CPU/processor is to check whether any interrupts have occurred. This is indicated by the presence of the interrupt signal. In case no interrupt needs service then the processor proceeds to the next instruction of the current program. In case an interrupt needs servicing then the interrupt is processed as follows:

- Suspend the execution of the current program and save its context and return address (point of interruption)
- Set the program counter to the starting address of this interrupt service program.
- This processor then executes the instructions in the interrupt-servicing program. The interrupt servicing programs are normally part of the operating system.
- After completing the interrupt servicing program the CPU can resume the program it had suspended earlier.



## Multiple Interrupts

Is it possible that multiple interrupts occur at the same time? Yes, it is possible to have multiple interrupts occurring at the same time. For example, a program from a communication line may receive the data and on the other side it is printing results. The printer on completion of every print operation will generate an interrupt, while the communication line controller will be generating the interrupt on arrival of a unit of data. Since these two interrupts are independent, the communication interrupt may occur while the printer interrupt is being processed.

### How to deal with these multiple interrupts

One of the approaches is to disable (do not allow other interrupts to be processed). If an interrupt occurs while the first interrupt is being processed then it will remain pending till the interrupt has been enabled again. Therefore, in this scheme the first few instructions in the processing of an interrupt disable other interrupts. After the interrupt service program for the current interrupt is completed, then the processor enables the interrupts and checks whether any other interrupt has occurred. Thus, in this approach interrupts are handled in sequential order.

The main drawback of this approach is that it does not consider the relative priority or time-critical nature of some interrupts. For example, while inputting from a communication line, the data need to be accepted quickly so that room can be made for more input. In case the first burst of data input is not processed before the second burst arrives, the data may be lost.

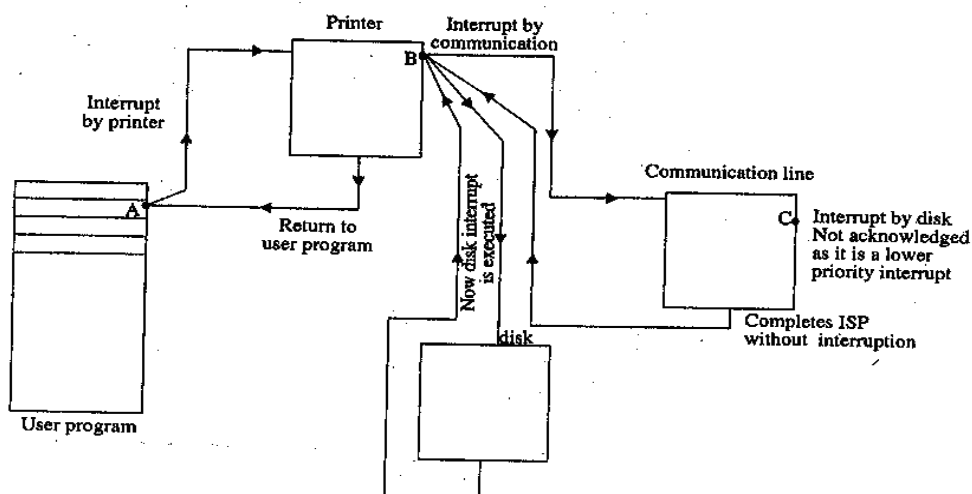


Figure 15: An example of multiple interrupts with priority scheme

For such cases another scheme in which priorities for interrupts is defined is followed. This scheme allows the higher priority interrupts to interrupt a lower priority interrupt service program in between.

For example, let us assume a system with three input/output devices: printer, disk, and communication line. Let us assume that these have priorities of 1, 2 and 3 respectively. Priority 3 is the highest while 1 is lowest in our case. An instance of program execution is shown with multiple interrupts in Figure 15.

Initially the user program is being executed by the CPU till the point A, where a printer interrupt occurs. The CPU acknowledges the printer interrupt by placing the context of the user program on the system stack and by starting to execute the interrupt servicing program (ISP) for the printer, before the ISP of the printer is finished as point B communication line interrupt occurs. Since this is a higher priority interrupt, the CPU acknowledges the interrupt and pushes the context of the ISP of the printer from the CPU to the system stack and starts executing the ISP (communication line). While processing the ISP (communication line) another interrupt of disk occurs but it is held back as the ISP (communication line) is of higher priority. When the ISP (communication line) finishes the context of ISP (printer) is executed again, but as the disk interrupt has higher priority and is pending, the disk interrupt is acknowledged and processed at point B. After completing the ISP (disk) the ISP (printer) is completed and control is passed back to user program, which resumes executing from the point of interruption.

#### **4.0 CONCLUSION**

This unit has taken you through the history of computers as well as how information is represented in a computer by first recapitulating the number systems.

Also, error detection and correction codes such as parity bit and having error-correction codes have been extensively discussed, together with the instruction (program) execution which is the basic function performed by a computer and the major responsibility of the CPU.

The basic concepts of the instruction cycle, interrupts and the correlation between this two have been thoroughly examined.

## 5.0 SUMMARY

This completes our discussion on the introductory concepts of computer architecture. The von-Neumann architecture discussed in the unit is not the only architecture but many new architectures have come up which you will find in the further readings.

The information given on various topics such as interrupts, data representation, error detection codes etc. although exhaustive, can be supplemented with additional readings. In fact, a course in an area of computer science must be supplemented by further reading to keep knowledge up to date, as the computer world keeps changing. You are advised to study several journals on, computers to enhance your knowledge.

## 6.0 TUTOR-MARKED ASSIGNMENT

1. Convert the following binary numbers to decimal
  - (a) 1100,1101
  - (b) 10101010
2. Convert the following decimal numbers to binary.
  - (a) 23    (b) 49.25    (c) 892
3. Convert the numbers given in question 2 to hexadecimal from decimal or from the binary.
4. Write the BCD equivalent the three numbers given in question 2.
5. Find the 1's and 2's complement of the following fixed-point numbers.
  - (a) 10100010
  - (b) 00000000
  - (c) 11001100
6. Add the following numbers in an 8-bit register using signed 2's complement notation.
  - (a) +50 and -5
  - (b) +45 and -65
  - (c) +75 and +85
  - (d) - 75 and – 85

Also indicate the overflow condition if any.

7. Find the even and odd parity bits for the following 7-bit data:  
 (a) 0101010 (b) 0000000 (c) 1111111 (d) 1000100
8. Find the length of SEC code and SEC-DED code for 16-bit word data transfer.

**State whether True or False**

9. The value of a PC will be incremented by 1 after fetching each instruction if the memory word is of one byte and an instruction is 16 bits long. True  False
10. Both MAR and MBR are needed to fetch the data or instruction from the memory. True  False
11. A clock may generate an interrupt. This is known as a timer interrupt. True  False
12. Context switching is not desired before interrupt processing True  False
13. In case multiple interrupts occur at the same time, only one of the interrupts will be acknowledged and the rest will be lost. True  False

**7.0 REFERENCES/FURTHER READINGS**

Mano, M. Morris (1993). *Computer System Architecture* (3<sup>rd</sup> ed). India: Prentice Hall.

Hayes, John P. (1988). *Computer Architecture and Organisation* (2<sup>nd</sup> ed). McGraw-Hill International editions.

Stallings William. *Computer Organisation and Architecture* (3<sup>rd</sup> ed). Maxwell Macmillan International Editions.

Baron, Robert J. and Higbie Lee. *Computer Architecture*. Addison-Wesley Publishing Company.

Tanenbaum, Andrew S. (1993) *Structural Computer Organisation* (3<sup>rd</sup> ed). India: Prentice Hall.

## UNIT 2 DIGITAL LOGIC CIRCUITS

### CONTENTS

- 1.0 Introduction
- 2.0 Objectives
- 3.0 Main Content
  - 3.1 Boolean Algebra
  - 3.2 Logic Gates
  - 3.3 Combination Circuits
    - 3.3.1 Minimising of Gates
    - 3.3.2 The Multiplexer
    - 3.3.3 Decoders
    - 3.3.4 Programmable Logic Array
    - 3.3.5 Adders
  - 3.4 Sequential Circuits
    - 3.4.1 Flip-Flops
    - 3.4.2 Registers
    - 3.4.3 Counters
  - 3.5 Interconnection Structures
- 4.0 Conclusion
- 5.0 Summary
- 6.0 Tutor-Marked Assignment
- 7.0 References/Further Readings

### 1.0 INTRODUCTION

By now you should be aware of the basic configuration of a computer system and the various terms related to it. In this unit, you will be exposed to some of the basic components which form the most essential part of a computer. You will come across terms like computer bus, binary adders, logic gates, flip-flops combinational and sequential circuits, etc. These circuits are the backbone of any computer system and knowing them is essential. After this unit we will discuss memory organisation and I/O organisation and then we will discuss CPU organisation later. But the bases for all those are the digital logic circuits only, which we are going to discuss in this unit.

### 2.0 OBJECTIVES

This unit is an attempt to answer one basic query “How does a computer actually perform computations?”

At the end of this unit you will be able to describe:

- What the flip-flops and gates are

- Combination and sequential circuits and their applications thereof
- Some of the useful circuits of a computer system such as multiplexers, decoders etc.
- How a very basic mathematical operation- the addition- is performed by a computer and
- One of the most widely used interconnecting mechanisms: the computer bus.

### 3.0 MAIN CONTENT

#### 3.1 Boolean Algebra

Before going further, let us briefly recapitulate the Boolean algebra, which will be useful in the discussions on logic circuits. The Boolean algebra is an attempt to represent the true-false logic of humans in mathematical form. George Boole proposed the principles of the Boolean algebra in 1854, hence the name Boolean algebra. Boolean algebra is used for designing and analysing digital circuits.

Let us first discuss the rules of Boolean algebra and thereafter we will discuss how it can be used in analysing or designing digital circuits.

##### Point 1:

A variable in Boolean algebra can take only two values

1 (TRUE) or 0 (FALSE)

##### Point 2:

There are three basic operations in Boolean algebra, viz:

AND, OR and NOT

(These operators will be given in capitals in this module for differentiating them from normal and, or, not, etc)

A AND B or A.B or AB

A OR B or A + B

NOT A or  $\neg A$  or A' or  $\bar{A}$

But how the value of A AND B changes with the values of A and B can be represented in tabular form, which is referred to as the “truth table”.

A	B	A AND B	A OR B	NOT A
0	0	0	0	1
0	1	0	1	1
1	0	0	1	0
1	1	1	1	0

In addition three more operators have been defined for Boolean algebra:

XOR (Exclusive OR), NOR (Not OR) and NAND (Not AND)

However, for designing and analysing a logical circuit, it is convenient to use AND, NOT and OR operators because AND and OR obey many laws as of multiplication and addition in the ordinary algebra of real numbers.

We will discuss XOR, NOR and NAND operators in the next subsection.

### Point 3:

The basic logical identities used in Boolean algebra are:

#### ***BASIC IDENTITIES***

$A.B = B.A$	$A+B = B+A$	Commutative law
$A.(B+C) = (A.B)+(A.C)$	$A+(B.C) = (A+B).(A+C)$	Distributive law
$1.A = A$	$0+A = A$	Identity law
$A.\bar{A} = 0$	$A+\bar{A} = 1$	Inverse law

#### ***OTHER IDENTITIES***

$0.A = 0$	$1+A = 1$	
$A.A = A$	$A+A = A$	
$\underline{A}.\underline{(B.C)} = \underline{(A.B)}.C$	$\underline{A}+\underline{(B+C)} = (A+B)+C$	Associative law
$A.B = \bar{A}+B$	$A+B = \bar{A}.B$	Demorgan's Theorem

We will not give any proofs of these identities. The use of some of the identities is shown in the example given after Boolean function definition. DeMorgan's law is very useful in simplifying logical expressions.

### **Boolean Function:**

A Boolean function is defined as an algebraic expression formed with the binary variables, the logic operation symbols, parenthesis, and equal to sign. For example,

$$F = A.B+C \text{ is a Boolean function.}$$

The value of Boolean function F can be either 0 or 1.

A Boolean function can be broken into logic diagram and vice versa (we will discuss this in the next section), therefore if we code the logic operations in Boolean algebraic form and simplify this expression we will design the simplified form of the logic circuits.

Let us see how DeMorgan theorem and other Boolean identities help in simplifying the digital logic circuits. Let us consider the Boolean function.

$$F = \overline{\overline{(A+B)} + \overline{B}}$$

Now let us simplify the function,

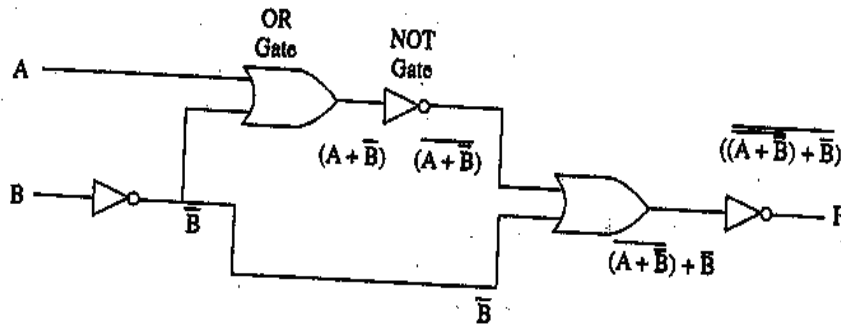
$$\begin{aligned} F &= \overline{\overline{(A+B)} + \overline{B}} \\ &= \overline{[A + (\overline{B})].\overline{B}} \quad \text{On applying DeMorgan's theorem in } \overline{\overline{(A+B)} + \overline{B}} \\ &= \overline{[(A + \overline{B}).B]} \quad \text{As } \overline{\overline{B}} = B \text{ and } \overline{\{A + \overline{B}\}} = (A + \overline{B}) \\ &= A.B + \overline{B}.B \quad \text{On applying Distributive law} \\ &= (A.B + 0) \quad \text{[Property } \overline{B}.B = 0] \\ &= (A.B) \quad \text{[Property } A+0 = A] \end{aligned}$$

The simplified Boolean function is

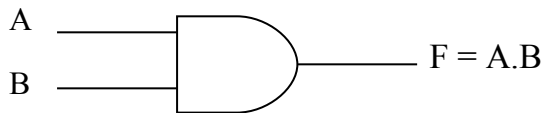
$$F = A.B$$

Figure 16 shows the two logic diagrams. The meaning of these symbols will be clear in the next subsection.





(a) Before simplifying the Boolean function



(b) After simplifying the Boolean function

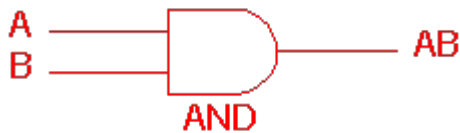
**Figure 16: The logic diagram of two equivalent circuits**

### 3.2 Logic Gates

Digital systems are said to be constructed by using logic gates. A logic gate is an electronic circuit which produces a typical output signal depending on its input signal. The output signal of a gate is a simple Boolean operation of its input signal(s). Gates are the basic logic elements. These gates are the AND, OR, NOT, NAND, NOR, EXOR and EXNOR gates. Any Boolean function can be represented in the form of gates.

The basic operations are described below with the aid of truth tables.

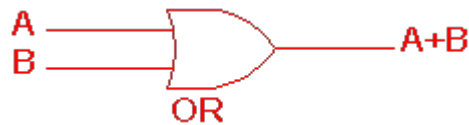
#### AND gate



2 Input AND gate		
A	B	A.B
0	0	0
0	1	0
1	0	0
1	1	1

The AND gate is an electronic circuit that gives a **high** output (1) only if **all** its inputs are high. A dot (.) is used to show the AND operation i.e. A.B. Bear in mind that this dot is sometimes omitted i.e. AB

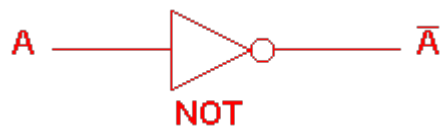
## OR gate



2 Input OR gate		
A	B	A+B
0	0	0
0	1	1
1	0	1
1	1	1

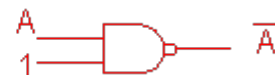
The OR gate is an electronic circuit that gives a high output (1) if **one or more** of its inputs are high. A plus (+) is used to show the OR operation.

## NOT gate



NOT gate	
A	$\bar{A}$
0	1
1	0

The NOT gate is an electronic circuit that produces an inverted version of the input at its output. It is also known as an **inverter**. If the input variable is A, the inverted output is known as NOT A. This is also shown as A', or A with a bar over the top, as shown at the outputs. The diagrams below show two ways in which the NAND logic gate can be configured to produce a NOT gate. It can also be done using NOR logic gates in the same way.



## NAND gate



2 Input NAND gate		
A	B	$\overline{A \cdot B}$
0	0	1
0	1	1
1	0	1
1	1	0

This is a NOT-AND gate which is equal to an AND gate followed by a NOT gate. The outputs of all NAND gates are high if **any** of the inputs are low. The symbol is an AND gate with a small circle on the output. The small circle represents an inversion.

### NOR gate



2 Input NOR gate		
A	B	$\overline{A+B}$
0	0	1
0	1	0
1	0	0
1	1	0

This is a NOT-OR gate which is equal to an OR gate followed by a NOT gate. The outputs of all NOR gates are low if **any** of the inputs are high.

The symbol is an OR gate with a small circle on the output. The small circle represents an inversion.

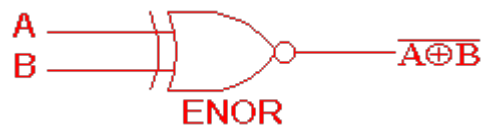
### EXOR gate



2 Input EXOR gate		
A	B	$A\oplus B$
0	0	0
0	1	1
1	0	1
1	1	0

The '**Exclusive-OR**' gate is a circuit which will give a high output if **either, but not both**, of its two inputs are high. An encircled plus sign ( $\oplus$ ) is used to show the EOR operation.

### EXNOR gate



2 Input EXNOR gate		
A	B	$\overline{A \oplus B}$
0	0	1
0	1	0
1	0	0
1	1	1

The '**Exclusive-NOR**' gate circuit does the opposite of the EOR gate. It will give a low output if **either, but not both** of its two inputs are high. The symbol is an EXOR gate with a small circle on the output. The small circle represents an inversion.

The NAND and NOR gates are called **universal functions** since with either one the AND and OR functions and NOT can be generated.

#### Note:

A function in **sum of products** form can be implemented using NAND gates by replacing all AND and OR gates by NAND gates.

A function in **product of sums** form can be implemented using NOR gates by replacing all AND and OR gates by NOR gates.

**Table 1: Logic gate symbols**

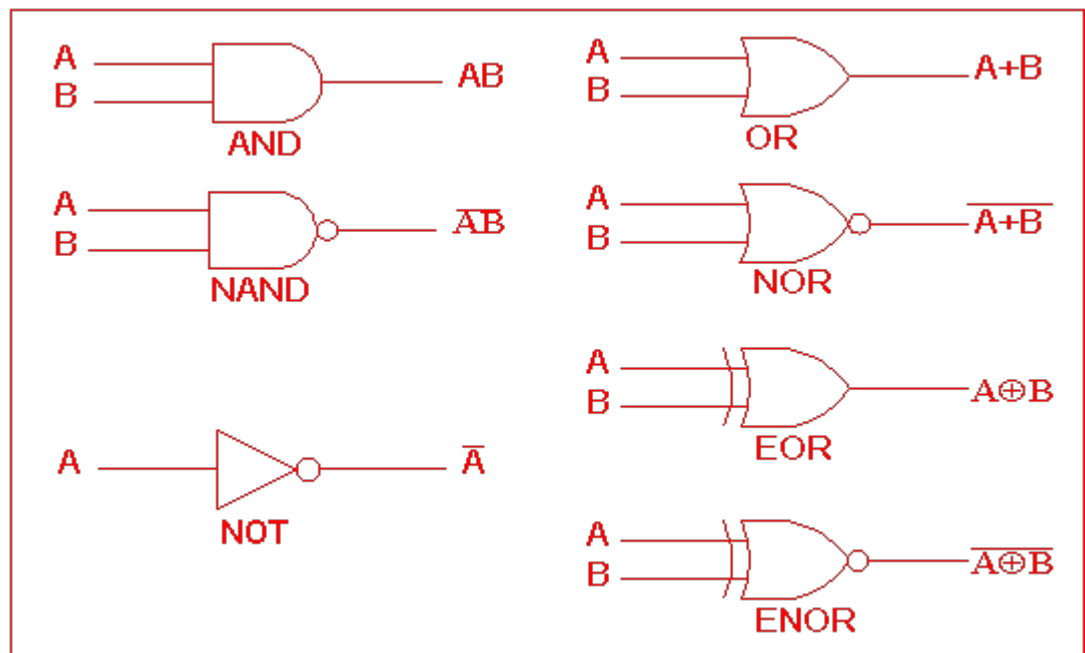


Table 2 is a summary truth table of the input/output combinations for the NOT gate together with all possible input/output combinations for the

other gate functions. Also note that a truth table with 'n' inputs has  $2^n$  rows. You can compare the outputs of different gates.


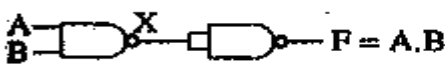
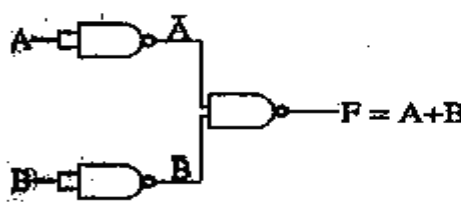
**Table 2: Logic gates representation using the Truth table**

		INPUTS		OUTPUTS					
		A	B	AND	NAND	OR	NOR	EXOR	EXNOR
<b>NOT gate</b>									
A	$\bar{A}$	0	1	0	1	1	0	1	0
0	1	1	0	0	1	1	0	1	0
1	0	1	1	1	0	1	0	0	1

The truth table of NAND and NOR can be made from NOT (A AND B) and NOT (A OR B) respectively. Exclusive OR (XOR) is a special gate whose output is one only if the inputs are not equal. The inverse of exclusive OR can be a comparator which will produce a one output if two inputs are equal.

The digital circuit use only one or two types of gates for simplicity in fabrication purposes. Therefore, one must think in terms of functionally complete sets of gates. What does a functionally complete set imply? A set of gates by which any Boolean function can be implemented is called a functionally complete set. The functionally complete sets are: (AND, NOT), (NOR), (NAND), (OR < NOT) etc.

**Example:** Let us take the NAND gate and try to represent NOT, AND and OR operations through it.

	Truth table		Output		
	<b>A</b>	<b>B</b>			
	0	0			
	1	1			
<p>⇒ This represent a NOT operation</p> 	<b>A</b>	<b>B</b>	<b>X</b>	<b>X</b>	<b>F</b>
	0	0	1	1	0
	0	1	1	1	0
	1	0	1	1	0
	1	1	0	0	1
<p>⇒ This is an AND operation</p> 	<b>A</b>	<b>B</b>	<b>A</b>	<b>B</b>	<b>F</b>
	0	0	1	1	0
	0	1	1	0	1
	1	0	0	1	1
	1	1	0	0	1
<p>⇒ This is an OR operation</p>					

Similarly NOR gate can be used to implement any Boolean function.

### SELF-ASSESSMENT EXERCISE 1

1. Simplify the Boolean function.

$$F = \overline{(\overline{A + B}) + (\overline{A + B})}$$

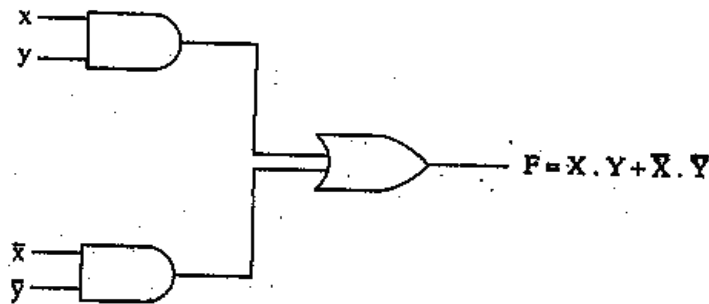
2. Draw the logic diagram of the above function
3. Draw the logic diagram of the simplified function.

### 3.3 Combination Circuits

Combinational circuits are interconnected circuits of gates according to a certain rule to produce an output depending on its input value. A well-formed combinational circuit should not have feedback loops. A combinational circuit can be represented as a network of gates and, can be expressed by a truth table or a Boolean expression.

The output of a combinational circuit is related to its input by a combinational function, which is independent of time. Therefore, for an ideal combinational circuit the output should change instantaneously according to changes in input. But in actual cases there is a slight delay. This delay is normally proportional to the depth or number of levels, i.e.

the maximum number of gates lying on any path from input to output. For example, the depth of the combinational circuit in figure 3 is two.



**Figure 17: A two level AND-OR combinational circuit**

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

- The depth of the circuit should not exceed a specific level.
- Number of input lines to a gate (fan in) and how many gates its output can be fed (fan out) are constrained by the circuit power constraints.

### 3.3.1 Minimisation of Gates

The simplification of the Boolean expression is very useful for combinational circuit designs. The following three methods are used for this.

- Algebraic simplification
- Karnaugh maps
- The Quine McCluskey method

But before defining any of the above stated methods let us discuss the forms of algebraic expressions. An algebraic expression can exist in two forms:

- Sum of products (SOP) e.g.  $(A.\neg B) + (\neg A.\neg B)$
- Product of sums (POS) e.g.  $(\neg A + \neg B).(A+B)$

If a product of SOP expression contains every variable of that function either in true or complement form then it is defined as a minterm. This minterm will be true only for one combination of input values of the variables. For example, in the SOP expression-

$$F(A,B,C) = (A.B.C) + (\neg A.\neg B.C) + (A.B)$$

We, have three product terms namely  $A.B.C$ ,  $\neg A. \neg B.C$  and  $A.B$ . But only the first two of them qualify to be a minterm as the third one does not contain variable  $C$  or its complement. In addition, the term  $A.B.C$  will be one only if  $A=1$ ,  $B=1$  and  $C=1$  for any other combination of values of  $A,B,C$  the minterm will have zero value. Similarly, the minterm  $\neg A. \neg B.C$  will have value 1 only if  $\neg A = 1$  i.e.  $A=0$ ,  $\neg B=1$  i.e.  $B=0$  and  $C=1$ . For any other combination of values the minterm will have a zero value.

A similar type of term used in POS form is called maxterm. Maxterm is a term of POS expression, which contains all the variables of the function in true or complemented form. For example,  $F(A,B,C)=(A=B=C).(\neg A+\neg B+C)$  have two maxterms. A maxterm have a value 0 for only one combination of input values. The maxterm  $A+B+C$  will be 0 value only for  $A=0$ ,  $B=0$  and  $C=0$ . For all other combination of values of  $A, B, C$  it will have a value one.

Now let us come back to the problem of minimising the number of gates.

### **Algebraic Simplification**

We have already discussed the algebraic simplification of a logical circuit. An algebraic expression can exist in POS or SOP forms. The algebraic functions can appear in many different forms. Although the process of simplification exists yet it is cumbersome because of the absence of routes, which tell what rule to apply next. The Karnaugh map is a simple direct approach of simplification of logical expressions.

### **Karnaugh Maps**

The Karnaugh map is a convenient way of representing and simplifying Boolean functions of 4 to 6 variables. Karnaugh maps can also be used for designing the circuits in situations where you can construct the truth table for an operation or a function. In other words, Karnaugh maps can be used to construct a circuit when the input and output to that proposed circuit are defined. For each output one Karnaugh map needs to be constructed. The stepwise procedure for Karnaugh map is as follows:

**Step 1** Create a simple map depending on the number of variables in the function. Figure 18(a) shows the map of two, three and four variables. A map of 2 variables contains 4 value positions or elements, while for 3 variables it has  $2^3=8$  elements; similarly for 4 variables it is  $2^4=16$  elements and so on. Special care is taken to represent variables in the



map. Value of only one variable changes in two adjacent columns or rows. The advantage of having a change in one variable is that two adjacent columns or rows represent a true form or complement form of a single variable. For example, in Figure 18(b) the columns which have positive A are adjacent to  $\neg A$ . Please note the adjacency of the corners. The rightmost column can be considered to be adjacent to the first column; since they differ only by one variable, therefore, they are adjacent. Similarly the topmost and bottommost rows are adjacent.

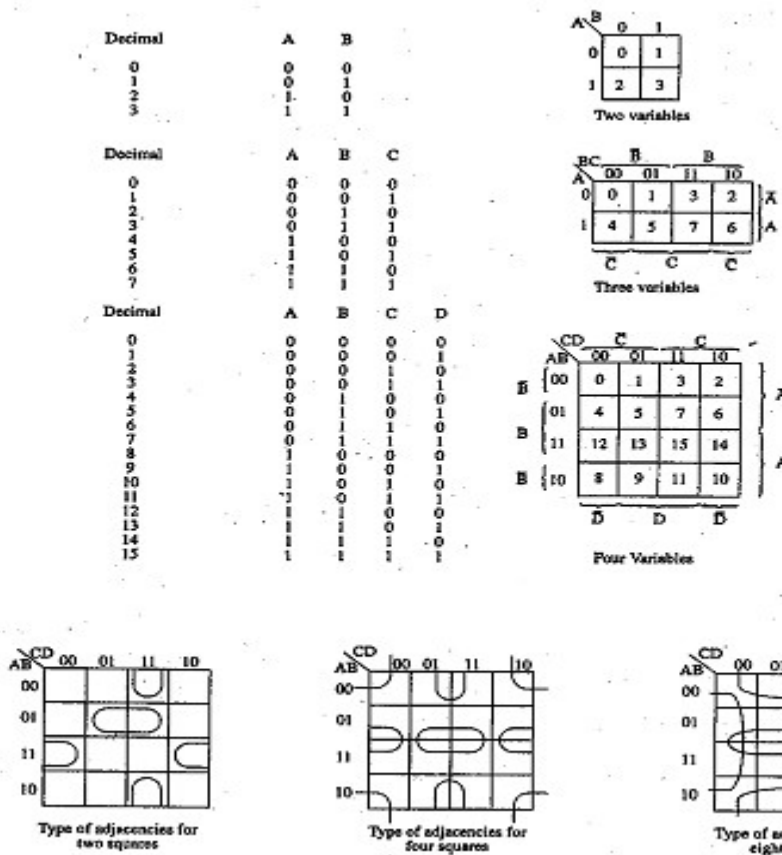


Figure 18: Maps of two, three and four variables and their adjacencies.

**Please note:**

1. Decimal equivalent of the cells are given for help in understanding where the position of the respective decimal equivalent is. It is not the value filled in a square. A square can contain one or nothing.
2. The 00, 01, 11 etc. written on the top implies the value of respective variables.

3. Wherever the value of a variable is zero it is said to represent its complement form.
4. The value of only one variable changes when we move from one row to the next row or from one column to the next column.

**Step 2:** The next step in the Karnaugh map is to map the truth table into the map. The mapping is done by putting a 1 in the respective squares belonging to the 1 value in the truth table. This mapped map is used to arrive at simplified Boolean expressions, which then can be used for drawing up the optimal logical circuits. Step 2 will be clearer in the example.

**Step 3:** Now create simple algebraic expressions from the Karnaugh map. These expressions are created by using adjacency if we have two adjacent 1's then the expressions for those can be simplified together since they differ in only one variable. Similarly we search for adjacent pairs of four, eight, and so on. A 1 can appear in more than one adjacent pairs. You must find adjacencies till all 1's in the Karnaugh map are covered. The following example will clarify step 3.

**Example 2:** Now let us see how to use Karnaugh map simplification for finding the Boolean function for the cases whose truth table is given as:

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

Another short representation of the truth table is  $\Sigma(0,1,2,6,8,9, 10)$  which indicate the decimal equivalent for A, B, C, D values for which the output is one.

Let us construct the Karnaugh map for this.

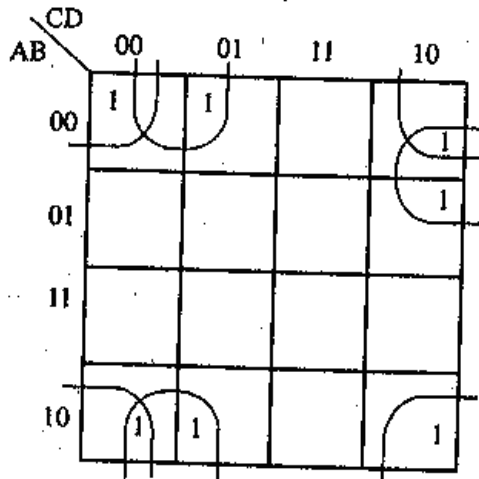


Figure 19: Karnaugh's map of the truth table of example 2

Let us see the pairs which can be considered adjacent in the Karnaugh map here.

The pairs are

1. The four corners
2. The four 1's as in top and bottom in columns 1 and 2
3. The two 1's in the top two rows of the last column.

The corners can be represented by the expressions:

$$\begin{aligned}
 1. \quad & (\neg A \neg B \neg C \neg D + \neg A \neg B C \neg D) + (A \neg B \neg C \neg D + A \neg B C \neg D) \\
 & = \neg A. \neg B. \neg D(\neg C + C) + A. \neg B. \neg D(\neg C + C) \\
 & = \neg A \neg B \neg D + A \neg B \neg D && \text{as } (\neg C + C) = 1 \\
 & = \neg B \neg D (\neg A + A) \\
 & = \neg B \neg D && \text{as } (\neg A + A) = 1
 \end{aligned}$$

2. The four 1's give the following term:

$$\begin{aligned}
 & (\neg A \neg B \neg C \neg D + \neg A \neg B \neg C D) + (A \neg B \neg C \neg D + A \neg B \neg C D) \\
 & = \neg A \neg B \neg C (\neg D + D) + A \neg B \neg C (\neg D + D) \\
 & = \neg A \neg B \neg C + A \neg B \neg C && \text{as } (\neg D + D) = 1 \\
 & = \neg B \neg C (\neg A + A) \\
 & = \neg B \neg C && \text{as } (\neg A + A) = 1
 \end{aligned}$$

3. The two 1's in the last column

$$\begin{aligned} & \neg A \neg B C \neg D + \neg A B C \neg D \\ & = \neg A C \neg D (\neg B + B) \\ & = \neg A C \neg D \qquad \qquad \qquad \text{as } (\neg B + B) = 1 \end{aligned}$$

Thus, the Boolean expression derived from this Karnaugh map is

$$F = \neg B \neg D + \neg B \neg C + \neg A C \neg D$$

The expressions so obtained through the Karnaugh map are in the form of the sum of the product form, i.e. it is expressed as a sum of the products of the variables. The expression is one of the minimal solutions. This expression can be expressed in product of the sum form, but for this, special methods need to be used.

Let us see how we can modify the Karnaugh map simplification to obtain the product of the sum form. Suppose in the previous example instead of using 1's we combine the adjacent zero square then we will obtain the inverse function and on taking NOT of this function we will get the product of sum form (the use of DeMorgan's theorem will be required).

Another important aspect of this simple method of digital circuit design is DONOT care conditions. These conditions further simplify the algebraic function. These conditions imply that it does not matter whether the output produced is zero or 1 for a specific input. These conditions can occur when the combination of the number of inputs are more than needed; e.g., calculation through BCD where 4 bits are used to represent a decimal digit, which implies we can represent  $2^4 = 16$  digits but since we have only 10 decimal digits, therefore, 6 of those input combinations do not matter and thus, are a candidate for DONOT care condition.

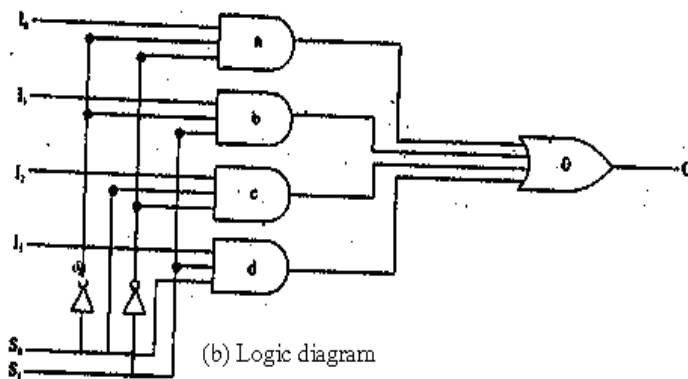
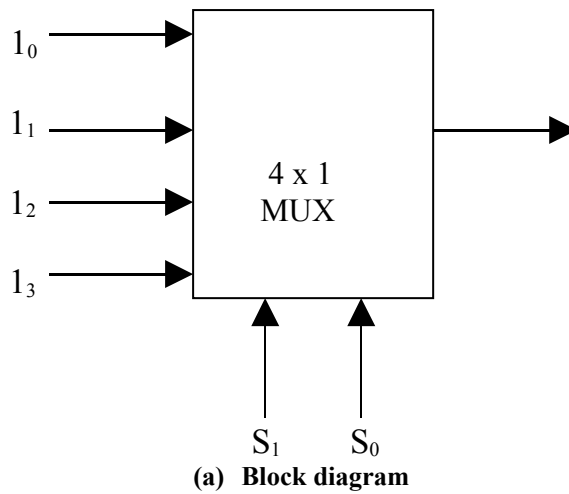
What will happen if we have more than 4-6 variables? As the number of the variables increases the Karnaugh map becomes more and more cumbersome (as the number of possible combination of inputs keeps on increasing). A method was suggested to deal with the increasing number of variables. This is a tabular approach and is known as the Quine-Mckluskey method. This method is suitable for programming and hence provides a tool for automating designs in the form of minimised Boolean expressions.

The basic principle behind the Quine- Mckluskey method is to remove the terms which are redundant and can be obtained by other terms. Discussions on this method are beyond this course.

Let us now discuss some important combinational circuits. We will not go to the details of their design in this unit.

### 3.3.2 The Multiplexer

The multiplexer is one of the basic building units of a computer system, which, in principle allows sharing of a common line by more than one input lines. It connects multiple input lines to a single output line. At a specific time one of the input lines is selected and the selected input is passed on to the output line. The diagram of a 4x1 multiplexer (MUX) is given in Figure 20.



S <sub>0</sub>	S <sub>1</sub>	O
0	0	I <sub>0</sub>
0	1	I <sub>1</sub>
1	0	I <sub>2</sub>
1	1	I <sub>3</sub>

(c) Truth table

**Figure20: A 4 x 1 multiplexer**

But how does the multiplexer know which line to select? The select lines control this operation. The select lines provide the communication

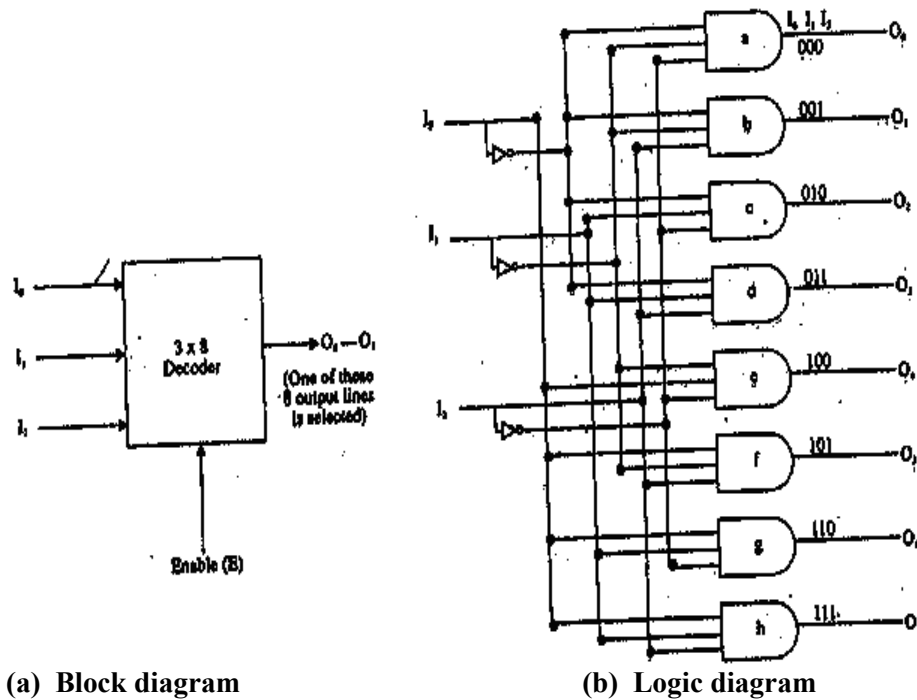
among the various components of a computer. Now let us see how the multiplexer also known as MUX works. Here for simplicity we will take the example of the 4 x 1 MUX i.e. there are 4 input lines connected to one output line. For the sake of consistency we will call input lines as  $I$ , and output lines as  $O$  and control line as a selection line  $S$  or enable as  $E$ .

Please notice the way in which  $S_0$  and  $S_1$  are connected in the circuit. To the 'a' AND gate  $S_0$  and  $S_1$  are inputted in complement form that mean 'a' gate will output  $I_0$  when both the selection lines have a value of 0 which implies  $\neg S_0 = 1$  that is  $S_0 = 0$  and  $S_1 = 0$  and hence the first entry in the truth table. Please note that at  $S_0 = 0$ , &  $S_1 = 0$  AND gates 'b', 'c' and 'd' will yield zero outputs and when all these outputs pass OR gate 'e' they will yield  $I_0$  as the output for this case. That is for  $S_0 = 0$  and  $S_1 = 0$ , the output becomes  $I_0$  which in other words can be said as "For  $S_0 = 0$  and  $S_1 = 0$ ,  $I_0$  input line is selected by MUX". Similarly the other entries in the truth table are corresponding to the logical nature of the diagram. Therefore, by having 2 control lines we could have a 4 x 1 MUX. To have 8 x 1 MUX we must have 3 control lines or with 3 control lines we could make  $2^3 = 8$  i.e. 8 x 1 MUX. Similarly with  $n$  control lines we can have  $2^n$  x 1 MUX. Another parameter that is predominant in MUX design is the number of inputs to OR gate. These inputs are determined by the voltage capacity of the gate, which normally is a maximum of eight inputs to a gate.

Where can these devices be used in a computer? The multiplexers are used in digital circuits for data and control signal routing. We have seen a concept where out of  $n$  input lines 1 can be selected. Can we have a reverse concept i.e. we have one input line and data is transmitted to one of the possible  $2^n$  lines where  $n$  represents the number of selection lines. This operation is called demultiplexing.

### 3.3.3 Decoders

Decoders convert one type of coded information to another form. A decoder has  $n$  input lines, one enable line (a sort of selection line) and  $2^n$  output lines. Let us see an example of a 3x8 decoder which decodes a 3-bit information and there is only one output line which gets a value 1 or in other words out of  $2^3=8$  lines only one output line is selected. Thus, depending on the selected output line the information of the 3 bits can be recognised or decoded.



I <sub>1</sub>	I <sub>2</sub>	I <sub>3</sub>	O <sub>0</sub>	O <sub>1</sub>	O <sub>2</sub>	O <sub>3</sub>	O <sub>4</sub>	O <sub>5</sub>	O <sub>6</sub>	O <sub>7</sub>
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

(c) Truth Table

**Figure 21: The logic diagram of a 3 x 8 decoder**

Please try to make the logic circuit of a decoder yourself by drawing a Karnaugh map for each output. Please note that in the logic diagram, wherever the values in the truth table are appearing as zero in input, and one in the output, the input should be fed in complemented form. E.g., the first four entries of truth table contain 0 in I<sub>0</sub> position and hence I<sub>0</sub> value zero is passed through a NOT gate and fed to AND gates 'a', 'b', 'c' and 'd' which implies that these gates will be activated/selected only if I<sub>0</sub> is 0. If I<sub>0</sub> value is 1 then none of the top four AND gates can be activated. A similar type of logic is valid for I<sub>1</sub>. Please note the output line selected is named 000 or 010 or 111, etc. The output value of only one of the lines will be 1. These 000, 010 indicate the label and suggest that if you have these I<sub>0</sub> I<sub>1</sub> I<sub>2</sub> input values then the labelled line will be selected for the output. The enable line is used for combining two 3x8

decoders to make one 4x16 decoder. How? Please find out from the suggested readings.

### 3.3.4 Programmable Logic Array

Till now, the individual gates are treated as basic building blocks from which various logic functions can be derived. We have also learnt about the strategies of minimisation of number of gates. But with the advancement of technology the integration provided by integrated circuit technology has increased resulting in the production of one to ten gates on a single chip (in Small Scale Integration). The gate level designs are constructed at the gate level only but if the design is to be done using these SSI chips, the design consideration needs to be changed as a number of such SSI chips may be used for developing a logic circuit. With MSI & VLSI we can put even more gates on a chip and can also make gate interconnections on a chip. This integration and connection bring the advantages of decreased cost, size, and increased speed. But the basic drawback faced in such VLSI & MSI chips is that for each logic function the layout of gates and interconnection need to be designed. The cost involved in making such a custom chip design is quite high. Thus came the concept of Programmable Logic Array (PLA), a general-purpose chip that can be readily adopted for any specific purpose.

The PLA are designed for SOP form of Boolean function and consist of a regular arrangement of NOT, AND & OR gates on a chip. Each input to the chip is passed through a NOT gate, thus, the input and their complement are available to each AND gate. The output of each AND gate is made available for each OR gate. The output of each OR gate is treated as chip output. By making appropriate connections any logic function can be implemented in these Programmable Logic Arrays.

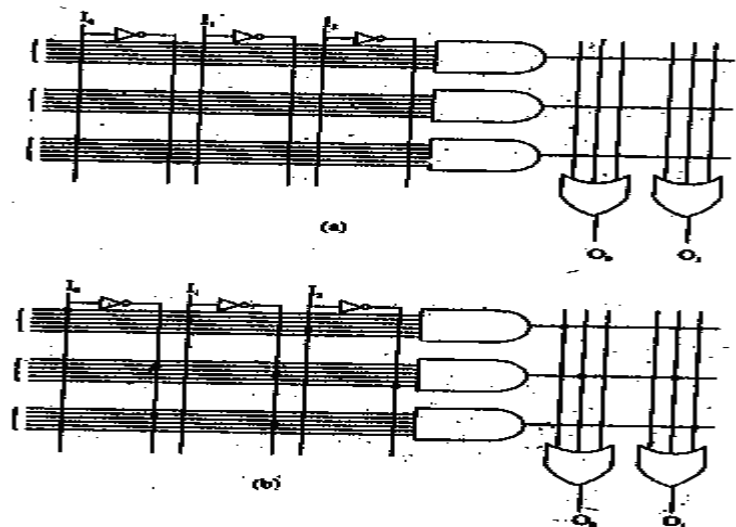


Figure 22: Programmable Logic Array



The figure 22(a) shows a PLA of three inputs and two outputs. Please note the connectivity points, all these points can be connected if desired.

Figure 22(b) shows an implementation of logic function:

$$O_0 = I_0, I_1, I_2 + \neg I_0 \neg I_1 \neg I_2 \text{ and } O_1 = \neg I_0 \neg I_1 \neg I_2 + \neg I_0, \neg I_1$$

Please note the second function is a non-optimal function and can be simplified using Boolean algebra.

### 3.3.5 Adders

Adders play one of the most important roles in binary arithmetic. In fact fixed-point addition time is often used as a simple measure to express processor's speed. Addition and subtraction circuits can be used as the basis for implementation of multiplication and division. (We are not giving details of this. You can find it in the further readings).

Thus, considerable efforts have been put in the designing of high-speed addition and subtraction circuits. It has been considered to be an important task since the time of Babbage. Number codes are also responsible for adding to the complexity of arithmetical circuits. The 2's complement notation is one of the most widely used codes for fixed-point binary numbers because of the ease of performing addition and subtraction through it.

A combinational circuit, which performs the addition of two bits, is called a half-adder, while the combinational circuit that performs the arithmetical addition of three bits (the third bit is a previous carry bit) is called a full adder.

In half adder the inputs are:

- The augend lets say 'x' bit and addend 'y' bit
- The outputs are Sum 'S' bit and Carry 'C' bit

The logical relationships between these are given by the following truth table.

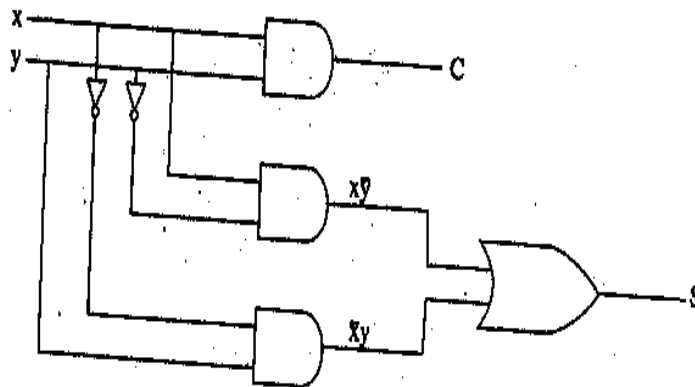
x	y	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

'C' can be obtained on applying AND gate on 'x' and 'y', therefore, while 'S' can be found from following the Karnaugh map.

S \ y	0	1	
x			
C		1	
1	1		

$S = x + y$

Therefore, the logic diagram of half adder is

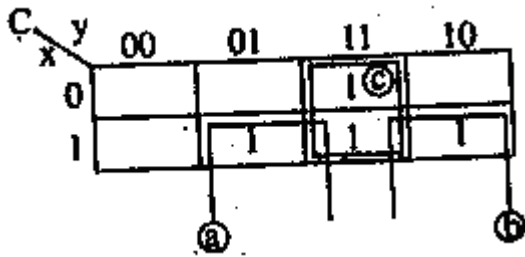


Let us take the full adder. In a full adder another variable carry from previous bit 'P' is used in addition. Thus, the truth for it is:

Input			Output	
x	y	P	C	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

$$C = f(x, y, p)$$

Let us represent it using the Karnaugh map:



Three adjacencies marked (a) (b) and (c)

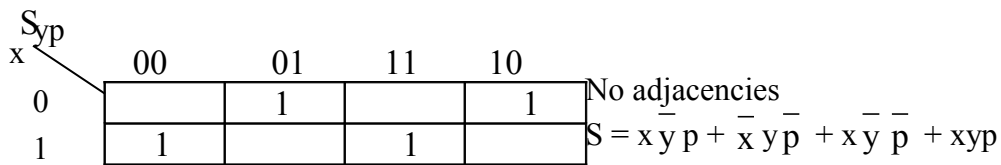
(a)  $x\bar{y}p + xyp$   
 $= xp(y + \bar{y})$   
 $= xp$

(b)  $xyp + xy\bar{p}$   
 $= xy$

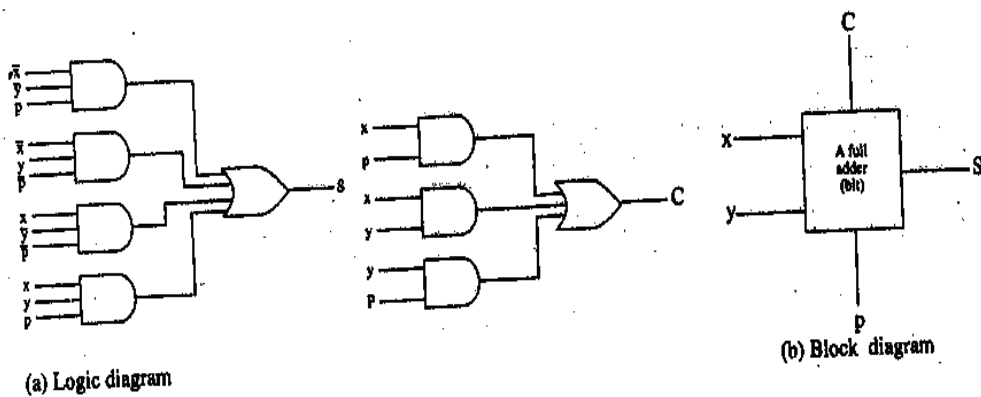
(c)  $\bar{x}yp + xyp$   
 $= yp$

$C = xy + xp + yp$

For finding sum S



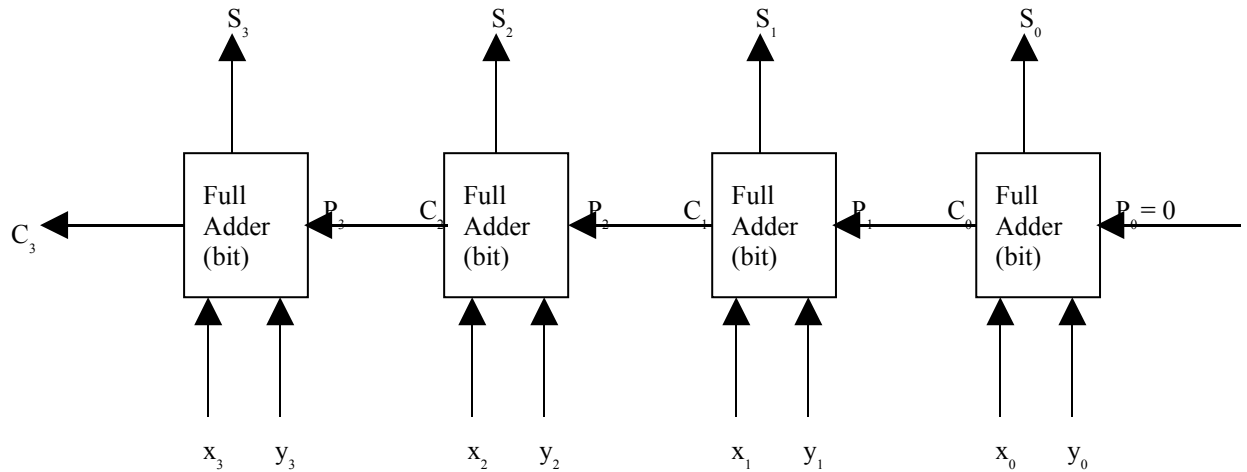
Therefore, the full adder can be represented as:



Till now we have discussed the addition of bits only, but what will happen if we are actually adding two numbers. A number in computer can be 4 byte i.e. 32-bit long or even more. Even for these cases the

basic unit is the full adder. Let us see (for example) how we can construct an adder which adds two 4-bit numbers. Let us assume that the numbers are:  $x_3 x_2 x_1 x_0$  and  $y_3 y_2 y_1 y_0$ , here  $x_i$ , and  $y_i$ , ( $i = 0$  to  $3$ ) represent a bit.

The 4-bit adder is shown below:

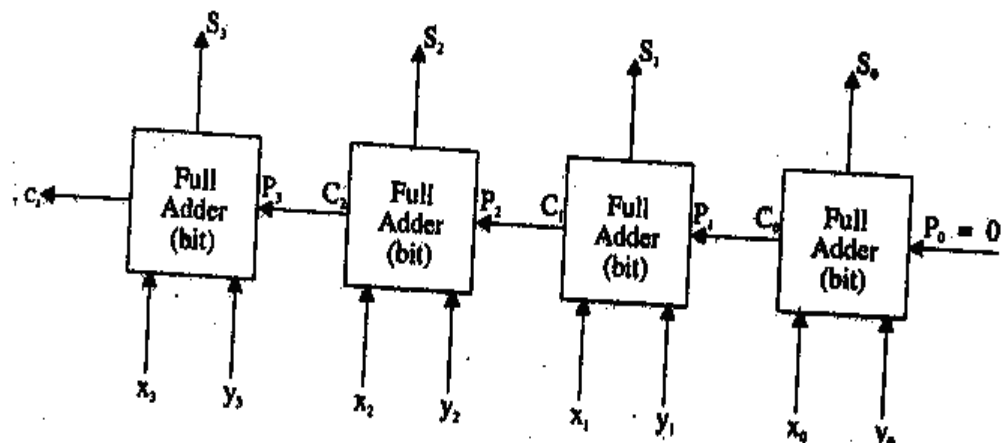


$S_3, S_2, S_1, S_0$  represent the overall sum and overall carry is  $C_3$  from the fourth bit adder. The main feature of this adder is that carry of each lower bit is fed to the next higher bit addition stage, it implies that addition of the next higher bit has to wait for the previous stage addition.

This is called ripple carry adder. The ripple carry becomes time consuming when we are going for addition of say 32 bits. Here the most significant bit i.e., the 32<sup>nd</sup> bit has to wait till the addition of the first 31 bits is complete. Therefore, a high-speed adder, which generates input carry bit of any stage directly from the inputs to previous stages, was developed. These are called carry look-ahead adders. In this adder the carry for various stages can be generated directly by the logic expressions such as:

$$C_0 = x_0 Y_0$$

$$C_1 = x_1 y_1 + (x_1 + y_1) C_0$$



The complexity of the look-ahead carry bit increases as the level of carry increases. But in turn it produces the addition in a very short time. The carry look-ahead becomes increasingly complicated with increasing numbers of bit, therefore, carry look-ahead adders are normally implemented for adding chunks of 4 to 8 bits and then the carry is rippled to next chunk of 4 to 8 bits carry look-ahead circuit.

## SELF-ASSESSMENT EXERCISE 2

1. Draw a Karnaugh map for five variables.
2. Map the function having four variables in a Karnaugh's map. The function is  $F(A,B,C,D) \Sigma(2,6,10,14)$ .
3. Find the optimal logic expression for the above function. Draw the resultant logic diagram.
4. What are the advantages of PLA?
5. Can a full adder be constructed using two half adders?

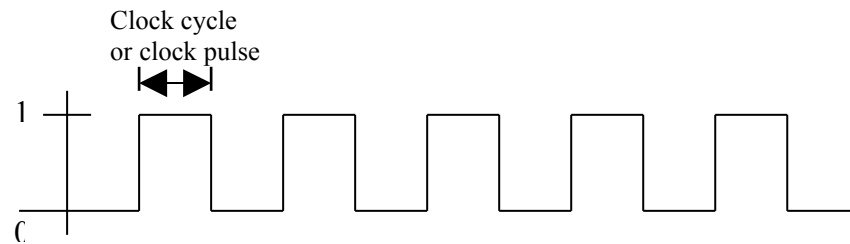
### 3.4 Sequential Circuits

These are logic circuits whose present output depends on the past inputs. These circuits store and remember information. The sequential circuits unlike combinational circuits are time dependent. Normally the current output of a sequential circuit depends on the state of the circuit and on the current input to the circuit. It is a connection of flip-flops and gates. What is a flip-flop? You will find the answer in this section. There can be two types of sequential circuits.

- Synchronous
- Asynchronous

Synchronous circuits use flip-flops and their status can change only at discrete instants. (Don't they seem a good choice for discrete digital devices such as computers?). The asynchronous sequential circuits may be regarded as combinational circuits with feedback path. Since the propagation delays of output to input are small they may tend to become unstable at times.

The synchronous in sequential circuits can be achieved using a clock pulse generator. It synchronises the effect of input over output. It presents signal of the following form:



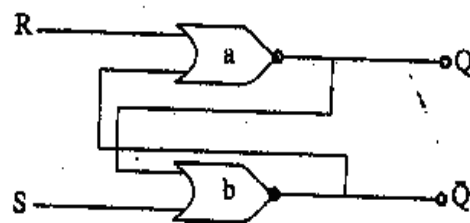
**Figure 23: Clock signal of a clock pulse generator**

The signal produced by a clock pulse generator is in the form of a clock pulse or clock signal. These clock pulses are distributed throughout the computer system for synchronisation.

A clock can have two states: an enable or active state, otherwise a disable or inactive state. Both of these states can be related to zero or one levels of clock signals (it depends on implementation). Normally, the flip-flops change their state only at the active state of the clock pulse. In certain designs the active state of the clock is triggered when the transition (this is sometimes termed edge-triggered transition) from 0 to 1 or 1 to 0 is taking place in a clock signal. A typical CPU is synchronised by a clock signal whose frequency is used as a basic measure of the CPU's speed of operation and hence you might have heard the term "CPU operating at 400 MHz" or so.

### 3.4.1 Flip-Flops

What is a flip-flop? A flip-flop is a binary cell, which can store a bit of information and which in itself is a sequential circuit. But, how does it do it? A flip-flop maintains any one of the two stable states that can be treated as zero or one depending on the presence and absence of output signals. The state of a flip-flop can only change when a clock pulse has arrived. Let us first see the basic flip-flop or a latch that was a revolutionary step in computers. The basic latch presented here is asynchronous. Let us see its logic diagram (Figure 24).



(a) Logic diagram

$I_0$	$I_1$	$O$
0	0	1
0	1	0
1	0	0
1	1	0

(b) Truth table for NOR gate

**Figure 24: A Basic latch (S-R latch using NOR gates)**

A flip-flop can be constructed using two NAND or NOR gates; it contains a feedback loop. The flip-flop in the figure has two inputs R (Reset) and S (set) and two outputs Q and  $\neg Q$ .

In a normal mode of operation both the flip-flop inputs are at zero i.e.  $S = 0$  &  $R = 0$ . This flip-flop can show two states: either the value Q is 1 (therefore  $\neg Q = 0$ ) we say the flip-flop is in **set** state or the value of Q is 0 (therefore  $\neg Q = 1$ ) we call it a **clear** state. Let us see how the S and R input can be used to set and clear the state of the flip-flop. The first question is, why in normal cases S and R are zero? The reason is that this state does not cause any change in state. Suppose the flip-flop was in set state i.e.  $Q = 1$  and  $\neg Q = 0$  and as  $S = 0$  and  $R = 0$ , the output of 'a' NOR gate will be 1 since both its input  $\neg Q$  and R are zero (refer the truth table of 1 NOR gate in Figure 24) and 'b' NOR gate will show output as 0 as one of its input Q is 1. Similarly if flip-flop was in clear state then  $\neg Q = 1$  and  $R = 0$ , therefore, output of 'a' gate will be 0 and 'b' gate 1. Thus, flip-flop maintains a stable state at  $S = 0$  and  $R = 0$ .

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

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

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

What will happen when both S and R go to 1 at the same time? Well, this is the situation which may create a set or clear state depending on which of the S and R stays longer in zero state. But meanwhile both of them are 1 and the value of Q and  $\neg Q$  becomes 1 which implies that both Q and its complement are one, an impossible situation. Therefore, the transition of both S and R to 1 simultaneously is an undesirable condition for this basic latch.

Let us try to construct a synchronous R-S flip-flop from the basic latch. The clock pulse will be used to synchronise the flip-flop. (What is a clock pulse?).

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

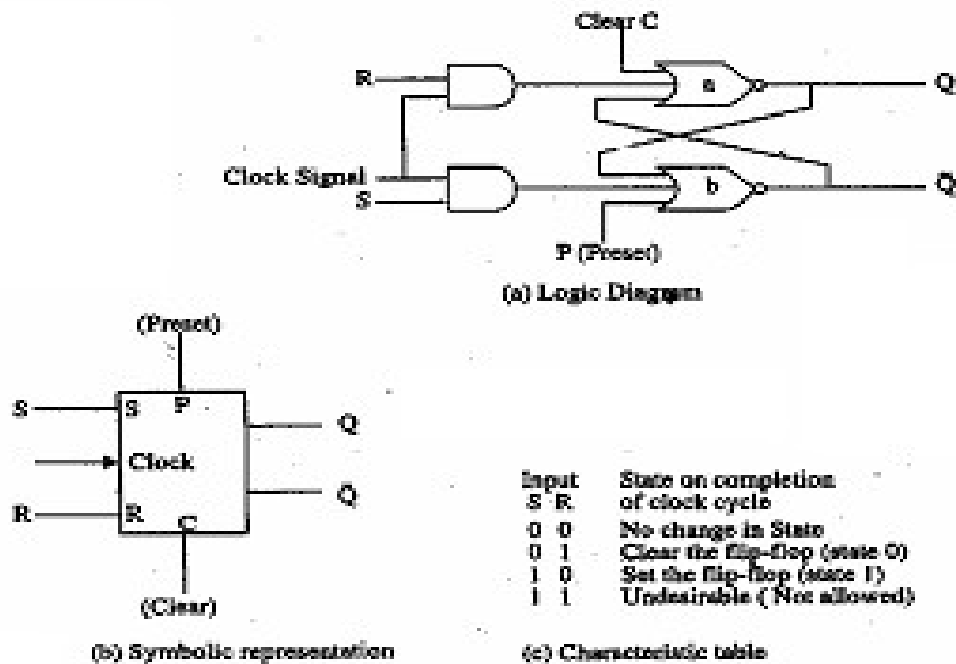


Figure 25: R-S flip-flop

The value  $\neg Q$  can be acquired as an additional output that is in complemented form.

The excitation or characteristic table basically represents the effect of S and R inputs on the state of the flip-flop, irrespective of the current state of the flip-flop. The other two inputs P (preset) and C (clear) are asynchronous inputs and can be used to set the flip-flop or clear the flip-flop respectively at the start of operation, independent of the clock pulse.

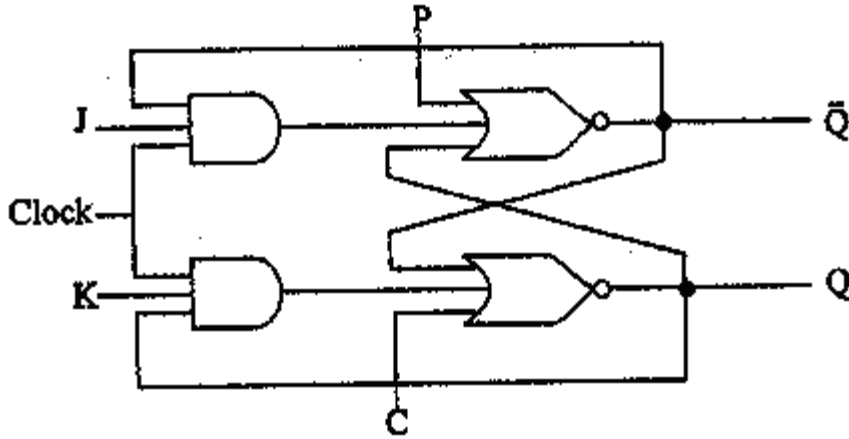
Let us have a look at some more typical and popular flip-flops.

## D Flip-Flop

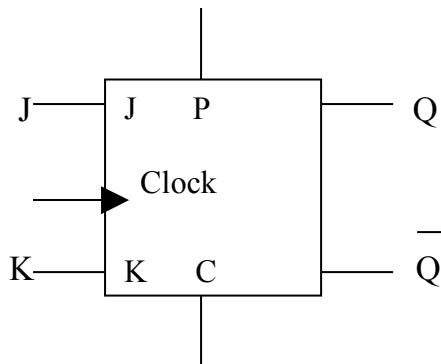
D flip-flop is a special type of flip-flop in the sense that it represents the currently applied input as the state of the flip-flop. Thus, in effect it can store 1 bit of data information and is sometimes referred to as Data flip-flop. Please note that the state of the flip-flop changes for the applied input. It does not have a condition where the state does not change as the



case in RS flip-flop, the state of R-S flip-flop does not change when  $S = 0$  and  $R = 0$ . If we do not want a particular input state to change then either the clock is to be disabled during that period or a feedback of the output can be embedded with the input D. D flip-flop is also referred to as Delay flip-flop because it delays the 0 or 1 applied to its input by a single clock pulse.



(a) Logic diagram

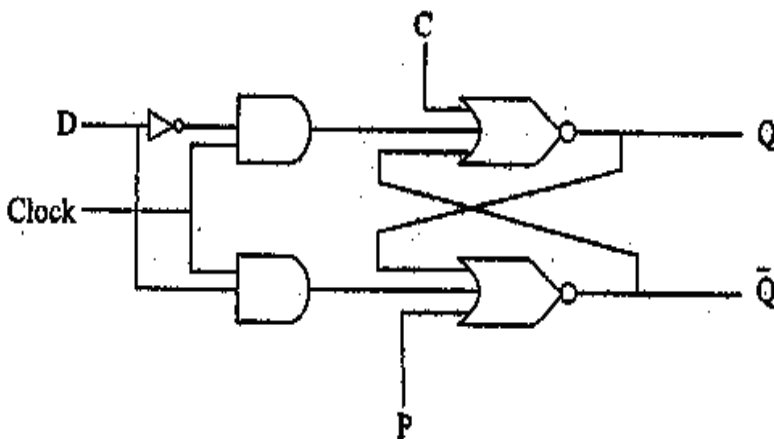


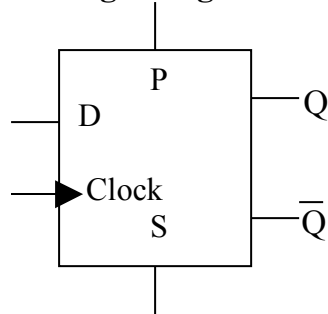
(b) Symbolic representation

Input		State at the completion of clock cycle
J	K	
0	0	No change in State
0	1	Clear the flip-flop (state 0)
1	0	Set the flip-flop (state 1)
1	1	Complement the state of flip-flop

(c) Characteristic table

**J-K flip-flop**



**(a) Logic diagram**

Input D	State after the completion of clock pulse
0	0
1	1

**(b) Symbolic representation**      **(c) Characteristic table**  
**D flip-flop**

**Figure 26: Other flip-flops**

### J K flip-flop

The basic drawback with the R S flip-flop is that one set of input conditions are not utilised and this can be used with a little change in the circuit. In this flip-flop the last combination is used to complement the state of the flip-flop.

After discussing some of the simple sequential circuits, that is flip-flop let us discuss some of the complex sequential circuits, which can be developed using simple gates, and flip-flops.

#### 3.4.1 Registers

A register is a binary function which holds the binary information in digital form. Thus, a register consists of a group of binary storage cells. A register consists of one and more flip-flops depending on the number of bits to be stored in a word. A separate flip-flop is used for storing a bit of a word. In addition to storage, registers are normally coupled with combinational gates enabling certain data processing tasks. Thus, a register in a broad sense consists of the flip-flop that stores binary information and gates, which controls when and how information is transferred to the register.

Normally in a register independent data lines are provided for each flip-flop, enabling the transfer of data to and from all flip-flops to the register simultaneously. This mode of operation is called Parallel Input-Output. Since the stored information in a set of flip-flops is treated as a single entity, common control signals such as clock, preset and clear can be used for all the flip-flops of the register. Registers can be constructed from any type of flip-flop. These flip-flops in integrated circuit registers are usually constructed internally using two separate flip-flop circuits.

The normally used special kind of arrangement is termed the master-slave flip-flop. This type of flip-flop helps in having a stable state at the output. It consists of a master flip-flop and a slave flip-flop.

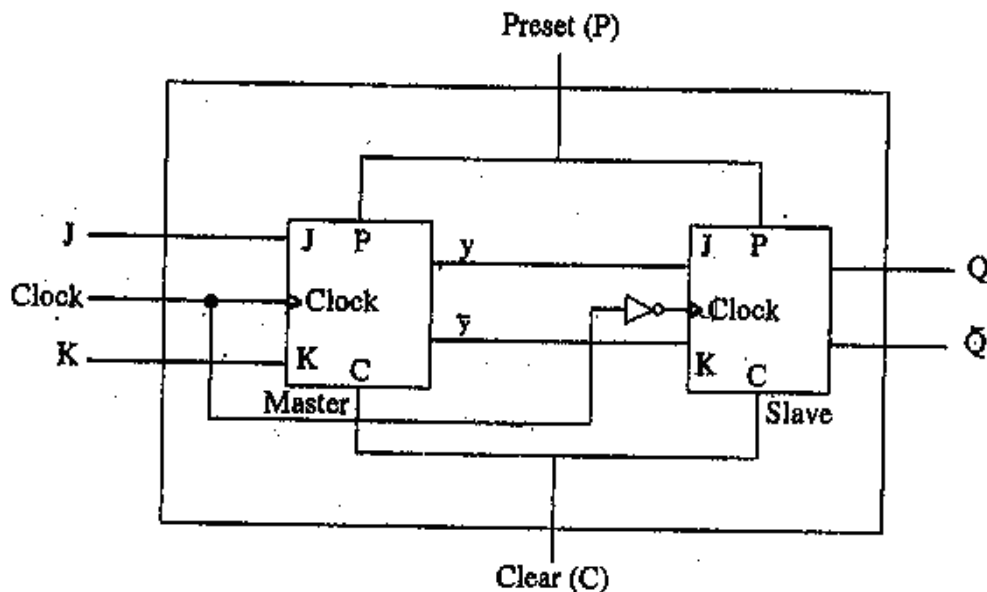


Figure 27: Masters-slave flip-flop using J-K flip-flop

**Note:** You can construct master-slave flip-flop with D flip-flop (Figure 27) or R-S flip-flop in the same manner.

Now, let us analyse this flip-flop.

1. When the clock pulse is 0 the master is disabled but the slave becomes active and its output Q and  $\bar{Q}$  becomes equal to Y and  $\bar{Y}$  respectively. Why? Well, the possible combination of the value of Y and  $\bar{Y}$  are either Y = 1 which means  $\bar{Y} = 0$ ; or Y = 0 which implies  $\bar{Y} = 1$ . Let us see the characteristic table for these two inputs for the J-K flip-flop. The SLAVE flip-flop, thus, can have value either J=1 and K=0 which will set the flip-flop that is Q=1 and  $\bar{Q}=0$ ; or J=0, K=1 which will clear the flip-flop. Therefore, Q is same as Y.
2. When inputs are applied at J and K and the clock pulse becomes 1, only the master gets activated, resulting in intermediate output Y go to state 0 or 1 depending on the input and previous state. Please note that during this time the slave is still maintaining its previous state. As the clock pulse become 0, the master becomes inactive and the slave acquires the same state as the master.

But why do we acquire this master-slave combination? There is a major reason for this master-slave form. Consider a situation where the output of a flip-flop is going to input of other flip-flops. Here, the assumption is

that the clock pulse inputs of all flip-flops are synchronised and occur at the same time. The change of state of the master occurs when the clock pulse goes to 1, but during that time the output of the slave still has not changed, thus the state of the flip-flops in the system can be changed simultaneously during the same clock pulse even though outputs of flip-flops are connected to the inputs of flip-flops. In other words there are no restrictions on feedback from the register's outputs to its inputs.

Let us come back to the register having parallel input-output. Figure 28 shows a 4-bit register with parallel input-outputs.

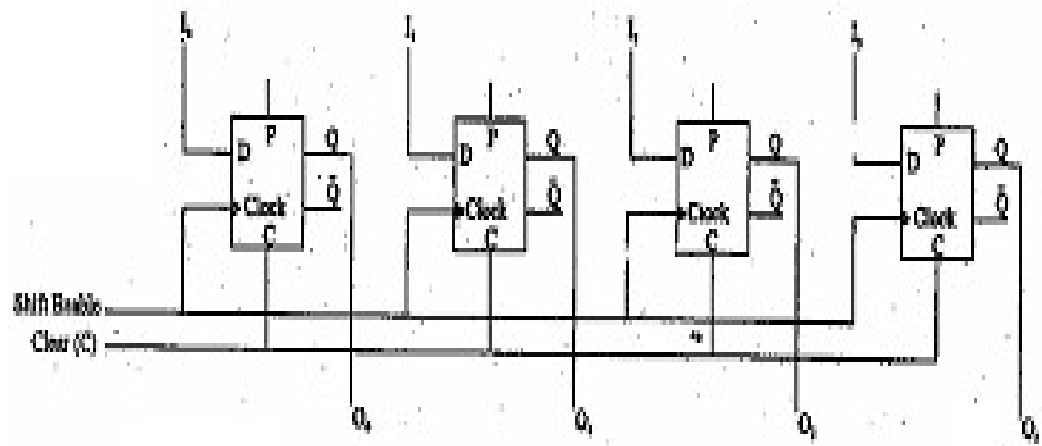


Figure 28: A register with a parallel input-output

It is one of the simplest registers and contains preset, clear (for clearing the register completely) and clock pulse input in addition to the data value through D input. All the four bits in this register can be input and output simultaneously (as master-slave configuration can be employed in D flip-flop) hence the name parallel input-output.

Another kind of register that is used for shifting the data to the left is called a shift register. A shift register may operate in serial input-output mode in which data is entered in the register one bit at a time from one end of the register and can be read from the other end as one bit at a time. Figure 29 gives a 4-bit right-shift register with serial input-output. This register is constructed using D flip-flops.

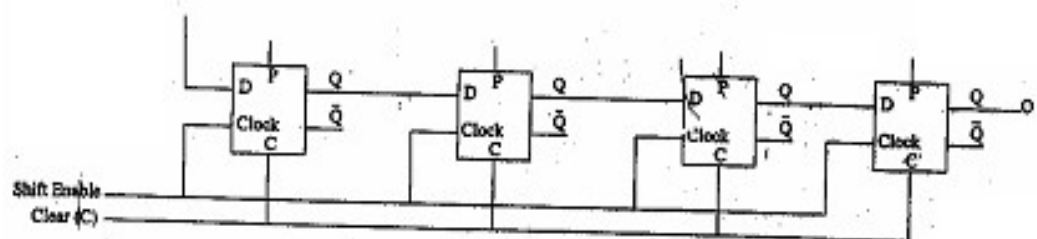


Figure 29: The right shift register with serial input-outputs

Please note that in this register shift enable is used instead of the clock pulse, since we do not necessarily want the register to perform shift on each clock pulse. Clear and preset inputs are also used here. The data is shifted in this register bit by bit. For example: 1 0 1 1 will be loaded in the register as:

Input	Flip-Flop States				
	1	2	3	4	
1	1	x	x	x	On the occurrence of the next shift enable which may be the next clock pulse
1	1	1	x	x	
0	0	1	1	x	
1	1	0	1	1	

It can be recovered in a similar fashion from the registers

Flip-Flop States				
1	2	3	4	Output
0	1	0	1	1
0	0	1	0	1
0	0	1	0	0
0	0	0	0	1

Shift enable controls the shifting of the data to the right here.

A shift register can be constructed for bi-directional shift with parallel input-output or serial input-output. A general structure may have the facility of parallel data transfer to or from the register. In addition, the facility of left or right shift may also be provided. But this structure will require additional control lines for indicating whether a parallel or serial output is desired; and whether the left shift or right shift operation is desired. A general symbolic diagram for such a register is given in Figure 30.

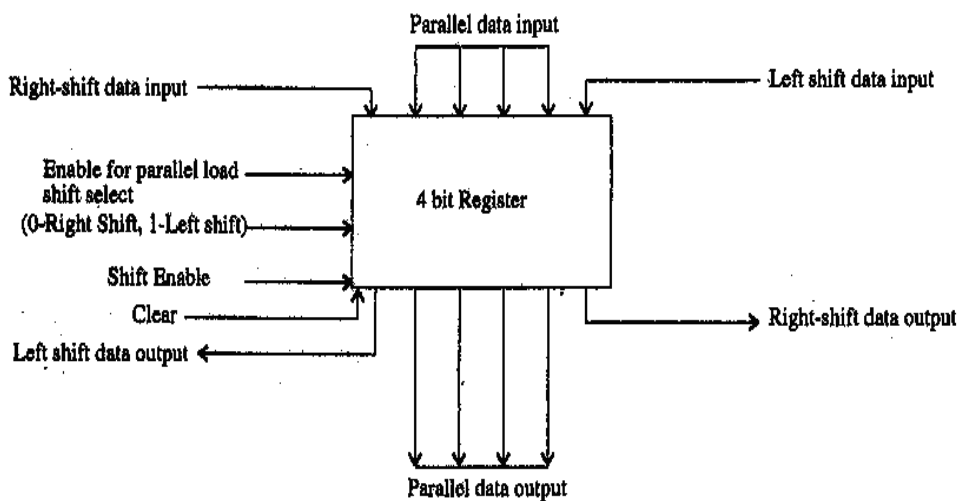


Figure 30: A four bit-right shift register with parallel load

The basic diagram of such a circuit resembles figure 28 but requires additional combinational circuit before an input is fed. You can refer to further readings for further reference.

These shift registers can be used in a number of applications such as storage of serial data, parallel to serial and serial to parallel data conversion and multiplication & division operations. For example, performing a left-shift for an unsigned binary number is equivalent to multiplication by binary two. Similarly, a right-shift operation on an unsigned binary number is equivalent to the division by two. For example, if we multiply:

Decimal	5	Binary	0101	Shift left and place a zero on
	$\begin{array}{r} 2 * \\ \hline 10 \end{array}$	shifted out	$\begin{array}{r} 0010 * \\ \hline 01010 \end{array}$	← shift in

### 3.4.2 Counters

Another useful sequential circuit is the counter. A counter in principle is a register whose value is incremented by one on the occurrence of some event. When the value stored in the counter reaches the maximum value it can store, the next incremented value becomes zero. The counters are used for counting the number of times an event occurs and are useful for generating the timing signals for controlling the sequence of operations in digital computers. A counter may also be used in the generation of counted timing signals of the clock pulse. It can be compared to the clock you use at home, where, basic clock signals may be generated from the quartz crystal; however, the clock shows a counted sequence of hours, minutes and seconds.

There are two types of counters- asynchronous and synchronous. This classification is based on the way they operate. In the asynchronous counter the state of one flip-flop changes at a time while in the synchronous counter the state of all the flip-flops can be changed at the same time.

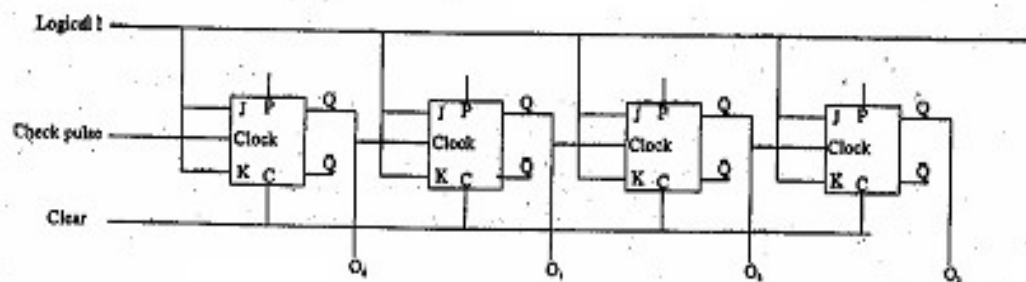


Figure 31: A 4-bit ripple counter

**The Asynchronous Counter:** This is also termed a ripple counter, since the change that occurs in order to increment the counter ripples through the counter from one end to the other. Figure 31 shows an implementation of a 4-bit counter implemented using J-K flip-flops. This counter is incremented on the occurrence of each clock pulse and it counts from 0 to 15.

The input line to J and K is kept high, i.e. logical one in this counter and each time a clock pulse occurs the value of the flip-flop is complemented (Refer to the characteristic table of J-K flip-flop). Please note the clock inputs to the flip-flops. The first flip-flop is fed with the clock pulse as clock input but the second, third and fourth flip-flops are provided with the output of its previous flip-flop as clock signals, implying that these flip-flops will be complemented if the previous flip-flop has a value 1. Thus, the effect of the complement will ripple through these flip-flops.

**The Synchronous Counter:** Take the instance when the state of the ripple counter is 0111 now the next state will be 1000 that means a change in the state of all the flip-flops, but will it occur simultaneously in the ripple counter? No, the first leftmost flip-flop will change state from 1 to 0, this will cause the next flip-flop to change state and so on till the last flip-flop changes state. Thus, a delay in changing the state is proportional to the length of the counter. Therefore, to avoid this delay, normally synchronous counters are used, in which all the flip-flops change state at the same time (see Figure 32).

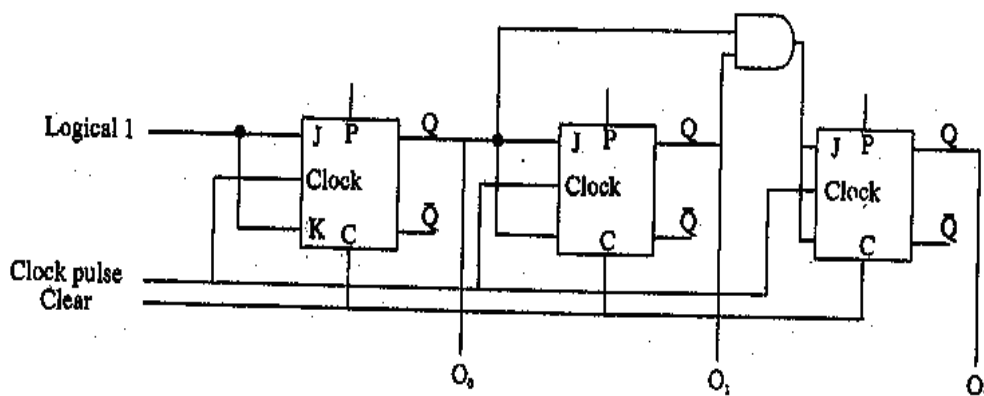


Figure 32: The logic diagram of a 3-bit synchronous counter

In a synchronous counter:

1. The first flip-flop is always complemented.
2. The second flip-flop is complemented in the next clock pulse if the current state of the first flip-flop is set (one).

3. The third flip-flop is fed by an AND gate which is connected with the output of the first and second flip-flops, thus the third flip-flop will be complemented only if the first AND second flip-flops are currently in 1 state. This will be more evident from the following truth table:

$O_0$	$O_1$	$O_2$
0	0	0
1	0	0
0	1	0
1	1	0
0	0	1
1	1	1
0	1	1
1	1	1
0	0	0
1		

4. The master-slave flip-flops are used here and, therefore all the flip-flops can change state simultaneously.

### 3.5 Interconnection Structures

A computer consists of three basic components:

- CPU
- Memory and
- Input/output components

If data is to be exchanged among these three components, we need to have an interconnection structure, which allows transfer of information among all these. This interconnecting structure must support

- The transfer of an instruction or a unit of data from memory to CPU.
- The transfer of a unit of data from CPU to memory
- Reading of data from input/output device i.e., transfers from input/output device to CPU.
- CPU sending data to input/output device.
- The transfer of input/output devices to memory
- The transfer of memory to input/output devices.

To support these transfers several interconnecting structures have been tried by various computer developers. All these structures can be broadly classified as four basic architectures.



1. **Input/Output to Central Processor:** In this structure all exchanges between input/output devices and memory pass through the CPU. It is a simple and inexpensive structure but reduces the CPU throughput as the CPU is tied up in doing input-output tasks, which are normally considerably slower than that of CPU operations.
2. **Input/Output to Memory:** This architecture involves direct transfer between input/output and memory without involving much time from CPU. The input/output here can be performed simultaneously when the CPU is doing other computations but this structure has a complex and flexible control mechanism.
3. **Input/Output to Central Switch:** A central switch is provided here which controls the access of the memory by CPU and input/output device. Here, the CPU is freed up to do other computations. It also provides control of input/output operations through the CPU directly. Therefore, it is a powerful flexible approach and is popular in large mainframes line. The major drawback of this approach is the complexity of the switch.
4. **Input/Output to Bus:** It is a flexible and simple structure and used commonly in micro and mini-computers. Let us discuss this in greater details.

### Bus Interconnection

A bus is a set of connections between two or more components/devices that are designed to transfer several/all bits of a word from a specific source to destination. It is a shared media of information transfer. A bus consists of multiple communication wires that are also termed lines. A line is capable of transferring one bit only. Thus, for transferring a word of 16 bits simultaneously over a bus we need to have 16 bus lines. In addition, some other lines are needed for controlling this transfer.

A bus may be unidirectional (capable of transmitting data in one direction) or bi-directional. In a shared bus only one source can transmit at a time while one (or more than one) receives that signal. Figure 33 shows the diagram of a shared bus. The shared bus is the one which we will discuss further.

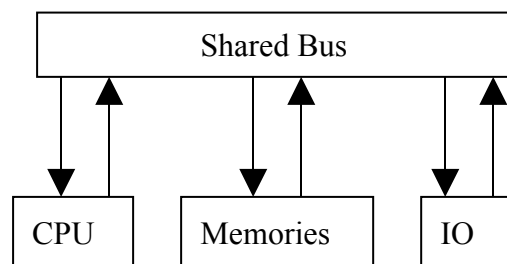


Figure 33: A shared bus

A computer system contains a number of buses, which provide pathways among several devices. A shared bus that connects the CPU, memory and input/output is called a system bus.

A system bus may consist of 50 to 100 separate lines. These lines can be broadly categorised into three functional groups:

- Data lines: these are collectively called Data Bus
- Address lines: these are collectively called Address Bus
- Control lines: these are collectively called Control Bus.

The data bus provides a path for moving data between the system modules. It normally consists of 8, 16 or 32 bit separate lines. The number of lines in data bus is called the width of data bus. Thus, a data bus width limits the maximum number of bits that can be transferred simultaneously between two modules e.g., CPU and memory. The width of the data bus also helps in determining the overall performance of a computer system.

For example, if the width of a data bus is 8 bits and an instruction is of 16 bits then for fetching each instruction two memory accesses will be needed by the CPU.

The address bus is used to designate the source of data for the data bus. As the memory may be divided into a linear array of bytes or words, therefore, for reading or writing any information on to memory, CPU needs to specify the address of a particular location. The address bus supplies this address. Thus, the width of the address bus specifies the maximum possible memory supported by a system.

For example, if a system has a 16 bit wide address bus then it can have a main memory size equivalent to  $2^{16} = 64K$ . The role of control lines (bus) is to control the access to data and address bus, as bus is a shared medium. The control lines are used for transmission of commands and timing signals (which validate data and address) between the system modules.

Some of the control lines of bus are required for bus request, bus grant, providing clock signals, providing reset signals, reading/writing to I/O devices or memory, etc.

The bus is used in the following ways:

1. Request for sending data
  - First the sender grabs the permit to use the bus.

- And then transfers the information
2. Request for receiving data from other module.
    - Grab the bus
    - Transfer the request to other module to send the data
    - Then wait for the other module to send the data

Physically, a bus is a number of parallel electrical conductors. These circuits are normally imprinted on printed circuit boards. The bus normally extends across most of the system components that can be trapped into the bus lines. You can see these wires in the printed circuit board of your personal computer. Let us now discuss some of the aspects related to the bus.

**Dedicated or Multiplexed Buses:** A dedicated bus line, as the name suggests, is assigned permanently to a function or to a physical subset of the components of the computer. The example of a functional dedicated bus is the dedicated address bus and data bus. As far as physical dedication is concerned a bus is dedicated to only a subset of modules. For example, an input/output bus can be used to connect all the input/output modules. The physical dedication increases the **throughput** of the bus as only few modules are in contention but it increases the overall size and cost of a system.

In certain computer buses some or all the address lines are used for data transfer operations, that is, the same lines are used for the address as well as data lines, but of course, at different times. This is known as time multiplexing and is a good example of a multiplexed bus. This multiplexing reduces the total number of lines to be used, which in turn results in reduction of cost and space. But the disadvantages of multiplexing are the complex circuitry and potential reduction in performance, as the multiplexed lines cannot be used in parallel.

**Synchronous or Asynchronous Timing:** Another important aspect of buses is the manner in which the data transfer is timed. In synchronous buses the data is transferred during a specific time that is known to source and destination. Synchronisation is normally achieved by using the clock pulse of the same clock source for both source and destination or different clocks of the same frequency. Normally, the synchronising source and destination keep a periodic communication in order to keep step with each other. The synchronous buses are easy to implement and test but restrict information transfer rate to that of the slowest device. The alternative approach to the synchronous bus is the asynchronous bus. In this approach each item that is to be transferred has a separate

control signal. This signal indicates the presence of the item on the bus to the destination. With the asynchronous buses a variety of fast and slow devices can share a bus. However, the bus-control circuitry is more complex and expensive in the case of asynchronous buses.

**Bus Arbitration:** Another important aspect of the system where buses are used is the control of a bus. In most of the systems more than one module need control of the bus. For example, input/output modules may need the control of the bus for transferring data to memory. Similarly, the CPU also needs the control of the bus for data transfer. Suppose both of these devices want to transfer information at the same time then? There should be a method for resolving the simultaneous data transfer requests on the bus. This process of selecting one of the units from various buses requesting units is called bus arbitration. Two broad categories of arbitration have been suggested. These are centralised and distributed.

In the centralised scheme a hardware circuit device that is referred to as the bus controller or bus arbiter processes the request to use the bus. The bus controller may be a separate module or can be constructed as the part of the CPU.

On the contrary the distributed scheme has shared access control logic among the various modules. These modules work together to share the bus. Irrespective of the scheme, the main role of the arbitration is to designate one device as master (which controls the bus during the time it is designated as master) and a second device as slave (which takes all the orders from the master). Let us discuss some of the arbitration schemes among various contending masters. Please note that all these arbitration schemes differ in the number of control lines needed and in the speed of response to various bus-access requests.

**Daisy Chaining:** In daisy chaining the control of the bus is granted to any module by a bus grant signal, which is chained through all the contending masters. Refer to Figure 34 to note how the bus grant signal is distributed among various modules. The other two control signals are bus request and bus busy.

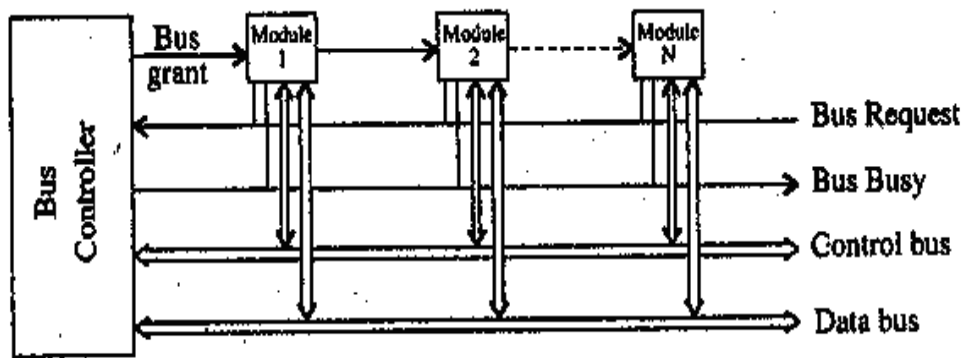


Figure 34: Daisy chaining arbitration

The bus request line, if activated, only indicates that one or more modules require the bus. The bus controller responds to the bus request only if the bus busy line is inactive, that is, the bus is free. The bus controller responds to the bus request signal by placing the signal on the bus grant line. The bus grant signal passes through the modules one by one. On receiving the bus grant the module that was requesting for bus access, blocks further propagation of bus grant signal and issue a bus **busy** signal and starts using the bus. If the bus grant is passed through a module which had not issued the bus request, then the bus grant signal is forwarded to the next module.

In this scheme the priority is wired in and cannot be changed by programs. In Figure 34 the assumed priority is (highest to lowest) Module 1, Module 2...Module N. If two modules, let us say 1 and N request the bus at the same time then the bus will be granted to Module 1 first as the signal has to pass through Module 1 to reach Module N. The basic drawback of this simple scheme is that if the bus request of Module 1 is occurring at a high rate then rest of the modules may not get the bus for quite some time. Another problem can occur when say the bus grant line between say Module 4 and Module 5 fails, or Module 4 is unable to pass the bus grant signal. In any of the above mentioned cases no bus access will be possible beyond Module 4.

**Polling:** Another method that is commonly used for bus arbitration is polling. In polling instead of single bus grant lines, as the case of daisy chaining, we encounter poll count lines. These lines are connected to all the modules connected on the bus (refer Figure 35). The bus request and bus busy are the other two control lines for bus control. A request to use the bus is made on the bus request line while the bus request will not be responded to till the bus busy line is active. The bus controller responds to a signal on bus request line by generating a sequence of numbers on poll count lines. These numbers are normally considered to be a unique address assigned to the connected modules. When the poll count

matches the address of a particular module that is requesting for the bus, the module activates the bus busy signal and starts using the bus. The polling basically is asking each module one by one whether it has something to do with the bus. The polling has two main advantages over daisy chaining. The first advantage is that in polling the priority of contending modules can be altered (if desired) by changing the sequence of the generation of numbers on the poll count lines. The second advantage of polling over daisy chaining is that the failure of one module will not affect any other module as far as bus grant is concerned. But it has certain inherent disadvantages also in comparison to daisy chaining. Polling requires more control lines which adds in cost and the maximum number of modules which can share the bus in polling is restricted by the number of poll count lines. For example, in the figure 35 we have three poll count lines, which implies that at most  $2^3 = 8$  modules can share this bus.

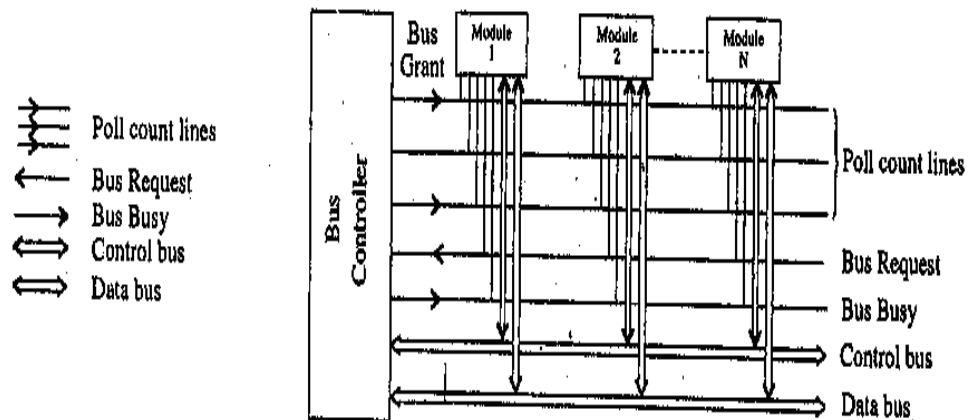


Figure 35: Polling

**Independent Requesting:** In this arbitration scheme, each module has its independent bus request and bus grant line. In this scheme the identification of the requesting unit is almost immediate and requests can be responded to quickly. Priority in such a system can be through the bus controller and can be changed through a program (if desired).

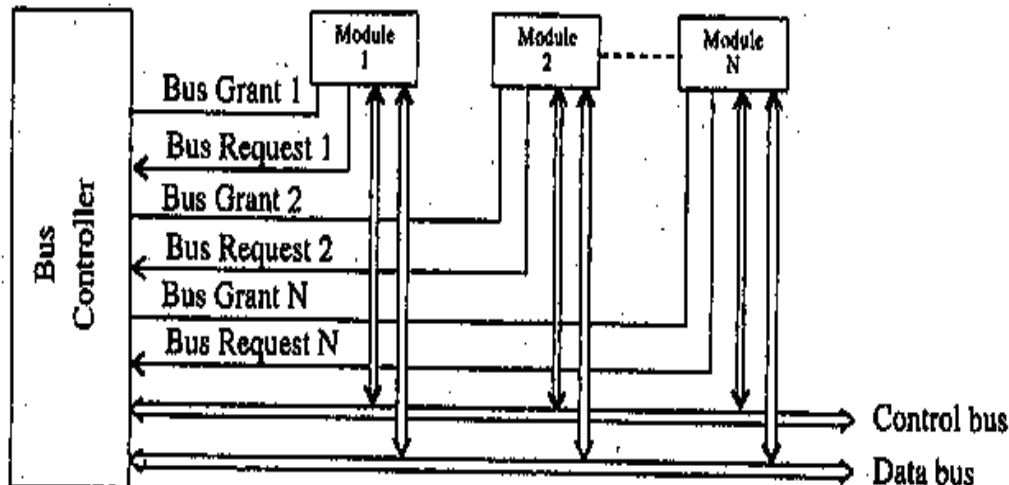


Figure 36: Independent requesting

In certain systems a combination of these arbitration schemes may be used.

**Types of Data Transfer:** Let us discuss the various types of data transfer on a bus. The buses support read and write transfer. A read data transfer is basically a transfer of information from slave to master while the write data transfer is from master to slave. The data transfer is also dependent on whether dedicated or multiplexed lines are used for address and data. In a write operation the data follows the address of slave (in multiplexed data and address lines) or can be transferred almost at the same time the address is transferred (in dedicated lines), while for a read operation the access time of data at the slave causes slight delay. In addition, the delay may be caused if a bus request is needed for grabbing the bus during read or the write operation.

Another important data transfer type is the read-modify-write operation. In this operation the read operation is followed immediately by a write operation for the same address. The address in such cases is broadcast only at the beginning of this operation. During this read-modify-write operation no other bus master can access the data element. These types of operation are used for shared memory which can be used by many programs at the same time.

In a read-after-write transfer the write is followed immediately by a read operation for the same address. This operation can be used for checking whether the correct data is written or not.

Finally, some buses allow transfer of a block of data. In such cases the address is followed by several data cycles. The address specifies the address of the location from which the first data item is to be transferred.

The subsequent data items are transferred from/to the subsequent locations.

### **The Local Bus**

With the development of graphical user interfaces the information that is to be transferred between the processor, memory, display, and secondary storage has increased tremendously. For example, the monochrome text is about 4 KB of information; compare it to a 256 colour. Windows screen requires about 300 KB. Since in microcomputers, the displayed information is also a part of memory-based information, thus, more I/O bandwidth is needed to handle large amounts of data transfer to and from the display mechanism and the increasingly larger and faster hard disks. Increasing the speed of the processor in such situations will not be used properly as the processor will always have to wait for the system bus to transmit data. This gave birth to the concept of the local bus.

One such popular local bus on modern computers is the *Peripheral Component Interconnect or PCI bus*.

The *Peripheral Component Interconnect (PCI)* bus was developed by Intel in 1993.

PCI is a 32-bit bus that normally runs at 33 MHz or 66 MHz etc.

## **4.0 CONCLUSION**

In this unit, we had tried to answer the basic query “How does a computer actually perform computation” which had led us into discussing Boolean algebra, computer logic gates, truth table, combinational circuits and the simplification of these circuits using Boolean algebra and Karnaugh maps.

Also various types of circuits and their applications in a computer system and how a basic mathematical operation such as addition is done by the computer have been extensively discussed in this unit.

## **5.0 SUMMARY**

This unit provides you information about the basis of a computer system. Some more details on these basics will be discussed later. But the key element for the design is the combinational and sequential circuit. With the advent of PLA's the designing of circuits is changing and now the scenario is moving towards microprocessors.



With this developing scenario in the forefront and the expectations of Ultra Large Scale Integration (ULSI) in view, it is not far off when the design of logical circuits will be confined to single microchip components. The bus structure discussed in this unit will be widely used throughout this module. We have not given any example of this structure, but if you need more details on it you can refer to further readings.

## 6.0 TUTOR- MARKED ASSIGNMENT

1. What are sequential circuits? How are they different from combinational circuits?
2. What are the advantages of the master-slave flip-flop?
3. Can a ripple counter be constructed from a shift register?
4. The system bus is used to connect the CPU to a central switch  
True  False
5. A bus can be used to transfer only data True  False
6. The same bus lines may be used to transfer data and addresses  
True  False
7. In the Daisy chaining arbitration scheme the priority of the connected modules can be changed. True  False
8. Microcomputers do not need a system bus True  False

## 7.0 REFERENCES/FURTHER READINGS

- Mano, M. Morris (1993). *Computer System Architecture* (4<sup>th</sup> ed). Prentice Hall of India.
- Hayes, John, P.(1988). *Computer Architecture and Organisation* (2<sup>nd</sup> ed). McGraw-Hill International.
- Stallings, William. *Computer Organisation and Architecture* (3<sup>rd</sup> ed). Maxwell Macmillan International Editions.
- Baron, Robert J. and Higbie, Lee. *Computer Architecture*. Addison-Wesley Publishing Company.

## UNIT 3 MEMORY ORGANISATION

### CONTENTS

- 1.0 Introduction
- 2.0 Objectives
- 3.0 Main Content
  - 3.1 Memory System
  - 3.2 Characteristics Terms for Various Memory Devices
  - 3.3 Random Access Memory
    - 3.3.1 Ferrite Core Memories
    - 3.3.2 Semiconductor Memories
    - 3.3.3 Read Only Memories
    - 3.3.4 Chip Organisation
  - 3.4 External/Auxiliary Memory
    - 3.4.1 Magnetic Disk
    - 3.4.2 Magnetic Tapes
    - 3.4.3 Charge-Coupled Devices (CCDs)
    - 3.4.4 Magnetic Bubble Memories
  - 3.5 High Speed Memories
    - 3.5.1 Interleaved Memories
    - 3.5.2 Cache Memory
    - 3.5.3 Associative Memories
- 4.0 Conclusion
- 5.0 Summary
- 6.0 Tutor- Marked Assignment
- 7.0 References/Further Readings

### 1.0 INTRODUCTION

We have already discussed the basic structure of the computer. We started our discussion, in the previous unit with the digit logic circuits, the basic building block of modern computers and then moved on to discuss one of the most important interconnection structures-- the Bus structure of the computer. Now, we will discuss the other basic components of a computer, to make the picture complete. We can start with the CPU's structure and go on to discuss the memory, input/output etc. However, for the sake of simplicity we will start with memory organisation and then move on to input/output organisation (Unit 4) and then we will devote a full module on CPU organisation (Module 2).

In this unit we will examine the main memory, cache memory, magnetic secondary memory and the optical memory. With the advancement of technologies the optical memories are becoming increasingly popular and therefore, will be discussed in sufficient detail in the unit.

## 2.0 OBJECTIVES

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

- describe the key characteristics of the memory system;
- distinguish among various types of random access memories;
- differentiate among various external memories;
- describe the latest secondary storage technologies and their data storage format; and
- describe the importance of cache memory and other high-speed memories.

## 3.0 MAIN CONTENT

### 3.1 The Memory System

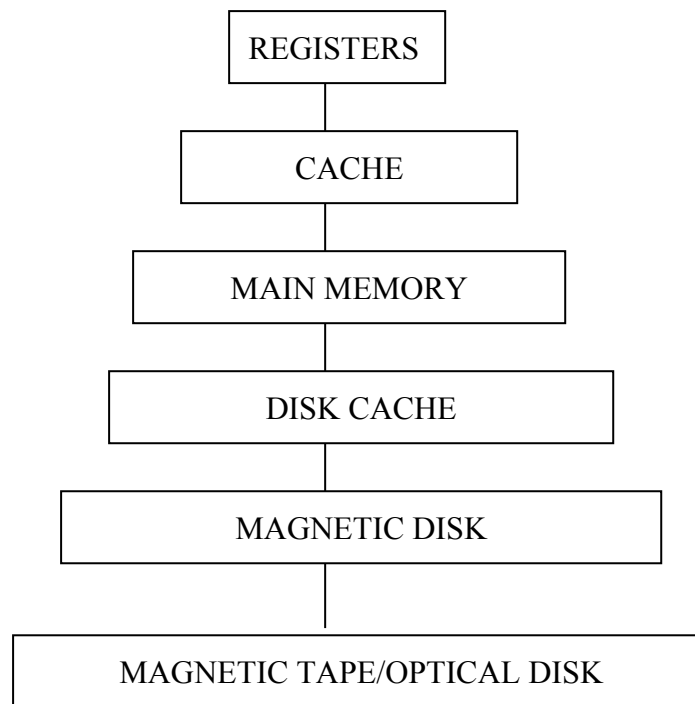
The memory in a computer system is required for storage and subsequent retrieval of the instructions and data. A computer system uses a variety of devices for storing these instructions and data that are required for its operations. Normally we classify the information to be stored on computer in two basic categories: data and instructions.

The storage devices along with the algorithm or information on how to control and manage these storage devices constitute the memory system of a computer.

A memory system is a very simple system yet it exhibits a wide range of technology and types. The basic objective of a computer system is to increase the speed of computation. Likewise the basic objective of a memory system is to provide fast, uninterrupted access by the processor to the memory such that the processor can operate at the speed at which it is expected to work.

But does this kind of technology where there is no speed gap between the processor and the memory speed exist? Yes, it does, but unfortunately as the access time (time taken by CPU to access a location in memory) becomes less and less the cost per bit of memory becomes increasingly higher. In addition, normally these memories require power supply till the information needs to be stored. These things are not very convenient, but on the other hand the memories with smaller cost have very high access time that will result in slower operations of the CPU. Thus, the cost versus access time anomaly has led to a hierarchy of memory where we supplement fast memories with large, cheaper, slower memories. These memory units may have very different physical and operational characteristics, therefore, making the memory system

very diverse in type, cost, organisation, technology and performance. This memory hierarchy will work only if the frequency of access to the slower memories is significantly less than the faster memories.



**Figure 37: The memory hierarchy**

You should be familiar with all the terms given in the above diagram, except the term, “disk cache”. The disk cache may be a portion of RAM, sometimes called the soft disk cache that is used to speed up the access time on a disk. In some newer technologies such a memory can be a part of a disk drive itself; such a memory is sometimes called the hard disk cache or buffer. These hard disk caches are more effective, but they are expensive, and therefore smaller. Almost all modern disk drives include a small amount of internal cache.

### 3.2 Characteristics Terms for Various Memory Devices

The following terms are most commonly used for identifying the comparative behaviour of various memory devices and technologies.

**Storage Capacity:** It is representative of the size of the memory. The capacity of internal memory and main memory can be expressed in terms of number of words or bytes. The storage capacity of the external memory is normally measured in terms of bytes.

**Unit of Transfer:** A unit of transfer is the number of bits read in or out of the memory in a single read or write operation. For main memory and

internal memory, the normal unit of transfer of information is equal to the word length of a processor. In fact it depends on the number of data lines in and out of the memory module. (Why?) In general, these lines are kept equal to the word size of the processor. What is a word? You have already learnt about this term in Unit 1 of this module. The unit of transfer of the external memory is normally quite large (Why? You will find the answer to this question later in this unit) and is referred to as the block of data.

**Access Modes:** Once we have defined the unit of transfer the next important characteristic is the access mode in which the information is accessed from the memory. A memory is considered to consist of various memory locations. The information from memory devices can be accessed in the following ways:

- Random Access;
- Sequential Access;
- Direct Access.

**Access Time:** The access time is the time required between the requests made for a read or write operation till the time the data is made available or written at the requested location. Normally it is measured for a read operation. The access time depends on the physical characteristics and access mode used for that device.

**Permanence of Storage:** Is it possible to lose information stored by the memory over a period of time? If yes, then what can be the reasons for the loss of information and what should be done to avoid it?

There are several reasons for information destruction; these are destructive readout, dynamic storage, volatility, and hardware failure. You are familiar with all these terms.

There can be some memories where the stored 1 loses its strength to become 0 over a period of time. These kinds of memories require refreshing. The memories, which require refreshing, are termed dynamic memories. In contrast, the memories, which do not require refreshing, are called static memories.

### **DRAM (Dynamic RAM)**

DRAM technologies are mainly used as main memories. Presently DRAMs are available in many different forms. The basic anomaly relating to DRAMs is the cost versus speed. A faster processor needs faster memories. However, it is important to have MORE main memory rather than BETTER system memory. The performance considerations

are normally associated with the performance of cache rather than the main memory.

Please note that DRAM at core is RAM. The basic difference among various acronyms of DRAM technologies are primarily because of connection of modules, configuration and addressing, or any special enhancement such as building a small portion of SRAM (Cache) in the DRAM module. A simple organisation of a DRAM chip is given in Figure 38(c).

SRAMs are mainly used as cache memories. These cache memories are discussed in more details later in this unit.

**Cycle Time:** It is the minimum time lapse between two consecutive read requests. Is it equal to access time? Yes, for most of the memories except the ones in which destructive readout is encountered or a refreshing cycle is needed prior to next read. Cycle time for such memories is the access time (time elapsed when a read request is made available) plus writing time as after the data has been made available, the information has to be written back in the same location as the previous value has been destroyed by reading.

**Data Transfer Rate:** The amount of information that can be transferred in or on the memory in a second is termed the data transfer rate or bandwidth. It is measured in bits per second. The maximum number of bits that can be transferred in a second depends on how many bits can be transferred in or out of the memory simultaneously and thus the data bus width becomes one of the controlling factors.

**Physical Characteristics:** In this respect the memory devices can be categorised into four main categories viz: electronic, magnetic, mechanical and optical. One of the requirements for a memory device is that it should exhibit two well-defined physical states, such that 0 and 1 can be represented in those two states. The data transfer rate of the memory depends on how quickly the state can be recognised and altered. The following table lists some of the memory technologies along with their physical and other important characteristics.

**Table 3** Characteristics of some memory technologies

Technology	Access time (in Sec)	Access mode	Permanence of storage	Physical nature of storage medium	Average cost (Rs/bit) (Approx.)
Semiconductor memories	$10^{-8}$	Random	Volatile	Electronic	$10^{-2}$
Magnetic disk	$10^{-2}$	Direct	Non-volatile	Magnetic	$10^{-5}$
Magnetic tape	$10^{-1}$	Sequential	Non-volatile	Magnetic	$10^{-6}$
Compact disk ROM	1	Direct	Non-volatile	Optical	$10^{-7}$

Some of the main attributes that are considered for the storage devices are physical size, energy consumption and reliability.

The physical size also depends on the storage density of the memories. This is also coupled with the question of portability of memory. The energy consumption is another key factor that determines the power consumption of the computer and cooling system requirements. The higher the power consumption, the costlier the equipment required for the internal cooling of the computer.

Reliability is measured as mean time to failure. The storage devices, which require mechanical motion e.g., hard disks, are more prone to failure rather than the semiconductor memories, which are totally electronic in nature. Very high-speed semiconductor memories are also prone to failure as technology is moving towards its limits.

**Cost:** Another key factor which is of prime concern in a memory system, is cost. It is normally expressed per bit. The cost of a memory system includes not only the device but also the cost of access circuitry and peripherals essential for the operation of the memory. Table 3 also shows per bit cost of these memories. Please note that as the access time for memories is increasing, the cost is decreasing.

### SELF-ASSESSMENT EXERCISE

State whether True or False

- Memory hierarchy is built in the computer system, as the main memory can not store very large data. True  False
- The secondary memory is slower than the main memory but has a larger capacity True  False

3. In random access memory any memory location can be accessed independently. True  False
4. Bandwidth is defined as the information that is stored or retrieved in the memory per second. True  False

### 3.3 Random Access Memory

In this section, we will confine our discussions in general to the random access memory, including the discussion on the main memory. The main memory is a random access memory. It is normally organised (logically) as words of fixed length. The length of a word is called word length. Each of these memory words has an independent address and each has the same number of bits. Normally the total numbers of words in the memory are some power of 2. Some typical memory word sizes are 8 bits, 16 bits, 32 bits, etc. The main memory can be both read and write into, therefore, it is called read-write memory.

The access time and cycle time in RAMs are constant and independent of the location accessed. How does this happen? To answer it, let us first discuss how a bit can be stored using a sequential circuit. Figure 38(a) shows the logic diagram of a binary cell.

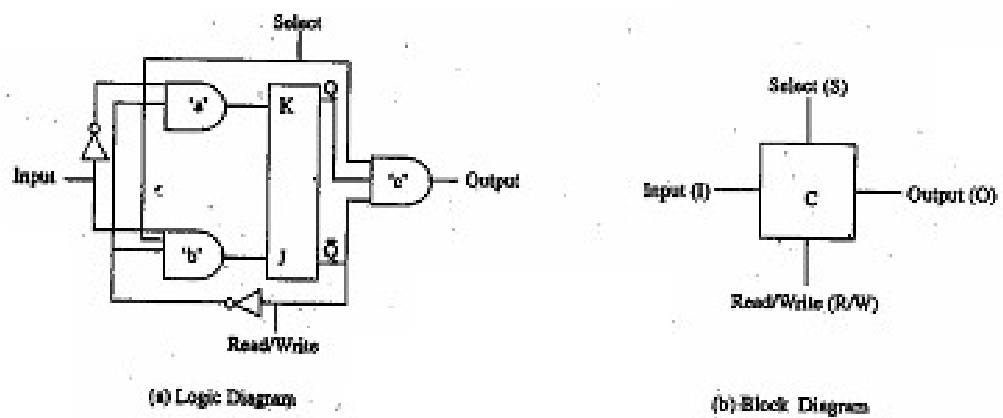


Figure 38(a) Bit Inputs



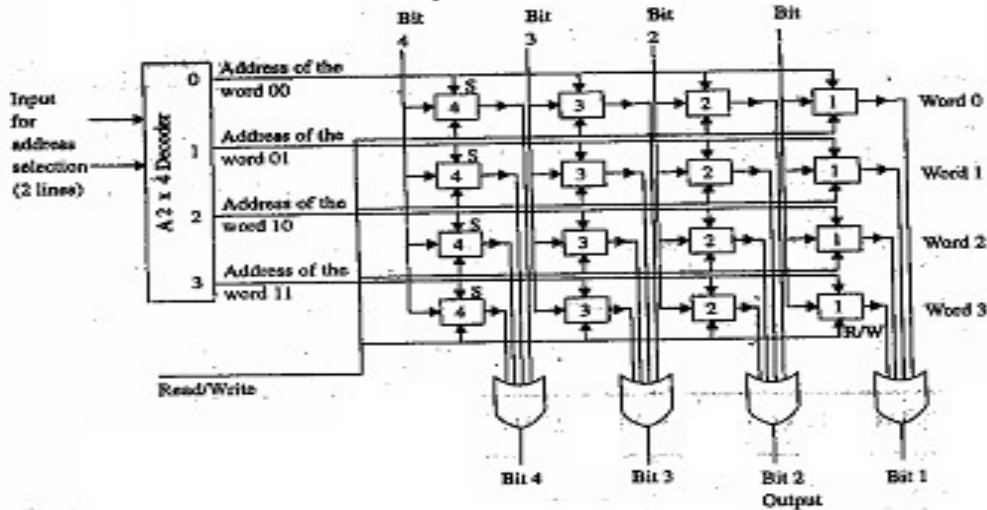


Figure 38(b): Logic diagram of RAM

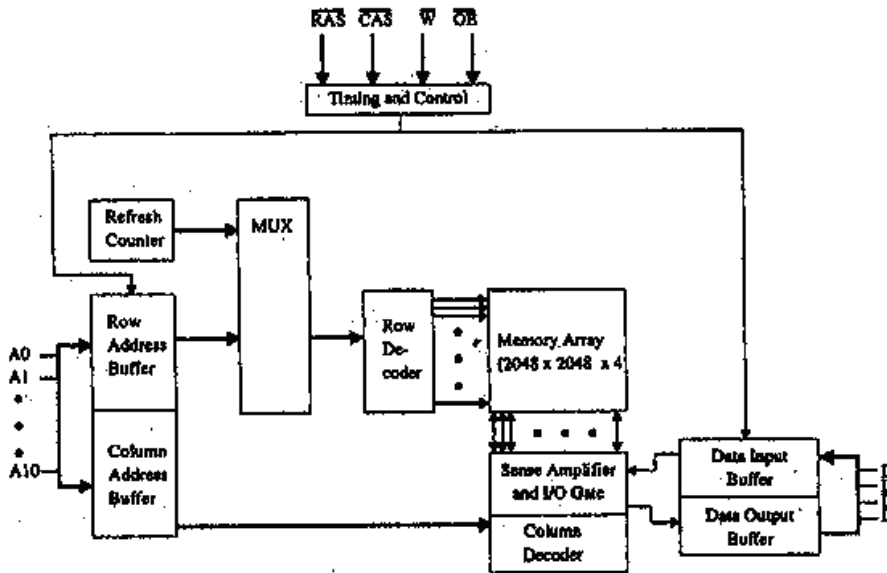


Figure 38(c): A typical 16 megabit DRAM (4M x 4)

The construction shown is made from one J-K flip-flop and three AND gates. The two inputs to the system are one input bit and read/write signal. Input is fed in complemented form to AND gate 'a'. The read/write signal has a value 1 if it is a read operation. Therefore, during the read operation the AND gate 'c' has the read/write input as 1. Since AND gate 'a' and 'b' have 0 read/write inputs, and if the select is 1, i.e., this cell is currently being selected, then the output will become equal to the state of flip-flop. In other words the data value stored in flip-flop has been read. In write operation only 'a' & 'b' gates get a read/write value 1 and they set or clear the JK flip-flop depending on the data input value. Please note that in case data input is 0, the flip-flop will go to the clear state and if data input is 1, the flip-flop will go to the set state. In effect, the input data is reflected in the state of the flip-flop. Thus, we say that the input data has been stored in the flip-flop or binary cell.

Figure 38(b) is the extension of this binary cell to an IC RAM circuit where a 2 x 4 bit decoder is used. Please note that each decoder output is connected to a 4-bit word and the read/write signal is supplied to each binary cell. The output is derived using an OR gate, since all the non-selected cells will produce a zero output. The word that is selected will determine the overall output.

Figure 38(c) is a modified organisation of figure 38(b). It shows a typical organisation of 2048 x 2048 x 4 bit DRAM chip. The memory array in this organisation is a square array, that is (2048 x 2048) words of 4 bits each.

Each element (which consists of 4 bits) of array is connected/identified by horizontal row lines and vertical column lines. The horizontal lines are connected to the select input in a row, whereas the vertical lines are connected to the output signal through a sense amplifier or data in signal through the data bit line driver. Please note that selection of input from this chip involves the understated:

- Row address selection specifying the present address values  $A_0$  to  $A_{10}$ . For the rows, it is stored in the row address buffer through the decoder.
- The row decoder selects the required row.
- The column address buffer is loaded with the column address values, which are also applied through  $A_0$  to  $A_{10}$  lines only. Please note that these lines should contain value for the column. This job will be done through change in external signal  $\overline{RAS}$ .
- CAS causes the column address to be loaded with these values.
- Each column is of 4 bits, these require 4 bit data lines from input/output buffer. On memory write operation data in bit lines being activated while on read sense lines being activated.
- This chip requires 11 address lines (instead of 22), 4 data in and out lines and other control lines.
- Refreshing of the chip is done periodically using a fresh counter. One simple technique of refreshing may be to disable read-write for some time and refresh all the rows one by one.
- The size of the chip is  $2^{11} \times 2^{11} \times 4 = 2048 \times 2048 = 16\text{M}$  bits. On increasing address lines from 11 to 12 we have  $2^{12} \times 2^{12} \times 4 = 64\text{M}$

bits, an increase of a factor of 4. Thus, possible sizes of such chip may be 16K, 256K, 1M, 4M, 16M, and so on.

Having discussed the general configuration of RAMs, we will go on to discuss a few technologies and techniques used in RAMs.

### **3.3.1 Ferrite-core Memories**

Ferrite core memories were used as main memory in computers before the semiconductor memories were invented. The ferrite core memories are based on the magnetic properties of the ferrite material. The core is a ring shaped ferrite material, which is used for storing binary information. The core can be magnetised clockwise or anti-clockwise thus representing logical 0 and 1. Electronic current is used to magnetise the core, and even after the current is removed the ferrite material stays in the specific magnetic state. This implies that the ferrite core memories were non-volatile. Large RAMs can be made from ferrite core by arranging them in multidimensional arrays.

The ferrite core memory requires two wires - one for writing the data and the other for reading or sensing. The reading process is a destructive readout hence it requires writing along with a reading operation. The core determines the cycle time. The smaller the core, the lower the cycle time, but the more complex are the writings. The main disadvantages of ferrite-core memories were:

1. They were incompatible with the processor technology, which are semiconductor-based logic circuits.
2. They were difficult to construct because of difficult wiring patterns. These wires are needed for magnetisation.

### **3.3.2 Semiconductor Memories**

The semiconductors were used for making high-speed CPU registers since 1950, but it was economic to construct RAM chip only in 1970 with the development of VLSI technology. At present RAMs are manufactured in a wide range of sizes i.e., from a few hundred bits to a megabit or more. The present limit on technology does not allow for constructing one giga bits on a single chip. To construct large memories the small IC RAMs can be combined. The semiconductor memories fall under two main technologies- bipolar semiconductor memories and metal oxide semiconductor (MOS) transistor semiconductor memories. For larger RAM chips normally MOS is used. At present there are two main categories of semiconductor RAMs - static and dynamic. The larger chips are normally constructed as dynamic RAMs because the

dynamic RAM require less number of transistors than the static RAM and hence can be packed more densely on a single chip. Thus, dynamic RAMs can achieve higher storage density.

On the other hand, dynamic RAMs tend to lose their charge with time and need periodic refreshing; thus, requiring extra control circuitry and interleaving of normal memory access with refreshing operations. Thus, the dynamic RAMs although can be packed more densely yet are more difficult to use than static RAMs. In contrast to ferrite core memory the semiconductor RAMs are volatile in nature.

These memories which we have discussed are both read/write type. But what about a memory in which we have only one of the operations possible, e.g., if we only allow writing in the memory and no reading then how can we use that memory? Probably it is wastage of memory, but what about having a memory where we cannot change the information in normal case and only read the information from the memory. This memory might have a lot of uses; for example, an important bit of the computer's operating system that normally does not change can be stored in these kinds of memory so that one cannot change the operating system accidentally. These memories are called ROMs (Read Only Memories).

### 3.3.3 Read Only Memories

A ROM is basically a combinational circuit and can be constructed as:

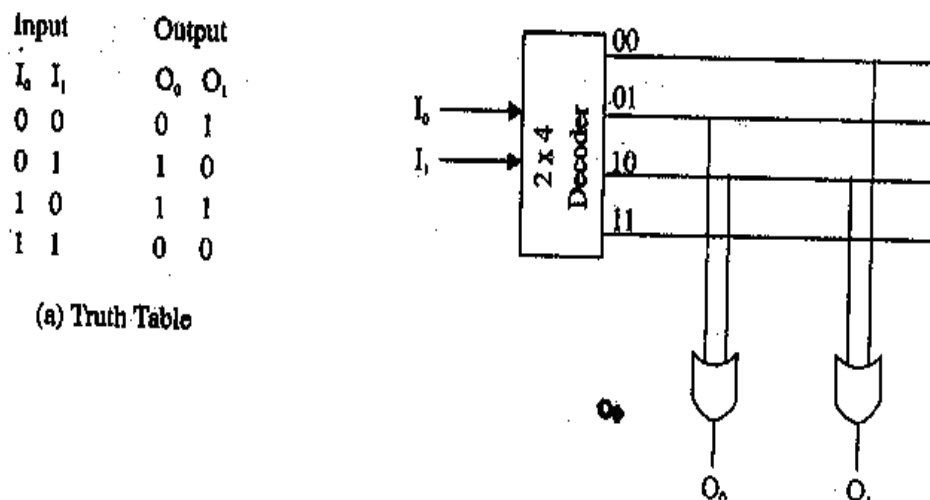


Figure 39: A sample ROM

Thus, using this hardwired combinational circuit we can create a ROM. Please note that on applying an input  $I_0 = 0$ ,  $I_1 = 0$ , the 00 line of the decoder is selected and we will get  $O_0 = 0$  and  $O_1 = 1$ ; on applying  $I_0 = 0$

and  $I_1=1$  we will get  $O_1 = 1$  and  $O_1 = 0$ . This same logic can be used for constructing larger ROMs.

ROMs (Read Only Memories) are the memories on which it is not possible to write the data when they are online to the computer. They can only be read. The ROMs can be used for storing micro-programs, system programs, and subroutines. ROMs are non-volatile in nature and need not be loaded in secondary storage devices. ROMs are fabricated in large numbers in a way where there is no room for even a single error. But, this is an inflexible process and requires mass production. Therefore, a new kind of ROM called PROM was designed which is also non-volatile and can be written only once and hence the name programmable ROM (PROM). The supplier or the customer can perform the writing process in PROM electrically. Special equipment is needed to perform this writing operation. Therefore, PROMs are more flexible and convenient than ROMs.

The ROMs/PROMs can be written just once (in ROMs at the time of manufacture and PROMs at any time later), but in both cases whatever is written once cannot be changed. But, what about a case in which you read mostly but write only very few times. This led to the concept of read mostly memories and the best examples of these are EPROMs (Erasable PROMs) and EEPROMs (Electrically Erasable ROMs). The EPROMs can be read and written electrically but, the write operation is not simple. It requires erasure of whole storage cells by exposing the chip to ultra violet light, thus bringing them to the same initial state. This erasure is a time consuming process. Once all the cells have been brought to the same initial state, the EPROM can be written electrically.

EEPROMs are becoming increasingly popular, as they do not require prior erasure of previous contents. However, in EEPROMs writing time is considerably higher than reading time. The biggest advantage of the EEPROM is that it is a non-volatile memory and can be updated easily, while the disadvantages are the high cost and that at present they are not completely non-volatile, and the write operation takes considerable time. But all these disadvantages are disappearing with the growth in technology. In general, ROMs are made of cheaper and slower technology than RAMs. Table 4 summarises the features of these read only and read mostly memories.

**Table 4: Features of read only and read mostly memories**

Memory Type	Write Time	Order of Read Time	Number of Write Cycles allowed
ROM	Once	Nano seconds	ONE
PROM	Hours	Nano seconds	TENS
EPROM	Minutes (including time of erasure)	Nano seconds	HUNDREDS
EEPROM	Milliseconds	Nano seconds	THOUSANDS

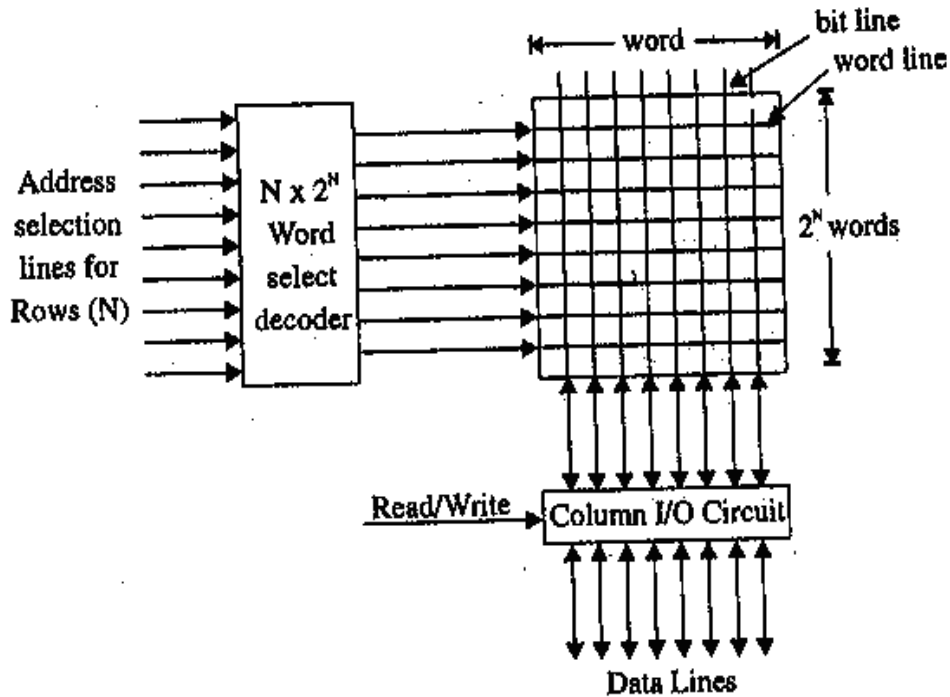
One of the new memory technologies is flash memory. These memories can be reprogrammed at high speed and hence the name flash. The flash memory characteristics such as cost and write time, etc., fall in between those of EPROM and EEPROM. In flash memories the entire memory can be erased in a few seconds (compare it to EPROM) by using electric erasing technology. There is another possibility in flash memory in which erasure of a block is possible. Flash memories are used for some of these chips where writing mode is not disabled in personal computers. BIOS may be written over/destroyed by some viruses.

We have discussed semiconductor memories; we will now discuss the chip organisation of these memories.

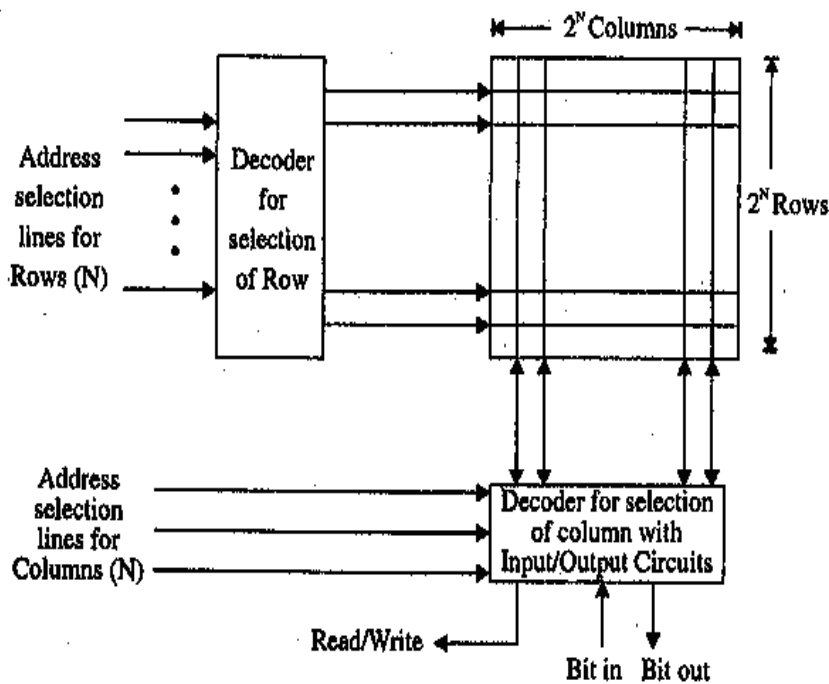
### 3.3.4 Chip Organisation

Most of the semiconductor memories are packaged in chips. As discussed earlier these memory chips may store information ranging from 64K bits to 1M bits. There are several memory organisation techniques used for a chip and the most common of these are 2D and 2½ D organisation.

*2 D Memory Organisation:* In this organisation the memory on a chip is considered to be a list of words in which any word can be accessed randomly e.g., the memory of PCs used to have 16 bit words and normally in a chip it has 64KB memory = 32K words. Figure 40(a) shows a typical 2-D organisation.



(a) 2D organisation



(b) 2 1/2 D organisation

Figure 40: 2 and 2 1/2 chip organisation

The memory in 2D chip is organised as an array of words. The horizontal lines are connected to the select input of the binary cell. This circuit is a simplified form of the circuit given in Figure 38(b). Each vertical bit line is connected to the data-in (or input) and sense (or

output) terminal of each cell in its respective column. Each decoded line of decoder drives a word line. A complete word can be an input or output from the memory simultaneously.

On write operation (input to memory) the address decoder selects the required word and the bit lines are activated for a value 0 or 1 according to the data lines values. Thus, enabling input of data to memory or in other words completes the write operation. On read (output from memory) the value of each bit line is passed through a sense amplifier and passed on to the data lines, thus, enabling the read operation. The word line identifies the word that has been selected for reading or writing. Usually, ROMs and read mostly memories use 2 D chip organisation.

Another chip organisation, which is popular presently, is  $2\frac{1}{2}$  D organisation. In  $2\frac{1}{2}$  D organisation, bits of a word are spread over a number of chips. For example a 32 bit word can be stored on four chips containing 8 bits of the word. But the ideal organisation in this will be to have 1 bit of a word on a single chip. A  $2\frac{1}{2}$  D organisation is given in Figure 40(b). The figure is a square array of cells (also refer to figure 38(c)). A row line and a column line are connected to each memory cell. The addresses supplied to this chip are divided into row and column addresses lines and is then used to input or output bit/bits from this memory chip. The other similar memory chips can deliver the rest of the bits of this word.

### **Comparison of 2D and $2\frac{1}{2}$ D Organisation**

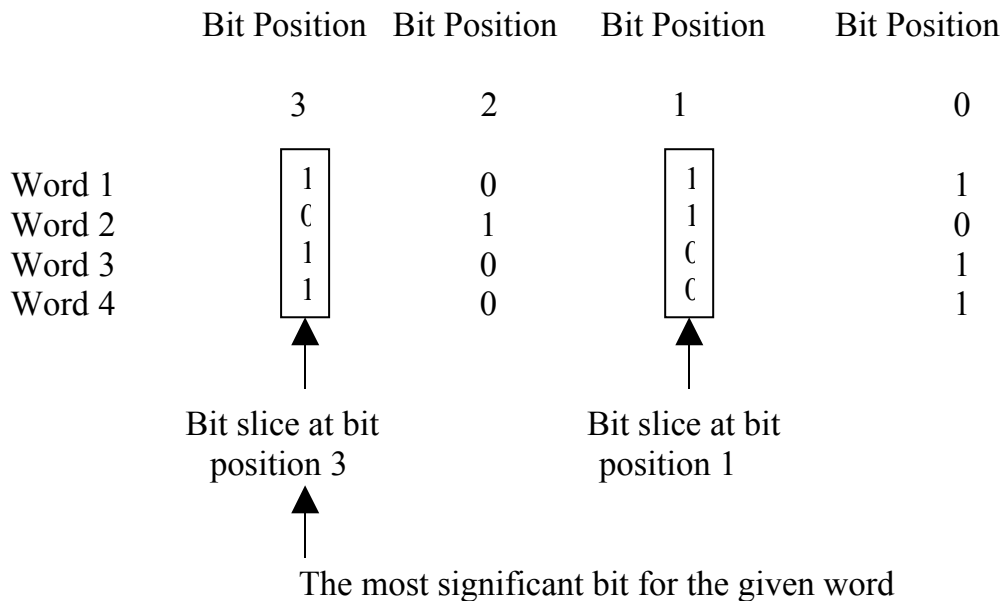
The  $2\frac{1}{2}$  D organisation of chips is supplied to be more advantageous because:

1. It requires less circuitry and gates. (Why? Find out from further readings).
2. The chip has only one input/output pin in  $2\frac{1}{2}$  D while in 2 D it has to have 16 or 32 input/output pins, thus in the chip packages less number of pins are required for  $2\frac{1}{2}$  D organisation which is a desirable feature.
3. In the 2-D organisation the error correction codes cannot be used effectively. For example, if electromagnetic disturbances have affected a chip, in  $2\frac{1}{2}$  D we can rectify the errors as only one bit of the word is lost but it does not happen in a 2-D organisation.



## Orthogonal Memory

Orthogonal memory can be accessed either by a word or by a bit-slice. A bit-slice is a set of all the bits of the same bit position of a specific set of words. The user may request for a word and on his request a word will be read. But on the other hand, if user requests a bit-slice read or write for a bit-slice, he will be allowed to do so. Figure 41 shows a bit-slice for a 4-bit word. Such memories may be advantageous in situations where set type operations are to be performed.



**Figure41: Bit-slice for a 4-bit word**

### 3.4 External/Auxiliary Memory

As discussed earlier, the cost of RAM is very high and the semiconductor RAMs are mostly volatile, therefore, it is highly likely that a secondary cheap media should be used which should show some sort of permanence of storage and should be relatively inexpensive. The magnetic material which was found to be inexpensive and quite long-lasting, therefore, became an ideal choice to do so. Magnetic tapes and magnetic disks are commonly used as storage media. With the advancement in optical technology now the optical disks are trying to make an inroad as one of the major external memory. We will discuss the characteristics of these memories in this subsection.

#### 3.4.1 The Magnetic Disk

A magnetic disk is a circular platter of plastic that is coated with magnetisable material. One of the key components of a magnetic disk is a conducting coil named the head, that performs the job of reading and

writing on the magnetic surface. The head remains stationary while the disk rotates below it for reading or writing operations.

For writing data on the magnetic disk, current is passed through the head, which produces the magnetic field. This magnetic field causes magnetic patterns to be recorded on the magnetic surface. These recorded magnetic patterns depend on the direction of the current in the head. While for reading the magnetic field recorded on the disk surface moves relative to the head resulting in the generation of electric current of the same polarity in the head. The capacities of magnetic disks range from 1MB (for floppy disks) to several MB.

### Data Organisation and Format

The head of the disk is a small coil and reads or writes on the position of the disk rotating below it, therefore, the data is stored in a concentric set of rings (refer Figure 42). These are called tracks. The width of a track is equal to the width of the head. To minimise the interference of magnetic fields and to minimise the errors of misalignment of the head, the adjacent tracks are separated by inter-track gaps.

As we go towards the outer tracks the size of a track increases but to simplify electronics, same numbers of bits are stored on each track. Therefore, the linear storage density from the outer to the inner track increases per linear inch. Figure 42 gives the details of a disk.

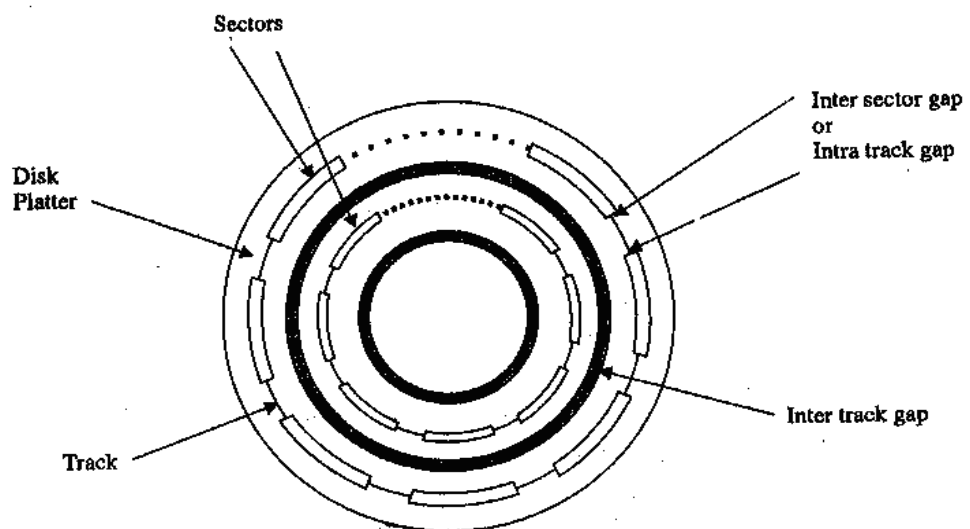


Figure 42: Logical layout of Magnetic Disk

The data is transferred from and to the disks in blocks. A block is a section of disk data and is normally equal to one or more sector(s). A track is divided into 10 – 100 sectors and these sectors should be either

fixed or variable length sectors. Two adjacent sectors are separated by intra-track gaps. This helps in reducing the precision requirements of sectors. To identify the sector position normally there may be a starting point of a track and a starting and end point of each sector. But how is a sector of a track recognised? A disk is formatted to record control data on it such that some extra data is stored on it for identification purposes. This control data is accessible only to the disk drive and not to the user. For example, Figure 43 shows the format of a Winchester disk.

Thus, in the Winchester disk a sector of 600 bytes can contain only 512 bytes of data. The rest is control information. The synchronisation byte is a special byte that signifies the starting of a field. Both the data and identification field start with synchronisation byte, and have 2 bytes for error-detecting codes. The identification field contains the basic information such as the track number, sector number and head number (as multiple platters are there in the Winchester disk) which are needed to identify a sector.

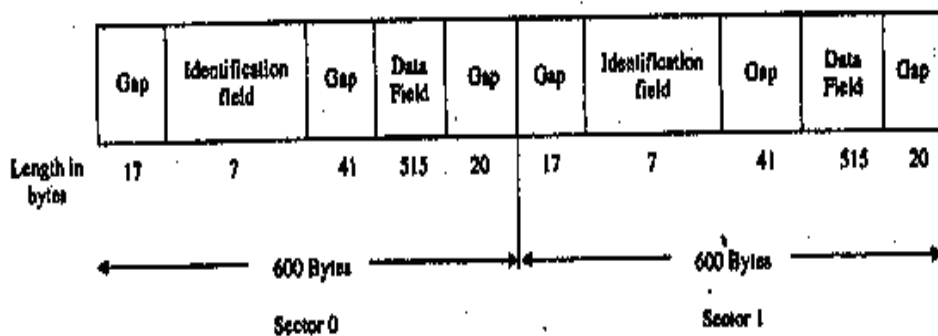
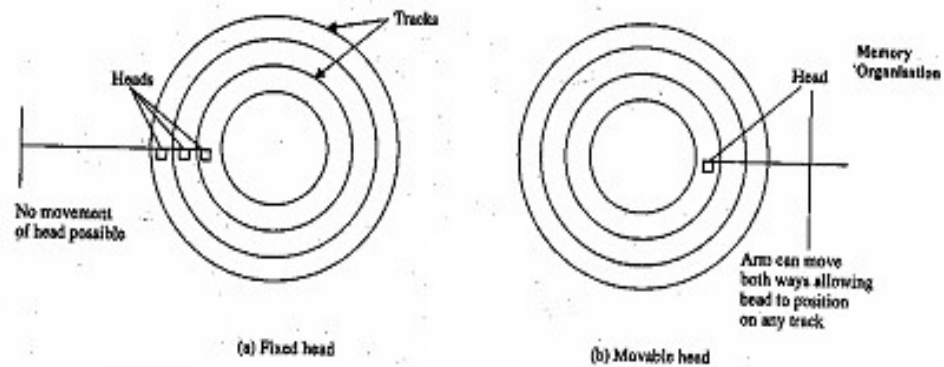


Figure 43: Format of two adjacent sectors on a Winchester disk

Various characteristics of disks can be classified as:

**Fixed Head/Movable Head Disks:** The disks in which the head does not move are called fixed head. Such disks require one read/write head per track. Heads can be mounted on a rigid arm, which extends to the centre of the disk. On the movable head disk normally we have one read/write head, which is fitted on an arm. This arm allows the head to position on any track of the platter. Figure 44 shows these fixed and movable head schemes.



**Figure 44: Fixed and movable head disks**

**Removable and Non-removable Disks:** The disks are normally mounted in a disk drive that consists of an arm and a shift along with the electronic circuitry for input-output of data. The disk rotates along with the shaft. A non-removable disk is permanently mounted on the disk drive. An example of a non-removable disk is the hard disk of a PC. The removable disks can be replaced by a similar disk on to the same or a different computer, thus, providing enormous data storage that is not limited by the size of the disk. Examples of such disks are floppy diskettes.

**Sides:** If the magnetic coating is applied to both sides of the platter, the disk is called a double-sided disk. The data can be recorded on either side of these disks. Some inexpensive disks were initially single-sided.

**Platters:** Some of disks have a single platter (e.g., floppy disks) while some disks have multiple platters that are stacked vertically, normally at a distance of an inch. This is known as a disk pack. In disk a pack is one cylinder. It is the ring of all concentric tracks. A disk pack can contain multiple heads mounted with the same arm.

### Head Mechanism

The head mechanism can broadly be categorised in three categories:

**Contact:** In this mechanism the head is in physical contact with the platter. This mechanism is used for floppy the disk that is a small flexible platter and is of the least expensive type. But with this type of head mechanism the chances of errors from impurities and imperfection are more.

**The Fixed Gap:** Traditionally the read-write heads are placed at a fixed distance from the platter, allowing an air gap. The gap of platter and head plays an important role. The smaller the size of the head the closer it should be to the platter surfaces in order to read and write properly. A

smaller head allows greater data density on the disk, but on the other hand, this head should be closer to the platter surface, therefore, it is prone to errors because of impurities. Other types of disks were developed which are commonly called Winchester disks.

**The Aerodynamic Gap (Winchester):** In the Winchester disks the heads are used in sealed drive assemblies which are almost free from contaminants. The heads operate closer to the disk surface thus allowing a high data density. In these disks, the head is in the form of an aerodynamic foil that rests on the platter surface when the disk is stationary. On rotating the disk an air pressure is generated because of the rotation. This air pressure results in the displacement of the head slightly above the disk surface. The term Winchester was initially used for an IBM disk model but now it is commonly used for any sealed unit disk drive which uses the aerodynamic head mechanism.

### Access Time on Disks

Disks operate in a semi-random mode and normally are referenced blockwise. The data access time on a disk consists of two main components:

- **Seek time:** This is the time to position the head on a specific track. On a fixed head disk, it is the time taken by the electronic circuit to select the required head while on a movable head disk, it is the time required to move the head to a particular track.
- **Latency time:** This is the time required by a sector to reach below the read/write head. On the average it is half of the time taken for a rotation by the disk.

In addition to these two times, the time taken to read a block of words can be considered but normally it is too small in comparison to latency and seek time and in general the disk access time is considered to be sum of seek time and latency time. Since access time of disks is large, it is advisable to read a sizeable portion of data in a single go and that is why the disks are referenced blockwise. In fact, you will find that in most computer systems, the input/output-involving disk is given a very high priority. The basic reason for such priority is the latency time that is once the block which is to be read passes below the read-write head; it may take the time of the order of milliseconds to do that again, in turn delaying the input/output.

## RAID

The technology of the processor and memory is improving at a very fast pace in comparison to the secondary storage technology. Thus, the main performance bottleneck for the computer system is the performance of the secondary storage devices and their interfacing with the CPU. One such attempt in the direction of improving disk performance is to have multiple components in parallel. Some basic questions for such a system are:

1. How are the disks organised?  
Maybe as an array of disks
2. Can separate I/O requests be handled by such a system in parallel?  
Yes, but only if the disk accesses are from separate disks.
3. Can a single I/O request be handled in parallel?  
Yes, but the data block requested should be available on separate disks.
4. Can this array of disks be used for increasing reliability?  
Yes, but for that, redundancy of data is essential.

One such industrial standard which exists for multiple-disk database schemes is termed RAID i.e. Redundant Array of Independent Disks. The basic characteristics of RAID disks are:

- The operating system considers the physical disks as a single logical drive.
- Data is distributed across the physical disks.
- In case of failure of a disk, the parity information that is kept on redundant disks is used to recover the data.

The term RAID was coined by researchers at University of Beckley. In this paper the meaning of RAID was Redundant Array of Inexpensive Disks. However, later the term Independent was adopted instead of Inexpensive to signify performance and reliability gains.

RAID has been proposed at various levels, which are basically aimed to cater for the widening gap between the processor and on-line secondary storage technology.

The basic strategy used in RAID is to replace large capacity disk drives with multiple smaller capacity disks. The data on these disks is

distributed such as to allow simultaneous access, thus, improving the overall input/output performance. It also allows for an easy way of incrementing the capacity of the disk. Please note that one of the main features of the design is to compensate for the increase in the probability of failure of multiple disks through the use of parity information. The six levels of RAID are given in Figure 45. Please note that RAID level 2 are not commercially offered.



(a) RAID 0 (Non-redundant data)



(b) RAID 1 Mirrored dishes



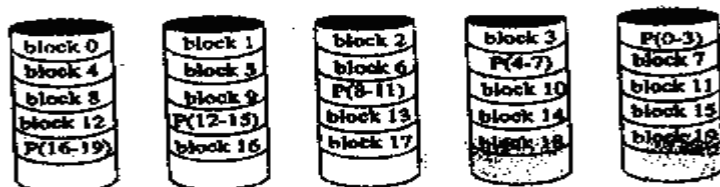
(c) RAID 2 (Redundancy using Hamming Code)



(d) RAID 3 (Bit-Interlevel Parity)



(e) RAID 4 (Block Level Parity of data bits)



(f) RAID 5 (Block Level Distributed Parity)

Figure 45: The six RAID levels

The features of these RAID levels are:

### **RAID Level 0**

- It distributes the data over the disk array in the form of strips, which may be a block, a sector or any other unit of a disk. Please note the mapping of strips (Figure 45 (a)) on various disks.
- Such layout of strips may result in reading/writing of different blocks of a single file from different physical disks in parallel, thus greatly reducing the input/output time.
- Array management software is needed to keep track of what block/strip is on which disk.
- For high data transfer capacity, there should be a high capacity path between memory and individual disk drives, and strip size should be small compared to input/output request size.
- If large numbers of input/output requests are to be processed relating to records, then it is advisable to have strips to cater for multiple requests in a single input/output.
- It is most useful in applications requiring high performance of non-critical data.

### **RAID Level 1**

- Duplicate the data on all the disks also called mirroring. (Please refer Figure 45(b)). Thus, every disk of the array has a mirror disk.
- Any read request can be serviced by any of the two disks, involving minimum seek and latency time of the two.
- Writing will require two parallel writing operations and the time taken is the larger of the two disks access timing.
- Recovery from failure is done using the mirrored disk.
- It is costly but provides real time back up.
- For read operation data, transfer rate and input/output request fulfilment is quite high, for writing, no appreciable gain.
- It is useful for applications such as system drives and critical data files.



## **RAID Level 2**

- This level uses parallel access technique, i.e. the member disks participate for execution of a single input/output request by synchronising the spindles of all disks to the same position at a time; therefore data strips are very small sometimes, of the order of a byte or a word.
- An error correcting code is used that corrects single-bit errors and does double error detection.
- On a single read operation, all the disks are activated and the requested data along with the error-correcting code are delivered simultaneously.
- Error correction feature on a single bit corrects the error instantaneously on a read operation.
- RAID 2 would be an effective choice in cases where data errors are many. However, as present day disks are highly reliable such systems are not industrially accepted.

## **RAID Level 3**

- It requires only a single redundant disk, which is used as parity bit.
- It also employs parallel access as that of level 2 with small data strips.
- Parity bit may be used for reconstruction of data in case a disk fails.
- As the data strips are small, level 3 may achieve a very high data transfer rate. However, only one request can be processed at a time so there is low input/output request fulfilment rate.
- It is most useful for applications where a single input/output request is of a large size, such as imaging or CAD application.

## **RAID Level 4**

- It uses the independent access technique, where each of the physical disks may be accessed independently, thus, enabling fulfilment of separate input/output requests in parallel.
- Data strip is large and bit by bit parity strip is created for bits of each disk.
- Write operation requires updating of parity bit also. This causes user software to read old strips of data as well as old parity strips.
- Parity disk is a bottleneck in level 4 as all input/output will use it. It is industrially not an accepted standard.

### RAID level 5

- In level 5, the level 4 bottleneck of parity disk is avoided by distributing the parity strips on the disk as per diagram shown.
- This RAID level is useful for applications where high input/output request rates are there, where one is having read-intensive or data lookup type operations.

### 3.4.2 Magnetic Tapes

Magnetic tapes consist of tapes constructed from a plastic material and covered with a magnetic oxide layer. These tapes are mounted on reels.

**Storage Format:** The data is stored as one byte at a time. Normally data representation on tapes allows 9 bits inclusive on one parity bit with each byte; therefore, a tape has 9 tracks, each track representing a bit position. Figure 46 shows the format of a magnetic tape. In addition to tracks, data on tapes are written in contiguous blocks. This is also termed a physical record on a tape. These records are separated by gaps known as inter-record gaps.

#### Characteristics

The tape is called a sequential access device, that is, to read a record N, if tape is presently at record 1 then it has to pass through all the records till N-1 sequentially. In case the tape head is positioned beyond Nth record then the tape needs to be re-wound to a particular distance and then forward reading is done to read Nth record. A tape moves only if a read or write operation is requested.

The tape is one of the earliest storage devices. Tapes are low cost, low speed, portable and are still widely used because of their low cost.

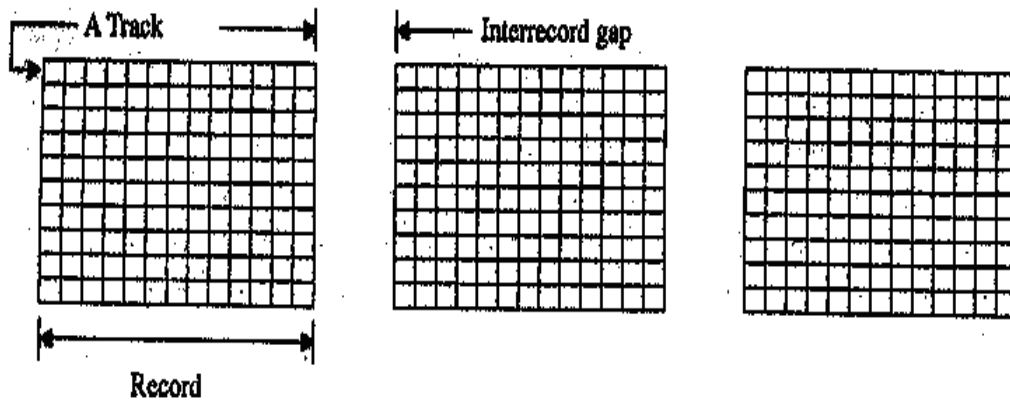


Figure 46: Logical format of a magnetic tape

### 34.3 Charge-Coupled Devices (CCDs)

CCDs are used for storing information. They have arrays of cells that can hold charge packets of electron. A word is represented by a set of charge packets; the presence of each charge packet represents the bit-value 1. The charge packets do not remain stationary and the cells pass the charge to the neighbouring cells with the next clock pulse. Therefore, cells are organised in tracks, with a circuitry for writing the data at the beginning and a circuitry for reading the data at the end. Logically the tracks (one for each bit position) may be conceived as loops since the read circuitry, pass the information back to the write circuit, which then re-creates the bit values in the tracks unless new data is written to the circuit.

These devices come under the category of semi-random operation since the devices must wait till the data has reached the circuit for detecting of change packets. The access time to these devices is not very high. At present this technology is used only in specific application and commercial products are not available.

### 3.4.4 Magnetic Bubble Memories

On applying the magnetic field in certain material such as garnets, certain cylindrical areas whose direction of magnetisation is opposite to that of the magnetic field are created. These are called magnetic bubbles. The diameter of these bubbles is found to be in the range of 1 micrometer. These bubbles can be moved at high speed by applying parallel magnetic field to the plate surface. Thus, the rotating field can be generated by an electromagnetic field and no mechanical motion is required.

In these devices deposition of a soft magnetic material called permalloy is made as a predetermined path, thus, it makes a track. Bubbles are

forced to move continuously in a fixed direction on these tracks. In these memories the presence of a bubble represents a 1 while its absence represents a 0 state. For writing data into a cell, a bubble generator (to introduce a bubble) and an annihilator (to remove a bubble) are required. A bubble detector performs the read operation. Magnetic bubble memories having capacity of 1M or more bits per chip have been manufactured. The cost and performance of these memories fall between semi-conductor RAMs and magnetic disks.

These memories are non-volatile in contrast to semi-conductor RAMs. In addition, since there are no moving parts, they are more reliable than the magnetic disk; but, these memories are difficult to manufacture and difficult to interface with conventional processors. These memories at present are used in specialised applications, e.g. as secondary memory of air or space borne computers, where extremely high reliability is required.

### **3.4.5 Optical Memories**

Optical memories are alternate mass storage devices with huge capacity. The advent of the compact disk digital audio system, a non-erasable optical disk, paved the way for the development of a new low cost storage technology. In optical storage devices the information is written using laser beam. These memories can store large amounts of data. We will discuss some of the optical memory devices that are now becoming increasingly popular in various computer applications.

#### **The CD-ROM**

The CD-ROM (compact disk read-only memory) is a direct extension of audio CD. CD-ROM players are more rugged and have error-correction facility. This ensures proper data transfer from CD-ROM to the main memory of the computer. The CD-ROM disk is normally formed from a resin named polycarbonate that is coated with aluminium to form a highly reflective surface. The information on the CD-ROM is stored as a series of microscopic pits on this reflective surface. A high-intensity laser beam is focused to create pits on the master disk. This master disk is then used to make a disk that is used to make copies. A topcoat of clear lacquer is applied on the CD-ROM's surface to protect it from dust and scratches.

For information retrieval from a CD-ROM, a low-powered laser, which is generated in an optical disk drive unit, is used. The disk is rotated and the laser beam is aimed at the disk on a particular mark. The intensity of the reflected laser beam changes as it encounters a pit. A photosensor

detects the change in intensity, thus, recognising the digital signals recorded on the CD-ROM surface.

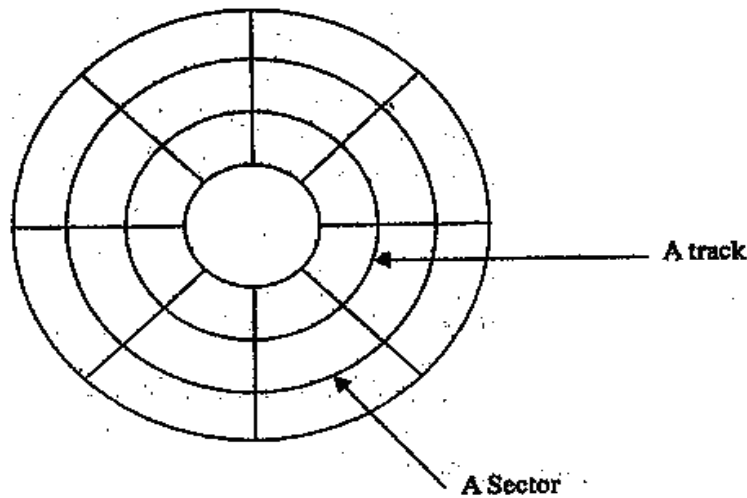


Figure 47: Layout of a CAV disk

Now, let us consider the rotation of a disk. A point that is closer to the center rotates at a slower rate in comparison to a point that is near to the periphery of the disk. Therefore, a pit, which is closer to the center of a rotating disk passes the laser beam slower than that of a pit that is on the outside. How can this variation of speed be compensated for? One of the solutions is by increasing the spacing between bits of information stored (same as in magnetic disks) as we move towards the outside of the disk. This information should be scanned at the same rate irrespective of distance from the centre, on rotating the disk at a constant speed. This mechanism is known as the constant angular velocity (CAV). Figure 47 shows the sectors of a disk using CAV. Please note that the density of information will decrease as we move towards outside of the disk. The main advantage of having CAV is that individual blocks of data can be accessed at semi-random mode. Thus, the head can be moved from its current location to a desired track and wait for specific sector to spin under it. The basic disadvantage of the CAV disk is that a lot of storage space is wasted, since the longer outer tracks are storing the data only equal to that of shorter innermost track.

Because of this disadvantage, the CAV method is not recommended for use on CD-ROMs. In CD-ROMs the information is stored evenly across the disk in segments of the same size. But here the pits need to be read by the laser beam at the same rate. This is achieved by rotating the disk at variable speed. This is referred to as constant linear velocity (CLV). In CD-ROMs data stored on a track increases as we go towards outer surface of the disk. The addresses in CLV disks are represented by units of minutes (0 to 59), seconds (0 to 59) and blocks (0 to 74). This address information is carried at the beginning of each block. A 60 minutes CD-

ROM can hold  $60 \times 60 \times 75 = 270,000$  blocks. Each of these blocks can store 2048 bytes or 2K of data. Thus, a usual CD-ROM can store 553 MB of data. The block format of a CD-ROM is shown in Figure 48.

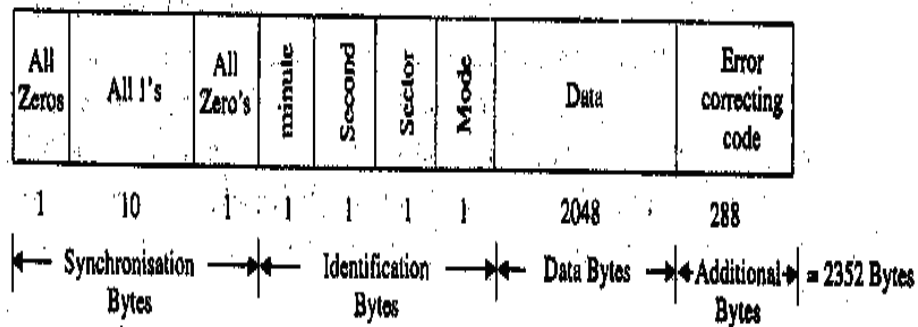


Figure 48: Block format for CD-ROM

The beginning of a block is marked by one byte of all 0s, followed by ten bytes of all 1s, followed by one byte of all 0s.

The block address is indicated by 4 bytes, one each for recording minute, second and sector and the fourth byte specifies a mode of data storage. For example, mode 0 indicates a blank data field, while mode 1 indicates that the last 288 bytes are used as error-detecting codes. Mode 2 indicates that the last field is used as an additional data field. Therefore, in mode 2, the block of CD-Rom stores  $2048 + 288 = 2336$  bytes of data.

The identification field is followed by 2KB of user data, followed by 288 bytes of additional data that may be used as an error-correcting code or as an additional data field, depending on the mode.

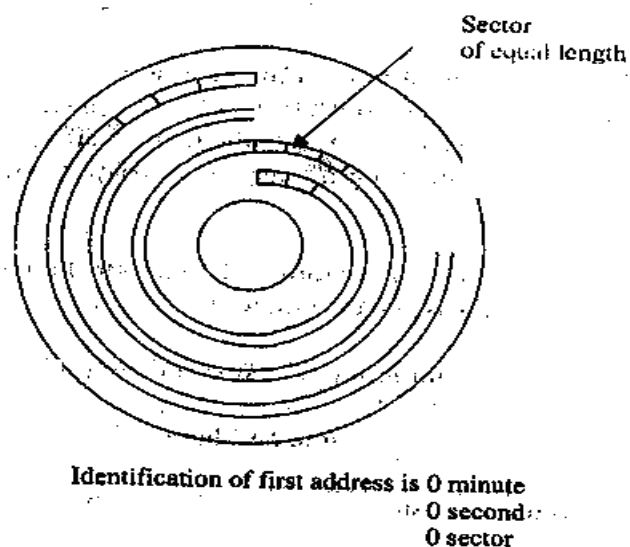


Figure 49: CLV CD-ROM's disk layout

Figure 49 indicates the layout used for CD-ROMS. As discussed earlier, the data is stored sequentially along a spiral track. In this disk random access becomes more difficult because locating a desired address involves first moving the head to the specific area then adjusting the rotation speed and then reading the address, and then finding and accessing the specific sector.

CD-ROMs are very good for distributing large amounts of information or data to large numbers of users. The three main advantages of CD-ROMs are:

- Large data/information storage capacity
- Mass replication is inexpensive and fast
- These are removable disks, thus, are suitable for archival storage.

The disadvantages of the CD-ROM are:

- It is read-only therefore, cannot be updated.
- Access time is longer than that of a magnetic disk.

## **WORM**

In certain applications only a few copies of compact disks are to be made which makes the CD-ROMs production economically unviable. For such cases the write-once read-many CD has been developed. WORM disks are prepared in such a way that they can be written only once by a laser beam of modest intensity. The disk controller of WORM is more expensive than that of the CD-ROM. WORM uses CAV mechanism to provide rapid access on account of some capacity. The WORM disk is first prepared by high-power laser. For example, a typical technique for preparing the disk is to produce a series of blisters in the disk using high-power laser. A low-powered laser then can be used to generate just enough heat to burst these blisters, wherever desired, in a WORM drive. The read operation is carried out by a laser in the WORM drive that illuminates the disk's surface. These burst blisters provide higher contrast to that of the surrounding area, thus, are recognised easily.

## **The Erasable Optical Disk**

The data in these disks can be changed repeatedly as the case with any magnetic disk. A feasible technology that has proved commercially feasible for the erasable optical disk is the magnetic-optical system. In such systems, a laser beam is used along with a magnetic field to read or write the information on a disk that is coated with a magnetic material. The laser beam is used to beat a specific spot on the magnetic coated

medium. At the elevated temperature, the magnetic field is then applied; thus, the polarisation of that spot can be changed in turn recording the desired data. This process does not cause any physical change in the disk, therefore it can be repeated many times. The read operation is performed by detecting the degree of rotation of the polarised laser beam reflected from the surface.

The erasable optical disk is a true secondary storage device (unlike CD-ROMs and WORM). The main advantages of the erasable optical disk over the magnetic disk are:

1. The erasable optical disks are portable while a Winchester disk is not.
2. The erasable optical disks are highly reliable and have a longer life.
3. The erasable optical disk also uses constant angular velocity, thus, making semi-random access feasible.

The only disadvantage of this disk is the high cost. This disadvantage will disappear in the near future.

### **Archival/back-up Storage Technologies**

Many such proprietary/commercial technologies such as ZIP drives, cartridge tape drives, read-write CDs are available presently in the market. You can find more information on them from the further readings or the World Wide Web.

## **3.5 High Speed Memories**

The Need: Why the high speed memories? Is the main memory not a high-speed memory? The answer to the second question is definitely “No”. But why so? Well for this we have to go to the fundamentals of semiconductor technology, which is beyond the scope of the unit. Then, if the memories are slower then how slow are they? On the average it has been found that the operating speed of main memories lack by a factor of 5 than that of the speed of processors (such as the CPU or input output processors).

In addition each instruction requires several memory accesses (It may range from 2 to 7 or even more sometimes). If an instruction requires even 2 memory accesses even then almost 80% time of executing an instruction, processor waits for memory access.



What can be done to increase this processor-memory interface bandwidth?

There are four possible answers to the questions. These are:

1. Decrease the memory access time, use a faster but expensive technology for the main memory, probably it will be feasible after few years.
2. Access more words in a single memory access cycle. That is instead of accessing one word from the memory in a memory access cycle, access more words.
3. Insert a high-speed memory known as Cache between the main memory and the processor.
4. Use associative addressing in place of random access.

Hardware researchers are taking care of the first point. Let us discuss the schemes for the remaining three points.

### 3.5.1 Interleaved Memories

The interleaved memories are the implementation of the concept of “accessing more words in a single memory access cycle”. But the question is how can it be achieved? By partitioning the memory into say, N separate memory modules. Thus N accesses can be carried out to the memory simultaneously. Figure 50 gives a block diagram for the interleaved memory system. But what are the requirements for such a system? For such a system we will be requiring:

- An independent addressing circuitry for each of the N modules.
- An appropriate bus controls mechanism, in case the physical buses between the process and memory are shared by the modules.
- Sufficient buffer storage in the “processor” to control the increased flow of information.

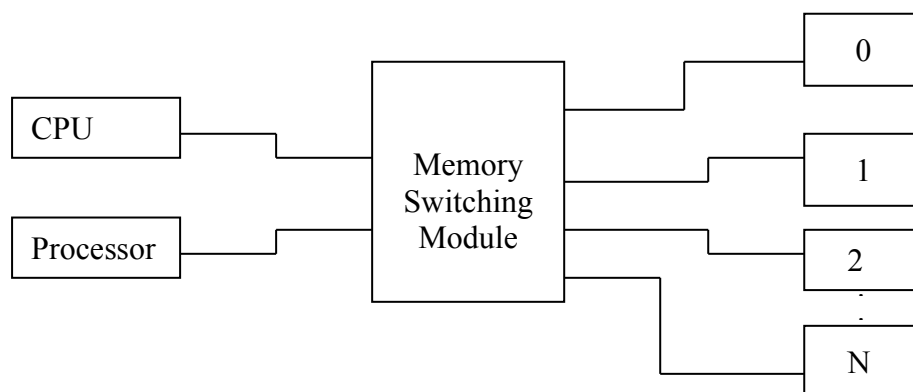
Where can this memory be useful? Well, this kind of interleaved memory has found application in multi-processor systems where many processors require access to common memory simultaneously. But different processors can access the memory simultaneously only if they desire access to different modules of the memory. For utilising such a system effectively, the memory references generated by referencing processors should be distributed evenly among the N module. In an ideal case for a set of N consecutive references to the memory every reference should be to a separate module. In such a case simultaneously memory

access can be carried out. In case two or more than two references are to the same module then these memory accesses cannot be carried out simultaneously. The maximum processor memory bandwidth in interleaved memory can be equal to the number of modules i.e.,  $N$  words per memory cycle.

To achieve the address, interleaving consecutive addresses are distributed among the  $N$  interleaved modules. For example, if we have consecutive addresses and 4 interleaved memory modules then the 1<sup>st</sup>, 5<sup>th</sup>, 13<sup>th</sup>... address will be assigned to the first memory module and so on.

1, 5, 9, 13, 17	Addresses to Memory Module 1
2, 6, 10, 14, 18	Addresses to Memory Module 2
3, 7, 11, 15, 19	Addresses to Memory Module 3
4, 8, 12, 16, 20	Addresses to Memory Module 4

For example, CRAY 1 supercomputers have a CPU cycle time (which is defined as the shortest time in executing a micro-operation by the CPU) of 12.5 ns and main memory of cycle time 50 ns and word size of 64 bits. Since one instruction in CRAY normally reads one instruction word, two operand words and one result word i.e. 4 words, therefore, 4 memory cycles are required for a CPU cycle. Since 1 memory cycle = 4 CPU cycle time, therefore, at least  $4 \times 4 = 16$  memory modules are used to make it 16 module interleaving for executing the instruction. This way CPU wait time is minimised.



**Figure 50: Interleaved memories**

### 3.6.2 Cache Memories

These are small fast memories placed between the processor and the main memory (see Figure 51). Caches are faster than the main memory. Then why do we need the main memory at all? Well once again it is the cost which determines this. The caches although are fast yet are very expensive memories and are used in only small sizes. For example, caches of sizes 128K, 256K etc. are normally used in typical Pentium based systems, whereas they can have 4 to 128 MB RAMs or even more. Thus, small cache memories are intended to provide fast speed of memory retrieval without sacrificing the size of memory (because of main memory size). If we have such a small size of fast memory how can it be advantageous in increasing the overall speed of memory references? The answer lies in the “principle of locality”, which says that if a particular memory location is accessed at a time then it is highly likely that its nearby locations will be accessed in the near future.

Cache contains a copy of certain portions of main memory. The memory read or writes operation is first checked with the cache and if the desired location data is available in cache then it is used by the CPU directly. Otherwise, a block of words is read from the main memory to cache and the word is used by the CPU from cache. Since cache has limited space, for this incoming block a portion called a slot needs to be vacated in cache. The contents of this vacating block are written back to the main memory at the position it belongs to. The reason for bringing a block of words to cache is once again locality of reference. We expect that the next few addresses will be close to this address and therefore, the block of words is transferred from the main memory to cache. Thus, for the word, which is not in cache, access time is slightly more than the access time for main memory without cache. But because of locality of references, the next few words may be in the cache, thus, enhancing the overall speed of memory references. For example, if memory read cycle takes 100 ns and a cache read cycle takes 20 ns, then for four continuous references (first one brings the main memory contents to cache and next three from cache).

The time taken with cache	= (100+20)	+ 20 x 3
	for the first	for last three
	read operation	read operation
	= 120+60 = 180	
Time taken without cache	= 100x4 = 400 ns	

Thus, the closer the reference interval, the better the performance of cache and that is why a structured code is considered a good programming practice, since it provides maximum possible locality.

The performance of cache is closely related to the nature of the programs being executed; therefore, it is very difficult to say what should be the optimum size of cache, but in general a cache size between 128k to 256k is considered to be optimum for most of the cases. Another important aspect in cache is the size of the block (refer Figure 51) that is normally equivalent to 4 to 8 addressable units (which may be words, bytes etc). Figure 51 gives the cache memory organisation.

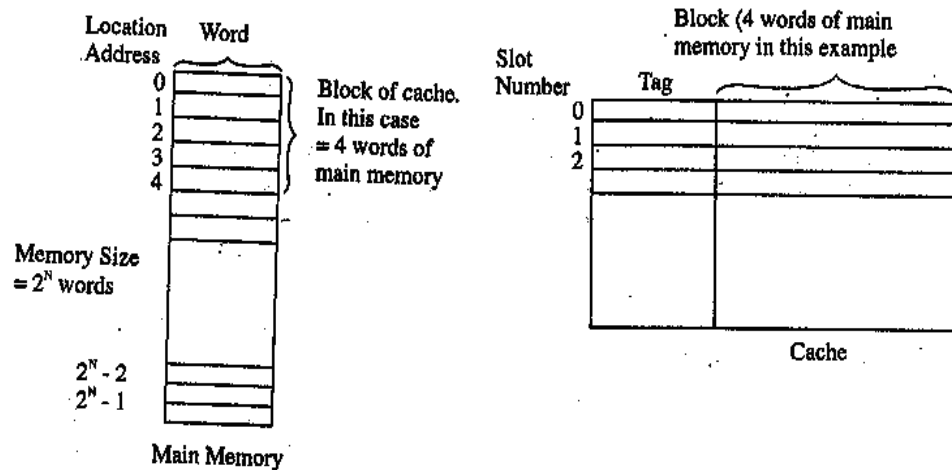


Figure 51: Cache memory organisation

A cache consists of a number of slots. As mentioned earlier, the cache size is smaller than that of the main memory; therefore, there is no possibility of one mapping of the contents of the main memory to cache. So how will the computer identify which block is residing in cache? This is achieved by storing the block address to the tag number. In the above figure four consecutive words are put in a single block. Thus if we have  $2^n$  words in the main memory then the total number of blocks which exist in the main memory are  $=2^n/4 = 2^{n-2}$  (size of one block is 4 byte). Therefore, we need at least  $(n-2)$  bits as the size of the tag field. The important feature in cache memory is how the main memory block can be mapped in cache. There are three ways of doing so: direct, associative and set associative.

**Direct Mapping:** In this mapping each block of memory is mapped in a fixed slot of cache only. For example, if a cache has four slots then the main memory blocks 0 or 4 or 8 or 12 or 16...can be found in slot 0, while 1 or 5 or 9 or 13 or 17...can be found in slot 1; 2 or 6 or 10 or 14 or 18...in slot 2; and 3 or 7 or 11 15 or 19...in slot 3. This can be mathematically defined as:

$$\text{Cache slot number} = \text{Block number of main memory} \text{ Modulo } \text{Total number of slots in cache} \text{ Operator}$$

In this technique, it can be easily determined whether a block is in cache or not. (How?)

This technique is simple, but there is a disadvantage of this scheme. Suppose two words which are referenced alternately repeatedly are falling in the same slot then the swapping of these two blocks will take place in the cache, thus, resulting in reduced efficiency of the cache. A direct mapping example is shown in Figure 52(a). Please note the mapping of main memory address to cache tag and slot. All addresses are in hexadecimal notation.

**Associative Mapping:** In associative mapping any block of the memory can be mapped on to any location of the cache. But here the main difficulty is to determine “Whether a block is in cache or not.” This process of determination is normally carried out simultaneously. The main disadvantage of this mapping is the complex circuitry required to examine all the cache slots in parallel to determine the presence or absence of a block in cache. An associative mapping example is shown in figure 52(b).

**Set Associative Mapping:** This is a compromise between the above mentioned two types of mapping. Here the advantages of both direct and associative cache can be obtained. The cache is divided in some sets, let’s say  $t$ . the scheme is that a direct mapping is used to map the main memory blocks in one of the “ $t$ ” sets and within this set any slot can be assigned to this block. Thus, we have associative mapping inside each set of the sets. Please refer to Figure 52(c) for an example.

Another important feature for cache is the replacement algorithm. For direct mapping no algorithm is needed since only one slot can be occupied by a block in cache but in associative and set associative mapping many slots may be used by a block. So which slot should be vacated for this new block? One of the common strategies used is to vacate the least recently used slot for this new block. The reason is just the probability of accessing a block, which was used quite long ago, is less in comparison to those of blocks which are used afterwards (or recently). This scheme is also derived from the principle of locality.

Other schemes, which may not be as effective as the least frequently used, can be:

- |                       |   |   |
|-----------------------|---|---|
| First in First Out    | : | Replace the block, which has entered first in cache, and free its slot for the incoming block |
| Random                | : | Replace any block at random   |
| Least Frequently used | : | Replace block, which is referenced the least number of times                                  |

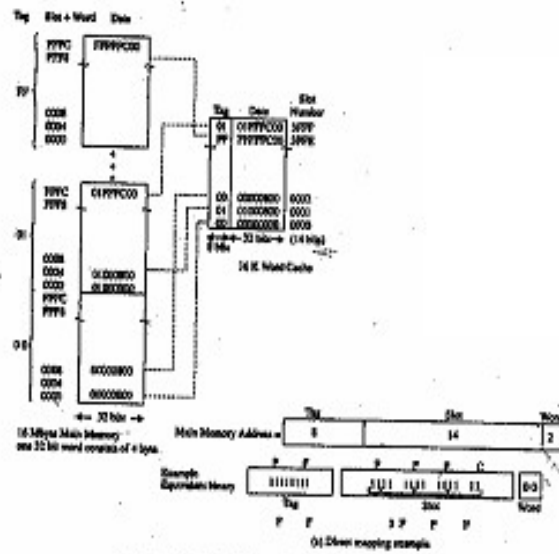
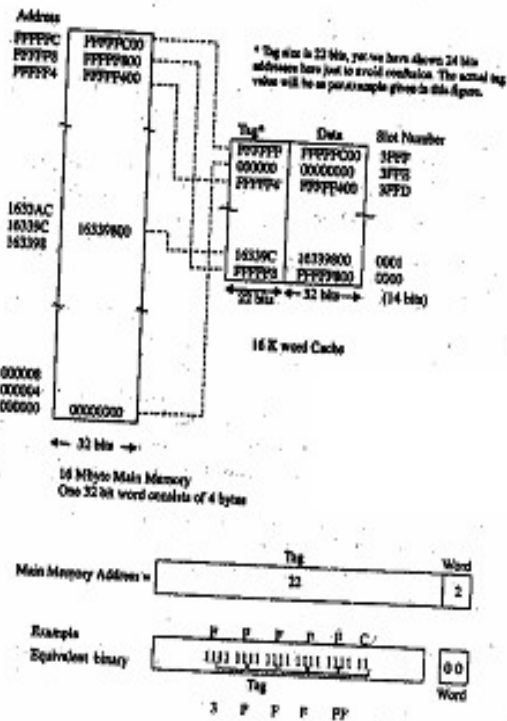


Figure 18(a): Direct mapping example



(b) Associative mapping example

Figure 52(b): An example of associative mapping

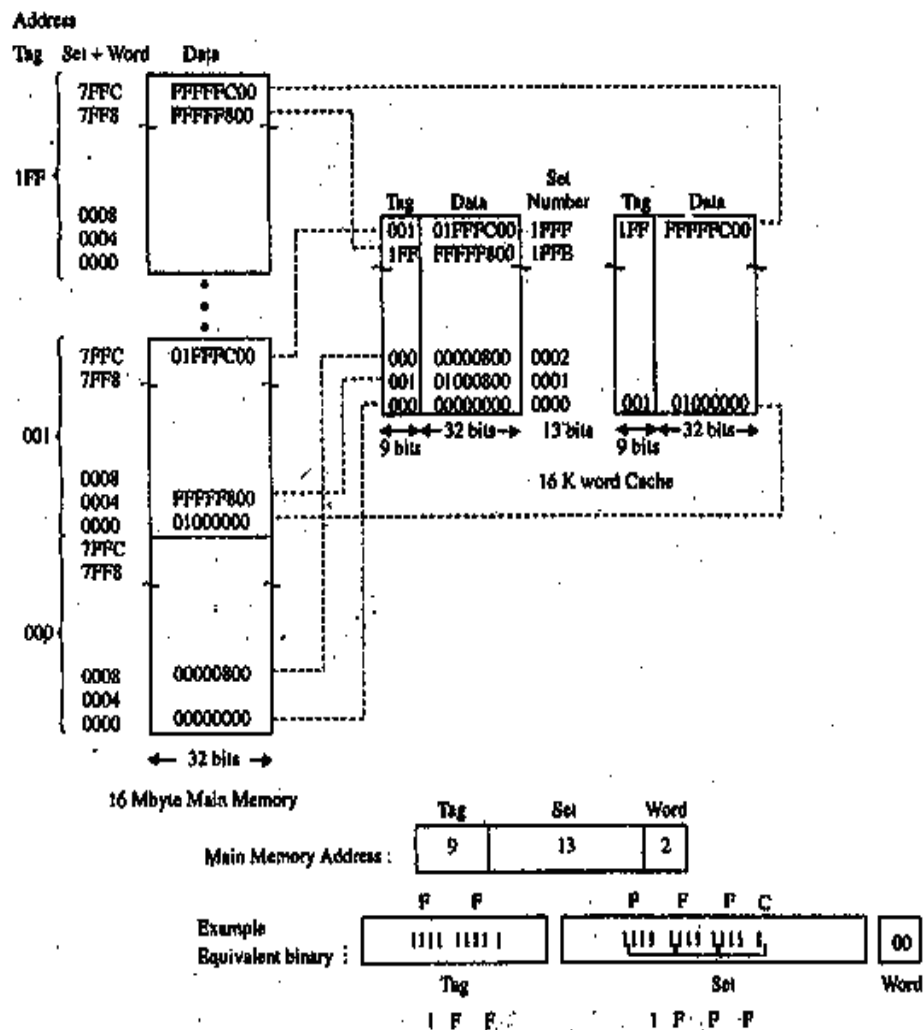


Figure 52(c): A two way set associative mapping

**Write Policy:** The data in the cache and the main memory can be written by processors or input/output devices. The main problems faced in writing with cache memories are:

1. The content of a cache and the main memory can be altered by more than one devices e.g., the CPU can write to caches and the input/output module can directly write memory. This can result in inconsistencies in the values of cache and main memory.
2. In the case of multiple CPUs with different caches a word altered in one cache automatically invalidates the word in other cache.

The suggested techniques for writing in systems with caches are:

- a) **Write through:** Write the data in cache as well as main memory. The other CPUs – cache combination (in multiprocessor system) have to watch the traffic to the main memory and make suitable

amendment in the contents of cache. The disadvantage of this technique is that a bottleneck is created due to large numbers of accesses to the main memory by various CPUs.

- b) **Write block:** In this method updates are made only in the cache, setting a bit-called update bit. Only those blocks whose update bit is set is replaced in the main memory. But here all the accesses to the main memory whether from the CPUs or input/output modules need to be from the cache resulting in complex circuitry.

Research shows that only 15% of memory references are write references and, therefore, write through policies being simple can be utilised.

- c) **Instruction Cache:** An instruction cache is the one which is employed for accessing only the instructions and nothing else. The advantage of such a cache is that; as the instructions do not change we need not write the instruction cache back to memory. This is unlike data storage cache.

### 3.5.3 Associative Memories

In associative memories any stored items can be accessed directly by assigning the contents of the item in question, such as name of a person, account number, number etc., as an address. Associative memories are also known as content addressable memories (CAMs). The entity chosen to address the memory is known as the key.

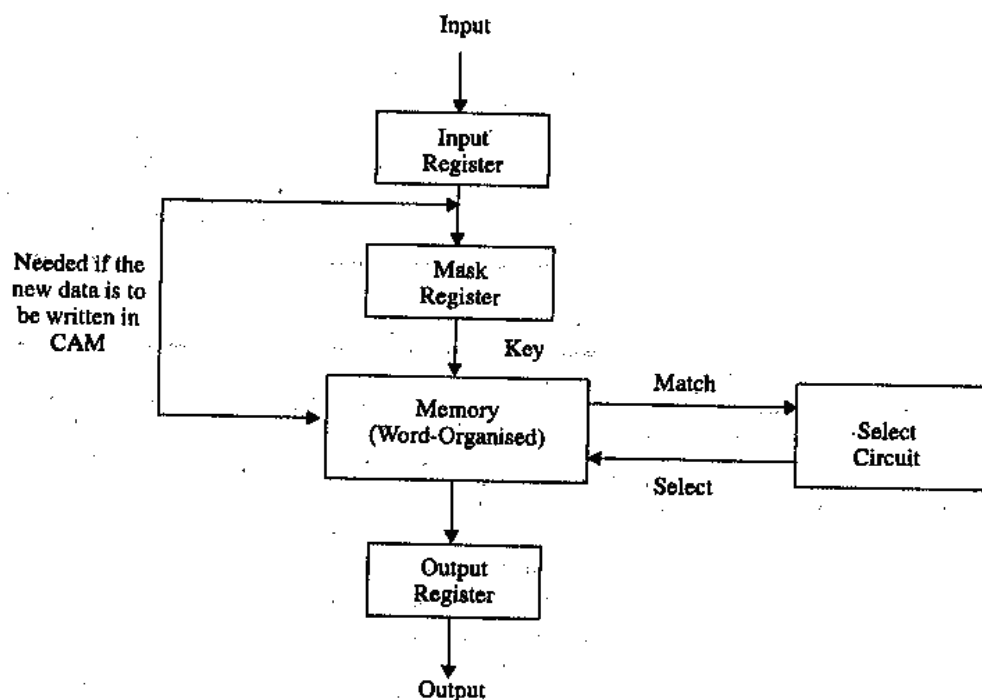


Figure 53: Structure of a simple word-organised associative memory



Figure 53 shows the structure of a simple associative memory. The information is stored in CAMs as fixed-length words. Any entity of the word may be chosen by the key field. The desired key is indicated by the mask register. Then, the key is compared simultaneously with all stored words. The words that match the key issue a match signal. This match signal then enters a select circuit. The select circuit in turn helps for the access of the required data field. In case more than one entry have the same by then it is the responsibility of the select circuit to determine the data field to be read.

For example, the select circuit read out all the matching entries in a pre-determined order. Each word is provided with its own match circuit, as all the words in the memory need to compare their keys with the desired key simultaneously. The match and select circuits thus, make these associative memories very complex and more expensive than any conventional memory. The VLSI technology has made these associative memories economically feasible. Even now the cost considerations limit the applications of associative memories to relatively small amounts of information, which need to be accessed very rapidly. An example of CAM use is in memory address mapping.

You can refer to further readings for more details in CAMs

## **4.0 CONCLUSION**

This unit has dealt with the key characteristics of the computer memory system: the various types of random access memories, external/auxiliary memories, high speed memories and their importance, the hierarchy of computer memory and the characteristics of memory technologies.

## **5.0 SUMMARY**

Thus far, we have taken a complete view of the memory system of the computer system along with the various technologies. The unit has outlined the importance of the memory system, the memory hierarchy, the main memory and its technologies, the secondary memories and their technologies and the high-speed memories.

We have also discussed the key characteristics of these memories and the technologies which are used for constructing these memories. One of the key concepts given in this unit is the data storage format of various secondary storage devices. There are several other concepts such as virtual memory; these have been taken up in the course on the system software. For more details on the memory system, you can go through the further readings.

## 6.0 TUTOR -MARKED ASSIGNMENT

1. What is the head of a disk? List the various head mechanisms.
2. Find out the average access time for a fixed head head disk rotating at 3000 rpm and contains 10 sectors in a track.
3. Match the following pairs:

CAV	Magnetic bubble memories
CLV	Magnetic tape
Low cost low speed devices	Magnetic disk
Space-borne applications	CD-ROM

4. High-speed memories are needed to bridge the gap of speed between I/O device memory True  False
5. Interleaved memory will not contribute in increasing the memory-processor bandwidth if all the accesses are aimed at the same memory module. True  False
6. Cray 1 supercomputer uses 16-way memory interleaving as it requires 16-memory access in all the instructions. True  False
7. The direct mapping of the main memory to cache is achieved by mapping a block of the main memory to any slot of cache. True  False
8. A replacement algorithm is needed only for associative and set associative mapping of cache. True  False
9. Write policy is not needed for instruction cache True  False
10. CAMs are addressed by their contents. True  False

## 7.0 REFERENCES/FURTHER READINGS

Mano, M. Morris (1993). *Computer System Architecture* (4<sup>th</sup> ed). Prentice Hall of India.

Hayes, John, P. (1988). *Computer Architecture and Organisation* (2<sup>nd</sup> ed). McGraw-Hill International.

Stallings William. *Computer Organisation and Architecture* (3<sup>rd</sup> ed). Maxwell Macmillan International Editions.

Baron, Robert J. and Higbie Lee. *Computer Architecture*. Addison-Wesley Publishing Company.

Tanenbaum, Andrew, S (1993). *Structural Computer Organisation* (3<sup>rd</sup> ed). Prentice Hall of India.

## UNIT 4 INPUT/OUTPUT ORGANISATION

### CONTENTS

- 1.0 Introduction
- 2.0 Objectives
- 3.0 Main Content
  - 3.1 Input/Output Module
    - 3.1.1 Functions of I/O Module
    - 3.1.2 Structure of I/O Module
  - 3.2 Input/output Techniques
    - 3.2.1 Programmed Input/output
    - 3.2.2 Interrupt-Driven Input/output
  - 3.3 Direct Memory Access
  - 3.4 Input/output processors
  - 3.5 External Interface
- 4.0 Conclusion
- 5.0 Summary
- 6.0 Tutor-Marked Assignment
- 7.0 References/Further Readings

### 1.0 INTRODUCTION

Till now we have discussed the various components and the memory system of a computer. We have also discussed one interconnecting structure called the Bus. In this unit we will briefly discuss input/output devices, then move on to the function and structure of an input/output module, then we will discuss input/output techniques and at the end we will discuss the input/output processors which were quite common in mainframe computers. Finally, we will examine two popular device interfaces. This unit is the last unit of the module and further discussion on the Central Processing Unit will be taken up in the next module i.e., Module-2 of the course.

### 2.0 OBJECTIVES

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

- identify the structure of the input/output module;
- describe the three types of input/output techniques, viz., programmed input/output interrupt-driven input/output and direct memory access;
- define an input-output processor; and
- identify the serial and parallel interfaces.

### **3.0 MAIN CONTENT**

#### **3.1 Input/Output Module**

The input/output module (I/O module) is normally connected to the system bus of the system on one end and one or more input/output devices on the other. Normally an I/O module can control one or more peripheral devices.

Is the I/O module merely a connector of input/output (I/O) devices to the system bus?

No, it performs additional functions such as communicating with the CPU as well as with the peripherals. But why use the I/O module at all; why not connect peripheral devices directly to the system bus? There are three main reasons for this:

1. Diversity and variety of I/O devices makes it difficult to incorporate all the peripheral devices logic (i.e. its control commands, data format etc.) into the CPU. This in turn will also reduce the flexibility of using any new development.
2. The I/O devices are normally slower than that of the memory and the CPU, therefore, do not use them on high-speed bus directly for communication purposes.
3. The data format and word length used by the peripheral may be quite different than that of the CPU.

##### **3.1.1 Functions of the I/O Module**

An I/O module is a mediator between the processor and the I/O devices. It controls the data exchange between the external devices and the main memory; or external devices and CPU registers. Therefore, an I/O module provides an interface internal to the computer which connects it to the CPU and the main memory and an interface external to the computer, connecting it to an external device or peripheral. The I/O module should not only communicate the information from the CPU to the I/O device, but it should also coordinate these two. In addition, since there are speed differences between the CPU and I/O devices, the I/O module should have facilities like buffer (storage area) and error detection mechanisms. Therefore, the functional requirements of an I/O module are:

### **1. It should be able to provide control and timing signals**

The need of I/O from various I/O devices by the CPU is quite unpredictable. In fact it depends on I/O needs of particular programs and normally does not follow any pattern, since the I/O module also shares system bus and memory for data input/output. Therefore, control and timing are needed to coordinate the flow of data from/to external devices to/from CPU or memory. A typical control cycle for transfer of data from I/O device to CPU is this:

- Enquire the status of an attached device from I/O module. The status can be busy, ready or out of order.
- I/O module responds with the status of the device.
- If the device is ready, CPU commands I/O module to transfer data from the I/O device.
- I/O module accepts data from the I/O device.
- The data is then transferred from I/O module to the CPU.

### **2. It should communicate with the CPU**

The example given above clearly specifies the need of communication between the CPU and I/O module. This communication can be: commands such as READ SECTOR, WRITE SECTOR, SEEK track number (which are issued by the CPU to the I/O module); or data (which may be required by the CPU or transferred out); or status information such as BUSY or READY or any error condition from I/O modules. The status recognition is necessary because of the speed gap between the CPU and I/O device. An I/O device might be busy in doing the I/O of previous instructions when it is needed again.

Another important communication from the CPU is the unique address of the peripheral from which I/O is expected or is to be controlled.

### **3. It should communicate with the I/O device**

Communication between the I/O module and the I/O device is needed to complete the I/O operation. This communication involves commands, status or data.

### **4. It should have a provision for data buffering**

Data buffering is quite useful for the purpose of smoothing out the gaps in speed of CPU and I/O devices. The data buffers are registers, which hold the I/O information temporarily. The I/O is performed in short bursts in which data is stored in buffer area while the device can take its own time to accept it. In I/O device to CPU transfer, data is first

transferred to the buffer and then passed on to the CPU from these buffer registers. Thus, the I/O operation does not tie up the bus for the slower I/O devices.

## **5. Error detection mechanisms should be in-built**

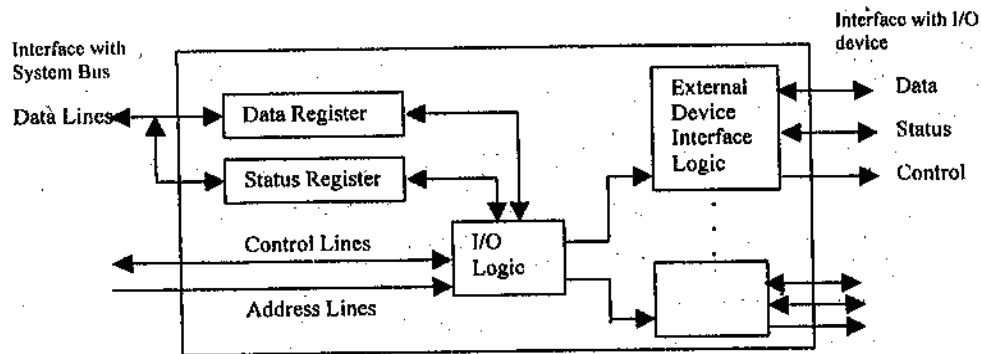
The error detection mechanisms may involve checking the mechanical as well as data communication errors. These errors should be reported to the CPU. Examples of the kind of mechanical errors that can occur in devices are paper jam in printer, mechanical failure, electrical failure, etc. data communication errors may be checked by using parity bit.

### **3.1.2 Structure of I/O Module**

Let us now focus our attention on the question. What should be the structure of an I/O module? Although, there is no standard structure of the I/O module, let us try to visualise certain general characteristics of the structure.

- There is a need for I/O logic, which should interpret and execute the dialogue between the CPU and I/O module. Therefore, there need to be control lines between the CPU and this I/O module. In addition, the address lines are needed to recognise the address of the I/O module and its specific device.
- The data lines connecting the I/O module to the system bus must exist. These lines serve the purpose of data transfer.
- Data registers may act as buffer between the CPU and the I/O module.
- The interface of the I/O module with the device should have interface logic to control the device, to get the status information and transfer of data.

Figure 54 shows the diagram of a typical I/O module which in addition to all the above have status/control registers which might be used to pass on the status information or can store control information.



**Figure 54: The general structure of the I/O Module**

The data is stored or buffered in data registers. The status registers are used to indicate the current state of a device, e.g., the device is busy, the system BUS is busy, the device is out of order etc. If an I/O module takes more processing burden then it is called an I/O channel or a processor. The primitive I/O module which requires detailed control by the processor is called the I/O controller or device controller. These I/O controllers are normally used in micro-computers while I/O processors are mainly used for mainframes because the I/O work for the micro-computer is normally limited to single user's job, therefore, we do not expect a very huge amount of I/O to justify the investment in I/O processors, which are expensive. Once we have defined a general structure for an I/O module, the next question is how actually are the I/O operations performed? The next section tries to answer this basic query in a general way.

### SELF-ASSESSMENT EXERCISE 1

State whether True or False

1. The devices are normally connected directly to system bus.  
True  False
2. The input/output module is needed for slower I/O devices  
True  False
3. Data buffering is helpful for smoothing out the speed differences between CPU and input/output devices.  
True  False
4. What is a device controller?

### 3.2 Input/Output Techniques

The input/output operations can be performed by three basic techniques. These are:

- Programmed input/output
- Interrupt-driven input/output

- Direct memory access

Figure 55 gives an overview of these three techniques.

	<b>Interrupt Required</b>	<b>I/O Module to/from Memory Transfer</b>
Programmed I/O	No	Through CPU
interrupt-driven I/O	Yes	Through CPU
DMA	Yes	Direct to memory

**Figure 55: An overview of the three input/out techniques**

In programmed I/O, the I/O operations are completely controlled by the CPU. The CPU executes programs that initiate, direct and terminate an I/O operation. It requires a little special I/O hardware, but is quite time consuming for the CPU, since the CPU has to wait for slower I/O operations.

Another technique suggested to reduce the waiting by the CPU is interrupt-driven I/O. The CPU issues the I/O command to I/O module and starts doing other work, which may be the execution of a separate program. When the I/O operation is complete, the I/O module interrupts the CPU by informing it that I/O has finished. The CPU then, may proceed with execution of this program.

In both programmed I/O and interrupt-driven I/O, the CPU is responsible for extracting data from the memory for output and storing data in the memory for input. Such a requirement does not exist in DMA where the memory can be accessed directly by the I/O module. Thus, the I/O module can store or extract data in/from the memory. We will discuss programmed I/O and interrupt-driven I/O in this section and DMA in the next section.

A situation in which the I/O is solely looked after by a separate dedicated processor is referred to as I/O channel or I/O processor. The basic advantage of these devices is that they free the CPU of the burden of input/output. Thus, during this time, the CPU can do other work, therefore, effectively increasing the CPU utilisation. We will discuss I/O channels and I/O processors in the next section.

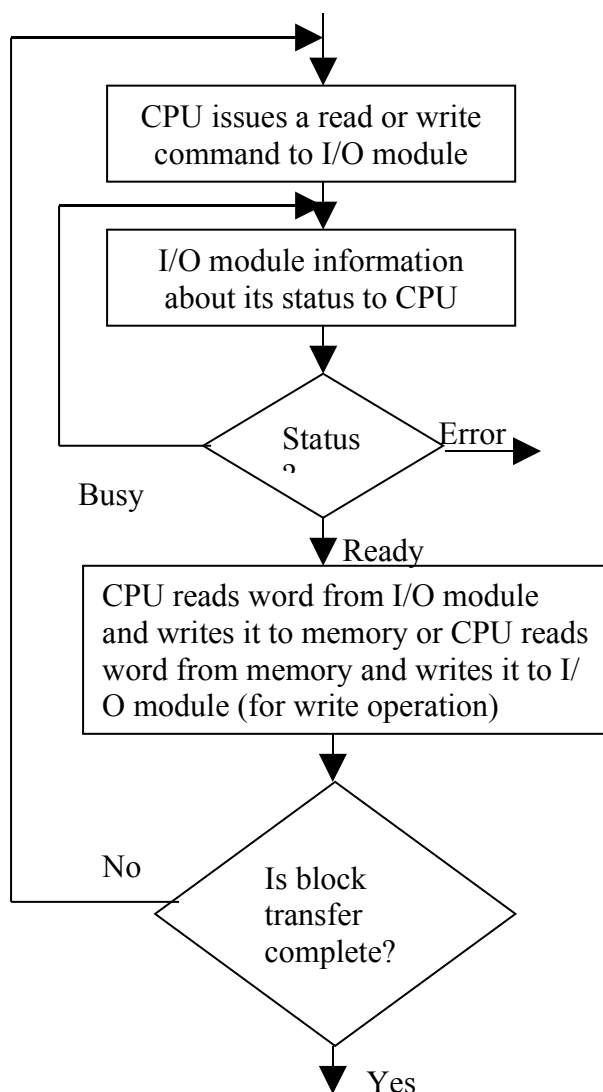
### **3.2.1 Programmed Input/Output**

Programmed input/output is a useful I/O method for computers where hardware costs need to be minimised. The input or output in such cases may involve:



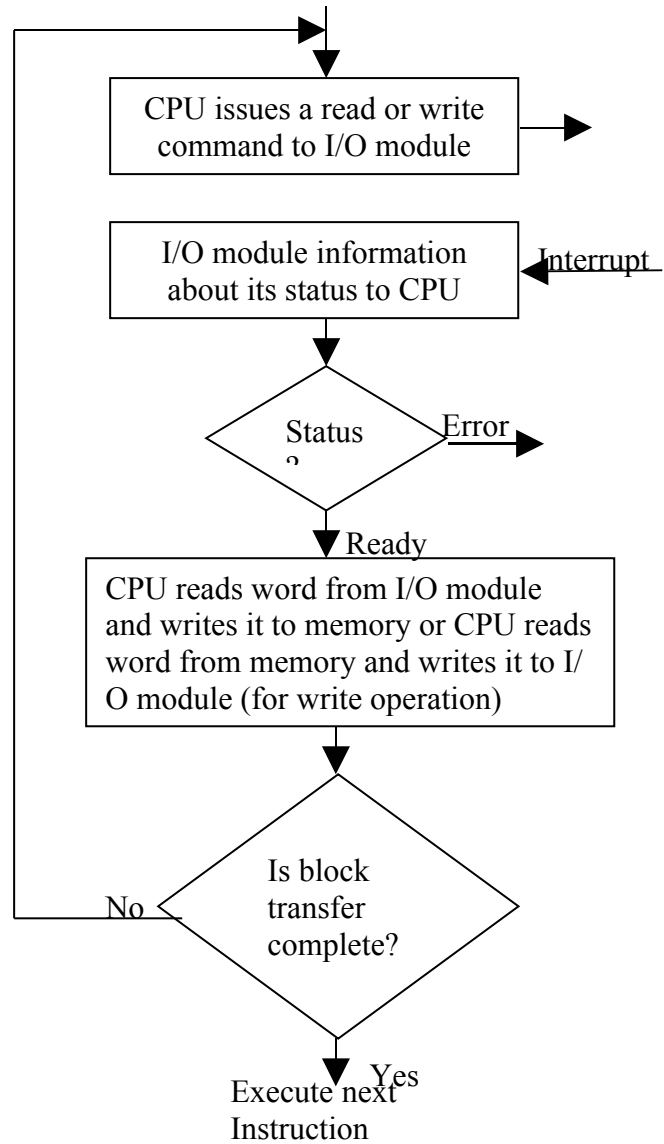
- Transfer of data from I/O device to the CPU registers.
- Transfer of data from CPU registers to memory.

In addition, in a programmed I/O method the responsibility of the CPU is to constantly check the status of the I/O device to check whether it has become free (in case output is desired) or it has finished inputting the current series of data (in case input is going on). Thus, programmed I/O is a very time consuming method where the CPU wastes lot of time for checking and verifying the status of an I/O device. Let us now try to focus how this input-output is performed. Figure 56(a) gives the block diagram of transferring a block of data word by word using programmed I/O technique.

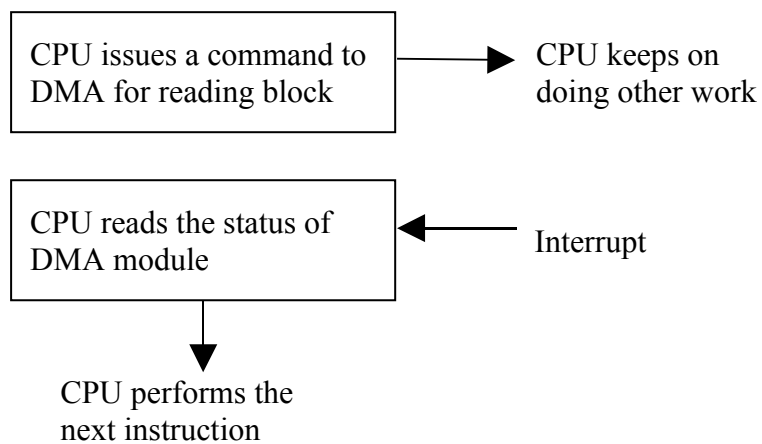


Execute next instruction

**(a) program I/O**



(b) interrupt-driven I/O



(c) DMA

Figure 56: Three techniques of input/output

**I/O Instructions:** To carry out input/output CPU issues I/O related instructions. These instructions consist of two components:

- The address of the input/output device specifying the I/O device and I/O module; and
- An input/output command.

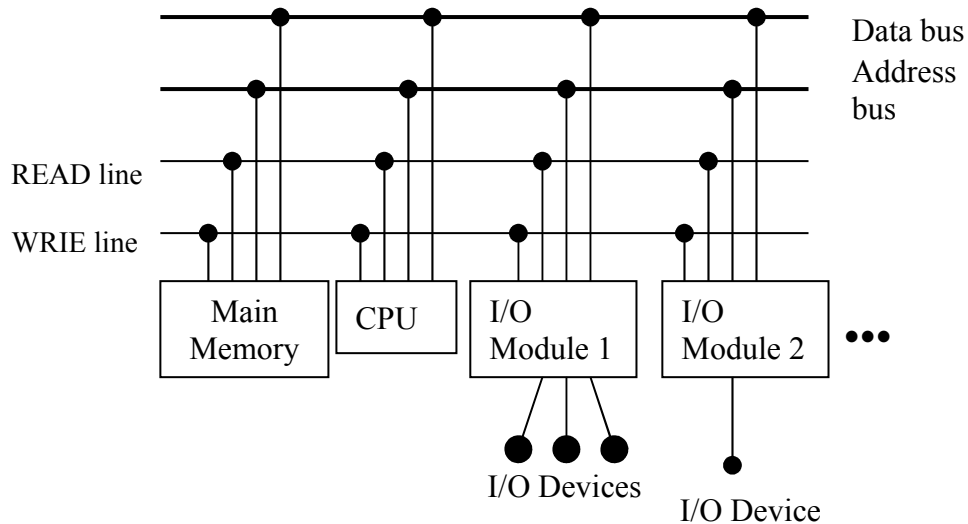
There are four types of I/O Commands, which can be classified as:

CONTROL, TEST, READ and WRITE.

CONTROL commands are specific devices and are used to control the specific instructions to the device; e.g., a magnetic tape requires rewinding or moving forward by a block. TEST command checks the status such as, if a device is ready or not or is in error condition. The READ command is used for input of data from input device and the WRITE command is used for output of data to input device.

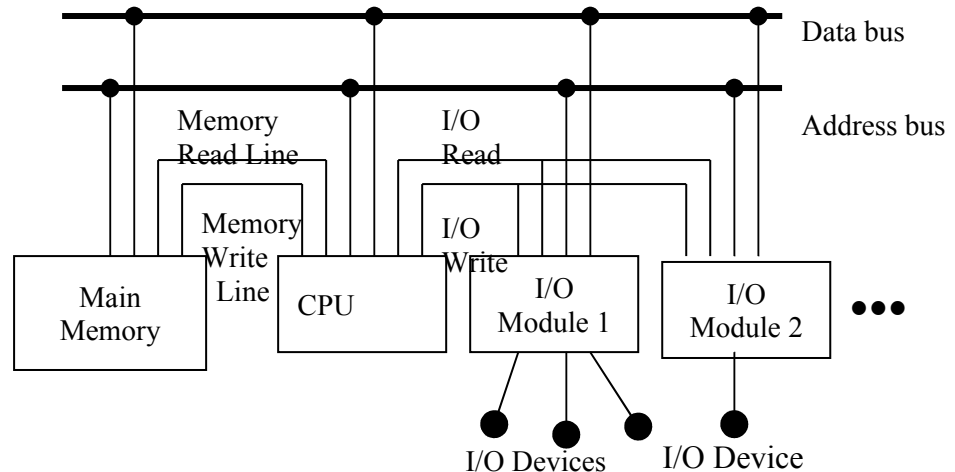
The other part of I/O instruction is the address of the I/O device. In systems with programmed I/O the I/O module, the main memory and the CPU normally share the system bus. Thus, each I/O module should interpret the address lines to determine if the command is for itself. Or in other words: How does CPU specify which device to access? There are two methods of doing so. These are called memory mapped I/O and I/O-mapped I/O.

If we use the single address space for memory locations and I/O devices, i.e., the CPU treats the status and data registers of the I/O module as memory locations, and then memory and I/O devices can be accessed using the same instructions. This is referred to as memory mapped I/O. For a memory mapped I/O, only a single READ and a single WRITE line are needed for memory or I/O module read or write operations. These lines are activated by the CPU for either memory access or I/O device access. Figure 57 shows the memory mapped I/O system structure. This scheme is used in Motorola 68000.



**Figure 57: Structure of a memory- mapped I/O**

In I/O-mapped I/O, the I/O devices and memory are addressed separately (Refer Figure 58). There are separate control lines for memory and I/O device read or write operations, thus, a memory reference instruction does not affect an I/O device. Here separate input/output instructions are needed which cause data transfer between addressed I/O module and the CPU. This structure is used in Intel 8085 & 8086 series.



**Figure 58: Structure of an I/O-mapped I/O**

Please note the difference of requirements: in the case of memory-mapped I/O the READ instruction may bring data to or from memory or I/O module, while in I/O-mapped I/O we need to have separate instructions for input/output.

### 3.2.2 Interrupt-driven Input/Output

What is the basic drawback of the programmed I/O? The speed of I/O devices is much slower in comparison to that of the CPU and because the CPU has to repeatedly check whether a device is free; or wait till the completion of I/O, the performance of CPU in programmed I/O goes down tremendously. What is the solution? What about the CPU going back to do other useful work without waiting for the I/O device to complete (what it is currently doing) or get freed up? But how will the CPU be intimated about the completion of I/O or that a device is ready for I/O? A well-designed mechanism was conceived for this, which is referred to as interrupt-driven I/O. In this mechanism, provision of interruption of CPU work, once I/O device has finished the I/O or when it is ready for the I/O, has been provided.

The interrupt-driven I/O mechanism for transferring a block of data is shown in Figure 56(b). Please note that after issuing a READ command (for input) the CPU goes off to do other useful work (it may be the execution of a different program) while I/O module proceeds for reading of data from the associated device. At the completion of an instruction cycle (already discussed in Unit 1 of this module) the CPU checks for interrupts (which will occur when data is in data register of I/O module and it now needs the CPU's attention).

Now the CPU saves the important register and processor status of the executing program in a stack and requests the I/O device to provide its data, which is placed on the data bus by the I/O device. After taking the required action with the data, the CPU can go back to the program it was executing before the interrupt.

**Interrupt:** As discussed in Unit 1, the term interrupt is loosely used for any exceptional event that causes a temporary transfer of control of the CPU from one program to the other which is causing the interrupt. Interrupts are primarily issued on:

- the initiation of an input/output operation
- the completion of an input/output operation
- the occurrence of hardware or software errors.

An interrupt can be generated by various sources, internal and external to the CPU. An interrupt generated internally by the CPU is sometimes termed "Traps". The traps are normally results of programming errors (such as division by zero) which occur during the execution of a program.

The two key issues in interrupt-driven input/output are:

- to determine the device which has issued an interrupt
- in the case of the occurrence of multiple interrupts, which one to be processed first.

There are several solutions to these problems. The simplest of them is to provide multiple interrupt lines, which will result in the immediate recognition of the interrupting device. The priorities can be assigned to various interrupts and the interrupt with the highest priority should be selected for service in case multiple interrupt occurs. But providing multiple interrupt lines is an impractical approach because only a few lines of the system bus can be devoted for the interrupt. Other methods for this are software poll, daisy chaining, etc.

**Software Poll:** In this scheme, on the occurrence of an interrupt, the CPU starts executing a software routine known as interrupt service program or routine which polls to each I/O module to determine which I/O module has caused the interrupt. This may be achieved by reading the status register of the I/O modules. The priority here can be implemented easily by defining the polling sequence, since the device polled first will have higher priority. Please note that after identifying the device, the next set of instructions to be executed will be the device service routines of that device, resulting in the desired input or output.

As far as **daisy chaining** is concerned, we have one interrupt acknowledge line, which is chained through various interrupt devices. (The mechanism is similar, as we have discussed in Unit 2). There is just one interrupt request line. On receiving an interrupt request, the interrupt acknowledge line is activated which in turn passes this signal device by device. The first device which has made the interrupt request thus grasps the signal and responds by putting a word which is normally an address of the interrupt servicing program or a unique identifier on the data lines. This word is also referred to as interrupt vector. This address or identifier in turn is used for selecting an appropriate interrupt-servicing program. The daisy chaining has an in-built priority scheme, which is determined by the sequence of devices on the interrupt acknowledge line.

**In bus arbitration technique**, the I/O module first needs to control the bus and only after that can it request for an interrupt. In this scheme, since only one of the modules can control the bus, only one request can be made at a time. The interrupt request is acknowledge by the CPU in response to which the I/O module places the interrupt vector on the data lines. An interrupt vector normally contains the address of the interrupt servicing program.

You can refer to further readings for more details on some typical interrupt structures of interrupt controllers.

### 3.3 Direct Memory Access (DMA)

When a large amount of data is to be transferred from the CPU, a DMA module can be used. But why? In both interrupt-driven and programmed I/O the CPU is tied up in executing input/output instructions while DMA acts as if it has taken over control from the CPU. The DMA operates in the following way:

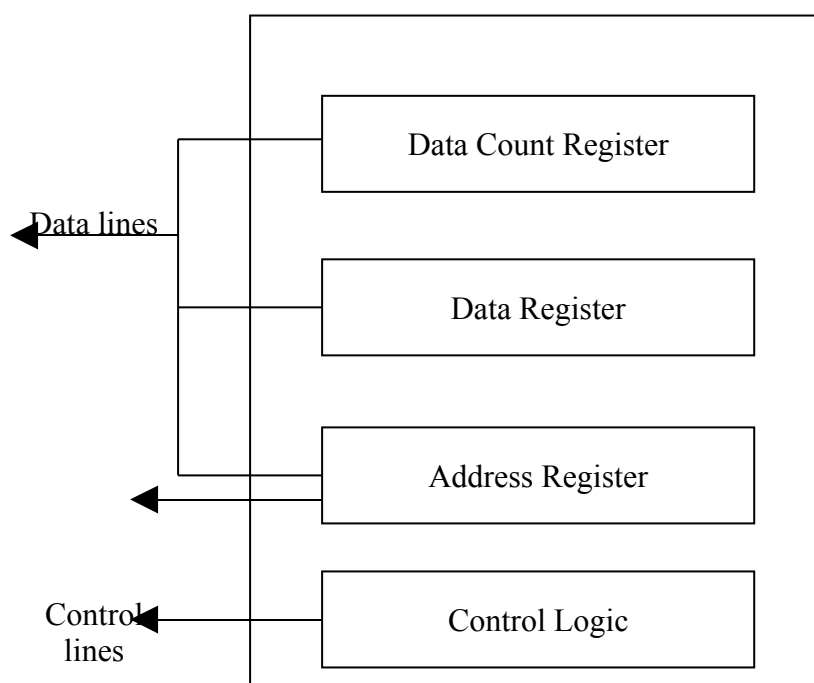
When an I/O is requested, the CPU instructs the DMA module about the operation by providing the information:

- Which operation to be performed (read or write).
- The address of the I/O device which is to be used.
- The starting location on the memory where the information will be read or written to the number of words to be written or to be read.

The DMA module transfers the requested block byte by byte directly to the memory without intervening the CPU.

On completion of the request the DMA module sends an interrupt signal to the CPU.

Thus, in DMA the CPU involvement can be restricted at the beginning and end of the transfer. But what does the CPU do while the DMA is doing input/output? It may execute another program or it may be another part of the same program. Figure 59 shows registers of a general DMA module. Please note that it contains additional registers for counting the data bytes and also note that address register and data count registers are fed with the data lines.

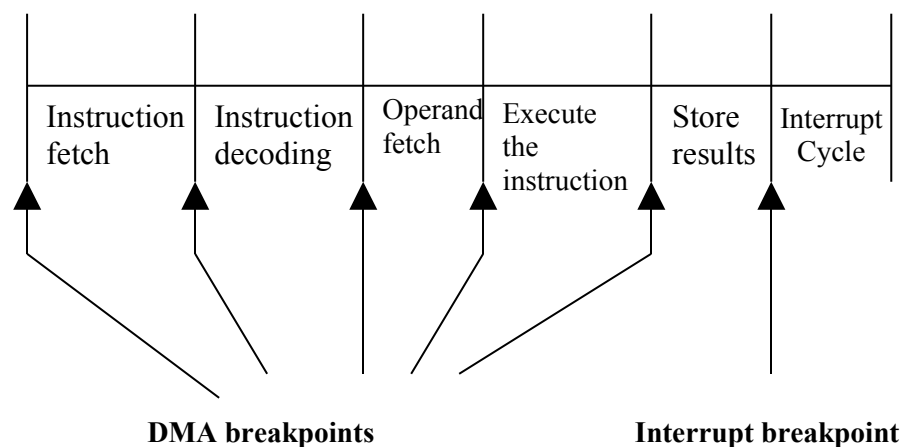


Address lines

**Figure 59: The registers on the DMA module**

Let us now see how this DMA mechanism works. Since the DMA module shares the system bus, it needs to have some way to take control of the bus such that the data is transferred to/from memory from/to the DMA module. A DMA module transfers an entire block of data or one word at a time directly to/from memory. But when should the DMA take control of the bus?

For this we will recall the phenomena of the execution of an instruction by the CPU. Figure 60 shows the five cycles for an instruction execution. The figure also shows the five points where a DMA request can be responded to and a point where the interrupt request can be responded to. Please note that an interrupt request is acknowledged only at one point of an instruction cycle.



**Figure 60: DMA and interrupt breakpoints**



Let us now discuss the data transfer modes of DMA. The first mode is the block transfer of data after taking control of the bus. The CPU may not use the bus during this time. In such a mode a complete block of data, for example a complete sector in a disk is transferred in a single continuous burst. During this transfer, the DMA controller/module remains master of the bus. Such a transfer is quite useful in cases where we have fast secondary memory such as the magnetic disk where a delay of one or two pulses may result in a delay by a rotation, thus, the burst of data is preferred through the DMA. The drawback in such a scheme is that the CPU has to wait for the bus for quite some time.

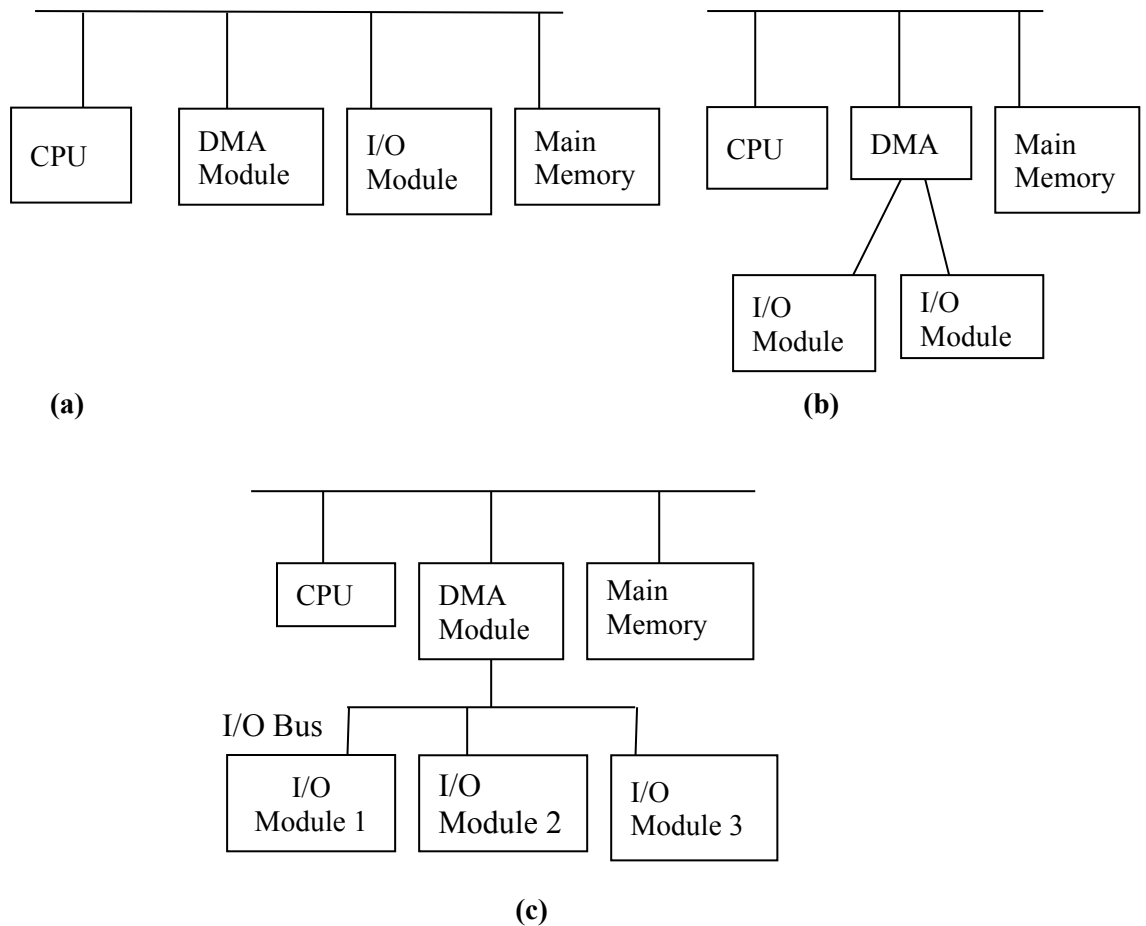
Alternatively, instead of transferring a complete block, a few words or a single word is transferred through the system bus to the memory through the DMA. In this mode the DMA forces the CPU to suspend its operations temporarily. After this transfer the control is returned to the CPU. This technique is called cycle stealing. In this scheme, although the rate of I/O by the DMA goes down but on the other hand it reduces the interference caused by the DMA to the CPU operation.

It is possible to minimise this interference of the CPU by the DMA controller such that the cycles are stolen only when the CPU is not using system bus. This is called a transparent DMA.

Finally, let us find out how the DMA can be configured., There are many ways in which this can be done; we will discuss some of them.

The simplest possibility is to allow the DMA, I/O and all the modules to share the system bus. This structure is shown in Figure 61(a). In this kind of configuration the DMA may act as a supportive processor and can use programmed I/O for exchanging data between the memory and I/O module through DMA module. But once again this spoils the basic advantage of the DMA in not using extra cycles for transferring information from memory to/from the DMA and the DMA from/to the I/O module.

The Figure 61(b) configuration suggests clear-cut advantages over the one shown in Figure 61(a). In these systems a path is provided between the I/O module and the DMA module, which does not include the system bus. The DMA logic may become a part of an I/O module and can control one or more I/O modules. In an extended concept an I/O bus can be connected to this DMA module. Such a configuration (shown in Figure 61(c) is quite flexible and can be extended very easily. In both these configurations the added advantage is that the data between the I/O module and the DMA module is transferred off the system bus. Thus, eliminating the disadvantage we have witnessed in the first configuration.



**Figure 61: Some of the DMA configurations**

### 3.4 Input/Output Processors

Before discussing I/O processors, let us briefly recapitulate the development in the area of input/output functions. These can be summarised as:

- Step 1:** Direct control of CPU on I/O device. Limited number of I/O devices
- Step 2:** Introduction of I/O controller or I/O module. Use of programmed I/O. CPU was separated from the details of external I/O interfaces.
- Step 3:** Contained use of I/O controllers but with interrupts. CPU need not wait for I/O operation to complete (increased efficiency)
- Step 4:** Direct access of I/O module to the memory via DMA. CPU involvement reduced at the beginning and at the end of DMA operation.

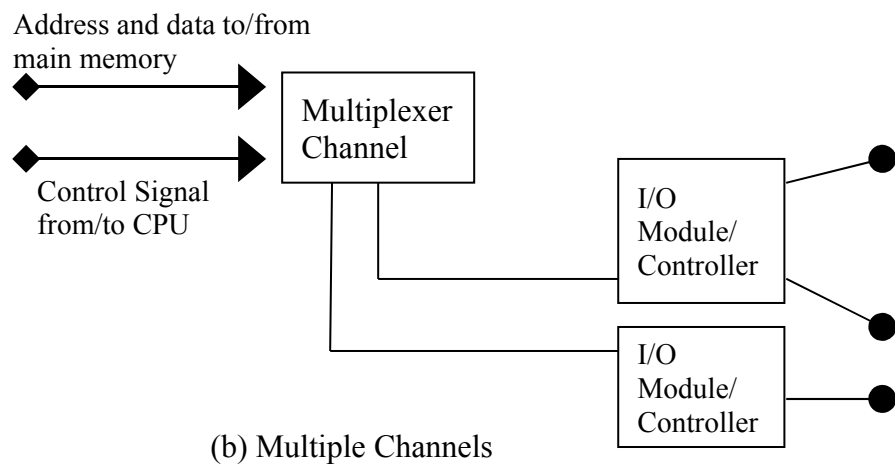
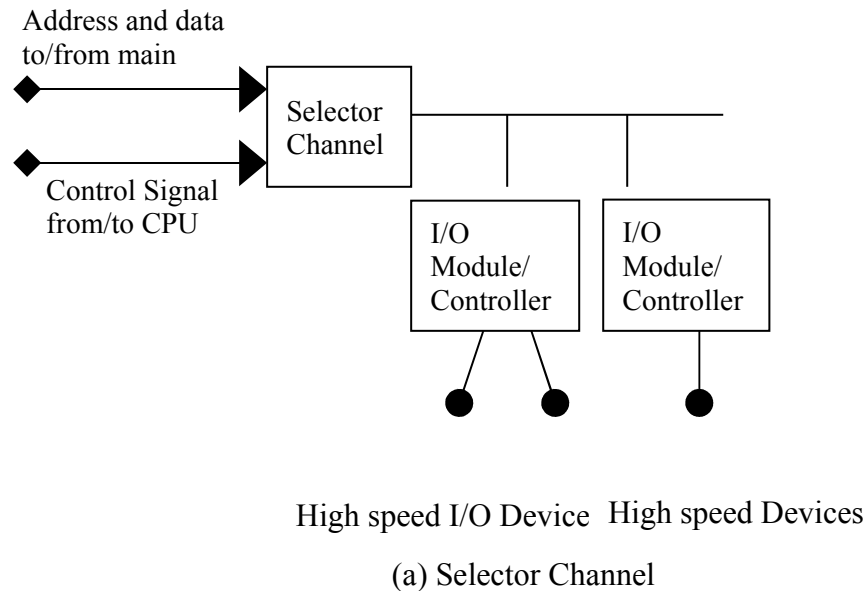
The concept of an I/O processor is an extension of the concept of a DMA. The I/O processor can execute the specialised I/O programs residing in the memory without the intervention of the CPU. Thus, the CPU only needs to specify a sequence of I/O activity to the I/O processor. The I/O processor then executes the necessary I/O instructions which are required for the task; and interrupt the CPU only after the entire sequence of I/O activity as specified by CPU has been completed. An advanced I/O processor can have its own memory, enabling a large set of I/O devices to be controlled without much involvement from the CPU. Thus, an I/O processor has the additional ability to execute I/O instructions, which provide a complete control on I/O operations. Thus, the I/O processor is much more powerful than the DMA which provides only a limited control of I/O device. For example, if an I/O device is busy then the DMA will only interrupt the CPU and will inform the CPU again when the device is free while the I/O processor's responsibility will be to keep on checking the status of the I/O device and once it has found it to be free go ahead with I/O and when I/O finishes, communicate it to the CPU.

The communication between I/O processor and CPU can be achieved by writing a message in the memory area shared by the two processors. The CPU instructs the I/O processor to execute an I/O program in the memory. The program will then specify the device or devices and the area of the memory where the I/O data is stored or to be stored. In addition, this program also contains the actions that are to be taken in case of errors, or what priority is to be given to various I/O devices.

In a computer system which has IOPs the CPU normally does not execute the I/O data transfer instruction. I/O instructions are stored in the memory and are executed by IOPs. The IOP can be provided with the direct access to the memory and can control the system bus. An IOP can execute a sequence of data transfer instructions involving different memory regions and different devices without the intervention of the CPU. The I/O processor is known as the channel in IBM machines.

Later on several other computers used the term "channel" also. The earlier channels did not have any memory but the present channels may have large cache memory, which may be used for data buffering. The Control Data Corporation (CDC) computers and some other computers use relatively sophisticated I/O systems. These are called peripheral processing units (PPUs). These PPU's also perform the job of the I/O processor. PPU's in themselves are complete, simple computers with their own memory. In addition to I/O they are capable of performing some additional data manipulations, which include data formatting, character translation and buffering.

Let us now discuss two common types of I/O channels. For high-speed devices a selector channel is used. This channel can transfer data from one high-speed device at a time. I/O modules can in turn handle each of these high-speed devices. Thus, we effectively have an I/O processor taking the place of CPU in controlling various I/O modules. Figure 62 (a) shows the selector channel.



**Figure 62: Architecture for input/output channel**

The second type of channel is a multiplexer channel, which can handle input/output with a number of devices at the same time. If the devices are slow then byte multiplexing is used. Let us explain this with an example. If we have three slow devices which need to send individual bytes as:

B <sub>1</sub>	B <sub>2</sub>	B <sub>3</sub>	B <sub>4</sub>	B <sub>5</sub>	...
Y <sub>1</sub>	Y <sub>2</sub>	Y <sub>3</sub>	Y <sub>4</sub>	Y <sub>5</sub>	...
T <sub>1</sub>	T <sub>2</sub>	T <sub>3</sub>	T <sub>4</sub>	T <sub>5</sub>	...

Then on a byte multiplexer channel they may send the bytes as B<sub>1</sub> Y<sub>1</sub> T<sub>1</sub> B<sub>2</sub> Y<sub>2</sub> T<sub>2</sub> B<sub>3</sub> Y<sub>3</sub> T<sub>3</sub> .....for high-speed device blocks of data from several devices is interleaved. These are called block multiplexer channels.

We are not including the example of an input/output processor here but you are advised to look into these examples after you have completed first two modules, from the further readings.

### 3.5 External Interface

Our discussion on I/O system will not be complete if we do not discuss external interfaces. The external interface is the interface between the I/O module and the peripheral devices. This interface can be divided into two main categories: parallel interface, and serial interface.

In parallel interface multiple bits can be transferred simultaneously. The parallel interface is normally used for high-speed peripherals such as tapes and disks. The dialogues that take place across the interface include the exchange of control information and data. A common parallel interface is centronics.

In serial interface only one line is used to transmit data, therefore, only one bit is transferred at a time. Serial interface is used for serial printers and terminals. The most common serial interface is RS-232C. A new standard, which is becoming very popular and allows multiple devices to be connected, is Universal Synchronous Bus (USB). It is an industrial standard today.

Irrespective of the type of interface the I/O module has to communicate with the peripheral in the following way for a read or write operation.

- A control signal is sent by the I/O module to the peripheral requesting the permission to send (for write operation) or receive (for read operation) data.
- The peripheral device acknowledges this request for data transfer.
- The data is transferred from the I/O module to peripheral (for write) or from peripheral to I/O module (for read).

- The acknowledgement of receipt of data is issued by the receiver of data.

Both serial and parallel transmission can be in two modes-- synchronous and asynchronous. In the synchronous mode several characters are transmitted in a single transmission while in asynchronous mode only few bits are transmitted at a time.

#### 4.0 CONCLUSION

Here, you have been introduced to the structure of input/output module and the three types of input/output techniques, viz: programmed input/output, interrupt-driven input/output and direct memory access. You have also been introduced to the input/output processor and the external (serial and parallel) interfaces and you should be able to describe these.

#### 5.0 SUMMARY

This unit is the last unit of this module. In this module we have covered the three major points, which include I/O devices, memory and interconnection structure. The last of the components i.e., the CPU will be discussed in Module 2. This unit was devoted mainly towards I/O of computer systems. We have discussed the I/O module, I/O techniques, external I/O interfaces, etc. The design details of these have been covered in this unit. For details on the design aspect you can refer to the further readings.

#### 6.0 TUTOR- MARKED ASSIGNMENT

State whether True or False

1. I/O mapped I/O scheme require no additional line from CPU to I/O device except for system bus. True  False
2. Memory mapped I/O scheme uses a separate instruction for data transfer from/to memory; and from/to I/O module. True  False
3. The advantages of interrupt-driven I/O over programmed I/O is that in interrupt-driven I/O the interrupt mechanisms free I/O devices quickly. True  False
4. In the transparent DMA the cycles are stolen only when the CPU is not using the bus. True  False

5. Most of the I/O processors have their own memory while a DMA module does not have its own memory except for register or a simple buffer area. True  False
6. Parallel interfaces are commonly used for connecting printers to a computer. True  False

## 7.0 REFERENCES/FURTHER READINGS

Mano, M. Morris (1993). *Computer System Architecture* (3<sup>rd</sup> ed) Prentice Hall of India,

Hayes, John P. (1988). *Computer Architecture and Organisation* (2<sup>nd</sup> ed) McGraw-Hill International Editions.

Stallings William. *Computer Organisation and Architecture* (3<sup>rd</sup> ed). Maxwell Macmillan International Editions.

Baron, Robert J. and Higbie Lee. *Computer Architecture*. Addison-Wesley Publishing Company.

Tanenbaum, Andrew S. (1993). *Structural Computer Organisation* (3<sup>rd</sup> ed). Prentice Hall of India.

## MODULE 2

Unit 1	Instruction Sets
Unit 2	Register Organisation and Micro-Operations
Unit 3	ALU and Control Unit Organisation
Unit 4	Microprogrammed Control Unit

### UNIT 1 INSTRUCTION SETS

#### CONTENTS

1.0	Introduction
2.0	Objectives
3.0	Main Content
3.1	Instruction Set Characteristics
3.1.1	Operand Data Types
3.1.2	Number of Addresses in an Instruction
3.1.3	Operation Types
3.2	Addressing Schemes
3.2.1	Immediate Addressing
3.2.2	Direct Addressing
3.2.3	Indirect Addressing
3.2.4	Register Addressing
3.2.5	Register Indirect Addressing
3.2.6	Displacement Addressing
3.2.7	Stack Addressing Scheme
3.3	Instruction Format Design
3.3.1	Instruction Length
3.3.2	Allocation of Bits
3.3.3	Variable Length of Instructions
3.4	Examples of Instruction Sets
4.0	Conclusion
5.0	Summary
6.0	Tutor-Marked Assignment
7.0	References/Further Readings

#### 1.0 INTRODUCTION

In the previous module, we discussed the structure of the computer and data representation. One term which we have commonly used is the instruction. In this respect, a few questions which still need to be answered are: what is an instruction? What are its components? How is the instruction executed by the CPU? This unit is an attempt to answer the first two questions, while, the third question is a complex one and is explained in the later units. In this unit we will discuss in detail the



instructions, their types, and the operands. We will also discuss the various addressing schemes which are popular among various PC's and how the effective address is calculated for these schemes. In addition we are also trying to highlight the basic design issues related to the instruction in the unit. We have presented here the instruction set of IBM System/370 as an example. However, you can study the details on the other instruction sets- for example the VAX machine- from the further readings. We have not included discussions on the instruction set of popular INTEL microprocessor.

## **2.0 OBJECTIVES**

At the end of this unit a student should be able to:

- discuss various elements of an instruction;
- distinguish various types of instructions;
- differentiate various types of operands;
- define a classification of computers on the basis of the number of addresses in instructions sets;
- discuss various operations which are performed by the instructions;
- identify various addressing schemes;
- calculate effective addresses for various schemes; and
- discuss the instruction format design characteristics.

## **3.0 MAIN CONTENT**

### **3.1 Instruction Set Characteristics**

Till now, we have discussed instruction in an abstract way. Now let us discuss in detail various characteristics of instructions. However we will first of all find out the significance of the instructions set. One thing which should be kept in mind is that the instruction set is a boundary which is looked upon in the same fashion by a computer designer and the programmer. From the computer designer's point of view, the instruction set provides the functional requirements of the CPU. In fact, for implementing the CPU design, one of the main tasks is to implement the instruction set for that CPU. However, from the user's point of view machine instructions or assembly instructions are needed for low level programming. In addition, a user should also be aware of registers, the data types supported by the machine and the functioning of the ALU. We will discuss the registers in Unit 2 and the ALU in Unit 3 of this module.

Explanation on the machine instruction set gives extensive details about the CPU of a machine. In fact, the operations which a CPU can perform

can be determined by the machine instructions. Now, let us answer some introductory questions about the instruction set.

### What is an instruction set?

An **instruction set** is a collection of all the *instructions* a CPU can execute. Therefore, if we define all the instructions of a computer, we can say we have defined the instruction set. Each instruction consists of several elements. An instruction element is a unit of information required by the CPU for execution.

### What are the elements of an instruction?

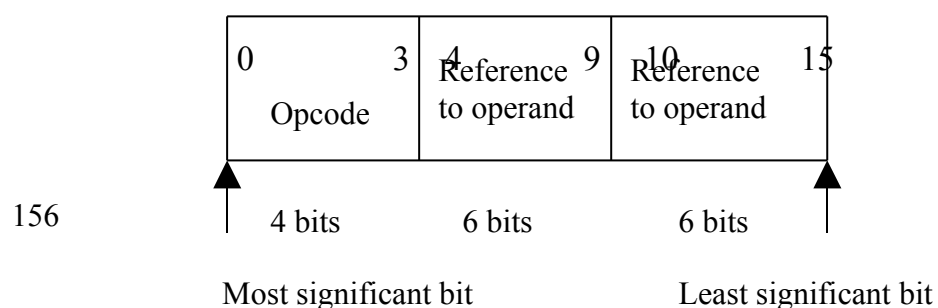
- An operation code, also termed an **opcode** which specifies the operation to be performed
- A reference to the operands on which data processing is to be performed. For example, an address of an operand
- A reference to the operands which may store the results of data processing operations performed by the instruction.
- A reference for the next instruction, to be fetched and executed.

The next instruction which is to be executed is normally the next instruction following the current instruction in the memory. Therefore, no explicit reference to the next instruction is provided. What if we do not want this normal flow of execution? You will find the answer to this question in this unit.

An important aspect, of the references to operands and results is: where are those operands located? In the memory or in the CPU registers or in the I/O device. If the operands are located in the registers then an instruction can be executed faster than that of the operands located in the memory. The main reason here is that memory access time is higher in comparison to the register access time.

### How is an instruction represented?

Instructions are represented as sequence of bits. An instruction is divided into a number of fields. Each of these fields corresponds to a constituent element of instruction. A layout of instruction is known as **instruction format**. For example, the following is the instruction format for an IAS computer. It uses four bits for **opcode** and only two operand references are provided here. No explicit reference is provided for the next instruction to be executed.



**Figure63: A sample instruction format**

In most instruction sets, many instruction formats are used. An instruction is first read into an instruction register (IR), and then the CPU decodes the instruction and extracts the required operands on the basis of references made through the instruction fields, and processes it. Since the binary representation of the instruction is difficult to comprehend and is seldom used for representations, we will be using symbolic representations to these instructions in this unit along with the comments wherever desired.

**What are the types of instructions?**

The instructions can be categorised under the following:

- **Data Processing Instructions:** These instructions are used for arithmetical and logic operations in a machine. Examples of data processing instructions are: Arithmetic, Boolean, shift, character and string processing instructions, stack and register, manipulation instructions, vector instructions, etc.
- **Data Storage/Retrieval Instructions:** Since the data processing operations are normally performed on the data stored in CPU registers, we need instructions to bring data to and from memory to registers. These are called data storage/retrieval instructions. Examples of data storage and retrieval instructions are load and store instructions.
- **Data Movement Instructions:** These are basically input/output instructions. They are required to bring in programs and data from various devices to memory or to communicate the results to the input/output devices. Some of these instructions can be: start input/output, halt input/output, TEST input/output etc.
- **Control Instructions:** These instructions are used for testing the status of computation through Processor Status Word (PSW). This register will be discussed in greater details in unit 2. Another of such

instruction is the branch instruction used for transfer of control. We will discuss in more details about these instructions.

- **Miscellaneous Instructions:** These instructions do not fit in any of the above categories. Some of these instructions are: interrupt or supervisory call, swapping, return from interrupt, halt instruction or some privileged instruction of operating systems.

### **What are the factors which play important roles for selection/designing of instruction sets for a machine?**

Instruction set design is the most complex yet interesting and very much analysed aspect of computer design. The instruction set plays an important role in the design of the CPU as it defines many functions of it. Since instruction sets are the means by which a programmer can control the CPU, therefore, users' views must be considered while designing the instruction set. Some of the important design issues relating to instruction design are:

- How many and what operations should be provided?
- What are the operand data types to be provided?
- What should be the instruction format? This includes issues like: instruction length, number of address, length of various elements of instructions, etc.
- What is the number of registers which can be referenced by an instruction and how are they used?
- What are the modes of specifying an operand address?

We will try to analyse these issues in some detail in this and subsequent sections. However, we have kept the level of discussion very general. A special example of an instruction set is given at the end of this unit.

#### **3.1.1 Operand Data Types**

An operand data type specifies the type of data on which a particular operation can be performed. For example, for an arithmetical operation, numbers are to be used as data types. In general the operands which can be used in an instruction can be categorised into four general categories. These are:

- Addresses
- Numbers
- Characters
- Logical data

**Addresses:** Addresses are treated as a form of data which is used in the calculation of actual physical memory address of an operand. In most of

the cases, the addresses provided in instruction are operand references and not the actual physical memory addresses.

**Numbers:** All machines provide numeric data types. One special feature of numbers used in computers is that they are limited in magnitude, and hence the underflow and overflow may occur during arithmetical operations on these numbers. The maximum and minimum magnitude is fixed for an integer number while a limit of precision of numbers and exponent exist in the floating point numbers. The three numeric data types which are common in computers are:

- Fixed point numbers or Integers (signed or unsigned)
- Floating point numbers
- Decimal numbers

All the machines provide instructions for performing arithmetical operations on fixed point and floating point numbers. Many machines provide arithmetical instructions which perform operations on packed decimal digits.

**Characters:** Another very common data type is the character or string of characters. The most widely used character representation is ASCII (American National Standard Code of Information Interchange). It has 7 bits for coding data pattern which implies 128 different characters. Some of these characters are control characters which may be used in data communication. The eighth bit of ASCII may be used as a parity bit. One special mention about ASCII which facilitates the conversion of a 7 bit ASCII and a 4 bit packed decimal number is that the last four digits of ASCII number are binary equivalents of digits 0-9.

That is

Decimal	Binary	ASCII
0	0000	011 0000
1	0001	011 0001
2	0010	011 0010
3	0011	011 0011
⋮	⋮	⋮
⋮	⋮	⋮
9	1001	011 1001

**Figure 64: Decimal digits in ASCII**

The other important code is Extended Binary Coded Decimal Interchange Code (EBCDIC). This is an 8-bit code and is compatible

with packed decimal in a similar way as that of ASCII. The digits 0 through 9 in this can be represented as 1111 0000 through 1111 1001.

**Logical Data:** In general a data word or any other addressable unit such as byte, half word etc. are treated as a single unit of data. But can we consider an n-bit data unit consisting of n items of 1 bit each? If we treat each bit of an n-bit data as an item then it can be considered to be logical data. Each of these n items can have a value 0 or 1.

What are the advantages of such a bit oriented view of data? The advantages of such a view will be:

- We can store an array of Boolean or binary data items most efficiently.
- We will be in a position to manipulate the bits of any data item.

But where do we need to manipulate bits of a data item? The example of such a case is shifting of significance bits in a floating point operation or for converting ASCII to packed decimals where we need only the 4 rights most bits of ASCII's byte.

Please note that for extracting decimal from ASCII, first the data is treated as logical data and then it can be used in arithmetical operations as numeric data. Thus, the operation performed on a unit of data determines the types of the unit of data at that instance. This statemetn may be true for high level language, but holds good for machine level language.

### SELF-ASSESSMENT EXERCISE 1

State whether True or False

1. An instruction set is meant only for the programmer and is not needed at the time of the implementation of a machine  
True  False
2. Explicit operand references are a must for an instruction?  
True  False
3. You can use only one instruction format for an instruction set of a machine.  
True  False
4. Data movement instructions are used for bringing in data from the memory to CPU registers. True  False
5. Numbers represented in computers are limited in magnitude.  
True  False

6. A data value in a machine language can be treated as of one type only. True False

### 3.1.2 Number of Addresses in an Instruction

The fewer number of addresses in an instruction lead to reduced length of instructions; however, it also limits the range of functions that can be performed by the instructions. In a sense this implies that a machine instruction set having less number of addresses have longer programs, which means longer execution time. However, more addresses may lead to more complex decoding and processing circuits.

Most of the instructions do not require more than three operand addresses. In instructions having fewer addresses than three, normally some of the operand locations are implicitly defined. Many computers have a range of instructions of different lengths and number of addresses. The following table gives an example of zero, one, two and three address instruction along with their interpretations.

Number of Addresses	Instruction	Interpretation
3	ADD A, B, C	Operation $A=B+C$ is executed
2	ADD A,B	Two plausible interpretations (i) $AC = A + B$ (ii) $A = A + B$ . In this case the original content of operand location A is lost
1	ADD A	$AC = AC + A$ A is added to accumulator
0	ADD	Top of stack contains the addition to top two values of the stack.

- AC is an accumulator register.  
A, B, C are operand locations.

**Figure 65: Examples of zero, one, two and three address instructions**

The register architecture, that is a general classification which is based on register set of the computer, is sometimes classified according to the number of addresses in instructions. These classifications are:

**Evaluation-Stack Architecture:** These machines are Zero address machines and their operands are taken from top of the stack implicitly. The ALU of such machines directly references a stack which can be implemented in main memory or registers or both. These machines contain instructions like PUSH and POP to load a value on stack and store a value in the memory respectively. Please note that PUSH, POP are not zero address instructions but contain one address.

The main advantages of such architecture are:

- Very short instructions
- Since stacks are normally made within CPU in such an architecture machine, the operands are close to the ALU, thus effecting fast execution of instructions
- Excellent support for subroutines.

While the main disadvantages of this architecture are:

- Not very general in nature, in comparison to other architectures
- Difficult to program for applications like text and string processing.

One example of such a machine is Burroughs B6700. However, these machines are not very common today because of the general nature of machines desired. The stack machine uses Polish notations for evaluation of arithmetical expressions. Polish notation was introduced by a Polish logician, Jan Lukasiewicz. The main theme of this notation is that an arithmetical expression  $AxB$  can be expressed as:

either	$xAB$	(Prefix notation)
or	$ABx$	(Suffix or reverse polish notation)

In stacks we use suffix or reverse polish notation for evaluating expression. The rule here is to push all the variable values on the top of stack and do the operation with the top elements of stack as an operand is encountered. The priority of operand is kept in mind for generation of reverse polish notation. For example, an expression  $AxB+CxD/E$  will be represented in reverse polish notation as:

$$ABxCDxE/+$$

Thus, the execution program for evaluation of  $F=AxB+CxD/E$  will be:

```
PUSH A    /Transfer the value of A on to the top of stack/
PUSH B    /Transfer the value of B on to the top of stack/
```



MULT	/Multiply. It will remove the value of A and B from the stack and multiply A x B/
PUSH C	/Transfer C on the top of stack/
PUSH D	/Transfer D on the top of stack/
MULT	/Remove the values of C & D from the stack and multiply A x D/
PUSH E	/Transfer E on the top of stack/
DIV	/C x D/E /
ADD	/Add the top two values on the stack/
POP F	/Transfer the results back to memory location F/

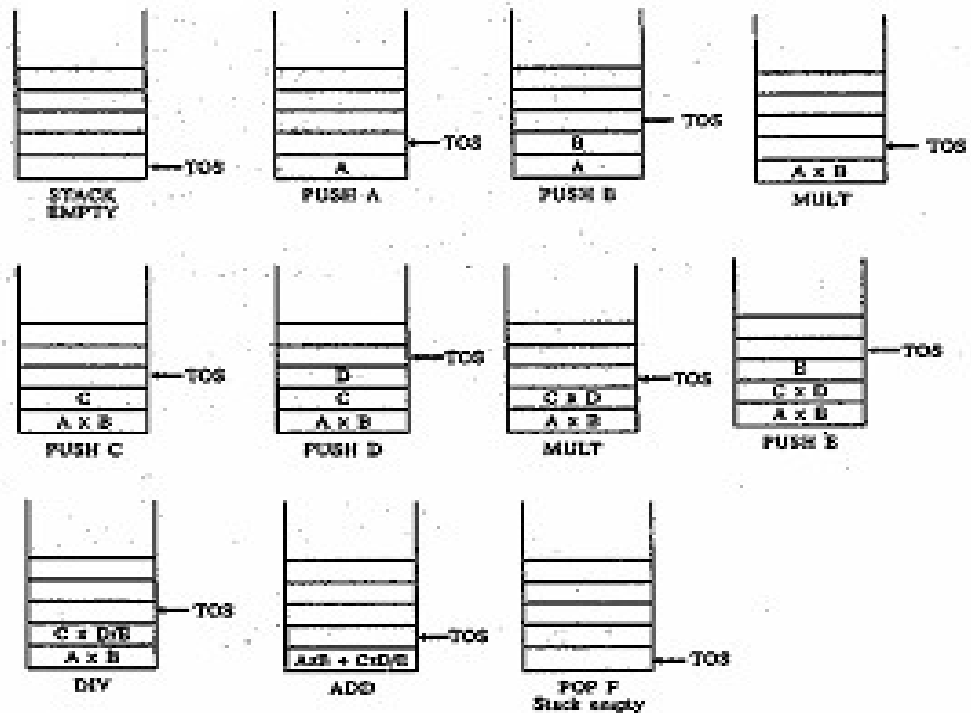
**Figure 66: Sample program for evaluating  $AxB+CxD/E$  using zero address instructions.**

Please note that PUSH and POP are not zero-address instructions.

The execution of the above program using stack is represented diagrammatically in figure 67.

**Accumulator Machines:** These machines contain a special register called accumulator which holds the results of arithmetical, logical or shift operations. The accumulator is an implicit address. The instructions in such machines are normally one-address instructions. The basic advantage of the one address machines is that the instructions are shorter than 2 and 3 address machines. However, the performances of these machines are somewhat slow because the machines require frequent memory accesses. These machines are made by various vendors.

For example, a program for evaluating the expression  $F=AxB+CxD/E$  written in this accumulator based machine is given in figure 68.



Assumption: TOS is top of stack is pointing to an empty location

**Figure 67:** Evaluation using a stack

LOAD	A	/Transfer A to Accumulator/
MULT	B	/Multiply Accumulator with B, keep the result of/ /multiplication in B/
STORE	T	/Store the intermediate result temporarily in a location T/
LOAD	C	/Transfer C to Accumulator/
MULT	D	/Accumulator = Accumulator X D/
DIV	E	/Accumulator = Accumulator/E /
ADD	T	/Accumulator = Accumulator + T/
STORE	F	/Store the Accumulator value to location F/

**Figure 68:** Sample program for evaluating  $A \times B \times C \times D / E$  using one address instruction.

**General Purpose Register Set Machines:** Many computers have been designed to have sets of registers known as general purpose registers. These general purpose register machines normally have multiple address instructions. In these machines any of the registers of the register set can be used as accumulator or an address register or an index register or a stack pointer, or even (in some cases) as a program counter. In such machines, each instruction specifies itself about the use of registers. Some examples of such computers are IBM system 370 & Digital VAX families.

The instructions in such machines can specify as many register operands as desired. Therefore, several operations can be performed solely on registers. This may increase the program execution speed as register references are faster than memory references. For the purpose of flexibility and the ability to use multiple registers, these computers may use two and three address instructions. Some of these computers have instructions for register to memory transfer, and memory to memory transfer. Since numbers of registers are few, these instructions are normally shorter.

The program for  $F = CxB + CxD/E$  for 2 address and 3 address machines will look like:

MOVE	T, A	/Move T A/		MULT	F, A, B	/ F=AxB /
MULT	T, B	/Multiply T=TxB /		MULT	T, C, D	/ T=CxD /
MOVE	F, C	/ Move F C /		DIV	T, T, E	/ T=T/E /
MULT	F, D	/ Multiply F=Fx D /		ADD	F, F, T	/ F=F+T /
DIV	F, E	/Divide F=F/E /				
ADD	F, T	/Add F=F+T/				

**Figure 69 Sample program for evaluating  $F=B+CxD/E$  using two and three instructions.**

**Special-Purpose Register Set Machines:** The main problem with this type of machine is too much of generality. In several machines, sets of registers may be used for only special purposes. For example, one set of registers may be used as index registers, another set for holding arithmetic, operands etc. Computers like CDC6000/7000 family fall under this category.

Many a machines fall in between the above mentioned two categories. They have some special purpose and some general purpose register sets. These machines are nowadays quite popular

### 3.1.3 Operation Types

Different Computers use a wide variety of **opcodes** and number of addresses. Some of the operations which are specified on one machine may not exist on a second machine. However, certain categories of operations exist on all the machines. We will try to provide details on some of the typical categories of operation. Broadly the operations specified in instructions, irrespective of the number of addresses in an instruction, can be categorised as:

- Data transfer operations
- Arithmetical operations

- Logical and shift operations
- Input/output operations
- Conversion operations
- System control operations
- Transfer of control operations

**Data Transfer Operations:** This is the most fundamental type of operation. A data transfer instruction needs to furnish the following information:

- The location of source and destination operands (This could be memory or register or stack-top). This will be clearer after you go through the next subsection.
- The length of data transfer
- The mode of address for each operand. These address modes are discussed in greater details in the next section.

Some of the common data transfer operations are:

<b>Operation</b>	<b>Description</b>
MOVE or TRANSFER	Transfers a word or a block of data from source to destination.
STORE	Transfers a word from the processor to a specified location in the main memory.
LOAD or FETCH	Brings a word from a location of the main memory to the processor.
EXCHANGE	Exchange the contents of the source with the destination.
CLEAR or RESET	Transfers a word containing all 0's to the destination.
SET	Transfers a word containing all 1's to the destination.
PUSH	Transfers a word from a source to top of stack.
POP	Transfers a word from the top of stack to a destination.

**Figure 70: Common data transfer operations**

For all these instructions the choice of source or destination can be a location in the main memory or a register or the top of stack. This

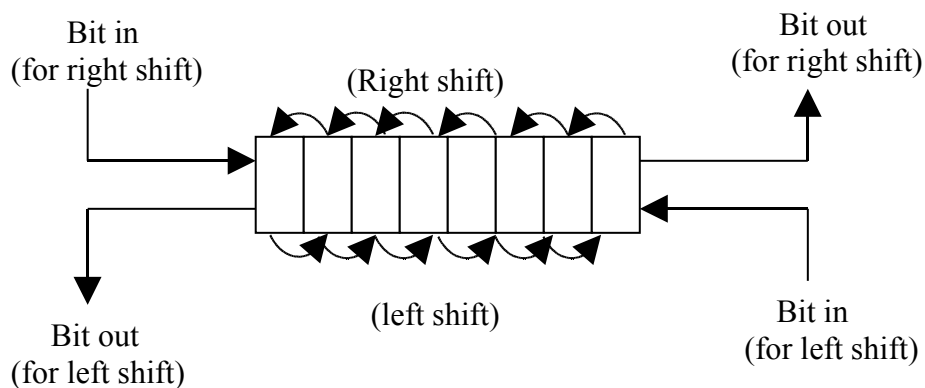
location can be indicated either in **opcode** or in the specification of an operand.

**Arithmetical Operations:** Almost all the machines provide the four basic arithmetical operations on signed fixed point integers. Some machines provide operations on floating point and packed decimal numbers. Some other arithmetical operations such as absolute of a number, negation of a number, incrementing or decrementing a number are also included as instructions in several machines. The execution of an arithmetical instruction requires bringing in the operands in the operational registers such that the data can be processed by the ALU. The arithmetical operations which can be provided in a machine in general are:

ADD, SUBTRACT, MULTIPLY, DIVIDE  
ABSOLUTE, NEGATE, INCREMENT, DECREMENT

**Logical and Shift Operations:** The logical operations are based on Boolean operations performed on binary data. Some of the logical operations are: AND, OR, NOT, Exclusive-OR.

In addition to these bitwise logical operations, machines provide a variety of shifting operations. A shift operation is performed either to the left or to the right. In a shift operation, all the bits move towards the left or right as desired. (Please refer to Figure71).



**Figure71: Shift operations schematic for an 8 bit register**

The following are some observations about shift operations.

- In logical *left/right shift* the “bit in” is a 0 bit
- An *Arithmetical shift* is the same as logical shift except for the sign bit which is not shifted
- A *circular shift* uses the “bit out” bit as the “bit in” bit.

The main usages of these shifts are:

- The logical shift is used in extracting fields from a word. How?
- The arithmetical shift if performed on numbers represented in signed 2's complement notation cause multiplication by 2 or division by 2, depending on left or right arithmetical, shift, provided there is no overflow or underflow.
- The circular shift preserves all the bits.

**Conversion Operations:** The conversion operations are needed to convert the format of the data. For example changing format from decimal to binary or ASCII to EBCDIC or vice versa. In general the two conversion operations are:

TRANSLATE	This instruction translates a given piece of data depending on a table of correspondence.
CONVERT	It converts the contents of a word from one format to the other.

Let us give an example of each of these. A common example for TRANSLATE instruction is conversion of ASCII to EBCDIC. This can be achieved by creating a table which is 256 byte long. We can call it as an array of EBCDIC equivalent values. Each index of this array represents the ASCII value, while the content of that location is the equivalent EBCDIC. For example, if 00110011 in ASCII is equivalent to 11001100 then

ARRAY location 00110011 contains 11001100  
That is TRANS [00110011] = 11001100

However, the conversion instruction converts the format based on certain rules; for example, decimal to binary.

**Input/Output Operations:** There is a lot of variety as far as input/output operations are connected as they depend on the type of input/output such as programmed I/O, DMA, etc. we have given some I/O operations in the unit 4 of Block 1. However, some of the common input/output instructions are:

READ (or INPUT)	This command is used for transferring data from input/output module or part to a destination which may be the main memory or processor register.
WRITE(or OUTPUT)	This command transfers data from a specified source to input/output module.
TEST I/O	It transfers the input/output systems status information to a specified destination.

**System Control Operations:** The system control operations generally come under the category of privileged instructions, that is, these instructions are executed only when the processor is in certain privileged state or the processor is executing a program which is stored in a special privileged area of memory. In general, these instructions are used by the operating system. A typical system control instruction is OSCALL. This instruction causes the interruption of execution of current program and passes the control to the operating system.

**Transfer of Control Operations:** In general, in a program execution the next instruction in sequence is executed next. However, in certain cases such as looping, decision-making and subroutine call, the next instruction to be executed may not be the next instruction in sequence. The instructions that disturb the normal flow of instruction execution are called transfer of control instructions. The most common transfer of control instructions which are found in instruction sets are:

- Branch
- Skip
- Subroutine Call

**Branch Instruction:** A branch instruction causes a jump to the new instruction to be executed. A branch instruction is also known as jump instruction. This instruction has one operand, that is, the address of the instruction to which branch is desired. The branch instruction, in general, is used as a conditional branch instruction; that is the branch is made only if a specified condition is satisfied, otherwise the next instruction in the normal sequence is executed.

But how is the condition tested? Most of the machines provided a 1-bit or a multiple-bit conditional code, which in certain cases can be treated as a user visible register (we will discuss user visible registers more in Unit 2 of this module). For example, a typical machine performing on an arithmetical operation can set a 2 bit condition code to either zero, positive, negative or overflow condition. On such a machine, we can have the conditional jump instructions as:

Conditional Code (C.C)	Meaning	Instruction	Meaning
00	Resultant positive	is BRP X	Branch to memory location X if the result is positive
01	Resultant negative	is BRN X	Branch to memory location X if the resultant is negative.
10	Resultant zero	is BRZ X	Branch to X if resultant is Zero.
11	Overflow occurred	has BRO X	Branch to X overflow has occurred.

All these branches may be implemented by assigning a program counter (PC) to the address of the location to which the branch is desired.

**Figure 72: An example of conditional branch instructions**

As the changes in condition code may take place on execution of each instruction, the branch instruction will depend on the most recent instructions which have modified the condition code. All the above four branches take place when a condition, is fulfilled, otherwise the next instruction in sequence is executed. Another type of conditional branch instruction which has the condition in itself can be devised for a three address instruction. For example:

BUN R1, R2, C (Branch to a memory location address “C” if the content of R1 is not equal to content of R2)

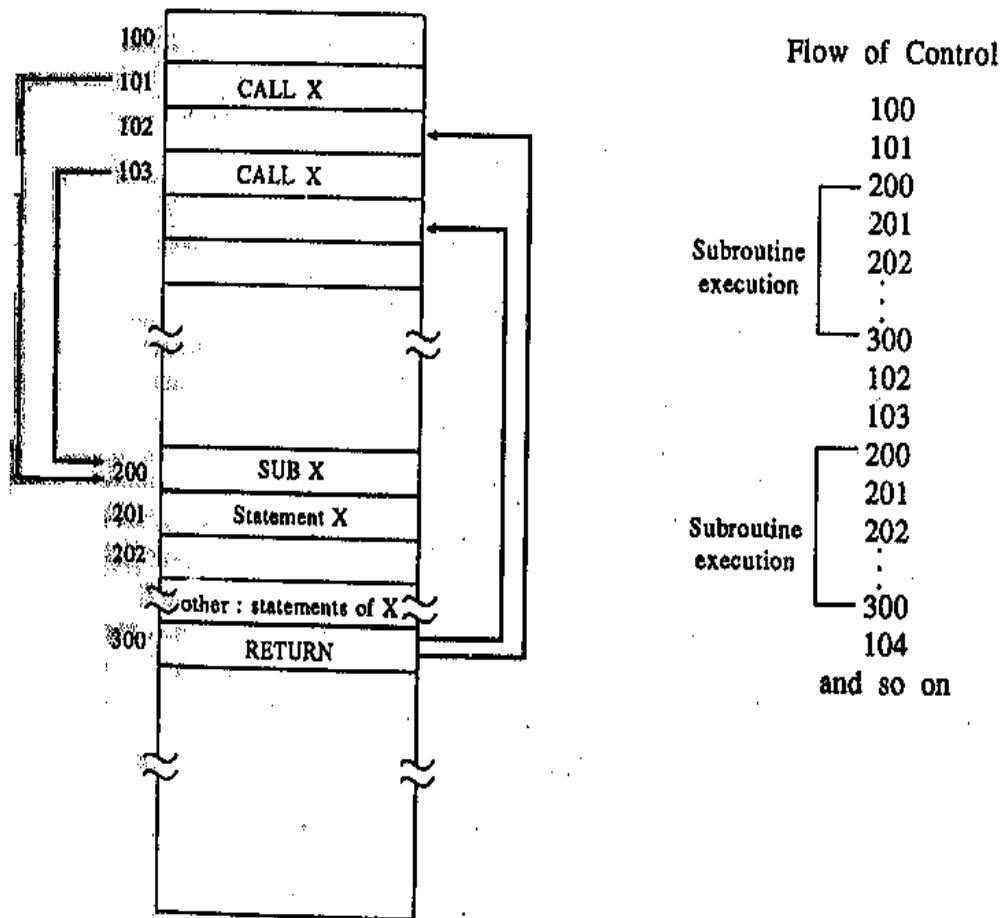
Here the condition is tested and the branch is determined in a single instruction. Please note that the branch can take place to a higher address or to a lower address

**Skip Instruction:** This instruction skips the next instruction to be executed in sequence. In other words this instruction increments the address of “next instruction to be executed” (in many computers it is the program counter) by one instruction length. The skip instruction can also be used with conditions; for example, ISZ instruction skips the next instruction only if the condition code indicates that the resultant of the most recent operation is zero. This instruction along with the branch instruction is used for implementing looping structures.

**Subroutine Call:** A subroutine is a self contained user program which contains the code often used repeatedly in a large program. This program is incorporated in a bigger program.



A subroutine is called explicitly by a program statement. This is explained in the following figure.



Assumption for the present flow of control: No transfer of control instruction in the subroutine.

**Figure 73: A subroutine call**

Two most important instructions related to the subroutine call are CALL and RETURN statements. CALL causes jump to the first instruction of subroutine, however, this jump must remember from where it has started as RETURN brings the control back to the instruction following the CALL instruction. The logic of subroutine call is similar to that of interrupt processing. The subroutine call is implemented in many cases by storing the contents of the program counter (PC) which in fact is the pointer to the return address.

But where do we store the return address? In general, the return address can be stored in a register or memory location specifically used for this purpose. In such a system the following steps will be followed on encountering a subroutine call.

- Store the content of PC in a predetermined memory location or register. Let us, call this register or memory location “R”
- Transfer the starting address of the subroutine which has been called in the PC.
- Execute the subroutine
- On encountering RETURN statement of this subroutine load the contents of R into the PC.

A second approach can be to store the return address at the start of the subroutine. For example, a call instruction CALL X will initiate the following steps:

- Store the contents of PC into memory location X.
- Place address X+1 in the PC, as subroutine statements start from this memory location only, as the Xth location is reserved for storing the return address
- Execute the subroutine
- On encountering RETURN statement, load the contents stored in memory location X to the PC.

However, these two approaches are not valid when more than one person wants to execute the same subroutine simultaneously, and when a subroutine calls itself. The reason is that in both cases, the location where the return address is stored will be rewritten by the new calling address, in turn, canceling the previously stored address.

Thus, a very general approach utilising stacks is used for subroutine call. The return address on a subroutine call is pushed on the top of the stack. On encountering a RETURN statement the POP operation is used to retrieve the most recent return address. This method also works for recursive subroutine calls.

Another important aspect of the subroutine call is the parameter passing. With a subroutine call, in general, parameters are passed. There are three approaches for parameter passing:

- Parameter passing through registers
- Parameter passing through memory location just after the call instruction location. The return address in such a scheme should be the memory location after these memory-based parameters.

- Use of stacks for parameter passing in addition to return address.

The drawback of the first approach is that proper utilisation of registers is to be ensured by the calling program. The second method fails in cases where variable numbers of parameters are to be passed. The third approach is quite general in nature. Stacks are used for not only storing the return address but are also used for storing the parameters which are to be passed to the subroutine which is called.

## SELF-ASSESSMENT EXERCISE 2

- Match the following pairs:
 

(a) Zero address instruction	(i) General purpose register set
(b) One address instruction	(ii) Stacks
(c) Two address instruction	(iii) Accumulator machine
- What are the advantages and disadvantages of evaluation-stack architecture?
- Match the following:
 

(i) MOVE	(a) Data transfer operation
(ii) WRITE	(b) I/O operation
(iii) LOAD	(c) Conversion operation
(iv) READ	(d) System control operation
(v) TRANSLATE	
- How is a subroutine call different from branching?
- What are the three methods for calling a subroutine?

### 3.2 Addressing Schemes

As discussed earlier, the main function of a computer is to execute instructions. An instruction contains an operation code (opcode) and operand(s) in coded form. The opcode field specifies the operation to be carried out and the operand(s) field specifies the location of the operand(s) in the memory. An operand may be specified as the part of the instruction, or the reference of the memory location where the value stored may be given. The term “addressing scheme” refers to the mechanism employed for specifying operands. The arrangement of opcodes and operands and their numbers within the instructions determines the form or the format of an instruction. We will discuss addressing formats more in the next section.

The choice of addressing schemes and instruction formats are governed by the efficiency, (time as well as space) economy and programming

flexibility considerations. All the available machines in the market today, therefore, employ more than one addressing scheme. In the subsequent sections, we describe the most common addressing schemes and show how the contents of operand (address) are mapped to the physical memory location where the operand being addressed is actually stored.

In the description that follows, the symbols  $A$ ,  $A_1$ ,  $A_2$ ,.....etc. denote the content of an operand field. Thus  $A_1$  may refer to a data or a memory address. In case the operand field is a register address, then the symbols  $R$ ,  $R_1$ ,  $R_2$ ,..... etc., are used. If  $C$  denotes the contents (either of an operand field or a register or of a memory location), the  $(C)$  denotes the content of the memory location whose address is  $C$ .

*The symbol EA (Effective Address) refers to a physical address in a non-virtual memory environment and refers to a register in a virtual memory address environment. This register address is then mapped to a physical memory address. What is a virtual address? Von Neumann had suggested that the execution of a program is possible only if the program and data reside in the memory. In such a situation the program length along with data and other space needed for execution cannot exceed the total memory. However, it was found that at the time of execution, the complete portion of data and instruction is not needed as most of the time only few areas of program are being referenced. Keeping this in mind a new idea was put forward where only a required portion is kept in the memory while the rest of the program and data reside in secondary storage. The data or program portion which is stored on secondary storage is brought to memory whenever needed and the portion of memory which is not needed is returned to the secondary storage. Thus, a program with a size bigger than the actual physical memory can be executed on that machine. This is called virtual memory.*

The typicality of virtual addresses is that:

- They are longer than the physical addresses as total addressed memory in virtual memory is more than the actual physical memory.
- If a virtual addressed operand is not in the memory then the operating system brings that operand to the memory.

The symbols  $D$ ,  $D_1$ ,  $D_2$ ... etc refer to actual operands to be used by instructions for their execution. Now let us examine the various addressing schemes.

### 3.2.1 Immediate Addressing

Under this addressing scheme, the actual operand D is A, the content of the operand field: i.e.

$$D = A$$

This addressing mode is used to initialise the value of a variable. The advantage of this mode is that no additional memory accesses are required for executing the instruction. However, as the size of instruction and operand field are limited, the type of data specified under this addressing scheme also gets restricted.

### 3.2.2 Direct Addressing

Under this addressing scheme, the content A of the operand field specifies EA, the effective address of the operand: i.e.

$$\begin{aligned} EA &= A, \text{ and} \\ D &= (EA) \end{aligned}$$

The second statement implies that the data is stored in the memory location specified by an effective address. In this addressing scheme only one memory reference is required. This simple addressing scheme provides a limited address space. If the address field has n bits then the address space available is  $2^n$  memory locations.

### 3.2.3 Indirect Addressing

Under this addressing scheme the effective address EA and the contents of the operand field are related by

$$\begin{aligned} EA &= (A) \text{ and} \\ D &= (EA) \end{aligned}$$

The disadvantage of this addressing scheme is that it requires two memory references to fetch the operand. The first memory reference is used to fetch the effective address from the memory and the second is for fetching the operand using EA. In this scheme the addressed space is determined by word length. In many machines multiple levels of indirection may be used where  $EA = (\dots (A)\dots)$  gives the relationship between A and EA. In such a case a bit (generally the most significant bit) in the word address specifies the indirection. If this bit is '1', then the contents of the word represent the address of the address of the operand and if it is '0' then the contents of the field represent the address of the operand. If the size of the address field is n then the

address space available under multiple indirections addressing scheme is  $2^{n-1}$ .

### 3.2.4 Register Addressing

In this addressing scheme, the instruction specifies the address of the register containing the operand:

$$\begin{aligned} \text{EA} &= \text{R} \\ \text{D} &= (\text{EA}) \end{aligned}$$

Please note that EA here is a register address and not a memory address. The advantage here is that only a few bits are needed to address the operand. For example, for a machine having 16 general purpose registers only 4 bits are needed to address a register.

In some cases the address of the register containing the operand may not be explicitly specified but is understood implicitly. This is generally the case where one of the operands is in a special register called the accumulator.

Register access is faster than memory access. So register addressing provides faster instruction execution. However, this statement is valid only if the registers are employed efficiently. For example, if an operand is moved into a register and processed only once and then returned to the memory, then no saving occurs, however if an operand is used repeatedly after bringing it into registers then we have saved few memory references. Thus the task of using registers efficiently deals with the task of finding what operand values should be kept in registers such that memory references are minimised. Normally, this task is done by a compiler of a high level language while translating the program to machine language.

### 3.2.5 Register Indirect Addressing

Under this addressing scheme the operand field specifies the registers which contain the address of the operand.

$$\begin{aligned} \text{EA} &= (\text{R}) \text{ and} \\ \text{D} &= (\text{EA}) \end{aligned}$$

The address capability of register indirect addressing scheme is determined by the size of the register.

### 3.2.6 Displacement Addressing

This is a very powerful scheme. It contains both the direct addressing as well as the register indirect addressing schemes. Here the content A of the operand field is related to EA by

$$EA = A+(R)$$

The register address R may be specified explicitly or implicitly in the instruction. Depending on the use and the implementation this address scheme may be known as:

**Indexed Addressing Scheme:** This addressing scheme is generally used to address the consecutive locations of memory (which may store the elements of an array). The interpretation of the expression  $EA = A+(R)$  is as follows:

The contents of the operand field A is taken to be the address of the initial or the reference location (or the first element of array). The contents of register R gives the displacement with respect to the reference location. For example, to address an element  $B_i$  of an array  $B_1, B_2, \dots, B_n$ , with each element of the array stored in two consecutive locations and the starting address of the array being 101, the operand field A shall contain the number 101 and the register R will contain the value of the expression  $(i-1) \times 2$ . Thus, for the first element of the array the register will contain 0. For addressing the 5<sup>th</sup> element of the array, the  $A = 101$  whereas register will contain  $(5-1) \times 2 = 8$ . Therefore, the address of the 5<sup>th</sup> element of array  $B_5 = 101 + 8 = 109$ . The  $B_5$  however, is stored in location 109 and 110. To address any other element of the array, changing the content of the register (let us call it index register) will suffice. As the index registers are used for iterative applications, therefore, an index register is incremented or decremented after each reference to it. In several systems this operation is performed automatically during the course of an instruction cycle. This feature is known as autoindexing. Autoindexing can be autoincrementing or autodecrementing. The choice of the register to be used as an index register differs from machine to machine. Some machines employ general purpose registers for this purpose while other machines may specify special purpose registers referred to as index registers.

Another related addressing scheme which couples the indirect addressing with indexing is also utilised by several systems. Here, there are two possibilities:

Indexing is performed after indirection (post-indexing):

In this scheme the memory address specified by opcode addresses the location that contains a direct address which is to be indexed. That is:

$$\begin{aligned} \text{DA} &= (A) \\ \text{EA} &= \text{DA} + (R) \\ \text{D} &= (\text{EA}) \\ &\quad (\text{DA is direct address}) \end{aligned}$$

Indexing performed before indirection (pre-indexing):

This means that the address generated after indexing is the address of the location of operand. That is

$$\begin{aligned} \text{IA} &= A + (R) \\ \text{EA} &= (\text{IA}) \\ \text{D} &= (\text{EA}) \\ &\quad (\text{IA is indexed address}) \end{aligned}$$

In normal circumstances both pre-indexing and post-indexing are not used in an instruction set simultaneously.

**Base Addressing Scheme:** This addressing scheme is generally employed to relocate the program in the memory especially in a multiprogramming environment. Here the register R, referred to as base register contains the initial address in the memory (referred to as the base address) of the program segment being relocated. The operand field A contains the displacement of an instruction or data with respect to the base address. In this case:

$$\begin{aligned} \text{EA} &= A + (B); \quad \text{D} = (\text{EA}) \\ &\quad (B) \text{ refers to the contents of a base register B.} \end{aligned}$$

The contents of the base register may be changed in the privileged mode only, i.e. in the user mode the contents of the base register cannot be changed.

The base addressing scheme while on one hand provides the enhanced addressable space on the other hand it provides protection of users from one another.

In a base addressing scheme the address of an index addressed element is given by:

$\text{EA} = A + (B) + (I)$ , where B and I are base register and index register respectively.



Like the index register a base register may be a general purpose register or a special register reserved for base addressing.

**Relative Addressing Scheme:** In this addressing scheme, the register R is the program counter (PC) containing the address of the current instruction being executed. The operand field A contains the displacement (positive or negative) of an instruction or data with respect to the current instruction. These addressing schemes have advantages if the memory references are nearer to the current instruction being executed.

### 3.2.7 Stack Addressing Scheme

This is not a very common addressing scheme. In this addressing scheme, the address of an operand is not specified explicitly. It is implied. The operand is found on the top of a stack. In some machines the top two elements of stack and top of stack pointer are kept in the CPU registers, while the rest of the elements may reside in the memory. Figure 74 sums up the operating principles of all these addressing schemes.

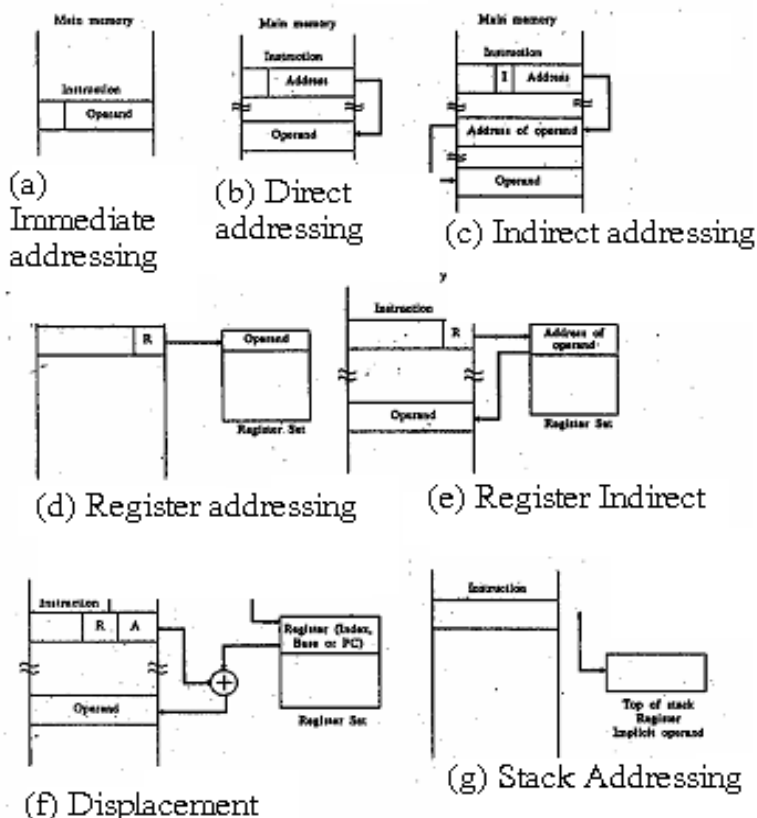


Figure 74: Basic addressing modes

In general not all of the above modes are used for all applications. However, some of the common areas where compilers of high level languages use them are:

Autoindex mode	→	For pushing or popping the parameters of a procedure
Direct mode	→	Normally used for global variables (used a little bit less than local variables)
Register	→	For holding local variables of procedures (frequently used)
Index	→	Accessing iterative local variables such as arrays
Immediate	→	For moving constants, initialisation of variables.
Register indirect	→	For holding pointers to structure in programming languages such as records in Pascal.

### 3.3 Instruction Format Design

As discussed earlier an instruction consists of an opcode and one or more operands which are addressed implicitly or explicitly. An instruction format is used to define the layout of the bits allocated to these elements of instructions. In addition, the instruction format explicitly or implicitly indicates the addressing modes used for each operand in that instruction.

As far as the designing of instruction format goes, it is a complex art. The computers have a variety of instruction designed for them in the last so many years. We will discuss in this section, the design issues for instruction sets of the machines. We will discuss only pointwise details of these issues.

#### 3.3.1 Instruction Length

**Significance:** It is the most basic issue of the format design. It determines the richness and flexibility of a machine.

**Basic Tradeoff:** Smaller instruction (less space) vs. desire for more powerful instruction repertoire.

Normally programmers desire:

- More opcode and operands: as they result in smaller programs.
- More Addressing modes: for greater flexibility in implementing functions like table manipulations, multiple branching.

However, a 32 bit instruction although will occupy double the space and can be fetched at double the rate of a 16 bit instruction, but cannot be doubly useful.

### Factors which must be considered for deciding instruction length

**Memory size** : If larger memory range is to be addressed, then more bits may be required in the address field.

**Memory organisation** : If the addressed memory is virtual memory then memory range which is to be addressed by the instruction is larger than physical memory size.

**Memory transfer length (in bus system is equal to the data bus length)** : Instruction length should normally be equal to data bus length or a multiple of it.

**Memory transfer** : The data transfer rate from the memory ideally should be equivalent to the processor speed. It can become a bottleneck if the processor executes instructions faster than the rate of fetching the instructions. One solution for such a problem is to use cache memory or another solution can be to keep the instruction short.

Normally an instruction length is kept as a multiple of the length of a character (that is 8 bits), and equal to the length of a fixed point number. The term word is often used in this context. Usually the word size is equal to the length of a fixed point number or equal to memory-transfer size. In addition, a word should store an integral number of characters. Thus, word sizes of 16 bits, 64 bit are becoming very common and hence the similar lengths of instructions are normally being used.

### 3.3.2 Allocation of Bits

The tradeoff here is between the numbers of opcode versus the addressing capabilities. An interesting development in this regard is the

development of variable length opcode. An opcode has a minimum length and then depending on the instructions which require fewer operands or less powerful addressing mode, the opcode can be changed.

Let us discuss the factors which are considered for selection of addressing bits:

- **Number of Addressing Modes:** The more the explicit addressing modes used, the more bits are needed for mode selection. However, some machines have implicit modes of addressing.
- **Number of Operands:** Fewer numbers of operand references in an instruction although require less bits yet result in longer programs. Nowadays many of the machines are having two operand references in an instruction. Each of these operands may need a mode indicator field of its own or both the operands may have the same addressing mode indicator field that is the same addressing mode.
- **Register Addressing versus Memory Addresses:** If more and more registers can be used for operand references then instructions are bound to be smaller as the number of registers is far less than the memory size, therefore, they require only a few bits in comparison to the bits needed for the memory addresses. In general, the number of user visible registers provided is 8 to 16. This number is found to be an optimum number of registers by several studies.
- **Register Sets:** Even in the case of registers addresses the trend is moving from a large number of general purpose registers to two or more sets of registers, for specialised data storage. For example, one special register set can store only the data for calculation, while the other can store only the addresses. These results in further decrease in the size of instruction bits for register addressing. For example, if a machine has 16 general purpose registers then a register address of it requires at least 4 bits; however, if these 16 registers are used as two specialised sets of registers then one of the 8 registers need to be addressed at a time, thus requiring only 3 bits for the register addressing mode.
- **Range of Addresses:** The range of main memory addresses which need to be addressed directly or indirectly are involved for having a specific number of bits in instructions. For example, if direct addressing is used then the addressed memory determines directly the number of instruction bits required. However, in displacement index addressing schemes it is the offset which can control the number of bits desired in the instruction.

- **Granularity of Address:** As far as memory references are concerned, granularity implies whether an address is referencing a byte or a word at a time. This is more relevant for machines which have 16 bit, 32 bit and higher bit words. Byte addressing although may be better for character manipulation, however, requires more bits in an address. For example, if a memory of 4K words (1 word = 16 bit) is to be addressed directly then it requires:

WORD Addressing = 4 K words  
 =  $2^{12}$  Words  
 ⇒ 12 bits are required for word addressing.

Byte Addressing =  $2^{12}$  Words  
 =  $2^{13}$  Bytes  
 ⇒ 13 bits are required for byte addressing.

### 3.3.3 Variable-length of Instructions

With the better understanding of computer instruction sets, the designers came up with the idea of having a variety of instruction formats of different lengths. What could be the advantages of such a scheme? The advantages of such a scheme are:

- Large numbers of operations can be provided which have different lengths of instructions.
- Flexibility in addressing schemes can be provided efficiently and compactly.

The basic disadvantage of such a scheme is to have a complex CPU. However, the advances in technology and increase in basic understanding about CPU designing may reduce the overheads required for the added complexity of such a scheme. However, one condition which still holds for the length of instructions is that all the possible instruction lengths should be multiple of word size.

An important aspect of these variable length instructions is: “The CPU is not aware of the length of the next instruction which is to be fetched”. This problem can be handled if each instruction fetch is made equal to the size of the longest instruction. Thus, sometimes in a single fetch multiple instructions can be fetched.

After discussing various concepts, let us look at the instruction set of IBM S/370 machine in the next section.

## 3.4 Examples of Instruction Sets

In this section we will look at the instruction set of a very important vendor- the IBM. We have not included instruction set details of any of the INTEL microprocessors in this unit. The other instruction sets can be studied by you from the further readings.

**IBM 370 Architecture:** The IBM S/370 is a popular architecture. Let us discuss the features provided in this architecture.

**Data Types:** It provides the following data types:

<b>Data Type</b>	<b>Allows Length</b>	<b>Characteristics</b>
<b>Binary Integers</b>	16 and 32 bits	Unsigned i.e. non-negative numbers or signed binary integer in signed 2's complement notation.
<b>Floating Point</b>	32, 64 and 128 bits	A 7-bit exponent with all the formats.
<b>Decimal Numbers</b>	1 to 16 bytes	Rightmost 4 bits are used for sign. Thus, 1 to 31 digit decimal numbers can be represented. Arithmetic on these packed decimal integers is provided.
<b>Binary logical</b>	8, 32 and 64 bits and variable length logic data till 256 byte	Logic operations are defined for the data units of the given length.
<b>Character</b>	8 bits	EBCDIC is used.

**Operation Types:** Although the IBM/370 principal operation manual classifies the machine instructions in six broad categories, yet for the sake of simplicity they have been separated according to function by Stallings. This classification categorises the instruction type in 12 categories.

<b>Instruction Type</b>	<b>Description</b>
<b>Fixed Point Arithmetic</b>	One register is used for storing one of the operands and the result, while the second

<b>Instructions</b>	operand can be in another register or in the memory. It provides operations in either unsigned 16 or 32 bit integer operands or 16 or 32 bit 2's complemented integer numbers
<b>Logical instruction</b>	Provides bitwise AND, OR and XOR operations.
<b>General Register Shifting Instructions</b>	Provides 8 shift instructions pairing the following options: <ol style="list-style-type: none"> <li>(1) Left or right shift</li> <li>(2) Use of a single or double register</li> <li>(3) Signed or logical shift.</li> </ol> For example, two of the eight shift instructions will be a left, single register, logical shift instruction and a left, double register, signed shift instruction. A second operand is used in the instruction to specify the amount of shift. The shift instruction affects either a single register or this can affect an even-odd register pair. The arithmetic shift instruction leaves the sign bit intact.
<b>General Register – Load and Store Instructions</b>	Provided transfer operations are: <ul style="list-style-type: none"> <li>- Register to register transfer</li> <li>- Register to memory transfer</li> <li>- Memory to register transfer</li> </ul>
<b>Compare Instructions</b>	It performs the comparing and testing functions. A condition code of 2-bit is set by the instructions.
<b>Branching Instructions</b>	The value of conditional code determines the branch. A special instruction called Branch and link is used for subroutine call and return. The user needs to specify the register to be used for storing return address.
<b>Conversion Instructions</b>	Converts various forms of data for example, from decimal to binary
<b>Decimal Instructions</b>	Provided for providing operations on decimal data. These instructions include arithmetic operations, shifting operations and unpacking operations.
<b>Floating Point Instructions</b>	Add, subtract, multiply and divide instructions are provided on 4 byte, 8 byte and 16 byte floating point numbers.
<b>Special Purpose Control Instructions</b>	These instructions cause a control passing. A example of one such instruction is the supervisory call which causes an interrupt those results in passing the control to the operating system.

<b>Privileged I/O Instructions</b>	These instructions are used for starting input/output using an I/O channels. One such command is start I/O.
<b>Privileged System Control Instructions</b>	These instructions are issued only by the operating system for controlling registers and data structures which are needed by the operating system.

### IBM 370 Addressing

In IBM 370 the memory is addressable at byte level. However, all the addresses are virtual addresses. In the system S/370 the length of virtual address is 24 bits whereas, the length in the case of IBM S/370 XA is 31 bits. In this system only three basic addressing modes are supported. These are:

- **Immediate Addressing:** It is a 1 byte operand, provided within the instruction.
- **Register Addressing:** In this mode a register operand is addressed. This register operand can be in one of the 16 32-bit general purpose register or 4 64-bit floating point registers. A register address consists of 4 bits as we are referencing a floating point register which can be referenced by 2 bits (4 floating point registers), 4 bits are used
- **Displacement Addressing:** In this mode the operand is stored in the virtual memory. Here two types of formats are used:

**The base register format:** The operand reference in such a scheme consists of two components:

#### *The displacement provided in the instruction:*

The 4-bit register address is one of the general purpose registers and will be used as a base register. However, only the right most 24 bits of a 32 bit register are used for computing addresses in IBM/360 whereas only rightmost 31 bits are used in IBM 370 XA model. A register address 0 specifies that no register has been used and the displacement becomes the direct address of memory location in that case.

#### *The use of the index registers along with the base register:*

In this scheme the instruction reference consists of the displacement, a four-bit reference to a general purpose register to be used as base



register and a general purpose register to be used as index register. The feature of autoindexing is also provided.

Let us summarise the addressing modes used in IBM/370

<i>Addressing Mode</i>	<i>Calculation of effective address</i>
Immediate	Operand = A
Register	EA = R
Displacement:	
Base Register	EA = A + (B)
Index on base register	EA = A + (B) + (I)

I → Indicates address of index register. It will be 4 bit long.

B → Indicated address of the base register. It will be 4 bit long

**Figure 75: A summary of IBM S/370 addressing modes**

The addressing schemes provided by this system although are simple and easy, yet a programmer must understand how to use the segments in IBM system/370. This discussion is beyond the scope of this unit. You can refer to further readings for more details.

### **Instruction Format**

We will present a simplified tabular representation for the instruction format of IBM system/370.

#### **Features:**

- Variable length instructions : bytes, 4 bytes, 6 bytes instructions
- Variable op-code length : 1 byte and 2 byte
- Eight different instruction formats are used.
- Mostly two operand instructions, however, one and three operands are also used in some instructions.
- First two bits of instructions specifies:  
length of the instruction and

## format of the instruction

First two bits of opcode    Instructions length (bytes)    Instruction opcode format

00	2	RR
01	4	RX
10	4	RRE/RS/S/SI
11	6	S S/S S E

The instruction formats are:

Register-Register (R R)	: A compact representation Instruction length Opcode Register addresses	: 2 bytes : 1 byte : 2 nos of 4 bit each.
Register Register Extended (R R E)	: Used for privileged instructions Used by the operating system. Instruction length Opcode Register addresses	: 4 bytes : 2 bytes : 2 of 4 bit each
Register_Indexed (R X)	: Instruction Length Opcode First operand Register operand Second operand Virtual memory operand	: 4 bytes : 1 byte : One of 4 bits : One 4 bit index register 4 bit base register 12 bit displacement
Register_Storage (R S)	: Instruction length Number of operands Opcode First operand and third operand Second operand	: 4 byte : 3 (the only three operand instruction format) : 1 byte : Register operands require 4 bit each for register reference. : Virtual memory operand using a base register (4 bit) and displacement (12 bits)

Storage Immediate (S I) : Instruction length : 4 bytes  
 : Opcode : 1 byte  
 : First operand : Virtual memory operand using a base register (4 bit) and displacement (12 bits)  
 : Second operand : Immediate operand of one byte

Storage (S) : Used for representing privileged instructions for I/O or system control.  
 : Instruction length : 4 bytes  
 : Opcode : 2 bytes  
 : Number of operands : One

Operand address is virtual address using a base register (4 bits) and displacement (12 bits)

Storage.Storage (S S) : Instruction length : 6 bytes  
 : Opcode : 1 byte  
 : Number of operands = 2  
 Both operands are virtual memory operands specified by base register (4 bits) and displacement (12 bits)  
 Use of remaining 1 byte :  
 One length format: The remaining byte specifies the number of bytes to be operated upon. Used for moving a block of characters from one location to another.  
 Two length format: two length fields of 4 bits each specifying the size of each of the two operands  
 Used for operations on BCD.  
 Register specifications: The byte designates two general purpose registers which contain the control information or length specifications. Used for privileged instructions.

Storage.Storage Extended (S S E) : Used for privileged instructions : 6 bytes  
 : Instruction length : 2 bytes  
 : Opcode  
 Two operands both in the form of base register

Thus, IBM System/370 formats tries to make efficient use of instruction lengths. Figure 76 gives the summary of instruction formats of this system.

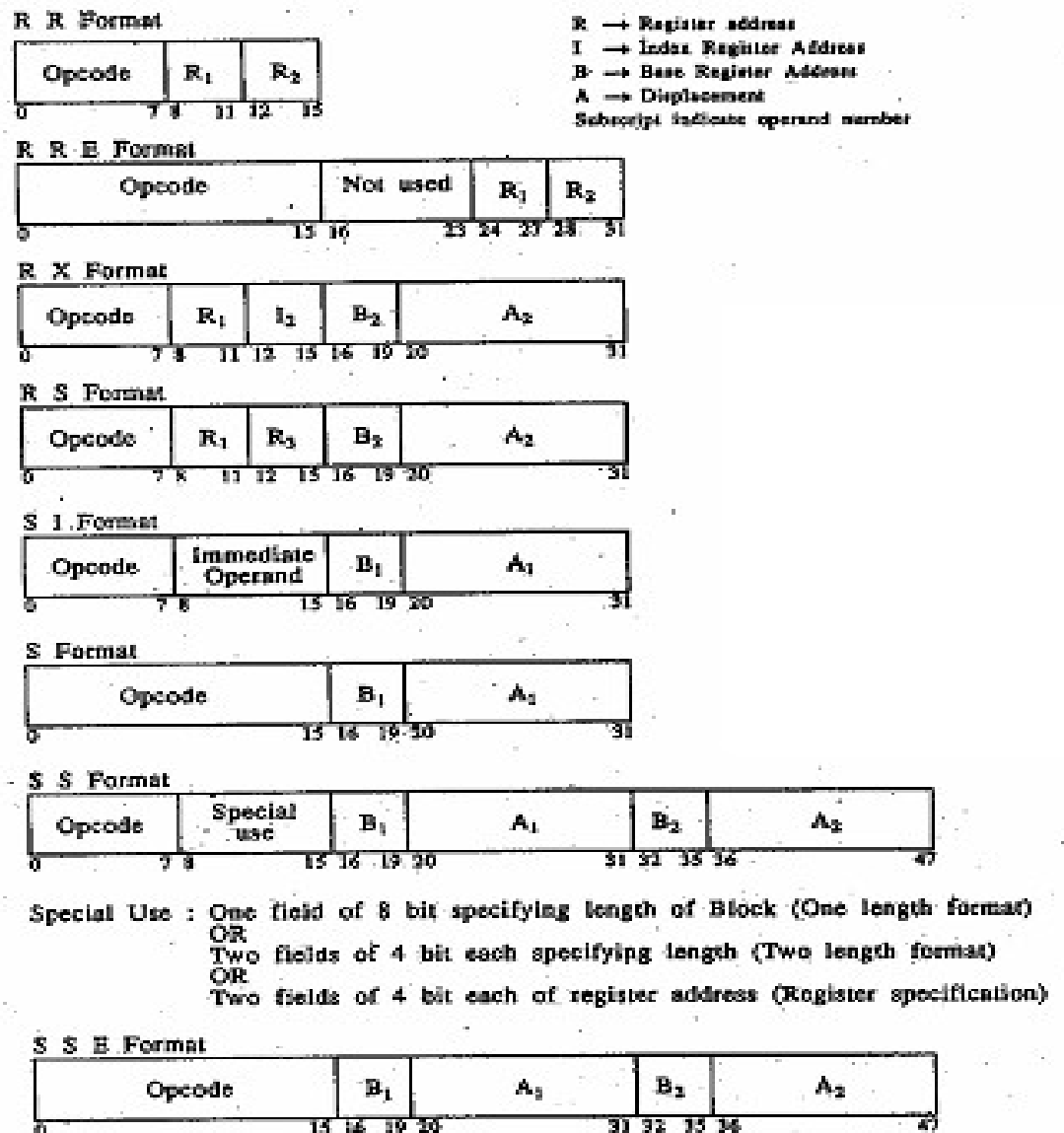


Figure 76: A summary of IBM system/370 instruction formats

## 4.0 CONCLUSION

In discussing the structure of the computer and data representation in the previous module, we mentioned the term “instruction” which is a paramount term in computer operation.

This unit has therefore been dedicated to exposing you to what an instruction is and its components. To this end, you have been taken through the characteristics of instruction sets, types of instructions, addresses, types of operands, types of operations, addressing schemes, instruction format design, etc.

## 5.0 SUMMARY

In this unit, we have introduced you to various concepts relating to “instructions”. We have discussed the basic characteristics, the number of addresses, type of operands and operations in instructions and various addressing modes. We have also highlighted the basic issues while designing instruction formats and presented details on the instruction set of IBM S/370. Please note that we have not provided you with the detailed instructions which this machine provides, but only the conceptual model behind this instruction set. You can refer to further reading for more details on instruction set for various machines.

## 6.0 TUTOR-MARKED ASSIGNMENT

1. Find out the memory references required to get the data for the following addressing modes:
  - Direct addressing
  - Indirect addressing
  - Register indirect addressing
  - Immediate addressing
2. What are the differences between preindexing and postindexing?
3. What are the differences between base and relative addressing schemes?
4. State whether True or False
  - (a) Immediate addressing is best suited for storing floating point numbers True  False
  - (b) Indirect addressing requires fewer memory accesses than that of index addressing True  False
  - (c) Index addressing is used for addressing arrays. True  False
5. A long instruction can be executed faster than short instructions. True  False
6. Virtual addresses require more bits for the address part of instructions in comparison to non-virtual addresses. True  False
7. The speed gap between processor and memory suggests that instruction size should be as big as possible. True  False

8. In general, the instruction length is kept equal to the size of floating point storage. True  False
9. Large numbers of operand addresses in instruction leads to smaller programs. True  False
10. Register addresses for specialised register set machines are smaller than that of the machines which have general purpose register sets. True  False
11. A machine using direct addressing mode having a memory addressing capability of 8 K bytes, requires 13 bits for byte addressing. True  False
12. In a variable length instruction format, an instruction fetch must fetch the words equal to the size of the smallest instruction. True  False

## 7.0 REFERENCES/FURTHER READINGS

- Mano, M. Morris (1993). *Computer System Architecture* (3<sup>rd</sup> ed). Prentice Hall of India.
- Hayes, John P. (1988) *Computer Architecture and Organisation* (2<sup>nd</sup> ed). McGraw-Hill International editions.
- Stallings, William. *Computer Organisation and Architecture* (3<sup>rd</sup> ed) Maxwell Macmillan International Editions.
- Baron, Robert J. and Higbie Lee. *Computer Architecture*. Addison-Wesley Publishing Company.
- Tanenbaum, Andrew S.( 1993). *Structural Computer Organisation* (3<sup>rd</sup> ed) Prentice Hall of India.

## **UNIT 2 REGISTER ORGANISATION AND MICRO-OPERATIONS**

### **CONTENTS**

- 1.0 Introduction
- 2.0 Objectives
- 3.0 Main Content
  - 3.1 Basic Structure of the CPU
  - 3.2 An Advanced Structure
  - 3.3 Register Organisation
    - 3.3.1 Program Visible Registers
    - 3.3.2 Status and Control Registers
  - 3.4 Micro-Operations
    - 3.4.1 Register Transfer Micro-Operations
    - 3.4.2 Arithmetic Micro-Operations
    - 3.4.3 Logic Micro-Operations
    - 3.4.4 Shift Micro-Operations
    - 3.4.5 Implementation of a Simple Arithmetic, Logic and Shift Unit
  - 3.5 Instruction Execution and Micro-operations
- 4.0 Conclusion
- 5.0 Summary
- 6.0 Tutor-Marked Assignment
- 7.0 References/Further Readings

### **1.0 INTRODUCTION**

As we discussed earlier, the main task performed by the CPU is the execution of instructions. In the previous unit, we discussed the instruction set of the computer system. But one thing which remained unanswered is: how will these instructions be executed by the CPU?

The above question can be broken down into two slightly simpler questions. These are:

- What are the steps required for the execution of an instruction?
- How are these steps performed by the CPU?

The answer to the first question lies in the fact that each instruction execution consists of several steps. Together they constitute an instruction cycle. We have already given a state diagram about instruction cycle in Unit 1, Module 1 of this course. In this unit we will present a more structured view of the instruction cycle. We will also

discuss the micro-operations, the smallest operations performed by the CPU.

In order to answer the second question, we must have an understanding of the basic structure of a computer. As discussed earlier, the CPU consists of an arithmetic logic unit, the control unit and operational registers. We will examine the register organisation in this unit, but we will discuss the arithmetic-logic unit and control unit organisation in the subsequent unit.

However, we have sequenced this unit by starting from the very basic structure about the CPU and following it with the discussion of the register organisation in general. This is followed by the discussion on micro-operations and their implementation. The discussion on micro-operations will gradually lead us towards the discussion of a very simple ALU structure. We will finally wind up the unit with the discussion on the instruction cycle.

## **2.0 OBJECTIVES**

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

- describe the register organisation of the CPU;
- differentiate among various structures of the CPU;
- define a micro-operation;
- differentiate among various micro-operations;
- discuss how the micro-operations can be implemented; and
- discuss an instruction cycle.

## **3.0 MAIN CONTENT**

### **3.1 Basic Structure of the CPU**

As discussed earlier, the CPU basically consists of two main components.

- An arithmetic and logic unit to perform the arithmetic or logic operation on data
- A control unit which plays an important role for the functioning of the CPU itself and in the transfer of data/information from/to another device to/from CPU.

In addition the CPU contains several operational registers. The basic task performed by the CPU is the instruction execution. An instruction is executed using several small operations called micro-operations.



The basic issues relating to the CPU can be:

- It should be as fast as possible; only the available technology should limit the CPU's speed but the number of components should be small in a good CPU.
- The capacity of the main memory needed by the CPU should be quite large. Since it is large, therefore, it should be constructed using a slower technology than that of a CPU.

Let us define the cycle time of the CPU ( $t_{cpu}$ ) as the time taken by the CPU to execute a well-defined shortest micro-operation, and memory cycle time as the speed at which the memory can be accessed by the CPU. It has been found that the memory cycle time ( $t_{mem}$ ) is approximately 1 – 10 times higher than that of the CPU cycle time. That is  $t_{mem}/t_{cpu} = 1$  to 10. Therefore, within the CPU temporary storage is provided in the form of CPU registers. The CPU registers are used to store instructions and operands within the CPU. The CPU registers can be accessed almost instantaneously. In other words, the ratio of the time to access a register by the CPU in comparison to the time to access memory by the CPU is approximately equal to  $t_{mem}/t_{cpu}$ . Thus, the instructions whose operands are stored in the fast CPU registers can be executed rapidly in comparison to the instructions whose operands are in the main memory of a computer. Thus, the instruction execution is implemented as:

- Bring in the operands required by the instruction from the main memory to the CPU registers.
- Perform the operation desired by the instruction on the operands stored in the registers.
- Finally, the results are transferred back to the memory if needed.

The input/output from the devices can also be carried out in the same way using I/O controllers.

The design of the CPU in modern form was first proposed by John von Neumann and his colleagues for the IAS computer. The IAS computer had a minimal number of registers along with the essential circuits. This computer had a small set of instructions and an instruction was allowed to contain only one operand address. A register called “accumulator” was used as a key register for the execution of most of the instructions as it was used for storing the input operand for the arithmetic logic unit.

Figure 77 gives the structure of the IAS computer.

This type of architecture is used in some modern mini and micro-computers. The structure shown in Figure 77 consists of the following registers.

**Accumulator (AC):** It interacts with ALU and stores the input or output operand.

**Data Register (DR):** It acts as a buffer storage between the main memory and the CPU. It also stores the operand for the instructions such as  $ADD\ DR\ or\ AC \rightarrow AC + DR$ , that is, the content of AC and the data register are added by ALU and the results are stored in the accumulator. Thus, data register can also store one of the input operands.

**Program Counter (PC):** It contains the address of the next instruction word to be executed.

**Instruction Register (IR):** It holds the current instruction.

**Memory Address Register (MAR):** It is used to provide address of memory location from where data is to be retrieved or to which data is to be stored.

The contents of the PC are modified either after fetching an instruction or by a branch or skip instructions. MAR and DR play important roles in transfer of data between CPU and the memory. In the computer systems which use system bus, MAR is directly connected to address bus, while DR is directly connected to data bus. DR is also used to interchange data among several other registers.

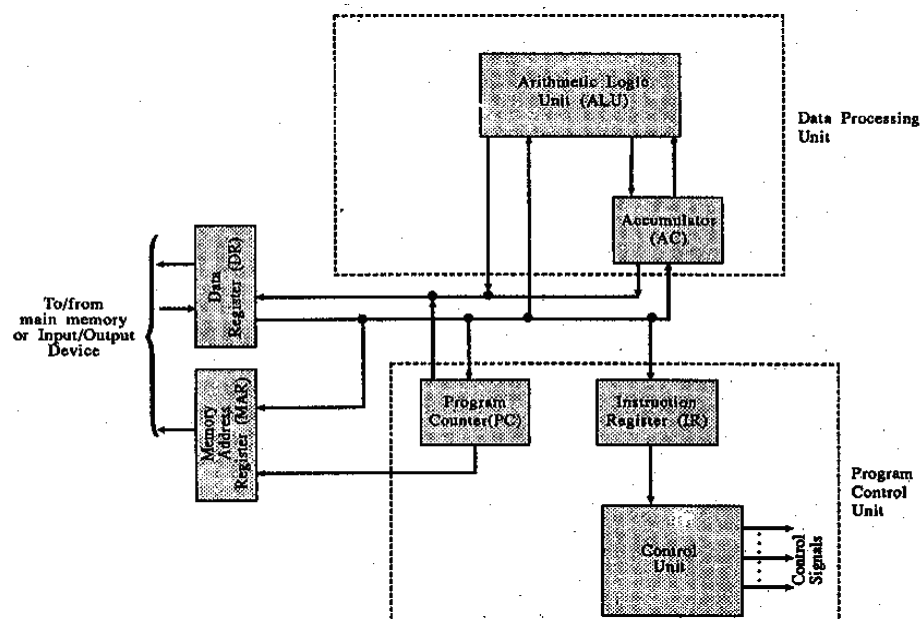


Figure 77: Basic Structure of the CPU

So, we have a simple basic CPU which can perform various computational tasks easily. How to make this simple structure more powerful in terms of processing efficiency and instructional style. In the next section we will discuss how the simple structure can be made more powerful.

### 3.2 An Advanced Structure

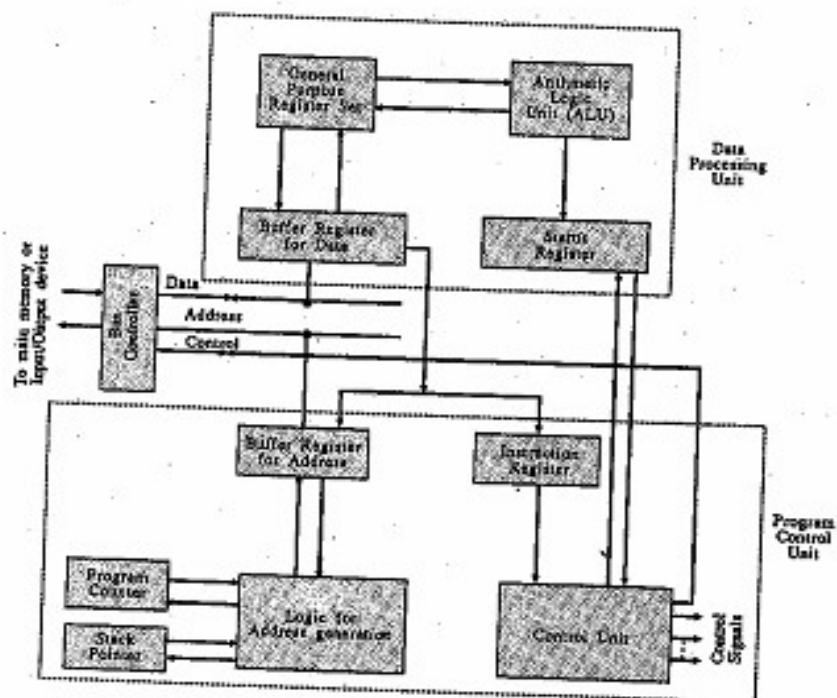
Let us have a few additional features in the simple structure of the CPU such that it can be made more powerful. Some of these aspects are:

- Provide additional registers for storing operands and addresses, thus, replacing a single accumulator by a set of registers. These registers can be used as general purpose registers which are multi-purpose in nature (e.g. can store either operands or addresses). A set of these general registers is called a register file. One of the key functions of these registers is to store the operands which are needed for calculation of memory addresses. The IBM S/360 computers had this general register organisation. In some microprocessor, some special address registers namely index register, base register, etc. are used. An example of such is Motorola 68020 microprocessor which has a set of 8 data registers along with a matching set of 8 address registers.
- Increase the capabilities of ALU circuits. For example, most microprocessors have capabilities for performing addition and subtraction on fixed point numbers. This capability with only little extra circuitry can be used for multiplication and division on fixed point numbers also. However for implementing arithmetic on floating point numbers a substantial increase of circuitry is needed.
- Include special register to facilitate conditional jumps within a program. A status register which gives information about various conditions such as the sign of the result, whether the results is zero, arithmetic overflow, etc. in the proceeding instruction execution, can be used. This status register can be checked for typical condition for execution of a branch instruction.
- Include special registers for transfer of control between different subroutines or subprograms or interrupts. One such register used in IBM 360/370 series is called PSW (Program Status Word) which stores program counters and various condition flags. Thus, PSW can record the execution status of the program. On occurrence of interrupt or a call to subroutine this PSW is stored in a specific area in the memory and is restored when the execution of this program is to be resumed. Many computers use a stack in the main memory for

controlling transfer of control. This is a very flexible approach. It uses a special area in the main memory and a register called stack pointer (SP) points to the top of the stack. The stack is a LIFO (Last In First Out) data structure. It has been found to be very useful for implementing program transfer of control, very efficiently.

- Provisions for execution of more than one instruction simultaneously. These provisions are not discussed at present but will be taken up in a later course.

Considering the requirements, let us have a more enhanced view of the CPU. Figure 78 gives the detailed view of the CPU.



**Figure 78:** The CPU with a general register organisation

The general purpose registers can be in general 8-16. ALU performs operations on the data stored in these general purpose registers and also stores results in these registers only. There are few special-purpose address registers. Two of these are the program counter (PC) and the stack pointer (SP). The status register stores the key characteristics or conditions of the result of the last ALU operation. A special simple arithmetic logic unit can be attached as an address generation logic performing the simple fixed point computations. The inputs to the control unit are from the instruction register which contains the operation code of the instruction to be executed and from the status register which helps in generating proper control signals on branch operations. The system bus plays the role of the communication media. Please note the direction of arrows on the address bus. There are several

intermediate buffer registers also which help in intermediate storage of information. In this organisation of CPU parallelism can be implemented within the ALU operation or through the overlapped operations of data processing and program control unit.

We will now discuss the components of the CPU. In the subsequent sections we will discuss about the Register organization in details followed by the discussions on microoperations. This will lay a foundation for discussions on arithmetic logic unit and control unit organisation which are the topics in Units 3 and 4 of this Module.

### SELF-ASSESSMENT EXERCISE

1. The program counter (PC) register is used for storing an instruction  
True  False
2. Program Status Word (PSW) is used for storing the instruction which is to be executed next.  
True  False
3. Accumulator machines may require more memory references than that of a general purpose machine for executing an optimised program.  
True  False
4. Stack pointer points to the top of register file.  
True  False

### 3.3 Register Organisation

In the previous subsection, we have given some hints to the types of registers a modern day CPU must have. Let us take a more general view on register organisation in this section. The internal processor memory of a CPU is served by its registers. One the key difference among various computers is the difference in their register sets. Some computers have very large sets while some have smaller sets. But on the whole, from a user's point of view the register set can be classified under two basic categories.

- **Programmer Visible Registers:** These registers can be used by machine or assembly language programmers to minimise the references to main memory.

- **Status Control and Registers:** These registers cannot be used by the programmers but are used to control the CPU or the execution of a program.

Different vendors have used some of these registers interchangeably, therefore you should not stick to these definitions rigidly. Yet this categorisation will help in a better understanding of register sets of machine. We will discuss these categories.

### 3.3.1 Program Visible Registers

These registers can be accessed using machine language. In general we encounter four types of program visible registers.

- General Purpose Registers
- Data Registers
- Address Registers
- Condition Code Registers

The general purpose registers are used for various functions desired by the processor. A true general purpose register can contain operands or can be used for calculation of the addresses of operands for any operation code of an instruction. But trends in today's machines show a drift towards dedicated registers. For example, some registers may be dedicated to floating point operations. In some machines there is a distinct separation between data and address registers.

The data registers are used only for storing intermediate results or data. These data registers are not used for the calculation of the address of the operand.

An address register may be a general purpose register, but some dedicated address registers are also used in several machines. Examples of dedicated address registers can be:

Segment Pointer : Used to point out a segment of memory

Index Register : These are used for index addressing schemes.

Stack Pointer (when program visible stack addressing is used)

There are several issues related to program visible registers. Some of the issues are:

Do we use general purpose registers or dedicated register in a machine?

Does the number of registers affect the design of an instruction of a computer?

Well, in the case of a specialised register the number of bits needed for register specific details are reduced as here we need to specify only a few registers out of a set of registers. However, this specialisation does not allow much flexibility to the program. Although there is no best solution to this problem, yet the trends are in favour of use of specialised registers.

Another issue related to the register set design is the number of general purpose registers or data and address registers to be provided in a micro-processor. The number of registers also affects the instruction design as the number of registers determines the number of bits needed in an instruction to specify a register reference. In general, it has been found that optimum number of registers in a CPU is in the range 8 to 32. In case registers fall below the range then more memory references per instruction on an average will be needed, as some of the intermediate results then have to be stored in the memory. On the other hand, if the numbers of registers go above 32, then there is no appreciable reduction in memory references. However, in some computers, hundreds of registers are used. These systems have special characteristics. Reduced instruction set computers (RISC) exhibit this property. What is the importance of having less memory references? As the time required for memory reference is more than that of a register reference, therefore, the increased number of memory references results in slower execution of a program.

**Register Length:** Another important characteristic related to registers is the length of a register. Normally, the length of a register is dependent on its use. For example, a register which is used to calculate address must be long enough to hold the maximum possible address. Similarly, the length of data register should be long enough to hold the data type it is supposed to hold. In certain cases two consecutive registers may be used to hold data whose length is double of the register length.

Condition code registers may only be partially available to the programs. These register contains condition codes which are also known as flags. These flags are set by the CPU hardware while performing an operation. For example, an addition operation may set the overflow flag or on a division by 0 the overflow flag can be set etc. these codes may

be tested by a program for a typical conditional branch operation. The condition codes are collected in one or more registers. RISC machines have several set of conditional code bits. In these machines an instruction specifies the set of condition codes which is to be used. Independent sets of condition code enable the provisions of having parallelism within the instruction execution unit.

One of the key operations which are needed with the program usable registers happens when a subroutine call is issued. On a subroutine call all the registers are saved by the call statement itself and restored on encountering a return statement from the subroutine. This operation in most machines is automatic yet in certain machines this is done by the programs. Similarly while writing interrupts service routine you need to save some or call program unusable registers.

### 3.3.2 Status and Control Registers

For the control of various operations several registers are used. These registers cannot be used in data manipulations; however, the content of some of these registers can be used by the program. Some of the control registers for a von Neumann machine can be the Program Counter (PC), Memory Addressing Register (MAR) and Data Register (DR).

Almost all the CPUs, as discussed earlier, have status registers, a part of which may be program visible. A register which may be formed by condition codes is called condition code register. Some of the commonly used flags or condition codes in such a register may be:

<b>Sign Flag</b>	:	This indicates whether the sign of a previous arithmetic operation was positive (0) or negative (1)
<b>Zero Flag</b>	:	This flag bit will be set if the result of last arithmetic operation was zero.
<b>Carry Flag</b>	:	This flag is set, if a carry results from the addition of the highest order bits or a borrow is taken on subtraction of the highest order bit.
<b>Equal Flag</b>	:	This bit flag will be set if a logic comparison operation finds out that both of its operands are equal
<b>Overflow Flag</b>	:	This flag is used to indicate the condition of an arithmetic overflow.
<b>Interrupt Enable/disable Flag</b>	:	This flag is used for enabling or disabling interrupts.
<b>Supervisor Flag</b>	:	This flag is used in certain computers to determine whether the CPU is executing in



supervisor or user mode. In case the CPU is in the supervisor mode it will be allowed to execute certain privileged instructions.

In most CPUs, on encountering a subroutine call or interrupt handling routine, it is desired that the status information such as conditional codes and other register information be stored and restored on initiation and end of these routines respectively. The register often known as program status Word (PSW) contains condition codes plus other status information. There can be several other status and control registers such as interrupt vector register in the machine using vectored interrupt, stack pointer if a stack is used to implement subroutine calls, etc.

The status and control register design is also dependent on the operating system (OS) support. The functional understanding of OS helps in tailoring the register organisation. In fact, some control information is only of specific use to the operating system.

One major decision to be taken for designing status and control registers organisation is: how to allocate control information between registers and the memory. Generally first few hundreds or thousands of words of memory are allocated for storing control information. It is the responsibility of the designer to determine how much control information should be in registers and how much should be in the memory. This in fact is a tradeoff between the cost and the speed.

### **3.4 Micro-Operations**

After discussing structure and register organisation. Our next task is to examine the functionality of ALU and control unit. However as the main task of the computer is instruction execution, the details on how instructions can be executed will help in understanding the functionality of ALU and control unit. In this section, we will first define the various micro-operations and their hardware implementation from where we will move on to the design of a very simple circuit of an arithmetic logic unit as it will logically follow the discussion. This discussion will be followed by a new look at instruction execution in the next section.

A micro-operation, in general, is a primitive action performed by a machine on the data stored in the registers. In digital computers in general, there are four types of micro-operations:

- Register transfer micro-operations
- Arithmetic micro-operations

- Logic micro-operations
- Shift micro-operations.

### 3.4.1 Register Transfer Micro-Operations

These micro-operations as the name suggests transfer information from one register to another. The information does not change during this micro-operation. A register transfer micro-operation may be designed as:

$$R1 \leftarrow R2$$

Which implies that transfer the content of register R2 to register R1. R2 here is a source register while R1 is a destination register. We will use this notation through out this section. For a register transfer micro-operation there must be a path for data transfer from the output of the source register to the input of the destination register. In addition, the destination register should have a parallel load capability, as we expect the register transfer to occur in a pre-determined control condition. We will discuss the control in the later units of this module.

A common path for connecting various registers is through a common internal data bus of the processor. In general the size of this data bus should be equal to the number of bits in a general register.

Let us briefly find out how this internal bus can be constructed. We will give you a very simple way of constructing the bus for four bit registers using 4 x 1 multiplexers (please refer to Figure 79).

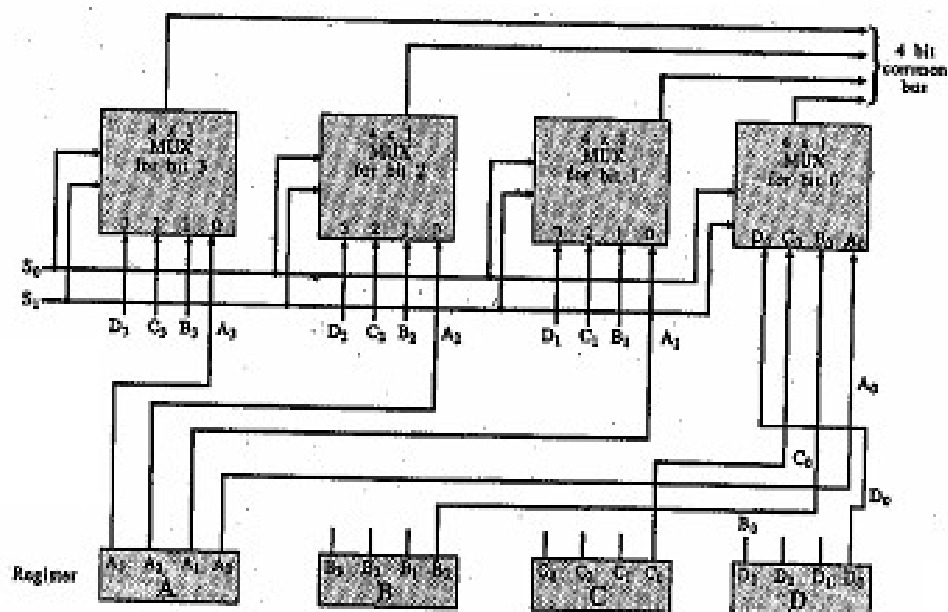


Figure 79: A four bit system for four bit four registers

Please note that the logic circuit of Figure 79 uses four multiplexers (MUX), two selection lines and four registers. Also note that the output of a MUX is one of the four inputs, that is, for  $i^{\text{th}}$  bit MUX the output can be either  $A_i$  or  $B_i$  or  $C_i$  or  $D_i$ , that is the  $i^{\text{th}}$  bit of any one of the four registers. The select line determines which of these register bits is to be selected. This is shown as follows:

$S_1$	$S_0$	Decimal equivalent of $S_1 S_0$ (Decimal Select (DS))	Output bit selected
0	0	0	$A_i$
0	1	1	$B_i$
1	0	2	$C_i$
1	1	3	$D_i$

Since, the same control bits  $S_1 S_0$  are passed to all the multiplexers, all the MUX will output bits of the same register. Thus, on having Decimal Select (DS) as 0 bus will output register A, on DS as 1 bus will output B and so on.

For receiving the data from this bus all the register input should be connected to this bus and all these registers should have a parallel load capability. Thus, by enabling load signal, a register can take data from the bus.

Now, let us focus on another important transfer which does not take place through the internal data bus, but through the system bus. These transfers are related to memory and input/output modules. The data transfer from input/output module to CPU or vice-versa is not very different from the transfer from memory to CPU or vice-versa. For a special case of memory mapped i/o both are handled in the same way. Also, the input/output operation is treated as a separate activity, where normally a program and therefore, instructions are executed. Let us focus our discussion on memory transfer which is the most important transfer for instruction execution as it has to take place at least once for every instruction.

**Memory Transfer:** Memory transfer is achieved via a system bus. Since the main memory is a random access memory, the address of the location which is to be used is to be supplied. This address is supplied by the CPU on the address bus. There are two memory transfer operations: Read and Write. Let us consider the CPU structure as shown in Figure 77, then the two memory operations will be performed as:

**Memory Read:**

- Put the memory address in the memory address register (MAR)
- Read the data of the location. This operation is achieved by putting the MAR data on the address bus along with a memory read control signal on the control bus. The resultant of memory read is put into the data bus which in turn stores the read data in the data register (DR). This whole operation can be shown as:

$$\text{DR} \leftarrow \text{M (MAR)}$$

**Memory Write:**

- Put the desired memory address in MAR and the data to be written in the DR.
- Write the data into the location: MAR puts the address on address bus and DR puts the data on data bus. A write control signal along with these two signals enables the data on data to be written into the memory location addressed by MAR.

$$\text{M(MAR)} \leftarrow \text{DR}$$

Normally, a memory read or write operation requires more cycles than a typical register transfer operation. The logic circuit of a memory is shown in Unit 3 of Module 1 of this course.

**3.4.2 Arithmetic Micro-Operations**

These micro-operations perform some basic arithmetical operations on the numeric data stored in the registers. These basic operations may be: addition, subtraction, incrementing a number, decrementing a number and arithmetical shift operation. An “add” micro-operation can be specified as:

$$\text{R3} \leftarrow \text{R1} + \text{R2}$$

It implies: add the contents of registers R1 and R2 and store them in register R3.

The add operation mentioned above requires three registers along with the addition circuit at the ALU. An alternate add micro-operation for the structure shown in Figure 77 will be:

$$\text{AC} \leftarrow \text{AC} + \text{DR.}$$

Subtraction in many machines is implemented through complement and addition operation as:

$$\begin{aligned} R3 &\leftarrow R1 - R2 \\ \Rightarrow R3 &\leftarrow R1 + (2\text{'s complement of } R2) \\ \Rightarrow R3 &\leftarrow R1 + (1\text{'s complement of } R2 + 1) \\ \Rightarrow R3 &\leftarrow R1 + R2 + 1 \end{aligned}$$

An increment operation can be symbolised as:

$$R1 \leftarrow R1 + 1$$

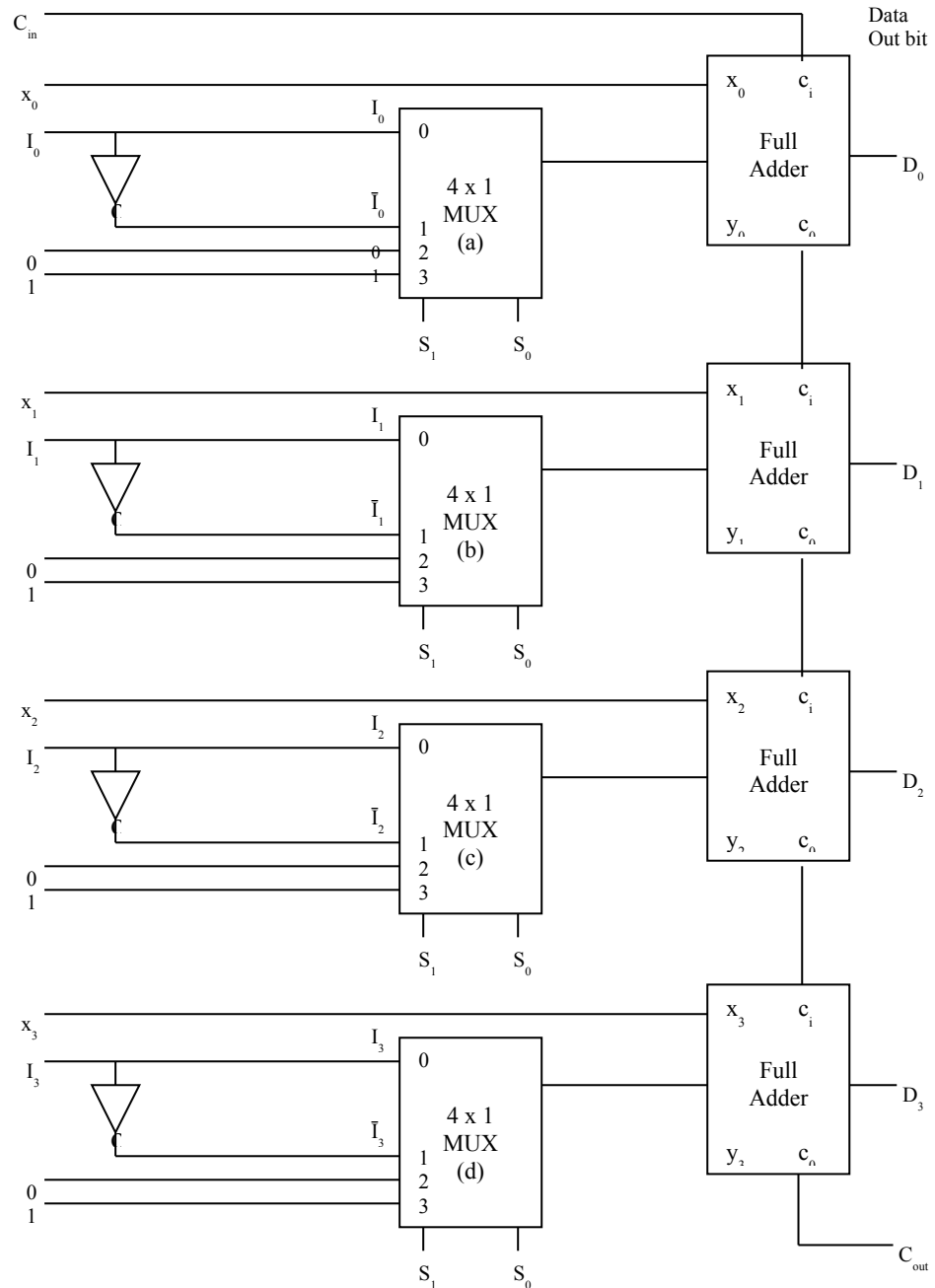
While a decrement operation can be symbolised as:

$$R1 \leftarrow R1 - 1$$

These increment and decrement operations can be implemented by using a combinational circuit or binary up/down counter. What about the multiplication and division operations? Are they not micro-operations? In most computers multiplication and division are implemented using add/subtract and shift micro-operations. If a digital system has implemented division and multiplication by means of combinational circuits then we can call these the micro-operations for that system.

### **Implementation of Arithmetic Circuits for Arithmetic Micro-operations**

An arithmetic circuit is normally implemented using parallel adder circuits. Figure 80 shows a logical implementation of a 4 bit arithmetic circuit. The circuit is constructed by using 4 full adders and 4 multiplexers.



**Figure 80: A four-bit arithmetic circuit**

Please note that each multiplexer (MUX) of the given circuit has two select inputs.  $S_0$  and  $S_1$ . We have drawn separate selection lines for each MUX for simplifying the circuit.

The two selection lines to the MUX along with the overall carry in ( $C_{in}$ ) bit determine the type of micro-operation to be performed by the circuit. How?

This four-bit circuit takes an input of two 4-bit data values and a carry in bit and outputs the four resultant data bits and a carry out bit. Please note in the circuit that one input, that is  $x$  is fed directly to the full adder,

while the second input I is fed through a multiplexer. The  $S_0$  and  $S_1$  inputs to the multiplexer determines what type of y input is to be selected for the full adder. Let us see how  $S_0$  and  $S_1$  determine the y input through the multiplexer with the help of the following truth table.

$S_1$	$S_0$	Output of 4 x 1 MUX				= y input	Comments
		MUX (a)	MUX (b)	MUX (c)	MUX (d)		
0	0	$I_0$	$I_1$	$I_2$	$I_3$	I	The data word I is input to Full Adder
0	1	$\bar{I}_0$	$\bar{I}_1$	$\bar{I}_2$	$\bar{I}_3$	$\bar{I}$	1's complement of I is input to Full Adder
1	0	0	0	0	0	0	Data word 0 is input to Full Adder.
1	1	1	1	1	1	$F_H$	Data Word 1111 is input to Full Adder.

**Figure 81: Multiplexer inputs and output of the arithmetic circuit of Figure 80**

Now let us discuss how by coupling  $C_{in}$  input bit with the selection input values we can obtain various micro-operations.

$S_1$	$S_0$	$C_{in}$	x	y	Equivalent Function	Equivalent Micro-operation	Micro-operation name
0	0	0	x	I	$F = x + I$	$R \leftarrow R_1 + R_2$	Add
0	0	1	x	I	$F = x + I + I$	$R \leftarrow R_1 + R_2 + 1$	Add with carry
0	1	0	x	$\bar{I}$	$F = x + \bar{I}$	$R \leftarrow R_1 + \bar{R}_2$	Subtract with borrow
0	1	1	x	$\bar{I}$	$F = x + (\bar{I} + 1)$	$R \leftarrow R_1 + 2's$	Subtract Complement of $R_2$
1	0	0	x	0	$F = x$	$R \leftarrow R_1$	Transfer
1	0	1	x	0	$F = x + 1$	$R \leftarrow R_1 + 1$	Increment
1	1	0	x	$F_H$	$F = x + F_H$	$R \leftarrow R_1 + (All\ is)$	Decrement
1	1	1	x	$F_H$	$F = x$	$R \leftarrow R_1$	Transfer

**Figure 82: Micro-operations which can be performed using the Arithmetic of Figure 80**

Some of the micro-operations given above need no explanation but let us discuss few of them.

In subtract with borrow we have:

$$F = x + \bar{I}$$

Or  $F = (x - 1) + (\bar{I} + 1)$   
 $(\bar{I} + 1)$  is 2's complement notation for  $-I$   
 $F = (x - 1) - I$

This implies that we are taking a borrow out of x before subtraction of I.

In decrement x we have function as

$$F = x + F_H$$

$$\text{Or } F = x + (F_H + 1) - 1$$

For this four bit words

$$\begin{array}{r} F_H = 1111 \\ + 0001 \\ \hline (1) 0000 \end{array}$$

↑  
carry out (ignored)

Therefore,

$$F = x + 0000 - 1$$

$$\Rightarrow F = x - 1$$

### 3.4.3 Logic Micro-operations

Logic operations are basically the binary operations which are performed on the string of bits stored in the registers. For a logic micro-operation each bit of a register is treated as a variable. A logic microoperation:

$R1 \leftarrow R1.R2$  specifies AND operation to be performed on the contents of R1 and R2 and store the results in R1. For example, if R1 and R2 are 8 bits registers and

R1 contains 10010011 and  
R2 contains 01010101

Then R1 will contain 00010001 after AND operation.

Some of the common logic micro-operations are AND, OR, NOT or Complement, Exclusive OR, NOR, NAND.

### Implementation of Logic Micro-operations

For implementation, let us first ask how many logic operations can be performed with two binary variables. We can have four possible combinations of the input of two variables. These are 00, 01, 10 and 11. Now, for all these 4 input combinations we can have  $2^4 = 16$  output combinations of a function. This implies that for two variables we can have 16 logical operations. The above stated fact will be clearer by going through the following figure.

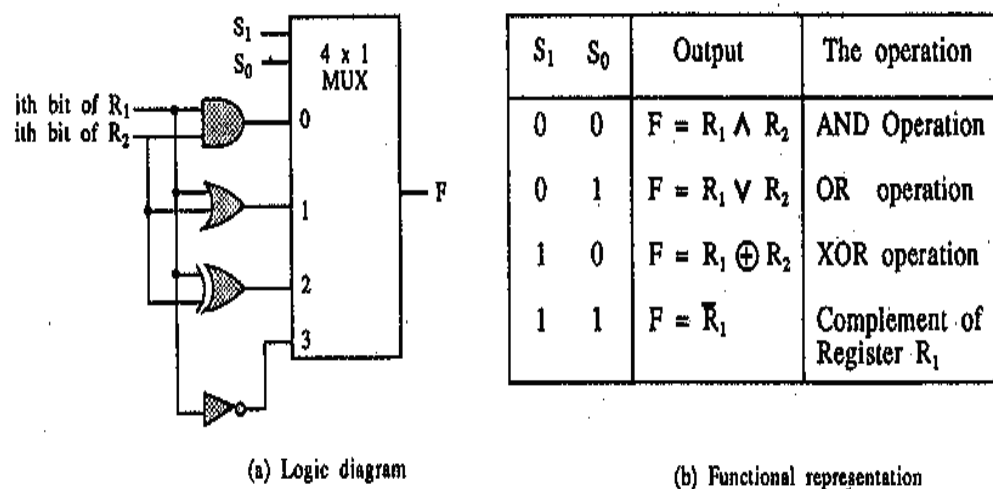


Output for Combination (xy)				Equivalent Boolean Function	Equivalent Micro-operation	Micro-operation Name
00	01	10	11	Function	Function	
0	0	0	0	$F_0$	$F_0 = 0$	$R \leftarrow 0$ Clear
0	0	0	1	$F_1$	$F_1 = x \cdot y$	$R \leftarrow R_1 \cup R_2$ AND
0	0	1	0	$F_2$	$F_2 = x \cdot \bar{y}$	$R \leftarrow R_1 \cup \bar{R}_2$ AND with complement $R_2$
0	0	1	1	$F_3$	$F_3 = x$	$R \leftarrow R_1$ Transfer of $R_1$
0	1	0	0	$F_4$	$F_4 = \bar{x} \cdot y$	$R \leftarrow \bar{R}_1 \wedge R_2$ AND with complement $R_1$
0	1	0	1	$F_5$	$F_5 = y$	$R \leftarrow R_2$ Transfer of $R_2$
0	1	1	0	$F_6$	$F_6 = x \bar{\wedge} y$	$R \leftarrow R_1 \oplus R_2$ Exclusive OR
0	1	1	1	$F_7$	$F_7 = x + y$	$R \leftarrow R_1 \vee R_2$ OR
1	0	0	0	$F_8$	$F_8 = \overline{(x + y)}$	$R \leftarrow \overline{R_1 \vee R_2}$ NOR
1	0	0	1	$F_9$	$F_9 = \overline{(x \oplus y)}$	$R \leftarrow \overline{(R_1 \oplus R_2)}$ Exclusive NOR
1	0	1	0	$F_{10}$	$F_{10} = \bar{y}$	$R \leftarrow \bar{R}_2$ Complement of $R_2$
1	0	1	1	$F_{11}$	$F_{11} = x + \bar{y}$	$R \leftarrow R_1 \vee \bar{R}_2$ $R_1$ OR with complement $R_2$
1	1	0	0	$F_{12}$	$F_{12} = \bar{x}$	$R \leftarrow \bar{R}_1$ Complement of $R_1$
1	1	0	1	$F_{13}$	$F_{13} = \bar{x} + y$	$R \leftarrow \bar{R}_1 \vee R_2$ $R_2$ OR with complement $R_1$
1	1	1	0	$F_{14}$	$F_{14} = \overline{(x \cdot y)}$	$R \leftarrow \overline{(R_1 \wedge R_2)}$ NAND
1	1	1	1	$F_{15}$	$F_{15} = 1$	$R \leftarrow \text{All } 1\text{'s}$ Set all the bits to 1

**Figure 83: Logic micro-operations on two inputs**

Please note that in the figure above the micro-operations are derived by replacing the  $x$  and  $y$  of Boolean function with registers  $R_1$  and  $R_2$  on each corresponding bit of the registers  $R_1$  and  $R_2$ . Each of these bits will be treated as binary variables.

In many computers only four: AND, OR, XOR (exclusive OR) and complement micro-operations are implemented. Other 12 micro-operations can be derived from these four micro-operations. Figure 84 shows one bit, that is the  $i^{\text{th}}$  bit stage of the four logic operations. Please note that the circuit consists of 4 gates and a 4 x 1 MUX. The  $i^{\text{th}}$  bits of register  $R_1$  and  $R_2$  are passed through the circuit. On the basis of selection inputs  $S_0$  and  $S_1$  the desired micro-operation is obtained.



**Figure 84:** The logic diagram of one stage of logic circuit

Let us now discuss how these four micro-operations can be used in implementing some of the important applications of manipulation of bits of a word, such as, changing some bit values or deleting a group of bits. We are assuming that the result of logic micro-operations goes back to Register  $R_1$  and  $R_2$  contain the second operand.

We will play a trick with the manipulations we are performing. Let us select 1010 as 4 bit data for register  $R_1$  and 1100 data for register  $R_2$ . Why? Because if you see the bit combinations of  $R_1$  and  $R_2$  they represent the truth table entries 00, 01, 10 and 11. Thus the resultant of the logical operation on them will tell us which logical micro-operation is needed to be performed for that data manipulation. The following table gives details on some of these operations.

<b>Operation name</b>	<b>What is the operation?</b>
Selective Set 1010 ( $R_1$ ) <u>1100</u> ( $R_2$ ) 1110	<p>Sets those bits in Register <math>R_1</math> for which the corresponding <math>R_2</math> bit is 1.</p> <p>The value 1110 suggests that a selective set can be done using logic OR micro-operation. Please note that all those bits of <math>R_1</math>, for which we have 0 bit in <math>R_2</math> have remained unchanged. The bits in <math>R_1</math> which need to be set selectively must have the corresponding <math>R_2</math> bits as 1.</p>
Selective Clear 1010 ( $R_1$ ) <u>1100</u> ( $R_2$ ) 0010	<p>Clear those bits in register <math>R_1</math> for which corresponding <math>R_2</math> bits are 1.</p> <p>The <math>R_1</math> value after the operations is 0010 which suggests that corresponding micro-operation is <math>R_1 \leftarrow R_1 \text{ AND } \bar{R}_2</math>. The bits in <math>R_1</math> which need to be cleared selectively must have corresponding <math>R_2</math> bits as 1.</p>
Selective Compliment 1010 ( $R_1$ ) <u>1100</u> ( $R_2$ ) 0110	<p>Complement those bits in register <math>R_1</math> for which the corresponding <math>R_2</math> bits are 1.</p> <p>The <math>R_1</math> value 0110 after the operation suggests that the selective complement can be done using exclusive – OR micro-operation. The bits in <math>R_1</math> which need to be complemented selectively must have the corresponding <math>R_2</math> bit as 1.</p>
Mask Operations 1010 ( $R_1$ ) <u>1100</u> ( $R_2$ ) 1000	<p>Clears those bits in Register <math>R_1</math> for which the Corresponding <math>R_2</math> bits are 0.</p> <p>The <math>R_1</math> value the operation is 1000 which suggests that the mask operation can be performed using AND micro-operation. However, the bits in <math>R_1</math> which are cleared or masked correspond to the bits on <math>R_2</math> having a 0 value. The mask operation is preferred over selective clear as most of the computers provide AND micro-operation while the micro-operation required for implementing selective clear is normally not provided in computers.</p>
Insert :	<p>For inserting a new value in a bit. It is a two step process:</p>

- step 1: Mask out the existing bit value  
 step 2: Insert the bit using OR micro-operation with the bit which is to be inserted.

**Example:**

Say contents of  $R_1 = 0011\ 1011$

Suppose, we want to insert 0110 in place of left most 0011 then:

	0011 1011	( $R_1$ before)
	0000 1111	( $R_2$ for masking)
	_____	Perform AND
operation (mask)	0000 1011	( $R_1$ after)
	Now insert: 0110 0000	( $R_2$ for insertion)
	_____	Perform OR
operation	01101011	$R_1$ after insert

Clear all the bits. Implemented by taking exclusive OR with the same number.

1101 ( $R_1$  before)

1101 ( $R_2$ )

0000 ( $R_1$  after clear by using exclusive OR)

The exclusive OR, thus, can also be used for checking whether two numbers are equal or not.

**3.4.4 Shift Micro-operations**

Shift is a useful operation which can be used for serial transfer of data. Shift operations can also be used along with other (arithmetic, logic, etc.) operations. For example, for implementing a multiply operation arithmetic micro-operation (addition) can be used along with shift operation. The shift operation may result in shifting the contents of a register to the left or right. In a shift operation a bit of data is input at the right most flip-flop while in shift right a bit of data is input at the left most flip-flop. In both cases a bit of data enters the shift register. Depending on what bit enters the register and where the shift out bit goes, the shifts are classified in three types. These are:

- Logical
- Arithmetic and
- Circular

In logical shift the data entering by serial input to left most or right most flip-flop (depending on right or left shift operations respectively) is a zero (0).

If we connect the serial output of a shift register to its serial input then we encounter a circular shift. In circular shift left or circular shift right information is not lost, but is circulated.

In an arithmetic shift a signed binary number is shifted to the left or the right. Thus, an arithmetic shift-left causes a number to be multiplied by 2; on the other hand a shift-right causes a division by 2. But as in division or multiplication by 2 the sign of a number should not be changed. Therefore, arithmetic shift must leave the sign bit unchanged. We have already discussed shift operations in Unit 1.

### **Implementation of a Shift Micro-operation**

As far as implementation of shift micro-operation is concerned it can be implemented by a left-right shift register having parallel load capabilities. Shift registers have already been discussed in Unit 2 of Module 1 of this course.

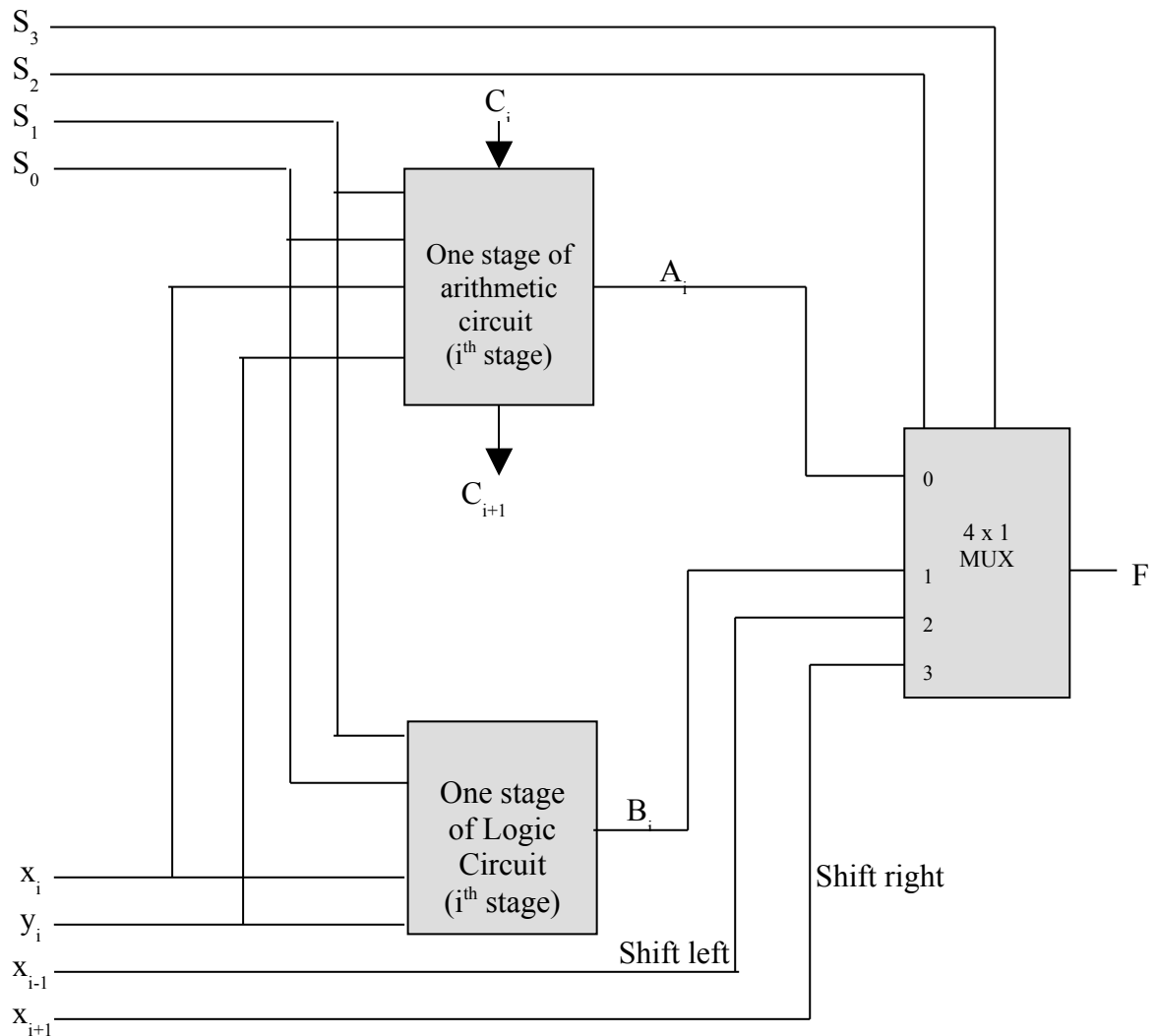
### **SELF-ASSESSMENT EXERCISE 2**

1. Draw a logic diagram for a 2 bit bus for 2 bit 2 register.
2. How is the memory read and write operation carried out using system bus?
3. Are multiplication and division arithmetic micro-operations?
4. What will be the value for  $R_2$  operand if:
  - (a) Mask operation, clears register  $R_1$
  - (b) Bits 1011 0001 is to be inserted in an 8 bit  $R_1$  register
5. What are the differences between circular and logical shift micro-operations?

### **3.4.5 Implementation of a Simple Arithmetic, Logic and Shift Unit**

As it is, we have discussed how all the micro-operations can be implemented individually. If we combine all these circuits, somehow, we can have a simple structure of ALU. In effect ALU is a combinational circuit whose inputs are contents of specific registers. The ALU performs the desired micro-operation as determined by control signals on the input and places the results in an output or destination register. The whole operation of ALU can be performed in a single clock pulse as it is a combinational circuit. The shift operation

can be performed in a separate unit but sometimes it can be made as a part of overall ALU. The following figure gives a simple structure of one state of an ALU.



**Figure 85: One stage of ALU with shift capacity**

Please note that in this figure we have given reference to two previous figures for arithmetic and logic circuits. This stage of ALU has two data inputs: the  $i^{\text{th}}$  bits of the registers to be manipulated. However, the  $(i-1)^{\text{th}}$  or  $(i+1)^{\text{th}}$  bit is also fed for the case of shift micro-operation of only one register. There are four selection lines which determine what micro-operation (arithmetic, logic or shift) on the input. The  $F_i$  is the resultant bit after a desired micro-operation. Let us see how the value of  $F_i$  changes on the basis of the four select inputs. This is shown in Figure 86.

Please note that in Figure 86, arithmetic micro-operations have both  $S_3$  and  $S_2$  bits as zero. Input  $C_i$  is important for only arithmetic micro-operations. For logic micro-operations  $S_3S_2$  values are 01. The values 10 and 11 cause shift micro-operations. For this shift micro-operation  $S_1$

and  $S_0$  values and  $C_i$  values do not play any role. Now let us wind up our discussions about the micro-operations with the discussion on instruction execution using micro-instructions.

$S_3$	$S_2$	$S_1$	$S_0$	$C_i$	F	Micro-operation	Name	
0	0	0	0	0	$F = x$	$R \leftarrow R_1$	Transfer	Addition with carry
0	0	0	0	1	$F = x + 1$	$R \leftarrow R_1 + 1$	Increment	
0	0	0	1	0	$F = x + y$	$R \leftarrow R_1 + R_2$	Addition	
0	0	0	1	1	$F = x + y + 1$	$R \leftarrow R_1 + R_2 + 1$	Addition with carry	
Arithmetic								
Micro-operations								
0	0	1	0	0	$F = x + \bar{y}$	$R \leftarrow R_1 + \bar{R}_2$	Subtract with borrow	Logic
0	0	1	0	1	$F = x + (\bar{y} + 1)$	$R \leftarrow R_1 - R_2$	Subtract	
0	0	1	1	0	$F = x - 1$	$R \leftarrow R_1 - 1$	Decrement	
0	0	1	1	1	$F = x$	$R \leftarrow R_1$	Transfer	
0	1	0	0	-	$F = x \cdot y$	$R \leftarrow R_1 \wedge R_2$	AND	Exclusive OR
0	1	0	1	-	$F = x + y$	$R \leftarrow R_1 \vee R_2$	OR	
0	1	1	0	-	$F = x \oplus y$	$R \leftarrow R_1 \oplus R_2$	Exclusive OR	
Micro-operations								
0	1	1	1	-	$F = \bar{x}$	$R \leftarrow \bar{R}_1$	Complement	
Micro-operations								
1	0	-	-	-	$F = \text{Shl}(x)$	$R \leftarrow \text{Shl}(R_1)$	Shift left	Shift
Micro-operations								
1	1	-	-	-	$F = \text{Shr}(x)$	$R \leftarrow \text{Shr}(R_1)$		Shift right
Micro-operations								

Figure 86: Micro-operation performed by ALU shown in Figure 85

### 3.5 Instruction Execution and Micro-Operations

Let us once again look at the instruction cycle since we have discussed instructions, registers sets and micro-operations. A simple instruction cycle will consist of the following steps:

- Fetching the instruction from the memory. This is also known as the fetch-cycle
- Decoding the instruction
- Finding out the effective address of the operand
- Executing the instructions. Normally decoding is also performed in an execution cycle
- Performing an interrupt cycle if an interrupt request is pending.

Let us explain how these steps of instruction can be broken down to micro-operations. For simplifying the discussion, let us assume that the machine has the structure as shown in Figure 77. In addition, the instruction set of the machine has only two addressing modes: direct and indirect memory addresses.

An instruction cycle in such a machine may consist of four sub-cycles. These four cycles are:

*The Fetch Cycle:* During this cycle the instruction which is to be executed next is brought from the memory to the CPU. The steps performed here are:

- The next instruction address is transferred from PC to MAR.  $MAR \leftarrow PC$  (Register transfer Micro-operation)
- MAR puts this signal on the address bus for main memory location selection, whereas the control unit uses a memory read signal. The result so obtained is placed on the data bus where it is accepted by the data register DR.  $DR \leftarrow (M)$  ( $M \rightarrow$  represents a memory read).
- The PC is incremented by one memory word length (Memory-read using bus. It may take more than one clock pulse depending on the  $t_{cpu}$  and  $t_{mem}$ ).  $PC \leftarrow PC + 1$   
This operation can be carried out in parallel to the above micro-operation.
- The instruction so obtained is transferred to the Instruction Register.  $IR \leftarrow DR$ .

*Indirect Cycle:* Once the instruction is fetched, the next step is to determine whether it requires memory references or register references or it is an input/output instruction. An input/output instruction can be used for: a transfer of information from or to AC; or checking the status of the I/O module; or enabling or disabling interrupts; or any other I/O related activity. We will not give details on these micro-operations in this unit. You can refer to further readings for details on these micro-operations. These instructions can go directly to execute the cycle.

The register reference instructions such as complement AC, clear AC etc. normally do not require any memory reference (assuming register indirect addressing is not being used) and can directly go to the execute cycle.

However, the memory reference instruction can use several addressing modes, depending on the type of addressing the effective address (EA)



of operands in the memory are calculated. In certain cases, the calculation of effective addresses requires one memory reference (for example in the case of indirect addressing). In such cases a special cycle which converts all the indirect addresses to direct addresses is required. This cycle is called as an indirect cycle.

The steps involved in the direct cycle are:

- Transfer the address bus of instruction to the MAR. This transfer can be achieved using DR only as DR and IR at this point of time contain the same value.

$MAR \leftarrow DR$  (Address)

- Once again a memory read operation as done in the fetch cycle is performed and the desired address of the operand is obtained in the DR.

$DR \leftarrow (M)$

- Transfer the address part so obtained in DR as the address part of instruction.

$IR$  (Address)  $\leftarrow DR$  (Address)

Thus, the purpose of the indirect cycle is to remove the indirection by converting the indirect address to the direct address.

*The “Execute” Cycle:* After the fetch and indirect cycle (if required) an instruction is ready to be executed. In the execute cycle the instruction gets actually executed. An execution cycle depends on the opcode. A different opcode will require different sequence of steps for the execution cycle. Therefore, let us discuss a few examples of the “execute” cycle of some simple instructions for the purpose of identifying some of the steps needed during this cycle. Let us start the discussions with a simple case of addition instruction. Suppose, we have an instruction: Add A which adds the content of memory location A to AC storing the result in AC. This instruction will be executed, following the steps:

- Transfer the address portion (this address in symbolic form is A) of the instruction to the MAR.

(Register transfer)  $MAR \leftarrow PC$

- Read the memory location A and bring the operand in the DR.

(memory-read)  $DR \leftarrow (M)$

- Add the DR with AC using ALU and bring the results back to AC.

(Add micro-operation)  $AC \leftarrow AC + DR$

Now let us try a complex instruction: a conditional jump instruction. Suppose the instruction ISZ A increments A and skips the next instruction if the contents of A have become zero. This is a complex instruction and requires intermediate decision making. The steps involved in this scheme are:

- Transfer the address portion of IR to the MAR.  

$$\text{MAR} \leftarrow \text{IR (Address)} \quad (\text{Register transfer})$$
- Read memory as we have done earlier. DR contains the operand A.  

$$\text{DR} \leftarrow (\text{M}) \quad (\text{Memory read})$$
- Transfer the contents of DR to AC as all the operation in the machine can be performed on AC.  

$$\text{AC} \leftarrow \text{DR} \quad (\text{Register transfer})$$
- Increment the AC.  

$$\text{AC} \leftarrow \text{AC} + 1 \quad (\text{Increment micro-operation})$$
- Transfer the content of AC to DR.  

$$\text{DR} \leftarrow \text{AC} \quad (\text{Register transfer})$$
- Store the contents of DR into the location A using MAR. This operation proceeds through as: Address bits are applied on address bus by MAR. The data is put into the data bus. The control unit providing control signal for memory write. Thus, resulting in a memory write at a location specified by MAR.  

$$(\text{M}) \leftarrow \text{DR} \quad (\text{Memory write})$$
- If the content of AC is zero then increment PC by one, thus skipping the next instruction. If  $\text{AC} = 0$  then  $\text{PC} \leftarrow \text{PC} + 1$  (Increment on a condition).

This operation can be performed in parallel to the memory write. Please note in the last step a comparison and an action is taken as a single step. This is possible as it is a simple comparison based on status flags.

Let us now take an example of a branching operation. Suppose, we are using the first location of subroutine to store the return address then the steps involved in this sub-routine call (CALL A) can be:

- Transfer the content of the address portion of IR to MAR.  

$$\text{MAR} \leftarrow \text{IR (Address)} \quad (\text{Register Transfer})$$
- Transfer the return address, which is the content of PC to DR.

$DR \leftarrow PC$  (Register transfer)

This micro-operation can be performed in parallel to the previous micro-operation.

- Transfer the branch address that is A to program counter.  
 $PC \leftarrow IR$  (address)      (Register transfer)
- Store the DR using MAR. Thus, the return address is stored at location A.

$(M) \leftarrow DR$  (Memory write)

This micro-operation can be performed in parallel to the previous micro-operation.

- Increment the PC as it contains address A, whereas the first instruction of subroutine starts from the next location.  
 $PC \leftarrow PC + 1$  (Increment)

Thus, the execute cycle is not a predictable cycle.

**The Interrupt Cycle:** After the completion of the execute cycle, the machine checks whether an interrupt that was enabled has occurred or not. If an enabled interrupt has occurred then an interrupt cycle is performed. The nature of the interrupt cycle varies from machine to machine. However, let us discuss one of the simplest illustrations of interrupt cycle events. Simple steps followed in interrupt cycle are:

- Transfer the contents of PC to DR as this is the return address from the interrupt and it is to be saved.  
 $DR \leftarrow PC$  (Register transfer)
- Place the address of location, where the return address is to be saved, into MAR. Please note that this address is normally predetermined in computers.  
 $MAR \leftarrow$  Address of location for saving return addresses.
- Store the contents of the PC in the memory using DR and MAR.  
 $(M) \leftarrow DR$  (Memory write)
- Transfer the address of the first instruction of the interrupt serving routine to the PC.  
 $PC \leftarrow$  service programs first instruction address interrupt  
 This micro-operation can be performed in parallel to the second micro-operation.

After completing the interrupt cycle the CPU will fetch the next instruction. During this time the CPU might be doing the interrupt processing or executing the user program. Please note that each instruction of an interrupt service routine is executed as an instruction in an instruction cycle. Please note that the instruction cycle discussed here involves many of the register transfer micro-operations. However, for an advanced structure, this requirement will be reduced because of the general nature of the registers.

Please note that for a complex machine the instruction cycle will not be as easy as this. You can refer to further readings for more complex instruction cycles.

#### **4.0 CONCLUSION**

In this unit, you have learnt the various possible structures of the CPU and the register organisation of the CPU. Also, you have learnt about various micro-operations. For better understanding an illustration of the implementation of a simple arithmetic, logic and shift unit was given.

#### **5.0 SUMMARY**

In this unit, we have discussed in details the register organisation and a simple structure of the CPU. After this we have discussed in details the micro-operations and their implementation in hardware, using simple logical circuits. While discussing micro-operations, our main emphasis was on simple arithmetic, logic and shift micro-operations, in addition to register transfer and memory transfer. However, the knowledge we have acquired about register sets and conditional codes, helped in giving us an idea, that conditional micro-operations can be implemented by simply checking flags and conditional codes. This idea will be clearer after we go through Units 3 and unit 4. We have completed the discussions on this unit with providing a simple approach of instruction execution with micro-operations. We will be using this approach for discussing control unit details in Units 3 and unit 4. You can refer to further readings for register organisation examples and for more details on the micro-operations and instruction execution.

#### **6.0 TUTOR -MARKED ASSIGNMENT**

1. What is an address register?

2. A machine has 20 general purpose registers. How many bits will be needed for the register address of this machine?
3. What is the advantage of having an independent set of conditional codes?
4. Can we store status and control information in the memory?  
State whether True or False:
5. An instruction cycle does not include an indirect cycle if the operands are stored in the register.  
True  False
6. For variable length instructions, each instruction fetched is of length = maximum instruction length/word size.  
True  False
7. Register transfer micro-operations are not needed for instruction execution.  
True  False
8. Interrupt cycle results only in jumping to an interrupt service routine. The actual processing of the instructions of this routine is performed in the instruction cycle.  
True  False

## 7.0 REFERENCES/FURTHER READINGS

- Mano, M. Morris (1993). *Computer System Architecture* (4<sup>th</sup> ed). Prentice Hall of India.
- Hayes, John, P.(1988). *Computer Architecture and Organisation* (2<sup>nd</sup> ed). McGraw-Hill International.
- Stallings William. *Computer Organisation and Architecture* (3<sup>rd</sup> ed). Maxwell Macmillan International Editions.
- Baron, Robert J. and Higbie Lee. *Computer Architecture*. Addison-Wesley Publishing Company.

## UNIT 3 ALU AND CONTROL UNIT ORGANISATION

### CONTENTS

- 1.0 Introduction
- 2.0 Objectives
- 3.0 Main Content
  - 3.1 ALU Organisation
    - 3.1.1 A Simple ALU Organisation
    - 3.1.2 Floating Point ALU
    - 3.1.3 Arithmetic Processors
  - 3.2 Control Unit Organisation
    - 3.2.1 Functional Requirements of a Control Unit
    - 3.2.2 Structure of Control Unit
    - 3.2.3 An Illustration of Control
    - 3.2.4 Hardware Control Unit
- 4.0 Conclusion
- 5.0 Summary
- 6.0 Tutor-Marked Assignment
- 7.0 References/Further Readings

### 1.0 INTRODUCTION

By now we have discussed the instruction sets and register organisation followed by a discussion on the micro-operations and a simple arithmetic logic unit circuit. In all the previous units, we have used a term, “control signals”, without any definition. In this unit we will first of all discuss the ALU organisation. We will also discuss the floating point ALU and arithmetic co-processors which are commonly used for floating point computations. This discussion will be followed by the discussions on the control unit, a component which causes all the components of the computer to behave effectively to achieve the basic objective, i.e. program execution. The control unit causes all the things to happen in the computer.

We will discuss the functions of a control unit, its structure, followed by the hardwired type of control unit. We will introduce the micro-programmed based control unit in the next unit. The details provided in Units 3 and 4 about control units can be supplemented by the details given in the further readings of the module.

## 2.0 OBJECTIVES

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

- discuss the basic organisation of ALU;
- discuss the requirements of a floating point ALU;
- define the term “arithmetic coprocessor”;
- define a control unit and its functions;
- describe a simple control unit organization; and
- define a hardwire control unit.

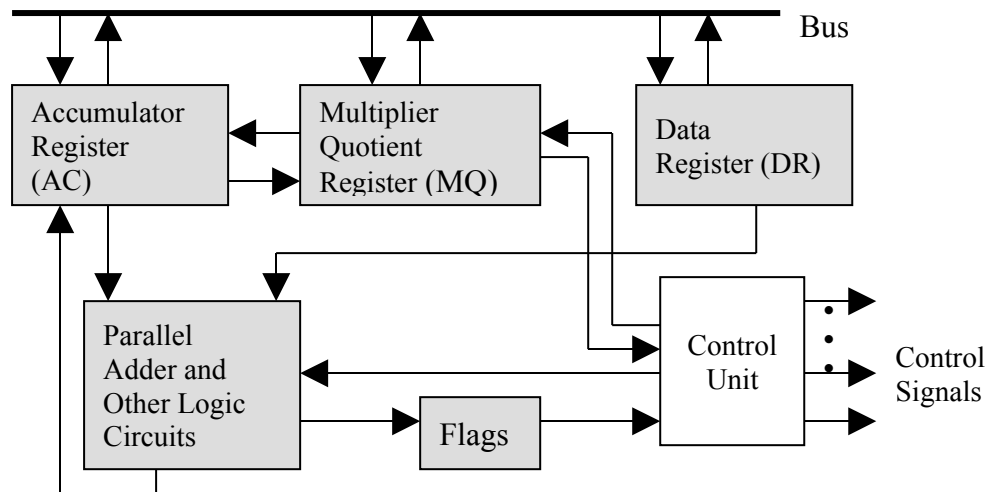
## 3.0 MAIN CONTENT

### 3.1 ALU Organisation

As discussed earlier, an ALU performs the simple arithmetic-logic and shift operations. The complexity of an ALU depends on the type of instruction set which has been realised for it. The simple ALUs can be constructed for fixed point numbers, on the other hand the floating point arithmetic implementation require more complex control logic and data processing capabilities, i.e. the hardware. Several micro-processor families utilise only fixed point arithmetic capabilities in the ALUs and for floating point arithmetic or other complex functions they may utilise an auxiliary special purpose unit. This unit is called arithmetic co-processor. Let us discuss all these in greater detail in this section.

#### 3.1.1 A Simple ALU Organisation

An ALU consists of various circuits which are used for execution of data processing micro-operations. But how are these ALU circuits are used in conjunction with other registers and control units? The simplest organisation in this respect for fixed a point ALU was suggested by John von Neumman in his IAS computer design. The structure of this simple organisation is given in Figure 87.



**Figure 87: The structure of a fixed point Arithmetic logic unit**

The organisations have three one word registers- AC, MQ and DR- which are used for data storage. Please note that the arithmetic, logic circuits have two inputs and only one output. In the present case the two inputs are AC and DR registers, while the output is AC register. AC and MQ are generally organised as a single AC.MQ register. This register is capable of left or right shift operations. Some of the micro-operations which can be defined on this unit are:

Addition	:	$AC \leftarrow AC + DR$
Subtraction	:	$AC \leftarrow AC - DR$
AND	:	$AC \leftarrow AC \wedge DR$
OR	:	$AC \leftarrow AC \vee DR$
Exclusive OR	:	$AC \leftarrow AC \oplus DR$
NOT	:	$AC \leftarrow \overline{AC}$

In this ALU organisation the multiplication and division are implemented using shift-add/subtract operations. The MQ (Multiplier-Quotient register) is a special register used for the implementation of multiplication and division. We are not giving the details of how this register can be used for implementing multiplication and division algorithms. For more details on these algorithms please refer to further readings. The MQ register stores the multiplier if multiplication is to be performed or the quotient if division is to be performed. For multiplication or division operations DR register stores the multiplicand or divisor respectively. The result of multiplication or division on applying certain algorithms can finally be obtained in an AC.MQ register combination. These operations can be represented as:

Multiplication	:	$AC.MQ \leftarrow DR \times MQ$
Division	:	$AC.MQ \leftarrow MQ \div DR$



DR is another important register which is used for storing a second operand. In fact, it acts as a buffer register which stores the data brought from the memory for an instruction. In machines where we have general purpose registers, for example Motorola 68020, any of the registers can be utilised as AC, MQ and DR.

### Bit Slice ALUs

It was feasible to manufacture smaller bits such as 4 or 8 bits fixed point ALUs on a single IC chip. If these chips are designed as expandable types then using these 4 or 8 bit ALU chips we can make 16, 32, 64 bit array like circuits. These are called bit-slice ALUs. The basic advantage of such ALUs is that they can be constructed for the desired word size. More details on bit-slice ALUs can be obtained from further readings.

### SELF ASSESSMENT EXERCISE 1

State whether True or False

1. A multiplication operation can be implemented as a logical operation. True  False
2. The multiplier-quotient register stores the remainder for a division operation. True  False
3. A word is processed sequentially on a bit slice ALU True  False

### 3.1.2 Floating Point ALU

A floating point ALU can implement floating point operations. But before discussing such a unit, let us first discuss briefly the floating point operations to get an idea of the requirements of such a unit.

#### Floating Point Arithmetic

Here is a brief introduction to floating point arithmetic and floating number representation.

A binary floating point number is represented in a normalised form, that is, the number is of the form  $\pm 0.$  (Significand starting with a non-zero bit)  $\times 2^{\pm (\text{Exponent Value})}$ . Figure 88 shows a format of a 32 bit floating point number.

Sign	Biased Exponent = 8 bits	Significand = 23 bits
------	--------------------------	-----------------------

Sign bit is for the significand.

**Figure 88: A floating point number representation**

The characteristics of a typical floating point representation of 32 bit in the above figure are:

- The leftmost bit is the sign bit of the number
- Mantissa or significand should be in normalised form
- The base of the number is 2
- A value of 128 is added to the exponent. (Why?) This is called a bias.

A normal exponent of 8-bit normally can represent exponent values as 0 to 255. However, as we are adding 128 in the biased exponent, thus, the actual exponent values represented will be  $-128$  to  $127$ .

Now, let us define the range which a normalised mantissa can represent. As for a normalised mantissa the leftmost bit cannot be zero, therefore, it has to be 1. Thus, it is not necessary to store this first bit and it is assumed implicitly for the number. Therefore, a 23-bit mantissa can represent  $23 + 1 = 24$ -bit significand.

Minimum value of the significand:

The implicit first bit as 1 followed by 23 zero's

0.1000 0000 0000 0000 0000 0000

Decimal equivalent =  $1 \times 2^{-1} = 0.5$

Maximum value of the significand:

The implicit first bit 1 followed by 23 one's

0.1111 1111 1111 1111 1111 1111

Decimal equivalent:

$$\begin{array}{r} \text{Binary: } 0.1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111 \\ + \underline{0.0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000} = 2^{-24} \\ 1.0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 = 1 \end{array}$$

So decimal equivalent of mantissa =  $(1 - 2^{-24})$

Therefore, in normalised mantissa and biased exponent form, the format of Figure 88 can represent a binary floating point number in the range:

Lowest negative number: Maximum significand and maximum exponent

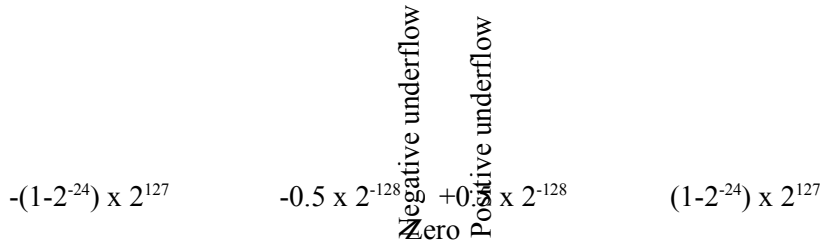
$$= -(1 - 2^{-24}) \times 2^{127}$$

Highest negative number: Minimum significand and Minimum exponent

$$= -0.5 \times 2^{-128}$$

Lowest positive number:  $0.5 \times 10^{-128}$

Highest positive number:  $(1-2^{-24}) \times 10^{127}$

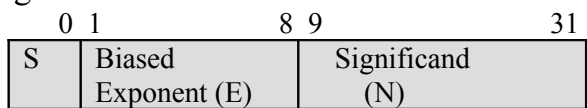


**Figure 89: Binary floating point number range for a 32 bit format**

In floating point numbers the basic trade-off is between range and accuracy. Negative numbers can be expressed down as small as significant. Positive numbers can be expressed up to some smaller. Let us examine an example which will clarify the term precision. Suppose we have one bit binary significand then we can represent only 0.10 and 0.11 in a normalised form. The values such as 0.101, 0.1011 and so on can not be represented as complete numbers. Either they have to be approximated or truncated and will be represented as either 0.10 or 0.11. Thus, it will create an error. The higher the number of bits in significand the better the precision will be.

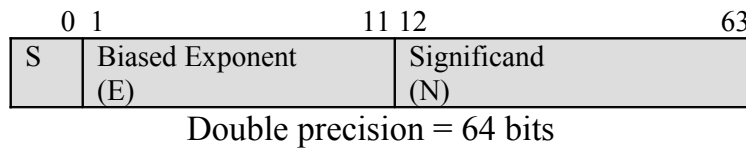
In floating point numbers for increasing both precision and range more numbers of bits are needed. This can be achieved by using the double precision format which is normally of 64 bits.

Institute of Electrical and Electronics Engineers (IEEE), a society which has created a lot of standards regarding various aspects of computer, has created IEEE standard 754 for floating point representations and arithmetic. The basic objective of developing this standard was to facilitate the portability of programs from one computer to another. This standard has resulted in the development of some standard numerical capabilities in various micro-processors. This representation is shown in Figure 90.



Single precision = 32 bits

- Implied base = 2
- Significand is in normalised form i.e. the first bit is implied and is 1
- S is sign bit



**Figure90: IEEE Standard 754 format**

Figure 91 gives the floating point numbers specified by the IEEE standard 754.

### Single Precision Numbers (32 bits)

<i>Exponent (E)</i>	<i>Significand (N)</i>	<i>Value/Comments</i>
255	Not equal to 0	Do not represent a number
255	0	- or + $\infty$ depending on sign bit
0 < E < 255	Any	$\pm (1.N) 2^{E-127}$ For example, if S is zero that is the positive number; N = 101 (rest 20 zeros) and E = 207 then the number is = $+(1.101)2^{207-127}$ $= + 1.101 \times 2^{80}$
0	Not equal to 0	$\pm (0.N)2^{-126}$
0	0	$\pm 0$ depending on the sign bit.

### Double Precision Numbers (64 bits)

<i>Exponent (E)</i>	<i>Significand (N)</i>	<i>Value / Comments</i>
2047	Not equal to 0	Do not represent a number
2047	0	- or + $\infty$ depending on the sign bit
0 < E < 2047	Any	$\pm (1.N) 2^{E-1023}$
0	Not equal to 0	$\pm (0.N)2^{-1022}$
0	0	$\pm 0$ depending on the sign bit.

**Figure 91: Values of floating point numbers as per IEEE standard 754**

Please note that IEEE standard 754 specifies plus zero and minus zero and plus infinity and minus infinity. Floating point arithmetic is stickier than fixed point arithmetic. For floating point addition and subtraction we have to:

- Check whether a typical operand is zero

- Align the significand such that both significands have same exponent
- Add or subtract the significand only and finally
- Ensure the significand is normalised again

These operations can be represented as:

$$x + y = (N_x \times 2^{E_x - E_y} + N_y) \times 2^{E_y}$$

And 
$$x - y = (N_x \times 2^{E_x - E_y} - N_y) \times 2^{E_y}$$

Here, the assumption is that exponent of x ( $E_x$ ) is greater than exponent by y ( $E_y$ ).  $N_x$  and  $N_y$  represent mantissa of x and y respectively.

While for multiplication and division operations the significand needs to be multiplied or divided respectively however, the exponent is to be added or to be subtracted respectively. In case we are using a bias of 128 or any other bias for exponent then on addition of exponents since both the exponents have bias, the bias gets doubled. Therefore, we must subtract the bias from the exponent on addition of exponents. However, a bias is to be added if we are subtracting the exponents. The division and multiplication operation can be represented as:

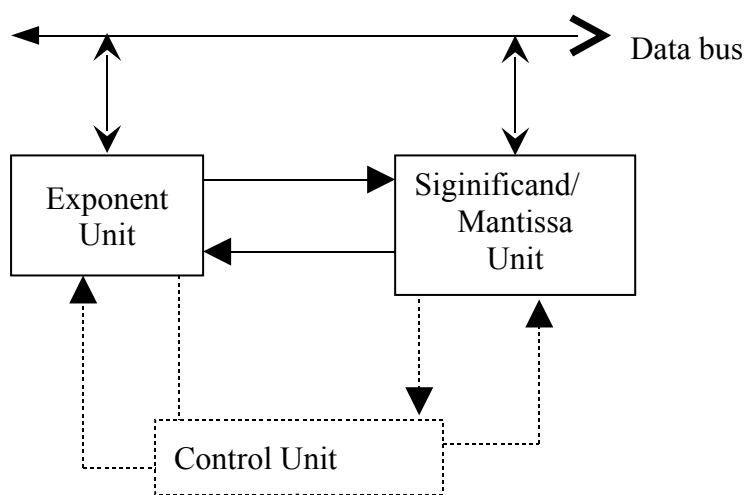
$$x \times y = (N_x \times N_y) \times 2^{E_x + E_y}$$

$$x \div y = (N_x \div N_y) \times 2^{E_x - E_y}$$

For more details on floating point arithmetic you can refer to the further readings.

### Floating Point ALU

A floating point ALU is implemented using two loosely coupled fixed point arithmetic circuits. Figure 92 shows a simple structure of such a unit.



**Figure 92: A floating point arithmetic unit**

The two units can be termed as exponent and mantissa units. The mantissa unit has to perform all the four arithmetic operations on the mantissa. Therefore, a general purpose fixed-point mantissa unit may be used for this purpose. However, for the exponent unit we only need circuits to add, subtract and compare the exponents. Thus, a simple circuit containing these functions will be sufficient. The comparison can be performed by a comparator or by simple subtraction operation.

The implementation details of floating point arithmetic on floating point ALUs can be seen from the further readings.

**3.1.3 Arithmetic Processors**

The very first question in this regard is: “What is an arithmetic processor?” and, “What is the need for arithmetic processors?”. A typical CPU needs most of the control and data processing hardware for implementing non-arithmetic functions. As the hardware costs are directly related to chip area, a floating point circuit being complex in nature is costly to implement. They are normally not included in the instruction set of a CPU. In such systems, floating point operations are implemented by using software routines. This implementation of floating point arithmetic is definitely slower than the hardware implementation. Now, the question is whether a processor can be constructed only for arithmetic operations. A processor if devoted exclusively to arithmetic functions can be used to implement a full range of arithmetic functions in the hardware at a relatively low cost. This can be done in a single IC. Thus, a special purpose arithmetic processor, for performing only the arithmetic operations, can be constructed. Although, this processor physically is separate, yet it will be utilised by the CPU to execute a class of arithmetic instructions. Please note that in the absence of arithmetic processors, these instructions may be executed using the slower software routines by the CPU itself. Thus, this auxiliary processor enhances the speed of execution of programs having lot of complex arithmetic computations. In addition, it also helps in reducing program complexity, as it provides more instructions to a machine. Some of the instructions which can be assigned to arithmetic processors can be: add, subtract, multiply and divide fixed and floating point

numbers of various lengths, exponentiation; logarithms and trigonometric functions.

How can this arithmetic processor be connected to the CPU? Two mechanisms are used for connecting an arithmetic processor to the CPU.

If an arithmetic processor is treated as one of the I/O or peripheral unit then it is known as a *peripheral processor*. The CPU sends data and instruction to the peripheral processor which performs the required operations on the data and communicates the results back to the CPU. Peripheral processors have several registers to communicate with the CPU. These registers may be addressed by the CPU as input/output register addresses. The CPU and peripheral processors are normally quite independent and communicate with each other by exchange of information using data transfer instructions. This data transfer instructions must be specific instructions in the CPU. This type of connection is called a loosely coupled processor.

On the other hand if the arithmetic processor has a register and instruction set which can be considered an extension of the CPU registers and an instruction set then it is called a tightly coupled processor. Here the CPU reserves a special subset of code for the arithmetic processor. In such a system the instructions meant for the arithmetic processor are fetched by the CPU and decoded jointly by the CPU and the arithmetic processor, and finally executed by the arithmetic processor. Thus, these processors can be considered a logical extension of the CPU. Such attached arithmetic processors are termed as co-processors. Let us discuss them in more details.

### The Peripheral Processor

An example of one such arithmetic processor is the AMD 9511/12 one chip floating point processor. The advantage of this processor is that it can be utilised with any CPU, while the disadvantages are that it needs explicitly programmed and slow communication links with the CPU. These processors can be utilised as given in the following figure.

	<i>Performer</i>	<i>Instructions executed</i>	<i>Processing Details</i>
1.	CPU	Data-transfer instructions	These instructions help in sending a set of input operands and commands, e.g. arithmetic operations,

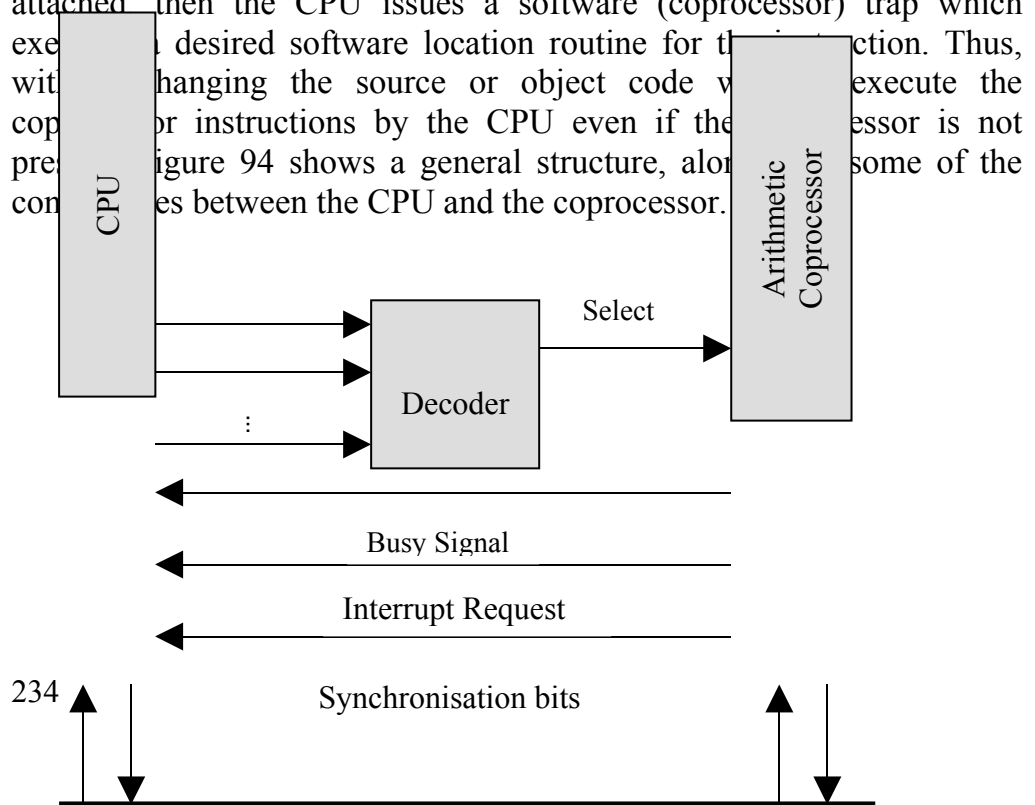
- |                         |   |  |
|-------------------------|---|--|
|                         |   | to the peripheral processor.   |
| 2. Peripheral Processor | Decodes & executes the command received from CPU using the operands.                                | Results are generated and placed in registers directly accessible to the CPU                       |
| 3. CPU                  | Checks status by polling a status register or by receiving interrupt from the peripheral processor. | It determines whether the peripheral processor has completed the task.                             |
| 4. CPU                  | Data transfer instruction is executed   | CPU obtains the results from the peripheral processor by executing this data transfer instruction. |

**Figure 93: Communication between the CPU and the peripheral processor**

In certain implementations the CPU has to wait for the peripheral processor to finish, therefore, it remains idle for that time.

### Coprocessors

Coprocessors, unlike peripheral processors, are tailor made for a particular family of CPUs. Normally, each CPU is designed to have a coprocessor interface. The control signal circuit of the CPU is designed for the interface beforehand. Special instructions are earmarked for execution by the coprocessors. These coprocessors instructions can appear in any assembly or machine language program similar to any other instruction. The CPU hardware takes care of the instruction execution by the coprocessors. The coprocessor instruction can be executed even if a coprocessor is not present, by already stored software routines at pre-determined memory locations. If a coprocessor is not attached then the CPU issues a software (coprocessor) trap which executes the desired software location routine for the instruction. Thus, without changing the source or object code we can execute the coprocessor instructions by the CPU even if the coprocessor is not present. Figure 94 shows a general structure, along with some of the connections between the CPU and the coprocessor.





**Structure Figure 94: General of CPU-Co-processor**

As both processors are directly linked, they can be synchronised easily. The control lines between them are few. The data transfer between the processors can take place through the system bus. The CPU may act as the master of coprocessor. The registers of coprocessor can be written into or read by the CPU directly as they do for the main memory. Sometimes it is useful to allow the coprocessor to control the bus as in such cases it can control data transfer from memory or it can initiate data transfer to the CPU.

In case the coprocessor can control the system bus, then it is allowed to decode and identify the instructions at the same time the CPU is doing so. The coprocessor then can execute the instructions meant for it directly. This type of approach is followed in 8087 arithmetic coprocessor of 8086; while in some CPUs, only the CPU can decode the coprocessor instructions. This is the case for the 68881 floating point coprocessor of Motorola 68000 series. A CPU can employ more than one different coprocessor.

### 3.2 Control Unit Organisation

Having discussed the ALU, we will move on to a very important component of the CPU i.e. the control unit.

The basic responsibilities of the control unit are to control:

- data exchange of the CPU with the memory or I/O modules
- internal operations in the CPU such as:
  - moving data between registers (register transfer operations)
  - making ALU to perform a particular operation on the data
  - regulating other internal operations.

But how does a control unit control the above operations? What are the functional requirements of the control unit? What is its structure? Let us explore answers to these questions in the subsequent sub-sections.

### 3.2.1 Functional Requirements of a Control Unit

Let us first try to define the functions which a control unit must perform in order to get the things to happen. However in order to define the functions of a control unit, one must know what resources and means it has at its disposal. A control unit must know about the:

1. basic components of the CPU
2. micro-operation the CPU performs

The CPU of a computer consists of the following basic function components:

- **The Arithmetic-Logic Unit (ALU):** which performs the basic arithmetical and logical operations?
- **Registers:** which are used for information storage within the CPU.
- **Internal Data Paths:** These paths are useful for moving the data between two registers or between a register and the ALU.
- **External Data Paths:** The roles of these data paths are normally to link the CPU registers with the memory or I/O modules. This role is normally fulfilled by the system bus.
- **The Control Unit:** which causes all the operations to happen in the CPU?

The micro-operations performed by the CPU can be classified as:

- Micro-operations for register to register data transfer
- Micro-operations for register to external interface (i.e. in most cases system bus data transfer)
- Micro-operations for external interface to register data transfer

- Micro-operations for performing arithmetic and logic operations. These micro-operations involve the use of registers for input and output.

The basic responsibility of the control unit is the fact that the control unit must be able to guide the various components of the CPU to perform a specific sequence of micro-operations to achieve the execution of an instruction.

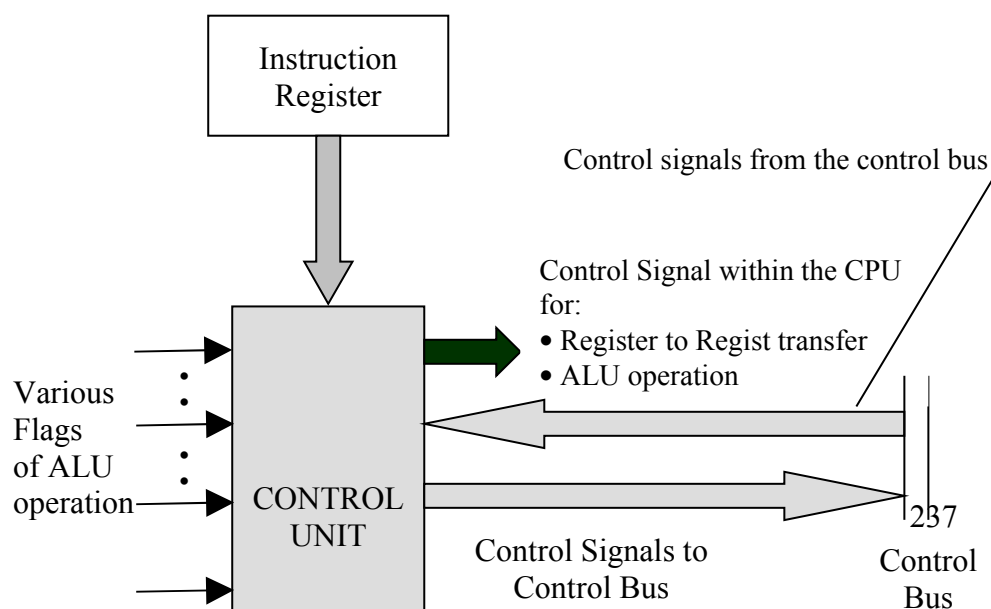
What are the functions which a control unit performs to make an instruction execution feasible? The instruction execution is achieved by executing micro-operations in a specific sequence. For different instructions this sequence may be different. Thus, the control unit must perform two basic functions:

- Cause the execution of a micro-operation.
- Enable the CPU to execute a proper sequence of micro-operations which is determined by the instruction to be executed.

But how are these two tasks achieved? The control unit generates control signals which in turn are responsible for achieving both tasks. But how are these control signals generated? We will answer this question in the later sections. But first let us discuss a simple structure of a control unit.

### 3.2.2 The Structure of Control Unit

A control unit has a set of input values on the basis of which it produces an output control signal which in turn performs micro-operations. These output signals control the execution of a program. A general model of a control unit is shown in Figure 95.



Master clock

**Figure 95: A general model of the control unit**

In the model given above the control unit is a black box which has certain inputs and outputs.

The *inputs* to the control unit are:

- **The Master Clock Signal:** This signal causes micro-operations to be performed. In a single clock cycle either a single or a set of simultaneous micro-operations can be performed. The time taken in performing a single micro-operation is also called the processor cycle time in some machines. However, some micro-operations, such as the memory read, may require more than one clock cycle if  $t_{\text{mem}}/t_{\text{cpu}}$  is greater than one.
- **The Instruction Register:** The operation code (opcode) which normally includes the addressing mode bits of the instruction helps in determining the various cycles to be performed and hence determines the related micro-operations which are needed to be performed.
- **Flags:** Flags are used by the control unit for determining the status of the CPU. The outcomes of a previous operation on ALU can also be detected using flags. For example, a zero flag will help the control unit while executing an instruction ISZ (skip the next instruction if zero flag is set). In case the zero flag is set then the control unit will issue control signals which will cause program counter (PC) to be incremented by 1. In effect, skipping the instruction, which the CPU was supposed to execute next.
- **Control Signals from the Control Bus:** Some of the control signals are provided for the control unit through the control bus. These signals are issued from outside the CPU. Some of these signals are interrupt signals and acknowledgment signals.

On the basis of the input signals the control unit activates certain output signals which in turn are responsible for the execution of an instruction. These output control signals are:

- **Control Signals which are required within the CPU:** These control signals cause two types of micro-operations, viz. for data transfer from one register to another; and for performing an ALU operation using input and output registers.
- **Control Signals to the Control Bus:** The basic purpose of these control signals is to bring or to transfer data from the CPU register to the memory or I/O modules. These control signals are issued on the control bus to activate a data path.

Now, let us discuss the requirements from such a unit. A prime requirement for the control unit is that it must know all the instructions to be executed and also the nature of the results along with the indication of possible errors. All this is achieved with the help of flags, opcodes, clock and some control signals to itself.

A control unit contains a clock portion, whose job is to provide clock pulses. This clock signal of the control unit is used for measuring the timing of the micro-operations. In general, the timing signals from the control unit are kept sufficiently long keeping in mind the propagational delays of signals within the CPU along various data paths. As within the same instruction cycle, different control signals are generated at different times for performing different micro-operations, therefore, a counter can be utilised with the clock to keep the count. However, at the end of each instruction cycle the counter should be reset to the initial condition. Thus, the clock to the control unit must provide counted timing signals. Examples, of the functionality of control units along with timing diagrams are given in the further readings. We will not discuss the timing diagrams in this module.

How are these control signals applied to achieve the particular operation? The control signals are applied directly as the binary inputs to the logic gates of the logic circuits. Do you remember the enable input defined in Unit 2 of Module-1 or the select inputs of multiplexers? All these inputs are the control signals which are applied to select a circuit (in case of enable) or a path (in case of MUX) or any other operation in the logical circuits.

One of the responsibilities of the control unit is to keep track of the instruction cycle. Therefore, the control unit can determine when which micro-operation is to be performed. Let us discuss this with the help of an example in the following subsection.

### 3.2.3 An Illustration of Control

Suppose we have an accumulator machine with the following registers (in the usual functions they do)

- Accumulator (AC)
- Instruction Register (IR)
- The Program Counter (PC)
- Memory Address Register (MAR)
- Data Register (DR)

Figure 96 is a simple representation of such a machine with the requirement of control signals.

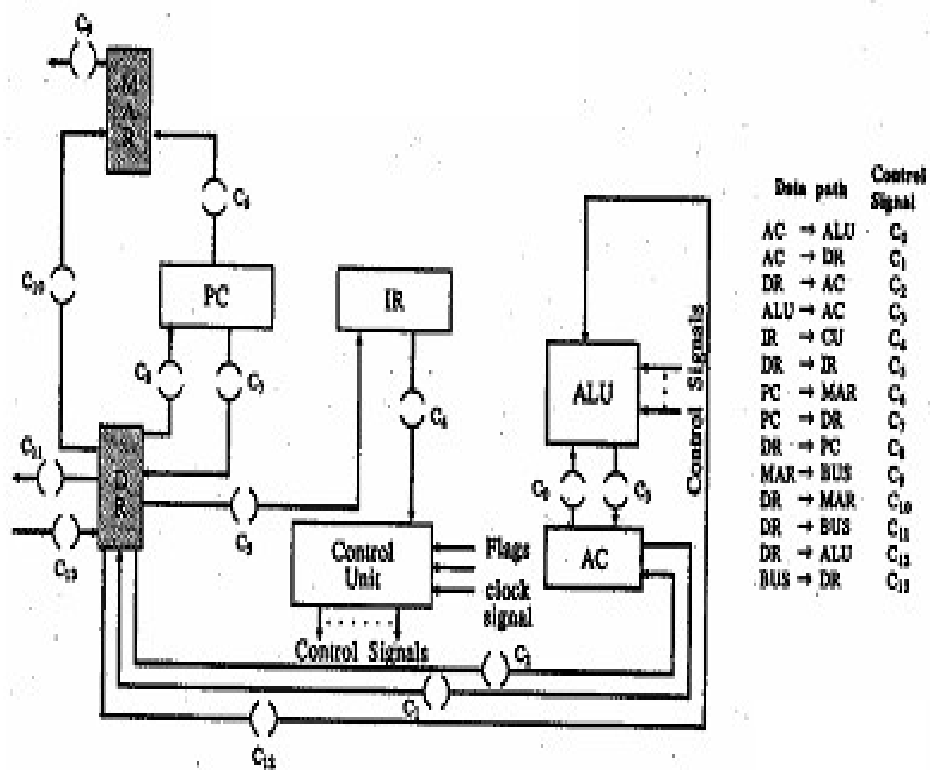


Figure 96: A typical CPU with some control signals

The register transfers for this machine are:

Register	Input from	Output to	Comments
AC	ALU, DR	DR,	AC receives and sends data to DR or

IR	DR	ALU CU	to ALU. Instruction register receives instruction fetched in DR. The op-code of the instruction is used by the control unit for generating control signals for instruction execution.
PC	DR	MAR, DR	Program counter is loaded by the address supplied by DR. However, it can send the next instruction address to MAR; or it can send the current address for storage, through DR in the case of a subroutine call.
MAR.	DR, PC	System Bus	MAR stores the address of the memory unit or the I/O module which is in turn passed on to the system bus. It can be loaded by PC or DR
DR	System Bus AC	System Bus, ALU, AC, IR, MAR, PC	DR can receive data on system bus from memory unit or I/O module or data for storage from AC. However, it can output data to system bus for storage or to ALU as second operand or to AC as first operand or MAR in case of address or to the program counter when it contains the address of a called subroutine.

Thus, the various sets of data transfer for the machine in Figure 96 are:  
(Source → Destination)

AC → ALU : AC → DR : DR → AC : ALU → AC;  
 IR → CU : DR → IR :  
 PC → MAR : PC → DR : DR → PC :  
 MAR → (BUS) : (PC → MAR) : DR → MAR :  
 DR → (BUS) : DR → ALU : (DR → AC) : (DR → IR);  
 (DR → MAR) : (DR → PC) : BUS → DR : (AC → DR).

Each of the above transfers requires a control signal. However, the data transfer entries given in branches are occurring again. Thus, we require 14 control signals as shown in Figure 10. A connection line in this figure does not indicate a control signal but it indicates a data path which exists between the two components. The direction of the arrow indicates the direction of data transfer. All the keys on the data transfer paths are triggered by a control signal which is marked there. Please note that for purposes of simplicity we have not shown how these control signals

come from the control unit. However, you must keep in mind that all these fourteen control signals are only a subset of control signals generated by the control unit. These control signals can go to their separate destinations. The control signals which are normally needed for different destinations can be categorised as:

- Control signals activating a data path: These control signals open a gate temporarily. This allows a flow of data on the path controlled by that gate. Some of these control signals are shown in the figure given above.
- Control signals activating a function/operation of ALU: These signals open various logic devices and gates inside the ALU. These signals are shown as a group of signals in Figure 96.
- Control signals activating the system bus: These control signals may activate a control line for a simple add instruction, where indirect addressing has been used.

### Fetch Cycle

<i>Timing</i>	<i>Micro-operation</i>	<i>Comment</i>	<i>Control Signals needed</i>
$t_1$	$MAR \leftarrow PC$	Memory address register is assigned the content of the programme counter.	$C_6$
$t_2$	$DR \leftarrow (BUS)$	Read the contents. An additional control signal (not shown here) is needed to active memory read.	$C_{13}$ and control signal for memory read. MAR address input is applied on the system bus.
	$PC \leftarrow PC + 1$	Increment program counter	Control signal for incrementing PC
$t_3$	$IR \leftarrow DR$	Transfer the fetched instruction into instruction register.	$C_5$ .

### Indirect Cycle

$t_1$	$MAR \leftarrow IR$ (ADDRESS)	Assign address from the instruction register	$C_{10}$
-------	----------------------------------	--	----------



to MAR. This content can be acquired from DR as it stores same contents as of IR at present.

t <sub>2</sub>	DR ← (BUS)	-	C <sub>13</sub> and control signal for memory read. MAR address input is applied on the system bus.
t <sub>3</sub>	IR (ADDRESS (ADDRESS))	← DR	C <sub>5</sub>

**Execute Cycle**

t <sub>1</sub>	MAR ← IR (ADDRESS)		C <sub>10</sub>
t <sub>2</sub>	DR ← (BUS)		C <sub>13</sub> and control signal for memory read and address is applied on the BUS.
t <sub>3</sub>	AC ← AC + DR		Control signals for performing this operation along with C <sub>0</sub> , C <sub>3</sub> and C <sub>12</sub>

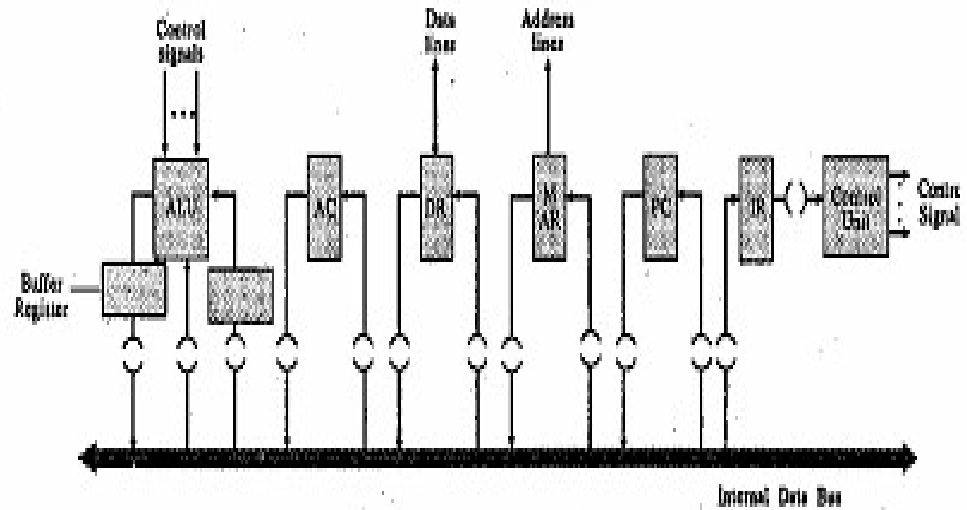
**Interrupt Cycle**

t <sub>1</sub>	DR ← PC	Store the content of PC in the memory at an address specified by machine	C <sub>7</sub>
t <sub>2</sub>	MAR ← ADDRESS OF LOCATION OF STORING RETURN ADDRESS PC ← Address of the interrupt service		Control signals for performing these operations.

program's  
first  
instruction

$t_3$  (BUS)  $\leftarrow$  DR      Save the DR  $C_{11}$  and control signal contents which enables memory write

But do we have an internal CPU organisation as shown in Figure 96? Such an organisation will have a large range of data paths hence it will be very complex if more registers are there in the CPU. A simple yet effective solution in such a case will be to use an internal data bus within CPU. This type of organisation is used in microprocessors, such as INTEL 8085. The organisation of the machine which we have shown in Figure 96, if developed with the internal data bus will be very much simplified. This is shown in Figure 97.



**Figure 97: A bus based CPU with same control signals**

In the case above, for each register one input and one output line is controlled by a gate and is connected to the data bus (except for IR where we have only input). The required data transfer can be initiated by activating two gates (one output and one input). We have provided two temporary storage with the ALU; otherwise the output of the ALU will go back to its input, as both input and output gates of the ALU are open for processing, which is undesirable.

The advantages of using internal bus arrangements are:

- Simple data path interconnections which mean easy layout for control
- Saving of CPU space as inter-register connection space is minimised. This is very useful in the case of microprocessors.

The next question about the control unit is: How can we implement a control unit such that it generates the necessary control signals? The control units are implemented using two general approaches. These are called:

- Hardwired control unit, and
- Micro-programmed control unit

We will give only a very brief account of hardwired control units in this unit. The micro-programmed control unit is discussed in details in the next unit of this module.

### **3.2.4 The Hardwired Control Unit**

A hardwired control unit is implemented as logic circuits in the hardware. The inputs to control unit are: the instruction register, flags, timing signals and control bus signals. On the basis of these inputs the output signal sequences are generated. Thus, output control signals are functions of inputs. Thus, we can derive a logical function for each control signal. This, however, will be very complicated if we have a large control unit. The implementation of all the combinational circuits may be very difficult. Therefore, a new approach microprogramming was used. This approach will be discussed in the next unit.

## **4.0 CONCLUSION**

This unit has discussed two basic organisations of the ALU and the control unit. To understand these better you were also taken through the structure of the control unit, types of control unit, classes of ALU such as floating point ALU, arithmetic processors, etc.

## **5.0 SUMMARY**

In this unit, we have discussed two main components of the CPU, the ALU and the control unit. We have explained the concepts of the basic ALU structure, floating point ALUs and coprocessors. Coprocessors, in today's computers, are used widely and help in implementing graphical and other computation intensive applications. As far as control unit is concerned, we have discussed a simple structure of a control unit along with an example. More details on these aspects with examples can be seen from the further readings. In this unit we have also introduced the concept of a hardwired control unit. A microprogrammed control unit which is more commonly used is the topic of the next unit.

## 6.0 TUTOR-MARKED ASSIGNMENT

- Find out the range of a number of the following floating point representation:

Base → 2  
 Sign → 1 bit  
 Exponent → 4 bits. Bias of 8 is used  
 Significand → 3 bits

Assume the normalised mantissa representation

- State whether True or False
  - A double precision number is used when accuracy requirements are higher. True  False
  - A zero cannot be represented in IEEE standard 754 formats. True  False
  - On multiplication of floating point numbers, the value of bias needs to be subtracted after adding the two exponents. True  False
  - The exponent unit of floating point ALU must perform all the four arithmetical operations. True  False
- What is an arithmetic processor? Compare the co-processor with the peripheral processor.
- What are the inputs to the control unit?
- How does a control unit control the instruction cycle?
- What is the importance of an internal data bus?
- What is a hardwired control unit?

## 7.0 REFERENCES/FURTHER READINGS

Mano, M. Morris (1993). *Computer System Architecture* (3<sup>rd</sup> ed). Prentice Hall of India.

Hayes, John P. (1988). *Computer Architecture and Organisation* (2<sup>nd</sup> ed). McGraw-Hill International Editions,

Stallings, William. *Computer Organisation and Architecture* (3<sup>rd</sup> ed). Maxwell Macmillan International Editions.

Baron, Robert J. and Highbie Lee. *Computer Architecture*. Addison-Wesley Publishing Company.

Tanenbaum, Andrew S. (1993). *Structural Computer Organisation* (3<sup>rd</sup> ed). Prentice Hall of India.

## UNIT 4 MICROPROGRAMMED CONTROL UNIT

### CONTENTS

- 1.0 Introduction
- 2.0 Objectives
- 3.0 Main Content
  - 3.1 What is a Micro-Programmed Control Unit?
  - 3.2 Wikes Control
  - 3.3 The Microinstruction
    - 3.3.1 Types of Microinstruction
    - 3.3.2 Control Memory Organisation
    - 3.1.3 Microinstruction
  - 3.4 A Simple Structure of Control Unit
  - 3.5 Microinstruction Sequencing
  - 3.6 Microinstruction Execution
  - 3.7 Machine Startup
- 4.0 Conclusion
- 5.0 Summary
- 6.0 Tutor-Marked Assignment
- 7.0 References/Further Readings

### 1.0 INTRODUCTION

This is the last unit of the module on CPU organisation. We have already discussed in the earlier units, the instruction sets, register set, ALU organisation and control unit organisation. In this unit, we will discuss the microprogrammed control unit, which is quite popular in modern computers because of flexibility by Wikes control unit. We will also discuss microinstruction and a simple structure of such a unit. Finally, we will visit the concepts involved in microinstruction sequencing and execution. Finally, we will examine the aspects of a machine startup.

The discussions given in this unit are at a conceptual level. You can however refer to the further readings for implementation-based examples of the microprogrammed control unit.

### 2.0 OBJECTIVES

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

- define the microprogrammed control unit;
- define the term “microinstruction”;
- identify types and formats microinstruction;
- discuss the control memory;

- explain the working of a microprogrammed control unit;
- define the microinstruction addressing;
- explain microinstruction; and
- identify the process of a machine startup.

### 3.0 MAIN CONTENT

#### 3.1 What is a Microprogrammed Control Unit?

As discussed earlier in Unit 3, the control unit seems to be a very simple unit; however, if we try to implement such a unit through hardware, we are bound to face problems. The hardwired control unit lacks flexibility in design. In addition, it is quite difficult to design, test and implement as in many computers the number of control lines is in hundreds.

Is there any alternate approach of implementing a control unit? What about a programming approach for implementing the control unit? Can we somehow implement the sequence of execution of micro-operations through a program? Such a program will consist of instructions, with each instruction describing:

- one or more micro-operations to be executed, and
- the information about the microinstruction to be executed next.

Such an instruction is known as a microinstruction and such a program is known as a microprogram or firmware. The firmware is a mid-way between hardware and software. Firmware, in comparison to hardware is easier to design, whereas in comparison to software, it is difficult to write. A control unit provides a set of control signal lines, distributed throughout the CPU, with each of the lines representing a zero or one. Therefore, a microinstruction is made responsible for generating control signals for desired control lines to implement a desired micro-operation. For example, to implement a register to register transfer operation, output of the source register and input of destination register need to be enabled by the respective control signal lines via a microinstruction. Thus, each microinstruction can generate a set of control signals on the control lines which in turn implement one or more micro-operations. A set of control signals with each bit representing a single control line is called a control word.

Thus, a microinstruction can cause execution of one or more micro-operations, and a sequence of microinstructions, that is microprogram, can cause execution of an instruction. The microprograms are mostly stored in read only memory, known as control store or control memory, as alterations in the control store are generally not required after the production of the control unit. What about having read-write control

memory? In such a memory, the instruction set of a computer can be changed by simply modifying the microprograms for various op-codes. Such a computer can be tailored for specific applications on the basis of microprograms. A computer which has a writable control memory is said to be “dynamically microprogrammable” as the content of the control memory for such a computer can be changed under program control. The control memory is a word organised unit with each word representing a microinstruction. Please note that the computers which have microprogrammed control units, have two separate memories- a main memory and the control memory.

How will the microprogrammed control unit control the instruction execution?

Well, the first point to mention here is that each computer instruction fetched from the main memory leads to the initiation of a series of microinstructions from the control memory. These microinstructions issue micro-orders to the CPU for an instruction fetch, the calculation of the effective address of operands, and the execution of the instruction and then prepare it again for fetching the next instruction from the main memory. Thus, for each instruction, one must determine the series of microinstructions which will be needed to get that instruction executed. This series of microinstruction will be different for different instructions.

### **Advantage of the Microprogrammed Control Unit**

Since the microprograms can be changed relatively easily, microprogrammed control units are very flexible in comparison to hardwired control units.

### **Disadvantages**

- Hardware costs more because of the control memory and its access circuitry.
- This is slower than the hardwired control unit because the microinstructions are to be fetched from the control memory which is time consuming.

But these disadvantages are gradually getting phased out as the new memory technologies are cheap and high-speed memories are gradually becoming common.



### 3.2 Wilkes Control

In 1951, Wilkes had proposed the use of microprogram control unit. In Wilkes design a microinstruction has two major components:

- The control field, and
- The address field

The control field indicates the control lines which are to be activated and the address field provides the address of the next microinstruction to be executed. Figure 98 shows a simple example of Wilkes control unit design.

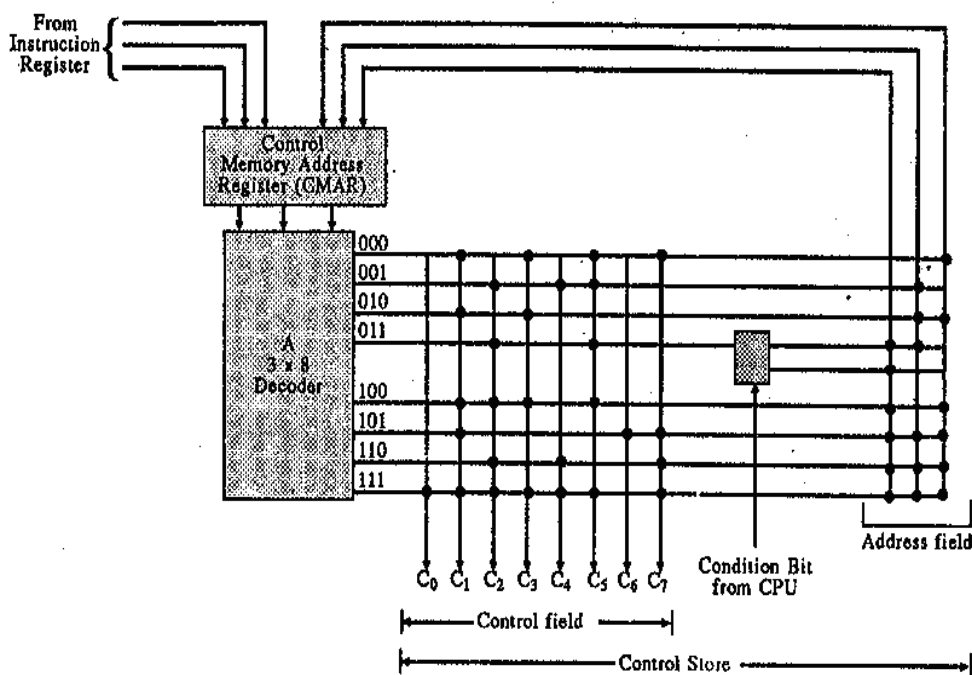


Figure 98: Wilkes control unit

The control memory in this control is organised as a program logic array like a matrix made of diodes, a simple electronic device. This is a partial matrix and it consists of two components- the control signals and the address of the next microinstruction. The control memory access register (CMAR) can be loaded by the instruction code register or by the address field of the control matrix. The control memory address register on taking an input from the instruction register provides a 3-bit address to the 3 x 8 decoder. This is an entry point address to the control memory. On the basis of this address, the decoder activates one of the eight output lines (horizontal). This activated line in turn generates control signals and the address of the next microinstruction to be executed. This address is once again fed to the CMAR, resulting in the activation of another control line and address field. This cycle is repeated till the execution of the instruction is achieved. For example, in

the given figure, the instruction register's op-code 000 causes the decoder to have an entry address for a machine instruction in the control memory at line 000. The decoder activates the lines in the sequence given below: (Please note that this is valid only for the diagram given here which is only an indicative example and not a complete Wikes control unit).

<b>Decode line activated</b>	<b>Control Signals generated</b>	<b>Address of next microinstruction</b>
000	$C_1, C_3, C_5, C_7$	001
001	$C_2, C_4, C_5$	010
010	$C_1, C_3$	011
011	$C_2, C_5$	?

Please note the “?” above. How do we get the address of the next microinstructions to be executed on activating the decode line 011? We have two possible options in the Figure 98. A typical requirement of a control unit is that it must respond to an external control condition. Thus, making conditional jumps possible within a microprogram. This is demonstrated in the Wikes control of Figure 98. The external condition switch causes the control unit to follow one of the two available paths.

EITHER

011	$C_2, C_5$	If external condition is true then 110
110	$C_2, C_4, C_7$	111
111	$C_0, C_1, C_2, C_3, C_4, C_5, C_7$	This may cause loading of next instruction in IR

OR

011	$C_2, C_5$	If external condition is false then 100
100	$C_1, C_2, C_3, C_5$	101
101	$C_1, C_6, C_7$	111
111	$C_0, C_1, C_2, C_3, C_4, C_5, C_7$	

Wikes not only proposed this control scheme but had proposed an example structure for the first microprogrammed control processing unit. However, we will not go into the details of this structure here. You can refer to further readings for this structure.

**SELF-ASSESSMENT EXERCISE**

1. What is firmware? How is it different from software?
2. State whether True or False
  - (a) A microinstruction can initiate only one micro-operation  
True  False
  - (b) A control word is equal to a machine word.  
True  False
  - (c) A dynamically microprogrammable control memory is writable  
True  False
  - (d) Microprogrammed control is faster than hardwired control.  
True  False
  - (e) Wikes control does not provide a branching microinstruction.  
True  False
3. What will be the sequence of decode lines activated in the Wikes control example of Figure 98 If the entry address for a machine instruction is 010 and the conditional bit values is true?

**3.3 The Microinstruction**

After discussing the Wikes control, we have some idea of the microprogrammed control unit. The key to a microprogrammed control unit is a microinstruction. So, let us explore the microinstruction more in this section.

A microinstruction, as defined earlier, is an instruction of a microprogram. It specifies one or more micro-operations, which can be executed simultaneously. On executing a microinstruction a set of control signals are generated which in turn cause the desired micro-operation/micro-operations to happen.

**3.3.1 Types of Microinstruction**

In general, the micro-instructions can be categorized in two general types. These are branching and non-branching. A non-branching microinstruction is the one, in which the next micro-instruction which is executed is the one following the current micro-instruction. However, this sequence of micro-instructions is relatively small and lasts only for 3 or 4 micro-instructions.

A conditional branching micro-instruction is a desirable instruction. The condition which is to be tested is a conditional variable or a flag generated by an ALU operation. Normally the branch address is contained in the microinstruction itself

### 3.3.2 Control Memory Organisation

The next important question about microinstructions is; how are they organised in the control memory? One of the simplest ways to organise the control memory is to arrange microinstructions for various sub cycles of the machine instruction in the memory. Figure 99 shows such an organisation.

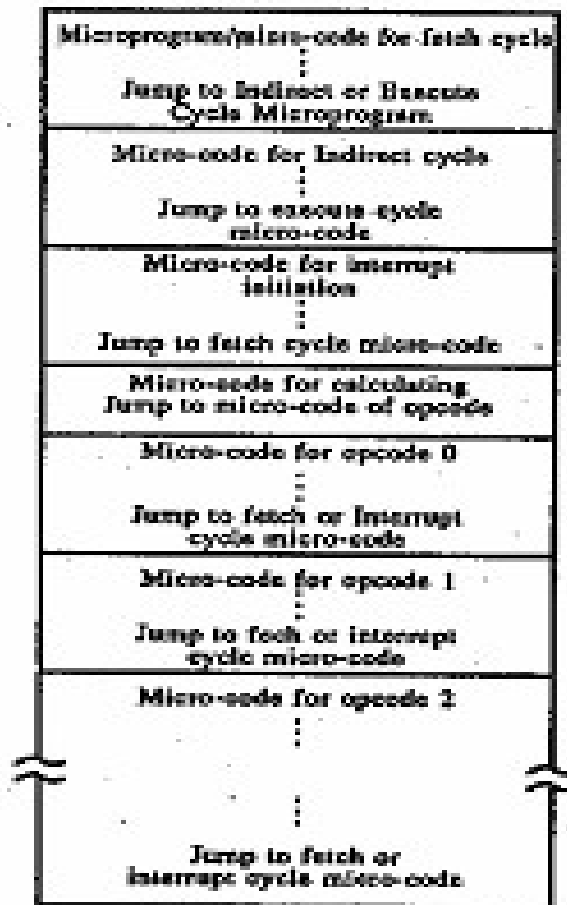


Figure 99: The control memory organisation

Please note the use of branching microinstructions in the organisation. Let us give an example of control unit organisation. Let us take a **machine instruction**: Branch on zero. This instruction causes a branch to a specified main memory address in case the result of the last ALU operation is zero, that is, the zero flag is set. The pseudocode of the microprogram for this instruction can be:

Test “zero flag”. ; If SET branch to ZERO

Unconditional branch to NON-ZERO

Zero : (The microcode which causes the replacement of the program counter with the address provided in the instruction)

Branch to interrupt of fetch cycle

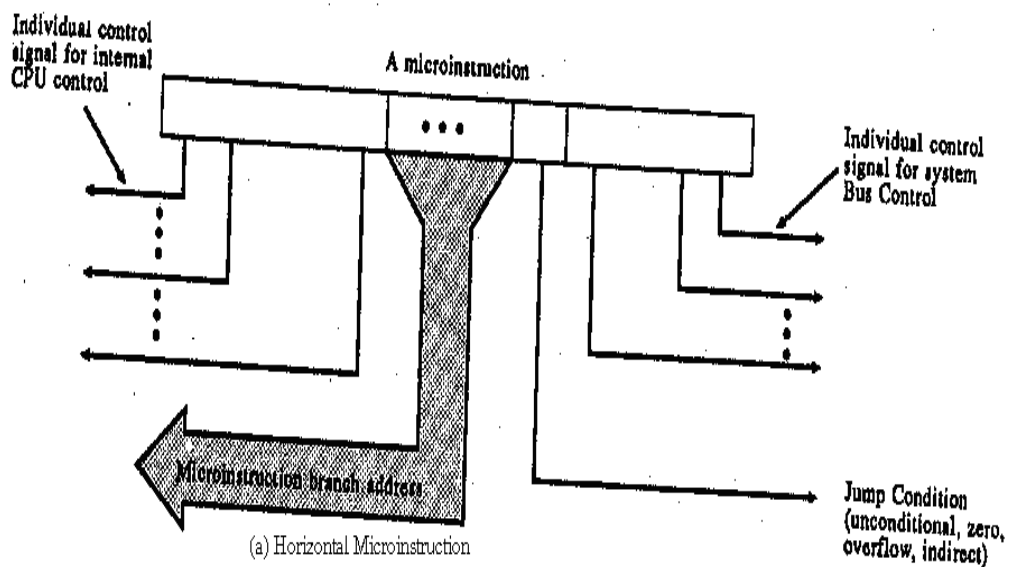
Non-zero : (The microcode which may set flags if desired, indicating that the branch has not taken place)

Branch to interrupt or fetch cycle.

The control memory provides a concise description of the various operations of the control unit. It may also define the sequences of micro-operations which are to be performed during an instruction execution.

### 3.3.3 Microinstruction Formats

Now, let us focus on how a microinstruction may be organised. The two widely used formats for microinstructions are horizontal and vertical. In the horizontal microinstruction each bit of the microinstruction represents a micro-order or a control signal which directly controls a single bus line or sometimes a gate in the machine. However, the length of such a microinstruction may be hundreds of bits. A typical horizontal microinstruction with its related fields is shown in Figure 100(a) below.



(a) A horizontal microinstruction

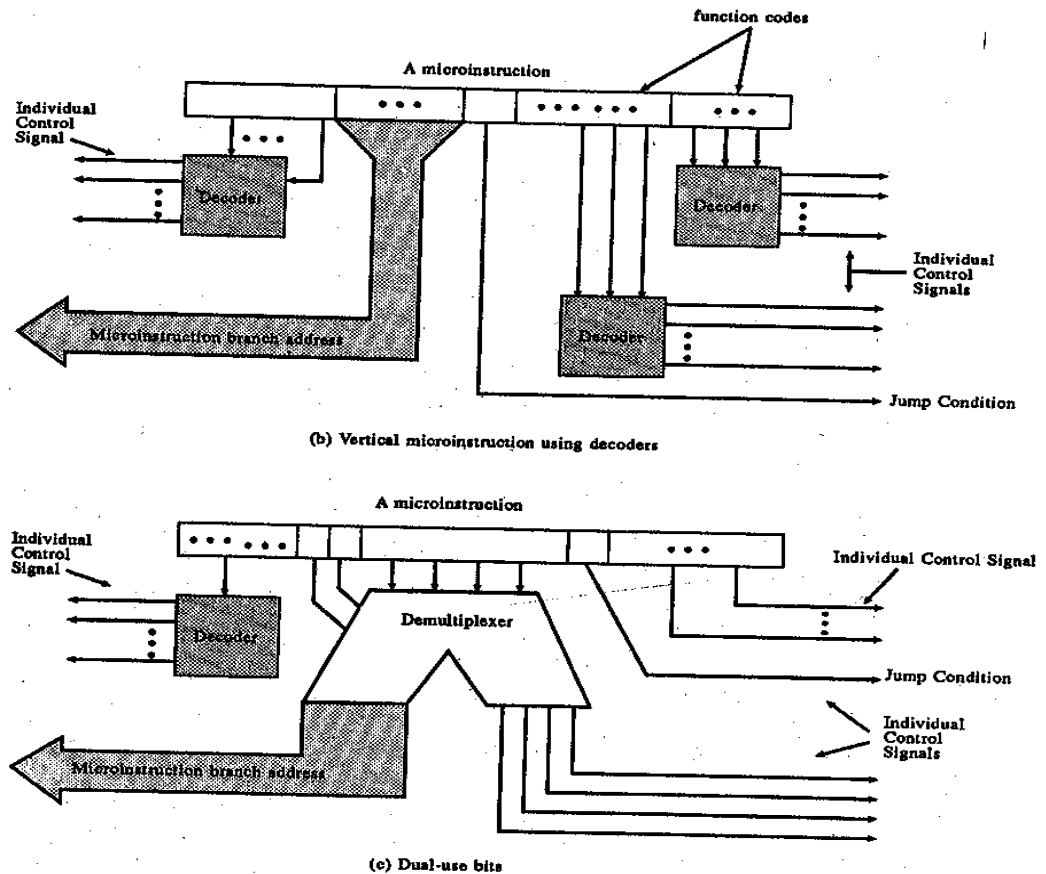


Figure 100: Microinstruction formats

In vertical microinstructions many similar control signals can be encoded into a few microinstruction bits. For example, for 16 ALU operations which may require 16 individual microcoders in a horizontal microinstruction, only 4 encoded bits are needed in a vertical microinstruction. Similarly, in a vertical microinstruction only 3 bits are needed to select one of the 8 registers. However, these encoded bits need to be passed from respective decoders to get the individual control signals. This is shown in Figure 100(b). Some of the microinstructions may be passed through a de-multiplexer causing selected bits to be used for a few different locations in the CPU (Refer to Figure 100(c)). For example, a 6 bit field in a microinstruction can be used as the branch address in a branching microinstruction. However, these bits may be utilised for some other control signals in a non-branching microinstruction. In such a case the de-multiplexer can be used. The vertical microinstructions are normally of the order of 32 bits. In certain control units, several levels of control are used. For example, a field microinstruction or the machine instruction may hold the address of a read only memory which holds the control signals. This secondary ROM can hold large address constants such as interrupt service routine addresses.

In general, horizontal control units are faster yet they require wide instruction words; whereas, vertical control units although require decoders, however, are shorter in length. Most of the systems use neither purely horizontal nor purely vertical microinstructions.

### 3.4 A Simple Structure of a Control Unit

We will now about the structure and functioning of a simple microprogrammed control unit. This will also clarify many of the concepts we have discussed so far. Figure 101 shows the simple structure of such a unit.

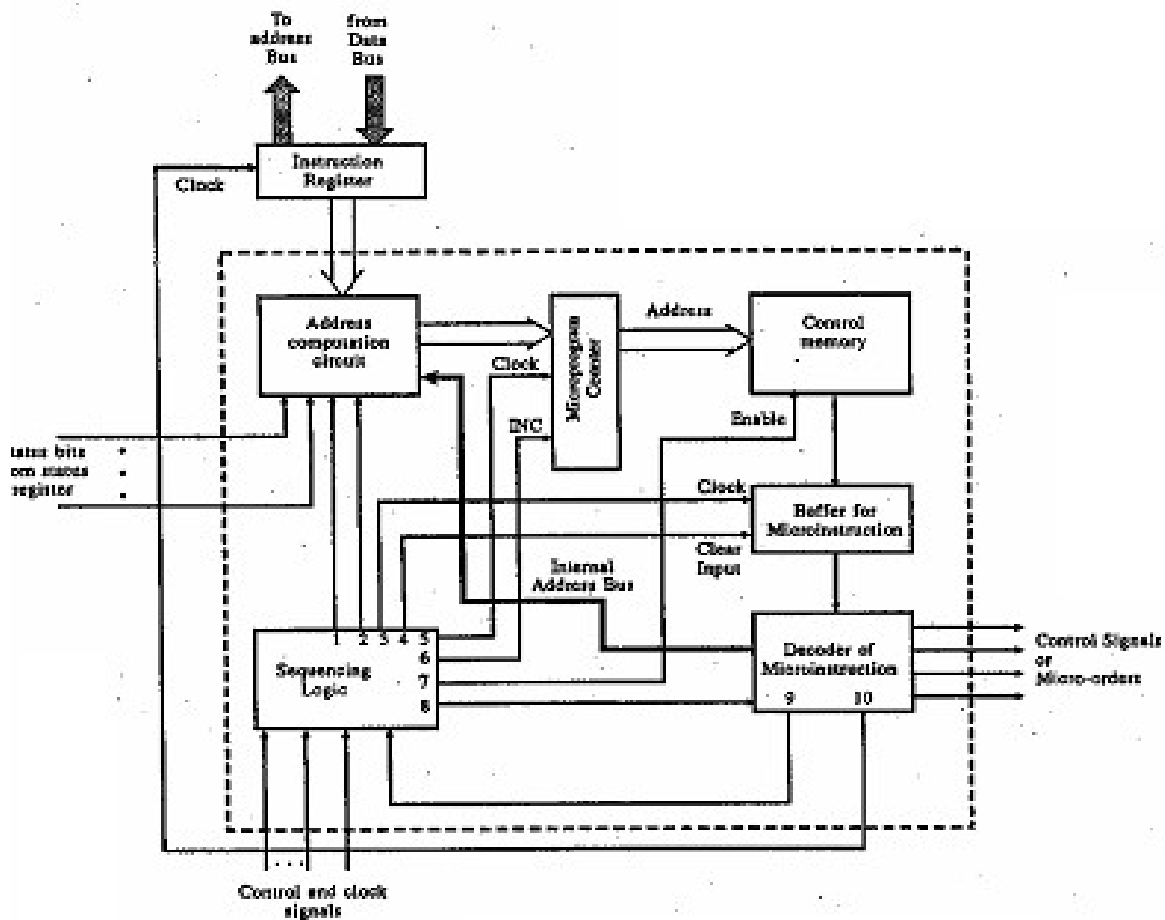


Figure 101: A simple microprogrammed control unit.

A fetched instruction is stored in the instruction register (IR). In certain machines the IR only holds the op-code of the instruction to be executed. The control memory is normally implemented in the ROM and not in the main memory. This memory is utilised for keeping the microprograms for all op-codes. In addition, the control memory may contain routines for instructions fetch, interrupt initiation and other machine startup routines. The address of the next microinstruction is computed by the address-computation circuit. We will answer this

question shortly. The role played by the microprogram counter register is similar to that of a program counter in the CPU. The microprogram counter holds the address of the next microinstruction to be executed. The microinstruction buffer holds the microinstruction which is currently being executed. The decoder of the microinstruction generates micro-orders on the basis of the microinstruction and op-code of the current instruction. The sequencing logic plays a key role in synchronising all these component of the microprogrammed control unit. The sequencing logic also plays an important role in the startup of the machine, the circumstances, sequencing logic controls the control unit.

How does this control unit function? In this control unit address computation circuitry calculates the address of the next microinstruction to be executed. The address of the microinstruction to be executed next is stored in the microprogram counter. It can acquire this address from:

- Address computation circuitry when the clock signal (No. 5) is enabled.
- Incrementing its own content by enabling INC signal (No. 6.). The microinstruction buffer can be cleared this time by enabling signal (No. 4).

Now, the control store can be read. The enable input (signal No.7) enables the control memory to transfer the microinstruction stored in the location addressed by the microprogram counter into the microinstruction buffer. Please note that for this, the transfer clock signal (No.3) should also be enabled so that the microinstruction buffer register can be loaded. This microinstruction is then decoded and respective micro-orders are generated in the microinstruction decoder on receiving a control signal (No.8).

This sequence continues till one microprogram is executed; in other words one machine instruction is executed, after this the microinstruction decoder issues the micro-order which enables the clock of the instruction register, control signal (No. 10). This signal causes the loading of the next machine instruction to be executed into the instruction register. One of the responsibilities of the control unit, once an instruction is fetched and the operand addresses calculated, will be to calculate the entry point address for that op-code in the control memory; that is, the address of the first microinstruction for the microprogram for the given op-code.

The responsibility of calculating the entry point address of the microprogram for a given op-code lies with the address computation



circuitry. Then, the sequencing logic increments the microprogram counters to acquire the next microinstruction in sequence. After executing the complete microprogram, a branch to another microprogram routine may take place. Thus, the last instruction of a microprogram is a branching microinstruction. This branching microinstruction holds an address and the control signals for the address computation circuitry. On the basis of these two pieces of information, the address computation circuitry calculates the address of the next microinstruction to be executed, in effect causing a branch.

Now let us discuss how the branching and non-branching microinstruction can be implemented in Figure 101. The control signal (No. 9) plays a key role in implementing branching and non-branching microinstructions. This has been decoded as a non-branching (value of signal No. 9 = 0) or a branching (value =1) microinstruction in the following way:

If the value of signal  $9 = 0 \Rightarrow$  a non-branching microinstruction Sequencing logic activates control signal 6 which increments the microprogram counter.

Otherwise (that is value of signal  $9 = 1$ ),  $\Rightarrow$  a branching microinstruction Sequencing logic activates control signal which causes the address calculated in the address computation circuitry to be transferred to the microprogram counter.

But, how does the address computation circuitry calculate the address of the next microinstruction? In order to answer this question, we will discuss how the address computation circuitry may function.

The address computation circuitry is involved in:

- Determining the entry point address of the control store depending on the opcode extracting the branch address indicated by a branching microinstruction. This address is supplied on the internal address bus.

This can be achieved by using two control signals 1 and 2 from the sequencing logic which tells the address computation circuitry to calculate the address of the next microinstruction on the basis of opcode or branching microinstruction respectively. In the case of the conditional branch, the address provided by the internal address bus may be modified, depending on the status bits supplied by the ALU. Thus, unconditional and conditional jumps can be implemented on the basis of the sequencing logic signals and status bits from the ALU. The address

of the next microinstruction is calculated by using the address obtained on the internal address bus. But, how can the address computation circuitry modify the address it gets from the internal address bus? Well, by using simple logical operations such as masking, ORing, etc.

### 3.5 Microinstruction Sequencing

As discussed earlier, the two basic functions of the control unit are microinstruction sequencing and microinstruction execution. We have explained some of the concepts in this regard. Let us discuss these aspects in greater detail in this and subsequent sections. Let us first try to find out the concerns for designing sequencing techniques.

The first concern which is applicable in general is “to minimise the size of the control memory”. The second concern is “to execute a microinstruction as fast as possible”. This implies that the address of the next microinstruction should be calculated at a fast rate.

Now, let us find out how these two concerns can be achieved. The factors responsible for reducing the size of the control memory depend on the length of a microinstruction. The length of the microinstruction is greatly influenced by the following three major factors:

- Degree of parallelism which is needed at the microoperation level or in other words “the number of microoperations which can be executed simultaneously”.
- Representation/encoding of control information
- The means of specifying the address of the next microinstruction

The number of microoperations which can be executed simultaneously in a processor may vary from one to a hundred. This degree of parallelism is frequently used for characterising the microprogrammable processors. A highly encoded instruction also tends to be short. We will discuss these more while we discuss microinstruction execution.

Let us focus this section on the calculation of the address of the next microinstruction. In general, the address of the next microinstruction is handled this way:

- The address of the next microinstruction in sequence
- Calculated on the basis of the op-code
- Branch address (conditional or unconditional)

The address is calculated only once from the op-code in one instruction cycle. The machine sequences are not long and branches are common after three or four sequences. Thus, by making branching algorithm better we can make a microinstruction addressing more time efficient. In general, three techniques based on the number of addresses have been utilised for sequencing. These are:

- Two address fields in each microinstruction
- A single address field and
- A variable format microinstruction

In the two address field - microinstructions, either of the two addresses or the address generated with the help of an op-code is selected using a branch logic which is based on the flags and control signal. In such a case, branching to a desired address can be made very easily. However, in this approach a lot of control memory is wasted as at least one of the addresses may not be needed in several microinstructions.

With some modified circuitry and added logic we can reduce the number of addresses to one. Here, a new register called the microprogram counter, as introduced in Figure 101 can be used. In this case, the next microinstruction address can be the address of:

the next sequential address

OR

the address generated using an op-code

OR

the address stored in the address field of the microinstruction

The address selection signal which will be based on branch logic can indicate which of the above mentioned addresses is to be selected. Although this scheme saves some space, yet the space provided for even one address is not used very often. Thus, there remains some inefficiency in the coding scheme of the microinstruction. However, this is a commonly used approach.

Another approach can be used to provide a variable format. In such a case two formats are used. The first format provides the control microinstruction, while the second format provides the branch logic and address. In such a scheme one bit is needed in the microinstruction to indicate whether this is a control microinstruction or a branching microinstruction. In case the microinstruction contains control signals, the next microinstruction address is calculated either by using the op-code of instruction registers or it is the address of the next microinstruction in sequence. In this approach, an extra cycle is needed

for a branch microinstruction which is undesirable. You can get more details on sequencing techniques in the further readings.

### **Generation of Address**

As discussed earlier, there are three ways of generating the address of the next microinstruction to be executed. This address generation once again depends on the number of address fields used in a microinstruction. If we use two address fields then all the addresses are explicit. However, in case of one field and variable format microinstruction, the conditional branch address is calculated on the basis of information like:

- ALU flags
- Address mode indicators
- Sign bits or portion of selected registers
- Status bits of the control unit itself

However, for an unconditional branch, address generation is direct. The opcode for each instruction is to be interpreted once per instruction cycle. One of the techniques which can be used for opcode interpretation is mapping of the opcode into a microinstruction address in the control store. Other techniques in this respect are the addition of two portions of address and residual control. In the addition approach, one part of the address which is fixed is added to a variable part to form a complete address. The residual control involves the use of an address of microinstruction which has been saved previously at a temporary storage location inside the control unit. More details on these approaches can be obtained from the further readings.

### **3.6 Microinstruction Execution**

The microinstruction cycle can consist of two basic cycles; the fetch and the execute. Here, in the fetch cycle the address of the microinstruction is generated and this microinstruction is put in a microinstruction register of execution. We have already dealt with this part in the previous sections. The execution of a microinstruction simply means the generation of control signals. These control signals may drive the CPU (internal control signals) or the system bus. The format of microinstruction and its contents determine the complexity of a logic module which executes a microinstruction.

One of the key features which are incorporated in a microinstruction is the encoding of microinstructions. What is encoding of

microinstructions? Let us recall the Wikes control unit. In Wikes control unit, each bit of information either generates a control signal or forms a bit of the next instruction address. Now, let us assume that a machine needs  $N$  total number of control signals. If we follow the Wikes scheme we require  $B$  bits, one for each control signal in the control unit. Since we are dealing with binary control signals, therefore, an  $N$  bit microinstruction can represent  $2^N$  combinations of control signals.

Do we need all these  $2^N$  combinations?

No, some of these  $2^N$  combinations are not used because:

1. Two sources may be connected by respective control signals to a single destination; however, only one of these sources can be used at a time. Thus, the combinations where both these control signals are active for the same destination are redundant.
2. A register cannot act as the source and the destination at the same time. Thus, such a combination of control signals is redundant.
3. We can provide only one pattern of control signals at a time to the ALU, making some of the combinations redundant.
4. We can provide only one pattern of control signals at a time to the external control bus also.

Therefore, we do not need all these  $2^N$  combinations. Suppose we only need  $2^K$  (which is less than  $2^N$ ) combinations then we need only  $K$  encoded bits instead of  $N$  control signals. The  $K$  bit microinstruction is an extreme encoded microinstruction. Let us examine the characteristics of the extreme encoded and unencoded microinstructions:

### **Unencoded Microinstructions**

- One bit is needed for each control signal; therefore, the numbers of bits required on a microinstruction are high.
- They present a detailed hardware view as the control signal needed can be determined.
- Since each of the control signals can be controlled individually, these microinstructions are difficult to program. However, concurrency can be exploited easily.
- We can provide only one pattern of control signals at a time to the external control bus also

### **Highly Encoded Microinstructions**

- The encoded bits needed in microinstructions are small.
- They provide an aggregated view, that is a higher view of the CPU as only an encoded sequence can be used for microprogramming.
- The encoding helps in the reduction of programming burden; however, the concurrency may not be exploited to the fullest.
- A complex control logic is needed, as decoding is a must. Thus, the execution of microinstructions has propagation delay through gates. Therefore, the execution of microprograms takes longer time than that of unencoded microinstructions.
- The highly encoded microinstructions are aimed at optimising programming effort.

In most of the cases, the design is kept between the two extremes. The LSI 11 (highly encoded) and IBM 3033 (unencoded) control units are close examples of these two approaches. You can find the details of these two in the further readings.

In general, in many of the microinstruction designs, more bits are used than absolutely necessary. In this respect, many terms based on different design characteristics were coined. The terms like horizontal or vertical microinstructions which were introduced earlier, can also give information about the length of the microinstruction. Typically, horizontal microinstructions are 40 to 100 bits, whereas vertical microinstructions are 16 to 40 bits.

### **Encoding Microinstructions**

As mentioned earlier, that microprogrammed control unit designs are neither completely unencoded nor highly encoded. They are slightly coded, in general, to reduce the width of control memory and microprogramming efforts. For encoding, a microinstruction is divided into a set of fields such that:

- A specific control signal can be activated by only one field, thus, making the fields independent of each other.
- Each field represents a pattern of control signals which in turn depicts an action. As the fields are independent, the action depicted by different fields can be performed simultaneously; that is, they are parallel.

- A specific field specifies actions which are mutually exclusive; that is, only one of these actions can occur at a time.

But how do we classify control signals in different fields? In this respect two approaches have been identified. When fields are designated by identifying various functions, we call it functional encoding. For example, a function “transfer of data to accumulator from different sources”, if designated by a single field will be encoded such that each code specifies a different source.

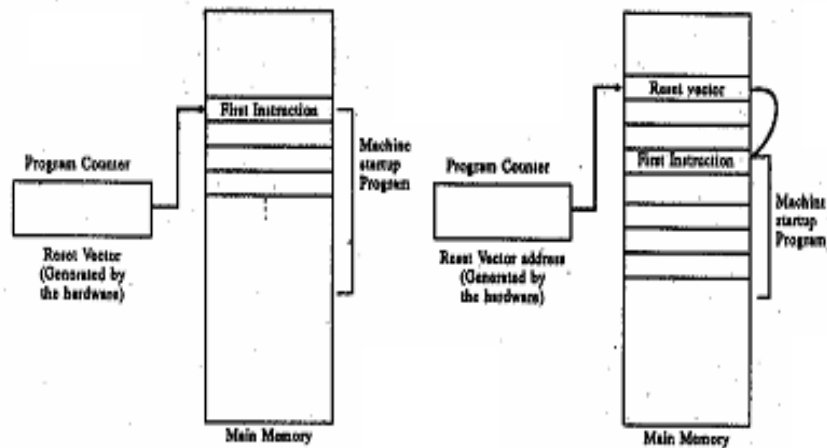
However, if we view the machine as a set of independent sources, then we can devote one field to one resource such as input/output module, memory, ALU, etc. This is known as resource encoding.

Finally, let us look at one of the responsibilities of the control unit which has not been discussed till now, in the next section.

### 3.7 Machine Startup

The control unit is also responsible for initialising various registers during the startup of the machine. The control unit loads a hardware generated address in the program counter (PC) and starts execution of the instruction stored in that location. This hardware generated address may be the *address of the first instruction to be executed* (Refer Figure 102(a)). This address of the first instruction (The first instruction is executed on a machine startup) is known as the reset vector. In some machines the hardware generated address points to the reset vector in the memory. This is the case for Figure 102(b).

In the first case, the control unit starts its normal operation immediately after loading the reset vector in the program counter. However, in the reset vector address machines, the control unit must first fetch the reset vector into the program counter before beginning its normal operation. What is the need of introducing this extra complexity? The advantage here is that the reset vector address machines allow generality of designs to the machine architects; that is, the machine architects will not have to assign the starting address of the startup program to be executed in the memory. They just need to assign the address of the reset vector in the memory, thus, different reset vector values can be chosen on the same machine allowing more flexibility. For example, if we have two different startup routines for two different operating systems then by changing the reset vector value in the memory we can change the loading of the operating system. Both of these operating systems, however, can reside on the main memory at the same time.



(a) Hardware generated reset-vector    (b) Hardware generated address of reset vector

**Figure 102: Loading of a machine startup program**

#### 4.0 CONCLUSION

In rounding up the discussion on CPU organisation, this unit has dealt with the micro-programmed control unit and the concept of microinstructions including microinstruction sequencing and execution.

#### 5.0 SUMMARY

In this unit, we have discussed the micro-programmed control unit. The key to such a unit is a microinstruction. A microinstruction has been defined in the unit. In addition we have also explained the basic structure of the microprogrammed control unit. Microinstruction sequencing and microinstruction execution have also been discussed in the section. Detailed examples of microprogrammed control unit have not been included in this unit. You can refer to further readings for these details.

With this unit, we come to the end of module 2 on CPU organisation. Although we have tried to discuss the aspects related to the CPU in detail, you must refer to the further readings for more details and examples.

#### 6.0 TUTOR- MARKED ASSIGNMENT

State whether True or False



1. A braching microinstruction can have only an unconditional jump. True  False
2. The control store stores microprograms only for opcodes. True  False
3. A true horizontal icroinstruction requires one bit for every control signal. True  False
4. A decoder is needed to find a branch address in the vertical microinstruction. True  False
5. The address computation circuit in Figure 101 is used for finding the next sequential address of a microinstruction True  False
6. One of the responsibilities of sequencing logic (Refer Figure 101) is to cause the reading of the microinstruction addressed by the microprogram counter into the microinstruction buffer. True  False
7. Status bits supplied from ALU to sequencing logic have no role to play with the sequence of a microinstruction True  False
8. What are the possibilities for the next instruction address?
9. Compare two address field-microinstructions with a one address-field microinstruction. Which of them is more commonly used?
10. How many address fields are there in Wikes control unit?
11. Compare and contrast unencoded and highly encoded microinstructions.

## 7.0 REFERENCES/FURTHER READINGS

- Mano, M. Morris, (1993). *Computer System Architecture* (3<sup>rd</sup> ed). Prentice Hall of India.
- Hayes, John P. (1998). *Computer Architecture and Organisation* (2<sup>nd</sup> ed). McGraw-Hill International editions.
- Stallings William. *Computer Organisation and Architecture* (3<sup>rd</sup> ed). Maxwell Macmillan International Editions.
- Baron, Robert J. and Higbie Lee. *Computer Architecture*. Addison-Wesley Publishing Company.
- Tanenbaum, Andrew S. (1993). *Structural Computer Organisation* (3<sup>rd</sup> ed) Prentice Hall of India.