

Block

1

Data Representation and Logic Circuits

UNIT 1

A Computer System

UNIT 2

Data Representation

UNIT 3

Logic Circuits – Introduction

UNIT 4

Logic Circuits – Sequential Circuits

FACULTY OF THE SCHOOL

Prof P. V. Suresh, Director
Dr Shashi Bhushan
Mr M. P. Mishra

Prof. V. V. Subrahmanyam
Mr Akshay Kumar
Dr Sudhansh Sharma

PROGRAMME/COURSE DESIGN COMMITTEE

Shri Sanjeev Thakur
Amity School of Computer Sciences,
Noida Shri Amrit Nath Thulal
Amity School of Engineering and Technology
New Delhi
Dr. Om Vikas(Retd),
Ministry of ICT, Delhi
Shri Vishwakarma
Amity School of Engineering and
Technology New Delhi
Prof (Retd) S. K. Gupta, IIT Delhi
Prof. T.V. Vijay Kumar, SC&SS, JNU,
New Delhi
Prof. Ela Kumar, CSE, IGDTUW, Delhi

Prof. Gayatri Dhingra, GVMITM, Sonipat
Sh. Milind Mahajani
Impressico Business Solutions, Noida, UP
Prof. V. V. Subrahmanyam,
SOCIS, New Delhi
Prof. P. V. Suresh,
SOCIS, IGNOU, New Delhi
Dr. Shashi Bhushan
SOCIS, IGNOU, New Delhi
Shri Akshay Kumar,
SOCIS, IGNOU, New Delhi
Shri M. P. Mishra,
SOCIS, IGNOU, New Delhi
Dr. Sudhansh Sharma
SOCIS, IGNOU, New Delhi

BLOCK PREPARATION TEAM

Dr Zahid Raja (*Content Editor*)
Jawaharlal Nehru University
New Delhi

Prof. (Dr.) Seema Verma (*Course Writer – Unit 1*)
Banasthali Vidyapith
New Delhi

Mr Akshay Kumar (*Course
Writer – Units 2 to 4*)
SOCIS, IGNOU

(*Language Editor*) School of Humanities
IGNOU

Course Coordinator: Mr Akshay Kumar

PRINT PRODUCTION

March, 2021

© Indira Gandhi National Open University, 2021

All rights reserved. No part of this work may be reproduced in any form, by mimeograph or any other means, without permission in writing from the Indira Gandhi National Open University.

Further information on the Indira Gandhi National Open University courses may be obtained from the University's office at Maidan Garhi, New Delhi-110068.

Printed and published on behalf of the Indira Gandhi National Open University, New Delhi by the MPDD, IGNOU, New Delhi

Laser Typesetting : Akashdeep Printers, 20-Ansari Road, Daryaganj, New Delhi-110002

Printed at :

COURSE INTRODUCTION

The objective of this course is to understand the basic components of the computer systems. The course includes ways of data representation in computers, interconnection structures connecting various components together, description of memory system, input-output system, and the Central Processing Unit of a typical computer system. Going ahead, the course presents an insight into the digital logic circuits responsible for realizing the computer, microprocessors as core of the computing, assembly language programming for understanding the interaction with the microprocessor system and interfacing of some of the important peripheral devices. Some portion from advance computer architecture has also been included to enhance the domain knowledge of the readers.

The first block of the course explains Data Representation, Instruction Execution, Interrupts, Buses, Boolean algebra, Design of Logic Circuits, etc. The second block deals with the Memory System, The Memory Hierarchy, Secondary Storage technologies, the concepts of high speed memory, Cache Organization , Input Output interfaces, Input Output techniques, DMA, Input Output processors, External Communication Interfaces, Interrupt Processing, BUS arbitration, etc. The third block deals with the Central Processing Unit. It includes the Instruction Set, the Instruction format, the Instruction Set Architecture, Micro-Operations, the organization of Arithmetic logic unit, Design of simple units of ALU, the Control Unit, The hardwired control, Wilkes control, the Micro-programmed control etc. The fourth block deals with the Assembly Language Programming, Microprocessor, RISC, and various types of multiprocessor technologies.

The List of Facebook Recorded Sessions available on IGNOU Facebook page (They were recorded for MCS012 but are very useful for MCS-202. You are advised to watch them for MCS-202): These Videos are also available through eGyankosh link

<http://egyankosh.ac.in/handle/123456789/60684>

1. The Basic Computer and Fixed Point

Numbers <https://www.facebook.com/Official.../videos/567832200754534/>

2. Floating Point Number representation and Error Detection

Codes <https://www.facebook.com/Official.../videos/179423369719644/>

3. Combinational Circuits <https://www.facebook.com/Official.../videos/2639701279681674/>

4. Sequential Circuits <https://www.facebook.com/Official.../videos/2997470290273216/>

5. Memory Organisation <https://www.facebook.com/Official.../videos/2621991574793648/>

6. Cache Mapping and I/O

Organisation <https://www.facebook.com/Official.../videos/265746954469764/>

7. Assembly Language Programming for 8086

Microprocessor <https://www.facebook.com/Official.../videos/3835800186493405/>

8. Discussion on 8086 Assembly Language

Programs <https://www.facebook.com/Official.../videos/3835800186493405/>

9. Discussion on MCS012 - Block 3: The Central Processing

Unit <https://www.facebook.com/Official.../videos/568476564051914/>

10. MCS012: Computer Organisation and Assembly Language Programming – An Overview <https://www.facebook.com/Official.../videos/357526481886539/>

11. Reduced Instruction Set Computer (RISC)

Architecture <https://www.facebook.com/Official.../videos/265809814701761/>

12. Procedure calls in 8086 Assembly Language

Programming <https://www.facebook.com/Official.../videos/292194305388266/>

BLOCK INTRODUCTION

The first block of the course introduces you to some of the basic concepts relation to a computer. The Block is divided into four units.

Unit 1 explains some of the basic aspects of instruction execution and some of the architectures that has been employed for design of a computer. This unit also introduces you to brief history of computers and interrupt mechanism of a computer system

Unit 2 explains the Data Representation in details. It introduces you to conversions among the basic number systems, such as Binary, Decimal, Octal, Hexadecimal. In addition, the unit discusses about the character representation including ASCII and Unicode. This Unit also introduces you to error detection and correction mechanism in the data units of a computer.

Unit 3 introduces you to the concepts of basic logic circuits used in making of a computer. It introduces the concept of Logic Gates, Boolean algebra, Combinational circuits. It also explains certain examples of combinational circuits such as Adders, Decoders, Multiplexers, ROM etc.

Unit 4 introduces you to concept of Sequential circuits. It explains the functioning of basic latches and introduces you to Flip flops, Excitation tables, Master-Slave flip flops etc. It also presents certain examples of sequential circuits such as Counters, Registers, RAM, etc.

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

Further Readings For The Block

- 1) Mano M Morris, *Computer System Architecture*, 3rd Edition/Latest Edition, Prentice Hall of India Publication, Pearson Education Asia
- 2) Stallings W., *Computer Organization & Architecture: Designing For Performance*, 10th/11th Edition, Pearson Education Asia
- 3) Hennessy/Patterson, *Computer Organization and Design : The Hardware/ Software Interface*; 5th/6th Edition, Morgan Kaufmann.

UNIT 1 Computer System

Structure	Page Nos.
1.0 Introduction	
1.1 Objectives	
1.2 A Brief History	
1.3 Structure of a Computer	
1.3.1 The CPU	
1.3.2 Register Sets	
1.3.3 Datapath	
1.3.4 Control Unit	
1.3.5 Memory Unit and I/O Devices	
1.3.6 What is an Instruction?	
1.4 How are Instructions Executed?	
1.5 Instruction Cycle	
1.6 Various Computer Architectures	
1.6.1 von Neumann Architecture	
1.6.2 Harvard Architecture	
1.6.3 Instruction Set Architecture (ISA)	
1.6.4 RISC	
1.6.5 Multiprocessor and multicore Architectures	
1.6.6 Mobile Architecture	
1.7 Summary	
1.8 Solutions/Answers	

1.0 INTRODUCTION

In the present competitive world, a business can survive if it uses most advanced Information technology to support various businesses processes. In this digital world, computers are an important part of your daily life. You use computer to use health services, banking services, teaching and learning services, online services made available by the Government and many more such services. Computer technology can be used to make all these processes more efficient and user friendly.

This Unit introduces you to some of the basic terminologies used to define computer system of today. In addition, the breakthrough in the history of computer systems has also been included in this Unit. This Unit also introduces you to some of the popular computer architectures, like von Neumann Architecture, which was one of the first computer architecture, and other contemporary architectures of Computer System. Also, a simple novel idea for execution of instructions has also been introduced. This basic process of instruction execution will be explained in more details in the later Units of the course. You can get more information on these principles from the further readings.

1.1 OBJECTIVES

After going through this unit, you will be able to:

- explain the basic structure of computer system
- list and explain the features of the von Neumann architecture of the computer;
- identify some of the important breakthroughs in history of computers;
- identify the process of instruction execution;
- define some of the contemporary computer organization

1.2 A Brief History

This section traces a brief historical background of computer. You should be aware of that the push to construct Computer has not come from one person or group or organisation. There were many attempts to develop an automatic and programmable computing device. In 1944, the University of Pennsylvania developed the first automatic computing device, which was called Electronic Numerical Integrator and Calculator (ENIAC). It was a general-purpose machine fabricated utilizing vacuum tubes. This machine was planned basically to compute the shooting scope of weapons during World War II. It was arranged by manual setting of exchanging and associating links. An improved ENIAC model was called Electronic Discrete Variable Automatic Computer (EDVAC), which was completed in 1952. Meanwhile, the specialists at the Institute for Advanced Study (IAS) in Princeton assembled (1946) an IAS machine, which was multiple times quicker than ENIAC.

A similar project was started in 1946 at Cambridge University. The objective of this project was to design a computer that stores instruction and data in the computer memory. This proposed computer was known as the Electronic Delay Storage Automatic Calculator (EDSAC). In 1949, EDSAC was completed and became the first computer of the world that stored the program instructions and related data in the memory of the computer. The architecture of these machines is discussed in section 1.6. Thereafter, Harvard launched a series of computers named Harvard Mark I, Mark II, Mark III, and Mark IV. The machines have different architecture than that of other contemporary machines. This architecture was called the Harvard Architecture. The Harvard architecture is discussed in section 1.6. The term Harvard architecture, now, used for machines, which have separate instruction and data cache memory. The term cache memory is explained in Block 2.

In 1951, UNIVersal Automatic Computer (UNIVAC I) was developed. This can be called the first commercial computer. In 1952, IBM announced its first computer, the IBM701. Later in 1964, IBM announced a family of computers, called IBM 360 series. The basic concept of family of computers was to keep the similar instruction set. However, the performance and price of the family increased with higher models. This allowed businesses to migrate to higher performance computer while retaining their investments in software. Another important computer series of this time was PDP-8 by Digital Equipment Corporation (DEC).

Advancement of technology lead to development of microprocessor on a single silicon chip. In 1971, Intel produced its first microprocess, called Intel 4004. The first personal computer (PC) was introduced by the Apple computer in 1977. Further, in this year itself world again saw the installation of a mainframe computer. A mainframe computer had many user terminal that used to solve the processing power of a single mainframe computer VAX-11/780 by the DEC. The year 1977 may be considered as an important year for computer industry, as in this year in addition to the developments as given above, one of the important microprocessor 8086, which you will study in Block 4, was introduced. This led to development of micro-computers. Different organisations launched their micro-computers based on Intel 8088/8086 microprocessors or many other microprocessors. In the subsequent years, the computers developed by Compaq, Apple, IBM, Dell, and many others became popular in the academia and

industry. A powerful computer that could perform millions of computation, called super computers were developed. The first such computer, the CDC 6600, was introduced in 1961 by the Control Data Corporation. Cray Research Corporation introduced the most expensive but most efficient super computer Cray-1, in 1976. During the decades of 1980s and 1990s, more powerful super computers were launched. These super computers had larger number of more powerful processors. These super computers were categorised on the basis of the sharing of memory by various processors, viz. shared memory and distributed memory super computers. A super computer differs from a simple multi-processor system, as it can have hundreds or even more processors. Examples of computers at this time included Intel iPSC, nCUBE, Imaging Equipment (CM-2, CM-5), and many others. The perfect design of computer is to install central servers through computer networks. These networks connect cheap desktop machines that have the ability to compensate for unmatched computing power. However, in late 1990s, with the popularity of Local area networks (LANs), which allowed sharing of resources, mainframes and minicomputers were replaced by network computers or personal computers, called desktop computers. These individual desktop computers were then connected to large computers over broadband (WAN) networks. The inclusiveness of the Internet has developed keen attention in the areas like network and grid computing. “Grids are geographically distributed platforms of computation”. Grid computing provides reliable, consistent, persistent, and low-cost access to high-end computational facilities.

Table 1.1 highlights the development of Computers, in terms of type of circuitry used, the size of Random Access Memory (RAM), the speed in terms of Instruction execution per second and Programming languages of those era. The Table also lists some of important computers of those days.

Subject	1st generation	2nd generation	3rd generation	4th generation	5th generation
Period	1940-1956	1956-1963	1964-1971	1971-present	present & beyond
Circuitry	Vacuum tube	Transistor	Integrated chips (IC)	Microprocessor (VLSI)	ULSI (Ultra Large Scale Integration) technology
Memory Capacity	20 KB	128KB	1MB	Magnetic core memory, LSI and VLSI. High Capacity	ULSI
Processing Speed	300 IPS instructions Per sec.	300 IPS	1MIPS (1 million inst. Per sec.)	Faster than 3rd generation	Very fast
Programming Language	Machine, Language	Assembly language & early high-level languages(FORTRAN, COBOL, ALGOL)	C,C++	Higher level languages,C,C++,Java	All the Higher level languages,,Neural networks,
Example of computers	UNIVAC, EDVAC	IBM 1401, IBM 7094, CDC 3600,D UNIVAC 1108	IBM 360 series, 1900 series	Pentium series,Multimedia,	Artificial Intelligence, Robotics

TABLE 1.1 Decades of Computing

[“Ref: <https://www.discovertips.in/2017/06/computer-study-notes-history-and.html>”]

Semi-Conductor Chips and Computer

A computer system consists of number of components, which are fabricated using semiconductor materials. The basic unit of these semi-conductor materials is an electronic transistors, which are used to integrate large computer circuitry on a single semi-conductor chip. Over the last five decades computer has shown an exponential rate of improvement due to growth and advancement in circuit integration technology on a semi-conductor chip; from a handful of transistors to millions of transistors on a single integrated chip. This resulted in multiple

fold increase in the size of computer memory and very large increase in processing capacities of computer processor. The integration technology has shown a rapid growth. Initially, only a few transistors were fabricated on a computer chip. This was called small-scale integration (SSI). Subsequently, with the advancements of technology, computer chips were made using the medium-scale integration (MSI), then using the large-scale integration (LSI), then using the very large-scale integration (VLSI), and currently to ultra large-scale integration (ULSI). Figure 1.2 includes various technologies and number of transistors per chip, also called as chip complexity. The integration growth can be more understandable in terms of “feature size”. Feature size is that dimension of transistor that is required to be minimized without changing the functionality of the device. So, a decrease in feature size will eventually increase the number of transistors per chip which in turn, will result in development of new advanced chip having advanced functionality. This has resulted in growth of semiconductor memories (RAM memories) so that now, designers can trade off speed with memory size.

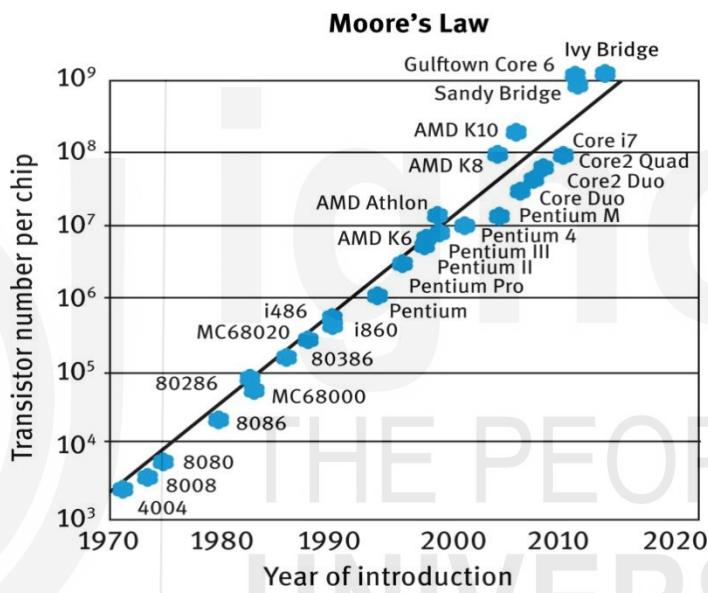


Figure 1.2: Numbers of Transistors per Chip
[“Ref: https://www.ncbi.nlm.nih.gov/books/NBK321721/figure/oin_tutorial.F3/”]

1.3 STRUCTURE OF A COMPUTER

As discussed in the previous section, computer is fabricated using semi-conductor chips. This technology primarily have two stable states of representation, such as presence or absence of output. These two states are used to denote bits 0 and bit 1 respectively. Thus, the computer is a digital device, which performs all the operations using binary digits (bits). Therefore, computer instructions, character set, data in computation etc everything is to be represented and processed using binary codes or binary number system. The minimal processing requirements of a computers system are:

- It should allow input of instructions or commands and data on which these commands are to be performed.
- There can be large number of instructions that may be required for data manipulation. Some of these instructions may involve decision making and/or repetitions, therefore, it may be a good idea to store instruction and data. Thus, a computer should have memory.

- The computer should be able to process data as per the command/instruction. Thus, it will require components, which may process data, store temporary results and transfer data among different units.
- It should be able to store the results of the processed data using some output unit.

Therefore, a computer system should have a processor to perform computations as per the instructions, a memory for data storage/instructions, Input/Output units to input the data and instruction and output the results and a data path to transfer data. In order to define a simple structure of a computer system, the role and functions of its basic components should be studied. This section defines the basic components of the computer, like CPU, the Memory, the Input/output devices, data paths etc.

1.3.1 The CPU

The CPU is responsible for executing a sequence of instructions, which are part of a computer program. A computer program along with the data is stored in the main memory of a computer system. The CPU consists of following components: (1) registers, (2) an arithmetic logic unit (ALU), and (3) a control unit (CU). The role of each component of CPU is well defined. The registers are used to store - (1) commands or instruction which is presently being executed by the Arithmetic logic unit, (2) they can also store addresses such as address of operands /data, “address of the next instruction to be executed”, (3) the data or operands itself. Arithmetic logic unit is designed to perform binary computations.

The control unit (CU) controls the execution of instructions by the computer. CU controls the fetching, interpreting and execution of the commands or instructions on a computer, which primarily results in processing of the data stored in registers or the main memory. Figure 1.3 shows the structure of the CPU and its interaction with the memory system and input / output devices. The CPU “reads commands from the memory, reads and writes data from and to the memory, and transfers the data to the input / output devices”. The most common and simple commands/instruction processing can be summarized as follows:

1. Get the next instruction or command to be executed from the main memory to the CPU registers
 - a. In general, the address of the next instruction is stored in the program counter register (PC).
 - b. The CU causes the instruction fetch operation using the PC.
 - c. The fetched instruction is stored in the CPU, in general, in an instruction Register (IR)
2. The instruction in the IR is interpreted by the control unit.
3. In addition to step 2, the operands are brought from the main memory to CPU registers.
4. The operation as interpreted at point 2, is performed on the data obtained in step 3.
5. Results of the operation in step 4 are stored in the CPU registers. In case the instruction specifies that the result of the operation is to be stored in the memory, then the result data in CPU registers are transferred to the specified memory locations.

The execution cycle is repeated as long as there are more commands to be executed. Sometimes the execution of a program is required to be terminated abnormally due to occurrence of certain error or other conditions, like division by zero. Thus, there may be need of a mechanism that can interrupt the execution

of a sequence of instruction. This mechanism is known as an interrupt mechanism, which uses an interrupt signal. On occurrence of an interrupt signal, CPU suspends the execution of next instruction to be executed, though it completes the execution of the current instruction. More specifically, when a request for interruption occurs, a move to an interrupt management process occurs. Interrupt management systems are the set of programs that are used to address the cause of the interruption and restores the system to the last instruction, where interruption was acknowledged.

Central Processing Unit (CPU)

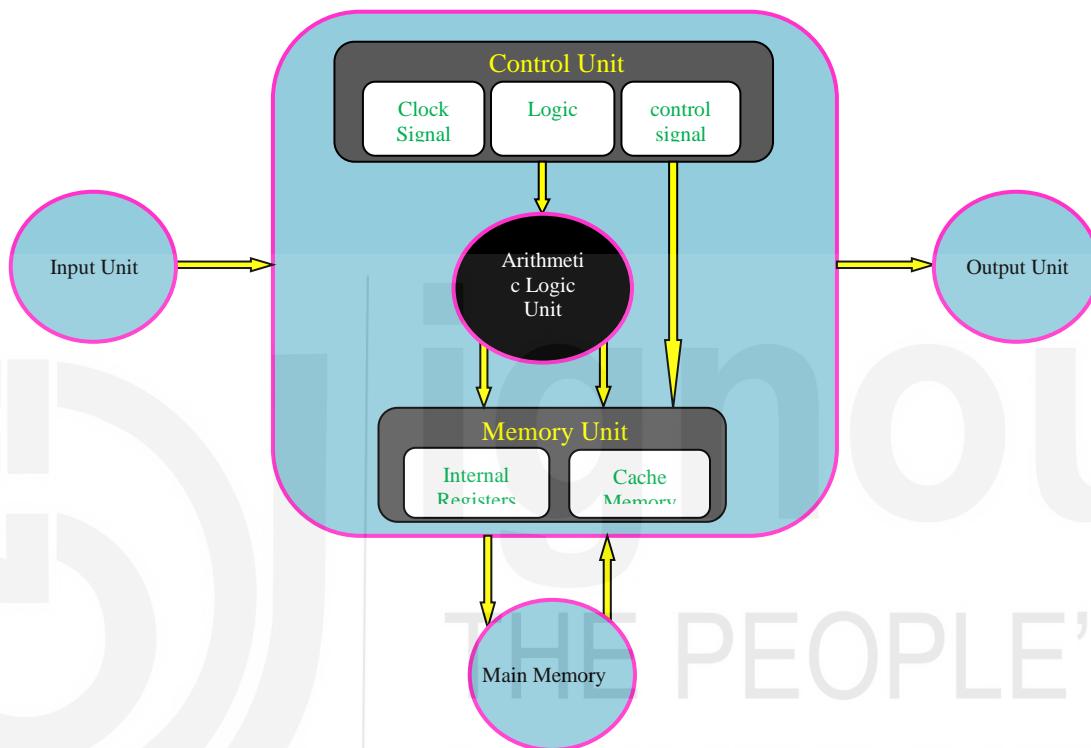


Figure 1.3: Central processing Unit and its components

1.3.2. Register Set

Registers are extremely fast, but temporary storage locations, within the CPU. Registers store the intermediate results, therefore, are used to enhance the CPU performance for performing arithmetic; logical or other operations. The names, number, type, and length of registers are different for different computers. In general, general purpose registers can be used for storing the data for an operation or address of an operand or any other purpose of various tasks in a program. Special purpose registers are limited to designated functionalities only. There are cases when some registers are used only for performing operations on data and cannot be used for operand address computations. These data registers length should be long enough to hold values for most types of data. Some machines allow two integrated registers to hold longer length data. Address registers are used for computing the address of an operand in the memory. This task may be assigned to specific types of register or the general address purpose register may be used for this purpose too. Address registers should be long enough to handle a very large address, which in general depends on the size of the memory. The number of registers in a computer affects the size of the instructions. Why? you will find answer to this question in Block 3. Some important registers are given below:

Register for Fetching and storing Instructions

In a computer system the instructions of a program are stored in the main memory. Two main registers, which are involved for transferring instructions from main memory to CPU are: Program counter (PC) and instruction/command or instruction register (IR). PC contains the address of the memory location from which the next command may be executed. Instructions are fetched to IR, so that it can be interpreted and executed.

Condition Registers and Status registers

In some computers, status registers or flag registers are used to store status information of various operations. Some computers have a special program status register (PSW). PSW contains the present status of the processor flags. A detailed discussion on flags is given in Block 3 and Block 4.

1.3.3. Datapath

As discussed earlier, a computer performs instruction execution using different components. But, how does the data and instructions get communicated from one component to other. This is achieved by datapaths. There can be two different types of datapaths:

(i) The datapaths which are internal to CPU: Such datapaths transfer two different categories of data. Data category, which includes data contained in different registers of ALU. The control data, which essentially pulls control signals from the datapaths for the use of control unit. In addition, the data is moved between two registers or between ALU and a register. This internal data migration is done by local buses, which may carry control information, instructions and addresses.

(ii) Externally, data may be transferred to and from registers to memory and I / O devices, usually using a special set of circuit called the **system bus**.

Internal data transfer between registers and between ALUs and registers can be done through various organizations including one, or more buses. Dedicated datapaths can also be used for data transfer devices between the CPU components on a regular basis. For example, Program counter register (PC) content is transferred to Memory Address Register (MAR) to fetch new commands at the commencement of each command cycle. Therefore, dedicated datapaths from PC to MAR can help speed up this part of command execution.

1.3.4. Control Unit

The control unit is primary unit which commands operations of the system by giving control signals to various units of computer. The control unit is also responsible to regulate internal and external flow of data from CPU. The data transfer from CPU to/from memory and CPU to/from I/O is also controlled by these signals. A continuous pulse sequence is generated by a system clock over a set period of time. This sequence of steps, identified as t_0, t_1, t_2, \dots , ($t_0 \leq t_1 \leq t_2, \dots$), is used to perform a specific command by enabling control signals in a specific order. The details on various types of control units and their operation are discussed in Block 3 of this course.

1.3.5 Memory Unit and I/O Devices

An interesting part of computer is the memory of a computer, which stores the data as well as instructions. Computer stores binary digits 0 and 1 called bits. However, bit is a very elementary, therefore, a meaningful combination of bits is generally, required to be stored in memory. A group of 8 bits is traditionally called a “Byte”. However present day data may include 16 bits (2 bytes), 32 bits (4 bytes), 64 bits (8 bytes) and so on.

Interestingly the size of the memory is represented as a byte in many situations, which was equal to one character in earlier data representation (Please refer to Unit 2 for data representation). Therefore, higher units are needed to measure the size of the memory. As computer is a binary device, an interesting combination as given in the following table is used to measure the memory capacity.

Unit	Equivalent to
1 Kilobyte (KB)	2^{10} Byte = 1024 bytes \approx 1000 bytes
1 Megabyte (MB)	2^{10} KB = 20^{20} byte
1 Gigabyte (GB)	2^{10} MB = 20^{30} byte
1 Terabyte (TB)	2^{10} GB = 20^{40} byte
1 Petabyte (PB)	2^{10} TB = 20^{50} byte
1 Exabyte (EB)	2^{10} PB = 20^{60} byte
1 Zettabyte (ZB)	2^{10} EB = 20^{70} byte
1 Yottabyte (YB)	2^{10} ZB = 20^{70} byte
and beyond	

As stated earlier, the purpose of memory in computer is to store the instructions of a program and the related data. These instructions and related data to these instructions are fetched by the CPU, which then executes these instructions. This memory is also called the Random Access Memory (RAM) as any location of the memory can be addressed randomly. Details on memory system are given in Block 2.

I/O devices are used to input data and programs, e.g. a keyboard can be used to input a program. Present day computer use pointing devices and screens to select options from Graphical user Interfaces (GUI) like Microsoft Windows. The output devices like printer, monitor etc. display the output either in printed (also called hard copy) form or are displayed on the details on I/O devices are also given in Block 2.

1.3.6 What is an Instruction?

In the entire discussion on CPU, one term was used consistently - an instruction. What is an instruction in the context of a computer system?

In general, you write programs in a high level computer language like C, Python, JAVA, C++ and so on. Most of these programming languages require you to compile the program into an object program, which is linked with library programs and loaded in the memory of the computer. A program also includes some basic data and I/O from keyboard or files. These loaded programs contain binary instructions, which operate on binary data.

Some of the common high level statements include arithmetic instructions like addition, subtraction; decision like if-then-else; repetitive loops which also involves decisions like while, for etc.; and procedure/function calls. In all these statements few common characteristics stand out, which are made part of binary instructions as given below:

- (1) Each binary instruction may define one operation using a binary operation code also called opcode.

(2) Each instruction may have one or more operands which may be data itself or can be used to compute the address of operand. A computer instructions, depending on machines, may consist of one to three operands.

(3) The result of operation of machine instruction can be stored in some machine register.

(4) Some instructions, like branch, function call etc., result in transfer of execution to a new instruction, which may be at a different address in the program. This, can be achieved by changing the value in program counter register, which stores the next instruction to be executed.

Binary instruction design of a computer system is a very complex task. In fact the machine instructions of different computers are different, that is why you require a separate compiler for a separate type of computing machine. A detailed discussion on computer instructions is given in Block 3.

Check Your Progress 1

1) State True or False

T/F

- a) Byte consists of 8 bits, and it may represent a character.
- b) A character representation requires at least one byte.
- c) One bit is an independent unit and can be accessed independently.
- d) A machine can have two paths called internal and external.

2) Explain the concept of an Instruction.

.....
.....
.....

3) Why does a computer need CU, ALU, memory and I/O devices?

.....
.....
.....

1.4 HOW ARE INSTRUCTION EXECUTED?

After getting a fair idea about basic structure of computer system, let us now understand about execution of a program. To learn about how a computer executes a program, let us take an example from high level language.

Consider the following program segment of a high level programming language:

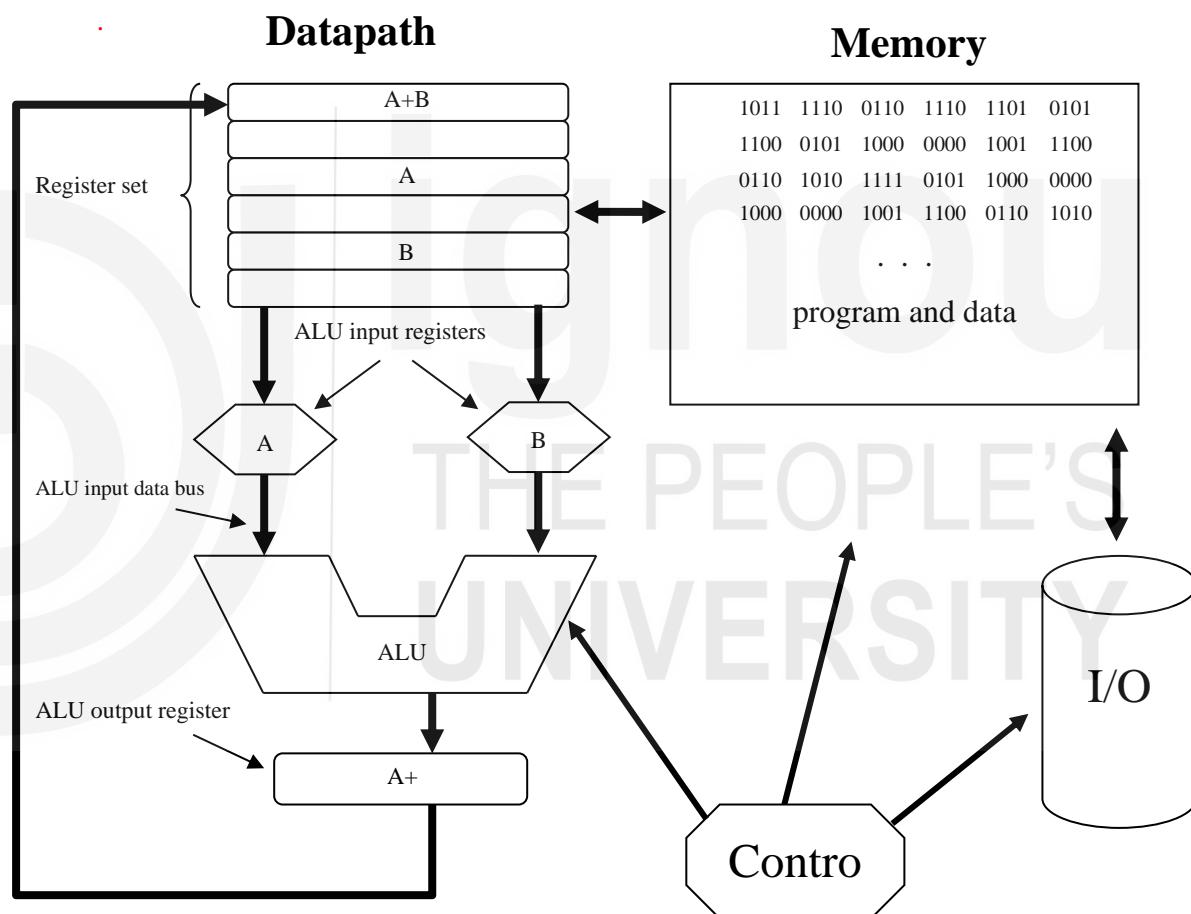
```
int x=10, y=5, z;  
z=x+y;
```

Please note that the instruction `int x=10, y=5, z;` will allocate three memory locations of type integer (how big will be one integer). The `x`, `y` and `z`, in the context of programming languages, are called variables. This instruction will also assign integer value 10 in first location, identified by variable name `x` and integer value 5 in second location identified by variable name `y`.

The instruction will also allocate a third memory location named z . Thus, first instruction, technically will be translated to create three integer location named x , y and z . Please note that these locations, when loaded in the memory will be identified as three separate addresses. The second instruction $z=x+y;$ will be executed by the CPU to produce the desired results. But, how does these set of instructions be executed by computer? As a first step, a compiler program will be executed by computer and all the High level programming statements will be translated to a machine language program consisting of data in the form of variable locations and values, and instructions as binary operation codes and operand addresses.

Please note in the C program segment, the declaration results in creation of variable locations, with data values.

It is the instruction $z=x+y;$ which gets converted to one or more machine instructions, which will have binary codes indicating addition operation, opened address on which this addition is to be performed, and where the result will be stored. Assuming a typical machine is shown in Figure 1.4



Fetch – Decode – Execute Cycle

Figure 1.4: “Instruction and data format of an assumed machine[<https://www.cise.ufl.edu/~mssz/CompOrg/CDA-lang.html>]”

Please notice the role of ALU registers. They get values of locations x and y first to be added by the ALU and the answer of this addition is stored in a register, which is sent back to the memory location z .

In general, a machine consists of several registers as shown in figure 1.5. The details on these registers will be discussed in Block 3 of this course.

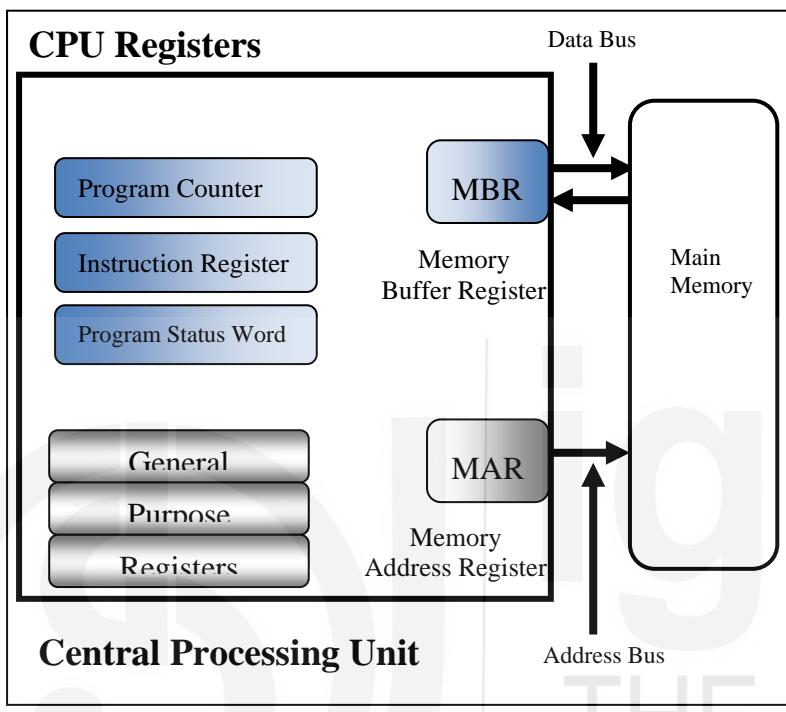


Figure 1.5: “Central Processing Unit with registers

[<http://digitaithinkerhelp.com/what-is-cpu-processor-register-in-computer-its-types-with-functions/>]”

1.5 INSTRUCTION CYCLE

A program is having a set of instruction which are stored in computer's memory. In general, the instructions of a program are executed sequentially by the processor. A computer system executes an instruction in the following four phases:

1. Fetching.
2. Decoding
3. Read the operands from the memory.
4. Execution.

These steps are explained in details in Block 3. In this unit they are just being introduced.

Fetching the instruction: An instruction is placed in the memory location or locations in RAM. Instruction fetch will bring this instruction into the Instruction register. This step requires the information about where the instruction is in the memory. In general, PC register contains this information.

Decode the Instruction: Decoding the instruction requires to interpret the operation code of the instruction, this will be followed by finding locations of the operands in the memory.

Read the operands from memory: Once the addresses of the operands are known, they are brought into the CPU registers, such that the required operation may be performed.

Execute the Instruction: Finally the instructions are executed and results are stored in the local temporary locations.

The process is shown in figure 1.6

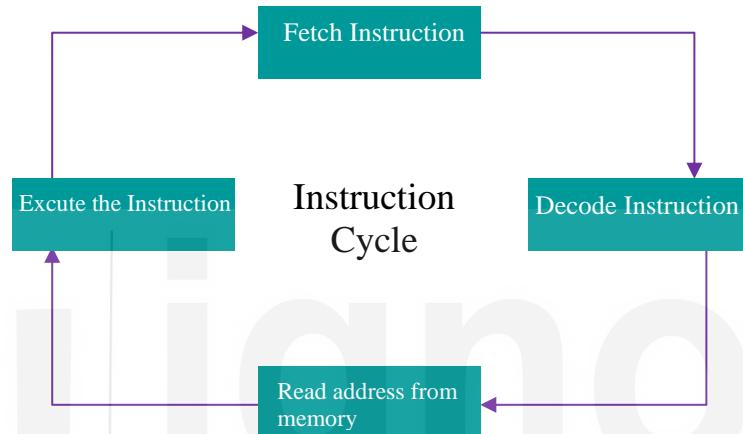


Figure 1.6: Instruction Cycle [<https://www.javatpoint.com/instruction-cycle>].

Can a program be interrupted while it is getting executed? A program may be required to be stopped, during the execution due to occurrence of errors or occurrence of completion of certain other important – I/O events. This process has been explained in the next subsection:

Interrupts

The meaning of word interrupt is to stop the ongoing activity. In computer, an interrupt, stops execution of next instruction in the instruction cycle, once the execution of ongoing instruction is completed. Why? You will get an answer with this question in Block 3.

Why does an interrupt occurs in a computer?

Interrupt occurs due to:	Example
Program instruction execution itself. Causes an interrupt.	<ul style="list-style-type: none"> □ Division by Zero occurs □ The result exceeds the limit of the number allowed. □ Security violations.
Clock initiated.	<ul style="list-style-type: none"> ● Expiry of time limit of a program.
I/O devices	<ul style="list-style-type: none"> □ Input/Output starting or completion □ Error due to an I/O device
Hardware generated.	<ul style="list-style-type: none"> □ Memory errors

Figure 1.7: Example of Interrupts in a Computer.

Interrupt mechanism is a very useful mechanism for increasing the efficiency of program execution. The instruction cycle many continue, till the time an interrupt occurs. As a result of an interrupt, the CPU knows that some event has occurred and then CPU stops the execution of the current program and goes on to process that event. CPU then uses the following steps to process the interrupt:

- The CPU identifies the source of the interrupt.
- CPU executes an Interrupt Servicing Routine (ISR), which services the event that has occurred.
- Meanwhile, the program which was being executed is moved to a hold state.
- Once the CPU completes the ISR, i.e. executes it till its completion, it resumes the execution of program it has put on hold.

Interrupts and Instruction Cycle

The interrupt process is summarised below:

- CPU is executing a program say “X”. and is in the decode stage of i^{th} instruction of the program X.
- Assume an interrupt due to an event occurred at this time.
- CPU waits till the i^{th} instruction execution is complete, and then stores the register values & PC, which has address of $(i+1)^{\text{th}}$ instructions into memory or special storage area.
- CPU identifies the interrupt and executes the interrupt servicing program till it is complete.
- CPU then return to execution of the $(i+1)^{\text{th}}$ instruction by restoring the register and PC.

Thus, after interrupt processing, the execution of the interrupt program is resumed. Figure 1.8 shows Interrupt cycle.

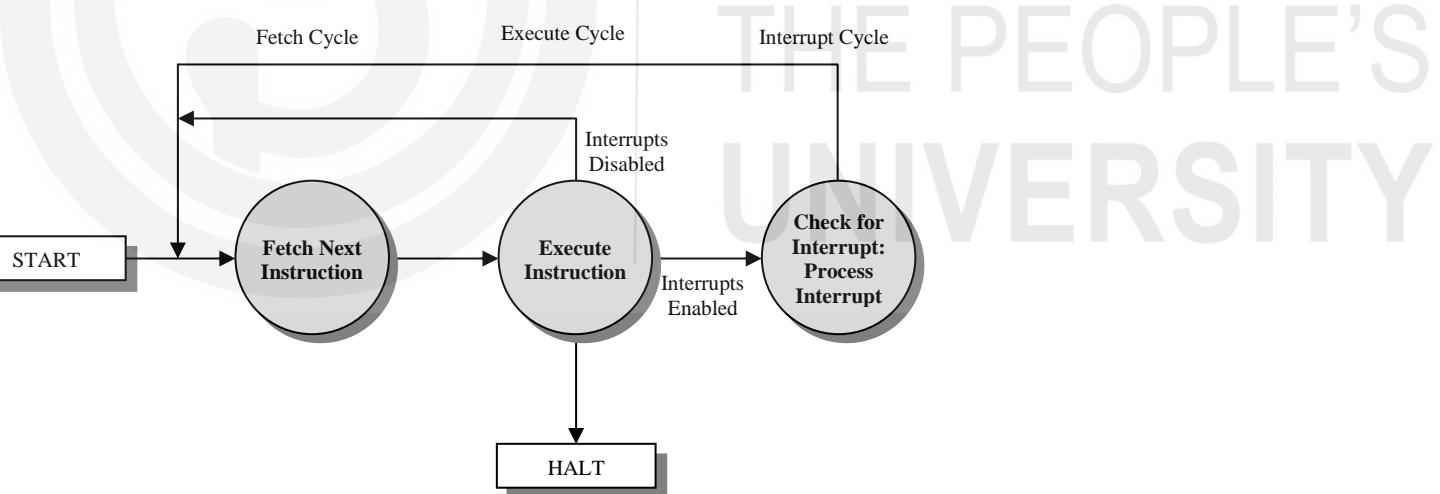


Figure 1.8: “Instruction Cycle with Interrupt Cycle
[\[https://www.ece.ucdavis.edu/~vojin/CLASSES/EEC70/W2004/presentations/Ch_03.pdf\]](https://www.ece.ucdavis.edu/~vojin/CLASSES/EEC70/W2004/presentations/Ch_03.pdf)

Please note an important point in the figure 1.8, which is - interrupt enable and interrupt disable conditions. If CPU is executing very important instruction, the response to an interrupt can be disabled. In this situation interrupts will not be processed or acknowledged by the CPU till they are enabled again.

Check Your Progress 2

1) State True or False

- i) Assume a computer has one-byte long instructions, its PC contains a decimal value 205 and only one byte is fetched from the memory at a time. For this machine, after fetching an instruction, the value of PC will become 206.

T/F

ii) PC register is needed to fetch the data from memory.

iii) A clock signal cannot cause as interrupt.

iv) The interrupt are answered by the hardware.

v) The processor processes one interrupt, even if, multiple interrupts may occur at a time.

2) Explain the term interrupt and its cause.

.....

.....

.....

3) Explain the interrupt processing in a computer.

.....

.....

.....

1.6 VARIOUS COMPUTER ARCHITECTURE

Architecture of a computer contains procedures or processes to explain the user about the functionality of a computer system. This is analogous to design of a building, i.e. the construction of buildings is tailored to the needs of the user by taking into account the cost threshold. The original design is designed on paper. Likewise the computer, after it was constructed using the logic of transistors and integrated circuits, the construction was tested and built in a hardware way. Computers can be evaluated based on their performance, efficiency, reliability, and cost of computer software, which works with software and computer technology standards. In this section, you will learn about various computer architectures.

1.6.1 von-Neumann Architecture

John von – Neumann has invented this architecture. Several computers are based on the construction of this architecture and other improvements on this architecture. The basic building blocks of this architecture are-

1. CPU
2. Memory Unit
3. I/O Devices
4. Connection Structure

Each memory unit having multiple locations will have different addresses. Same memory is having instructions and data, which has multiple locations with each location having unique address. A computer program consists of instructions and data. CPU process the data stored in the memory or obtained term input devices as per the instruction the program. The results are stored in the memory or output devices. The data processing operation in CPU use registers.

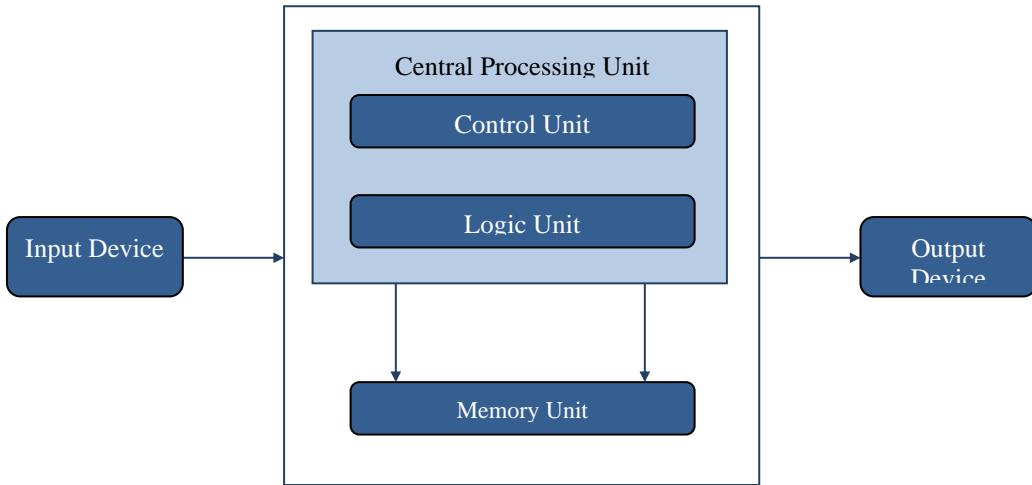


Figure 1.9: Von Neumann Machine Ref: www.educba.com

Buses (address bus / data bus / control bus) are used for communicating the address, data and control signals. The input device fetches data or commands and the Central processing unit (CPU) executes it. When the task is completed, the result is stored on output device.

As discussed in the previous section, a computer system executes instruction/commands using processor and memory while the registers provide the processor's temporary storage requirements, whereas memory works with medium-term and long-term data storage requirements. Any data processing system consists of a sequence of instructions that enable the processor to perform the functionality you want. These instructions operate on data, which may be input from input devices. The processed data is the output of the data processing systems. We may store instruction and data together or separately. In the Princeton architecture, data and instruction share the same memory space. In Harvard architecture, system or instruction memory space differs from data memory space. Princeton configuration can lead to easy hardware connection to memory, as only one connection is required. Harvard configuration requires, dual connections, can simultaneously read instruction and operand data, so it can lead to improved performance. Most machines have a Princeton design. Intel 8051 is a well-known Harvard architecture. Figure 1.10 is a diagrammatic representation of Harvard architecture, whereas figure 1.11 represents the Princeton architecture of a computer.

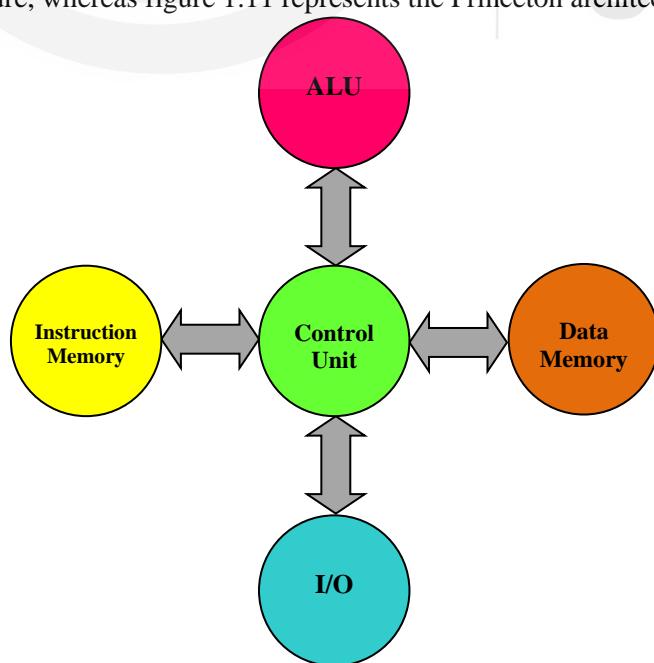


Figure 1.10: Harvard Architecture

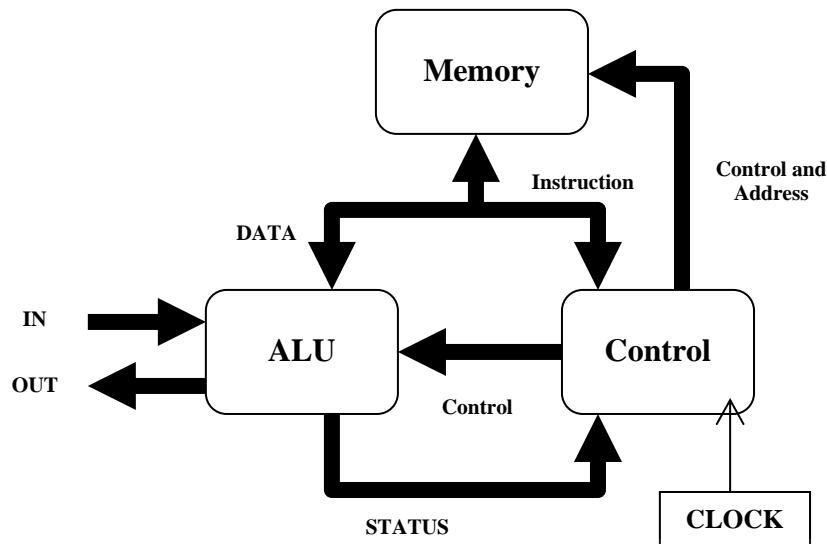


Figure 1.11: Princeton (von Neumann) Architecture

Stored System Computers –

Such computers can be programmed to do various tasks and also, applications are kept on them that is why they are named as Stored System Computers. This concept of a stored system was presented by John von Neumann. In such computer systems, a single memory is used to store programs and data and is treated in the same way. A computer built in this way will be easy to imitate.

The von Neumann architecture, also called as Princeton model, is a computer architecture designed by the 1945 definition of John von Neumann and others in the First Draft of a Report on the EDVAC. The draft defines the design of a digital computer. As per this document a general purpose computer should contain:

- ❑ a Processing unit comprising of arithmetic unit, logic unit and processor registers
- ❑ A control unit comprising of command register and a program counter register
- ❑ Data storage in data memory
- ❑ Limited external storage
- ❑ Input and output methods

The name "von Neumann architecture" has been attributed to any computer where the stored program and data are not transmitted simultaneously because they share the same medium of data transfer, called system bus. This is called a von Neumann bottleneck, as it restricts simultaneous access of data and instructions, thus, may result in reduction in the performance of the system. However, the single path simplifies the design of the von Neumann machine. In comparison, a Harvard architecture machine uses one dedicated bus each for address memory and data memory respectively, therefore, is more complex.

The basic characteristics of Von Neumann architecture are summarized below.

(1) A computer has the following operational units:

- The control unit (CU), which interprets commands to be executed, and generates control signal which instructs other units about how to perform the operation.

- Registers and other circuitry.
 - Input/output system, which are used for input of instructions and output of results.
 - A memory, which stores instructions as well as data (both).
 - The inter connection structures, which allow communication between various components.
- (2) von Neumann machine works on a concept of stored program, which requires that data and commands both must be loaded into the memory of the main computer prior to execution & called as Random Access Memory (RAM).
- (3) The instruction/commands in a Von Neumann machine are executed in a sequence, therefore, to change this order of execution, a special instruction may be used.
- (4) A Von Neumann machine has a single path from control unit to main memory so only one of two-data or instruction can be transferred between the two at a time.

1.6.2 Harvard Architecture

When data and instructions are kept in separate memory then it is Harvard architecture. Data is fetched from one memory location and instruction is retrieved from a separate location. Pipelining can be done in this architecture. It is complicated to design this architecture. The CPU can fetch, decode and execute instructions and data. The construction of this architecture has separate access for codes and data address.

The modified Harvard architecture is similar to the Harvard architecture and has a standard address space for a separate data and instruction cache. It has digital signal processors that can handle audio and video data efficiently. It also has microcontrollers, the processing circuits that process small number of applications and has small data memory and speed up processing by performing the commands and data access simultaneously.

Figure 1.12 shows different connection paths for modified Harvard architecture. All these four units are contained in CPU. It can perform simultaneous input / output operations and has a separate mathematical and logic component.

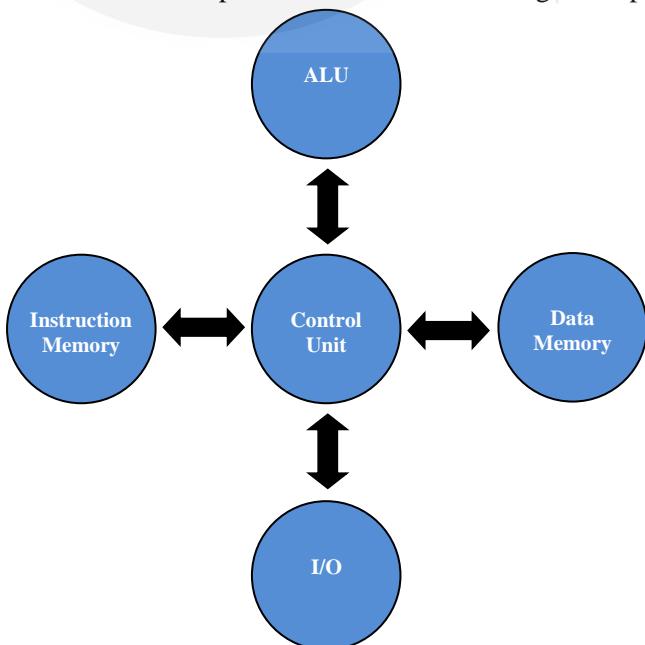


Figure 1.12: The modified Harvard architecture.

1.6.3 Instruction Set Architecture (ISA)

Instruction set architecture (ISA) defines a set of instructions that are to be supported by a processor. From system point of view, ISA does not care about certain computer implementation details. It is only about setting up or collecting the basic functions that a computer should support. Some of the common examples of ISA are Intel upgraded x86, ARM, MIPS, and AMD.

An ISA includes different kinds of instructions, sizes of differ instruction formats. These concepts will be explained in more details in the Block 3. The following example of MIPS ISA very briefly defines the description that should be supported by an ISA. You may refer to further readings for more details in this ISA.

1. *ISA defines the types of commands that the processor will support.*

Depending on class of work they are doing MIPS Instructions are divided into three types:

- Arithmetic / Logic Instructions
- Data transfer instructions
- Branch and Jump Instructions

2. *Maximum length of each type of instruction has been defined in ISA.*

3. *Instruction format for each type of instruction has been defined in ISA.*

Various formats in MIPS ISA:

- R-Instruction format
- I-Instruction Format
- J-Instruction Format

As every format is having separate command coding schemes in terms of operation code, number of operands etc., so they are required to be understood differently by the processor.

The following diagram shows the hierarchy of abstraction of Architecture:

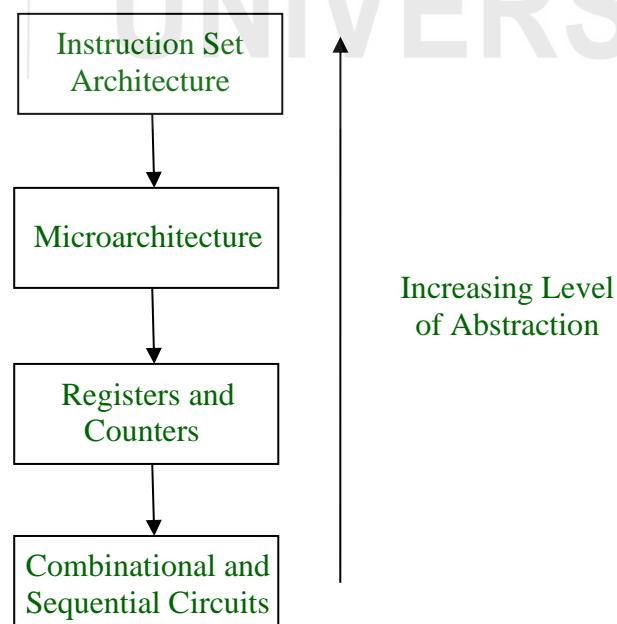


Figure 1.13: The Abstraction Hierarchy

As Micro-architectural standard is placed just below the ISA standard and is therefore deals with the implementation of computer-based core functions as suggested by ISA.

What is the need to differentiate between Micro-architecture and ISA? It is required to establish and maintain system compliance across all ISA-based hardware applications. Adapting different machines to the same set of basic commands (ISA) allows the same system to run smoothly on multiple different machines, thus, making it easier for editors to write and store code for many different machines simultaneously and efficiently.

This abstraction hierarchy supports flexibility, which is why first the ISA is developed and then various micro architectures are created that are compatible with this machine-operated ISA. The micro-architecture is implemented using various logic circuits.

1.6.4 RISC

RISC formulation is being used by ARM core. RISC is a strong design and it delivers simple commands in a single cycle with high clock speeds. RISC targets to reduce the hardware complexity of instructions because hardware has less flexibility in comparison to software. As a result, the structure of the RISC places expectations on the compiler. Conversely, complex instructions (CISC) rely heavily on hardware for operational performance, and as a result CISC commands are complex. Figure 1.14 shows this major difference.

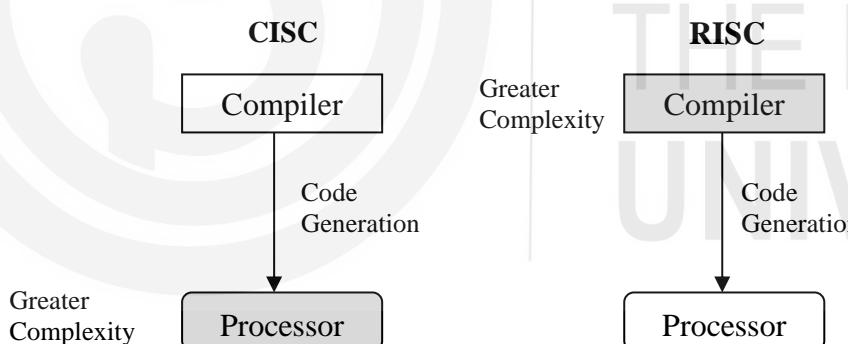


Figure 1.14: RISC vs. CISC

CISC emphasizes the complexity of hardware. The RISC emphasizes the complexity of the compiler.

The RISC philosophy is based on four main building codes:

1. *Instructions* - Number of instructions has been reduced in RISC. An instruction of RISC is executed in several segments. Instruction execution is overlapped in these segments (called pipeline segment). Each segment takes about one clock cycle time. For example, a processor performs complex tasks by combining several simple commands. Each command is of a limited length to allow the “pipeline to pick up future commands before deciding on current commands”. CISC instructions are usually of varied length.

2. *Pipes* - The processing of the instructions is divided into smaller units that can be made similar to pipes. Appropriately the pipe continues one step in each cycle of execution.
3. *Registers* – Large registers for general purposes have been included in RISC which may contain data or address. CISC processors have been provided with specific registers for specific tasks.
4. *Load-store creation* - The processor operates on data managed in registers. Memory access is expensive, so separating memory access to data processing provides an opportunity. With CISC design the data processing functionality can work in direct memory.

The RISC is explained in more details in Block 3.

1.6.5 Multiprocessor and Multicore Architecture

One of the ways to improve computer system efficiency is to have processors with faster clock speeds. But, owing to the thermal wall problem, when the clock speeds started hitting the thermal barrier, focus was on to extract maximum work out of the available system as an alternative of increasing processor speed.\

One of the ways to introduce parallelism in computers is to have more than one processor available in the single computer system referred to as multiprocessor system. It is very likely that at times the job (application) to be executed on a processor can be divided into various tasks with two or more tasks being capable of running independent of each other. Multiprocessor system corresponds to the architecture in which rather than having only one processor we have more than one processor available on the chip. Thus, if the job demanding execution comes with independent tasks, a processor can be dedicated to each task in order to exploit the parallelism in the job. In a way, multiprocessor system can be looked as a solution to offer hardware parallelism to match the available software parallelism in the job. This architecture corresponds to instruction level parallelism allowing independent instructions (or sub tasks) being assigned to different processors in the multiprocessor system allowing their parallel execution. If all the processors are same, the system is referred to as a Symmetric Multi-Processor (SMP). The multiprocessor architecture shares the computing environment viz. memory, OS, system clock etc.

In a conventional computer system, you had only one CPU available, which is responsible for the job execution focusing on only one task at a time. However, now a days, you hear about terms like a computer with quad core processor or an octa core processor. This architecture is referred to as the multicore architecture corresponding to both instruction level and thread level parallelism in a uniprocessor system. In this case, a single processor CPU is fabricated to have more than one CPU cores inside it. Each core acts an independent processing unit (CPU) and can execute independent threads, if permitted by the program. Thus, a quad core processor comprises for four cores whereas an octa core processor has eight cores capable of working independently on the same processor.

The difference between multiprocessor architecture and multicore architecture lies in the fact that multiprocessor architecture has more than one processor available whereas multicore processors have a single processor with multiple cores used as independent CPUs. Thus, multicore architecture is multiprocessing in a single packaged unit. Both multiprocessor and multicore architectures correspond to the MIMD category of Flynn's classification with multiprocessor architecture being more pure form of parallel processing. Practically, for improved performance, you can have multiprocessor machines with each processor having multiple cores.

1.6.6 Mobile Architecture

Nowadays, mobile handsets are no longer a means of enabling conversations but have turned into a small but powerful computer having many features like fast memory, support for software for numerous applications like chatting, document editing, entertainment, news etc. These phones have become really handy equipped with efficient performance and it has become really difficult to have a demarcation between a computer and a mobile phone.

Just like any computer system, mobiles too have an input unit, a Central Processing Unit (CPU) and an output unit as shown in Figure 1.15. The input unit could be a keyboard based or a touch screen based input mechanism and the output unit can be the screen or audio or video system. However, the CPU in the case of mobiles can be considered to be having two processing units viz. Communications Processing Unit and an Applications Processing Unit to cater to mobile calls and handling various applications available in the form of Apps. In addition, some other functional units like Display management, Memory management, Power management, Data management can be considered to be associated with the mobile system architecture but the discussion is beyond the scope of this course. All these units, under the mobile CPU can be understood to be working under the control of a mobile Control Unit for control and synchronization purposes.

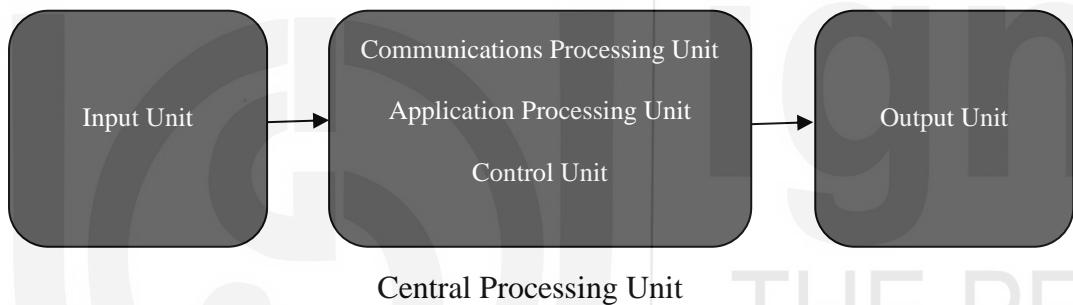


Figure 1.14: Block diagram of Mobile System

Check your progress 3:

Q1: Differentiate between the micro-architecture and ISA.

Q2: Differentiate von Neumann architecture and Harvard architectures.

Q3: Differentiate multiprocessor and multicore processor

1.7 SUMMARY

This unit introduces you to some of the basic architectural features of a computer system. A computing device consists of some basic units, like processing unit, which includes control unit and arithmetic logic unit, memory, input and output devices and data paths. This unit also explains the concepts of various computer architecture, which represents, how these various component of computer can be used to execute an instruction set. The unit also introduces to the concept of ISA, multiprocessor and multicore and mobile architecture. A detailed discussion on the various aspects of Computer organization has not been included in this unit, they will be discussed in the subsequent blocks and units. This unit also introduces you to the concept of instruction and its execution by a computer.

1.8 SOLUTIONS/ANSWERS

Check Your Progress 1

- 1) (a) True (b) True (c) False (d) True
- 2) An instruction in a computer system is a binary code, which is interpreted by the control unit of a computer and it communicates the following information to CPU:
 - the operation which is to performed
 - the operands and their location
 - the possible information of storage of results etc.
- 3) The CU controls - (1) the sequence of instruction execution, (2) the communication through the data paths, (3) the communication among different units (4) the operation to be performed by ALU (5) what to do in case of errors etc. Thus, in general, CU is responsible for complete control of a computer

The ALU primarily performs the arithmetic and logical and shift operations on the specified data. Memory stores the data as well as instructions and I/O devices are used to input data or instructions as well as display of results.

Check Your Progress 2

- 1) (i) True (ii) False (iii) False (iv) False (v) True
- 2) An interrupt is the process of stopping the execution of currently executing program due to occurrence of some event, which requires attention of the CPU. The causes of the interrupt may be division by zero; arithmetic overflow, program address space violation, starting or completion of I/O, errors etc.
- 3) Interrupt can be processed by suspending the execution of currently executing program and executing the ISR of the interrupting event. It may be noted that interrupts can be acknowledged only if the interrupts are enabled.

Check your progress 3:

- 1) An ISA defines the set of instructions, but a micro-architecture provides details on how various functions will be performed by various units. An ISA is a top level design, whereas micro-architecture is a detailed design. A faster compatible computer can be designed with same ISA but more efficient

micro-architecture. ISA simplifies the job of programmers, who may use same instruction set to write programs.

- 2) The von Neumann architecture uses same memory for data and instruction, the Harvard architecture may have separate memories for instructions and data. In von Neumann architecture data and instructions cannot be accessed simultaneously but in Harvard architecture it is possible. The von Neumann architecture is simpler to implement in comparison to Harvard architecture.
- 3) A multiprocessor system may have multiple processors, whereas multicore processor have multiple CPUs in a single processor. Today, a multiprocessor system can be constructed using multicore processor chips. Both these technology are of the type MIMD, which is multiple instruction and multiple data form of multiprocessing, thus, allow multiple sections of programs being executed at the same time operating on separate data.



UNIT 2 DATA REPRESENTATION

Structure	Page Nos.
2.0 Introduction	
2.1 Objectives	
2.2 Data Representation in Computer	
2.3 Representation of Characters	
2.4 Number Systems	
2.5 Negative Number Representation Using Complements	
2.5.1 Fixed Point Representation	
2.5.2 Binary Arithmetic using Complement notation	
2.5.3 Decimal Fixed Point Representation	
2.6 Floating Point Representation	
2.7 Error Detection and Correction Codes	
2.8 Summary	
2.9 Solutions/ Answers	

2.0 INTRODUCTION

In the first Unit of this Block, you have learnt the concepts relating to different architectures of a Computer System. It also explains the process of execution of an instruction highlighting the use of various components of a Computer system. This Unit explains about how the data is represented in a computer system.

The Unit first defines the concepts of number systems in brief, which is followed by discussion on conversion of numbers of different number systems. An important concept of signed complement notation, which is used for arithmetic operations on binary numbers, has been explained in this Unit. This is followed by discussion on the fixed point and floating point numbers, which are used to represent the numerical data in computer systems. This Unit also explains the error detection and correction codes and introduces you to basics of computer arithmetic operations.

2.1 OBJECTIVES

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

- Represent numeric data using binary, octal and hexadecimal numbers;
- Perform number conversions among various number bases;
- Define the ASCII and UNICODE representation for a character set;
- Explain the fixed and floating point number formats;
- Perform arithmetic operations using fixed and floating point numbers ; and
- Explain error detection and correction codes

2.2 DATA REPRESENTATION IN COMPUTER

A computer system is an electronic device that processes data. An electronic device, in general, consists of two stable states represented as 0 and 1. Therefore, the basic unit of data on a computer is called a **Binary Digit** or a **Bit**. With the advances in quantum computing technology a new basic unit called Qubits has emerged, which also represent 0 and 1, but with the difference that it can also represent both the states at the same time. The concepts of Quantum computing is beyond the scope of this unit.

A computer performs three basic operation on data, viz. data input, processing and data output. The data input and information output, in general, is presented in text, graphics, audio or other human recognizable form. Therefore, all human readable characters, graphics, audio and video should be coded using bits such that computer is able to interpret them. The most common code to represents characters into computer are ASCII and UNICODE. Pictures and graphs can be represented using pixel (picture elements), digital sound and video are represented by coding the frames in digital formats. Since graphics, digital audio and digital video, which are stored on storage devices as files, are very large in size, therefore, a large number of storage formats that use data compression techniques are used for represent digital information. Some of these concepts are explained in Unit 8.

The numeric data is used for computation in computer. However, as computer is an electronic device, it can only process binary data. Thus, in general, numeric data is to be converted to binary for computation. Computer uses fixed point and floating point representation for representing numeric data. Data in computer is stored in random access memory (RAM) and is required to be transferred in or out of the RAM for the purpose of processing, therefore, an error detection mechanism may be employed to identify and correct simple errors while transfer of binary data. The subsequent sections of the Unit explains the character representation, representation of binary numbers and error detection mechanism.

2.3 REPRESENTATION OF CHARACTERS

A character can be presented in a computer using a binary code. This code should be same for different types of computers, else the information from one computer will not be transferable to other computers. Thus, there is need of a standard for character representation. A coding standard has to address two basic issues, viz. the length of code, and organisation of different types of characters, which include printable character set of different languages and special characters. Two important character representation schemes are ASCII and UNICODE, which are discussed next.

American Standard Code for Information Interchange (ASCII)

ASCII was among the first character encoding standard. The length of ASCII is 7-bit. Thus, it can represent $2^7=128$ characters. It represents printable characters - English alphabets (both lower case and upper case), decimal digits, special characters as present on the present day keyboard, certain graphical characters etc.; and non-printable control characters. American National Standards Institute (ANSI) has designed a standard ISO 646 in 1964, which is based on ASCII.

However, as the basic unit of computer storage was 8, 16, 32 or 64 bits, the ASCII was extended to create an 8 bit code. This code could represent $2^8=256$ characters, most of the additional characters in extended code were graphics characters. ANSI has designed a code ISO 8859 for extended ASCII. It may be noted that ASCII has many variants, which are based on characters used in different countries.

In ASCII, the coding sequence of characters is very interesting. Simple binary arithmetic operations can result in conversion of lower case to upper case characters. For example, character 'A' in ASCII is represented as the binary value 100 0001, which is equivalent to a value 65 in decimal notation, whereas the character 'a' is stored as binary value 110 0001, which is the decimal 97. Thus, conversion from lowercase to upper case and vice-versa may be performed by subtracting or adding 32, as the case may be.

ISO 8859 which is based on extended ASCII is a good representation; however, all the languages cannot be represented using ASCII as its length is very small. Therefore, a

new standard that could represent almost all the characters of all the languages was developed. This is called the UNICODE.

Unicode

Unicode is a standard for character representation, which provides a unique code also called *code point*, for every character of almost all the languages of the world. The set of all the codes is called *code space*. The code space is divided into 17 continuous sequences of codes called *code planes*, with each code plane can represent 2^{16} codes. Thus, Unicode values ranges from $U+0000_{16}$ to $U+10FFFF_{16}$. Here $U+$ represents the Unicode followed by the hexadecimal value of a code point. The code planes of the Unicode being $U+00000_{16}$ to $U+0FFFF_{16}$; $U+10000_{16}$ to $U+1FFFF_{16}$; $U+20000_{16}$ to $U+2FFFF_{16}$; ..., $U+F0000_{16}$ to $U+FFFFF_{16}$; and $U+100000_{16}$ to $U+10FFFF_{16}$. You can learn about more details on Unicode from the further readings. Also read the hexadecimal number system given in the next section to learn about the hexadecimal values given above.

One of the major advantages of using Unicode is that it helps in seamless digital data transfer among the applications that use this character formatting, thus, not causing any compatibility problem.

Unicode code points may consist of about 24 binary digits, however, all of these code points may not be required for a given set of data. In addition, a digital system requires the data in the units of bytes. Thus, a number of encodings has been designed to represent Unicode code points in a digital format. Two of these popular encodings - Unicode Transformation Formats are UTF-8 and UTF-16. UTF-8 uses 1 to 4 bytes to represent the code points of Unicode. Most of the 1 byte UTF-8 code points are compatible to ASCII. UTF-16 represents code points as one or two 16-bit code units. The standard ISO 10646 represents various Unicode coding formats.

In general, if you are working with web pages having mostly English language, UTF-8 may be a good choice of character representation. However, if you are creating a multi-lingual web page, it may be a good idea to use UTF-16.

Indian Standard Code for information interchange (ISCII)

The ISCII is and ASCII compatible code consisting of eight-bits. The code for values 0 to 127 in ISCII is similar to ASCII; however, for the values 128 to 225 it represents the characters of Indian scripts. IS 13194:1991 BIS standard defines the details of ISCII. However, with the popularity of Unicode, its use has now been limited.

2.4 NUMBER SYSTEMS

A number system is used to represent the quantitative information. This section discusses the binary, octal and hexadecimal number systems

Formally, a number system is represented using a base or radix, which is equal to the distinct digits used by that system, and the position of a digit in a number. For example, the decimal number system has a base 10. It consists of ten decimal digits, viz. 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9; and a place value as shown in the following example:

$$9 \times 10^4 + 8 \times 10^3 + 7 \times 10^2 + 6 \times 10^1 + 5 \times 10^0 + 4 \times 10^{-1} + 3 \times 10^{-2} + 2 \times 10^{-3} + 1 \times 10^{-4} \\ = 98765.4321$$

Binary Numbers: A binary number system has a base 2 and consists of only two digits 0 and 1, which are also called the bits. For example, 1001_2 represent a binary number with four binary digits. The subscript 2 represents that the number 1001 has a base 2 or in other words is a binary number.

Note: The subscript shown in the numbers represents the base of the number. In case a subscript is not given then please assume it as per the context of discussion.

Conversion of binary number to Decimal equivalent:

A binary number is converted to its decimal equivalent by multiplying each binary digit by its place value. For example, a seven digit binary number 1001001_2 can be converted to decimal equivalent value as follows:

Binary Digits of Number	1	0	0	1	0	0	1
The place value	2^6 =64	2^5 =32	2^4 =16	2^3 =8	2^2 =4	2^1 =2	2^0 =1
Binary digit \times Place value	1×64	0×32	0×16	1×8	0×4	0×2	1×1
Computed values	64	0	0	8	0	0	1
Sum of the computed values	$64+0+0+8+0+0+1 = 73$ in Decimal						

You may now try converting few more numbers. Try 0010001 , which will be $16+1=17$; 1111111 will be $64+32+16+8+4+2+1=127$. So a 7 bit binary number can contain decimal values from 0 to 127.

Octal Numbers: An Octal number system has a base of 8, therefore, it has eight digits, which are 0,1,2,3,4,5,6,7. For example, 76543210_8 is an octal number.

Conversion of Octal number to Decimal equivalent:

An Octal number is converted to its decimal equivalent by multiplying each octal digit by its place value. For example, an octal number 5432_8 can be converted to decimal equivalent value as follows:

Octal Digits of Number	5	4	3	2
The place value	8^3 =512	8^2 =64	8^1 =8	8^0 =1
Octal digit \times Place value	5×512	4×64	3×8	2×1
Computed values	2560	256	24	2
Sum of the computed values	$2560+256+24+2=2842_{10}$			

Hexadecimal Numbers: A hexadecimal number system has a base of 16, therefore, it uses sixteen digits, which are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A(10), B(11), C(12), D(13), E(14), F(15). For example, $FDA9_{16}$ is a hexadecimal number.

Conversion of Hexadecimal number to Decimal equivalent:

A hexadecimal number is converted to its decimal equivalent by multiplying each hexadecimal digit by its place value. For example, a hexadecimal number $13AF_{16}$ can be converted to decimal equivalent value as follows:

Hexadecimal Digits of Number	1	3	A=10	F=15
The place value	16^3 =4096	16^2 =256	16^1 =16	16^0 =1
Hexadecimal digit \times Place value	1×4096	3×256	10×16	15×1
Computed values	4096	768	160	15
Sum of the computed values	$4096+768+160+15=5039_{10}$			

Conversion of Decimal to Binary: A decimal number can consists of Integer and fraction part. Both are converted to binary separately.

Process:

For Integer part: Repetitively divide the quotient of integer part by 2 keeping remainder separate till quotient is 0. Collect all the remainders from last remainder to first to make equivalent binary

For Fractional part: Repetitively multiply the fraction by 2 and maintain the list of integer value that is obtained till fraction becomes 0. Collect all the integer values.

The following example explains the process of Decimal to binary conversion.

Example 1: Convert the decimal number 22.25 to binary number.

Solution:

For Integer part: Repetitively divide the quotient of integer part by 2 keeping remainder separate till quotient is 0. Integer value of example: 22			For Fractional part: Repetitively multiply the fraction by 2 and maintain the list of integer value that is obtained till fraction becomes 0. Fraction value of example: .25			
Integer Part	After Division by 2		Direction of Reading the Result	Fraction Part	After multiplication by 2	Direction of Reading the Result
	Quotient	Remainder		Result	Integer part	
22	11	0	↑	.25	0.50	0
11	5	1		.50	1.00	1
5	2	1		.00	STOP	Ans: 01
2	1	0				
1	0	1				
0	STOP			Ans: 10110		

22.25_{10} in binary is 10110.01_2

Verification

Binary Digits of Number	1	0	1	1	0	.	0	1
The place value	2^4 =16	2^3 =8	2^2 =4	2^1 =2	2^0 =1		2^{-1} =1/2	2^{-2} =1/4
Digit \times Place value	1×16	0×8	1×4	1×2	0×1		0×0.5	1×0.25
Computed values	16	0	4	2	0		0	0.25
Sum of the computed values	22.25_{10}							

--

The method as shown above is a standard technique. You must start using this method for various problems. However, you may use the following simple technique of converting integer part of decimal numbers to binary.

Conversion from Decimal to Binary a simpler Process:

Assume a decimal number N is to be converted to binary. Now perform the following steps:

1. Is the decimal number N equal to a binary place value, then assign that place value to P and move to step 3.
2. Else, find the binary place values which is just lower to the decimal number N . Assign this place value to P . For example, for number 73 just lower place value is 64 , as $2^6 = 64$ and $2^7 = 128$.
3. Put 1 in the position of P and subtract the place value P from N .

4. If $(N-P) \neq 0$ then Repeat the steps 1 to 3 by taking new $N=N-P$

5. Put 0 in all the remaining places, where 1 has not been put.

The following example demonstrate this technique showing conversion of 73 to binary.

Example 2: Convert decimal numbers 73, 39 and 20 into binary using the method as above.

The following table shows the process of the conversion.

The place value	2^6 =64	2^5 =32	2^4 =16	2^3 =8	2^2 =4	2^1 =2	2^0 =1
$N = 73$				128 > 73 > 64, therefore, $P=64$			
Step 2 and 3	1			$N = N-P = 73-64 = 9$ (Not 0 so repeat)			
Step 2			16 > 9 > 8 so new $P=8$ and New $N=9-8=1$				
Step 3	1			1			
Step 1			Since 1 is a place value, new $P=1$ and New $N=1-1=0$				
Step 3	1			1			1
Step 4	1	0	0	1	0	0	1
Place values	64	32	16	8	4	2	1
$N=39$		1					
New $N=7$		1			1		
New $N=3$		1			1	1	
New $N=1$		1			1	1	1
Step 4	0	1	0	0	1	1	1
Place values	64	32	16	8	4	2	1
$N=20$			1				
New $N=4$			1		1		
Step 4	0	0	1	0	1	0	0

The logic as presented here can be extended to the fractional part, however, it is recommended that you may follow the repeated multiplication method as explained earlier for the fractions.

Conversion of Binary number to Octal Number

The base of a binary number is 2 and the base of octal number is 8. Interestingly, $2^3=8$. Thus, if you simply group three binary digits, the equivalent value may form the octal digit. However, you may be wondering how to group binary numbers. This is explained with the help of following example.

Example 3: Convert the binary 11001101.00111₂ into equivalent Octal number.

Process: The process is to group three binary digits. The grouping before the binary point is done from right to left and after the binary point from left to right. Each of the group then is converted to equivalent octal digit. The following table shows this conversion process.

Binary Number	-	1	1	0	0	1	1	0	1	.	0	0	1	1	1	-		
Grouping Directions		←————→						.		.	→————←							
Grouped (- replaced by 0)	0	1	1		0	0	1		1	0	1	.	0	0	1	1	1	0
Binary place values	4	2	1	4	2	1	4	2	1	.	4	2	1	4	2	1		
Equivalent Octal Digit	0+2+1=3	0+0+1=1		4+0+1=5	.	0+0+1=1	4+2+0=6			.	0+0+1=1	4+2+0=6						
Octal Number	3		1		5	.		1		6								

Therefore, 11001101.00111₂ is equivalent to 315.16₈

The base of a binary number is 2 and the base of hexadecimal number is 16. You may notice that $2^4=16$. Therefore, conversion of binary to hexadecimal notation may require grouping of 4 binary digits. This is explained with the help of following example.

Example 4: Convert the binary 11001101.00111_2 into equivalent hexadecimal number.

Process: The process is almost similar to binary number to octal number conversion except now four binary digits are combined as given in the following table.

Binary Number	1	1	0	0	1	1	0	1	.	0	0	1	1	1	-	-	-
Grouping Direction			←						.		→						
Grouped	1	1	0	0	1	1	0	1	.	0	0	1	1	1	0	0	0
Binary place values	8	4	2	1	8	4	2	1	.	8	4	2	1	8	4	2	1
Hexadecimal digit	8+4+0+0=12	8+4+0+1=13	.	0+0+2+1=3	8+0+0+0=8												
Hexadecimal	12 is C	13 is D	.	3	8												

Therefore, 11001101.00111_2 is equivalent to 315.16_8 and $CD.38_{16}$

As computer is a binary device, therefore, all the numbers of different number systems may be represented in binary format. This is shown in the following table.

Decimal Number	Binary Coded Decimal	Equivalent Octal Number	Binary coded Octal	Hexadecimal Number	Binary-coded Hexadecimal
0	0000	0	000	0	0000
1	0001	1	001	1	0001
2	0010	2	010	2	0010
3	0011	3	011	3	0011
4	0100	4	100	4	0100
5	0101	5	101	5	0101
6	0110	6	110	6	0110
7	0111	7	111	7	0111
8	1000	10	001 000	8	1000
9	1001	11	001 001	9	1001
10	0001 0000	12	001 010	A	1010
11	0001 0001	13	001 011	B	1011
12	0001 0010	14	001 100	C	1100
13	0001 0011	15	001 101	D	1101
14	0001 0100	16	001 110	E	1110
15	0001 0101	17	001 111	F	1111
16	0001 0110	21	010 000	10	0001 0000
17	0001 0111	22	010 001	11	0001 0001
...					
49	0100 1001	61	110 001	31	0011 0001
...					
63	0110 0110	77	111 111	3F	0011 1111

Table 1: Decimal, Octal, Hexadecimal Numbers

Please note the following points in the Table 1 given above.

- The Binary coded decimal (BCD) is the representation of each decimal digit to a sequence of 4 bits. For example, a decimal number 12 in BCD is 0001 0010. This representation is used in several calculators for performing computation.
- It may be noted that BCD is not binary equivalent value. For example, the BCD value of decimal 49 is 0100 1001 but its binary equivalent value is 0011 0001.
- Please also note that binary coded hexadecimal values are equivalent to binary value of a number. For example, decimal value 63 in hexadecimal binary notation is 0011 1111, which is same as its binary value.

The conversion of decimal to octal and hexadecimal may be performed in the same way as done using repeated division or multiplication of binary. The process is exactly same except, in decimal number to octal or hexadecimal number conversion division is done by 8 or 16 respectively.

Check Your Progress 1

1) Perform the following conversions:

- 11100.01101₂ to Octal and Hexadecimal
 - 1101101010₂ to Octal and Hexadecimal
-
.....
.....

2) Convert the following numbers to binary.

- 119₁₀
 - 19.125₁₀
 - 325₁₀
-
.....
.....

3) Convert the numbers to hexadecimal and octal.

- 119₁₀
 - 19.125₁₀
 - 325₁₀
-
.....
.....

2.5 NEGATIVE NUMBER REPRESENTATION USING COMPLEMENTS

You have gone through the details of binary representation of character data and the number systems. In general, you use positive and negative integers and real numbers for computation. How these numbers can be represented in binary? This section describes how positive and negative numbers can be represented in binary for performing arithmetic operations.

In general, Integer numbers can be represented using the sign and magnitude of the number, whereas real numbers may be represented using a sign, decimal point and magnitude of integral and fractional part. Real numbers can also be represented using a scientific exponential notation. This section explains how integers can be represented as binary numbers in a computer system.

Integer representation in binary:

An integer is represented in binary using fixed number of binary digits. One of the simplest representations for representing integer would be - to represent the sign using a bit; and magnitude may be represented by the remaining bits. Fortunately, the value of sign can be either positive or negative, therefore, it can easily be represented in binary. The + sign can be represented using 0 and – sign can be represented using 1.

For example, a decimal number 73 has a sign + (bit value 0) and magnitude 73 (binary equivalent 1001001). The following table shows some of the numbers using this *Sign-magnitude Representation*:

Number	Sign Bit	Magnitude						
		1	0	0	1	0	0	1
+73	0	1	0	0	1	0	0	1
-73	1	1	0	0	1	0	0	1
+39	0	0	1	0	0	1	1	1
-39	1	0	1	0	0	1	1	1
+127	0	1	1	1	1	1	1	1
-127	1	1	1	1	1	1	1	1
0	0	0	0	0	0	0	0	0
-0	1	0	0	0	0	0	0	0

Table 2: 8-bit Sign-Magnitude representation

Please note the following points about this 8 bit sign-magnitude representation:

- It represents number in the range 0 to +127 and -0 to -127. Therefore, the range of the numbers that can be represented using these 8 bits is -127 to +127.
- It represents 255 different numbers viz. -127 to -1, ±0 and +1 to +127.
- The number of bits, which are used to represent the magnitude of the number, can be used to determine the maximum and minimum numbers that are representable.
- There are two representations of zero +0 and -0

Is this representation suitable for representing numbers for computation? It has one basic problem that the sequences of steps to perform arithmetic operations are not straight forward. For example, for adding +73 and -39, first you need to compare the sign of the numbers, and as the signs are different in this case, therefore, you should perform subtraction of smaller number from the bigger number and finally assigning the sign of bigger number to the result.

Is there any better representation? Yes, an interesting representation that uses complement of a number to represent negative numbers has been designed. What is a complement of a number?

Complement notation: A complement, by definition, is a number that makes a given number complete. For the decimal numbers, this completeness can be defined with respect to the highest value of the digit, i.e. 9 or the next higher value, i.e. 10. These are called 9's and 10's complement respectively for the decimal numbers.

For example, for a decimal digit 3, the 9's complement would be $9-3=6$ and 10's complement would be $10-3=7$.

In general, for a number with base B two types of complements are defined – $(B-1)$'s complement and B 's complement. For example, for decimal system base value B is 10. Therefore, for decimal numbers two complements, viz. 9's and 10's complements, are defined. Thus, for binary system where base is 2, the two complements, viz. 1's complement and 2's complement, are defined. The following example illustrates the steps of finding 9's and 10's complement for decimal numbers.

Example 5: Compute the 9's complement and 10's complement for a four digit decimal number 1095, 8567 and 0560.

Solution: Following table shows the process:

Complement	Operation	The Number
9's Complement	Number	1 0 9 5
	Subtract each digit from 9	8 9 0 4
10's Complement	Add 1 in the 9's complement	- - - 1
	It results in 10's complement	8 9 0 5
9's Complement	Number	8 5 6 7
	Subtract each digit from 9	1 4 3 2
10's Complement	Add 1 in the 9's complement	- - - 1
	It results in 10's complement	1 4 3 3
9's Complement	Number	0 5 6 0
	Subtract each digit from 9	9 4 3 9
10's Complement	Add 1 in the 9's complement	- - - 1
	It results in 10's complement	9 4 4 0

Table 3: Computation of 9's and 10's complement

Please note that the sum of the number and its 9's complement for the numbers of 4 digits is 9999, and the sum of the number and its 10's complement is 10000. The 9's and 10's complement of the numbers can be used in a computer system when BCD numbers are used instead of binary numbers. Similarly, the 1's and 2's complement can be computed for binary numbers. The following example demonstrates the complement notation in binary.

Example 6: Compute the 1's and 2's complement for the binary numbers 1001_2 , 1111_2 and 0000_2 using representation, which has four bits.

Solution:

Solution: Following table shows the process:

Complement	Operation	The Number
1's Complement	Number	1 0 0 1
	Subtract each digit from 1	0 1 1 0
2's Complement	Add 1 in the 1's complement	- - - 1
	It results in 2's complement	0 1 1 1
1's Complement	Number	1 1 1 1
	Subtract each digit from 1	0 0 0 0

2's Complement	Add 1 in the 1's complement	-	-	-	1
	It results in 2's complement	0	0	0	1
1's Complement	Number	0	0	0	0
	Subtract each digit from 1	1	1	1	1
2's Complement	Add 1 in the 1's complement	-	-	-	1
	It results in 2's complement. There will be a carry bit from the last digit	0	0	0	0

Table 4: Computation of 1's and 2's complement

Please note the following in the table as above:

- On subtracting 1 from binary digit will result in change of bit from 0 to 1 OR 1 to 0.
- When you add binary digit 1 with 1, then it results in a sum bit of 0 and carry bit as 1.
- 1's complement of 0000 is 1111, when 1 is added to it, you will get 10000 as the 2's complement. Since, only 4 binary digits are used in the notation as above, the fifth digit, which is 1, is ignored while taking the complement.

An interesting observation from the Table 4 is that 1's complement can be obtained simply by changing 1 to 0 and 0 to 1. For obtaining 2's complement leave all the trailing zeros and the first 1 intact and after that complement the remaining bits. For example, for an eight bit binary number 10101100, the complement can be done as follows:

Number	1	0	1	0	1	1	0	0
1's Complement change every bit from 0 to 1 OR 1 to 0	0	1	0	1	0	0	1	1
Number	1	0	1	0	1	1	0	0
For 2's complement leave the trailing 0's till first 1						1	0	0
then complement remaining bits(change 0 to 1 or 1 to 0)	0	1	0	1	0			
2's Complement of the Number	0	1	0	1	0	1	0	0

Table 5: Computation of 1's and 2's complement

But, how are these complement notations used in a computer system to represent integers? The next sub-section explains the integral representation of computers.

2.5.1 Fixed Point Representation

A computer system uses registers or memory locations to store arithmetic data like numbers. The number stored in these locations are of fixed size, such as 8 or 16 or 32 or 64 or 128 bits etc. Interestingly, binary point is not represented in the numbers, rather its location is assumed. The fixed point number representation assumes that the binary point is at the end of all the binary digits, thus, can be used to represent integers. Since, Integers include positive and negative number both, therefore, fixed point number also use one bit as the sign bit as shown in Table 2. Fixed point numbers may use either signed magnitude notation or complement notation. However, as explained in the previous section signed magnitude notation is not a natural notation for binary arithmetic, the complement notation is used in computers. The complement notation works well for the digital binary numbers as they are of fixed length. For the sake of simplicity, in this unit we will use an complement notation having a length of 8-bits.

For the fixed point number representation signed 1's complement and signed 2's complement notation can be used. The signed complement notation is same as the complement notation as introduced in the previous section, except that it uses a sign bit, in addition to represent magnitude. In signed 1's and 2's complement notation the positive number has the same magnitude as that of binary number with the sign bit as zero, however, the negative numbers are represented in complement form. The

following example explains the process of conversion of decimal numbers to signed 1's or signed 2's complement notation.

Example 7: Represent the +73, -73, +39, -39, +127, -127 and 0 using signed 1's complement notation.

Solution: The table 6 shows the values in signed 1's complement notation of length 8 bits (S is the sign bit). Please note that even in signed 1's complement notation there are two representations for 0. The number range for 1's complement for this 8 bit representation is -127 to -0 and +0 to +127. So it can represent $2^8 - 1$ (as two representation of 0) = 255 numbers.

Number	Process	S	7 bits						
			0	1	0	0	1	0	0
+73	Sign is 0 (positive) and 7 bit magnitude is same as binary equivalent value of 73	0	1	0	0	1	0	0	1
-73	Take 1's complement of all the 8 bits (including sign bit) to obtain -73	1	0	1	1	0	1	1	0
+39	Follow same process as stated for +73	0	0	1	0	0	1	1	1
-39	Follow same process as stated for -73	1	1	0	1	1	0	0	0
+127	Follow same process as stated for +73	0	1	1	1	1	1	1	1
-127	Follow same process as stated for -73	1	0	0	0	0	0	0	0
0	Follow same process as stated for +73	0	0	0	0	0	0	0	0
-0	Follow same process as stated for -73	1	1	1	1	1	1	1	1

Table 6: 8-bit Signed 1's complement notation

Example 8: Represent the +73, -73, +39, -39, +127, -127, 0 and -128 using signed 2's complement notation.

Solution: The table 7 shows the values in signed 2's complement notation of length 8 bits (S is the sign bit). Please note that in signed 2's complement notation there is a unique representations for 0, therefore, -128 can also be represented. Thus, the range of the number that can be represented using signed 2's complement notation is -128 to +127. Thus, a total of 256 numbers can be represented using signed 2's complement notation.

Number	Process	S	7 bits						
			0	1	0	0	1	0	0
+73	Sign is 0 (positive) and 7 bit magnitude is same as binary equivalent value of 73	0	1	0	0	1	0	0	1
-73	Take 2's complement of the number (including sign bit) to obtain -73	1	0	1	1	0	1	1	1
+39	Follow same process as stated for +73	0	0	1	0	0	1	1	1
-39	Follow same process as stated for -73	1	1	0	1	1	0	0	1
+127	Follow same process as stated for +73	0	1	1	1	1	1	1	1
-127	Follow same process as stated for -73	1	0	0	0	0	0	0	1
0	Follow same process as stated for +73	0	0	0	0	0	0	0	0
-0	Follow same process as stated for -73	0	0	0	0	0	0	0	0
-128	-127-1 is = -128	1	0	0	0	0	0	0	0

Table 7: 8-bit Signed 2's complement notation

In general, a signed 2's number representation of n bits can represent numbers in the range -2^{n-1} to $+(2^{n-1}-1)$. Therefore, an 8 bit representation can represent the numbers in the range -2^{8-1} to $+(2^{8-1}-1)$; i.e. -2^7 to $+(2^7-1)$, which is -128 to +127. For a 16 bit representation this range will be -2^{15} to $+(2^{15}-1)$, which is -32768 to +32767. Please relate these ranges to range given in programming languages like C.

Signed 2's complement notation is one of the best notation to perform arithmetic on numbers. Next, we explain the process of performing arithmetic using fixed point numbers.

2.5.2 Binary Arithmetic using Complement notation

Data Representation

In this section, we discuss about binary arithmetic using fixed point complement notation.

Arithmetic addition: Arithmetic addition operation can be performed using any of the signed-magnitude, singed 1's complement and signed 2's complement notation.

Addition using signed-magnitude notation:

The process of addition of two numbers using signed magnitude notation will require the following steps:

Step 1: Check if the numbers have similar or different signs.

Step 2: If signs are same then just add the two numbers, otherwise identify the number having bigger magnitude (in case both numbers have same magnitude then first number may be assumed as bigger number) and subtract the smaller number from bigger number.

Step 3: If signs of the number are same then check if the result exceeds the size of number of bits of the representation, if that happens then report overflow of number. For the numbers with different signs overflow cannot occur.

Step 4: The sign of the final number is the sign of any operand, if signs are same; or the sign of the bigger number, if signs are different.

The following example explains the process of addition using signed-magnitude notation. The example, uses an 8 bit representation.

Example 9: Add the decimal numbers 75 and -80 using signed magnitude notation, assuming the 8-bit length of the notations.

Solution: The numbers are (The left most bit is the Sign bit):

Number	Signed Magnitude				Signed 1's Complement				Signed 2's Complement														
+75	0	1	0	0	1	0	1	1	0	1	0	0	1	0	1	1							
+80	0	1	0	1	0	0	0	0	1	0	1	0	0	0	1	0	0	0	0				
-80	1	1	0	1	0	0	0	0	1	0	1	0	1	1	1	1	0	1	1	0	0	0	0

Table 8: 8-bit representation of +75, +80 and -80

For the present example, signs of two numbers are different and the magnitude of -80 is higher than 75, therefore, 75 is subtracted from 80 as shown in the following table and the sign of the -80, which is minus is selected for the output. Please note that during the subtraction, carry is required to be taken as done in the decimal numbers. In binary, if a carry is taken then the number changes to two digit number as shown in the table. Please note that 10 of binary is a value 2 in decimal.

Number		Signed Magnitude							
Carry taken out from the bit		No	No	No	yes	yes	yes	yes	No
Updated bit value					0	10 1	10 1	10 1	10
-80		1	1	0	1	0	0	0	0
+75		0	1	0	0	1	0	1	1
Subtraction	Sign bit	1-1	0-0	0-0	1-1	1-0	1-1	10-1	
Result = -5		1	0	0	0	1	0	1	

Table 9: 8-bit addition using Signed magnitude notation

Example 10: Add the decimal numbers 75 and 80 using signed magnitude notation, assuming the 8-bit length of the notations.

Solution: The process of addition of +75 and +80 in signed magnitude representation is shown below:

Number	Signed Magnitude							
	No	Yes	No	No	No	No	No	No
Carry from previous bit addition	No	Yes	No	No	No	No	No	No
Carry bit value	1							
+75	0	1	0	0	1	0	1	1
+80	0	1	0	1	0	0	0	0
	1	1+1=10	0+0	0+1	1+0	0+0	1+0	1+0
Result is -27 (error)	1	0	0	1	1	0	1	1

Table 10: 8-bit addition using Signed magnitude notation having overflow

The addition of 7 bit magnitude has resulted in 8 bit output, which cannot be stored in this notation as this notation has a length of 8 bits with 1 sign bit. The last bit will be lost and you will obtain an incorrect result -27. This problem has occurred as the size of the number is fixed. This is called the overflow. You may please note that the actual addition of the 75 and 80 is 155, which is beyond the range of 8 bit signed magnitude representation, which is -127 to +127. This is why you should be careful while selecting the integral data types in a programming languages. For example, in case you have selected small unsigned integer of byte size for a variable, then you can store only the number values in the range 0 to 255 in that variable.

Addition using signed-1's complement notation:

In signed 1's complement notation the addition process is simpler than signed-magnitude representation. An interesting fact is that in this notation you do not have to check the sign, just add the numbers, why? This is due to the fact that complement of a number as defined makes it complete, the binary digits are complement of each other and even the sign bits are complement. Therefore, the process of addition of two signed 1's complement notation just requires addition of the two numbers, irrespective of the sign. The process of addition in signed 1's complement representation will require the following steps:

Step 1: Just add the numbers, irrespective of sign.

Step 2: Now check the following conditions:

Carry in to the Sign Bit	Carry out of the Sign bit	Comments
No	No	Result is fine
Yes	Yes	Add 1 to result and it is fine
No	Yes	Overflow, incorrect result
Yes	No	Overflow, incorrect result

Table 11: The conditions of 1's complement notation, while addition

The following example demonstrates the process of addition .

Example 11: Add the decimal numbers 75 and -80 using signed 1's complement notation, assuming the 8-bit length of the notations.

Solution: The numbers are (The left most bit is the Sign bit). The Table 8 shows the values of +75 and -80 in signed 1's complement notation.

Number	Signed 1's Complement Notation							
	No	No	No	yes	yes	yes	yes	-
Carry from previous bit								

addition								
Updated bit value	-	-	-	1	1	1	1	-
+75	0	1	0	0	1	0	1	1
-80	1	0	1	0	1	1	1	1
Addition	Sign bit (0+1=1)	1+0 =1	0+1 =1	1+0+0 =1	1+1+1 =11	1+0+1 =10	1+1+1 =11	1+1 =10
Result (-5)	1	1	1	1	0	1	0	0
Since, the result is negative, so taking a 1's complement to verify the magnitude								
-Result=+5	0	0	0	0	0	1	0	1

Table12: 8-bit addition using Signed 1's complement notation

Example 12: Add the decimal numbers 75 and 80 using signed 1's complement notation, assuming the 8-bit length of the notations.

Solution: The process of addition of +75 and +80 in signed magnitude representation is shown below:

Number	Carry out (9th bit)	Signed 1's Complement Notation							
Carry from previous bit addition	No	Yes	No	No	No	No	No	No	-
Updated bit value	No Carry out of the sign bit addition	Carry in to Sign Bit 1	-	-	-	-	-	-	-
+75		0	1	0	0	1	0	1	1
+80		0	1	0	1	0	0	0	0
Addition	Sign bit (0+0+1=1)	1+1 =10	0+0 =0	0+1 =1	1+0 =1	0+0 =0	1+0 =1	1+0 =1	=1
Result (A negative number)		1	0	0	1	1	0	1	1

There is carry in the sign bit (1) and no carry out of the sign bit. Therefore, as per Table 11, there is an overflow and the result is incorrect. You can observe that the result is negative for addition of two positive numbers, which is NOT possible.

Table 13: 8-bit addition using Signed 1's complement notation having overflow

Once again, please observe that the range of numbers in 8-bit 2's complement notation is -127 to +127, and the addition of two numbers 155 cannot be represented in 8-bits. Hence, there is an overflow.

Addition using signed-2's complement notation:

In signed 2's complement notation the addition process is simplest of these three representations. In this notation also, you do not have to check the sign, just add the numbers including the sign bit. The process of addition in signed 2's complement representation will use the following steps:

Step 1: Just add the numbers, irrespective of sign.

Step 2: Now check the following conditions:

Carry in to the Sign Bit	Carry out of the Sign bit	Comments
No	No	Result is fine
Yes	Yes	Result is fine
No	Yes	Overflow, incorrect result
Yes	No	Overflow, incorrect result

Table 14: The conditions of 2's complement notation, while addition

The following example demonstrates the process of addition using signed 2's complement notation.

Example 13: Add the decimal numbers (i) -69 -59 (ii) -69+59 (iii) +69-59 and (iv) +69=59

Solution: The numbers are (The left most bit is the Sign bit). The Table 14 shows the numbers in signed 2's complement notation. The left most bit is the sign bit

Number	Signed 2's Complement							
+69	0	1	0	0	0	1	0	1
- 69	1	0	1	1	1	0	1	1
+59	0	0	1	1	1	0	1	1
-59	1	1	0	0	0	1	0	1

Table 15: Numbers of example 13 in 2's complement notation

(i) -69-59

Number	Carry out (9th bit)	Signed 2's Complement Notation							
Carry from previous bit addition	yes	Carry in to Sign bit yes	yes	yes	yes	yes	yes	yes	-
Carry for addition	1	1	1	1	1	1	1	1	-
-69		1	0	1	1	1	0	1	1
-59		1	1	0	0	0	1	0	1
Addition of bits given above		1+1+1 =11	1+0+1 =10	1+1+0 =10	1+1+0 =10	1+1+0 =10	1+0+1 =10	1+1+0 =10	1+1 =10
Result	1	1	0	0	0	0	0	0	0

There is carry in to the sign bit (1) and there is a carry out of the sign bit (1). Therefore, as per Table 14, there is NO overflow and the result is correct and equal to -128. Discard the carry out bit (the 9th bit).

Table 16: Addition of two negative numbers without overflow

(ii) -69+59

Number	Carry out (9th bit)	Signed 2's Complement Notation							
Carry from previous bit addition	No	Carry in to Sign bit No	yes	yes	yes	No	yes	yes	-
Carry for addition	-	-	1	1	1	-	1	1	-
-69		1	0	1	1	1	0	1	1
+59		0	0	1	1	1	0	1	1
Addition of bits given above		1+0 =1	1+0+0 =1	1+1+1 =11	1+1+1 =11	1+1 =10	1+0+0 =1	1+1+1 =11	1+1 =10
Result	-	1	1	1	1	1	0	1	0

There is No carry in to the sign bit and there is No carry out of the sign bit. Therefore, as per Table 14, there is NO overflow and the result is correct and equal to -10. Verify the result yourself.

Table 17: Addition of bigger negative number and smaller positive numbers. No overflow is possible.

(iii) +69-59

Data Representation

Number	Carry out (9th bit)	Signed 2's Complement Notation							
Carry from previous bit addition	yes	Carry in to Sign bit yes	No	No	No	yes	No	yes	-
Carry for addition	1	1				1		1	-
+69		0	1	0	0	0	1	0	1
-59		1	1	0	0	0	1	0	1
Addition of bits given above		1+0+1 =10	1+1 =10	0+0 =0	0+0 =0	1+0+0 =1	1+1 =10	1+0+0 =1	1+1 =10
Result	1	0	0	0	0	1	0	1	0

There is a carry in to the sign bit (1) and there is a carry out of the sign bit (1). Therefore, as per Table 14, there is NO overflow and the result is correct and equal to +10. Discard the carry out bit (the 9th bit). Verify the result yourself.

Table 18: Addition of smaller negative number and bigger positive numbers. No overflow is possible.

(iv) +69+59

Number	Carry out (9th bit)	Signed 2's Complement Notation							
Carry from previous bit addition	No	Carry in to Sign bit yes	yes	yes	yes	yes	yes	yes	-
Carry for addition	-	1	1	1	1	1	1	1	-
+69		0	1	0	0	0	1	0	1
+59		0	0	1	1	1	0	1	1
Addition of bits given above		1+0+0 =1	1+1+0 =10	1+0+1 =10	1+0+1 =10	1+0+1 =10	1+1+0 =10	1+0+1 =10	1+1 =10
Result	-	1	0	0	0	0	0	0	0

There is a carry in to the sign bit (1) but there is NO carry out of the sign bit. Therefore, as per Table 14, there is an *overflow* and the result is incorrect. Verify the result yourself. Overflow has occurred as the addition of the two numbers is +128, which is out of the range of numbers that can be represented using 8-bit signed 2's complement notation.

Table 19: Addition of two positive numbers.

It may be noted that for the signed 2's complement notation, which is using 8 bits representation, is -128 to +127, which can be checked from table 16 and table 19.

Overflow formally is defined as the situation where the result of operation on two or more numbers, each of size n digits, exceeds the size n.

Overflow may cause even your correct programs to output incorrect results, therefore, is a very risky error. One of the ways of avoiding overflow in programs is to select appropriate data types and verifying the results range.

Arithmetic Subtraction: In general, a computer system uses the signed 2's complement notation, which simplifies the process of addition and subtraction as well as has a single representation for 0. You can perform subtraction by just taking the 2's complement of the number that is to be subtracted, and thereafter just adding the two numbers just like it has been shown in this section.

Multiplication and division: Multiplication and division operations using signed 2's complement notations are not straight forward. One of the simplest approach to multiply two signed 2's complement numbers is by multiplying the positive numbers and then adjusting the result based on the sign. However, this approach is time consuming as well as not used for implementation of multiplication operation. There are a number of algorithms for performing multiplication and division. One such algorithm is the Booth's algorithm. A detailed discussion on these topics is beyond the scope of this course.

In several arithmetic computations binary representation of decimal number is used for performing arithmetic operations. The next subsection briefly explains this representation.

2.5.3 Decimal Fixed Point Representation

Decimal digits can be represented in binary directly using four bits, as there are only 10 decimal digits, whereas $2^4=16$ different values can be expressed using 4 bits. Thus, a BCD may be represented as 0000 (for decimal digit 0) to 1001 (for decimal digit 9). In addition, the sign can be represented using a single bit; however, it may change the format of representation. Thus, in decimal fixed point representation even sign is represented as four bits. Interestingly, the positive sign is represented using 1100 and negative sign is represented using 1101. Please note these two combinations are different from the representation of decimal digits, which is 0000 to 1001.

Example14: Represent +125 as BCD and a binary number.

+125 in BCD is given below:

Sign Digit	1	2	5
1100	0001	0010	0101

+125 in Binary:

S	7-bit magnitude						
-	64	32	16	8	4	2	1
0	1	1	1	1	1	0	1

Why is this representation needed? In several computing devices the computations are performed on binary coded decimals directly, without conversion to binary. One such device was old calculator. You may refer to further readings for more details on BCD arithmetic.

Check Your Progress 2

- 1) Write the BCD for the following decimal numbers:
 - i) -23456
 - ii) 17.89
 - iii) 299

.....
.....

- 2) Compute the 1's and 2's complement of the following binary numbers. Also find the decimal equivalent of the number.

- i) 1110 0010
- ii) 0111 1110
- iii) 0000 0000

- 3) Add the following decimal numbers by converting them to 8-bit signed 2's complement notation.

- i) +56 and -56
- ii) +65 and -75
- iii) +121 and +8

Identify, if there is overflow.

2.6 FLOATING POINT REPRESENTATION

In most real numbers, you may use a decimal point to distinguish integer and fraction part. However, in a computer system the position of binary point is assumed in the numbers. Fixed point representation, in general, fixes the location of the point towards the right most. Thus, integer values are represented using fixed point representation. What about the real numbers. A real number can be represented using an exponential notation. This forms the basis of binary number representation called floating point representation. For example, a decimal real number 29.25 can be represented as 0.2925×10^2 or 2925×10^{-2} .

The first part of the number is called the “mantissa” or “significand” and second part of the number is called the exponent. You may please note that the mantissa can either be integer or fraction as shown in the example; the exponent value is adjusted accordingly. In computer the mantissa and exponent both are represented as binary numbers and the location of binary point is assumed. The following example explains the binary floating point representation. This representation is IEEE 754 standard for 32-bit floating point number.

It has the following format:

Bit Positions from the left	1	2 to 9	10 to 32
-----------------------------	---	--------	----------

Introduction to Digital Circuits	Length of Field	1 bit	8 bits	23 bits	(a) Basic details
Purpose	To store the <i>Sign</i> bit	To store the <i>Exponent</i>		Stores the fractional <i>Significand</i> of the Number	
Comment	The <i>Sign</i> bit is for the <i>Significand</i>	The exponent is stored in <i>biased</i> form with a <i>bias</i> of 127		The <i>Significand</i> is stored as a <i>normalized</i> binary number	
		<p>Exponent (8 bits) so possible values 0 to 255. A bias of 127 is assumed. Let the exponent be <i>exp</i></p>			
		For <i>Exponent</i> value (<i>exp</i>) 0	All the bits of <i>M</i> are zeros. <i>M</i> is NOT zero (<i>M</i> may not be normalized)	The number is ± 0 depending on the sign bit. The Number is $\pm 0.M \times 2^{-126}$	
		For <i>Exponent</i> values (<i>exp</i>) from 1 to 254	Normalized representation is used, therefore, <i>the first bit is assumed to be 1</i> .	The Number is $\pm 1.M \times 2^{\exp-127}$	
		For <i>Exponent</i> value (<i>exp</i>) 255	All bits of <i>M</i> are zeros <i>M</i> is NOT zero.	The number is $\pm \infty$ depending on the sign bit It does NOT represent a valid Number	

(b) Single Precision 32-bit IEEE-754 Standard

Table 20: IEEE 754 Floating Point 32-bit Number Representation

The three terms in Table 20 are *fractional Significand*, *bias* and *normalized*. They are explained below:

fractional Significand: Floating point number assumes that the position of binary point is prior to the *Significand*, therefore, *Significand* is a fraction (Refer to example 15).

Bias: It is an interesting way to store signed numbers without using any sign bit. It stores the number by adding a value in the exponent. For example, a 4 bit binary number can store values 0000 to 1111, i.e. values 0 to 15. A bias of 8 will allow values -8 to +7 to be stored in this range by adding the bias. In other words exponent value -8 will be coded as $(-8+8)$ 0, -7 will be coded as $(-7+8)$ 1, and so on till +7, which will be coded as $(+7+8)$ 15. But, why is biasing used for exponent? The basic reason here is that biased numbers simplify the floating point arithmetic. This is explained later with the help of an example.

Normalized: A fraction is called normalized if it starts with a bit value 1 and not with bit value 0. For example, the values .1001, .1111, .1000, .1010 are normalized, but the values .0100, .0001, .0010, .0011 are not normalized.

The following example explains the process of converting a decimal real number to a floating point number representation using IEEE-754 standard (32-bit representation). You may solve similar problems using double precision representation also, where only the size of exponent (and bias) and significand is different.

Example 15: Represent the number -29.25 using IEEE 754 (32 bit) representation as shown in Table 20.

Solution:

Step 1: Convert the number to binary

The number should first be converted to binary as follows:

Sign bit = 1 as number is negative

29 can be represented in 7 bits as 001 1101

.25 can be represented in 4 bits as .0100

Thus, 29.25 without sign is 001 1101 . 0100

Step 2: Normalize the number

Normalizing the number requires binary point to be moved before the most significant 1, it requires point to be shifted to left by 5 spaces. Thus, the normalized number now is: $0.1\ 1101\ 0100 \times 2^5$.

Step 3: Adjust the normalized number

It may be noted from Table 20(b), that in IEEE-754 representation, when the exponent is between 1 and 254, the first bit is assumed to be 1, therefore, the *Significand* whose size is 23 bits, actually represents 24 bit *Significand*. In addition, please note that as the number is assumed to be $\pm 1.M \times 2^{\text{exp}-127}$, therefore, the value to be represented ($0.1\ 1101\ 0100 \times 2^5$) must be adjusted to this format by shifting the binary point one place to the right and adjusting the exponent. Thus, the adjusted number is 1.11010100×2^4 .

Step 4: Compute the exponent using the bias

Finally add bias to the exponent value to obtain *exp* value of IEEE-754. In this case, *exp* = 4+127 (127 is the bias value)=131.

Step 5: Represent the final number

Represent the sign bit (S), *exp* in 8 bits and *Significand* in 23 bits, as follows:

S	exp of length 8 bits (value 131)	Significand of length 23 bits (value 1.11010100) Represented as ± 110 1010 0
1	1 0 0 0 0 0 1 1	1 1 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Example 16: A number using IEEE 754 (32 bit) is given below, what is the equivalent decimal value.

S	exp of length 8 bits	Significand of length 23 bits (M)
1	1 0 0 0 1 0 0 1	1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Solution:

The number is represented as: $\pm 1.M \times 2^{\text{exp}-127}$

The sign bit states it is a negative number

M is 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Exp is 1 0 0 0 1 0 0 1 = 137 in decimal.

The number is $-1.111100000000000000000000000000 \times 2^{137-127}$

$$= -1.111100000000000000000000000000 \times 2^{10}$$

$$= -11111000000.0000000000000000$$

$$= -11111000000$$

$$= -1984 \text{ in decimal}$$

In floating point numbers a term *precision* is very important. What is precision? The precision defines the correctness of representation. For example, suppose you just use 2 decimal digits in a fractional decimal numbers, then you can represent the numbers 0.10, 0.11, 0.99 etc precisely. The number 0.985 may be either truncated to 0.98 or rounded off to 0.99. This introduces an error in number, which is due to the fact that the size of *Significand* is limited. For scientific computations such errors may lead to failure. Therefore, IEEE-754 defines many different precision of numbers, few such popular precisions are single precision IEEE-754 number, which is a 32-bit representation, explained above; **IEEE-754 double precision number**, which is a 64

bit representation with 1 sign bit, 11 bit exponent and 52 bit *Significand*; and IEEE-754 quadruple precision number, which is a 128 bit representation with 1 sign bit, 15 bit exponent and 112 bit *Significand*. It may be noted that in programming languages you use data types *float* and *double*, which corresponds to the IEEE-754 single and double representation respectively.

Finally, what is the range of the numbers that can be represented using the IEEE-754 representation? As stated in Table 20, the minimum exponent value for a normalized number is 1 and maximum is 254. Therefore, the minimum (negative) number will be:

<i>S</i>	<i>exp</i> of length 8 bits	<i>Significand</i> of length 23 bits (<i>M</i>)
1	0 0 0 0 0 0 0 1	0 0

This will be equal to $\pm 1.M \times 2^{\text{exp}-127}$

$$= -1.000\ 0000\ 0000\ 0000\ 0000 \times 2^{1-127}$$

$$= -1 \times 2^{-126}$$

The maximum (positive) number will be:

<i>S</i>	<i>exp</i> of length 8 bits	<i>Significand</i> of length 23 bits (<i>M</i>)
1	1 1 1 1 1 1 1 0	1 1

This will be equal to $\pm 1.M \times 2^{\text{exp}-127}$

$$= +1.111\ 1111\ 1111\ 1111\ 1111 \times 2^{254-127}$$

$$= +(1.111\ 1111\ 1111\ 1111\ 1111\ 1111$$

$$\quad +0.000\ 0000\ 0000\ 0000\ 0000\ 0001$$

$$\quad -0.000\ 0000\ 0000\ 0000\ 0000\ 0001) \times 2^{127}$$

$$= +(10.000\ 0000\ 0000\ 0000\ 0000\ 0000$$

$$\quad -0.000\ 0000\ 0000\ 0000\ 0000\ 0001) \times 2^{127}$$

$$= +(2-1 \times 2^{-23}) \times 2^{127}$$

You may please note that IEEE-754 has a representation for *0 and infinite*.

Arithmetic Using Floating Point Numbers:

As you have noticed that addition and subtraction using 2's complement notation was direct, but addition and subtraction of floating point number requires several steps. These steps are explained with the help of the following example.

Example 17: Add the following floating point numbers

Equivalent Numbers		IEEE 754 32 bit representation		
Decimal	Binary	<i>S</i>	<i>exp</i>	<i>Significand (M)</i>
-7	$-1.11 \times 2^{129-127}$ = -1.11×2^2 = -111.0	1	1000 0001	110 0000 0000 0000 0000
+24	$+1.1 \times 2^{131-127}$ = $+1.1 \times 2^4$ = +11000.0	0	1000 0011	100 0000 0000 0000 0000

Solution:

Step 1: Find the difference in exponents of the numbers

$$1000\ 0011 - 1000\ 0001 = 0000\ 0010 = 2 \text{ in decimal}$$

Step 2: Align the *Significand* of the smaller number by denormalizing it

Shift the smaller number to right by the difference of exponents as shown:

	The value of $1.M$
<i>Significand of First Number (Smaller)</i>	1.110 0000 0000 0000 0000
Shift it to right twice (denormalized)	0.011 1000 0000 0000 0000

Step 3: Check the sign of the two numbers, if same add else subtract smaller number from bigger number.

The signs are different; therefore, subtract smaller number from larger number

	The value of $1.M$
Significand of Second Number (Larger)	1.100 0000 0000 0000 0000
Denormalized first number (smaller)	0.011 1000 0000 0000 0000
Result of Subtraction	1.000 1000 0000 0000 0000

Step 4: Select the sign and exponent of the bigger number as sign and exponent of the result and Normalize the *Significand* by adjusting the exponent

The result is shown below. Please note that in this case, there is no need to normalize the result as it is already normalized.

Result of addition operation (verification)		IEEE 754 32 bit representation		
Decimal	Binary	S	exp	$1.M$
+17	$+1.0001 \times 2^{131-127}$ = $+1.0001 \times 2^4$ = +10001.0	0	1000 0011	1.000 1000 0000 0000 0000

Likewise, subtraction operation can be performed.

Multiplication and division operations in floating point number require multiplication or division of significands as well as addition or subtraction of exponents. In addition, these operations may require normalizing the result. The following example shows the multiplication of two floating point numbers.

Example 18: Multiply the following floating point numbers

Equivalent Numbers		IEEE 754 32 bit representation		
Decimal	Binary	S	exp	Significand (M)
-7	- 111.0	1	1000 0001	110 0000 0000 0000 0000
+24	+11000.0	0	1000 0011	100 0000 0000 0000 0000

Solution:

Step 1: Multiply the *Significand* values of the two numbers and truncate to 23 bits (plus one assumed bit)

	The value of $1.M$
Significand of First Number	1.110 0000 0000 0000 0000
Significand of Second Number	1.100 1000 0000 0000 0000
Significand after multiplication	10.101 0000 0000 0000 0000

Step 2: If multiplication, add the exponents and subtract the bias as both the numbers have biased exponents; if division, subtract the exponent of divisor from the exponent of dividend and add the bias as it get canceled in subtraction.

Exponent of First Number	1000 0001
Exponent of Second Number	1000 0011
Multiplication operation, so add and subtract bias -127	1 0000 0100 -0111 1111
The new exponent	1000 0101

Step 3: Check the sign of the two numbers, if same result has + sign else - sign

The signs are different; therefore, result has negative sign. Also, normalize the *Significand* of the result

Result of Multiplication	IEEE 754 32 bit representation
--------------------------	--------------------------------

operation		S	exp	I.M
Decimal	Binary	S	exp	I.M
Normalize the result		1	1000 0101	10.10 1000 0000 0000 0000
Normalized result		1	1000 0110	1.010 1000 0000 0000 0000
-168	$-1.0101 \times 2^{134-127}$ = -1.0101×2^7 = -10101000.0	1	1000 0110	1.010 1000 0000 0000 0000

Likewise, division operation can be performed. You may refer to further readings for finding more details on floating point numbers and arithmetic.

2.7 ERROR DETECTION AND CORRECTION CODES

In the previous sections you have gone through various binary codes and representations. A computer works on these binary numbers and during the operations of the computer data is transferred from a source to one or more destinations. During this process of transmission of data, there is a possibility of transmission errors. The purpose of error detection and correction codes is to identify those data transmission errors and correct the data, as far as possible. As the data in computer consists of binary digits, therefore, an error in a bit can result in change of its value from 0 to 1 or vice versa. This section explains one error detection code called Parity and one error detection and correction code called Hamming Error correction code.

Parity bit: The purpose of a parity bit is to detect an error in a group of bits. But how does it perform the task of checking error? It is explained with the help of following process:

Steps	Source Side	Destination Side
1	A parity bit is generated at source, which ensures that: Number of 1's in sources data and source parity bit is odd (called Odd parity) OR Number of 1's in sources data and source parity bit is even (called Even parity)	
2	The source data and source parity bits are sent to the designation.	
3		The source data and source parity bits are received at the designation and a destination parity bit is generated using only the data received (not source parity) by using the same process i.e. Even or Odd parity used at source.
4		Source parity bit and destination parity bit are compared. If they are same, then no error in data is detected, else either the data or parity bit is in error, which is reported.

Table 21: Error detection using parity bit

Example 19 explains the process as given above.

Example 19: 7-bit data 010 1001 is sent from a source, such as CPU register, to a destination, such as RAM. The data is received at the destination as 010 1000 having error in one bit. How does this error be detected by parity bit?

Steps	The Process
Step 1: At Source	Data to be sent: 010 1001 Odd Parity bit is computed as: The data has 3 bits having value 1. So odd parity bit =0
Step 2: At Source	The source parity + source data is sent as: 0 010 1001
Step 3: At Destination	As per the statement of the example the data is received as: 0 010 1000 Source Parity = 0 is received correctly as error is in one bit only Destination parity is computed on data received, which is 010 1000. It has 2 bits as 1, therefore, Odd parity bit at destination=1
Step 4: At Destination	Source parity bit (0) ≠ Destination parity bit(1) ERROR in data

It may be noted that parity bit can detect errors in case 1 bit is in error. In case 2 bits are in error, then it will fail to detect the error.

Hamming Error-Correcting Code: The Hamming code was conceptualized by Richard Hamming at Bell Laboratories. This code is used to identify and correct the error in 1 bit. Thus, unlike parity bit, which just identifies the existence of error, this code also identifies the bit that is in error. The idea of Hamming's code is to divide the data bits into a number of groups; and using the parity bit to identify, which groups are in error; and based on the groups in error, identify the bit which has caused the error. Thus, the grouping process has to be very special, which is explained below:

How to Group data bits? Before grouping, you may assume the placement of data and parity bits using the following considerations.

A bit position that is exact power of 2 will be used for storing parity bit. For example, $2^0=1$, that is 1st bit position will be used to store parity bit, likewise $2^1=2$, $2^2=4$, and $2^3=8$, i.e. 2nd, 4th and 8th bit positions will also be used to store parity bit. Thus, you have now 7 bit data and 4 parity bits, so a total of 11 bit positions. (*p* indicates parity bit and *d* indicates data bit)

Bit Position	12	11	10	9	8	7	6	5	4	3	2	1
Stores	<i>d</i> 8	<i>d</i> 7	<i>d</i> 6	<i>d</i> 5	<i>p</i>4	<i>d</i> 4	<i>d</i> 3	<i>d</i> 2	<i>p</i>3	<i>d</i> 1	<i>p</i>2	<i>p</i>1
<i>For grouping the data bit number is used to identify the parity bit to which data should be member of</i>												
<i>Bit position 12 (8+4) contains (d8)</i>					8				4		-	-
<i>Bit position 11 (8+2+1) contains (d7)</i>					8				-		2	1
<i>Bit position 10(8+2) contains (d6)</i>					8				-		2	-
<i>Bit position 9(8+1) contains (d5)</i>					8				-		-	1
<i>Bit position 8 contains (p4)</i>					<i>p</i> 4							
<i>Bit position 7(4+2+1) contains (d4)</i>					-				4		2	1
<i>Bit position 6(4+2)</i>					-				4		2	-

<i>contains (d3)</i>									
<i>Bit position 5(4+1) contains (d2)</i>				-			4	-	1
<i>Bit position 4 contains (p3)</i>							<i>p3</i>		
<i>Bit position 3(2+1) contains (d1)</i>				-			-	2	1
<i>Bit position 2 contains (p2)</i>								<i>p2</i>	
<i>Bit position 1 contains (p1)</i>									<i>p1</i>

Table 22: Placement of data and parity bits for Hamming's error detection and correction code

Groups for parity bits: The groups are made for each on the basis of bit positions, on the basis of above Table. A bit position, which includes a parity bit position is included in the group of that parity bit. For example, the bit at bit position 12 will be included in group of parity bit p_4 and p_3 ; similarly, bit position 7 will be included in group of parity bit p_3 , p_2 and p_1 . But why these grouping? You may please note that each data bit is part of unique combination of groups, so if it is in error, it will cause errors in all those groups to which it is a part of. Thus, by identifying all the groups, which has parity mismatch, will identify the bit which is in error. The following table shows these groups for 8 bit data.

Group for Parity bits	Bit positions and data bit
p_4	<i>Bit position 12 data bit d8, Bit position 11 data bit d7, Bit position 10 data bit d6 and Bit position 9 data bit d5</i>
p_3	<i>Bit position 12 data bit d8, Bit position 7 data bit d4, Bit position 6 data bit d3 and Bit position 5 data bit d2</i>
p_2	<i>Bit position 11 data bit d7, Bit position 10 data bit d6, Bit position 7 data bit d4, Bit position 6 data bit d3 and Bit position 3 data bit d1</i>
p_1	<i>Bit position 11 data bit d7, Bit position 9 data bit d5, Bit position 7 data bit d4, Bit position 5 data bit d2 and Bit position 3 data bit d1</i>

Therefore, the parity bits will be generated using the following data bits:

Parity bit	Compute Odd parity of Data bits
p_4	d_8, d_7, d_6 and d_5
p_3	d_8, d_4, d_3 and d_2
p_2	d_7, d_6, d_4, d_3 and d_1
p_1	d_7, d_5, d_4, d_2 and d_1

So, how the data bit in error be recognised? It is illustrated with the help of following example

Example 20: 8-bit data 1010 1001 is sent from a source to a destination. The data is received at the destination as 1000 1001 having error in only one bit. How does this error be detected and corrected by Hamming's error detection and correction code?

Solution:

Step 1: Place the bits as shown in Table 22 and generate parity bits at the source, for example, the odd parity bit p_4 is computed using d_8, d_7, d_6 and d_5 (shown as shaded cells in the following table). Their values are 1, 0, 1, 0 as shown in the table, as there are only two bits containing 1, therefore, the odd parity value for p_4 is 1. Likewise compute the other parity bits as shown in Table 23.

Step 2: Data and the associated parity bits in the sequence as shown below are sent to the destination, where once again parity bits are computed for the received data.

Step 3: Compare the source parity bits and destination parity bits as shown in Table 23. Please note when two parity bit match, a 0 is put in the compare word else a 1 is put. The magnitude of comparison word, indicates the bit position that is in error.

Step 4: If there is an error, then the data at bit position that is in error is complemented.

Step 5: The data is used at the destination after omitting the parity bits.

Bit Position	12	11	10	9	8	7	6	5	4	3	2	1
Stores	d_8	d_7	d_6	d_5	p_4	d_4	d_3	d_2	p_3	d_1	p_2	p_1
Data Bits	1	0	1	0		1	0	0		1		
Compute Odd parity bit p_4 using d_8, d_7, d_6 and d_5					1							
Compute Odd parity bit p_3 using d_8, d_4, d_3 and d_2									1			
Compute Odd parity bit p_2 using d_7, d_6, d_4, d_3 and d_1										0		
Compute Odd parity bit p_1 using d_7, d_5, d_4, d_2 and d_1											1	
Data and Parity bits at Source	1	0	1	0	1	1	0	0	1	1	0	1
<i>Data is sent to the destination, where data is received with 1 bit in error (given), therefore all the source parity bits are received without any error</i>												
Data received at destination including parity bits	1	0	0	0	1	1	0	0	1	1	0	1
<i>Step 2: Compute the parity bits using the data received at the destination</i>												
Data Bits Received	1	0	0	0		1	0	0		1		
Compute Odd parity bit p_4 using d_8, d_7, d_6 and d_5					0							
Compute Odd parity bit p_3 using d_8, d_4, d_3 and d_2									1			
Compute Odd parity bit p_2 using d_7, d_6, d_4, d_3 and d_1										1		
Compute Odd parity bit p_1 using d_7, d_5, d_4, d_2 and d_1											1	
<i>Step 3: Compare the source parity bits and destination parity bits.</i>												
Source Parity bits					1				1		0	1
Destination Parity bits					0				1		1	1
Parity Comparison word (0 if source and destination parity match else 1)					1				0		1	0
The comparison word is 1010 = 10 in decimal, i.e. bit position 10 is in error. The error in this bit can be corrected by complementing the bit position 10.												
Corrected Data	1	0	1	0		1	0	0		1		

Table 23: Example of Hamming's error detection and correction code

It may be noted in Table 23 that the value of comparison word 0000 would mean that there is no error in transmission of data. In addition, the values 1000, 0100, 0010 and 0001 would mean that one bit error has occurred in the transmission of source parity

bits p_4, p_3, p_2 and p_1 respectively. Thus, no change would be needed in the received data bits at the destination in such cases.

It may please be noted that Hamming's code presented in this section can detect and correct errors in a single bit ONLY. It will not work, in case two or more bits are in error. One final question is about the size of the code needed to correct single bit error. The size will be dependent on the size of data. A simple rule is that the size of code and the data should be less than the possible bit positions that can be flagged by the comparison word. If the data to be transmitted is of size D bits and P is the number of parity bits needed for the given Hamming's code, then size of the code is the smallest value of P , which satisfies that following equation:

$$D + P < 2^P$$

For example, for a $D=4$ bits, the value of P would be 3 as:

$$4 + 3 < 2^3 \text{ as } 7 < 8$$

and for a $D=8$ bits, the value of P would be 4 as:

$$8 + 4 < 2^4 \text{ as } 12 < 16$$

Check Your Progress 3

- 1) Represent the following numbers using the IEEE-754 32bit standard:
 - i) 39.125
 - ii) -0.000011000₂
- 2) Compute the Odd and Even parity bits for the following data:
 - i) 0111110
 - ii) 0110000
 - iii) 1110111
 - iv) 1001100
.....
.....
.....
.....
- 3) A 4 bit data 1011 is received at the destination as 1111, assuming single bit is in error, illustrate how Hamming's single error correction code will detect and correct the error
.....
.....
.....
.....

2.7 SUMMARY

This Unit has introduced you to the basic aspects of data representation. It introduces the character representing including ASCII and Unicode. In addition, the Unit explains number conversion and fixed point representation of binary number. The Unit also highlights the arithmetic operations. This was followed by a detailed discussion on the floating point numbers. Though only IEEE 754 32-bit single precision numbers are explained, however, the logic discussed is applicable to double precision numbers too. The Unit finally introduces you to error detection code -parity bit and error detection and correction code. You must practice the data conversions and these codes as they would be useful, when you deal with binary numbers.

You should refer to the further readings for more detailed information on these topics. You are advised to take the help of further readings, Massive Open Online Courses (MOOCs), and other online resources as Computer Science is a dynamic area.

2.8 SOLUTIONS/ANSWERS

Check Your Progress 1

- 1) i) 11100.01101_2 to Octal and Hexadecimal

Binary Number	0 1 1 1 0 0 . 0 1 1 0 1 0
Grouping Directions	←—————→
Grouped (- replaced by 0)	0 1 1 1 0 0 . 0 1 1 0 1 0
Binary place values	4 2 1 4 2 1 . 4 2 1 4 2 1
Equivalent Octal Digit	0+2+1=3 4+0+0=4 . 0+2+1=3 0+2+0=2
Octal Number	3 4 . 3 2

Binary Number	0 0 0 1 1 1 0 0 . 0 1 1 0 1 0 0 0
Grouping Directions	←—————→
Grouped	0 0 0 1 1 1 0 . 0 1 1 1 0 0
Binary place values	8 4 2 1 8 4 2 1 . 8 4 2 1 8 4 2 1
Equivalent Hexadecimal Digit	0+0+0+1=1 8+4+0+0=C . 0+4+2+0=6 8+0+0+0=8
Hexadecimal Number	1 C . 6 8

- ii) 1101101010_2 to Octal and Hexadecimal

Binary Number	0 0 1 1 0 1 1 0 1 0 1 0 1 0
Grouping Directions	←—————→
Grouped (- replaced by 0)	0 0 1 1 0 1 1 0 1 0 1 0 1 0
Binary place values	4 2 1 4 2 1 4 2 1 4 2 1 4 2 1
Equivalent Octal Digit	0+0+1=1 4+0+1=5 4+0+1=5 0+2+0=2
Octal Number	1 5 5 2

Binary Number	0 0 1 1 0 1 1 0 1 0 1 0 1 0
Grouping Directions	←—————→
Grouped	0 0 1 1 0 1 1 0 1 0 1 0 1 0
Binary place values	8 4 2 1 8 4 2 1 8 4 2 1 8 4 2 1
Equivalent Hexadecimal Digit	0+0+2+1=3 0+4+2+0=6 8+0+2+0=A
Hexadecimal Number	3 6 A

- 2) i) 119_{10} to binary

The place value	$2^6 = 64$	$2^5 = 32$	$2^4 = 16$	$2^3 = 8$	$2^2 = 4$	$2^1 = 2$	$2^0 = 1$
N = 119	119 - 64 = 55; 55 - 32 = 23; 23 - 16 = 7; 7 - 4 = 3; 3 - 2 = 1; 1 - 1 = 0						
Equivalent Binary	1	1	1	0	1	1	1

- ii) 19.125_{10}

The place	2^4	2^3	2^2	2^1	2^0	.	2^{-1}	2^{-2}	2^{-3}
-----------	-------	-------	-------	-------	-------	---	----------	----------	----------

value	=16	=8	=4	=2	=1		=0.5	=0.25	=0.125
N = 19.125		19-16=3; 3-2=1; 1-1=0;		and $2^3=8$	=0.125				
Equivalent Binary	1	0	0	1	1	.	0	0	1

iii) 325_{10}

The place value	2^8 =256	2^7 =128	2^6 =64	2^5 =32	2^4 =16	2^3 =8	2^2 =4	2^1 =2	2^0 =1
N = 325		325-256=69	69-64=5	5-4=1; 1-1=0					
Equivalent Binary	1	0	1	0	0	0	1	0	1

3 i) 119_{10}

The place value	2^6 =64	2^5 =32	2^4 =16	2^3 =8	2^2 =4	2^1 =2	2^0 =1
Equivalent Binary	1	1	1	0	1	1	1
Equivalent Octal	1		6			7	
Equivalent Hexadecimal			7				7

ii) 19.125_{10}

The place value	2^4 =16	2^3 =8	2^2 =4	2^1 =2	2^0 =1	.	2^{-1} =0.5	2^{-2} =0.25	2^{-3} =0.125
Equivalent Binary	1	0	0	1	1	.	0	0	1
Equivalent Octal	2		3			.		1	
Equivalent Hexadecimal	1		3			.		0010=2	

iii) 325_{10}

The place value	2^8 =256	2^7 =128	2^6 =64	2^5 =32	2^4 =16	2^3 =8	2^2 =4	2^1 =2	2^0 =1
Equivalent Binary	1	0	1	0	0	0	1	0	1
Equivalent Octal		5			0			5	
Equivalent Hexadecimal	1			4				5	

Check Your Progress 2

Check Your Progress 2

1) i) -23456 to BCD

Sign Digit	2	3	4	5	6
	1101	0010	0011	0100	0101

ii) 17.89

Sign Digit	1	7	.	8	9
	1100	0001	0111	.	1000

iii) 299

Sign Digit	2	9	9
	1100	0010	1001

2)

Deci	The Number	Signed 1's Complement	Signed 2's Complement

mal	1	1	1	0	0	0	1	0	0	0	0	1	1	1	0	1	0	0	0	1	1	1	1	0
-30	0	1	1	1	1	1	0	1	0	0	0	0	0	0	1	1	0	0	0	0	0	1	0	0
+126	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0

3) i) +56 and -56

Number	Carry out (9th bit)	Signed 2's Complement Notation								
Carry for addition	1	1	1	1	1				-	
+56		0	0	1	1	1	0	0	0	
-56		1	1	0	0	1	0	0	0	
Addition of bits given above		1+0+1 =10	1+0+1 =10	1+1+0 =10	1+1+0 =10	1+1 =10	0+0 =0	0+0 =0	0+0 =0	
Result	+	0	0	0	0	0	0	0	0	

There is a carry in to the sign bit (1) and there is a carry out of the sign bit (1). Therefore, as per Table 14, there is NO overflow and the result is correct and equal to 0. Discard the carry out bit (the 9th bit). Verify the result yourself.

ii) +65 and -75

Number	Carry out (9th bit)	Signed 2's Complement Notation								
Carry for addition								1	-	
+65		0	1	0	0	0	0	0	1	
-75		1	0	1	1	0	1	0	1	
Addition of bits given above		0+1 =1	1+0 =1	0+1 =1	0+1 =1	0+0 =0	0+1 =1	1+0+0 =1	1+1 =10	
Result	1	1	1	1	0	1	1	1	0	

There is No carry in to the sign bit and there is No carry out of the sign bit. Therefore, as per Table 14, there is NO overflow and the result is correct and equal to -10. Verify the result yourself.

iii) +121 and +8

Number	Carry out (9th bit)	Signed 2's Complement Notation								
Carry for addition		1	1	1	1				-	
+121		0	1	1	1	1	0	0	1	
+8		0	0	0	0	1	0	0	0	
Addition of bits given above		1+0+0 =1	1+1+0 =10	1+1+0 =10	1+1+0 =10	1+1 =10	0+0 =0	0+0 =0	0+1 =1	
Result	1	0	0	0	0	0	0	0	1	

There is a carry in to the sign bit (1) and there is NO carry out of the sign bit. Therefore, as per Table 14, there is OVERFLOW and the result is incorrect.

Check Your Progress 3

1) i) 39.125

Step 1: Convert the number to binary

Sign bit = 0 as number is positive

39 can be represented in 7 bits as 010 0111

.125 can be represented in 4 bits as .0010

Thus, 39.125 without sign is 0100111.0010

Step 2: Normalize the number

Normalizing the number requires binary point to be moved before the most significant 1, it requires point to be shifted to left by 6 spaces. Thus, the normalized number now is: 0.1001110010×2^6 .

Step 3: Adjust the normalized number

The number is assumed to be $\pm 1.M \times 2^{exp-127}$, therefore, the value to be represented (0.1001110010×2^6) must be adjusted. The adjusted number is 1.001110010×2^5 .

Step 4: Compute the exponent using the bias

Add bias to the exponent value.

$exp = 5+127$ (127 is the bias value)=132.

Step 5: Represent the final number

Represent the sign bit (S), exp in 8 bits and Significand in 23 bits, as follows:

S	exp of length 8 bits (value 132)	Significand of length 23 bits (value 1.001110010) Represented as +.001110010
0	1000 0100	001 1100 1000 0000 0000 0000

ii) -0.000011000_2

Sign bit = 1 as number is negative

The number is of the format $\pm 1.M \times 2^{exp-127}$,
 1.1000×2^{-5}

$$exp = -5+127=122$$

S	exp of length 8 bits (value 122)	Significand of length 23 bits (value 1.1000) Represented as -.1000
0	0111 1010	100 0000 0000 0000 0000 0000

2)

The Number	Even Parity	Odd Parity
0111110	1	0
0110000	0	1
1110111	0	1
1001100	1	0

3) The size of data (D)=4 bits, the value of P would be 3 as:

$$4 + 3 < 2^3 \text{ as } 7 < 8$$

The bit positions for this code would be:

Bit Position	7	6	5	4	3	2	1
Stores	<i>d4</i>	<i>d3</i>	<i>d2</i>	<i>p3</i>	<i>d1</i>	<i>p2</i>	<i>p1</i>
<i>For grouping the data bit number is used to identify the parity bit to which data should be member of</i>							
<i>Bit position 7(4+2+1) contains (d4)</i>				4		2	1
<i>Bit position 6(4+2) contains (d3)</i>				4		2	-
<i>Bit position 5(4+1) contains (d2)</i>				4		-	1
<i>Bit position 4 contains (p3)</i>				<i>p3</i>			
<i>Bit position 3(2+1) contains (d1)</i>				-		2	1
<i>Bit position 2 contains (p2)</i>						<i>p2</i>	
<i>Bit position 1 contains (p1)</i>							<i>p1</i>

Parity bit	Compute Odd parity of Data bits
<i>p3</i>	<i>d4, d3 and d2</i>
<i>p2</i>	<i>d4, d3 and d1</i>
<i>p1</i>	<i>d4, d2 and d1</i>

Bit Position	7	6	5	4	3	2	1
Stores	<i>d4</i>	<i>d3</i>	<i>d2</i>	<i>p3</i>	<i>d1</i>	<i>p2</i>	<i>p1</i>
<i>Data Bits</i>	1	0	1		1		
<i>Compute Odd parity bit p3 using d4, d3 and d2</i>				1			
<i>Compute Odd parity bit p2 using d4, d3 and d1</i>						1	
<i>Compute Odd parity bit p1 using d4, d2 and d1</i>							0
<i>Data and Parity bits at Source</i>	1	0	1	1	1	1	0
<i>Data received at destination including parity bits</i>	1	1	1	1	1	1	0
<i>Data Bits Received</i>	1	1	1		1		
<i>Compute Odd parity bit p3 using d4, d3 and d2</i>				0			
<i>Compute Odd parity bit p2 using d4, d3 and d1</i>						0	
<i>Compute Odd parity bit p1 using d4, d2 and d1</i>							0
<i>Source Parity bits</i>				1		1	0
<i>Destination Parity bits</i>				0		0	0
<i>Parity Comparison word (0 if source and destination parity match else 1)</i>				1		1	0
<i>Location 6 is in error, which is decimal equivalent of 110 the comparison word. So Corrected Data</i>	1	0	1		1		

UNIT 3 LOGIC CIRCUITS - AN INTRODUCTION

Structure	Page Nos.
3.0 Introduction	
3.1 Objectives	
3.2 Logic Gates	
3.3 Boolean Algebra	
3.4 Logic Circuits	
3.5 Combinational Circuits	
3.5.1 Canonical and Standard Forms of an Boolean expression.	
3.5.2 Minimization of Gates	
3.6 Design of Combinational Circuits	
3.7 Examples of Logic Combinational Circuits	
3.7.1 Adders	
3.7.2 Decoders	
3.7.3 Multiplexer	
3.7.4 Encoder	
3.7.5 Programmable Logic Array	
3.7.6 Read Only Memory ROM	
3.8 Summary	
3.9 Solutions/ Answers	

3.0 INTRODUCTION

In the previous units, we have discussed the basic configuration of computer system, simple instruction execution, data representation and different computer organisations. In addition, ISA and micro-architecture was discussed in unit 1 of this block. It may be noted that, as stated in the unit1, gates and logic circuits are the building blocks of a computer system. This unit introduces you to some of the basic components of computer system that are essential for learning the logic of binary computing devices. In this unit, you will be introduced to the concepts of logic gates, binary adders, logic circuits and combinational circuits. These circuits form the backbone of any computer system and knowing them will be useful in lower level programming.

3.1 OBJECTIVES

After going through this unit you will be able to:

- explain the basic functions of logic gates;
- describe role of Boolean algebra in digital circuits;
- perform the minimization the number of gates for a Boolean expression;
- identify and explain the basic circuits in a computer system
- design simple combinational circuits.

3.2 LOGIC GATES

A computer system is a binary device that uses electronic signals to perform basic computation on the digital data, which is also stored electronically. The

basis of such computation, called digital logic, are the electronic circuits fabricated on the semi-conductor chips that are used to formulate a set of operations. These basic sets of operations are then used to create complex circuitry, which is able to perform arithmetic, logical and control operations in a computer system. Thus, the simplest form of binary logic is: how a set of inputs can be used to create a typical output sequence, which is achieved using electronic gates.

A logic gate is an electronic circuit made of transistors, which produces a characteristic output signal for a typical input. In general, the input accepts one to several input values and produces a specific output. This output can be 0 or 1. Logic gates are used for implementing basic Boolean operations, which is explained in the subsequent sections. A logic gate is represented using a graphics symbol and performs a simple binary function, which can be represented with the help of a truth table. Figure 3.1 shows the basic logic gates. Please note that in Figure 3.1, the character *I* represent input values and *F* represent output values.

Gate	Graphical Symbol	Truth Table	Description															
NOT		<table> <thead> <tr> <th>I</th> <th>F</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>1</td> </tr> <tr> <td>1</td> <td>0</td> </tr> </tbody> </table>	I	F	0	1	1	0	It simply inverts the input value, i.e. input 0 will be converted to 1 and vice-versa.									
I	F																	
0	1																	
1	0																	
OR		<table> <thead> <tr> <th>I₁</th> <th>I₂</th> <th>F</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>0</td> <td>0</td> </tr> <tr> <td>0</td> <td>1</td> <td>1</td> </tr> <tr> <td>1</td> <td>0</td> <td>1</td> </tr> <tr> <td>1</td> <td>1</td> <td>1</td> </tr> </tbody> </table>	I ₁	I ₂	F	0	0	0	0	1	1	1	0	1	1	1	1	This gates output a value 1, if at least one of its input is 1.
I ₁	I ₂	F																
0	0	0																
0	1	1																
1	0	1																
1	1	1																
AND		<table> <thead> <tr> <th>I₁</th> <th>I₂</th> <th>F</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>0</td> <td>0</td> </tr> <tr> <td>0</td> <td>1</td> <td>0</td> </tr> <tr> <td>1</td> <td>0</td> <td>0</td> </tr> <tr> <td>1</td> <td>1</td> <td>1</td> </tr> </tbody> </table>	I ₁	I ₂	F	0	0	0	0	1	0	1	0	0	1	1	1	This gate output is 1, if both the input values are 1 else output is 0.
I ₁	I ₂	F																
0	0	0																
0	1	0																
1	0	0																
1	1	1																
NAND		<table> <thead> <tr> <th>I₁</th> <th>I₂</th> <th>F</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>0</td> <td>1</td> </tr> <tr> <td>0</td> <td>1</td> <td>1</td> </tr> <tr> <td>1</td> <td>0</td> <td>1</td> </tr> <tr> <td>1</td> <td>1</td> <td>0</td> </tr> </tbody> </table>	I ₁	I ₂	F	0	0	1	0	1	1	1	0	1	1	1	0	This is NOT AND, so where ever AND gate produce 0, this gate outputs 1.
I ₁	I ₂	F																
0	0	1																
0	1	1																
1	0	1																
1	1	0																
NOR		<table> <thead> <tr> <th>I₁</th> <th>I₂</th> <th>F</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>0</td> <td>1</td> </tr> <tr> <td>0</td> <td>1</td> <td>0</td> </tr> <tr> <td>1</td> <td>0</td> <td>0</td> </tr> <tr> <td>1</td> <td>1</td> <td>0</td> </tr> </tbody> </table>	I ₁	I ₂	F	0	0	1	0	1	0	1	0	0	1	1	0	This is NOT OR.
I ₁	I ₂	F																
0	0	1																
0	1	0																
1	0	0																
1	1	0																
XOR		<table> <thead> <tr> <th>I₁</th> <th>I₂</th> <th>F</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>0</td> <td>0</td> </tr> <tr> <td>0</td> <td>1</td> <td>1</td> </tr> <tr> <td>1</td> <td>0</td> <td>1</td> </tr> <tr> <td>1</td> <td>1</td> <td>0</td> </tr> </tbody> </table>	I ₁	I ₂	F	0	0	0	0	1	1	1	0	1	1	1	0	Exclusive OR produces output 1 when the two input are dissimilar.
I ₁	I ₂	F																
0	0	0																
0	1	1																
1	0	1																
1	1	0																

Figure 3.1: Logic Gates

In the next few sections, we explain how these simple logic gates can be used to construct logic circuits. The next section explains the mathematics of logic circuits.

3.3 BOOLEAN ALGEBRA

Boolean algebra was designed by George Boole in the 19th century. It presents mathematical foundation for performing various functions on binary variables. Please recall that binary variables can have only two values 0 or 1. The value 0 by convention is taken as False and 1 as True. Please also refer to Figure 3.1, which shows the truth table for various gates. These truth tables can also be represented using the Boolean function. Figure 3.2 shows the Boolean algebraic representation of logic gates of Figure 3.1.

Gate	Boolean Representation	Explanation
NOT	$F = I'$	The symbol “’” in Boolean expression represents negation operator. Thus, output F is complement or negation of the value of I.
OR	$F = I_1 + I_2$	The OR is represented by Boolean operator ‘+’, which represents that the value of F be zero only if both I_1 and I_2 are zero else 1.
AND	$F = I_1 \cdot I_2$	The Boolean operator ‘.’ represents AND operation. The value of F be 1 if both I_1 and I_2 are 1, else F will be 0.
NAND	$F = (I_1 \cdot I_2)'$	NOT of AND
NOR	$F = (I_1 + I_2)'$	NOT of OR
XOR	$F = I_1 \oplus I_2$	\oplus is an exclusive – OR operator.

Figure 3.2: Gates and related Boolean algebraic expression.

The Boolean algebra is very useful for mathematically representing a binary operation. For example, addition of two binary digits can be represented in truth table form as:

I_1	I_2	Carry (C)	Sum (S)
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

∴ It can be represented using two Boolean functions, one for each output, viz. Carry and Sum, as:

$$C = I_1 \cdot I_2 \quad \text{and}$$

$$S = I_1 \oplus I_2 \quad (\text{Please refer to Fig. 3.1 & Fig 3.2})$$

The Boolean algebra is used to simplify logic circuits that are made of logic gates. However, before we demonstrate this process of simplification, first you may go through the basic rules of Boolean algebra. Figure 3.3 shows these rules. Please note that some of the rules are shown with proof using truth table. You can make truth table yourself for the cases for which the proof is not shown.

(i)	Input	Identities			
		$I + 0 = I$	$I + I = I$	$I \cdot 0 = 0$	$I \cdot I = I$
	0	$0 + 0 = 0$	$0 + 1 = 1$	$0 \cdot 0 = 0$	$0 \cdot 1 = 0$
	1	$1 + 0 = 1$	$1 + 1 = 1$	$1 \cdot 0 = 0$	$1 \cdot 1 = 1$

(ii)	Input	Identities			
		$I + I = I$	$I + I' = I$	$I \cdot I = I$	$I \cdot I' = 0$
	0	$0 + 0 = 0$	$0 + 1 = 1$	$0 \cdot 0 = 0$	$0 \cdot 1 = 0$
	1	$1 + 0 = 1$	$1 + 1 = 1$	$1 \cdot 1 = 1$	$1 \cdot 0 = 0$

(Please note $0' = 1$ and $1' = 0$)

(iii) The rules (given without proof)

$$I_1 + I_2 = I_2 + I_1 ;$$

$$I_1 \cdot I_2 = I_2 \cdot I_1 ;$$

$$I_1 + (I_2 + I_3) = (I_1 + I_2) + I_3 ;$$

$$I_1 \cdot (I_2 \cdot I_3) = (I_1 \cdot I_2) \cdot I_3$$

(iv) The rules (given without proof)

$$I_1 \cdot (I_2 + I_3) = (I_1 \cdot I_2 + I_1 \cdot I_3) ;$$

$$I_1 + I_2 \cdot I_3 = (I_1 + I_2) \cdot (I_1 + I_3)$$

(v) Demorgan's Laws:

$$(I_1 + I_2)' = I_1' \cdot I_2'$$

$$(I_1 \cdot I_2)' = I_1' + I_2'$$

(Very important laws for algebraic simplification.)

(vi) Complement of complement of a number is the Number itself

I	I'	$(I)'$	so $(I)'' = I$
0	1	0	
1	0	1	

Figure 3.3: The Rules of Boolean algebra

All the rules and identities as given in Figure 3.3 can be used for simplification of Boolean function. This is explained with the help of following example.

Example: Simplify the Boolean function:

$$F = ((A' + B')' + (A \cdot B))'$$

Solution:

$$\begin{aligned}
 F &= ((A' + B')' + (A \cdot B))' \\
 F &= ((A' + B')')' \cdot ((A \cdot B))' && \text{(Using Demorgan's Law)} \\
 &= (A' \cdot B') \cdot (A \cdot B) && \text{Using Rule (vi)} \\
 &= (A \cdot B) \cdot (A' \cdot B') && \text{Reversing the terms - Rule (iii)} \\
 &= ((A \cdot A) \cdot B) + (A \cdot (B \cdot B')) && \text{Using Rule (vi) taking } (A \cdot B) \text{ as } I_1 \\
 &= (A \cdot B) + (A \cdot 0) && \text{Using Rule (iii)} \\
 &= 0 \cdot B + A \cdot 0 && \text{Using Rule (ii)} \\
 &= 0 + 0 && \text{Using Rule (i)} \\
 &= 0
 \end{aligned}$$

$$F = 0$$

You can check the above using the following Truth Table

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

The Boolean algebra is very useful in simplification of logic circuits. It is explained in the next section.

3.4 LOGIC CIRCUITS

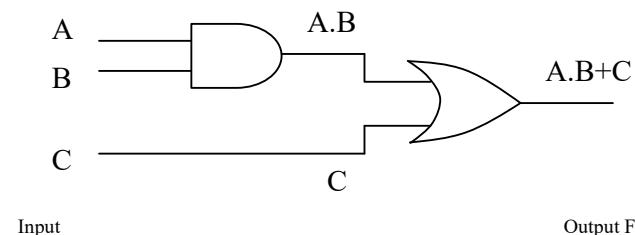
A logic circuit performs the basic operation on binary data. The operation of a logic circuit can be represented using a Boolean function, or in other words a Boolean function can be implemented as a logic circuit. The three basic gates that can be used for drawing logic circuits are - AND, OR and NOT gates. Consider, for example, the Boolean function:-

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

The relationship between this function and its binary variables A, B, C can be represented in a truth table as shown in Figure 3.4(a). Figure 3.4(b) shows the corresponding logic circuit.

Inputs			Output
A	B	C	$F = A.B+C$
0	0	0	$F = 0 . 0 + 0 = 0$
0	0	1	$F = 0 . 0 + 1 = 1$
0	1	0	$F = 0 . 1 + 0 = 0$
0	1	1	$F = 0 . 1 + 1 = 1$
1	0	0	$F = 1 . 0 + 0 = 0$
1	0	1	$F = 1 . 0 + 1 = 1$
1	1	0	$F = 1 . 1 + 0 = 1$
1	1	1	$F = 1 . 1 + 1 = 1$

(a) Truth Table



(b) Logic Circuit

Figure 3.4: Truth table & logic diagram for $F = A . B + C$

While fabricating these logic circuits, it is expected that fewer gate types are used; however, these gate types should be able to create all kinds of circuits. Therefore, functionally complete set of gates, which are a set of gates by which *any* Boolean function may be implemented, are used to fabricate the logic circuits. Examples of functionally complete sets are: [AND, OR, NOT]; [NOR]; [NAND] etc. NAND gate, also called universal gate, is a special gate and can be used for fabrication of all kinds of circuits. You may refer to further readings for more details on Universal gates.

Check Your Progress 1

- 1) What is a logic gate? What is the meaning of term Universal gate?

.....
.....

- 2) Prove the identity $I_1 + I_2 \cdot I_3 = (I_1 + I_2) \cdot (I_1 + I_3)$ using Truth Table

.....
.....

- 3) Simplify the function $F = ((A' + B)' + (A \cdot B)')$

.....
.....
.....
.....

- 4) Draw the logic diagram of the function before simplification.

.....
.....
.....

- 5) Draw the logic diagram of the simplified function.

.....
.....
.....

3.5 COMBINATIONAL CIRCUITS

Combinational circuit is an interconnection of gates, which produces one or more output based on some Boolean function for which it has been designed. A good combinational circuit does not include feedback loops. A combinational circuit is also represented using equivalent truth table or a Boolean function.

The output of the combinational circuit changes instantaneously with respect of input, though some delay is introduced due to transfer of signal from the circuit. This delay is dependent on the *depth* which is computed as number of gates in the longest path from input to output. For example, the depth of the combinational circuit of Figure 3.5 is 2.

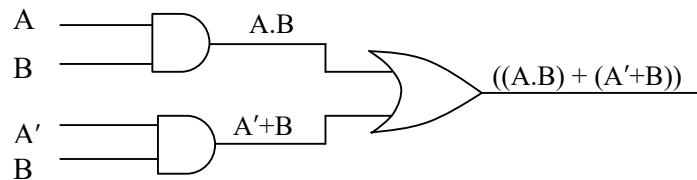


Figure 3.5: A two level AND-OR combinational circuit

Combinational circuits are primarily used to create the computational circuits of computer system logic; therefore, efficient design of combinational circuit may enhance the performance of a computer. Thus, one of the design goals of combinational circuits design is to minimize the number of gates in a combinational circuit. The constraints for combinational circuit design are:

- a combinational circuit should have limited depth.
- The number of input lines to a gate and number of gates to which the output of gate is fed, should be limited.

How can it be achieved? The following sub-sections explains the basic issues for combinational circuit design.

3.5.1 Canonical and Standard Forms of a Boolean expression.

An algebraic expression can exist in two standard forms:

- i) Sum of Products (SOP) form
- ii) Product of Sums (POS) form

Sum of product form

A SOP expression consists of terms consisting of operator (AND) which are joined by + operator (OR). For example, the expression $A'B.C + A.B$ is an expression consisting of three variables A, B and C. This expression is in SOP from with two terms $A'B.C$ and $A.B$. Each of these terms consists of product of variables using AND (.) operator, and the two terms are joined by an OR(+) operator, that is why the name Sum of Products form. In a SOP expression, a term which includes every variable in normal or complement form is called a minterm or standard product term. For example, in the above expression the term $A'.B.C$ is a minterm, but $A.B$ is not a minterm. However, if needed the term $A.B$ can be converted to two minterms as:

$$\begin{aligned} A.B &= A.B.(C+C') \\ &= A.B.C + A.B.C' \end{aligned}$$

In addition, please note that the value of minterm be ONE for exactly one possible combination of input values of A, B and C variables. For example, the minterm $A'.B.C$ will have a value 1 if $A'=1$ and $B=1$ and $C=1$. i.e., $A=0$, $B=1$ and $C=1$; for any other combination of values of A, B, C the minterm will have a ZERO value.

Interestingly, the number of minterms depends on number of variables. Given, n variables, the number of minterms will be 2^n . For example, for two variables,

$n=2 \Rightarrow 2^n=2^2=4$. The possible minterms for two variables are shown in the Figure 3.6.

Variables		Minterm	
A	B		
0	0	$A'B'$	m_0
0	1	$A'B$	m_1
1	0	AB'	m_2
1	1	AB	m_3

Figure 3.6: Minterms for two variables

A function can be represented as a sum of minterms, for example a function F in two variables using minterms $A'B + AB$ can be represented as:

$$F(A,B) = A'B + AB$$

which can be represented as:

$$F(A,B) = \sum (1,3)$$

(Please note that $A'B$ is minterm m_1 or 1 and AB is minterm m_3 or 3,

Product of Sum form

In this form an expression is written as the product (AND) of the terms, which use OR(+) as the basic operation, e.g. a three variables expression

$(A+B'+C).(A'+B')$ is in POS form having the two terms $(A+B'+C)$ and $(A'+B')$. Both the terms uses + operator and are joined by the AND operator, thus, the name Product of Sum form. In POS form a term, which include all the variables either in normal or complemented form is called a maxterm. For example, the expression $(A+B'+C).(A'+B')$ has a maxterm $(A+B'+C)$. A maxterm can have a value 0 for exactly one combination of input. For example the maxterm $A+B'+C$ will have value 0 if $A=0$, $B'=0$ and $C=0$, which is $A=0$, $B=1$ and $C=0$.

For any other combination of values of A, B and C, it will have a value 1.

The following Figure shows the maxterms. Please note that the output of maxterm is 0, only for the given combination of input.

A	B	Maxterm	
0	0	$A+B$	M_0
0	1	$A+B'$	M_1
1	0	$A'+B$	M_2
1	1	$A'+B'$	M_3

Figure 3.7: Maxterms for two variables

Example: Represent the function, whose SOP form is given below into an equivalent function in POS form.

$$F(A,B) = A'.B + A.B \text{ or } F(A,B) = \sum (1,3) \text{ or the truth table representation is:}$$

A	B	F(A,B)
0	0	0
0	1	1
1	0	0
1	1	1

Solution:

The complement of this function in SOP form is represented as (the minterms that has 0 as function output).

$$F'(A,B) = A'.B' + A.B' \quad \dots \dots (1)$$

Taking complement of equation (1), you will get the function F in POS form.

$$\begin{aligned} (F'(A,B))' &= (A'.B'+A.B')' \\ \Rightarrow F(A,B) &= (A'.B')' + (A.B')' \\ &= ((A')' + (B')) \cdot (A' + (B')) \\ &= (A+B) \cdot (A'+B) \end{aligned}$$

From table you can determine that function in POS form is:

$$F(A,B) = \prod (0,2) \text{ as the terms are } M_0 \text{ and } M_2$$

Thus, you can see:

$$F(A,B) = \sum (1,3) = \prod (0,2)$$

(SOP form) (POS form)

With this background of minterm and maxterm, you now are ready to perform the process of grouping of minterms, which will result in minimization of gates needed for a digital circuit. This is discussed in the next section.

3.5.2 Minimization of Gates

The simplification of Boolean expression is useful for the design of a good combinational circuit. There are several methods of doing so, however in this unit only the following two methods are discussed in details.

- Algebraic Simplification
- Karnaugh Maps

Algebraic Simplification

The following example explains the process of algebra simplification

Example : Simplify the function: $F(A,B,C) = \sum (0,1,4,5,6,7)$

Solution: Expanding the Minterms of the functions as:

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

Please note that C input has no effect on the function.

The truth table for the function and the equivalent expression is:

A	B	C	$F(A,B,C) = \sum (0,1,4,5,6,7)$	$B' + A \cdot B$
0	0	0	1	1
0	0	1	1	1
0	1	0	0	0
0	1	1	0	0
1	0	0	1	1
1	0	1	1	1
1	1	0	1	1
1	1	1	1	1

Thus, the logic circuit for the simplified equation $F(A,B,C) = AB + B'$

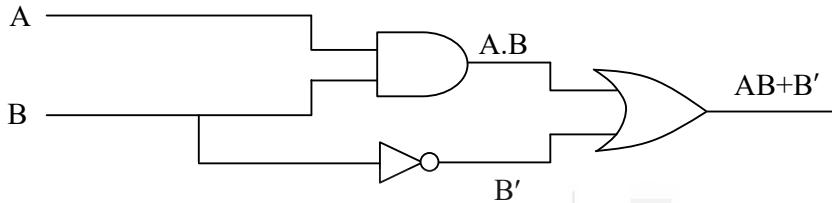


Figure 3.8: Simplified logic function using algebraic Simplifications

The logic diagram of the simplified expression is drawn using one NOT, OR and AND gate each.

The algebraic simplification becomes cumbersome because it is not clear which simplification should be applied next. The Karnaugh map is a simplified process of design of logic circuit using graphical approach. This is discussed next.

Karnaugh Maps

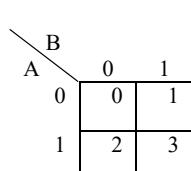
Karnaugh map is graphical way of representing and simplifying a Boolean function. They are useful for design of circuits involving 2 to 6 variables. The following is the process for simplification of logic circuit using Karnaugh map (K map).

Step 1: Create a Rectangular K-map and Assign binary and decimal equivalent values to each cell

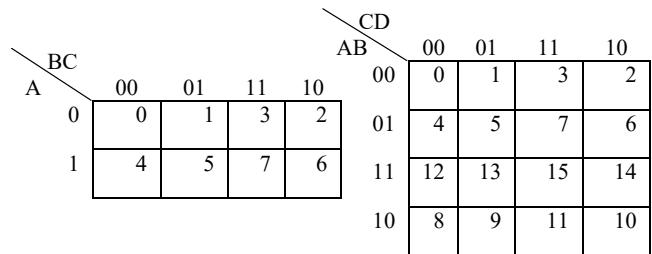
Create a rectangular grid of variables in a function. Figure 3.9 shows the map of two, three and four variables. A map of 2 variables consists of a grid $2^2 = 4$ elements or cells, while a map of 3 variables has $2^3 = 8$ cells and 4 variables has $2^4 = 16$ cells. Please note that the number of cells are same as the maximum possible number of minterms for those number of variables.

Each cell corresponds to a set of variable values, shown on the top or left of the K-map. For example, the values 00, 01, 11, 10 are written on the top of the cells of K-maps of 3 and 4 variables. These represent the values of the variables. For example, for the 3-variable k-map values written on BC side for the first cell 00 indicate B=0 and C=0. Please note that variable values are assigned such that any two adjacent cells (horizontal or vertical) differ only in one variable. For example, cell values 01 and 11 differ in 1 bit only, so are the values 11 and 10. The decimal equivalent values are shown inside the cells. For example, for a 3-variable K map cell having A=1 and BC=11, which is ABC as 111 is 7. Please note that the sequence of the number is not sequential in 3 variable and 4 variable K maps. This is because of the condition of change in only one variable between two adjacent cells. The decimal equivalent of minterm variable values are marked inside the cells. For example, decimal equivalent (or minterm equivalent) number placed in the cell having ABCD values as 1111 in the 4 variable k-map is 15.

Please note that bottom row is adjacent to top row; and last column is adjacent to first column as they differ in only one variable respectively.



2-variables K map



3-variables K map
Fig. 3.9: K-map of 2, 3 and 4 variables

4-variables K-map

Step 2: MAP the Boolean function or truth table of Boolean function into K-map.
Put a value 1 for every minterm for which the function output is 1.

Step 3: Simplify algebraic expression: Find adjacency of 1's in the K-map. You must find maximum adjacency in a sequence of ..., 8, 4, 2. A cell having 1 can appear in more than one adjacencies. Find the maximal adjacencies till all 1's are part of at least one adjacency.

Step 4: Write the Boolean term for each adjacency and join these terms using OR operator.

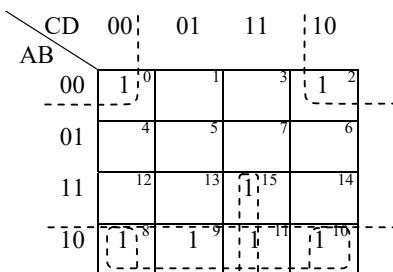
Resultant function is the simplified Boolean expression.

Example: Use K-map for finding the simplified Boolean function for the function $F(A,B,C,D) = \sum (0,2,8,9,10,11,15)$

Solution: The Truth table for the function is given below.

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

(a) Truth table



- (i) Adjacency 1: Four Corners (cells Numbered 0, 2, 8, 10)
- (ii) Adjacency 2: The bottom Row (cells Numbered 8, 9, 11, 10)
- (iii) Adjacency 3: Cell 11 and Cell 15

(b) Karnaugh's map

Figure 3.10: Truth table & K-Map of Function $F = \sum (0, 2, 8, 9, 10, 11, 15)$

The three adjacencies of the K-map are shown in the Figure 3.10. You can write the Boolean expression for each adjacency.

- 1) The adjacency 1 of four corners (cells Numbered 0, 2, 8, 10) can be written algebraically as:

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

Please note that an adjacency of 8/4/2 reduces the variables by 3/2/1 respectively.

A direct way of doing so is to identify the variable values of the adjacent cells which does not change, e.g. for this adjacency cell variable ABCD are 0000, 0010, 1000 and 1010. Thus the variable values of B and D does not change in all these 4 cells. In addition, since B and D have zero values among all these four cells, therefore, the expression is $B'D'$.

- 2) The four 1's in the bottom row (cells Numbered 8, 9, 11, 15)

The values of variable AB does not change and is 10 for the entire row, therefore, the expression for this adjacency would be $A.B'$

- 3) The two 1's in cell 11 and 15

$$\begin{aligned}
 & A.B.C.D + A.B'.C.D \\
 & = A.C.D.(B+B') \\
 & = A.C.D
 \end{aligned}$$

Also please infer it directly from values 1111 1011

Thus, the simplified Boolean expression using K-Map is

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

The simplified expression using K-map is in SOP form. In order to obtain expression in POS form the K-map is created for 0 values and adjacency identified. The following example explains these steps.

Example: Use K-map to find the simplified Boolean function for the function $F(A,B,C,D) = \sum (0,2,8,9,10,11,15)$ in POS form.

Solution: The truth table is shown in previous example. It can be used to draw K-map for 0 values, which will be for the complement of the function, i.e. $F'(A,B,C,D)$, as:

CD \ AB	00	01	11	10
00	0	0	3	2
01	4	5	7	6
11	12	13	15	14
10	8	9	11	10

K-map for F'

Four Adjacencies:

- (i) Cells (1,3,5,7) : A' and D does not change, so the term is A'.D
- (ii) 2nd Row (cells 4,5,7,6) : A' B does not change A'.B
- (iii) Cells 4,5,12,13 : B and C' does not change B.C'
- (iv) Cells 6 & 14 : B,C,D' does not change B.C.D'

Since, you have found the adjacencies of 0's, therefore

$$\begin{aligned}
 F'(A,B,C,D) &= A'.D+A'.B+B.C'+B.C.D' \\
 \text{or } F(A,B,C,D) &= (A'.D+A'.B+B.C'+B.C.D)' \\
 &= (A'.D+A'.B)' . (B.C'+B.C.D)' \\
 &= (A'.D)' . (A'.B)' . (B.C)' . (B.C.D)' \\
 &= ((A')'+D') . ((A')'+B') . (B'+(C)') . [(B.C)'+(D)'] \\
 F(A,B,C,D) &= (A+D') . (A+B') . (B'+C) . (B'+C'+D), \text{ which is in POS form.}
 \end{aligned}$$

In certain digital design situations, some of the input combination has no significance, for example, while designing the circuit for BCD, the output for the input combinations 0000 (digit 0) to 1001 (decimal digit 9) are needed. For the rest of input 1010 to 1111, the output does not matter. Such K-maps are designed using DONOT CARE condition. The output for DONOT CARE input combinations is marked as X in the K-map. The cells marked X can be used for determining the maximal dependencies, but need not be covered as the case is for all 1's output. A detailed discussion on this is beyond the scope of this unit.

What will happen if you went to design circuits for more than 6 variables? With the increase in number of variables K-Maps become more cumbersome and are not suitable. Other methods have been designed to do so, which are beyond the scope of this course.

Check Your Progress 2

- 1) Draw the truth table for the following Boolean functions:

$$\begin{aligned}
 \text{(i)} \quad F(A,B,C) &= A'.B.C'+A.B.C+A.B.C'+B.C+A.C \\
 \text{(ii)} \quad F(A,B,C) &= (A+B) . (A'+C') . (C'+B')
 \end{aligned}$$

.....

.....

- 2 Simplify the following using algebraic simplification. And draw the logic diagram for the function so obtained

$$\begin{aligned}
 \text{(i)} \quad F(A,B) &= (A'.B'+B')' \\
 \text{(ii)} \quad F(A,B) &= (A.B+A'.B')
 \end{aligned}$$

.....

.....

- 3) Simplify the following Boolean functions in SOP and POS forms using K-Maps. Draw the logic diagram for the resultant function.

$$F(A,B,C,D) = \Sigma (0,2,5,7,12,13,15)$$

.....

.....

3.6 DESIGN OF COMBINATIONAL CIRCUITS

The digital circuits, are constructed with NAND or NOR gates instead of AND–OR–NOT gates as they are *Universal Gates*. Therefore, any digital circuit can be implemented using these gates. To prove this point in the following diagram AND, OR and NOT gates are implemented using NAND and NOR gates. This is shown in figure 3.11 to 3.13 below.

NOT Operation:

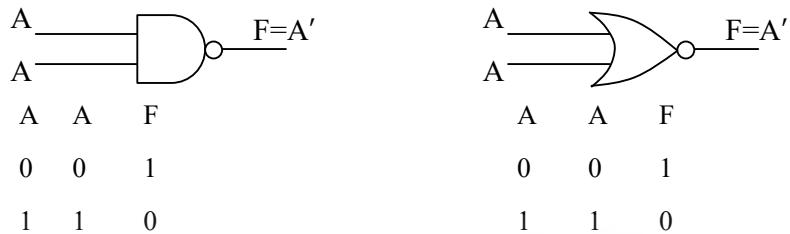


Figure 3.11: NOT Operation using NAND or NOR gates

AND Operation:

Performing AND using NAND gates can be achieved by first performing the NAND of the input followed by inverting the output as shown in Figure 3.12

$$\begin{aligned} F &= A \cdot B \\ &= ((A \cdot B)')' \\ F &= (A \text{ NAND } B)' \end{aligned}$$

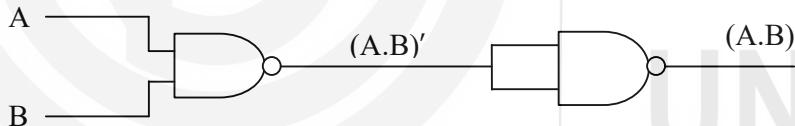


Figure 3.12: Logic circuit of AND Operation using NAND gates

AND operation can also be implemented using NOR gates. The following Boolean expression identifies that first NOR gates are used to invert the A and B input followed by taking NOR of A' with B'

$$\begin{aligned} F &= A \cdot B \\ F &= ((A \cdot B)')' \\ &= (A' + B')' \\ &= A' \text{ NOR } B' \end{aligned}$$

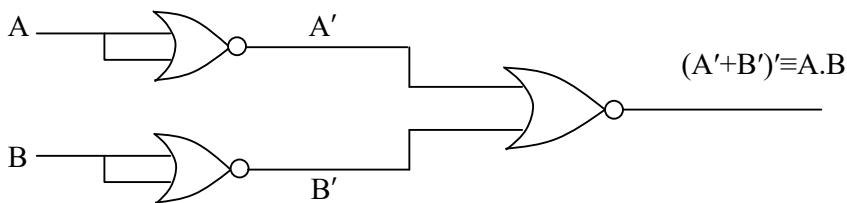


Figure 3.13: Logic circuit of AND Operation using NOR gates

OR Operation:

OR operation can be performed using NAND gate. Please refer to following Boolean expressions:

$$\begin{aligned} F &= A+B \\ &= ((A+B)')' \\ F &= (A'.B')' \Rightarrow A' \text{ NAND } B' \end{aligned}$$

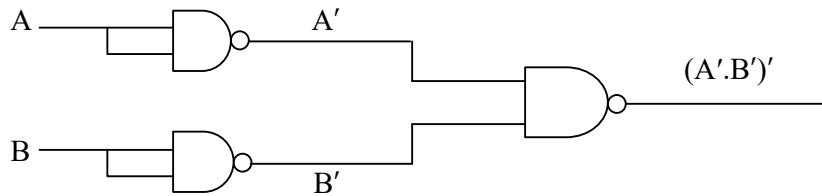


Figure 3.14: Logic circuit of OR Operation using NAND gates

$$\begin{aligned} F &= (A+B) \\ F &= ((A+B)')' \\ F &= (A \text{ NOR } B)' \end{aligned}$$

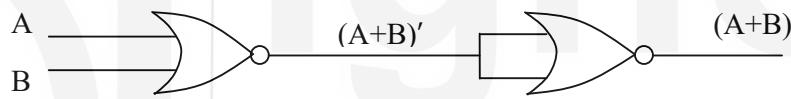


Figure 3.15: Logic circuit of OR Operation using NOR gates

A Boolean function can be implemented using the universal NAND or NOR gates by expressing the function in sum of product form as explained in the following example.

Example: Draw the circuit for $F(A,B,C) = \sum(0,1,3,7)$ using NAND gates.

Solution: Find the optimal Boolean function using K-map in SOP form:

	BC	00	01	11	10
A	0	0	1	1	0
	1	1	0	0	1
	4	5	7	6	

$$F(A,B,C) = A'B' + BC$$

The AND – OR gate logic circuit for this is:

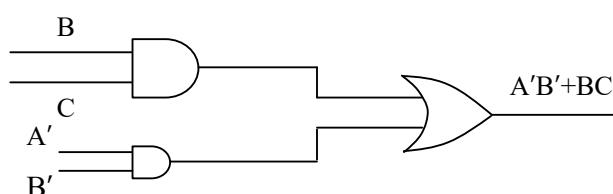


Figure 3.16 Logic Circuit Using AND-OR gate

For NAND gate logic circuit

$$\begin{aligned} F(A,B,C) &= (A'B' + BC) \\ &= ((A'B')' + (BC)')' \\ &= ((A' \cdot B') \cdot (B \cdot C))' \quad (\text{Use of Demorgan's law}) \\ &= ((A' \text{ NAND } B') \cdot (B \text{ NAND } C))' \\ &= (A' \text{ NAND } B') \text{ NAND } (B \text{ NAND } C) \end{aligned}$$

Thus, the circuit can be made simply by replacing two levels AND-OR circuit by NAND gates:

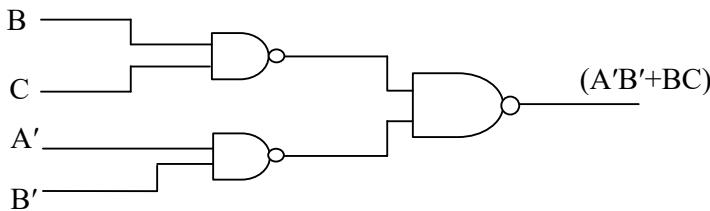


Figure 3.17: Logic Circuit by Replacing AND - OR circuit by NAND gates

A combinational circuit is required to produce a specific set of output for a given step of input. The design of a combinational circuit simply requires the following steps:

Step 1: Make the Truth table for the required design. You must draw the truth table for every output value.

Step 2: Use K-map or any other method to create optimal Boolean function that creates the desired output. One function is designed for each output.

Step 3: Draw the resultant circuit using universal gates.

The next section first design the design of half adder circuit as a combinational circuit. In addition, next section discusses some of the combinational circuits, the design of which is not detailed in this Unit..

3.7 EXAMPLES OF COMBINATIONAL CIRCUITS

In this section, first the combinational circuits design is demonstrated using basic combinational circuits like half and full adders. This is followed by discussion on combinational circuits like decoders, multiplexers etc.

3.7.1 Adders

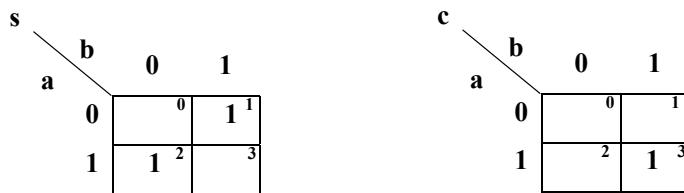
Addition is one of the most common arithmetic operations. In this section two different kinds of addition circuits are designed. The first of the two circuit adds two binary digits and is called a half adder, while the second adds three bits-two addend and one carry bit, and is called a full adder.

Half Adder:

Let us assume that a half adder circuit is adding two bits a and b to produce one sum bit (s) and one carry bit (c). The following truth table shows this operation. Please note one adding $a = 1$ and $b = 1$ you get a carry as 1 and sum bit as 0 as shown in truth table. The K-maps for the addition is shown in figure 3.18.

a	b	c	s
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

(a) Truth table



(b) K-map for sum bit

(c) K-map for carry bit

Figure 3.18: Truth table and K-maps for half adder

The Boolean expression for them from the k-maps are:

$$s = a'b + ab' \quad \text{and}$$

$$c = a.b$$

The logic circuit for the half adder is based on the Boolean expressions given are shown in Figure 3.19.

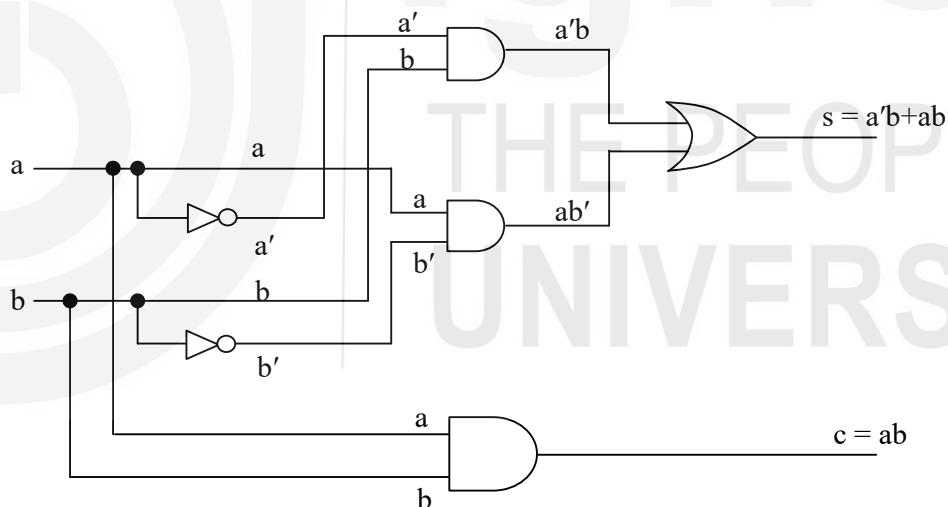


Figure 3.19: The half adder circuit-input addend bits a, b; output sum bit (s) and carry bit (c)

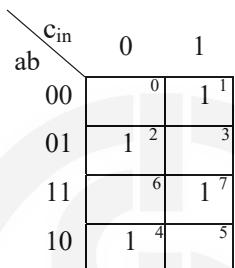
Full Adder:

Full adder is a circuit that adds 3 bits, viz. 2 addend bits and one carry bit. The truth table for full adder is shown in Figure 3.20. Please note that in figure 3.20, c_{in} is carry in bit and c_{out} is carry out bit.

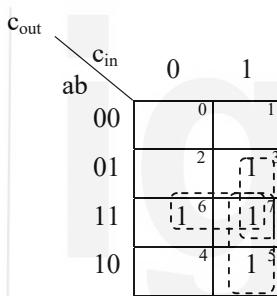
Input				Output	
Decimal equivalent	a	b	c _{in}	Carry out (c _{out})	Sum (s)
0	0	0	0	0	0
1	0	0	1	0	1
2	0	1	0	0	1
3	0	1	1	1	0
4	1	0	0	0	1
5	1	0	1	1	0
6	1	1	0	1	0
7	1	1	1	1	1

Figure 3.20: The Truth Table for Full Adder

Please note that in the truth table, when $a = 1$, $b = 1$ and $c_{in} = 1$, than the output is 11, which means sum bit (s) is 1 and carry out bit is also 1. The K-map for these are also shown in Figure 3.21


K-map for sum bit

No adjacency


K – Maps for c_{out}

Three adjacencies

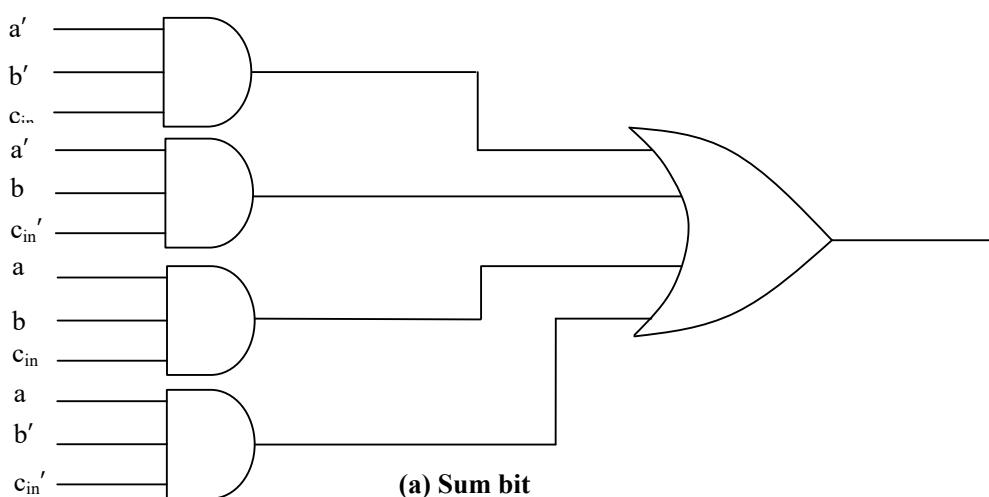
Figure 3.21: The K-maps for Full Adder

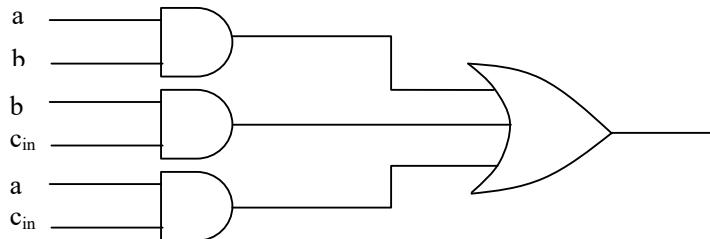
The Boolean functions based on the K-maps given in Figure 3.21 are given below:

$$s = a' \cdot b' \cdot c_{in} + a' \cdot b \cdot c_{in}' + a \cdot b \cdot c_{in} + a \cdot b' \cdot c_{in}'$$

$$c_{out} = a \cdot b + b \cdot c_{in} + a \cdot c_{in}$$

Figure 3.22 shows the full adder circuit. Please note that for simplicity the circuits for inverting the input values are not drawn.





(b) Carry Out bit

Figure 3.22: Full Adder

Full adder and half adder only perform bit addition of two operands without or with carry bit respectively. However, binary numbers have several bits e.g. integers can be 4 byte long. How will they be added? This is performed by creating a sequence of full adders, where carry out bit of the lower bit addition is fed as carry in bit of next higher bit addition, as shown in figure 3.23.

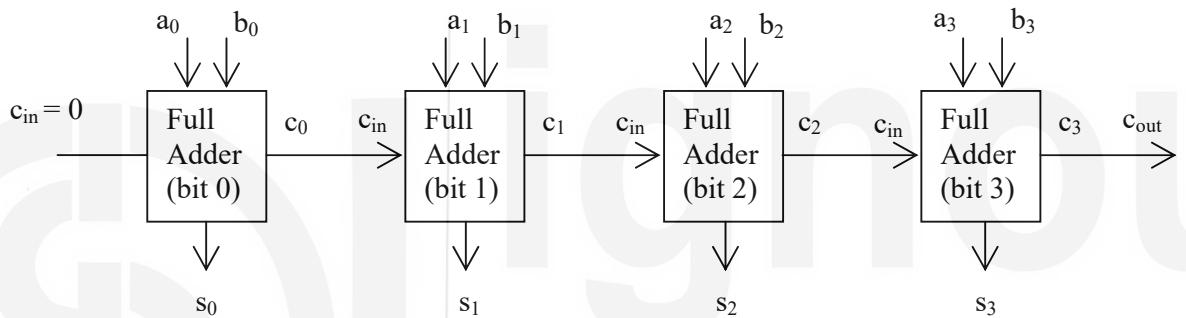
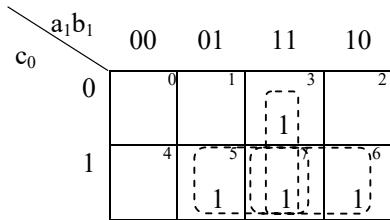


Figure 3.23: Addition of a 4 bit number using 4 full adders

Please note that in the Figure 3.23, the carry out of the previous bit addition is input as carry in bit of the next bit addition. For example, the value of c_0 bit, which is carry out of bit 0 addition, will be available if the full adder of bit 0 has been completed. Drawback of this circuit is the time taken to add the number is large as each full adder will take some signal propagation time, and the addition of the next bit cannot be performed till the c_{in} bit is available. A faster binary adder circuit would predict the carry bit. These are called look-ahead carry adders. It may be noted that carry out of the 0th bit can be computed using the Boolean function $c_0 = a_0.b_0$. The value c_0 is input as c_{in} of full adder of bit 1. The truth table of Figure 3.24 has been drawn for full adder of bit 1. This truth table can be used to design circuit that computes the value of c_1 prior to actual addition.

Decimal equivalent	Input			Output	
	c_0	a_1	b_1	c_1	s_1
0	0	0	0	0	0
1	0	0	1	0	1
2	0	1	0	0	1
3	0	1	1	1	0
4	1	0	0	0	1
5	1	0	1	1	0
6	1	1	0	1	0
7	1	1	1	1	1

Figure 3.24: Truth table for Full adder (bit 1)

**Figure 3.24: K-map for c_1 output of Full adder (bit 1)**

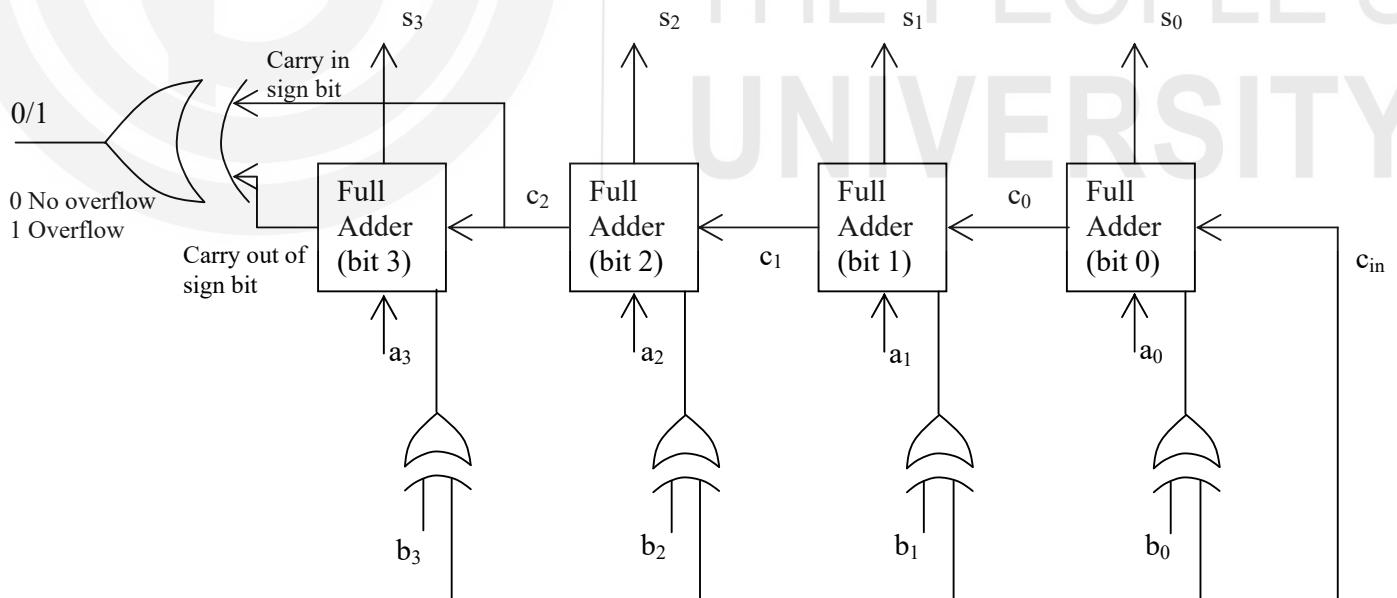
There are three adjacencies in the K-map of Figure 3.24. The resultant Boolean function for c_1 would be:

$$\begin{aligned}c_1 &= a_1 \cdot b_1 + c_0 \cdot a_1 + c_0 \cdot b_1 \\c_1 &= a_1 \cdot b_1 + c_0 \cdot (a_1 + b_1) \quad (\text{Taking } c_0 \text{ common}) \\c_1 &= a_1 \cdot b_1 + a_0 \cdot b_0 \cdot (a_1 + b_1) \quad (\text{Replacing } c_0 \text{ by its equivalent})\end{aligned}$$

Logic circuits can be designed for prediction of carry bits c_0 , c_1 , etc. and resultant circuits can be implemented along with full adder circuits. You can observe that Boolean expression for higher order carry bits like c_2 , c_3 etc. will become more complex, which results in complex logic circuits. Thus, look ahead carry bit adders may be implemented for addition of binary numbers of size 4-8 bits.

Adder-Subtractor Circuit

Adder subtractor circuit is an interesting design, in which a same circuit is used for addition as well as subtraction. This example shows how with some additional logic, you may be able to perform additional operations. ALU is a fine example of extension of such logic. Figure 3.25 shows the circuit of 4 bit adder-subtractor circuit by using full adders.



Mode bit: Addition
Operation mode bit = 0
Subtraction operation mode
bit = 1

Figure 3.25: Adder Subtractor circuit using full adders for addition and subtraction of 4-bit 2's complement numbers

You may please note that the mode bit controls the b input. The following Figure shows the details of operation.

Mode bit = 0		Mode bit = 1	
Input b	Inputs the bits of b input XOR with mode bit value (0)	Input b	Input value after taking XOR of input b and mode bit
0	0 XOR 0 = 0	0	0 XOR 1 = 1
1	1 XOR 0 = 1	1	1 XOR 1 = 0
Thus, when mode bit is 0, the input to full adder is the value of input b.		Thus, when mode bit is 1, the input to adder is the value of 1's complement of b	
$c_{in} = 0$ as mode bit = 0 so the circuit adds input a and input b.		$c_{in} = 1$, so the addition is $r = a+b'+1$ or $r = a+2'$'s complement of b $r = a - b$; subtraction of a and b	

Figure 3.26: Use of Mode bit to control b input in Adder subtractor circuit

Please also note that in 2's complement notation the last bit is treated as sign bit. The overflow condition is checked by finding if the carry into the sign bit and carry out of sign bit are same or not same. In case carry in to the sign bit is not the same as carry out of the sign bit, then overflow is set to 1 (by XOR gate) else overflow is set to 0 (No overflow).

3.7.2 Decoders

Decoder, as the name suggests, decodes the input to one of the output line. Figure 3.27 shows the truth table and logic circuit of a 2×4 decoder.

Truth table

Input		Output			
a	b	c	d	e	f
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

The Boolean functions for various output values are

$$c = a' b'$$

$$d = a' b$$

$$e = a b'$$

$$f = a b$$

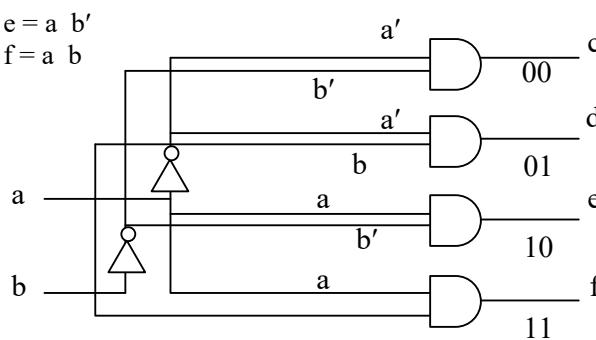


Figure 3.27: 2×4 decoder

A decoder line would be selected if it has an output 1. In general, decoder is a very useful circuit for selecting lines and forms the basis of Random Access Memory.

Please note that numbers of output for 2 bit decoder are $2^2 = 4$; hence the name 2×4 decoder. Similarly, the number of output for a 3 bit input would be $2^3 = 8$ and it is called 3×8 decoder.

3.7.3 Multiplexer

A multiplexer allows sharing of a line by multiple inputs. It may be very useful for serialization of data bits over a single output line. The design of a multiplexer is however, different from other combinational circuits as it is the selection lines which control the selection of input line. The following is the truth table of a 4×1 multiplexer. A 4×1 multiplexer selects one of the 4 input lines to be transmitted over a single output. Out of these 4 lines, which will be selected, will be determined by 2 selection lines. How many selection lines will be required for 8×1 multiplexer? Since $2^3 = 8$, so 3 selection lines would be required for 8×1 multiplexer.

Input		Input	Output
Selection Lines			
s_1	s_0		
0	0	I_0	I_0
0	1	I_1	I_1
1	0	I_2	I_2
1	1	I_3	I_3

Please note the values of output can be I_i , where the value of subscript i can vary from 0 to 3.

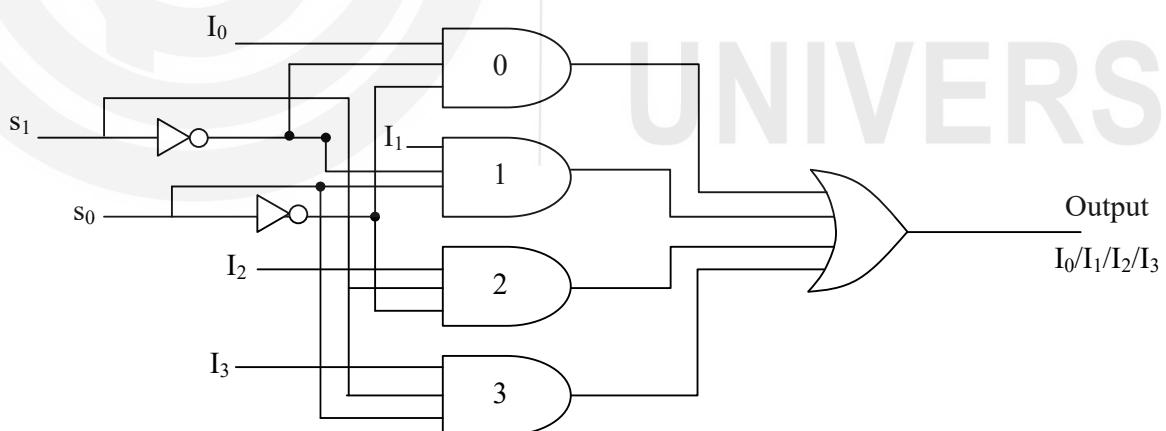


Figure 3.28: 4×1 Multiplexer

Please note this is a very important circuit for sharing of an output line across many sources of input.

3.7.4 Encoders

An encoder, in general, is the inverse of a decoder. Based on its input it produces a specific output. For example, the truth table of a 4×2 encoder is shown in Figure 3.29

Input				Output	
I_0	I_1	I_2	I_3	O_1	O_0
1	0	0	0	0	0
0	1	0	0	0	1
0	0	1	0	1	0
0	0	0	1	1	1

Figure 3.29: Truth Table of 4×2 encoder

The simple expression for various output can be

$$O_1 = I_2 + I_3$$

$$\text{and } O_0 = I_0 + I_1$$

Thus, the simple circuit for this encoder is

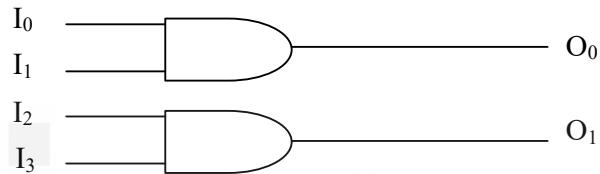


Figure 3.30 Logic Diagram of a simple encoder

3.7.5 Programmable Logic Array

The basic combinational circuits can be implemented using AND-OR-NOT gates. PLAs are circuits which are prefabricated for all possible combination of AND, OR, NOT gates. They can be used to fabricate any kind of logic circuit. They are primarily designed for SOP form of logic circuit.

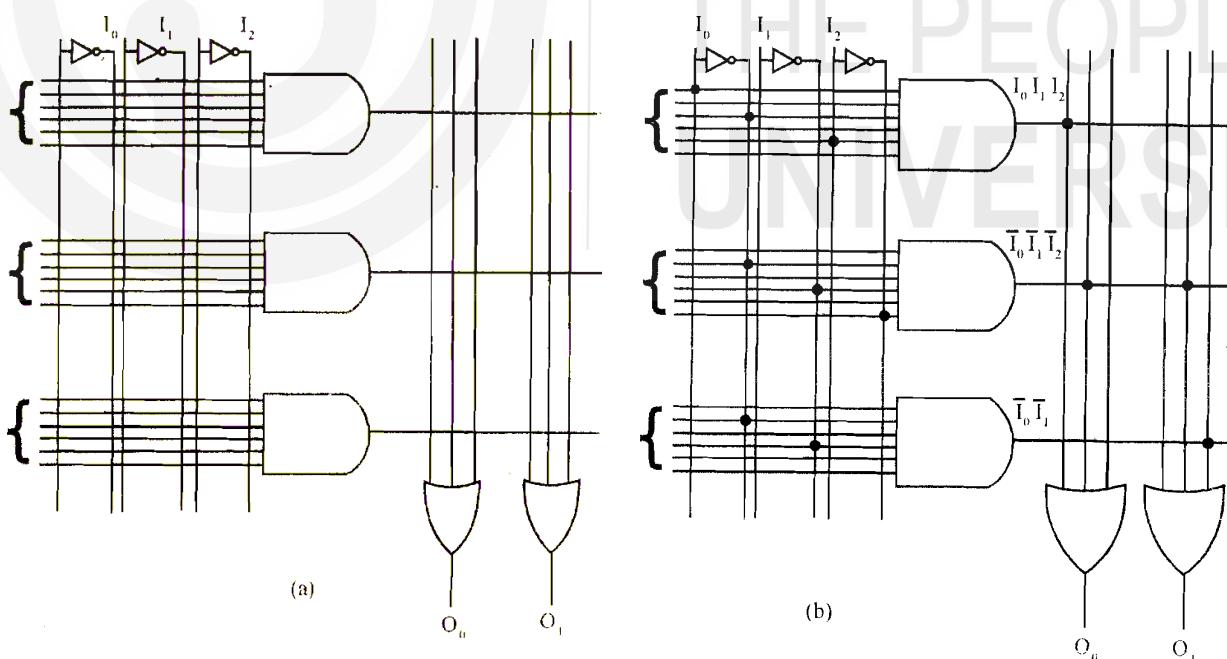


Figure 3.31: Programmable Logic Array

The figure 3.31(a) shows a PLA of 3 inputs and 2 outputs. Figure 3.31(b) shows an implementation of logic function given below using the PLA of figure 3.31 (a).:

$$O_0 = I_0 \cdot I_1 \cdot I_2 + I_0' \cdot I_1' \cdot I_2'$$

$$O_1 = I_0' \cdot I_1' \cdot I_2' + I_0' \cdot I_1 \cdot I_2'$$

3.7.6 Read-only-Memory (ROM)

ROM is an example of use of Programmable Logic Devices (PLD). It stores the binary information using a combinational circuit. The RAM follows the simple sequence. Figure 3.32 shows a ROM of size 4×2 , which has 4 lines of 2 bits each. Please note the use of 2×4 decoder. Also note that wherever the line will be connected an output will appear. These connections are embedded within the hardware. Thus the information of ROM is not lost even after the power failure..

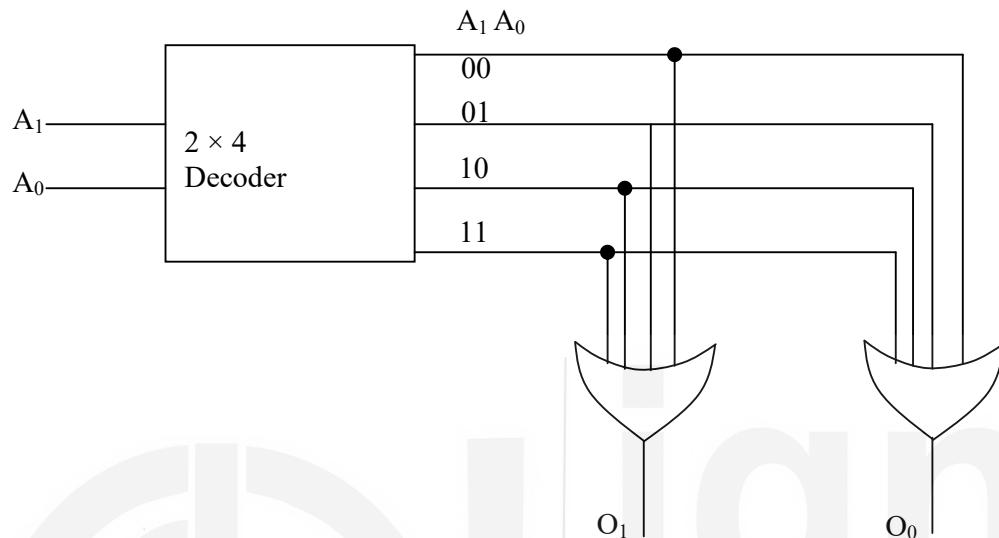


Figure 3.32: ROM Design

Please note that the ROM shown in Figure 3.32 have the following content:

Address Line selection $A_1 A_0$	ROM content $O_1 O_0$
0 0	1 0
0 1	0 0
1 0	1 0
1 1	1 0

Please note the number of words in the ROM is 4 which is $2^2 = 4$ as it has 2 address lines. A ROM with 3 address lines will have 8 words. ($2^3 = 8$).

The size of word can be chosen by the designer and, in general, it can be 8, 16, 32 or 64 bit as per the machine firmware designer. The address lines of ROM select any one line as per the address.

Check Your Progress 3

- Design a combinational circuit, which takes four bit input and produces an output 1 if the input contains three consecutive 1 bits.
-
.....

- Draw the logic diagram of the function as above using
 - AND-OR-NOT gates &

(ii) NAND gate

.....
.....
.....

3) Consider the circuit of Figure 3.26, what would be the output of the circuit if:

- (i) Input a is 1010 and input b is 1100 and mode bit is 0
(ii) Input a is 0010 and input b is 0100 and mode bit is 1

.....
.....
.....

4) Why is PLA needed?

.....
.....
.....

5) Design a full adder using two half adder circuits.

.....
.....
.....

3.8 SUMMARY

This Unit introduces you to some of the basic concepts relating to computer logic. The Unit first introduces the concept of logic gates, the most fundamental unit of logic circuits. The Unit then explains the process of making simple logic circuits, including combinational circuit. The mathematical foundation of the logic circuit design, the Boolean algebra is also introduced. The Karnaugh's map was used to design simpler circuit. The Unit also explains the desing of different kinds of adders circuit, highlighting , how complex circuit can be desinged using K-map. Finally, the Unit explains some of the most fundamental combinational circuits like decoder, multiplexer, encodes, PLA's etc. It may be noted that the objective of this Unit is not to make you a computer hardware designer, but to introduce you to some of the basic concepts of circuit design.

You can refer to latest trends of design and development including VHDL (a hardware design language) in the further readings.

3.9 SOLUTIONS/ANSWERS

Check Your Progress 1

- 1) A logic gate is most fundamental circuit that can be fabricated on a silicon chip. A logic gate operates at signal level to produce simple logic like AND, OR, NOT etc. A Universal gates can be used to implement each and every kind of logic circuit. Two examples of universal gates are NAND and NOR.

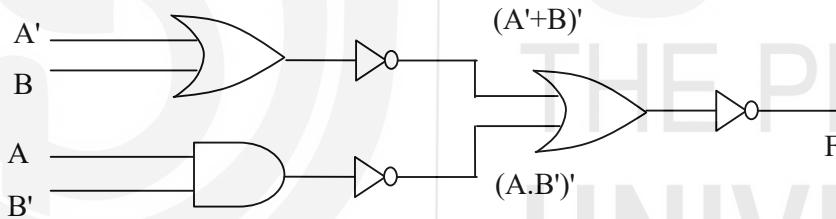
2)

I_1	I_2	I_3	I_1+I_2, I_3	$(I_1+I_2).(I_1+I_3)$
0	0	0	0	0
0	0	1	0	0
0	1	0	0	0
0	1	1	1	1
1	0	0	1	1
1	0	1	1	1
1	1	0	1	1
1	1	1	1	1

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

$$\begin{aligned}
 &= ((A'+B)')' . ((A.B')')' \quad (\text{by DeMorgan's Law}) \\
 &= (A'+B) . (A.B') \quad (\text{as } (a')' = a) \\
 &= ((A'+B).A).((A'+B).B') \\
 &= ((A'.A)+(B.A)).((A'.B')+B.B') \\
 &= ((0+A.B).(A'.B') + 0) \\
 &= (A.B.A'.B') = 0
 \end{aligned}$$

4) Draw the logic diagram of the function before simplification.



5) Just the F can be connected to logical 0 input as $F=0$

Check Your Progress 2

1)

A	B	C	$F=A'.B.C' + A.B.C + A.B.C' + B.C + A.C$	$F = (A+B) . (A'+C') . (C'+B')$
0	0	0	0	0
0	0	1	0	0
0	1	0	1	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	1
1	1	1	1	0

2 (i) $F(A,B) = (A'.B'+B')'$

$$\begin{aligned}
 &= (A'.B')' . (B')' \quad (\text{DeMorgan's Law}) \\
 &= ((A')'+(B')) . B \quad (\text{DeMorgan's Law}) \\
 &= (A+B) . B
 \end{aligned}$$

$$= A \cdot B + B \cdot 1 = B \cdot (A+1) = B$$

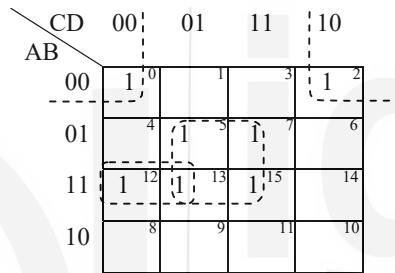
B ————— F

$$\begin{aligned} \text{(ii)} \quad F(A,B) &= (A \cdot B + A' \cdot B')' \\ &= (A \cdot B)' \cdot (A' \cdot B')' \\ &= (A' + B) \cdot (A + B) \\ &= (A' + B) \cdot A + (A' + B) \cdot B \\ &= A' \cdot A + B' \cdot A + A' \cdot B + B' \cdot B \\ &= A \cdot B' + A' \cdot B \end{aligned}$$

The logic diagram is same as sum bit of half adder. It is also the A XOR B.

- 3) Simplify the following boolean functions in SOP and POS forms using K-Maps.
Draw the logic diagram for the resultant function.

$$F(A,B,C,D) = \Sigma (0,2,5,7,12,13,15)$$



Three adjacencies

- i) Cells 0 and 2: The variables does not change A' B' D'
- ii) Cells 12 and 13; The variable does not change A B C'
- iii) Cells 5,7,13,15; two variables does not change B D

The expression is $F = A' \cdot B' \cdot D' + A \cdot B \cdot C' + B \cdot D$

Check Your Progress 3

- 1) The Truth table:

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

The K-map for the Truth table:

		CD \ AB	00	01	11	10
		00	0	1	3	2
		01	4	5	1	7
		11	12	13	1	15
		10	8	9	11	10

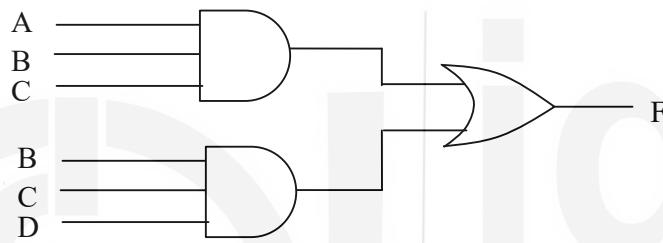
Only two adjacencies: Cells 7 and 15 B C D and

Cells 15 and 14 A B C

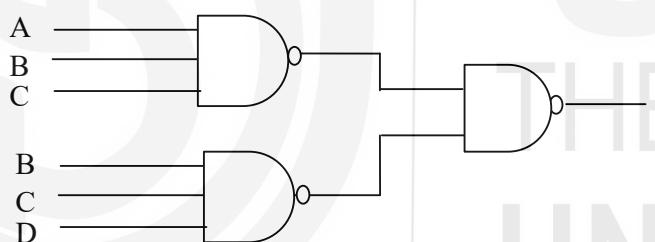
Therefore, the function is: $F = A \cdot B \cdot C + B \cdot C \cdot D$

2) Logic diagram of the function as above

(i)



(ii)



3) (i) Input a is 1010 and input b is 1100 and mode bit is 0

Bit wise addition will be as follows:

a	1	0	1	0
b	1	1	0	0
c	0	0	0	0
sum bit	0	1	1	0
carry out of sign bit	1			

carry in to sign bit NOT equal to carry out of sign bit,
OVERFLOW

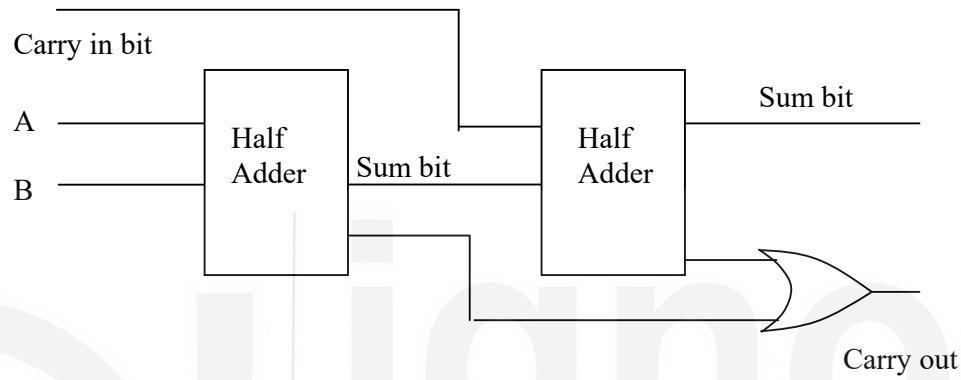
(ii) Input a is 0010 and input b is 0100 and mode bit is 1

Bit wise addition will be as follows:

a	1	0	1	0
b (1's complement)	0	0	1	1
c	0	1	1	1
sum bit	1	1	1	0
carry out of sign bit	0			

carry in to sign bit IS EQUAL to carry out of sign bit,
NO OVERFLOW

- 4) PLA's can be fabricated as a chip that can be customised as per the need of the SOP logic.
- 5) A half adder adds two addend bits, whereas a full adder adds the two addends and previous carry bit, therefore, one half adder will be needed to add two addend bits, and second half adder will be needed to sum the sum of first half adder and previous carry bit. The output carry will be set, if any of the two half adder produce the carry out. The following block diagram shows this construction:



Structure	Page Nos.
4.0 Introduction	
4.1 Objectives	
4.2 Sequential Circuits: The Definition	
4.3 Latches and flip-flops	
4.3.1 Latches	
4.3.2 Flip-Flop	
4.3.3 Excitation Tables	
4.3.4 Master Slave Flip Flops	
4.3.5 Edge Triggered Flip-flops	
4.4 Sequential Circuit Design	
4.5 Examples of Sequential Circuits	
4.5.1 Registers	
4.5.2 Counters Circuit	
4.5.3 Synchronous Counters	
4.5.4 Random Access Memory	
4.6 Summary	
4.7 Solutions/ Answers	

4.0 INTRODUCTION

The first Unit of this Block explained the basic structure and process of instruction execution. Unit 2 provided a detailed description of data representation and Unit 3 presented the concepts of basic functional unit of a computer, viz. the logic gates and combinational circuits. In this unit, you will be introduced to one of the most fundamental circuit that can store one bit of data called flip flops. The unit also explains how flip-flops and additional logic circuit can be used to make registers, counters, sequential circuits etc. Finally, the Unit also introduces you to simple design of a sequential circuit.

4.1 OBJECTIVES

After going through this unit you will be able to:

- explain the functioning of flip-flops;
- determine the behaviour of various latches;
- construct excitation table of a flip-flop;
- explain circuits of a computer system like registers, counters etc.

4.2 SEQUENTIAL CIRCUITS: THE DEFINITION

A sequential circuit is an interconnection of combinational circuits and storage elements. The storage elements, called flip-flops, store binary information that indicates the state of sequential circuit at that point of time.

Figure 4.1 highlights that a sequential circuit may involve combinational circuits (which were discussed in Unit 3) the flip-flops (which are discussed in this unit) and a system clock, which is a useful timing device of a computer system.

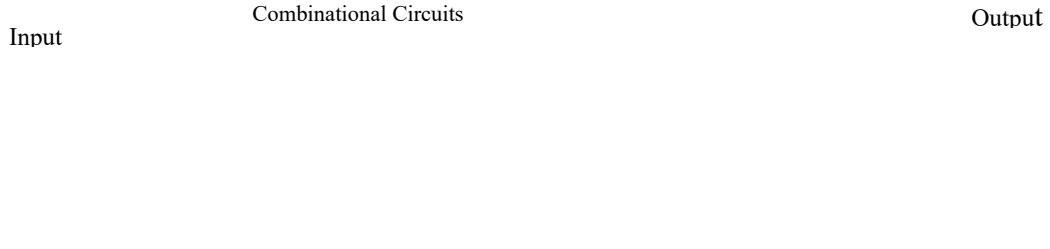


Figure 4.1: Block Diagram of sequential circuits.
 (Ref: M. Morris Mano, Charles R. Kime: Logic and compute design fundamental, 2nd Edition, Pearson Education)

The sequential circuits are time dependent. The present state of a combinational circuit is identified by the present output of flip-flop. This output may change over a passage of time and can also be used as one of the input. This change in state can occur either in synchronous or asynchronous manner with respect to system clock.

Synchronous circuits use flip-flops and their state can change only at discrete intervals. Asynchronous sequential circuits are regarded as combinational circuit with feedback path. Such circuits may unstable at times, when the propagation delays of output to input are small. Thus, complex asynchronous circuits are difficult to design.

Clock Pulse and sequential circuits

A sequential circuit uses clock pulse generator, which gives continuous clock pulse to synchronize change in the state of the circuit. Figure 4.2 shows the form of a clock pulse.

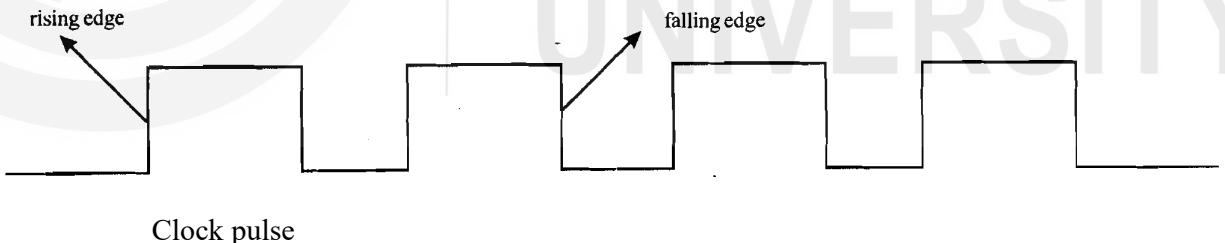


Figure 4.2: Clock signals of clock pulse generator

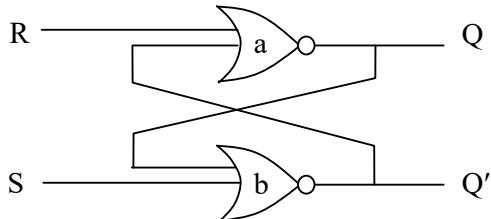
A clock pulse can have two states, viz. 0 or 1, which are also called disabled or active state. Flip-flops are allowed to change their states, in general, with the rising or falling edge of the clock pulse, so as to make stable changes in states of the flip-flops.

4.3 LATCHES AND FLIP-FLOPS

A **latch** is a storage element with basic logic circuit, which can store 1-bit of data. It itself is a sequential circuit. Flip-flops are constructed using latches and have more complex timing sequences than latches. Therefore, in order to learn flip-flops, learning the basic concept of latches is very useful, which is discussed next.

4.3.1 Latches

A basic latch can be constructed using either two NOR or two NAND gates. Figure 4.3 (a) shows logic diagram for S-R latch using NOR gates. This latch has two inputs viz. S and R for Set and Reset respectively; and one output Q. Please note Q' output is complement of the output Q. This flip flop exhibits two states called SET state (when the flip-flop output Q is 1, that is Q'=0) and RESET state or clear state (Q=0; Q'=1).



(a) Logic Diagram

S	R	Q	Q'	Comment
0	0	0/1	0/1	No Change in State
0	1	0	1	Reset State
1	0	1	0	Set State
1	1	-	-	Undefined Input

(b) Truth Table

Figure 4.3: SR Latch using NOR gates

The following table shows the truth table for NOR gates using in the S-R latch of Figure 4.3 (a).

The Truth tables for NOR gates of Figure 4.3							
NOR gate Marked 'a'				NOR gate Marked 'b'			
	Input		Output		Input		Output
	R	Q'	Q		S	Q	Q'
0	0	0	1	0	0	0	1
1	0	1	0	1	0	1	0
2	1	0	0	2	1	0	0
3	1	1	0	3	1	1	0

Let us examine the latch in more details. Assume that initially latch is in clear state, i.e. Q=0 and Q'=1; also assume that both S and R input are 0. The states of the latch will be as follows (refer to the NOR gate truth table given above):

Gate 'a'

$$\text{Input } R \ Q' :: 0 \ 1 \Rightarrow \text{Output } (Q) \ 0$$

Gate 'b'

$$\text{Input } S \ Q :: 0 \ 0 \Rightarrow \text{Output } (Q') \ 1$$

Output of latch stays in CLEAR state

(i) *Setting the latch:*

Now assume that S is changed to 1 and R remains 0 during this time, then the output of Gate 'b' will change first:

$$S \ Q :: 1 \ 0 \Rightarrow Q' \text{ will become } 0$$

Gate 'a' now has the following input:

$$R \ Q' :: 0 \ 0 \Rightarrow Q \text{ will be set to } 1.$$

Gate 'b' now has the following input

$$S \ Q :: 1 \ 1 \Rightarrow Q' \text{ will stay at } 0.$$

SET state

Thus, Flip-flop will be in SET state.

Finally, after some time S will become 0;

At that time, gate 'a'

$$R \ Q' :: 0 \ 0$$

Q stays at 1

Gate 'b'

$$S \ Q :: 0 \ 1$$

Q' stay at 0

Latch will stay in SET State

(ii) *Reset the latch:*

Now assume that input S remains at 0 and input R is changed to 1, also assume that at this time the latch is in Set state ($Q = 1 \& Q' = 0$), then the output of Gate 'a' will change as Gate 'a'

$$\begin{aligned} R Q' :: 1 0 \Rightarrow & \quad Q \text{ will become } 0. \\ \text{Gate 'b'} & \\ S Q :: 0 0 \Rightarrow & \quad Q' \text{ will become } 1 \end{aligned} \quad \left. \begin{array}{l} \text{Latch is in Reset state.} \\ \end{array} \right]$$

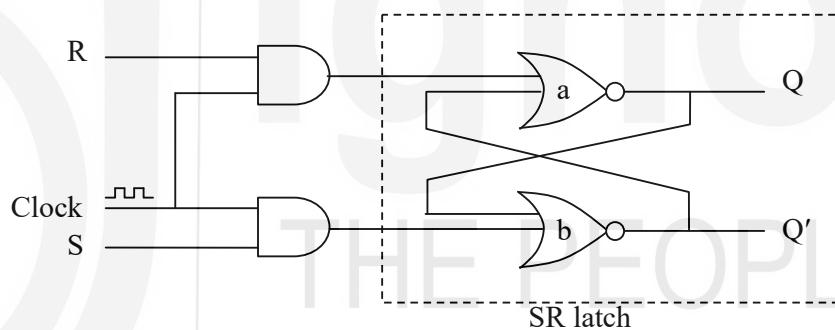
Once again, when S and R both input will become 0, latch will remain in RESET state.

(iii) *When both S and R become 1 simultaneously, then?*

A basic S-R latch, in general, changes state at any time, which may result in asynchronous changes in Q output, which can make system unstable. Therefore, latches are constructed with controlled input using clock. This is explained next.

SR latch with Clock

The following diagram shows an SR latch which changes its data only with the occurrence of a clock pulse.



(a) Logic Diagram

Clock(c)	S	R	Present State Q_t before the clock pulse	Next State/ Q_{t+1} after occurrence of clock pulse.	Comments
0	Any	Any	0/1	0/1	No change in state
1	0	0	0/1	0/1	No change in state
1	0	1	0/1	0	Reset the latch
1	1	0	0/1	1	Set the flip-flop
1	1	1	0/1	-	Not defined.

(b) Characteristic Table

Figure 4.4: R-S latch with clock.

Operations on this clocked SR latch are given below:

- 1) If no clock signal i.e. $\text{clock}=0 \Rightarrow$ No change in state of latch.
- 2) Presence of clock signal
 - (i) if $S=0$ and $R=0$, No change in state/output stays same as earlier state.
 - (ii) if $S=1$, $R=0$, then next state is the SET state $Q=1 \& Q'=0$
 - (iii) if $R=1$ $S=0$, then next state is the RESET state $Q=0 \& Q'=1$
 - (iv) if both S and R become 1, then next state/output is not defined.

D Latch

Principles of Logic Circuits II

The D (data) latch is modification of RS latch. D latch only uses one input named D, it stores the value of D in the latch, e.g. if the D input is 1, then the next state of latch will also be 1. Figure 4.4 shows the clocked D latch.

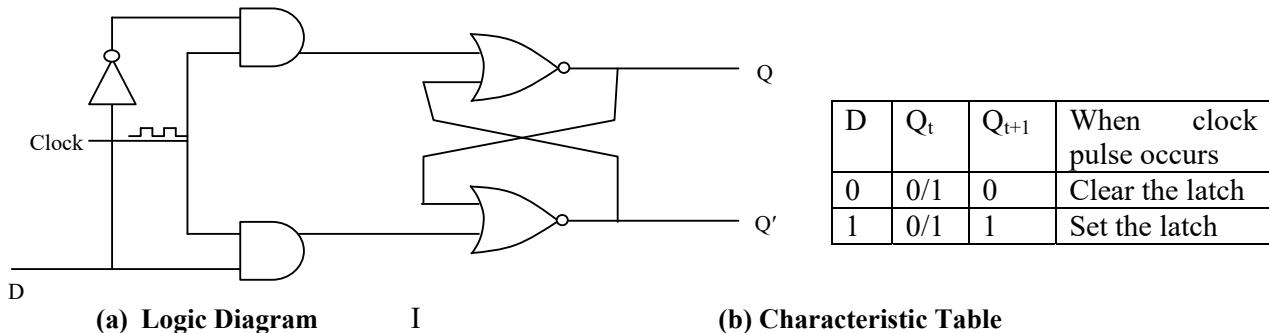


Figure 4.5: D latch with clock

You may please go through the circuit and identify various changes in Q, Q' with D as shown for SR latch.

4.3.2 Flip-Flops

Latches suffer from the problem due to frequent changes of output, e.g. the output of latch may change depending on the value of R and S input, which may change from 1 to 0 or vice-versa during a single clock pulse. Therefore, they are less suitable for sequential circuits. Flip-flops add more circuitry in latches so that changes in states occur during the rising or falling edge of clock pulse (these are called edge triggered flip-flop). R-S latch with clock can be used with additional circuits to make R-S flip-flop. The flip-flops can also be represented using a block diagram. Figure 4.6 shows the block diagram of basic flip-flops. Please note that in the block diagram the arrow head in front of the clock signal represents that the flip-flop will respond to input during the leading or rising edge (when transition from 0 to 1 takes place) of the clock

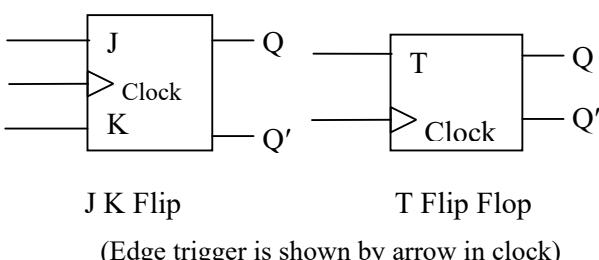
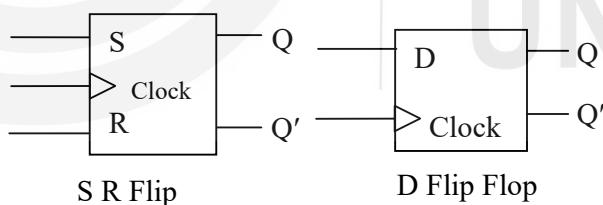


Figure 4.6: Graphical Symbols of basic Flip-Flops

JK flip is almost identical to SR flip-flop, except the last combination of $J = 1$ and $K = 1$ is used to complement the current state of the flip-flop. T-flip-flop is obtained by joining the J and K input, thus, it shows just two input values. When $T = 0$, there is no change of state and at $T = 1$, the current state is complemented. The following figure shows the characteristics table for the basic flip-flops shown in Figure 4.7

SR Flip-flop				JK Flip-Flop			
S	R	Q_{t+1}	Comments	J	K	Q_{t+1}	Comments
0	0	Q_t	No Change in state	0	0	Q_t	No Change in state
0	1	0	Clear state	0	1	0	Clear state
1	0	1	Set state	1	0	1	Set state
1	1	-	Not Defined	1	1	Q'_t	Complement of Q_t

D Flip-flop			T Flip-flop		
D	Q_{t+1}	Comments	T	Q_{t+1}	Comments
0	0	Clear State	0	Q_t	No Change in state
1	1	Set State	1	Q'_t	Complement of Q_t

Figure 4.7: Characteristic Table for flip-flops

4.3.3 Excitation Tables

The characteristic tables of flip-flops as shown in Figure 4.7 show how the state of flip-flop will change to the next state based on the present state and input values. The characteristic tables are used for analysis of the sequential circuits. While designing sequential circuits, you need to consider for what transition what possible input combinations would be required. This is done with the help of excitation table. Figure 4.8 shows excitation tables for the basic flip-flops.

Q_t	Q_{t+1}	J	K	Q_t	Q_{t+1}	S	R
0	0	0	X	0	0	0	X
0	1	1	X	0	1	1	0
1	0	X	1	1	0	0	1
1	1	X	0	1	1	X	0

(a) JK Flip flop

(b) SR Flip flop

Q_t	Q_{t+1}	D
0	0	0
0	1	1
1	0	0
1	1	1

(c) D Flip flop

Q_t	Q_{t+1}	T
0	0	0
0	1	1
1	0	1
1	1	0

(d) T Flip flop

X-denotes **DONOT CARE** condition.

Figure 4.8: Excitation Tables for basic flip-flops

Q_t and Q_{t+1} indicate present and next state of a flip flop, respectively. Symbol X in the table means do not care condition i.e. it does not matter whether the input is 0 or 1.

How these excitation tables are created? This is explained with an example of creation of excitation table of JK Flip flop.

- a) The state transition from $Q_t = 0$ to $Q_{t+1} = 0$
 - (i) As both Q_t and Q_{t+1} are 0 it means that there is no change in the state of flip flop, which can be achieved by $J=0$, $K=0$;
 - (ii) Using the input, $J=0$, $K=1$, the flip flop can be RESET, i.e. $Q_{t+1} = 0$.
- b) The state transition from $Q_t = 0$ to $Q_{t+1} = 1$
 - (a) Using the input, $J=1$, $K=0$, the flip flop is SET, i.e. $Q_{t+1} = 1$
 - (b) Using the input, $J=1$, $K=1$, the flip flop is complemented from Q_t having a value 0 to $Q_{t+1} = 1$
- c) State transition from $Q_t = 1$ to $Q_{t+1} = 0$
 - (a) Using the input, $J=0$, $K=1$, flip flop is RESET, i.e. $Q_{t+1} = 0$
 - (b) Using the input, $J=1$, $K=1$, the flip flop is complemented from Q_t having a value 1 to $Q_{t+1} = 0$
- d) For state transition from $Q_t = 1$ to $Q_{t+1} = 1$
 - (a) Using the input, $J=0$, $K=0$, no change in flip flop so $Q_{t+1} = 1$
 - (b) Using the input, $J=1$, $K=0$, flip flop is SET, i.e. $Q_{t+1} = 1$

These entire set of input for various transitions can be summarized in the table below:

Present State (Q_t)	Next State (Q_{t+1})	Input J and K	Input using DONOT CARE
0	0	(i) $J=0$, $K=0$ (ii) $J=0$, $K=1$	J=0, K=X
0	1	(i) $J=1$, $K=0$ (ii) $J=1$, $K=1$	J=1, K=X
1	0	(i) $J=0$, $K=1$ (ii) $J=1$, $K=1$	J=X, K=1
1	1	(i) $J=0$, $K=0$ (ii) $J=1$, $K=0$	J=X, K=0

The excitation table has been derived for J-K flip-flop as above. You may draw the excitation table for all other flip-flops using the same method.

Check Your Progress 1

- What is a sequential circuit? How are sequential circuits different from combinational circuits?

.....
.....
.....

- What is a latch? How is different from a flip-flop?

.....
.....
.....

- What is an excitation table? Draw the excitation table for SR, D and T flip-flops.

.....
.....

4.3.4 Master-Slave Flip-Flop

The master slave flip-flop is constructed using two or more latches. Figure 4.9 shows how two S-R flip-flops can be used to construct a master-slave flip-flop.

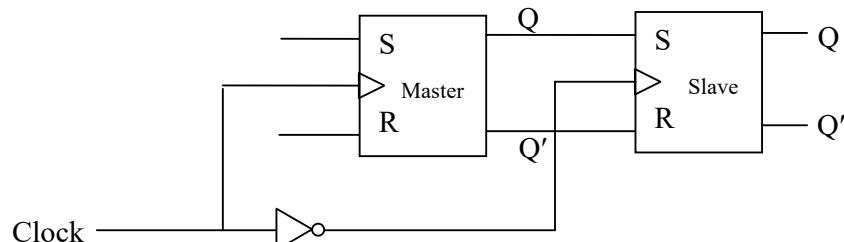


Figure 4.9: Master – Slave flip- flop

You may please note that you can construct a master-slave flip-flop using D or JK flip-flop also. This flip-flop consists of master which changes state when clock pulse occurs. The slave flip flop goes to the state of master flip-flop when the clock signal is 0. (Refer to figure 4.9) This is explained below:

The flip-flop operates in two steps:

- When a clock pulse input is 1: As this time the Master flip-flop, based on the value of S and R, goes to Set or Clear state as the case may be. At this time the slave flip-flop cannot change its state as it receives the inverse of clock pulse. Thus, on the occurrence of clock pulse ‘Master’ flip-flop goes to the next state (Q_{t+1}), whereas the output from slave flip-flop is the present state (Q_t).
- When the clock pulse input is 0: In this time the input to Master flip-flop will not have any effect on the Master flip-flop output, which has been put in the next state (Q_{t+1}) in the previous step. However, now this Q_{t+1} output of master flip-flop will be applied on the slave flip, which will result in transition of state of slave flip flop to Q_{t+1} . Thus, on completion of a clock cycle master and slave flip-flops both will be in Q_{t+1} . Please note that for slave flip flop only following transitions are possible:

Master output \equiv Slave input				Slave Output		
Q	Q'	S	R	Q	Q'	
1	0	1	0	1	0	(Set)
0	1	0	1	0	1	(Reset)

4.3.5 Edge-Triggered flip-flops

An edge-triggered flip-flop triggers the change either during the rising edge or positive transition (0 to 1 transition) or the falling edge or negative transition of the clock (1 to 0 transition). Fig 4.10 shows the clock pulse signal in positive & negative edge-triggered flip-flops.

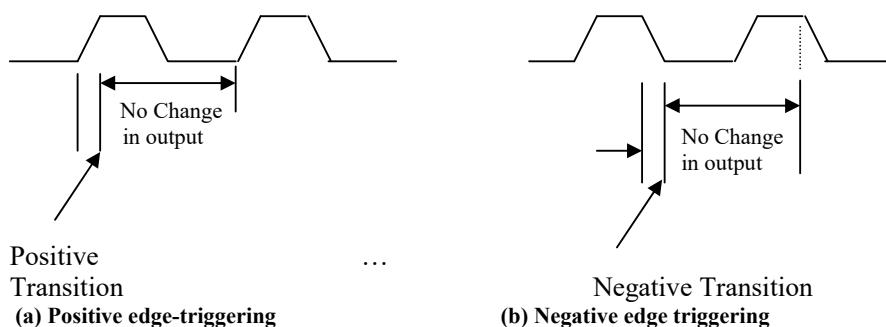
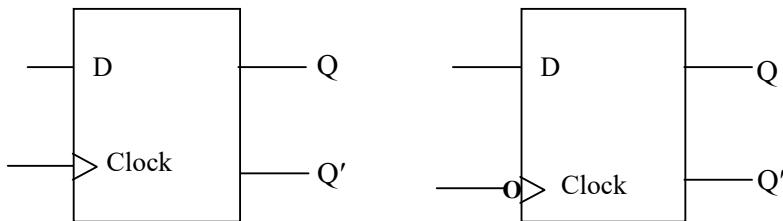


Figure 4.10: Clock Pulse Signal

The following figure shows the block diagram of edge triggered D flip-flop.



(a) Positive edge-triggered D flip-flop (b) Negative edge-triggered D flip-flop
Figure 4.11: Edge triggered and master slave D flip-flop

More detailed discussion on these flip-flops are beyond the scope of this unit. You may refer to further readings for the same.

Check Your Progress 2

1. List the advantages of master-slave flip-flop.

.....

.....

2. How edge-triggered flip-flops are different to master-slave flip-flops?

.....

.....

4.4 SEQUENTIAL CIRCUIT DESIGN

A sequential circuit not only consists of external input and external output, but also an internal state which is characterised by the state of flip flops internal to the circuit.

The state of sequential circuit changes as per its design based on some control signal like the clock control. Therefore, design of a sequential circuit is required to address the changes in the internal state of itself. Therefore, in addition to the logic circuit, sequential circuit design requires the information about the changes in state of flip-flops. The process of design of a sequential circuit is explained with the help of an example of design of a 2-bit counter circuit given below.

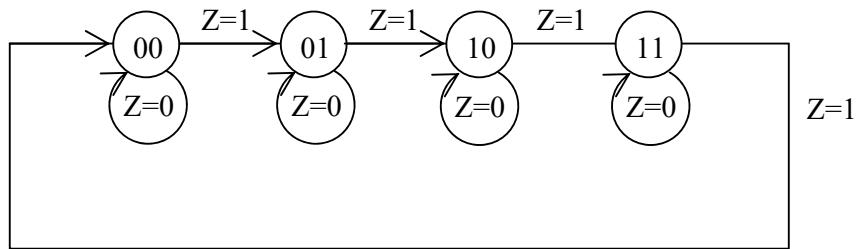
Example: Design a 2-bit counter circuit.

Solution: A counter is a special circuit which counts the timing sequences. A 2-bit counter will require two flip-flops. The state sequence of 2-bit counter would be 00, 01, 10, 11, 00 and so on. Thus, using a 2-bit counter, you can have 4 distinct internal states of the circuit and counter should move in each transition from one state to next as:

00 → 01 → 10 → 11 → 00

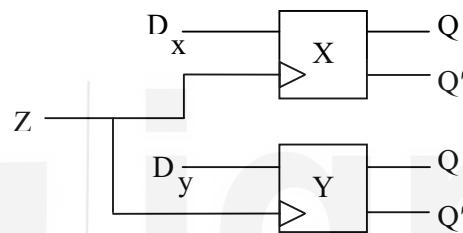
Assuming that these transitions are triggered by a control signal, say Z which can be a clock signal or any other signal generated for this purpose, a state change sequence can be presented as shown in the diagram given below:

State



This circuit uses two bits to store the state, therefore, requires two flip-flops. The state of the circuit changes to next state, when $Z=1$, else it stays in the same state. Thus, in this sequential circuit, you require 2 flip-flops and one control signal Z . But, what would be other input and output to this sequential circuit. Well! The other input will be the current states of flip-flops which will govern the next states of flip-flops.

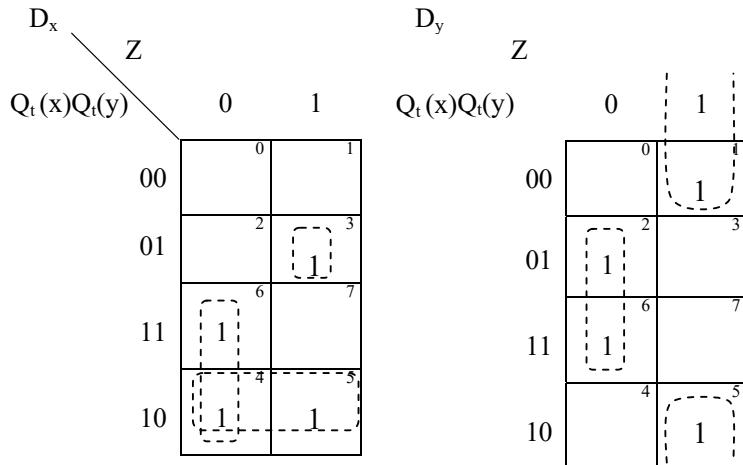
Next, you may take D flip-flop to design the circuit then a Rough design of the circuit would be:



In order to design the logic circuit, which generates the signal D_x and D_y , let us first draw a truth table for flip-flop's X and Y. This truth table is shown in the following table:

	Present States of Flip-Flops			Next State of Flip-Flops		Required value of D_x for transition of X and D_y for the transition of Y	
	Flip-flops		Input	Flip-flops		D_x	D_y
	Q_t of X	Q_t of Y		Q_{t+1} of X	Q_{t+1} of Y		
0	0	0	0	0	0	0	0
1	0	0	1	0	1	0	1
2	0	1	0	0	1	0	1
3	0	1	1	1	0	1	0
4	1	0	0	1	0	1	0
5	1	0	1	1	1	1	1
6	1	1	0	1	1	1	1
7	1	1	1	0	0	0	0

Interestingly, it is the D_x and D_y input that should be generated from the present state and Z input, so that the Next state (Q_{t+1}) of the flip-flops can be derived from the present state of the flip-flop (Q_t). Thus, for the design of counter circuit, you can draw K-map for the design of D_x and D_y with input $Q_t(X)$, $Q_t(Y)$ and Z . The K-maps for D_x and D_y can be drawn as:



$D_x = \text{Terms of (Adjacency of 4,5 + Adjacency of 4,6 + Cell 3)}$

$$D_x = Q_t(X) \cdot Q_t'(Y) + Q_t(X) \cdot Z' + Q_t'(X) \cdot Q_t(Y) \cdot Z$$

$D_y = \text{Terms of (Adjacency of 2,6 + Adjacency of 1,5)}$

$$D_y = Q_t(Y) \cdot Z' + Q'(Y) \cdot Z$$

Thus, the final 2-bit counter circuit will be drawn as shown in Figure 4.12

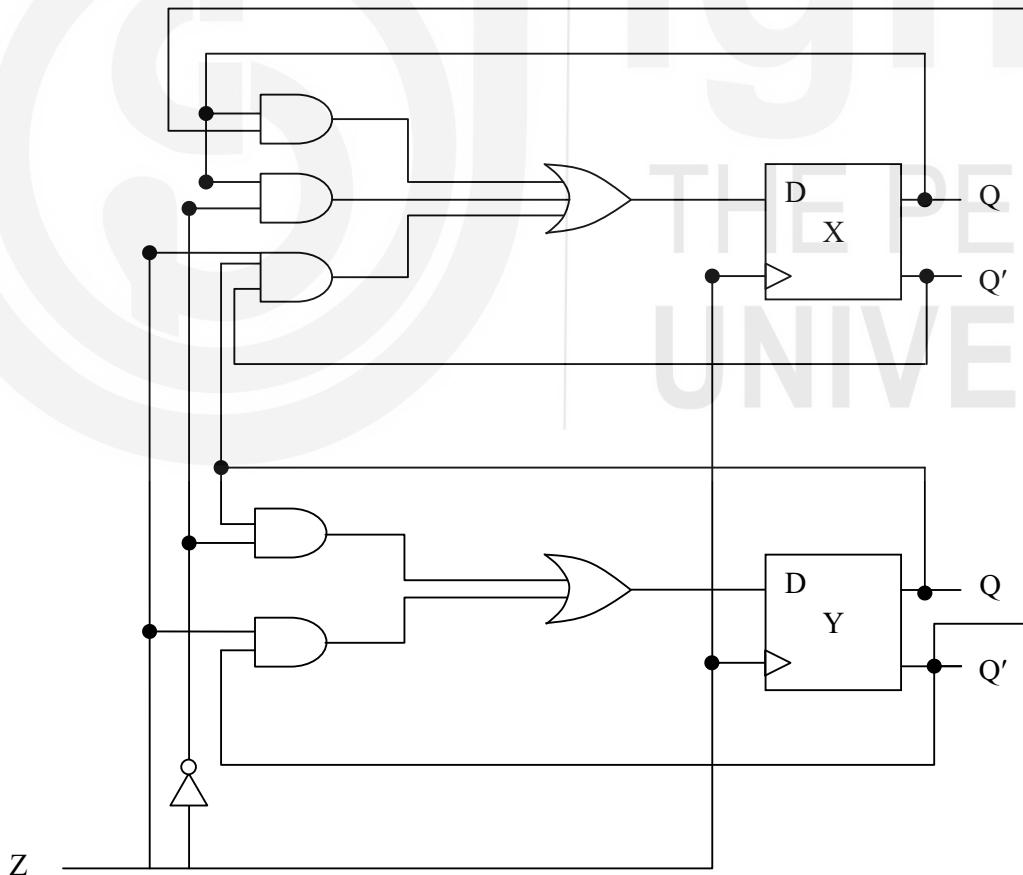


Figure 4.12: 2-bit counter

4.5 EXAMPLES OF SEQUENTIAL CIRCUITS

Let us now explain the basic function of some of the useful examples of sequential circuits like registers, counters etc.

4.5.1 Registers

Registers are the basic storage unit of a computer. Since register temporarily stores certain values, therefore, it requires flip-flops. The size of registers is computed using number of bits it stores. One bit storage requires, at least, one flip-flop. Thus, in general, an n bit register would use n flip-flops. Two common operations on register are:

- To load all bits of a register simultaneously or parallel load.
- Shifting of bits, of register, towards left or right

Figure 4.13 shows a parallel load register..

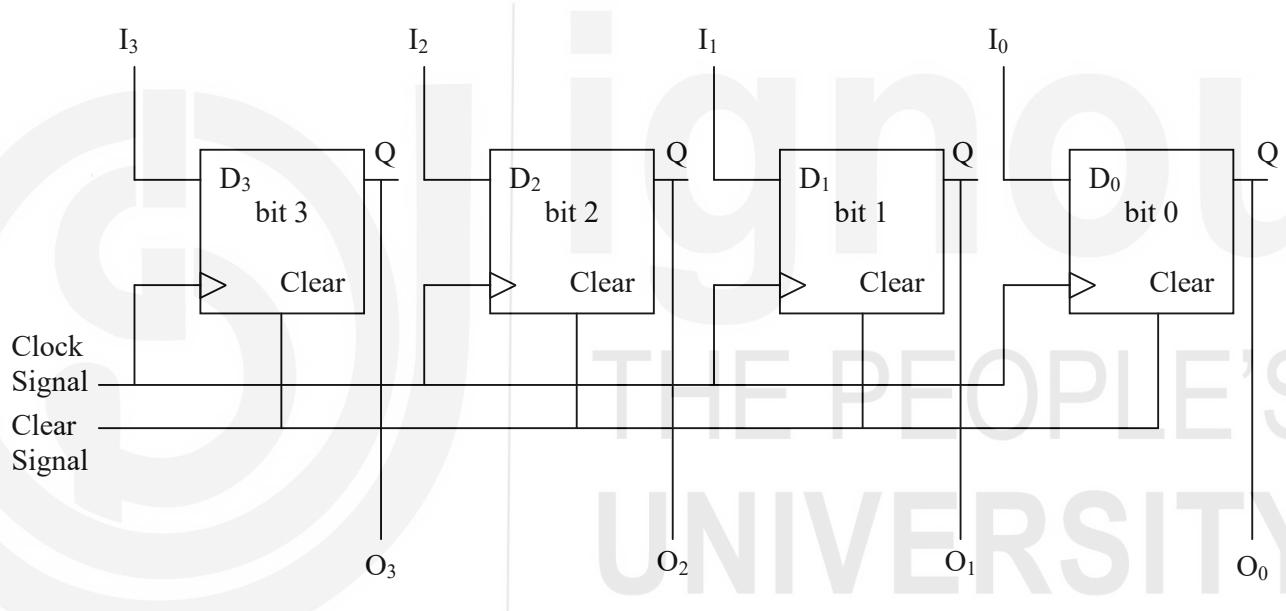


Figure: 4.13 A register with parallel load.

Please note the following point about the register circuit as above:

- (1) The 4-bit register is made up of 4 D flip-flops.
- (2) Clock signal is applied to all flip-flops simultaneously; therefore, loading operation will load the values I_3 , I_2 , I_1 , and I_0 respectively into the four flip-flops, simultaneously.
- (3) Special clear signal is used, which can clear all the bits of the register simultaneously, if needed.
- (4) The output of register O_3 , O_2 , O_1 , O_0 can be used for any arithmetic operations. Please note that registers output changes synchronously.

Shift register: Shift operation is very special operation for a computer ALU. A shift register is capable of shifting the content of a register either to left or to the right by one bit at a time. The following figure shows a right shift register, however, you can construct a left shift register in a similar manner.

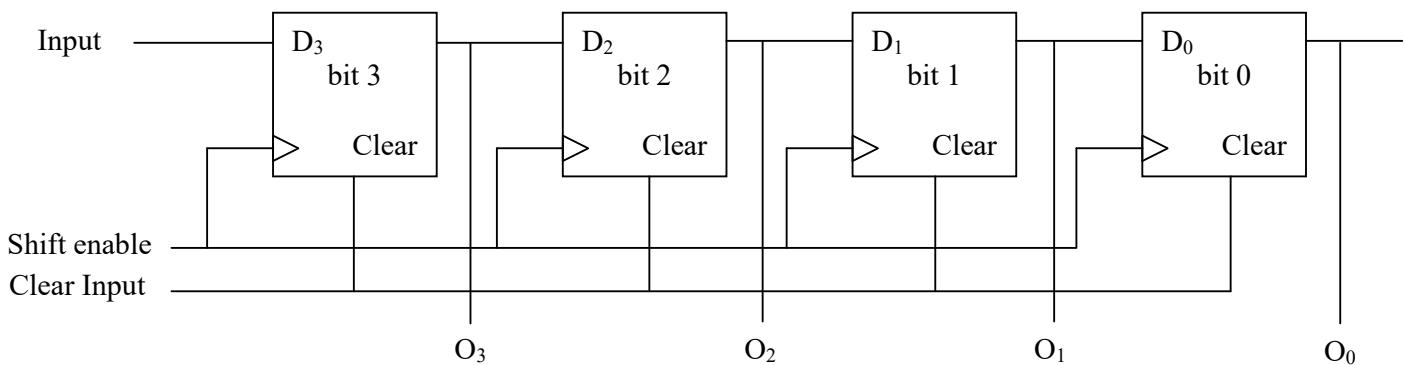
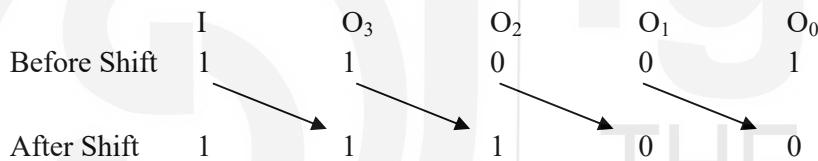


Figure 4.14: 4-bit Right Shift Register

Please note the following points.

- The external input is applied to D_3 . The output of D_3 is applied to D_2 , and so on.
- The shift enable is applied as clock input. It enables the shift operation.
- For example, assume the shift register had state $1 \ 0 \ 0 \ 1$ and input bit is 1, then after the right shift operation the output will change as:



A single registers can be included with the facility of left shift, right shift and parallel load. Such a register is called bi-directional shift register with parallel load. You may create its block diagram as an exercise.

4.5.2 Counters Circuit

Counters are sequential circuits, which produce output in a sequence on the occurrence of a transition signal. The counters may be used in keeping sequence such as steps of execution of a single instruction. There are two types of counters-asynchronous and synchronous.

In synchronous counter the flip-flop change their state one by one, while in synchronous counter all flip-flops may change the state simultaneously

Asynchronous Counter: An asynchronous counter is also called a ripple counter as the changes in the state of flip-flop is done one by one like a wave. Figure 4.17 shows a 3-bit ripple counter using T-flip flop (Please note the earlier 2-bit counter was designed using D flip-flop). These counters also have all clear input but for simplicity it is omitted in the figure.

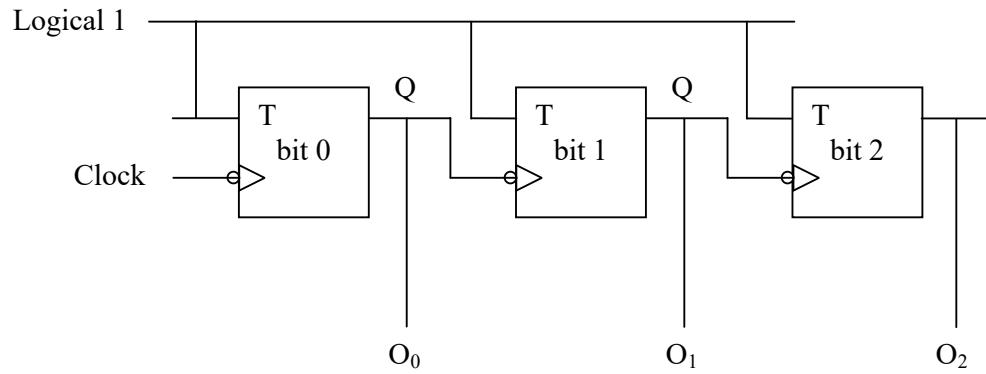
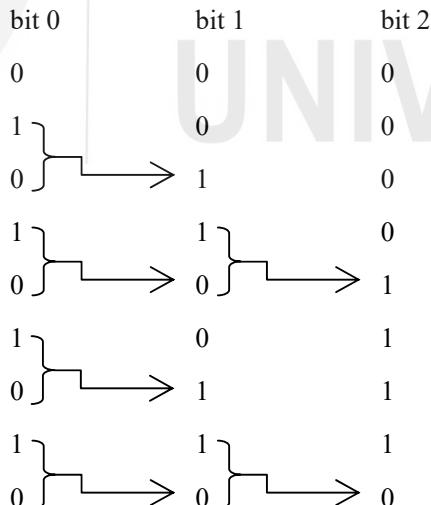


Figure: 4.15: 3-bit ripple counter

Please note the following points in the figure.

- (i) The bit 0 will complement each time a clock pulse occurs as it is connected to clock.
- (ii) For bit 1 flip flop the transition will be triggered, if Q_t of bit 0 flip-flop is 1. In that case Q_{t+1} of bit 1 will be complemented. This occurs because the transition signal of (bit 1) flip-flop is connected to Q_t output of bit 0 flip-flop. Similarly, the transition of bit 2 flip flop will occur, if Q_t of bit 1 flip-flop was in state 1.
- (iii) The transition is expected to occur with the falling edge (indicated by o before the clock input).
- (iv) Please note change in states would be as follows. Assuming initial state to be 0 0 0



} indicates the falling edge.

4.5.3 Synchronous Counter:

The flip-flops of the synchronous counter can change their state simultaneously. A 3-bit synchronous counter with rising edge of clock signal is shown in figure 4.16

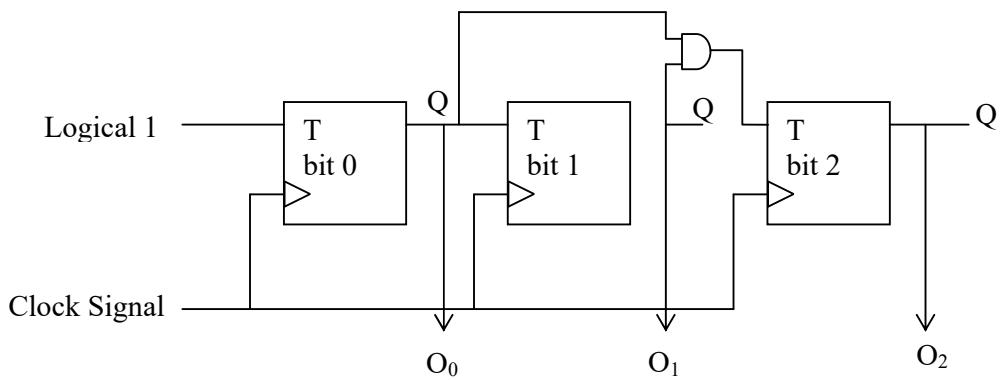


Figure 4.16: Synchronous Counter

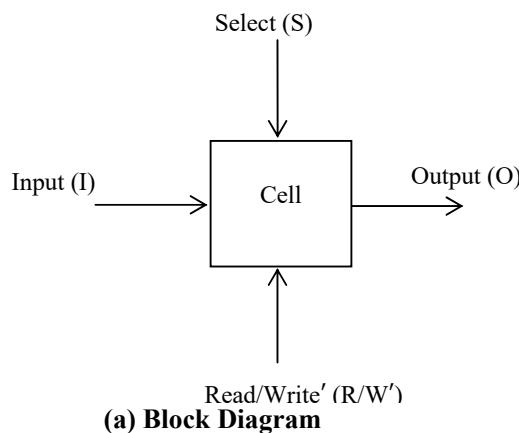
Please note in the figure above

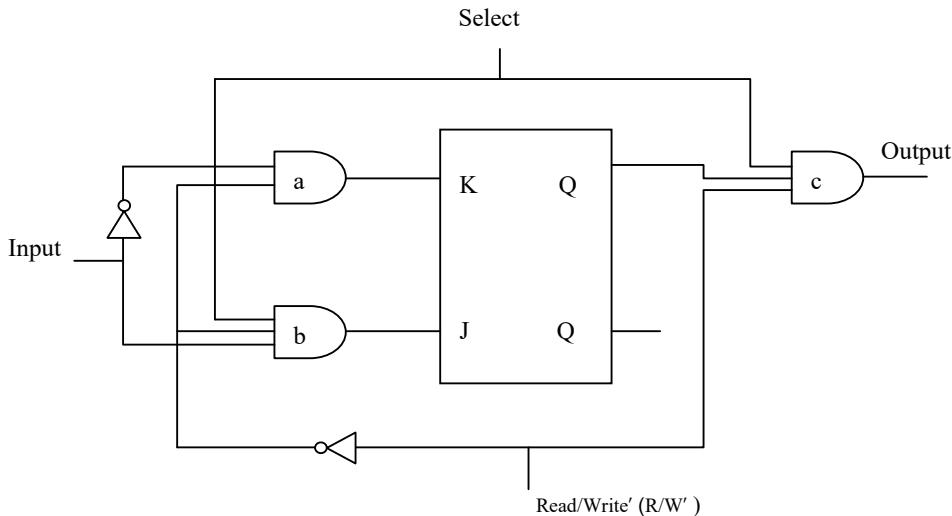
- (i) bit 0 is complement on occurrence of clock pulse
- (ii) bit 1 is complemented, if (Q_t of bit 0) was 1.
- (iii) bit 2 is complemented, if Q_t of bit 0 and Q_t of bit 1, both are 1.

Flip-Flop		
bit 0	bit 1	bit 2
0	0	0
1	0	0
0	1	0
1	1	0
0	0	1
1	0	1
0	1	1
1	1	1
0	0	0

4.5.4 Random Access Memory

In this section, a general configuration for flip-flop based random access memory (RAM) is proposed. A RAM essentially stores bits, therefore, it (especially DRAM technology) may be a sequential circuit. Two basic operations are performed on RAM: *Reading information from RAM*, this operation requires decoding operation, which identifies the cells or lines that are to be read; and *writing to RAM*, which in addition to identifying the cell, also requires changing the state of selected RAM flip-flops based on the input value. The figure 4.21 shows the block diagram and logic diagram and of a RAM cell, which is a single flip-flop.





(b) Logic Diagram

Figure 4.17: Binary Cell

A RAM cell as shown consists of one flip-flop. The behavior of this cell is exhibited in the following table.

(i) Read/Write' bit 1 \Rightarrow Operation is Read

Select bit	Read bit	Q_t	Output of cell (c flip-flop)
0	1	0/1	Not activated
1	1	0/1	Q_t

(ii) Read/Write' bit 0 \Rightarrow Write Operation

Assume Input bit to the circuit in Figure 4.17 (b) is I

Select Bit (S)	Input Bit (I)	Qt of Gate		Flip-flop Input		Q_{t+1}	Comments
		'a'	'b'	J	K		
0	-	Any	Any	0	0	Q_t	Not selected
1	0	0	1	0	1	0	Clear memory flip-flop
1	1	1	0	1	0	1	Set the memory cell

The write operation as shown in the table above changes the content of memory cell to the value of Input (I), or in other words memory cell has been written into by the value of input (I).

In addition to read/write' to memory cell, additionally a RAM is to be organized as an array of RAM cells, so as to decode the address of the cells, which is shown in Figure 4.18.

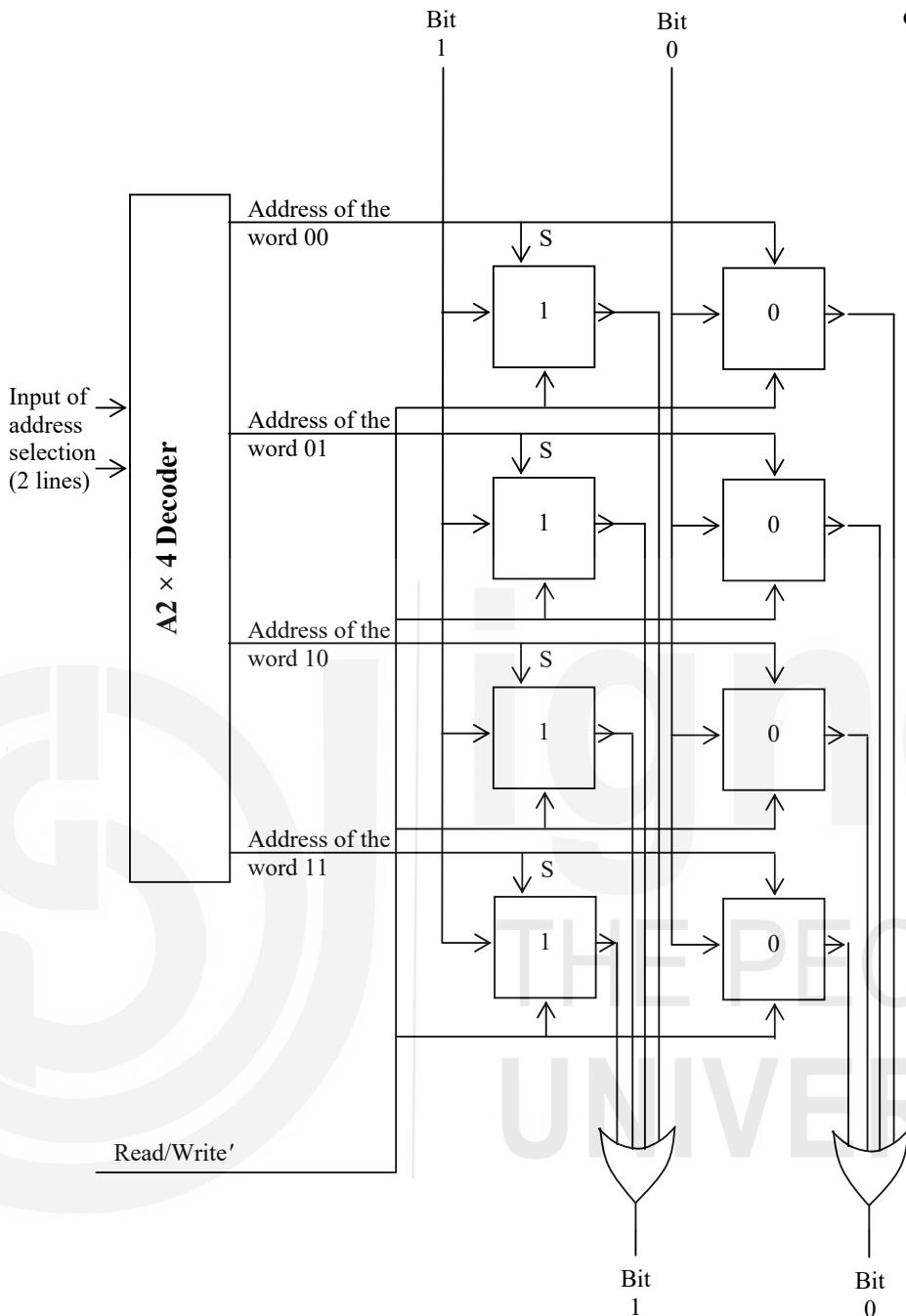


Figure 4.18: Two-dimensional Array based 4×2 RAM

The RAM has 4 words, which are decoded by the address decoder. Please note as there are 4 words or lines, therefore, you require 2×4 decoder. This logic can be extended, e.g. a RAM of size 1024×8 , would require 10×1024 decoder as $2^{10} = 1024$. So it will have 10 address lines which will decide which word of the RAM array is to be selected.

For this implementation, the number of bits stored in each word would be 2 only, that is why every memory line will have 2 cells. Please note that for a word size of 2 bits, the RAM array would require 2 input and 2 output lines.

For this memory array, in case an address 01 is given as input of address selection bits, it will activate the Select input of cells of address 01 for read or

write operation. Please note that current RAM chip design is not a 2 dimensional design as shown in Figure 4.18. It may follow a different more optimal organization, discussion on which is beyond the scope of this unit.

Check Your Progress 3

- 1) What are the differences between synchronous & asynchronous counters?

.....
.....
.....

- 2) Is ripple counter same as shift register?

.....
.....
.....

- 3) Design a two bit counter, which has the states 00, 01, 10, 00, 01, 10....

.....
.....
.....

4.6 SUMMARY

This unit introduces you the concepts of sequential circuits which is the foundation of digital design. Flip-flops are also a sequential circuit and the basic storage unit of a computer system. This unit also explains the working of a latch, which is the basic circuit that can be used for storing one bit of information. The sequential circuit can be formed using combinational circuits (discussed in the last unit) and flip flops. The unit also discusses the construction of some of the important sequential circuits like registers, counters, RAM. For more details, the students can refer to further reading.

4.7 SOLUTIONS / ANSWERS

Check Your Progress 1

1. A sequential circuit is designed to process and store data. Therefore, it consists of flip-flops for storing a state representing 0 or 1 and additional combinational circuit that may result in change of state depending on the combinational logic. Example of sequential circuits are - registers, counters etc. The main difference is that a sequential circuit also has a state.
2. Latch is a basic asynchronous sequential circuit designed with feedback to exhibit two different states, viz. 0 or 1. These states can be modified as per the input to latch. Example of latch is SR latch. Latches can change their state at any point of time based on input, whereas flip-flops are designed to change their states at specific time, for example on the occurrence of a clock pulse. Therefore, flip-flops have more complex circuitry than latch.
3. Excitation table are used for analysis and design of sequential circuits. They represent different combination of input to a flip-flop that may cause a specific state transition in the flip-flop. The following are the excitation tables of different flip-flops:

SR Flip-flop

Present State (Q _t)	Next State (Q _{t+1})	Input S and R	Input using DONOT CARE
0	0	(i) S=0, R=0 (ii) S=0, R=1	S=0, R=X
0	1	S=1, R=0	S=1, R=0
1	0	S=0, R=1	S=0, R=1
1	1	(i) S=0, R=0 (ii) S=1, R=0	S=X, R=0

D Flip-flop

Present State (Q _t)	Next State (Q _{t+1})	Input D	Input using DONOT CARE
0	0	D=0	D=0
0	1	D=1	D=1
1	0	D=0	D=0
1	1	D=1	D=1

T Flip-flop

Present State (Q _t)	Next State (Q _{t+1})	Input D	Input using DONOT CARE
0	0	T=0	T=0
0	1	T=1	T=1
1	0	T=1	T=1
1	1	T=0	T=0

Check Your Progress 2

- The master-slave flip flop is a simple structure, which changes its output during the clock pulse, when it is 0, thus, will result in synchronous state transitions of flip-flop.
- An edge-triggered flip-flop changes its state either during the rising or falling edge of the clock pulse, thus, has a different construction than master-slave flip-flop.

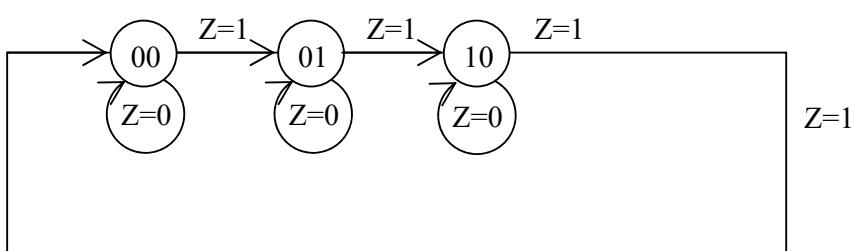
Check Your Progress 3

- All the flip-flops in the synchronous counter may change their state simultaneously, whereas in asynchronous counter, change of state of previous flip-flop may cause that effect to take place.
- No, shift register causes shifting of state of a flip-flop to next flip-flop, whereas ripple counter is governed by the change of state.
- The states 00, 01, 10, 00, 01, 10.....

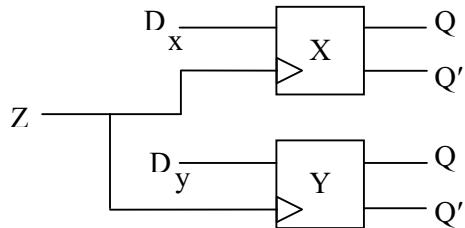
$$00 \longrightarrow 01 \longrightarrow 10 \longrightarrow 00 \longrightarrow 01$$

Assuming the control signal, say Z, state transitions are:

State



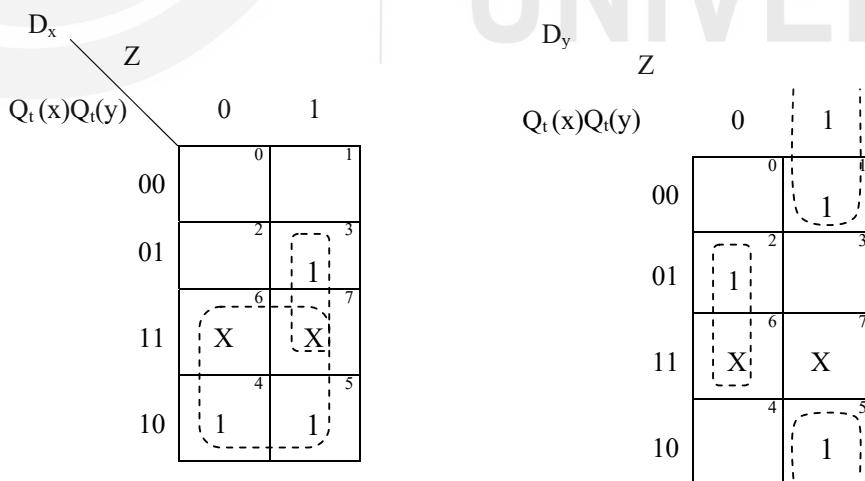
Rough design of the circuit would be:



Truth table for flip-flop's X and Y:

	Present States of Flip-Flops			Next State of Flip-Flops		Required value of D_x for transition of X and D_y for the transition of Y	
	Flip-flops		Input	Flip-flops			
	Q_t of X	Q_t of Y	Z	Q_{t+1} of X	Q_{t+1} of Y	D_x	D_y
0	0	0	0	0	0	0	0
1	0	0	1	0	1	0	1
2	0	1	0	0	1	0	1
3	0	1	1	1	0	1	0
4	1	0	0	1	0	1	0
5	1	0	1	0	0	1	1
6	1	1	0	-	-	X	X
7	1	1	1	-	-	X	X

The K-maps for D_x and D_y can be drawn as:



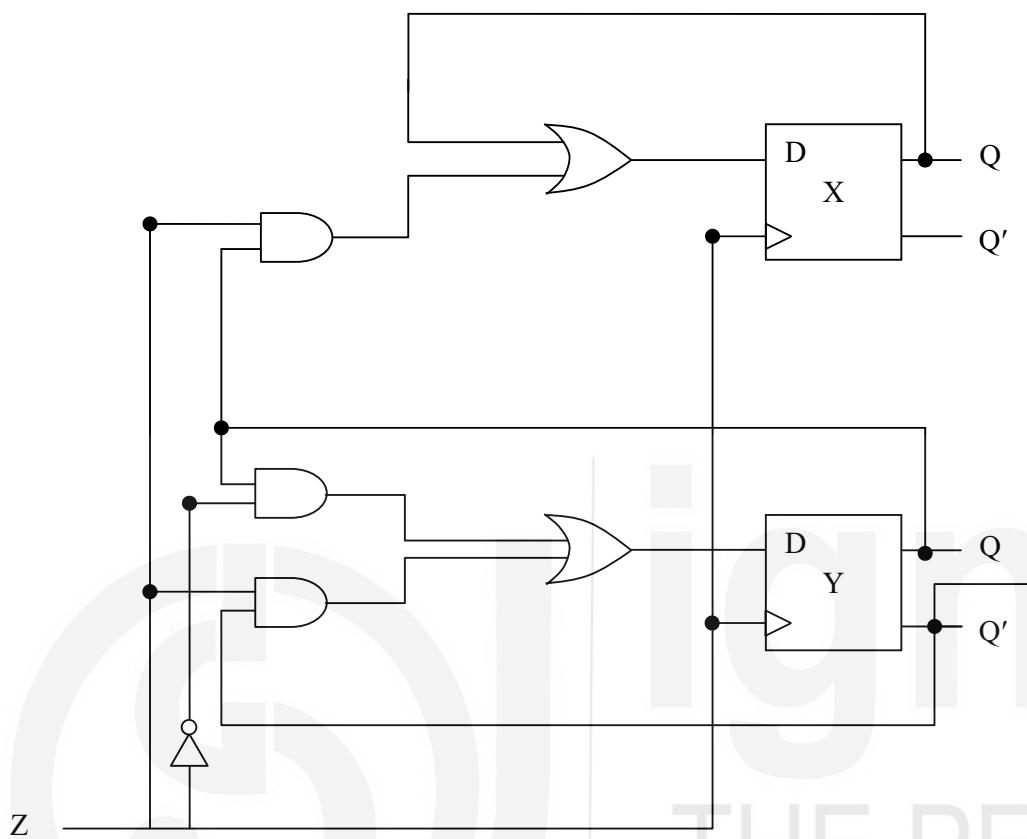
$$D_x = \text{Terms of (Adjacency of 4,5 + Adjacency of 3,7)}$$

$$D_x = Q_t(X) + Q_t(Y) \cdot Z$$

$$D_y = \text{Terms of (Adjacency of 2,6 + Adjacency of 1,5)}$$

$$D_y = Q_t(Y) \cdot Z' + Q'(Y) \cdot Z$$

Thus, the final counter circuit for the given states would be:



Indira Gandhi
National Open University
School of Computer and
Information Sciences

Block**2****Memory and Input/Output Organisation****UNIT 5****The Memory System**

UNIT 6**Advance Memory Organisation**

UNIT 7**Input/Output Organisation**

UNIT 8**Device Technology**

FACULTY OF THE SCHOOL

Prof P. V. Suresh, Director
Dr Shashi Bhushan
Mr M. P. Mishra

Prof. V. V. Subrahmanyam
Mr Akshay Kumar
Dr Sudhansh Sharma

PROGRAMME/COURSE DESIGN COMMITTEE

Shri Sanjeev Thakur
Amity School of Computer Sciences,
Noida
Shri Amrit Nath Thulal
Amity School of Engineering and Technology
New Delhi
Dr. Om Vikas(Retd),
Ministry of ICT, Delhi
Shri Vishwakarma
Amity School of Engineering and
Technology New Delhi
Prof (Retd) S. K. Gupta, IIT Delhi
Prof. T.V. Vijay Kumar, SC&SS, JNU,
New Delhi
Prof. Ela Kumar, CSE, IGDTUW, Delhi

Prof. Gayatri Dhingra, GVMITM, Sonipat
Sh. Milind Mahajani
Impressico Business Solutions, Noida, UP
Prof. V. V. Subrahmanyam
SOCIS, New Delhi
Prof. P. V. Suresh
SOCIS, IGNOU, New Delhi
Dr. Shashi Bhushan
SOCIS, IGNOU, New Delhi
Shri Akshay Kumar,
SOCIS, IGNOU, New Delhi
Shri M. P. Mishra,
SOCIS, IGNOU, New Delhi
Dr. Sudhansh Sharma,
SOCIS. IGNOU. New Delhi

BLOCK PREPARATION TEAM

Dr Zahid Raja (*Content Editor*)
Jawaharlal Nehru University
New Delhi

Mr Vikas Mittal (*Course Writer – Unit 5 & 6*)
Maharaja Agrasen College,
University of Delhi
Delhi

Dr Mohammad Sajid (*Course
Writer – Units 7 & 8*)
Department of Computer
Science
Aligarh Muslim University
Aligarh

(*Language Editor*) School of Humanities
IGNOU

Course Coordinator: Mr Akshay Kumar

PRINT PRODUCTION

March, 2021

© Indira Gandhi National Open University, 2021

All rights reserved. No part of this work may be reproduced in any form, by mimeograph or any other means, without permission in writing from the Indira Gandhi National Open University.

Further information on the Indira Gandhi National Open University courses may be obtained from the University's office at Maidan Garhi, New Delhi-110068.

Printed and published on behalf of the Indira Gandhi National Open University, New Delhi by the MPDD, IGNOU, New Delhi

Laser Typesetting : Akashdeep Printers, 20-Ansari Road, Daryaganj, New Delhi-110002

Printed at :

BLOCK 2 INTRODUCTION

In the first block this course, you are introduced to basic computer architectures, instruction execution, data representation and logic circuits of a computer system. This Block covers two of the most important units that are part of computer system architecture, viz. the memory system and Input/output system. This block consists of 4 units (unit 5 to unit 8).

Unit 5 explains the basic memory hierarchy of a computer system. It also explains different kinds of memories used in a computer system. The logic of RAM and ROM has also been explained in details. The unit also discusses secondary memories like magnetic and optical disks in details.

Unit 6 provides details on advance memory organisation. It provides details on different aspects of cache memory and main memory to cache mapping schemes. This unit also discusses the concept of interleaved memory, associative memories and virtual memories used in a computer system.

Unit 7 explains the basic interfaces and mechanisms that are used to perform input and output. This unit also presents the concept of DMA and input/output processor.

Unit 8 introduces you to basic technology of some of the popular input/output devices including keyboard, mouse, monitor, printer etc.

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

Further Readings For The Block

- 1) Mano M Morris, *Computer System Architecture*, 3rd Edition/Latest Edition, Prentice Hall of India Publication, Pearson Education Asia
- 2) Stallings W., *Computer Organization & Architecture: Designing For Performance*, 10th/11th Edition, Pearson Education Asia
- 3) Hennessy/Patterson, *Computer Organization and Design : The Hardware/ Software Interface*; 5th/6th Edition, Morgan Kaufmann.

UNIT 5 THE MEMORY SYSTEM

Structure	Page Nos.
5.0 Introduction	
5.1 Objectives	
5.2 The Memory Hierarchy	
5.3 SRAM, DRAM, ROM, Flash Memory	
5.4 Secondary Memory and Characteristics	
5.4.1 Hard Disk Drives	
5.4.2 Optical Memories	
5.4.3 Charge-coupled Devices, Bubble Memories and Solid State Devices	
5.5 RAID and its Levels	
5.6 Summary	
5.7 Answers	

5.0 INTRODUCTION

In the previous block, fundamentals of a computer system were discussed. These fundamentals included discussion on von-Neumann architecture based machines, instruction execution, representation of digital data and logic circuits etc. This Block explains the most important component of memory and Input/output systems of a computer. This unit covers the details of the Memory. This unit discusses issues associated with various components of the memory system, the design issues of main memory and the secondary memory. Various characteristics of secondary memory and its types that are used in a computer system, would also be discussed. The unit also defines how multiple disks can be used to create a redundant array of disks that can be used to provide a faster and reliable storage.

5.1 OBJECTIVES

After going through this Unit, you will be able to:

- explain the key characteristics of various types of memories and memory hierarchy;
 - explain and differentiate among various types of random access memories;
 - explain the characteristics of secondary storage devices and technologies;
 - explain the latest secondary storage technologies;
 - identify the various levels of RAID technologies
-

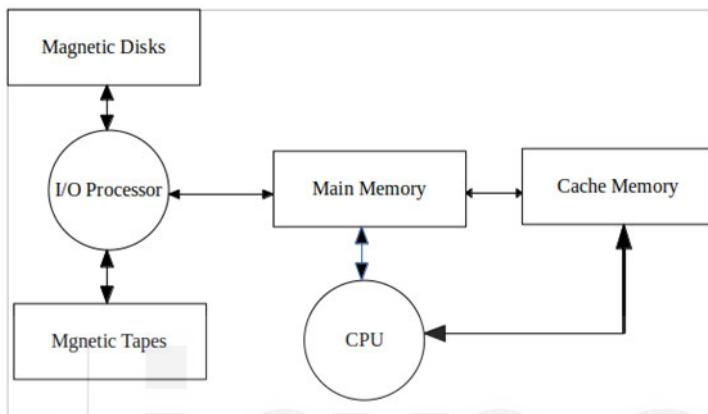
5.2 THE MEMORY HIERARCHY

In computers, memory is a device used to store data in binary form. Smallest unit of binary data is called ‘bit’. Each bit of binary data is stored in a different cell or storage unit and collection of these cells is defined as the memory. A memory system is composed of a memory of fixed size and procedures which tells how to access the data stored in the memory. Based on the persistence of the stored data, memory is classified into two categories:

- Volatile memory: which loses its data in the absence of power.
- Non-volatile memory: Do not lose data when power is switched off.

Another classification of memory devices, which is also the objective of this unit is based on the way they interact with the CPU which can be determined from figure 5.1 Main/ Primary memory interact directly with the CPU e.g. RAM and ROM.

Auxiliary/ secondary memory need I/O interface to interact with the CPU e.g. magnetic disks and magnetic tapes. There are other memories like cache and registers, which directly interacts with the CPU. Such memories are used to speed up the program execution. For execution, a program must be loaded into the main memory and should be stored on the secondary storage when it completes its execution. Auxiliary memory is used as a backup storage, whereas main memory contains data and program only when it is required by the CPU.



Various memory devices in a computer system forms a hierarchy of components which can be visualised in a pyramidal structure as shown in Figure 5.2. As you can observe in the Figure 5.2 that at the bottom of the pyramid, you have magnetic tapes and magnetic disks; and registers are at the top of the pyramid. Main memory lies at the middle as it can interact directly with the CPU, cache memory and the secondary memory. As you go up in the pyramid, the size of the memory device decreases, the access speed, however, increases and cost per bit also increases. Different memories have different access speeds. CPU registers or simply registers are fastest among all and are used for holding the data being processed by the CPU temporarily but because of very high cost per bit they are limited in size. Instruction execution speed of the CPU is very high as compared to the data access speed of main memory. So, to compensate the speed difference between main memory and the CPU, a very high speed special memory known as cache is used. The cache memory stores current data and program plus frequently accessed data which is required in ongoing instruction execution.

You may note the following points about memory hierarchy:

- ✓ The size of the memory increases as you go down the memory hierarchy.
- ✓ The cost of per unit of memory increases as you go up in the memory hierarchy i.e. Memory tapes and auxiliary memory are the cheapest and CPU Registers are the costliest amongst the memory types.
- ✓ The amount of data that can be transferred between two consecutive memory layers at a time decreases as you move up in the pyramid. For example, from main memory to Cache transfer one or few memory words of size in Kilobytes are accessed at a time, whereas in a hard disk to main memory transfer, a block data of size of 1 Megabyte is transferred in a single access.
- ✓ One interesting question about the memory hierarchy is why having faster smaller memories does not slow down the computer? This is primarily due to the

fact that there is very high probability that a program may access the instructions and data in the closed vicinity of presently executing instruction and data. This concept is further explained in next unit.

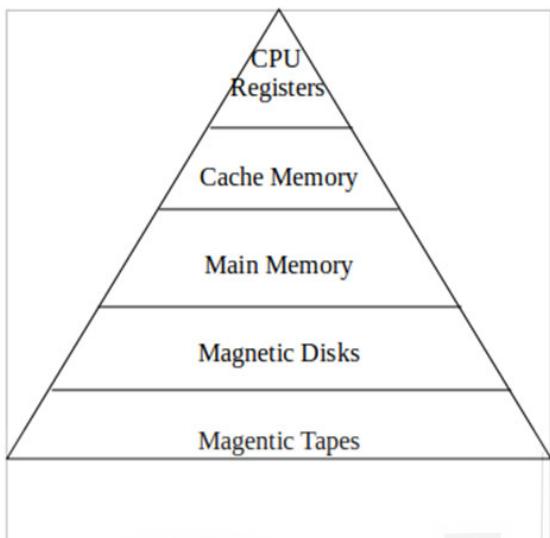


Figure 5.2. Memory hierarchy

In subsequent sections and next unit, we will discuss various types of memories in more detail.

5.3 SRAM, DRAM, ROM, FLASH MEMORY

The main memory is divided into fixed size memory blocks called *words*. Size of the memory word may be limited by the communication path and the processing unit size. As word size/ length denotes the amount of bits that can be processed by the processor at one time. Each memory word is addressed uniquely in the memory. A 32-bit processor uses a word size of 32 bits whereas 64-bit processor uses a word of 64 bits. RAM (random access memory) is a volatile memory i.e. content of the RAM vanishes when power is switched off. RAM is a major constituent of the main memory. Both read and write operations can be performed on RAM, therefore, it is also known as read-write memory. Access time of each memory word is constant in random access memory. RAM can be constructed from two types of technologies - Static Random Access Memory (SRAM) and Dynamic Random Access Memory (DRAM). The main difference being that DRAM loses its content even if power is on, therefore requires refreshing of stored bits in DRAM. Thus, DRAM is slower than SRAM, however, the DRAM chips are cheaper. In general, DRAM is used as the main memory of the computer, while SRAM is used as the Cache memory, which is discussed in details in the next unit.

SRAM

SRAM can be constructed using flip-flops. It is a sequential circuit. A SRAM cell using SR flip flop is shown in figure 5.3. As you can observe, this sequential circuit has three inputs: *select*, *read/write*, and *input* and single output: *output*. When *select* input is high "1" circuit is selected for read/write operation and when *select* input is low "0" neither read nor write operation can be performed by the binary cell. Thus, *select* input must be high in order to perform read/write operation by the binary cell. Binary cell reads a bit when *read/write* input is low "0" and writes when *read/write* input is high "1". Third input *input* is used to write into the cell. The only caution over here is that when *read/write* input is low "0" i.e. we want to perform a read operation, then read operation must not be affected by the input *input*. This is ensured by

inverted *input* to the first AND gate which guarantees the input to both R and S to be low and thus prevents any modification to the flip flop value. The characteristic table of SR flip flop is given in Unit 4 Block 1 for better understanding of the functioning of the binary cell.

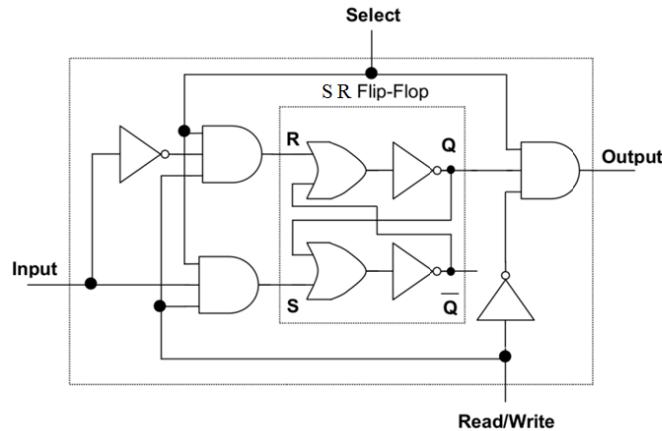


Figure 5.3: Logic Diagram of RAM cell

Read operation: select is high “1”, read/write is low “0” and *input* is either low “0” or high “1” then input to R and S will be 0 and flip flop will keep its previous state and that will be the output.

Write operation: select is high “1”, read/write is high “1” and if *input* is low “0” then R will be high “1” and S will be low “0” and flip flop will store “0” and if *input* is high “1” then R will go low “0” and S will go high “1” and flip flop will store “1”.

A RAM chip is composed of several read/write binary cells. A block diagram of $2^m \times n$ RAM is shown in Figure 5.4. The RAM shown has a total capacity of 2^m words and each word is n bits long e.g. in 64×4 RAM, the RAM has 64 words and each word is 4 bits long. To address 64 i.e. 2^6 words, we need 6 address lines. So in a $2^m \times n$ RAM, we have 2^m words where each word has n bits and RAM has m-bit address which requires m address lines. The RAM is functional only when chip select (CS1) signal = 1 and $\overline{CS2} = 0$. If chip select signal is not enabled or chip select signal is enabled and neither read nor write input is enabled then data bus will be in high impedance state and no operation can be performed. During high impedance state, other input signals will be ignored which means output has no logical significance and does not carry a signal

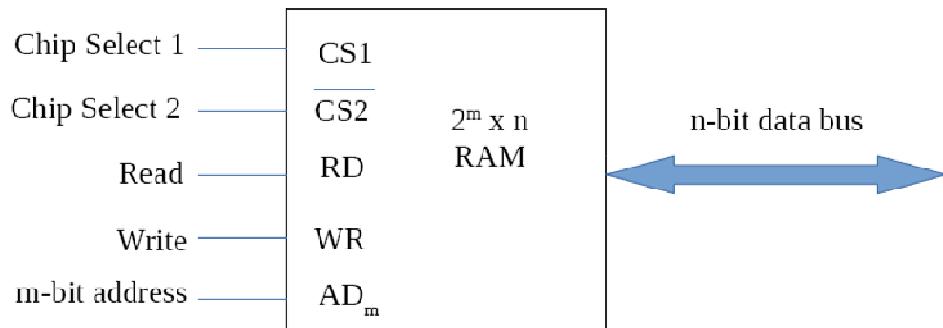


Figure 5.4: Block Diagram of $2^m \times n$ RAM

DRAM

Dynamic Random Access Memory (DRAM) is a type of RAM which uses 1 transistor and 1 capacitor (1T1C cell) for storing one bit. A block diagram of a single DRAM cell is shown in Figure 5.5. In DRAM, transistor is used as a gate which opens and closes the circuit and thus stops and allows the current to flow. Charging level of the capacitor is used to represent the bit “1” and bit “0”. As capacitors tends to discharge in a very short time period DRAM cells need to be refreshed periodically to store the binary information despite continuous power supply. Hence they are called dynamic random access memory. With low power consumption and very compact in size because of 1T1C architecture DRAM offers larger storage capacity in a single chip. Each DRAM cell in the memory is connected with Word Line (Rows) and Bit Line (Columns) as shown in Figure 5.5. Word line (rows) controls the gates of the transfer lines while Bit lines (columns) are connected to sense amplifiers i.e. to determine “0” or “1”.

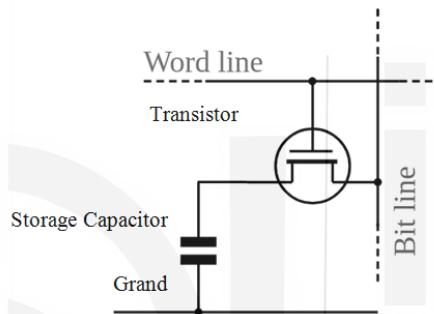


Figure 5.5: A DRAM cell

Figure 5.6 presents the general block diagram of $2^M \times 2^M \times N$ DRAM, where binary cells are arranged in a square of $2^M \times 2^M$ words of N bit each. For example, 4 megabit DRAM is represented in a square arrangement of (1024×1024) or $(2^{10} \times 2^{10})$ words of 4 bit each. Thus, in the given example we have 1024 horizontal/ word lines and 1024×4 column/ bit lines. In other words, each element, which consists of 4 bits of array, is connected by horizontal row lines and vertical column lines.

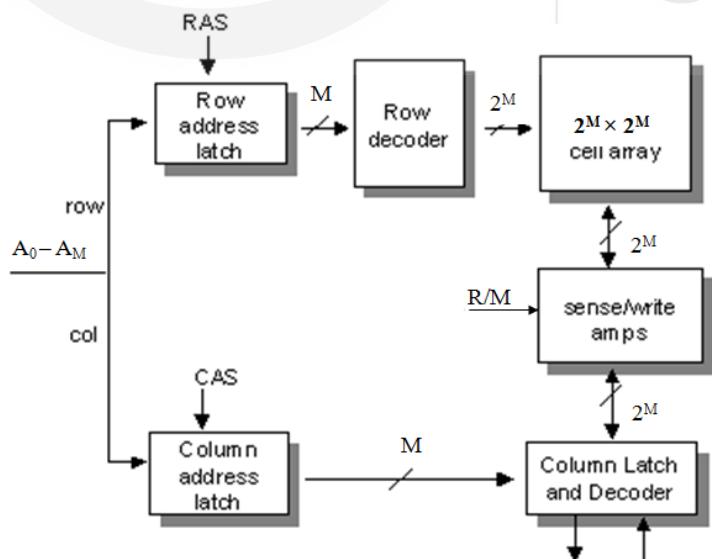


Figure 5.6: Block Diagram of DRAM

Selection and role of various signals for read and write operation is as follows:

1. RAS (Row Address Strobe): On the falling edge of RAS signal, it opens or strobe the address lines (rows) to be addressed.
2. /CAS (Column Address Strobe): Similar to /RAS, on the falling edge, this enables a column to be selected as mentioned in the column address from the rows opened by the /RAS to complete the read-write operation.
3. R(/W), Write enable: This signal determines whether to perform a read operation or a write operation. While the signal is low, write operation is enabled and data input is also captured on falling edge of /CAS whereas high enables the read operation.
4. Sense amplifier compares the charge of the capacitor to a threshold value and returns either logic “0” or logic “1”.

For a read operation once the address line is selected, transistor turns ON and opens the gate for the charge of the capacitor to move to the bit line where it is sensed by the sense amplifier. Write operation is performed by applying a voltage signal to the bit line followed by the address line allowing a capacitor to be charged by the voltage signal.

ROM (Read-Only Memory)

Another constituent of the main memory is ROM (read only memory). Unlike RAM, which is read-write memory and volatile, ROM's are read only and non-volatile memory i.e. content of the ROM persist even if power is switched-off. Once data is stored at the time of fabrication, it cannot be modified. This is why, ROM is used to store the constants and the programs that are not going to change or get modified during their lifetime and will reside permanently in the computer. For example, bootstrap loader, which loads the part of the operating system from secondary storage to the main memory and starts the computer system when power is switched on, is stored in ROM.

A block diagram of $2^m \times n$ ROM looks similar to that of RAM. As ROM is a read-only memory there is no need of explicit read and write signals. Once the chip is selected using chip select signals a data word is read and placed on to the data bus. Hence, in the case of ROM, you need an unidirectional data bus i.e. only in output mode as shown in figure 5.7. Another interesting fact about ROM is that, ROM offers more memory cells and thus, memory as compared to the RAM for same size chip.

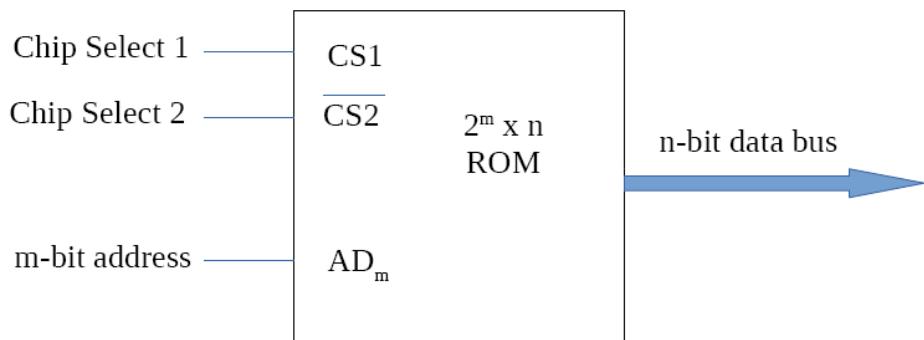


Figure 5.7: Block Diagram of $2^m \times n$ ROM

As shown in Figure 5.7, $2^m \times n$ ROM has 2^m words of n bits each for which it has m address lines and n output data lines. For example, in 128×8 ROM, you have 128 memory words of 8-bit each. For 128×8 ROM i.e. $2^m = 2^7$, $m = 7$, you need 7 address lines (minimum number of bits required to represent 128) and 8-bit output data bus.

Figure 5.8 shows a 32×8 ROM.

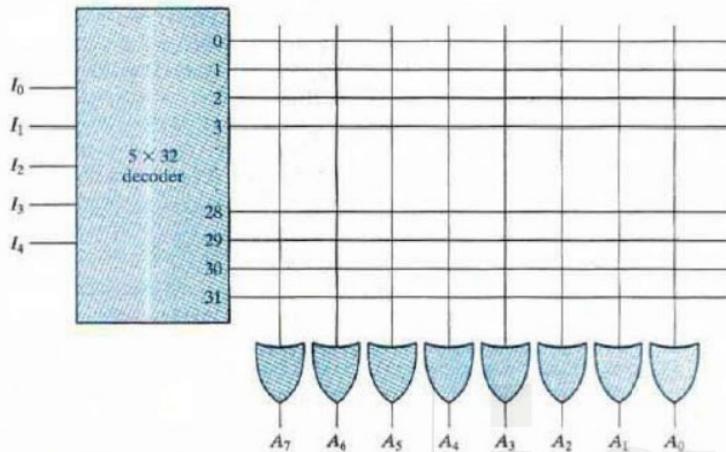


Figure 5.8: Internal diagram of 32×8 ROM

Unlike RAMs, which are sequential circuits, ROMs are combinational circuits. Typically, to design a RAM of specific size you need a decoder and OR gates. For example, to design a ROM of size 32×8 bits you need a decoder of size 5×32 and 8 OR gates. 5×32 decoder will have 5 input lines, which will act as 5 address lines of the ROM, the decoder will convert 5-bit input address to 32 different outputs. Figure 5.8 shows the construction of 32×8 ROM using 5×32 decoder and eight OR gates for data output. ROMs of other sizes can be constructed similarly. For example, to construct a ROM of 64×4 ROM, you need a 6×64 decoder and four OR gates and to construct a ROM of size 256×8 , you need 8×256 decoder and 8 OR gates.

As discussed, ROMs are non-volatile memory and content of the ROM once written, cannot be changed. Therefore, ROMs are used to store the look-up tables for constants to speed up the computation. In addition, ROM can store the boot loader programs and gaming programs. All this requires, zero error in writing of such programs and therefore, ROM device fabrication requires very high precision. Constructing a ROM, as shown in figure 5.8, requires decision about which interconnections in the circuit should be open and which interconnections should be closed. There are four ways you can program a ROM which are as follows:

1. **Mask ROM (MROM):** Masking of ROM is done by the device manufacturer in the very last phase of the fabrication process on customers special request. Mask ROMs are customised as per the user requirements, thus, are very costly as different masks are required for different specifications. Because of very high cost of masking, this customization is generally used in manufacturing of ROM at very large scale.
2. **Programmable ROM (PROM):** MROMs are not cost effective for small productions, PROMs are preferred for small quantities. PROMs are programmed using a special hardware which blow fuses with a very high voltage to produce logic "0" and intact fuse defines logic "1". The content of PROM is irreversible once programmed.
3. **Erasable PROM (EPROM):** EPROMs are third type of ROMs which are restructured or reprogrammed using shortwave radiations. An ultraviolet light for

- a specific duration is applied to the EPROM, which destroys/ erases the internal information and after which EPROMs can be programmed again by the user.
4. Electrically EPROM (EEPROM) : EEPROMs are similar to EPROMs except of using ultraviolet radiations for erasing PROM, EEPROM uses electrical signals to erase the content. EEPROM can be erased or reprogrammed by the user without removing them from the socket.

Flash Memory

Flash memory is a non-volatile semiconductor memory which uses the programming method of EPROM and erases electrically like EEPROM. Flash memory was designed in 1980s. Unlike, EEPROM where user can erase a byte using electrical signals, a section of the memory or a set of memory words can be erasable in flash memory and hence the name flash memory i.e. which erases a large block of memory at once. Flash memory is easily portable and mechanically robust as there is no mechanical movement in the memory to read-write data. Flash memory is widely used in USB memory, SD and micro SD memory cards used in cameras and mobile phones respectively.

There are two types of flash memory, viz. NAND flash memory, where read operation is performed by paging the contents to the RAM i.e. only a block of data is accessed not an individual byte or word; and NOR flash memory, which are able to read an individual memory byte/word or cell.

The features of various semiconductor memories are summarised in the Table 1.

Memory	Type	Erase Mechanism/ Level	Write Mechanism	Volatile/ Non- Volatile
Random-access Memory (RAM)	Read–Write	Electrical/ Byte	Electrical	Volatile
Read –only Memory (ROM)	Read–Only	Not Applicable	Masks	Non-volatile
Programmable ROM (PROM)	Read–Only	Not Applicable	Electrical	Non-volatile
Erasable PROM (EPROM)	Read-mostly	UV light/ Chip	Electrical	Non-volatile
Electrically Erasable (EEPROM)	Read-mostly	Electrical/ Byte	Electrical	Non-volatile
Flash memory	Read-mostly	Electrical/ Block	Electrical	Non-volatile

Table 1: Features of Semiconductor Memories

Check Your Progress 1

1. Differentiate among RAM, ROM, PROM and EPROM.
-
.....

2. What is a flash memory? Give a few of its typical uses.
-
.....

3. A memory has a capacity of $16K \times 16$
(a) How many data input and data output lines does it have?
(b) How many address lines does it have?
-
.....

4. A DRAM that stores 4K bytes on a chip and uses a square register array. Each array is of size 4 bits. How many address lines will be needed? If the same configuration is used for a chip which does not use square array, then how many address lines would be needed?
-
.....

5. How many RAM chips of size $256K \times 4$ bit are required to build 1M Byte memory?
-
.....

5.4 SECONDARY MEMORY AND CHARACTERISTICS

In previous section, we have discussed various types of random access and read only memories in detail. RAM and ROM together make the main memory of the computer system. You know that a program is loaded into main memory to complete its execution. Computational units or CPU can directly interact with the main memory. Hence, faster main memory, which can match with the speed of CPU, is always desirable. In the previous section configuration of two types of RAMs, viz SRAM and DRAM were discussed. As you may observe the SRAM consists of flip-flop based circuits, therefore, is quite fast in comparison to DRAM. However, the cost per bit of DRAM is much less than the SRAM. Thus, you may observe the size of main memory is much more than cache. It is discussed in more details in the next unit. To achieve high speed, cost per bit of main memory is generally high which also limits its size.

On the other hand, as we have discussed, RAM, which is a major constituent of the main memory, is volatile i.e. content of the main memory is lost when power is switched off. Because of above mentioned issues, you need a low cost and high capacity, non-volatile memory to store program files and the data for later use. Secondary memory devices which are at the bottom of the memory hierarchy pyramid are ideal for the said purpose. We will discuss various secondary storage devices in this section.

5.4.1 Hard Disk Drive

In the era of Big Data, in which variety of data is generated rapidly, large secondary storage has become an important component of every computer system. Today, hard disk drives (HDD) is the primary type of secondary storage. The size of hard disk drives in modern computer system ranges from Gigabytes (GB) to Terabytes (TB). Internal hard drives extends the internal storage of a computer system whereas external hard drives are used for back up storage.

HDD are electro-mechanical storage devices, which store digital data in the form of small magnetic fields induced on the surface of the magnetic disks. Data recorded on the surface of magnetic disks is read by disks read/write head, which transforms magnetic signal to electrical signal for reading and electrical signal to magnetic field

for writing. HDD is composed of many concentric magnetic disks mounted on a central shaft as shown in Figure 5.8.

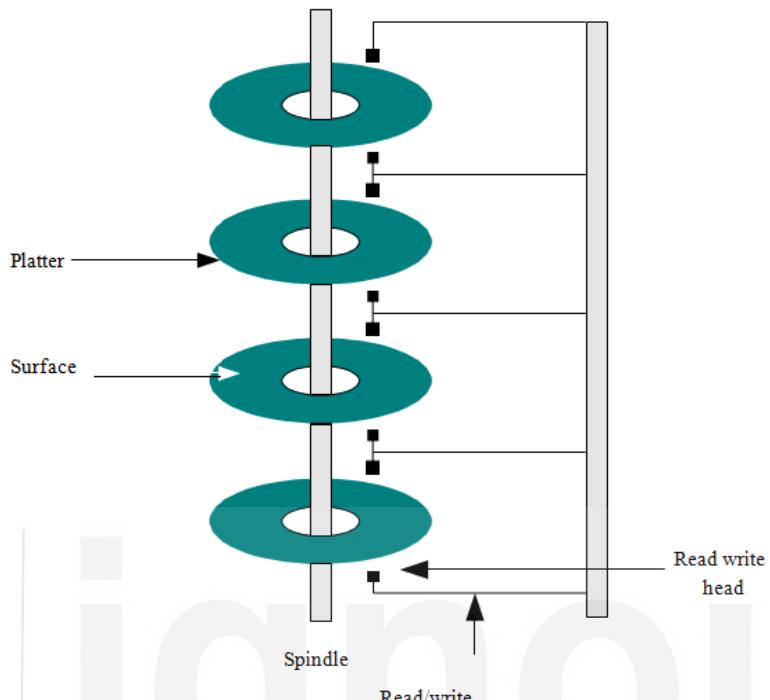


Figure 5.8: Internal structure of Hard disk drives (HDD)

Figure 5.8 shows the internal structure of an HDD. An HDD is made of several concentric magnetic disks mounted on a central shaft called spindle. Each magnetic disk is made of either glass or an aluminium disk called platter. Each platter is coated with ferromagnetic material for storing data. Platter itself is made of non-ferromagnetic material so that its own magnetic field should not interfere the magnetic field of the data. Generally, both sides of the platter is coated with magnetic material for good storage capacity at low cost.

Data recorded on the disk is accessed through a read/write head. Each side of the disk has its own read write head. Each read/write head is positioned at a distance of tens of nanometer called flying height to the platter so that it can easily sense or detect the polarization of the magnetic field.

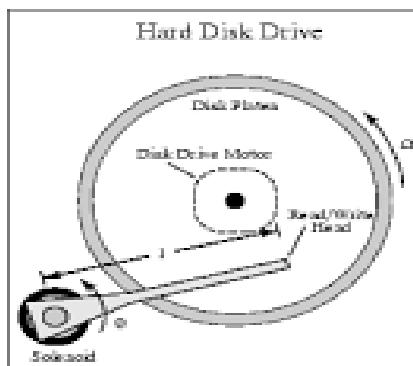


Figure 5.9: Read/ Write Head

Two motors are used in HDD. First one is called the spindle motor, which is used to rotate the spindle on which all the platters are mounted. Second motor is used to move

the read/write heads across the entire surface of the platter radially and is called actuator or access arm.

The Memory System

Magnetic Read and Write Mechanisms

During a read/ write operation, read/write head is kept stationary while platter is rotated by the spindle motor. As you know, data on the disk is recorded in the form of magnetic field. The current is passed through the read/write head which induces a magnetic field on the surface of platter and thus, records a bit on the surface. Different directions of current generates magnetic fields with different polarities and hence are used for storing “1” and “0”. Similarly, to read a bit from the surface, the magnetic field is sensed by the read/write head which produces an electric current of the same polarity and hence the bit value is read.

Data Organization and Formatting

As discussed and shown in figure 5.8, hard disk drives consists of number of concentric platters which are mounted on a spindle forming a cylindrical structure. Data is written in the form of magnetic fields on both surfaces of these platters and is read by read/write head which is connected to an actuator. In this section, we will discuss structure of magnetic disk in detail.

Structure of the disk is shown in figure 5.10. As you know, each magnetic disk is a circular disk mounted on a common spindle but entire disk space is not used for data. Disk surface is divided in to thousands of concentric circular regions called *tracks*. The width of every track is kept the same. Data is stored in these tracks. Magnetic field of one track should not affect the magnetic region in the other track thus two tracks are kept apart with each other by a constant distance. Further, each track is divided into number of sectors and two sectors are kept apart using inter-sector gap. Data is stored in these sectors. Each track forms a cylindrical structure with other tracks on other platters below or above it. For example, an outer most cylinder will have outer most track of all the platters. So, if we have n tracks in a platter then there will be n concentric cylinders too.

Components of the drive are controlled by a disk controller. Now a days, disk controllers are built in to the disk drive. A new or blank magnetic disk is divided into sectors. Each sector has three components: header, 512 byte (or more) data area and a trailer. This process of is called physical / low level formatting. Header and trailer contains metadata about the sectors e.g. sector number, error correcting code etc. Disk controller uses this information whenever it writes or reads a data item on to a sector.

Data is stored in series of logical blocks. The disk controller maps the logical blocks on to the physical disk space and also manages sectors which have been used for storing data and which are still free. This is done by the operating system after partitioning the disk in to one or more groups of cylinders. Disk controller stores the initial data structure file of every sector on to the disk. This data structure file contains a list of used and free sectors, list of bad sectors etc. Windows uses File Allocation Table (FAT) for the said purpose.

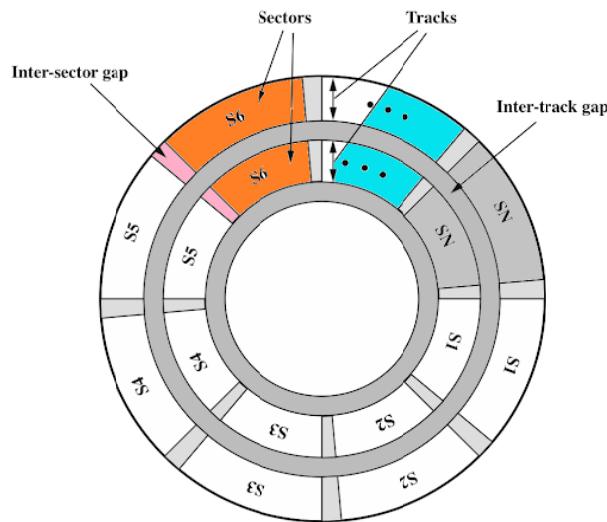


Figure 5.10: Magnetic Disk Structure of CAV

There are two arrangements with which platters are divided into tracks and sectors. The first arrangement is called as *constant linear velocity (CLV)*, in which the density of bits per track is kept uniform, i.e. outer tracks are longer than the inner tracks and hence contains more number of sectors and data. Outermost tracks are generally 40% longer than the innermost track. In this arrangement, in order to maintain uniform bit/ data rate among tracks, the rotation speed is increased from outermost to inner most track. This approach is used by CD-ROM and DVD-ROM drives.

In another approach called as *constant angular velocity (CAV)*, the density of bits / data per track is decreasing as we move from innermost track to outermost track by keeping the disk rotation speed constant. As disk is moving at a constant speed, the width of the data bits increases in the outer tracks, which results in the constant data rate. Figure 5.10 shows that the width of sectors in outer tracks is increasing and density of bits is decreasing.

Disk Performance

Data is read and written on the disks by the operating system for usage at later stage. A disk stores the programs and related data. However, disk is a much slower device and the programs stored on it cannot be executed by the processing unit directly. Therefore, the programs and its related data, which are not in the main memory, are loaded in the main memory from the secondary storage. Since, the speed of disk read/write is very slow in compared to RAM, time to read or write a byte from or on to the disk affects the computer overall efficiency. Therefore, *in a single read/write operation on disk data of one or more sectors is transferred to/from the memory*. An operating system, in general, request for read/write to one or more sectors on the disk. The time taken by the disk to complete a read/ write request of the operating system is known as disk access time. There are number of factors which affect the performance of the disk. These factors are:

1. Seek Time: It is defined as a time taken by the read/write head, or simply as a head, to reach the desired track on which the requested sector is located. Head should reach the desired track in minimum time. Shorter seek time leads to faster I/O operation.
2. Rotational Latency: Since, every track consists of a number of sectors, therefore, the read/write operation can be completed only when the desired sector is available under the read/write head for the I/O operation. It depends on the

rotational speed of the spindle and is defined as a time taken by a particular sector to get underneath the read/write head.

3. Data Transfer Rate: Since, large amount of data is transferred in one read/write operation, therefore, the data transfer rate is also a factor for I/O operation. It is defined as the amount of data read or written by the read/write head per unit time.
4. Controller overhead: It is the time taken by the disk controller for mapping logical blocks to physical storage and keep track of which sectors are free and which are used.
5. Queuing Delay: time spent waiting for the disk to be free.

The disk access time is defined as the summation of seek time, rotational latency, data transfer rate, controller overhead and queuing delay and is given by the equation.

$$\text{access}_{\text{time}} = \text{seek}_{\text{time}} + \text{rotational}_{\text{latency}} + \text{data}_{\text{transferrate}} + \text{controller}_{\text{overhead}} \\ + \text{queuing}_{\text{delay}}$$

Out of the five parameters mentioned in the above equation, most of the time of the disk controller goes in moving the read/write to the desired location and thus seeking the information. If the disk access requests are processed efficiently then performance of the system can be improved. The aim of disk scheduling algorithm is to serve all the disk access requests with least possible head movement. There are number of disk scheduling algorithms which are presented here in brief.

First Come First Serve (FCFS) scheduling: This approach serves the disk access request in the order they arrived in the queue.

Shortest Seek Time First (SSTF) scheduling: Shortest Seek Time First disk scheduling algorithm selects the request from the queue which requires least movement of the head.

SCAN scheduling: The current head position and the head direction is the necessary input to this algorithm. Disk access requests are serviced by the disk arm as disk arm starts from one end of the disk and moves towards the other end. On reaching the other end the direction of the head is reversed and requests are continued to be serviced.

C-SCAN scheduling: Unlike SCAN algorithm, C-SCAN does not serve any request in the return trip. Instead, on reaching to the end, it reverses back to the beginning of the disk and then serves the requests.

LOOK scheduling: LOOK is similar to SCAN algorithm with only a single difference, after serving the last request, LOOK algorithm does not go till the end instead it immediately reverses its direction and moves to the beginning of the other end.

5.4.2 Optical Memories

So far, the storage devices you have studied are based on either electric charge or magnetic field. Magnetic memories are primarily used as a secondary storage device, but they can easily be damaged. However they have lower cost per bit than solid state devices.

First laser based memory was developed in 1982 by Phillips and Sony. Laser based storage devices uses a laser beam to read or write data and are called as optical memories or optical storage devices. As laser beams can be controlled more precisely and accurately than magnetic read/write heads. Data stored on optical drives remains unaffected by the magnetic disturbances in its surrounding.

Initially, these optical storage devices commonly known as compact disk (CD) or CD-DA (Digital Audio) were used to store only audio data of 60 minute duration. Later, huge commercial success of CD lead to development of low cost optical disk technology. These CDs can be used as auxiliary storage and can store any type of digital data. A variety of optical-disk devices have been introduced. We briefly review some of these types.

Compact Disk ROM (CD-ROM)

Compact Disk or CD-ROM are made of a 1.2 mm thick sheet of a polycarbonate material. Each disk surface is coated with a reflective material generally aluminium. The standard size of a compact disk is 120 mm in diameter. An acrylic coat is applied on top of the reflective surface to protect the disk from scratches and dust.

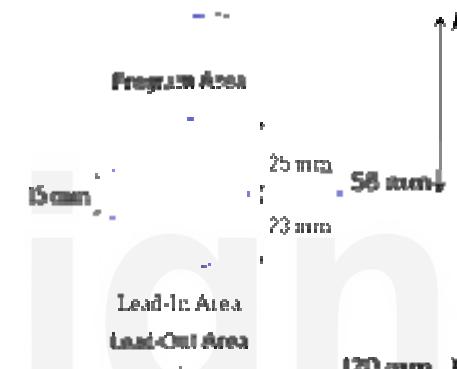


Figure 5.11: Outer Layout of a CD

Unlike magnetic disks, data on an optical disk is recorded in a spiral shape tracks. Each track is separated by a distance of 1.6 mm. Data in a track is recorded in the form of land and pit as shown in Figure 5.13. When a focused laser beam incident on to the optical disk, the disk is burned as per the digitally recorded data forming a pit and land structure. The data is read from the surface by measuring the intensity of the reflected beam. The pit area scatters the incident beam, whereas the land reflects the incident beam, which are read as "0" and "1" respectively.

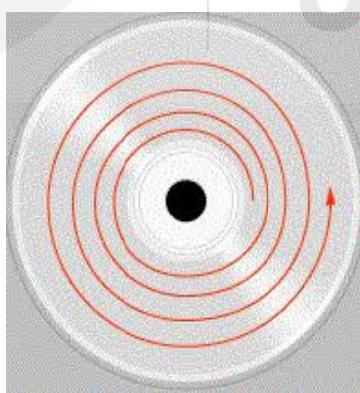


Figure 5.12: Spiral track of CD

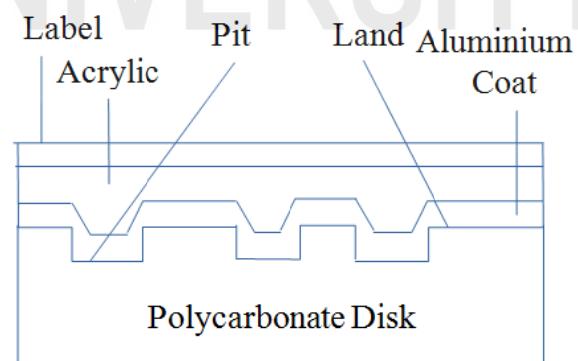


Figure 5.13: Land & Pit formation in CD track

As shown in Figure 5.12, the tracks in CD are in spiral shape. The tracks in CDs are further divided into sectors. All sectors in CDs are equal in length. This means that density of data recorded on the disk is uniform across all the tracks. Inner tracks have less number of sectors whereas outer tracks have more sectors. CD-ROM devices uses constant linear velocity (CLV) method for reading the disk content. In this method, the

disk is rotated at lower velocity as we move away from the center of the disk. This ensures a constant linear velocity at each track of the CD. The format of a sector of CD is shown in Figure 5.14.

SYNC	HEADER	DATA	L-ECC
12 Bytes	4 Bytes	2048 Bytes	288 Bytes

Figure 5.14: Sector format of CD

Data on the CD-ROM are stored in a track as a sequence of sectors. As shown in the Figure 5.14 each sector has four fields *viz.* sync, header, user data followed by error correcting codes. Each part of the sector is described below:

- Sync: It is the first field in every sector. The sync field is 12 byte long. The first byte of sync field contains a sequence of 0s followed by 10 bytes of all 1s and 1 byte of all 0s.
- Header: Header is four byte field in the sector. Three bytes are used to represent the sector address and one byte is used to represent the mode i.e. how subsequent fields in the sector are going to use. There are 3 modes:
 - Mode Zero: Specifies a no user data i.e. blank data field.
 - Mode One: Specifies an user data of 2048 bytes followed by 288 bytes of error correcting code.
 - Mode Two: No error correcting code will be used thus subsequent field will contain 2336 bytes of user data.
- Data: Data field contains the user 2048 byte of user data when mode is 1 or mode 2.
- L-ECC: Layered error correcting code field is 288 byte long field which is used for error detection and correction in mode 1. In mode 2, this field is used to carry an additional 288 bytes of user data.

Compact Disk Recordable (CD-R)

CD-Recordable are the compact disks which are capable of storing any type of digital data. The physical structure of CD-R is same as that of CD-ROM as discussed in previous section except that polycarbonate disk has a very thin layer of an organic dye before the Aluminum coating. CD-R can record user data only once but user can read the data many times thus these are also known as CD-WO (write once), or WORM (write once read many). Many CD writers allow the users to write CD-R in multiple session until CD is full. In each writing session, a partition is created in the CD-R. But once written, data on CD-R cannot be changed or deleted. There are three types of organic dyes used in CD-R.

Cyanine dyes are the most sensitive dye amongst the three types. CD-Rs have cyanine dyes are green in color. Very sensitive to UV rays and even can lose the data if exposed to direct sunlight for few days.

Phthalocyanine dye does not need a stabilizer as compared to cyanine dyes. They are silver, gold or light green in color. They are very less sensitive as compared to cyanine dyes but if exposed to direct sunlight for few weeks, it may lose the data.

Azo dye is the most stable among all types. It is most resistant to UV rays but if exposed to direct sunlight for 3-4 weeks, the CD-R may lose the data.

Compact Disk Rewritable (CD-RW)

The CD-RW are re-writable optical disks. The data on CD-RW can be read or written multiple times. But for writing again on the already written CD-RW, the disk data must be erased first. There are two approaches of erasing the data written on CD-RW. In the first approach, the entire disk data is erased completely i.e. all traces of any previous data is erased. This is called full blanking. Whereas in another approach called as fast blanking, only the meta data is erased. The later approach is faster and allows rewriting the disk. The first approach is used for confidentiality purposes.

The phase change technology is used in CD-RW. The phase change disk uses a material that has significantly different reflectivity in two different phase states. There is an amorphous state, in which the molecules exhibit a random orientation and which reflects light poorly; and a crystalline state, which has a smooth surface that reflects light well. A beam of laser light can change the material from one phase to the other. The phase change technology of CD-RW uses a 15-25 % degree of reflection whereas CD-R works on 40-70 % degree of reflection.

Digital Versatile Disk (DVD)

Digital versatile disk commonly known as DVD is also an optical storage device like CD, CD-R, CD-WR. Among the three DVDs have highest storage capacity ranges from 1.4 GB to 17 GB on a single side. The higher storage capacity is enabled by the use of laser beams of shorter wavelength as compared to compact disks. DVD uses a laser beam of 650 nm whereas compact disk uses a laser beam of 780 nm. Shorter wavelength laser beam creates shorter pits on the polycarbonate disk, thus offers higher storage capacity for similar dimensions. DVD-Audio and DVD-video are a standard format for recording audio and video data on DVDs. Like compact disks, DVD also comes in various variants like DVD-ROM, DVD-R, DVD-WR etc.

Blue Ray Disk

A blue ray disk is a digital disk that can store several hours high definition videos. Blue ray disks is of the same size of DVD, but can store 25 GB to 128 GB of data. A blue ray disk is designed to replace DVD technology. It has its applications in gaming applications, which uses very high quality animations.

5.4.3 Charge-coupled Devices, Bubble Memories and Solid State Devices

Charge-coupled Devices CCDs (CCDs)

Charge couple devices are photo sensitive devices which are used to store digital data. CCD is an integrated circuit of MOS-capacitors called cells, which are arranged in an array like structure in which each cell is connected with its neighbouring cell. Each capacitor can hold the charge which is used to represent the logic “1”. While reading the array of capacitors, the capacitor moves its charge to the neighbouring capacitor with next clock pulse. CCD arrays are mainly used in representing images and video data, where presence and absence of charge in the capacitor represents the corresponding pixel intensity.

As mentioned, CCD are highly photo-sensitive in nature and thus, produces a good quality picture even if light is dim or in low illumination intensity. Now a days, CCDs are widely used in digital cameras, satellite imagery, radar images and other high resolution imagery applications.

Magnetic Bubble Memories

Working principle of magnetic bubble memory is similar to that of charge coupled devices (CCD) discussed in the previous section. Magnetic bubble memory is an

arrangement of small magnetic area called bubble on a series of parallel track made of magnetic material. Each bubble represents a binary “1” and absence of a bubble on magnetic material is interpreted as “0”. Binary data is read from the memory by moving these bubbles towards the edge of a track under the influence of external magnetic field. Magnetic field produces as bubbles remain persistent and do not demagnetise by its own. So, magnetic bubble memories are non-volatile type memories.

Solid State Devices (SSD)

Solid state drives also known as solid state storage devices are based on flash memory. As discussed, flash memory, a non-volatile type memory uses semiconductor devices to store the data. The major advantage of SSD is that it is purely an electronic device i.e. unlike HDD, SSD does not have mechanical read/ write head other mechanical components. Hence, reading and writing through SSD is faster than HDD. Now a days, SSD have replaced HDD in computer systems, however, SSD disks are more expensive than HDDs.

Check Your Progress 2

- What will be the storage capacity of a disk, which has 8 recording surfaces, 32 tracks with each track having 64 sectors. Also, what would be the size of one cylinder of the disk? You may assume that each sector can store 1 MB data.
-
.....

- What would be the rotation latency time for the disk specified above, if it has a rotational speed of 6000 rpm?
-
.....

- What are the advantages and disadvantages of using SSD over HDD?
-
.....

- What are the differences between CLV and CAV disks?
-
.....

5.5 RAID AND ITS LEVELS

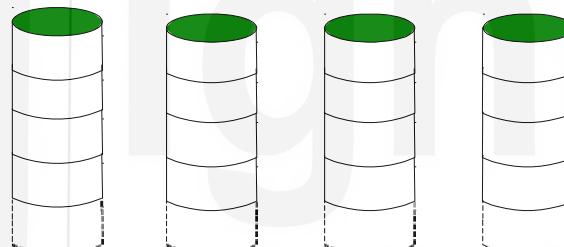
Continuous efforts have been made by researchers to enhance the performance of the secondary storage devices. As pointed out in previous sections performance of the secondary storage is inversely affected by disk access time. Lower the disk access time higher would be the performance. What about an idea of providing parallel access to a group of disks? With the use of parallel access the amount of data that can be accessed per unit time can be enhanced by a significant factor. A mechanism which splits the data on multiple disk is known as data striping. Data access through parallel access allows users to access data stored at multiple disks simultaneously, thus reduces effective reading time. Does data striping ensure protection of data against disk failure?

Another important factor for secondary storage is the reliability of data storage system. Storing same data on more than one disks enhances reliability. If one disk fails, then data can be accessed through another disk. Replicating data on multiple disks is called mirroring. Mirroring brings redundancy in data. So many schemes have been employed to enhance the performance and reliability of data and collectively they are called as redundant arrays of inexpensive disks (RAID). Based on the trade-off between reliability and performance RAID schemes have been categorised into various RAID levels.

Data striping increases the data transfer speed as different data bytes are accessed in parallel from different disks in a single disk access time. Whereas mirroring protects data from disk failures. If one disk fails then same data is accessed from the copy of the data stored in other disk.

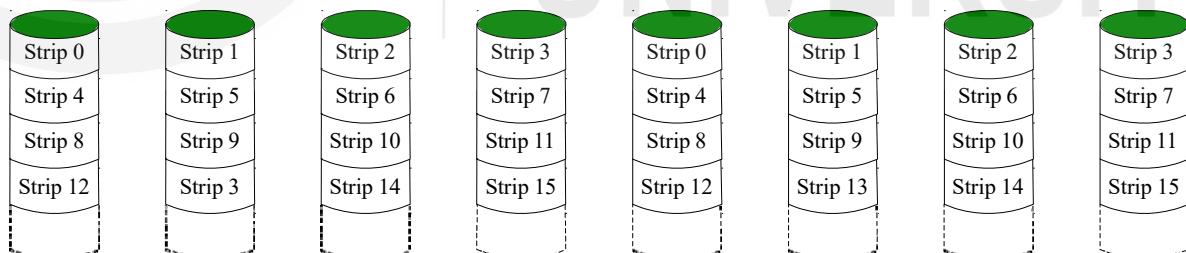
RAID Levels

RAID Level-0: RAID level-0 implements block splitting of data with no protection against disk failures. In block splitting, each block is stored in a different disk in the array. For example, i^{th} block of a file will be stored in $(i \bmod n) + 1$ disk, where n is the total number of disks in the array. In this case, a significant enhancement on the performance can be observed as n blocks can be accessed (one each from each disk) in a single disk access time.



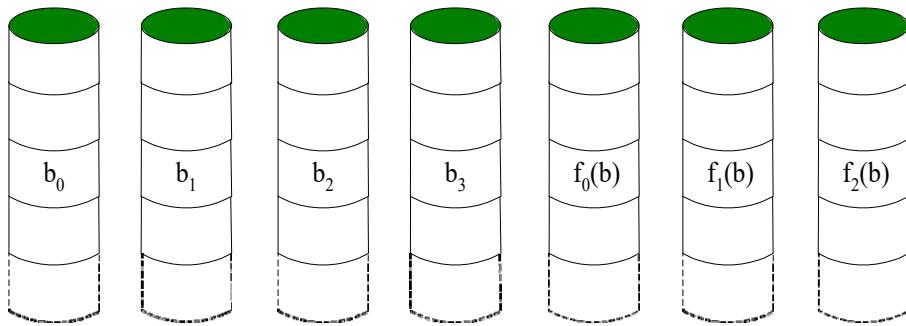
(a) RAID Level 0

RAID Level-1: This level protects data by implementing mirroring. If a system has 2 disks then each block of information will be stored in both of the disks. This ensures, if one disk fails then same copy of the block is accessed from the second disk. Mirroring introduces redundancy unlike level-0 which increases the data transfer rate.



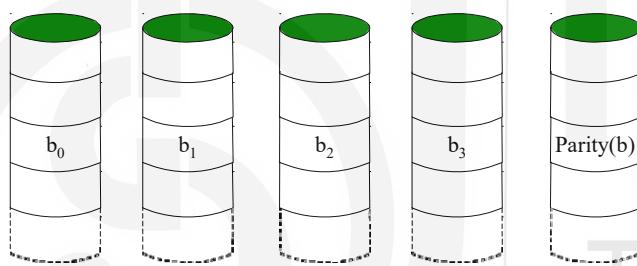
(b) RAID Level 1

RAID Level-2: This level uses error detection and correction bits, which are extra bits used for detection and correction of a single bit error in a byte. This is why this level is also known as memory-style error correction code organization. If one of the disk fails then parity bits and remaining bits of the byte are used to recover the bit value.



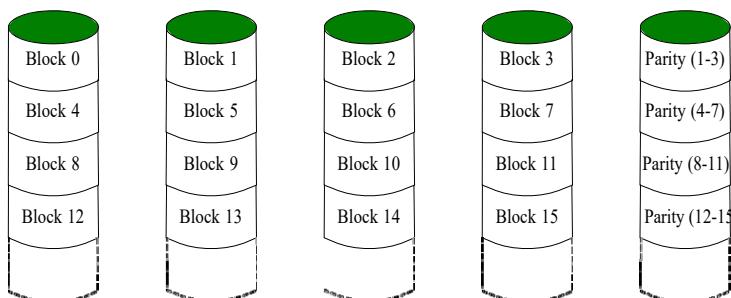
(c) RAID 2 (Redundancy through Hamming Code)

RAID Level-3: Single parity disk is used in this scheme. Parity bit for a sector is computed and stored in a parity disk. During the access, parity bit of the sector is computed and if computed parity bit is equal to the stored parity, the missing bit is 0 otherwise it is 1. This RAID level is also known as bit-interleaved parity organization. Thus has an advantage over level-2 that only single parity disk is used as compare to number of parity disks in level-2. The biggest drawback of this approach is that all the disks are used for single I/O operation in computation of the parity bit which slows down the disk access and also restricts parallel access.



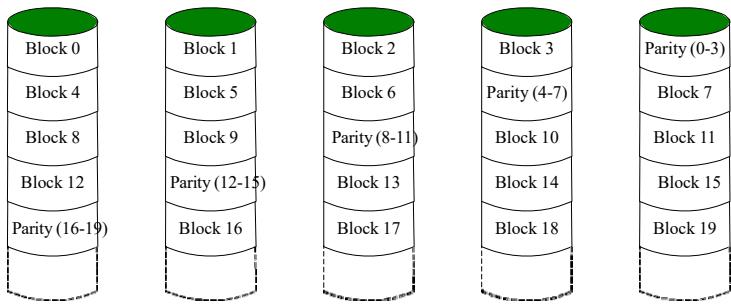
(d) RAID Level 3

RAID Level-4: This level uses block striping and one disk is used to keep parity block. This is also called block-interleaved parity organization. The advantage of block interleaving is that parity block along with corresponding blocks on other disks is used to retrieve the damaged block or the blocks of the failed disk. Unlike in level-3, block access reads one disk which allows parallel access to other blocks stored in other disks in the array.



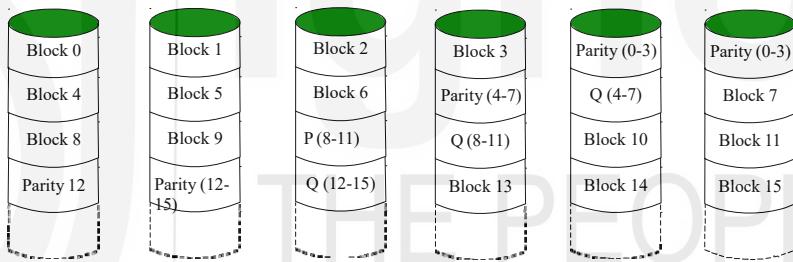
(e) RAID 4 (Block level Parity)

RAID Level-5: This level stores block of data and parity in all the disks in the array. One disk store the parity while data is spread out on different disks in the array. This structure is also known as block-interleaved distributed parity.



(f) RAID 5 (Block-level Distributed Parity)

RAID Level-6: Level-6 uses error correcting codes for recovery of damaged data while other levels uses parity. It also provides protection against multiple disks failures. For the recovery purposes, this arrangement is used to store redundant data on some of the disks, hence it is also called as $p + q$ redundancy scheme. Here, p is the number of disks that store the error correcting codes while q is the number of disks that store redundant data.



(g) RAID Level 6

Table below summarises characteristics of various RAID levels.

RAID Level	Category	Features	I/O Request Rate (Read /write)	Data Transfer Rate (Read /write)	Typical Application
0	Striping	a) The disk is divided into blocks or sectors. b) Non-redundant.	Large Blocks: Excellent	Small Blocks: Excellent	Applications which requires high performance for non-critical data
1	Mirroring	a) Mirror disk which contains the same data is associated with every disk. b) Data Recovery is simple. On failure, data is recovered from the mirror disk.	Good / fair	Fair /fair	May be used for critical files

2	Parallel Access	a) All member disks participate in every I/O request. b) Synchronizes the spindles of all the disks to the same position. c) The blocks are very small in size (Byte or word). d) Hamming code is used to detect double-bit errors and correct single-bit error.	Poor	Excellent	Not useful for commercial purposes.
3	Parallel Access	a) Parallel access as in level 2, with small data blocks. b) A simple parity bit is computed for the set of data for error correction.	Poor	Excellent	Large I/O request size application, such as imaging CAD
4	Independent access	a) Each member disk operates independently, which enables multiple input/output requests in parallel. b) Block is large and parity strip is created for bits of blocks of each disk. c) Parity strip is stored on a separate disk.	Excellent/fair	Fair / poor	Not useful for commercial purposes.
5	Independent access	a) Allows independent access as in level 4. b) Parity strips are distributed across all disks. b) Distribution avoids potential input/output bottleneck found in level 4.	Excellent / fair	Fair / poor	High request rate read intensive, data lookup
6	Independent access	Also called the p+q redundancy scheme, is much like level 5, but stores extra redundant information to guard against multiple disk failures.	Excellent/poor	Fair / poor	Application requiring extremely high availability

Check Your Progress 3

1. What is the need of RAID?

.....

.....

2. Which RAID levels provide good data transfer rate?

.....

3. Which RAID level is able to fulfil large number of I/O requests?

.....
.....

5.6 SUMMARY

This unit introduces the concept of memory hierarchy, which is primarily required due to the high cost per bit of high speed memory. The processing unit have register, cache, main memory and secondary or auxiliary memory. The main memory consists of RAM or ROM. This unit explains the logic circuit and organisation of RAM and ROM. The unit also explains several different types of secondary storage memories. The unit provide details on hard disk and its characteristics. It also gives details of different kind of optical disk. The concept of access time and constant linear and angular velocity has also been explained in details. For larger computer systems simple hard disk is not sufficient, rather an array of disks called RAID are used for such systems to provide good performance and reliability. The concept of RAID and various levels of RAID has been defined in this unit. The next unit will introduce you to the concept of high speed memories.

5.7 ANSWERS

Check Your Progress 1

1. RAM is a sequential circuit, volatile, requires refreshing (DRAM) and is a read/write memory; ROM, PROM and EPROM are mostly non-volatile memories. ROM is a combinational circuits. All these ROMs are written mostly once and read many times.
2. Flash memory is a non-volatile semiconductor memory, where a section of the memory or a set of memory words can be erased. They are portable and mechanically robust as there is no mechanical movement in the memory to read/write data. Flash memory is used in USB memory, SD and micro SD memory cards used in cameras and mobile phones respectively.
3. (a) Since a word of data is 16 bits, it will have 16 data input and 16 data output lines, if not multiplexed.
(b) The number of words are 16K, which is 2^{14} . Thus, 14 address lines would be required.
4. The memory must select one of the 4K bytes, which is 2^{12} . In case a square array is used (as shown in Figure 5.6), then 6 row address and 6 column address lines would be needed, which can be multiplexed. So just 6 address lines be sufficient. However, for a non square memory you may require all 12 address lines.
5. Two chips will be required to make 256×8 memory. 4 such combinations would be required to make 1 MB memory. Thus, you will require 8 such chips.

Check Your Progress 2

1. Storage capacity of a disk = recording surfaces \times tracks per surface \times sectors per track \times size of each sector

$$\text{Storage capacity of a disk} = 8 \times 32 \times 64 \times 1 \text{ MB} = 2^3 \times 2^5 \times 2^6 \times 2^{20} = 2^{34} = 16 \text{ GB}$$

$$\text{One cylinder will have} = 8 \times 64 \times 1 \text{ MB} = 2^3 \times 2^6 \times 2^{20} = 512 \text{ MB}$$

2. The time of one rotation = $1/6000 \text{ min} = 60/6000 \text{ sec} = 1/100 \text{ sec} = 10 \text{ millisec}$
Rotational latency = on an average time of half rotation = 5 ms
3. SSD drives does not require any mechanical rotation, therefore are less prone to failure. In addition, they are much faster than HDD. But they are more expensive than HDD
4. The size of sectors on CLV disks is same on the entire disk, therefore, these disks are rotated at a different speed. Density of data is same in all the sectors. In CAV disks the rotation speed is same, thus, sector size is more in the outer tracks. However, reading/writing process, in general, is faster.

The Memory System

Check Your Progress 3

1. RAID are a set of storage devices put together for better performance and reliability. Different kind of RAID levels have different objectives.
2. Good data transfer rate are provided by RAID level 0, 2 and 3.
3. Large number of I/O requests are fulfilled by RAID level 0, 1, 4, 5, 6.



UNIT 6 ADVANCE MEMORY ORGANISATION

Structure	Page Nos.
6.0 Introduction	
6.1 Objectives	
6.2 Locality of Reference	
6.3 Cache Memory	
6.4 Cache Organisation	
6.4.1 Issues of Cache Design	
6.4.2 Cache Mapping	
6.4.3 Write Policy	
6.5 Associative Memory	
6.6 Interleaved Memory	
6.7 Virtual Memory	
6.8 Summary	
6.9 Answers	

6.0 INTRODUCTION

In the last unit, the concept of Memory hierarchy was discussed. The Unit also discussed different types of memories including RAM, ROM, flash memory, secondary storage technologies etc. The memory system of a computer uses variety of memories for program execution. These memories vary in size, access speed, cost and type, such as volatility (volatile/ non-volatile), read only or read-write memories etc. As you know, a program is loaded in to the main memory for execution. Thus, the size and speed of the main memory affects the performance of a computer system. This unit will introduce you to concepts of cache memory, which is small memory between the processing unit and main memory. Cache memory enhances the performance of a computer system. Interleaved memory and associative memories are also used as faster memories. Finally, the unit discusses the concept of virtual memory, which allows programs larger than the physical memory.

6.1 OBJECTIVES

After going through this Unit, you will be able to:

- explain the concept of locality of reference;
- explain the different cache organisation schemes;
- explain the characteristics of interleaved and associative memories;
- explain the concept of virtual memory.

6.2 LOCALITY OF REFERENCE

Memory system is one of the important component of a computer. A program is loaded in to the main memory for execution. Therefore, a computer should have a main memory, which should be as fast as its processor and should have large size. In general, the main memory is constructed using DRAM technology which is about 50 to 100 times slower than the processor speed. This may slow down the process of instruction execution of a computer. Using SRAM may change this situation as it is almost as fast as a processor, however, it is a costly memory. So, what can you do? Is it possible to use large main memory as DRAM, but use a faster small memory between processor and main memory? Will such a configuration enhance performance of a computer? This section will try to answer these questions.

The important task of a computer is to execute instructions. It has been observed that on an average 80-85 percent of the execution time is spent by the processor in accessing the instruction or data from the main memory. The situation becomes even worst when instruction to be executed or data to be processed is not present in the main memory.

Another factor which has been observed by analysing various programs is that during the program execution, the processor tends to access a section of the program instructions or data for a specific time period. For example, when a program enters in a loop structure, it continues to access and execute loop statements as long as the looping condition is satisfied. Similarly, whenever a program calls a subroutine, the subroutine statements are going to execute. In another case, when a data item stored in an array or array like structure is accessed then it is very likely that either next data item or previous data item will be accessed by the processor. All these phenomena are known as *Locality of Reference* or *Principle of Locality*.

So, according to the principle of locality, for a specific time period, the processor tends to make memory references closed to each other or accesses the same memory addresses again and again. The earlier type is known as *spatial locality*. Spatial locality specifies if a data item is accessed then data item stored in a nearby location to the data item just accessed may be accessed in near future. There can be special case of spatial locality, which is termed as sequence locality. Consider a program accesses the elements of a single dimensional array, which is a linear data structure, in the sequence of its index. Such accesses will read/write on a sequence of memory locations one after the other. This type of locality, which is a case of spatial locality, is referred to as sequence locality.

Another type of locality is the *temporal locality*, if a data item is accessed or referenced at a particular time, then the same data item is expected to be accessed for some time in near future. Typically it is observed in loop structures and subroutine call.

As shown in Figure 6.1, when the program enters in the loop structure at line 7, it will execute the loop statements again and again multiple times till the loop terminates. In this case, processor needs to access instructions 9 and 10 frequently. On the other hand, when a program accesses a data item store in an array, then in the next iteration it accesses a data item stored in an adjacent memory location to the previous one.

```
1 int main()
2 {
3     int i, num;
4     int fact = 1, sum = 0;
5
6     cout<<"\n Please Enter any number to Find Factorial\n" ;
7     cin>>num ;
8
9     for (i = 1; i <= num; i++)
10    {
11        fact = fact * i;
12        sum = sum + fact;
13    }
14
15    cout<<"\nFactorial of "<<num<<" is" <<fact;
16    cout<<"\nsum is" <<sum;
17    return 0;
18 }
```

Figure 6.1: Loop structure

The locality of reference, be it spatial or temporal, suggests that in most cases accesses to a program instruction and data confines to a locality, hence, a very fast memory that captures the instructions and data nearer to the current instructions and data accesses can potentially enhance the overall performance of a computer. Thus,

attempts are continuously made to utilize the precious time of the processor efficiently. A high speed memory, called cache memory, was developed. Cache memory utilises the principle of locality to reduce the memory references to the main memory by keeping not only the currently referenced data item but also the nearby data items. The cache memory and its organisation is discussed in the next sections.

6.3 CACHE MEMORY

A processor makes many memory references to execute an instruction. A memory reference to the main memory is time consuming as main memory is slower compared to the processing speed of the processor. These memory references, in general, tends to form a cluster in the memory, whether it is a loop structure, execution of a subroutine or an access to a data item stores in an array.

If you keep the content of the cluster of expected memory references in a small, extremely fast memory then processing time of an instruction can be reduced by a significant amount. Cache memory is a very high speed and expensive memory as compared to the main memory and its access time is closer to the processing speed of the processor. Cache memory act as a buffer memory between the processor and the main memory.

Because cache is an expensive memory so its size in a computer system is also very small as compared to the main memory. Thus, cache stores only those memory clusters containing data/ instructions, which have been just accessed or going to be accessed in near future. Data in the cache is updated based on the principle of locality explained in the previous section.

How data access time is reduced significantly by using cache memory?

Data in main memory is stored in the form of fixed size blocks/pages. Cache memory contains some blocks of the main memory. When processor wants to read a data item from the main memory, a check is made in the cache whether data item to be accessed is present in the cache or not. If data item to be accessed is present in the cache then it is read by the processor from the cache. If data item is not found in the cache, a memory reference is made to read the data item from the main memory, and a copy of the block containing data item is also copied into the cache for near future references as explained by the principle of locality. So, whenever processor attempts to read the data item next time, it is likely that the data item is found in the cache and saves the time of memory reference to the main memory.

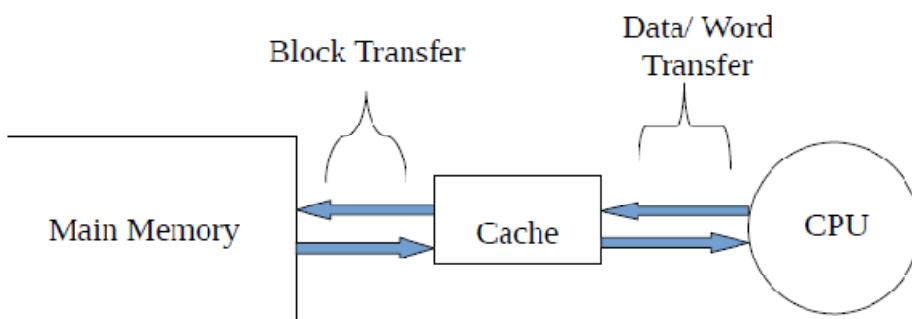


Figure 6.2: Cache Memory

As shown in the Figure 6.2, if requested data item is found in the cache it is called as **cache hit** and data item will be read by the processor from the cache. And if requested data item is not found in cache, called **cache miss**, then a reference to the main memory is made and requested data item is read and block containing data item will also be copied into the cache.

Average access time for any data item is reduced significantly by using cache than without using cache. For example, if a memory reference takes 200 ns and cache takes 20 ns to read a data item. Then for five continuous references will take:

$$\begin{aligned} \text{Time taken with cache : } & 20 \text{ (for cache miss) + 200 (memory reference)} \\ & + 4 \times 20 \text{ (cache hit for subsequent access)} \\ & = 300 \text{ ns} \end{aligned}$$

$$\text{Time without cache : } 5 \times 200 = 1000 \text{ ns}$$

In the given example, the system first looks into the cache for the requested data item. As it is the first reference to the data item it will not be present in the cache, called as cache miss, and thus, requested data item will be read from the main memory. For subsequent requests of the same data item, the data item will be read from the cache only and no references will be made to the main memory as long as the requested data remains in the cache.

Effective access time is defined as the average access time of memory access, when a cache is used. The access time of memory access is reduced in case of a cache hit, whereas it increases in case of cache miss. In the above mentioned example processor takes 20 + 200 ns for a cache miss, whereas it takes only 20 ns for each cache hit. Now suppose, we have a hit ratio of 80%, i.e. 80 percent of times a data item would be found in the cache and 20 % of the times it would be accessed from the main memory. So effective access time (EAT) will be computed as :

$$\begin{aligned} \text{effective access time} &= (\text{cache hit} \times \text{data access time from cache only}) \\ &+ (\text{cache miss} \times \text{data access time from cache and main memory}) \end{aligned}$$

$$\begin{aligned} \text{effective access time} &= 0.8 \text{ (hit ratio)} \times 20 \text{ (cache hit time)} \\ &+ 0.2 \text{ (miss ratio)} \times 220 \quad (\text{cache miss and memory reference}) \end{aligned}$$

$$\begin{aligned} \text{effective access time} &= 0.8 \times 20 + 0.2 \times 220 \\ &= 16 + 44 \\ &= 60 \text{ ns} \end{aligned}$$

From the example it is clear that cache reduces the average access time and effective access time for a data item significantly and enhance the computer performance.

Check Your Progress 1

1. What is the importance of locality of reference?

.....
.....
.....

2. What is block size of main memory for cache transfer?

.....
.....
.....

3. Hit ration of computer system is 90%. The cache has an access time of 10ns, whereas the main memory has an access time of 50ns. Computer the effective access time for the system.

.....
.....
.....

6.4 CACHE ORGANISATION

The main objective of using cache is to decrease the number of memory references to a significant level by keeping the frequently accessible data/instruction in the cache. Higher the hit ratio (number of times requested data item found in cache / total number of times data item is requested), lower would be the references to the main memory. So there are number of questions that need to be answered while designing the cache memory. These Cache design issues are discussed in the next subsection.

6.4.1 Issues of Cache Design

In this section, we present some of the basic questions that should be asked for designing cache memory.

What should be the size of cache memory?

Cache is an extremely fast but very expensive memory as compared to the main memory. So large cache memory may shoot up the cost of the computer system and too small cache might not be very useful in real time. So, based on various statistical analyses, if a computer system has 4 GB of main memory then the size of the cache may go up to 1MB.

What would be the block size for data transfer between cache and main memory?

Block size directly affects the cache performance. Higher block size would ensure only fewer number of blocks in cache, whereas small block size contains fewer data items. As you increase the block size, the hit ratio first increases but it decreases as you further increase the block size. Further increase in block size will not necessarily result in access of newer data items, as probability of accessing data items in the block with larger number of data items tends to decrease. So, optimal size of the block should be chosen to maximise the hit ratio.

How blocks are going to be replaced in cache?

As execution of the process continues, the processor requests for new data items. For new data items and thus, new blocks to be present in the cache, the blocks containing old data items must be replaced. So there must be a mechanism which may select the block to be replaced which is least likely to be needed in near future.

When changes in the blocks will be written back on to the main memory?

During the program execution, the value of a data item in a cache block may get changed. So the changed block must be written back to the main memory in order to reflect those changes to ensure data consistency. So there must be a policy, which may decide when the changed cache block is written back to the main memory.

In certain computer organisations, the cache memory for data and instruction are placed separately. This results in separate address spaces for the instructions and data. These separate caches for instructions and data are known as **instruction cache** and **data cache** respectively. If processor requests an instruction, then it is provided by the instruction cache, whereas requested data item is provided by the data cache. Using separate cache memories for instruction and data enhances computer performance. While some computer systems implement different cache memories for data and instructions other implements multiple levels of cache memories. Two-level cache popularly known as **L1 cache** and **L2 cache** is most commonly used. Size of level 1 cache or L1 cache is smaller than the level 2 or L2 cache. Comparatively more frequently used data/instructions are stored in L1 cache.

As discussed earlier, the main memory is divided into blocks/ frames/ pages of k words each. Each word of the memory unit has a unique address. A processor requests for read/write of a memory word. When a processor's request of a data item cannot be serviced by cache memory, i.e. a *cache miss* occurs, the block containing requested data item is read from the main memory and a copy of the same is stored in cache memory. A cache memory is organised as a sequence of line. Each cache line is identified by a cache line number. A cache line stores a tag and a block of data. Cache and main memory structure is shown in Figure 6.3. General structure of cache memory having M lines and $N=2^n$ main memory size is shown in figure 6.3(a) and figure 6.3(b) respectively.

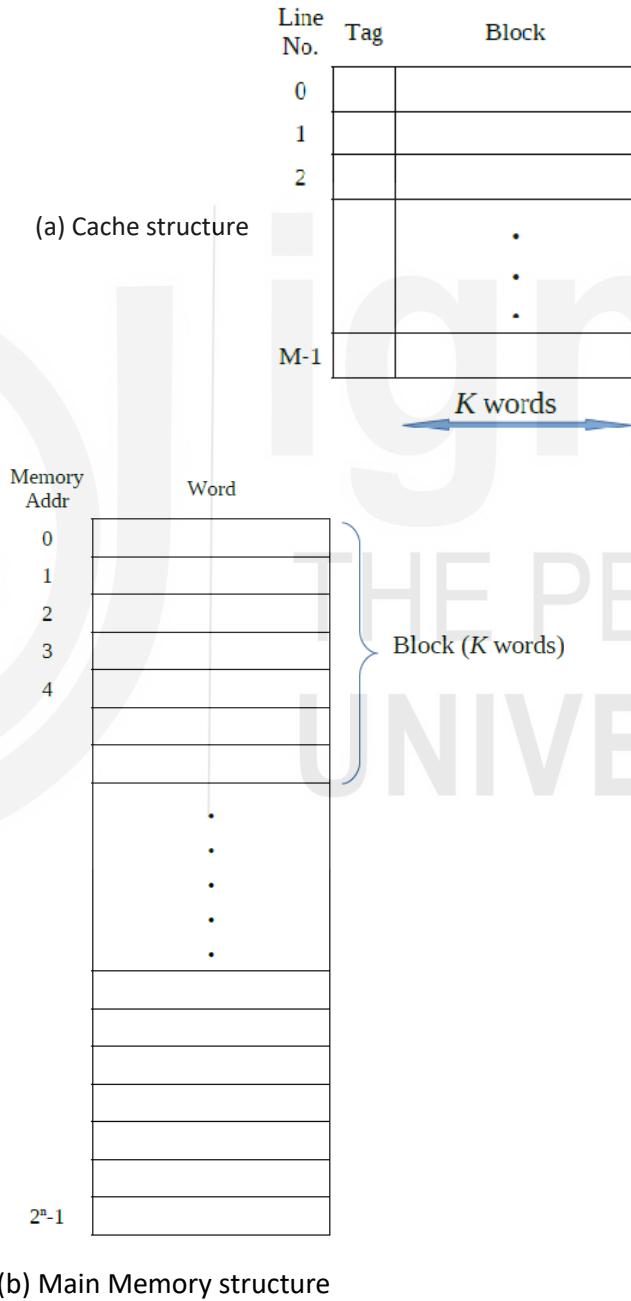


Figure 6.3: Structure of Cache and Main Memory

An example of cache memory of size 512 words is shown in Figure 6.4. The example shown in Figure 6.4 has a main memory of 64 K words of 16 bits each and cache

memory can have 512 words of 16 bits each. To read a data item processor sends a 16 bit address to the cache and if cache misses then the data item/ word is fetched from the main memory and accessed data item/ word is also copied into the cache. Please note that the size of block is just 1 memory word in this case.

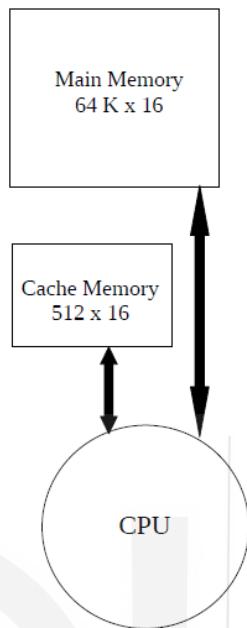


Figure 6.4: Example of Cache and Main Memory

6.4.2 Cache Mapping

As discussed earlier, a request of processor to access a main memory is checked in cache memory. Where can a block of main memory be placed in cache and how processor can determine, if the data requested is present in cache? Answer to these questions are provided by cache mapping scheme. A mapping mechanism maps the block from the main memory to a cache line. The mapping is required as cache is much smaller in size than the size of the main memory. So only few blocks from the main memory can be stored in cache. There are three types of mapping in cache:

Direct Mapping:

Direct mapping is the simplest amongst all three mapping schemes. It maps each block of main memory into only one possible cache line. Direct mapping can be expressed as a *modulo M* function, where M is the total number of cache lines as shown:

$$i = j \bmod M$$

where, i = number of cache line to which main memory block would be mapped.

j = the block address of main memory, which is being requested

M = total number of cache lines

So, line 0 of the cache will store block 0, M , $2M$ and so on. Similarly, line 1 of cache will store block 1, $M + 1$, $2M + 1$, and so on.

An address of main memory word, as shown in Figure 6.3(b), consists of n bits. This address of each word of main memory has two parts: block number ($n-k$ bits) and word number within the block (k bits). Here, each block of the main memory contains 2^k number of words. The cache memory interprets $n-k$ bit block number in to two parts

as: tag and line number. As indicated in Figure 6.3(a), the cache memory contains M lines. Assuming m address bits ($2^m = M$) are used to identify each line, then most significant $(n-k) - m$ bits of $(n-k)$ bit block number are interpreted as tag and m bits are used as line number of the cache. Please note the tag bits are used to identify, which of the main memory block is presently in that cache line. The following diagram summarizes the bits of main memory address and related cache address:

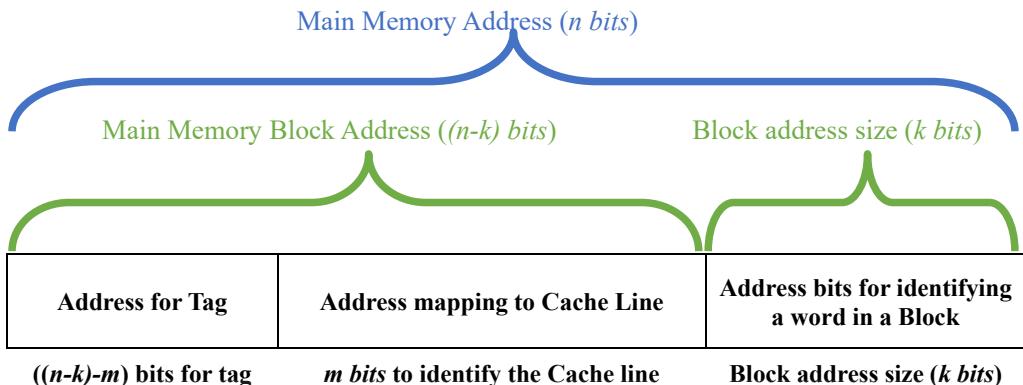


Figure 6.5: Main Memory address to Cache address mapping in Direct mapping Scheme

Please note the following points in the diagram given above.

The size of Main Memory address: n bits

Total number of words in main memory: 2^n

The size of a Main memory block address: most significant $(n-k)$ bits

Number of words in each block: 2^k

In case, a referenced memory address is in cache, bits in the tag field of the main memory address should match the tag field of the cache.

Example: Let us consider a main memory of 16 MB having a word size of a byte and a block size is of 128 bits or 16 words (one word is one byte in this case). The cache memory can store 64 KB of data. Determine the following:

- Size of various addresses and fields
- how will you determine that a hexadecimal address is in cache memory, assuming a direct mapping cache is used?

Solution:

a) Size of main memory= $16\text{MB} = 2^{24}$ Bytes

Each block consists of 16 words, thus total number of blocks in main memory would be $= 2^{24}/ 16 = 2^{20}$ blocks, thus $n = 24$ and $k = 4$ (as $2^4=16$). Therefore, main memory block address ($n-r$) = 20

Data size of cache is 64 KB = 2^{16} Bytes

Total number of cache lines (M) = cache size/ block size = $2^{16} / 2^4 = 2^{12}$, therefore, number of cache lines = 2^{12} and $m = 12$

Length of address field to identify a word in a Block (k) = 4 bits

Length of address to identify a Cache line (m) = 12 bits

Length of Tag field = $(24- 4) - 12 = 8$ bits.

Thus, a main memory address to cache address mapping for the given example would look like this:

Main Memory Address $n = 24$ bits		
Address of a Block of data = 20 bits		$k=4$ bits
Address mapping for direct cache mapping scheme		
Tag = 8 bits	Cache line number address $m = 12$ bits	$k=4$ bits

Figure 6.6: Direct Cache mapping

b) Consider a 24 bit main memory address in hexadecimal as FEDCBA. The following diagram will help in identifying, if this address is in the cache memory or not in case direct mapping scheme is used.

Main Memory Address $n = 24$ bits = 6 hex digits					
F	E	D	C	B	A
<i>Address of a Block of data = 20 bits</i>				<i>k=4 bits</i>	
F E D C B				A	
<i>Address mapping for direct cache mapping scheme</i>					
<i>Tag = 8 bits</i>	<i>Cache line number address m = 12 bits</i>			<i>k=4 bits</i>	
F E	D C B			A	
1111 1110	1101 1100 1011			1010	

Figure 6.7: Direct Cache mapping example

Now, the following steps will be taken by the processing logic of processing unit and hardware of Cache memory:

1. The tag number (FE in this case) is compared against the Tag number of data stored in the cache line (DCB in this case).
2. In case both are identical
then (this is the case of cache hit): Ath word from the cache line DCB is accessed by the processing logic.
else (this is a case of cache miss): The cache line 16 words data is read to cache memory line (DCB) and its tag number is now FE. The required Ath word is now accessed by the processing logic

Direct mapping is very easy to implement but has a disadvantage as location in which a specific block is to be stored in cache is fixed. This arrangement leads to low hit ratio as when processor wants to read two data items belongs to two different blocks, which map to single cache location, then each time other data item is requested, the block in the cache must be replaced by the requested one. This phenomenon is also known as *thrashing*.

Associative Mapping:

Associative mapping is the most flexible mapping in cache organisation as it allows to store any block of the main memory in any of the cache line/or location. It uses complete $(n-k)$ bits of block address field as a tag field. Cache memory stores $(n-k)$ bits of Tag and $(2^k \times \text{Word Size in bit})$ data. When a data item/ word is requested, $(n-k)$ bit tag field is used by the cache control logic to search all the tag fields stored in the cache simultaneously. If there is a match (cache hit) then corresponding data item is read from the cache, otherwise (cache miss) the block of data that contains the word to be accessed is read from the main memory. It replaces any of the cache line. In addition, the block address of the accessed block from the main memory replaces the tag of the cache line. It is also the fastest mapping amongst all types. Different block replacement policies are used for replacing the existing cache content by newly read data, however, those are beyond the scope of this unit. This mapping requires most complex circuitry, as it requires all the cache tags to be checked simultaneously with the block address of the access request.

Main Memory Address :

Address of a block of data is same as Tag	Address bits for identifying a word in a Block
$(n-k)$ bits	k bits

Every line of Associative Cache has the following format:

Tag	Data Block of k words	
$(n-k)$ bits		Data bits

Figure 6.8: Associative mapping

The following example explains the set associative mapping.

Example: Let us consider a main memory of 16 MB having a word size of a byte and a block size is of 128 bits or 16 words (one word is one byte in this case). The cache memory can store 64 KB of data. Determine the following size of various addresses and fields, if associative mapping is used

Solution:

$$\text{Size of main memory} = 16\text{MB} = 2^{24} \text{ Bytes}$$

Each block consists of 16 words, thus total number of blocks in main memory would be $= 2^{24}/16 = 2^{20}$ blocks, thus $n = 24$ and

$k = 4$ (as $2^4=16$). Therefore, main memory block address $(n-r) = 20$

Data size of cache is 64 KB = 2^{16} Bytes

Total number of cache lines (M) = cache size/ block size $= 2^{16}/2^4 = 2^{12}$, therefore, number of cache lines $= 2^{12}$ and $m = 12$

Length of Tag field $= (24-4) = 20$ bits.

Size of data $= 2^k \times \text{Word Size in bit} = 2^4 \times 8 = 128$ bits.

Thus, size of one line of cache $= 128+20=148$ bits.

Set Associative Mapping:

The major disadvantage of direct mapping is that location of the cache line onto which a memory block is going to be mapped is fixed which results in poor hit ratio and unused cache locations. The associative mapping removes these hurdles and any block of memory can be stored anywhere in cache location. But associative cache uses complex matching circuit and big tag size. Set Associative mapping reduces the disadvantages of both the above mentioned cache mapping techniques and is built on their strengths. In set associative mapping scheme, cache memory is divided into v sets where each set contains w cache lines. So, total number of cache lines M is given as:

$$M = v \times w$$

where v is the number of sets and w is the number of cache lines in v

The cache mapping is done using the formula:

$$i = j \bmod v$$

where i is the set number and j is the block address of word to be accessed.

Cache control logic interprets the address field as a combination of *tag* and *set* fields as shown:

Tag	Set	Word
$((n-k)-d)$ bits	d bits	k bits

Figure 6.8: Set Associative mapping

Cache mapping logic uses d -bits to identify the set as $v = 2^d$ and $((n-k)-d)$ bits are used to represent the *tag field*. In set-associative mapping, a *block* j can be stored at any of the cache line of *set* i . To read a data item, the cache control logic first simultaneously looks into all the cache lines using $((n-k)-d)$ bits of *tag field* of the *set* identified by d -bits of the *set field*, otherwise a data item is read from the main memory and corresponding data is copied into the cache accordingly. Set associative mapping is also known as w -way set-associative mapping. It uses lesser number of bits $((n-k)-d)$ bits as compare to $(n-k)$ bits in associative mapping in *tag field*.

A comprehensive example showing possible locations of main memory blocks in Cache for different cache mapping schemes is discussed next.

Example: Assume a main memory of a computer consists of 256 bytes, with each memory word of one byte. The memory has a block size of 4 words. This system has a cache which can store 32 Byte data. Show how main memory content be mapped to cache if (i) Direct mapping (ii) Associative mapping and (iii) 2 way set associative memory is used.

Solution:

$$\text{Main memory size} = 256 \text{ words} (\text{a word} = \text{one byte}) = 2^8 \Rightarrow n=8 \text{ bits}$$

$$\text{Block Size} = 4 \text{ words} = 2^2 \Rightarrow k=2 \text{ bits}$$

The visual representation of this main memory:

Block Number of memory in equivalent decimal	Memory Location Address		Assume data stored in the location
	Block Address	Word Address	
0	000000	00	1001010
	000000	01	1101010
	000000	10	0001010
	000000	11	0001010
1	000001	00	1111010
	000001	01	0101010
	000001	10	1001010
	000001	11	1101010
2	000010	00	1101010
	000010	01	0001010
	000010	10	0101010
	000010	11	0011010
...
7	000111	00	0000010
	000111	01	0000011
	000111	10	0000011
	000111	11	0001110
...
63	111111	00	1111010
	111111	01	1111011
	111111	10	0101011
	111111	11	0101110

Figure 6.9: An example of Main Memory Blocks

(i) *Direct Mapping Cache*:

The size of cache = 32 bytes

The block size of main memory = words in one line of cache = 4 $\Rightarrow k=2$ bits

The cache has $= 32 / 4 = 8$ lines with each line storing 32 bits of data (4 words)

$$\text{Therefore, } m=3 \text{ as } 2^3 = 8$$

Thus, Tag size = $(n-k) - m = (8 - 2) - 3 = 3$

The address mapping for an address: 11111101

Block Address of Main Memory		Address of a word in a Block
111	111	01
111	111	01
Tag	Line Number	

Line Number = 111 = 7 in decimal

Tag = 111

The address mapping for an address: 00001011

Block Address of Main Memory		Address of a word in a Block
000	010	11
000	010	11
Tag	Line Number	

Line Number = 010 = 2 in decimal

Tag = 000

The following cache memory that uses direct mapping shows these two words (along with complete block in the cache)

Line Number of Cache in Decimal	Contents of Cache Memory				
	Tag of Data	Data in Cache = 4 words = 32 bits			
		Word 11	Word 10	Word 01	Word 00
0					
1					
2	000	0011010	0101010	0001010	1101010
3					
4					
5					
6					
7	111	0101110	0101011	1111011	1111010

Figure 6.10: An example Cache memory with Direct mapping

The access for an address: 00011110

Block Address of Main Memory		Address of a word in a Block
000	111	10
000	111	10
Tag	Line Number	

In case, a word like 00011110 is to be accessed, which is not in the cache memory and as per mapping should be mapped to line number 111 = 7, the cache access logic will compare the tags, which are 000 for this address, and 111 in the cache line 7. This is the situation of cache miss, so accordingly this block will replace the content stored in line 7, which after replacement is shown below:

7	000	0001110	0000011	0000011	0000010
---	-----	---------	---------	---------	---------

(ii) Associative Mapping Cache:

The size of cache = 32 bytes

The block size of main memory = words in one line of cache = 4 $\Rightarrow k=2$ bits
Therefore, cache has = $32 / 4 = 8$ lines with each line storing 32 bits of data (4 words)

Tag size = $n-k = (8 - 2) = 6$

The address mapping for an address: 11111101

Block Address of Main Memory	Address of a word in a Block
111111	01
111111	01
Tag	

The address mapping for an address: 00001011

Block Address of Main Memory	Address of a word in a Block
000010	11
000010	11
Tag	

The following associative cache shows these two words.

Line Number of Cache in Decimal	Contents of Cache Memory				
	Tag of Data	Data in Cache = 4 words = 32 bits			
		Word 11	Word 10	Word 01	Word 00
0	111111	0101110	0101011	1111011	1111010
1	000010	0011010	0101010	0001010	1101010
2	000111	0001110	0000011	0000011	0000010
3					
4					
5					
6					
7					

Figure 6.11: An example Cache memory with Associative mapping

The access for an address: 00011110

Block Address of Main Memory	Address of a word in a Block
000111	10
000111	10
Tag Number	

A word like 00011110 can be in any cache line, for example, in the cache memory shown above it is in line 2 and can be accessed.

(iii) 2way set associative Mapping:

The size of cache = 32 bytes

The block size of main memory = words in one line of cache = 4 $\Rightarrow k=2$ bits

The number of lines in a set (w) = 2 (this is a 2 way set associative memory)

The number of sets (v) = Size of cache in words/(words per line $\times w$)
 $= 32/(4 \times 2) = 4$

Thus, set number can be identified using 2 bits as $2^2 = 4$

Tag size = $(n-k)-v = (8 - 2) - 2 = 4$

The address mapping for an address: 11111101

Block Address of Main Memory		Address of a word in a Block
1111	11	01
1111	11	01
Tag	Set Number	

Set number = 11 = 3 in decimal

Tag = 1111

The address mapping for an address: 00001011

Block Address of Main Memory		Address of a word in a Block
0000	10	11
0000	10	11
Tag	Set Number	

Set number = 10 = 2 in decimal

Tag = 0000

Contents of Cache Memory Way 0					Set #	Contents of Cache Memory Way 1				
Tag	Word 11	Word 10	Word 01	Word 00		Tag	Data in Cache = 4 words = 32 bits			
	Word 11	Word 10	Word 01	Word 00	Word 11	Word 10	Word 01	Word 00	Word 11	Word 10
					0					
					1					
					2	0000	0011010	0101010	0001010	1101010
1111	0101110	0101011	1111011	1111010	3	0001	0001110	0000011	0000011	0000010

Figure 6.12: An example Cache memory with Set Associative mapping

The access for an address: 00011110

Block Address of Main Memory		Address of a word in a Block
0001	11	10
0001	11	10
Tag	Set Number	

Set number = 11 = 3 in decimal

Tag = 0001

Word 00011110 can be stored and accessed from the cache set 11 at the second line (way 1).

6.4.3 Write Policy

Many processes read and write data in cache and main memory either by the processor or by the input/ output devices. Multiple read possess no challenge to the state of the data item, as you know, cache maintains a copy of frequently required data items to improve the system performance. Whenever a process writes/ updates the values of the data item in cache or in main memory, it must be updated in the copy as well. Otherwise it will lead to an inconsistent data and cache content may become invalid. Problems associated with writing in cache memories can be summarised as:

- Caches and main memory can be altered by multiple processes which may result in inconsistency in the values of the data item in cache and main memory.

- If there are multiple CPUs with individual cache memories, data item written by one processor in one cache may invalidate the value of the data item in other cache memories.

These issues can be addressed in two different ways:

1. **Write through:** This writing policy ensures that if a CPU updates a cache, then it has to write/ or make the changes in the main memory as well. In multiple processor systems, other CPUs-Cache need to keep an eye over the updates made by other processor's cache into the main memory and make suitable changes accordingly. It creates a bottleneck as many CPUs try to access the main memory.
2. **Write Back:** Cache control logic uses an *update bit*. Changes are allowed to write only in cache and whenever a data item is updated in the cache, the update bit of the block is set. As long as data item is in the cache no update is made in the main memory. All those blocks whose update bit is set is replaced in the main memory at the time when the block is being replaced in the cache. This policy ensures that all the accesses to the main memory are only through cache, and this may create a bottleneck.

You may refer to further readings for more details on cache memories.

Check Your Progress 2

1. Assume that a Computer system have following memories:
RAM 64 words with each word of 16 bits
Cache memory of 8 Blocks (block size of cache is 32 bits)
Find in which location of cache memory a decimal address 21 can be found if Associative Mapping is used.

.....
.....

2. For the system as given above, find in which location of cache memory a decimal address 27 will be located if Direct Mapping is used.
3. For the system as given above, find in which location of cache memory a decimal address 12 will be located if two way set associative Mapping is used.

.....
.....

6.5 MEMORY INTERLEAVING

As you know that cache memory is used as a buffer memory between processor and the main memory to bridge the difference between the processor speed and access time of the main memory. So, when processor requests a data item, it is first looked into the cache and if data item is not present in the cache (called cache miss), only then main memory is accessed to read the data item. To further enhance the performance of the computer system and to reduce the memory access time of the main memory, in case of cache miss, the concept of memory interleaving is used. Memory interleaving is of three type, viz. lower order memory interleaving, higher order memory interleaving and hybrid memory interleaving. In this section we will discuss the lower order memory interleaving only. Discussion on other memory interleaving techniques is beyond the scope of this unit.

In memory interleaving technique, main memory is partitioned into n number of equal sized modules called as memory banks and technique is known as *n-way memory*

interleaving. Where each memory module has its own memory address register, base register and instruction register, thus each memory bank can be accessed individually and simultaneously. Instructions of a process are stored in successive memory banks. So, in a single memory access time n successive instructions of the process can be accessed from n memory banks. For example, suppose main memory is divided into four modules or memory banks denoted as M1, M2, M3 and M4 then first n instructions of a process will be stored as: first instruction in M1, second instruction in M2, third instruction in M3, fourth instruction in M4 and again fifth instruction in M1 and so on.

When processor issues a memory fetch command during the execution of the program, memory access system creates n consecutive memory addresses and places them in respective memory address register of all memory banks in the right order. Instructions are read from all memory modules simultaneously and loads them into n instruction registers. Thus, each fetch for a new instruction results in the loading of n consecutive instructions in n instruction registers of the CPU, in the time of a single memory access. Figure 6.13 shows the structure of 4-way memory interleaving. The address is resolved by interpreting the least significant bits to select the memory module, and rest of the most significant bits are the address in the memory module. For example, in an 8-bit address and 4-way memory interleaving, two least significant bits will be used for module selection and six most significant bits will be used as an address in the module.

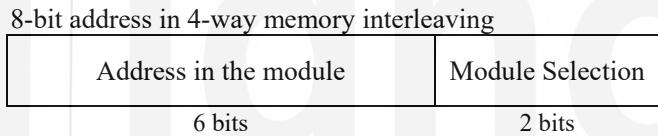


Figure 6.13: Address mapping for Memory interleaving

The following example demonstrates how the main memory words be distributed to different interleaved memory modules. For this example, only a four bit address of main memory is used.

Main Memory		Module 00		Module 01	
Address	Data	Address	Data	Address	Data
0000	10	00	10	00	20
0001	20	01	50	01	60
0010	30	10	46	10	25
0011	40	11	23	11	78
0100	50				
0101	60				
0110	80				
0111	76				
1000	46				
1001	25				
1010	58				
1011	100				
1100	23				
1101	78				
1110	35				
1111	11				

Module 10		Module 11	
Address	Data	Address	Data
00	30	00	40
01	80	01	76
10	58	10	100
11	35	11	11

Figure 6.14: Example of Memory interleaving

6.6 ASSOCIATIVE MEMORY

Though cache is a high speed memory but still it needs to search the data item stored in it. Many search algorithms have been developed to reduce the search time in a sequential or random access memory. Searching time of a data item can be reduced further to a significant amount of data item is identified by its content rather by the address. Associative memory is a content addressable memory (CAM), that is memory unit of associative memory is addressed by the content of the data rather by the physical address. The major advantage of this type of memory is that memory can be searched in parallel on the basis of data. When a data item is to be read from an associative memory, the content of the data item, or part of it, is specified. The memory locates all data items, which matches the specified content, and marks them for reading. Because of the architecture of the associative memory, complete data item or a part of it can be searched in parallel.

Hardware Organization

Associative memory consists of a memory array and logic for m words with n bits per word as shown in block diagram in Figure 6.15. Both argument register (A) and key register (K) have n bits each. Each bit of argument and key register is for one bit of a word. The match register M has m bits, one each for each memory word.

The key register provides a mask for choosing a particular field or key in the argument word. The entire argument is compared with each memory word only if the key register contains all 1s. Otherwise, only those bits in the argument that have 1s in their corresponding positions of the key register are compared. Thus, the key provides a mask or identifying information, which specifies how reference to memory is made.

The content of argument register is simultaneously matched with every word in the memory. Corresponding bits in the mach register is set by the words that have match with the content of the argument register. Set bits of the matching register indicates that corresponding words have a match. Thereafter, memory is accessed sequentially, to read only those words whose corresponding bits in the match register have been set.

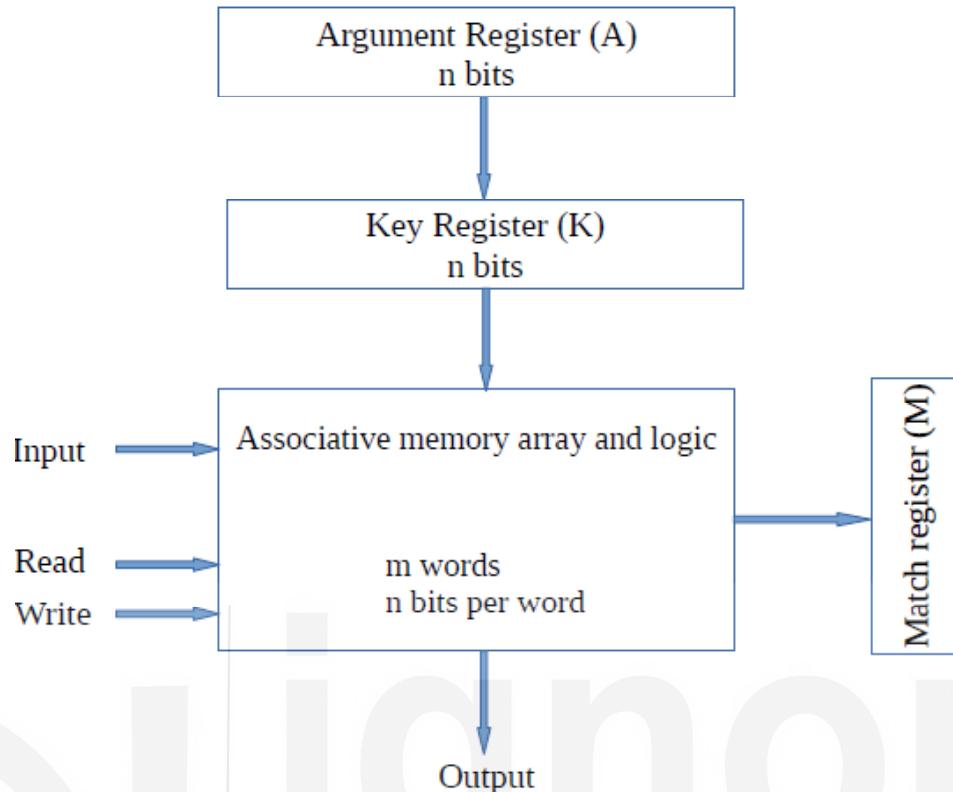


Figure 6.15: Block diagram of associative memory

Example: Consider an associative memory of just 2 bytes. The content register and argument registers are also shown in the diagram.

Description	The content of associative Memory								Match Word
Argument Register	0	1	1	0	0	0	0	1	
Key Register	1	1	1	1	0	0	0	0	
Bits to be matched	0	1	1	0					
Word 1	0	1	1	0	0	1	1	0	Match
Word 2	1	0	0	1	1	0	0	0	Not macheted

Figure 6.16: An Example of Associative matching

Please note as four most significant bits of key register are 1, therefore only they are matched.

6.7 VIRTUAL MEMORY

As we know that a program is loaded into the main memory for execution. The size of the program is limited by the size of the main memory i.e. cannot load a program in to the main memory whose size is larger than the size of the main memory. Virtual memory system allows users to write programs even larger than the main memory. Virtual memory system works on the principle that portions of a program or data are loaded into the main memory as per the requirement. This gives an illusion to the programmer that they have very large main memory at their disposal. When an address is generated to reference a data item, virtual address generated by the processor is mapped to a physical address in the main memory. The translation or mapping is handled automatically by the hardware by means of a mapping table.

Let us say, you have a main memory of size 256K (2^{18}) words. This requires 18-bits to specify a physical address in main memory. A system also has an auxiliary memory as large as the capacity of 16 main memories. So, the size of the auxiliary memory is $256K \times 16 = 4096$ K which requires 24 bits to address the auxiliary memory. A 24-bit virtual address will be generated by the processor which will be mapped into an 18-bit physical address by the address mapping mechanism as shown in Figure 6.17.

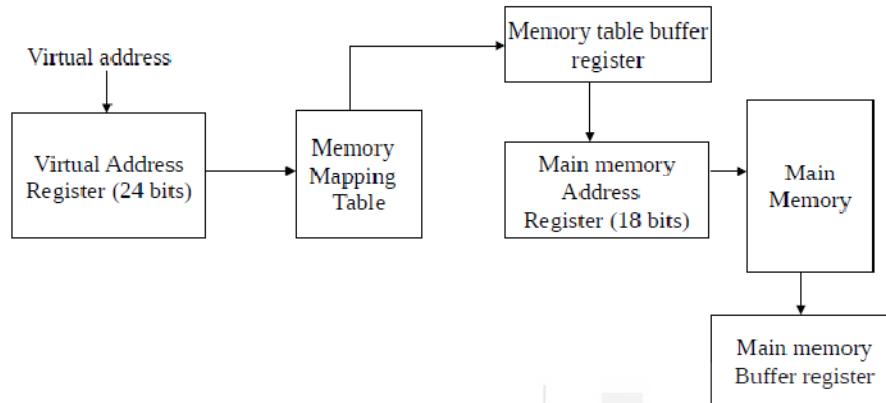


Figure 6.17: Virtual Address Mapping to Physical Address

In a multiprogramming environment, programs and data are transferred to and from auxiliary memory and main memory based on demands imposed by the processor. For example program 1 is currently being executed by the CPU. Only program 1 and a portion of its associated data as demanded by the processor are loaded from secondary memory into the main memory. As programs and data are continuously moving in and out of the main memory, space will be created and thus both program or its portions and data will be scattered throughout the main memory.

Check Your Progress 3

- How can interleaved memory be used to improve the speed of main memory access?
-
.....

- Explain the advantages of using associative memory?
-
.....

- What is the need of virtual memory.
-
.....

6.8 SUMMARY

This unit introduces you to the concept relating to cache memory. The unit defines some of basic issues of cache design. The concept of cache mapping schemes were explained in details. The direct mapping cache uses simple modulo function, but has limited use. Associative mapping though allows flexibility but uses complex circuitry and more bits for tag field. Set-associative mapping uses the concept of associative and direct mapping cache. The unit also explains the use of memory interleaving, which allows multiple words to be accessed in a single access cycle. The concept of

content addressable memories are also discussed. The cache memory, memory interleaving and associative memories are primarily used to increase the speed of memory access. Finally, the unit discusses the concept of virtual memory, which allows execution of programs requiring more than physical memory space on a computer. You may refer to further readings of the block for more details on memory system.

6.9 ANSWERS

Check Your Progress 1

1. While executing a program during a period of time or during a specific set of instructions, it was found that memory reference to instructions and data tend to cluster to a set of memory locations, which are accessed frequently. This is referred to as locality of reference. This allows you to use a small memory like cache, which stores the most used instructions and data, to enhance the speed of main memory access.
2. Typical block size of main memory for cache transfer may be 1, 2, 4, 8, 16, 32 words.
3.
$$\begin{aligned} \text{effective access time} &= 0.9 (\text{hit ratio}) \times 10 (\text{cache hit time}) \\ &\quad + 0.1 (\text{miss ratio}) \times (50+10) \quad (\text{cache miss and memory reference}) \\ \text{effective access time} &= 0.9 \times 10 + 0.1 \times 60 \\ &= 9 + 6 \\ &= 15 \text{ ns} \end{aligned}$$

Check Your Progress 2

1. Main memory size = 64 words (a word = 16 bits) = $2^6 \Rightarrow n=6$ bits
 Block Size = 32 bits = 2 words = $2^1 \Rightarrow k=1$ bit
 The size of cache = 8 blocks of 32 bits each = 8 lines
 Tag size for associative mapping = $n-k = (6 - 1) = 5$
 The address mapping for an address: 21 in decimal that is 010101

Block Address	Address of a word in a Block
01010	1
01010	1
Tag	

In set associative memory the given tag can be stored in any of the 8 lines.

2. Main memory size = 64 words (a word = 16 bits) = $2^6 \Rightarrow n=6$ bits
 Block Size = 32 bits = 2 words = $2^1 \Rightarrow k=1$ bit
 The size of cache = 8 blocks of 32 bits each = 8 lines $\Rightarrow m=3$ bits
 Tag size for direct mapping = $(n-k) - m = (6 - 1) - 3 = 2$
 The address mapping for an address: 27 in decimal that is 011011

Block Address of Main Memory	Address of a word in a Block
01 101	1
01 101	1
Tag	Line Number

The required word will be found in line number 101 or 5 (decimal)

3. Main memory size = 64 words (a word = 16 bits) = $2^6 \Rightarrow n=6$ bits
 Block Size = 32 bits = 2 words = $2^1 \Rightarrow k=1$ bit

The number of sets (v) = 4 sets of 2 lines each, thus, $d = 2$

Tag size for direct mapping = $(n-k) - d = (6 - 1) - 2 = 3$

The address mapping for an address: 12 in decimal that is 001100

Block Address of Main Memory		Address of a word in a Block
001	10	0
001	10	0
Tag	Set Number	

Set number = 10 = 2 in decimal

Thus, required word can be in any of the line in set number 2.

Check Your Progress 3

- Memory interleaving divides the main memory into modules. Each of these module stores the words of main memory as follows (example uses 4 modules and 16 word main memory).

Module 0: Words 0, 4, 8, 12 Module 1: Words 1, 5, 9, 13

Module 2: Words 2, 6, 10, 14 Module 3: Words 3, 7, 11, 15

Thus, several consecutive memory words can be fetched from the interleaved memory in one access. For example, in a typical access words 4, 5, 6, and 7 can be accessed simultaneously from the Modules 0, 1, 2 and 3 respectively.

- Associative memory do not use addresses. They are accessed by contents. They are very fast.
- Virtual memory is useful, when large programs are to be executed by a computer having smaller physical memory.

Structure

- 7.0 Introduction
- 7.1 Objectives
- 7.2 Input/Output (I/O) Devices
- 7.3 The Input/Output (I/O) Interface
 - 7.3.1 System Bus and I/O Interface Modules
 - 7.3.2 I/O and Memory Bus
 - 7.3.3 Isolated and Memory-Mapped I/O
- 7.4 Device Controllers
 - 7.4.1 Device Controller and I/O Interface
- 7.5 Device Drivers
- 7.6 Asynchronous Data Transfer
 - 7.6.1 Strobe Control
 - 7.6.2 Handshaking
- 7.7 Input/Output (I/O) Techniques
 - 7.7.1 Programmed I/O
 - 7.7.2 Interrupt-Driven I/O
 - 7.7.3 Interrupt Handling
 - 7.7.4 Direct Memory Access (DMA)
- 7.8 Input Output Processor (IOP)
 - 7.8.1 Characteristics of I/O Channels
- 7.9 External Communication Interfaces
- 7.10 Summary
- 7.11 Solutions / Answers

7.0 INTRODUCTION

In the preceding units, you have learned the concepts of the memory system for a computer system. The memory system of a computer includes primary, secondary and auxiliary, and high-speed memories. As discussed, the main memory of the computer system is used for storing the instructions and data of the programs, which are getting executed. To execute the program, the computer may need some data which is known as input. The program execution results in the creation of processed data, which is known as output. In addition to the memory system, another important component is the input and output system which is used to receive/send data from/to the external environment. This unit introduces you to various Input/Output (I/O) techniques and controllers, device drivers, structure of I/O interface, and asynchronous data transfer. This unit also explains the I/O processor which is exclusively used for I/O operations.

7.1 OBJECTIVES

After going through this unit, you should be able to:

- define the structure of input/output (I/O) interface and I/O devices;
- explain the structure of controllers;
- explain different data transfer modes;
- explain various techniques used for Input/Output in a computer system;
- discuss the need of an input/output (I/O) processor;
- explain the role of external serial and parallel communication interfaces;
- explain the concepts of interrupt processing

7.2 INPUT/ OUTPUT (I/O) DEVICES

The input/output devices are used by a computer system to provide an interface with the external environment i.e., humans and other devices, which are connected to a computer. The major components of a typical microcomputer system are *microprocessor/CPU, memory*

system and an *I/O interface*. These components are connected through buses. The primary function of the system bus is to transfer control information, addresses, instructions, and data between these components. Figure 7.1 depicts the block diagram of a typical microcomputer system.

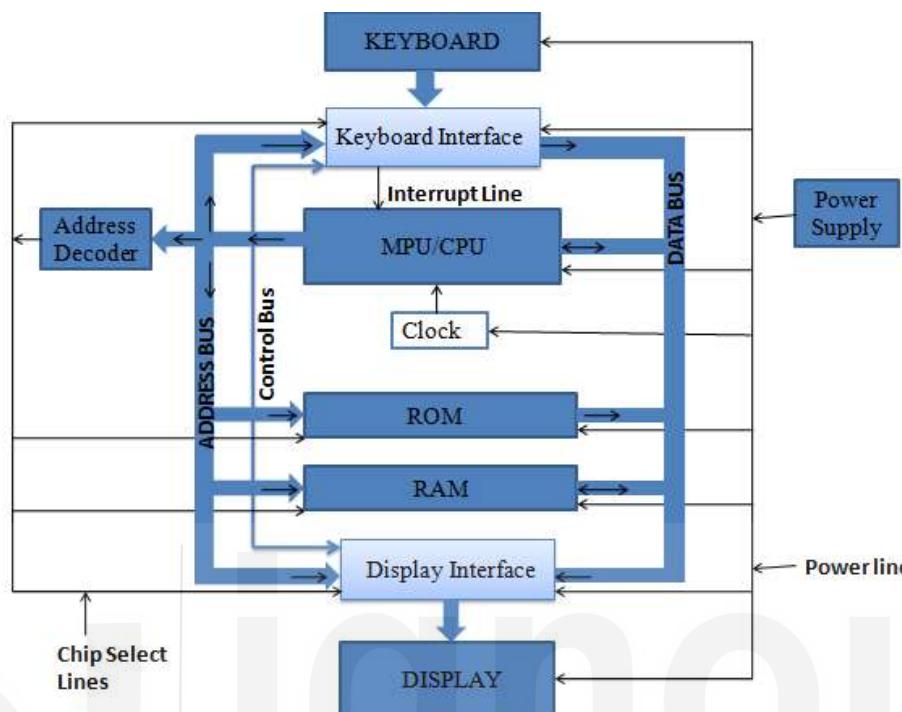


Figure 7.1: A Typical Microcomputer System's Block Diagram

The microcomputer, as shown in Figure 7.1, has a single micro processing unit (MPU). It also has RAM and ROM, which may be constructed by a number of RAM and ROM chips. The diagram also shows two basic interfaces, viz. keyboard interface and display interface, which are connected through the system bus. You may please note that other I/O interface may be connected in a similar way. Please also note that the system bus has been shown in the diagram as three distinct buses, viz. control bus, address bus, and data bus.

An Input/Output (I/O) subsystem includes all the input/output interfaces and connected Input/output devices. The basic objective of an I/O subsystem is to provide an efficient communication medium between the computer system and the external environment (humans and other devices). An I/O interface is used to connect an external I/O device with the computer system. The I/O interface interchanges control, status and data with the external device. The I/O interfaces can also be used to transfer instruction/data/control within the computer units, including processor registers and memory units. An I/O device attached to the computer is also known as **peripheral or external device**. The external devices may be categorized on the basis of their communication endpoints as:

- **Human readable:** These devices provide information in human readable form. Examples—*display terminals, printers, etc.*
- **Machine-readable:** These devices provide information in machine readable form. Examples—*magnetic disks, CD-RW, etc.*
- **Communication:** These devices provide information to communication devices such as *MODEM*.

7.3 THE INPUT/OUTPUT (I/O) INTERFACE

An I/O interface (also known as **I/O Module**) is used to transfer information between internal memory and peripheral devices. Each peripheral device has its own set of characteristics. An I/O interface helps in resolving the differences between the processing unit components and peripherals. The following table lists major differences:

Table 7.1: Peripheral devices and processing unit components

Sr. No.	Processing unit components	Peripheral Devices	Remarks
1	Processor and memory are electronic devices.	Peripheral devices are electromagnetic devices.	Hence, they perform operations in a different manner
2	Data transfer rate of processor is very fast.	Data transfer rate of peripheral devices is slow.	Thus, a synchronization mechanism is required.
3	Processor uses word format.	Peripherals, in general, use bytes/blocks format.	
4	Processor may communicate either directly with different manners or indirectly using an interface in similar fashion.	Each peripheral device may have a different operating mode.	Thus, an interface to handle different operating modes is required.
5		Each peripheral device must be controlled in such a way that the operation of one does not disturb the operation of another peripheral device.	

To address the issues mentioned in Table 7.1, a hardware component between the peripheral devices and CPU is required to control and synchronize all input/output operations in the computer systems. These hardware components, which are used to provide an interface between the peripheral devices and the processor, are known as **interface units**. An I/O interface acts as a bridge between the processor and peripheral devices.

The I/O interface offers an interface which connects the internal components i.e., processor and main memory as well as external components i.e., peripheral or external devices. In addition to data transfer between processor to I/O devices, it also establishes the coordination between them. Moreover, the I/O interface also has components like **buffer system** and **error detection** mechanism to deal with the speed differences between processor and peripheral devices.

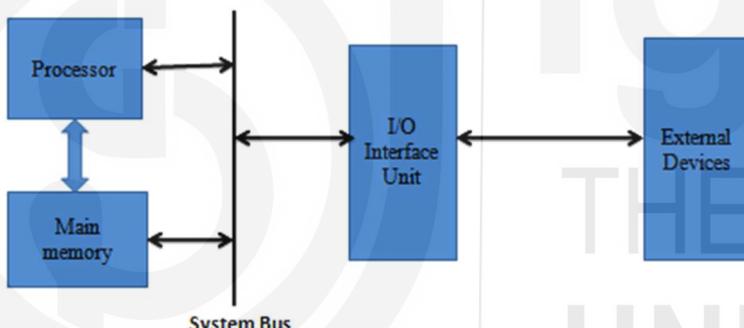


Figure 7.2: A Block Diagram of an I/O Interface connecting with Processor and External Devices

Roles of I/O Interface

The major roles of I/O interfaces are:

1. Communication with the processor

To provide data transfer between processor and I/O interface, the control signals (Read/Write/Status), address of memory locations and actual data are required to exchange. The system bus works as a communication medium in order to facilitate the exchange of information/data between the processor and I/O interface. In system bus, the address bus is used to send addresses; the control bus for control signals whereas the data bus is employed for actual data transfer between processor and I/O interface.

2. Synchronization of control and timing signals

The I/O interface shares system bus with memory in order to provide data transfer. The timing and control signals are required in order to synchronize the flow of data from peripheral devices to processor/memory and vice versa. For instance-An I/O interface may request for control of data bus for data transfer from a peripheral device to the processor.

3. Communication with the I/O device

To complete an I/O operation, the exchange of data between I/O device and I/O interface is necessary. An I/O operation may require sending status signals, commands, or data.

4. Provision for data buffering

Due to the speed mismatch between the I/O devices and processor, the facility for data storage for a temporary period is required. The I/O interface stores information in registers temporarily which is referred to as data buffering. An I/O operation occurs in short bursts and the data are temporarily stored in a buffer area of I/O interface until the peripheral device/processor is ready.

to receive the data. Therefore, it is not required to hold the system bus for I/O operation due to slower I/O devices.

5. Error detection mechanism

The I/O interface also provides an in-built mechanism for error detection to check the communication errors along with mechanical errors. The errors are reported to the processor using parity bits and other mechanisms for further actions. Some mechanical errors may occur in devices such as mechanical and electrical failures, printer's paper jam etc.

In Figure 7.3, the internal block diagram of I/O interface unit consisting of different signals for processor as well as for I/O device is shown.

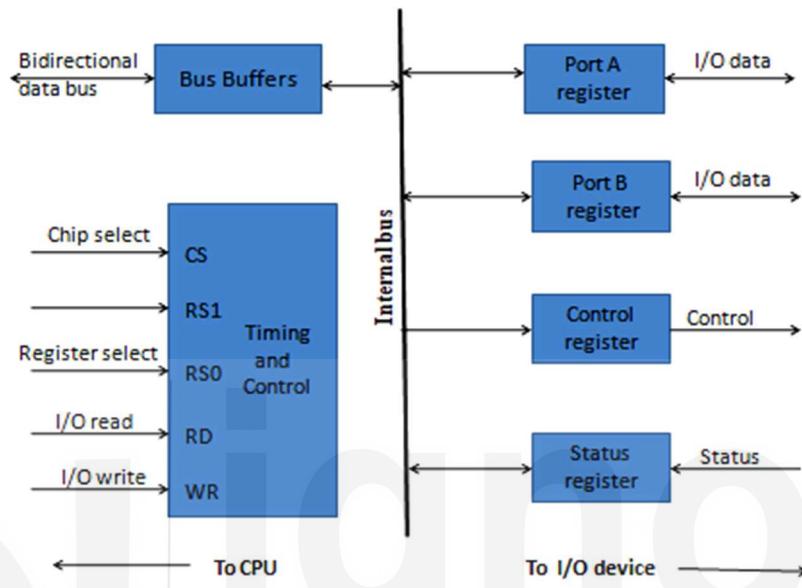


Figure 7.3: I/O Interface Unit's Block Diagram

7.3.1 System Bus and I/O Interface Modules

The control bus, data bus and address bus form a single bus which is known as system bus and it is used to establish the connection between the processor and I/O devices. Figure 7.4 depicts the communication links between the processor and several peripheral devices.

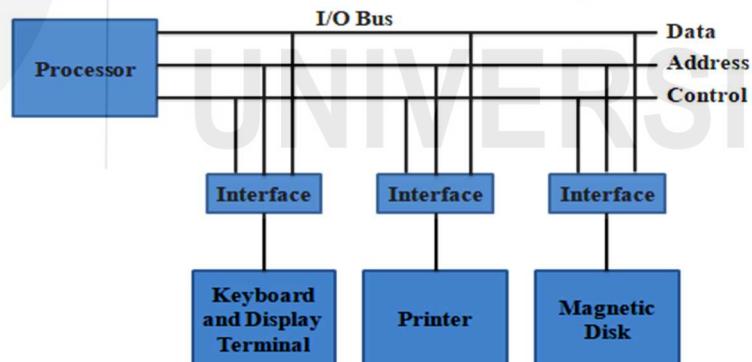


Figure 7.4: Communication links between Processor and Peripheral devices

The I/O bus is connected to all peripheral devices through I/O interfaces. In order to communicate with an intended device, the processor sends the address of the device using address bus. Every I/O interface consists of an address decoder to continuously monitor the content of address bus. When an I/O interface observes the address of its own peripheral device, it activates the associated device and the bus; otherwise, the peripheral device is disabled. A control signal (*I/O command*) is also provided simultaneously through the control bus. The four types of I/O commands that are arising out of the processor are as follows:

1. **Control Command:** This is the control signal code which is sent to the corresponding peripheral device and informs it about the action it has to perform.
2. **Status Command:** This is the status signal which is used to test status conditions of the peripheral devices and interface. Some status commands are BUSY, DATA AVAILABLE, ERROR or NOT IN BUFFER etc.

3. **Data Output Command:** The signal/command is utilized to activate the I/O interface for data transfer from the processor to buffer of I/O interface. The data from the buffer is ultimately sent to the peripheral device. The data is sent from the CPU to the buffer of interface after this command is provided.
4. **Data Input Command:** The processor sends this signal whenever there is a need to read data from any peripheral device. After the issuance of this command, the data from the intended peripheral device are extracted into the interface's buffer and this is followed by the data read operation by the processor.

7.3.2 I/O and Memory Bus

A processor requires communicating with the I/O devices and memory system. The system bus (I/O Bus & Memory Bus) is used to control the exchange of data among the processor, memory system and I/O devices. The I/O bus and memory bus; both buses consist of data, address and control lines. To establish communication with the memory and I/O devices, the system bus can be used as follows:

- i. One bus for each memory system and I/O.
- ii. Shared data and address buses for both I/O devices and memory system but exclusive control bus for both.
- iii. Shared system bus for both memory system and I/O devices.

7.3.3 Isolated and Memory-Mapped I/O

In case of isolated I/O, the data and address buses are shared between I/O and memory but separate read/write control lines are used for I/O devices. Whenever the processor decodes instruction for an I/O device, it places the address on the address line and activates I/O read or write control line which causes data transfer between CPU and I/O device.

In other alternative, the computer employs the same set of read/write signals for I/O and memory and does not differentiate between I/O and memory addresses. This configuration is known as memory-mapped I/O.

7.4 DEVICE CONTROLLERS

The system bus is used to establish communication between the processor and all components including I/O devices. Some components are directly connected to the processor through system while some components are not directly connected to system bus. The intermediate electronic device, known as *device controller*, is used to connect the I/O device and the system bus. The I/O device is connected at one end while it is connected with the system bus on another end. Thus, a device controller works as an interface to provide the communication medium between the system bus and I/O device.

Device Controller

It is not necessary that a device controller controls a single peripheral device. It can generally control more than peripheral devices. The device controller is available as an electronic circuit board, which can be directly plugged into the system bus. The device controller is also connected through a cable with a peripheral device. The connectors at the rear panel of a computer are in fact the end points of these cables. These connectors are called the ports and are used to plugin the external peripheral device to a computer. Figure 7.5 depicts the connection of computer system and I/O devices through device controllers.

In Figure 7.5, the following points are important to consider:

- i. Each peripheral device is attached to a hardware interface known as I/O Port.
- ii. A single-port device controller can control only one peripheral device whereas the multi-port device controller controls multiple peripheral devices.
- iii. In case of direct memory access (DMA), the communication between I/O controller and memory is established through system bus only, while the communication path passes through the CPU for non-DMA communication.

The following are the major benefits for connecting computer system with I/O devices using device controllers:

- i. A single device controller may be used to connect multiple I/O devices with the computer system simultaneously.
- ii. The device controllers allow I/O devices to upgrade/change without any update/change required in the current configuration of a computer system.
- iii. The device controllers allow connecting the I/O devices of different configurations/manufacturers with the computer system. It allows computer users to purchase I/O devices of different configurations/manufacturers.

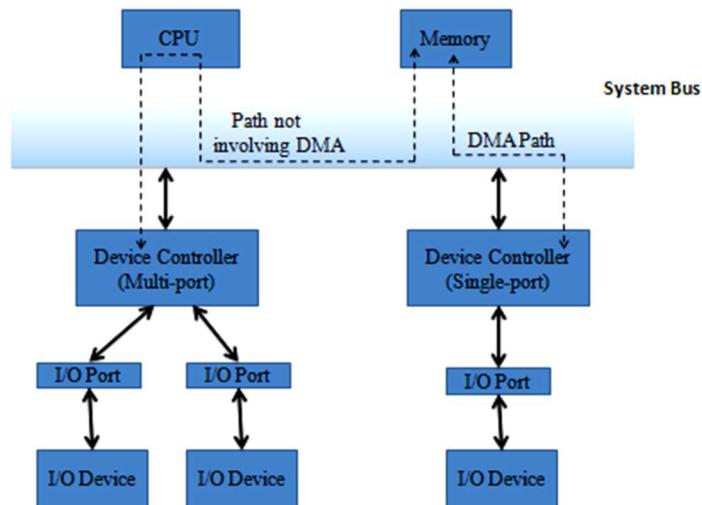


Figure 7.5: Device Controller connecting I/O Devices and Computer System

7.4.1 Device Controller and I/O Interface

Due to the different input/output needs and design complexities, there exist various peripheral devices with different structures. Thus, the common/standard structure of I/O interface does not exist. Figure 7.6 shows a general block diagram of an I/O interface. The common characteristics of the general structure of an I/O interface are as follows:

- i. I/O logic is required in order to decode and execute the data between the processor and I/O interface. Hence, control lines between the I/O interface and processor are required.
- ii. Data lines between the system bus and I/O interface are necessary to facilitate the actual data transfer.
- iii. The data registers may behave like a buffer between processor and I/O interface.
- iv. To deal with each I/O device with different configuration, the I/O interface must have a separate logic specific.

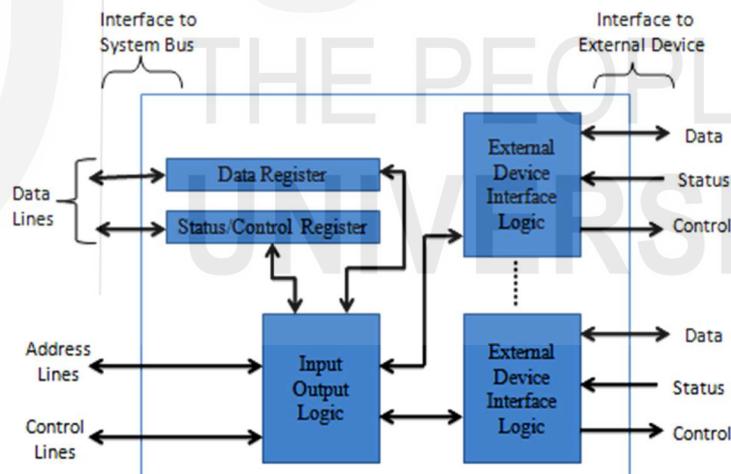


Figure 7.6: Block Diagram of an I/O Interface

7.5 DEVICE DRIVERS

The device driver is a software interface to handle communication with a specific peripheral device. The purpose of the device driver is to decode a logical request from the computer user into a series of specific actions performed by the peripheral device. For instance, a computer user may request to read a record from a disk. The device driver converts this logical request into a series of discrete commands i.e., check the status of the intended disk in the drive, locating the desired file, setting the position of R/E head, etc.

Device Drivers in Windows Systems

The device drivers are realized in the form of dynamic linked libraries (DLLs). The DLL consists of shareable code and thus, a single copy of the DLL code is loaded into the main memory. The hardware/software vendors can implement a device driver for new

devices without modifying or affecting the code of Windows operating system. Moreover, it allows multiple optional drivers to be configured for different peripheral devices.

For Windows based system, the device installation in the form of **Plug and Play** is necessary in order to add new peripheral devices. The objective of **Plug and Play** is to convert the device connecting process from manual to automatic. In automatic device connecting process, the device will be attached which is followed by installation of driver software. After this, the installation process will be automatic and the settings will be changed according to the configuration of the host computer system.

Device Drivers for UNIX Systems

The device drivers are generally linked to the object code of the core of operating system which is known as kernel. To use a new device not included in operating system, the device driver object code has been re-linked with UNIX kernel. The advantages of this technique are simplicity and run-time efficiency while the main disadvantage is it requires regeneration of the kernel in order to attach a new device. Each entry of the /dev directory of the UNIX system is associated with a device driver which in turn is attached with the related device. Table 7.2 consists of information of some device as follows:

Table 7.2: Device Name in UNIX System

Device name	Description
/dev/console	console of a system
/dev/lp	line printer
/dev/tty01	user terminal 1
/dev/tty02	user terminal 2

Check Your Progress 1

1. What do you understand by I/O Interface? List major functions of I/O interface?

.....
.....
.....

2. What is the need of device controller? What are the major benefits?

.....
.....
.....

3. What is the need of a device driver? Give examples of device drivers.

.....
.....
.....

7.6 ASYNCHRONOUS DATA TRANSFER

This section discusses the different data transfer modes. There exists two ways to transfer data from sender to receiver in a digital computer system. It is necessary to synchronize the data transfer operations using clock pulses. When the internal registers of the two units i.e. I/O interface and CPU share a common clock, the mode of data transfer between them is known as synchronous data transfer. On the contrary, both of the units i.e., I/O interface and CPU are designed to work independent manner and each unit uses its own private clock in order to establish coordination. This mode of data transfer is known as asynchronous data transfer.

The asynchronous data transfer faces some problems. Since there exists no fixed time slot to send or receive data. Therefore, there is no guarantee that whether the data is old or new. This problem can be solved using the following two methods:

- i. Strobe Control Method
- ii. Handshaking Method

7.6.1 Strobe Control method

In strobe control method a single control line is used each time for data transfer and this control signal is referred to as *Strobe*. The strobe may be activated in the following two different ways:

A. Source-initiated Strobe – Figure 7.7 depicts the block diagram and timing diagram for source-initiated strobe method. In this method, the data transfer is performed as:

- i. Initially, source unit puts the data on data bus and the strobe signal is switched ON.
- ii. The destination unit reads data from the data bus.
- iii. After reading data, the strobe gets OFF.

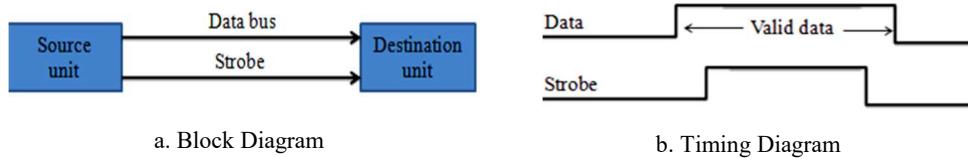


Figure 7.7: Source Initiated Strobe

B. Destination-initiated Strobe – Figure 7.8 depicts the block diagram and timing diagram for destination-initiated strobe method. The steps for data transfer are as follows:

- i. First, the destination unit switches ON the strobe signal.
- ii. After observing strobe ON, the source unit puts data on the data bus.
- iii. The destination unit reads the data from the data bus and the strobe signal gets OFF.



Figure 7.8: Destination Initiated Strobe

Disadvantages of Strobe Method:

The disadvantages of strobe methods are as follows:

- In source-initiated strobe method, it is not possible to confirm the status of data at destination unit i.e., whether the data received is valid or not.
- In destination-initiated strobe method, it is not possible to confirm the status of data at the data bus i.e., whether the data extracted from the data bus is valid or not.

7.6.2 Handshaking method

In handshaking method of asynchronous data transfer, the problems of strobe control method can be handled. The handshaking method can be realized in the following two different ways:

A. Source-Initiated Handshaking – It consists of the following signals:

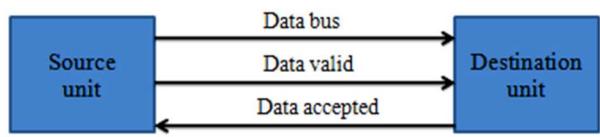
Data Valid: When activates, it specifies that data on the data bus is valid otherwise invalid.

Data Accepted: When activates, it specifies that data is accepted otherwise not accepted.

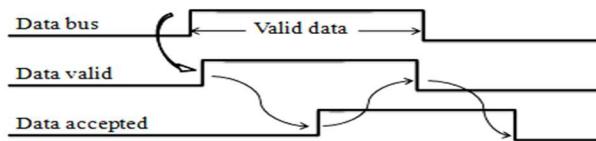
The steps of data transfer in Source Initiated Handshaking method are as follows:

- i. First, source unit copies data on the data bus and enables *Data Valid* signal.
- ii. The destination unit receives data from the data bus and enables *Data Accepted* signal.
- iii. After this, *Data Valid* signal is disabled which implies that data on data bus is invalid now.
- iv. Finally, *Data Accepted* signal is disabled which indicates that the process of data transfer has been completed.

Now, it can be ensured that destination unit has read the valid data from the data bus using *Data Accepted* signal. Figure 7.9 depicts the block diagram and timing diagram of the source-initiated handshaking method.



a. Block Diagram



b. Timing Diagram

Figure 7.9: Source Initiated Handshaking

B. Destination-Initiated Handshaking – It employs the following signals:

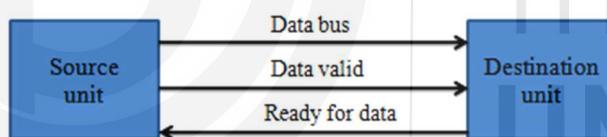
Request for Data: When activated, it requests for putting data on the data bus.

Data Valid: When activated, it specifies data is valid on the data bus otherwise invalid data.

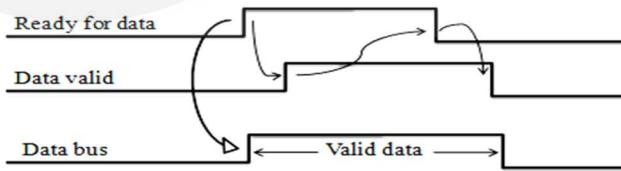
The steps of data transfer in Source Initiated Handshaking method are as follows:

- When destination unit is ready to receive data, the appropriate signal i.e., **Request for Data** gets activated.
- In response, the source unit places data on the data bus as well as it also enables **Data valid** signal.
- Next, the destination unit accepts data from the data bus. After receiving data, it disables **Request for Data** signal.
- Lastly, the **Data valid** signal is disabled implies that the data bus consists of invalid data now.

Using **data valid** signal in **Destination Initiated Handshaking**, it is possible to ensure that source has put the data on the data bus. Figure 7.10 shows the block diagram and timing signal in destination-initiated handshaking. It consists of **Request for Data** as well as **Data valid** signals.



a. Block Diagram



b. Timing Diagram

Figure 7.10: Destination Initiated Handshaking

7.7 INPUT/OUTPUT (I/O) TECHNIQUES

This section explains the methods to use I/O interface to support data transfer (input/output) from peripheral devices. The data transfer (I/O operation) between the computer system and peripheral devices may be performed using three techniques given as follows:

- Programmed I/O
- Interrupt-driven I/O
- Direct memory access (DMA)

Table 7.3 presents important characteristics of the above three techniques.

Table 7.3: Three I/O Techniques

I/O Technique	Data Transfer Path through	Processor Intervention	Interrupt
Programmed I/O	Processor	Continuous	Not-Required
Interrupt-driven I/O	Processor	Discontinuous	Required
Direct memory access (DMA)	Direct to Memory	At the beginning and end	Required

7.7.1 Programmed I/O

In this technique, the processor is completely responsible for managing all I/O operations. The program which requests for I/O operation directly and continuously controls the I/O operation through the processor. The processor runs a program that starts, continuously monitors and ends an I/O operation. The I/O operation may involve the following data transfer operations:

- a. Data transfer from processor registers to I/O device.
- b. Data transfer from I/O device to main memory through processor registers.

The major steps in programmed I/O are as follows:

- i. The processor issues READ/WRITE command.
- ii. The processor requests for I/O operation to I/O interface.
- iii. I/O interface sets status bits.
- iv. The processor continuously checks the status bits.
- v. The processor waits for I/O device.
- vi. When I/O device is ready, I/O operation (Read/Write data) is performed.

Figure 7.11 depicts the diagram of programmed I/O technique. When the CPU issues a command for an Input or an output to the I/O interface, it must wait until the I/O operation is completed. This process is known as polling.

Disadvantage: With the programmed I/O method, the processor continuously checks the status of the I/O device whether it is ready or not. The processor has to wait for slower I/O operations to complete. Therefore, this technique is wasteful of processor's time.

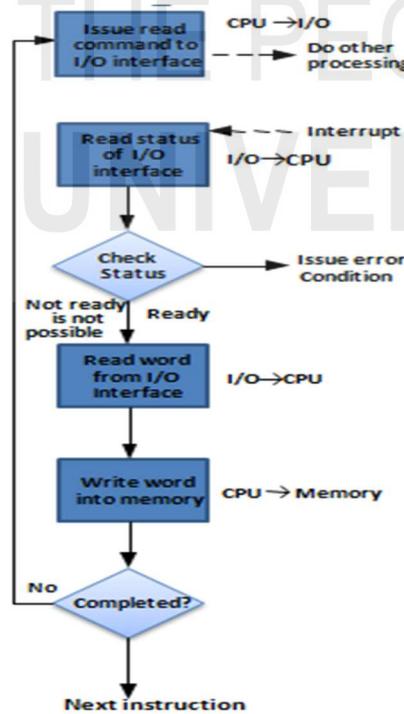


Figure 7.11: Programmed I/O

I/O Commands

Whenever the processor addresses an I/O interface, it sends an address and I/O command. There are four I/O commands which a processor may send to I/O interface for I/O operation. The I/O commands are given as follows:

- **Control:** Control commands are used to activate the device and also specify what operation is to perform. For example- a USE command to make a specific device as current device for read/write operation.
- **Test:** The Test I/O command is used to check the status of a device. For instance- Whether the device is in **error condition**, **ready state**, or **not ready state**.
- **Read:** This command is used to receive one item of input data from the I/O device which is in communication.
- **Write:** This command is used to send one item of output data to the respective output device.

7.7.2 Interrupt-driven I/O

For an I/O operation in programmed I/O, the processor continuously waits for the operation to be completed and the data transfer occurs when the device is ready. This process is known as polling which leads to slow performance of the processor. Interrupt-driven I/O can reduce the polling time efficiently. In interrupt-driven I/O, the processor issues a read/write command and it starts the execution of some other program/instructions. Whenever the desired device is ready or has completed the assigned I/O task, an interrupt signal is sent to processor for further actions. Figure 7.12 depicts the procedure of interrupt-driven I/O technique. The complete list of steps in interrupt-driven I/O is as follows:

- i. The processor issues read /write command
- ii. The processor executes some other program/instructions
- iii. The I/O interface reads data from the desired I/O device
- iv. I/O interface interrupts the processor
- v. The processor checks the interrupt after completing each instruction cycle
- vi. The processor saves the context of the program in execution
- vii. The processor requests for the desired data
- viii. The I/O interface transfers data
- ix. The processor resumes the previous program it had been executing before the interrupt.

Advantage: This technique reduces the overall waiting time of the processor.

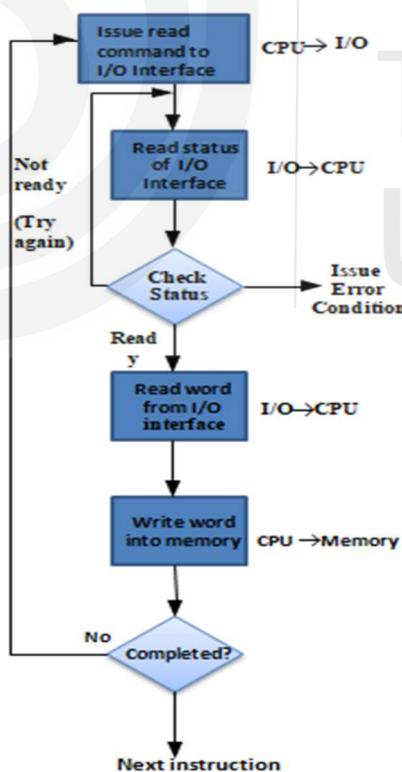


Figure 7.12: Interrupt Driven I/O

7.7.3 Interrupt Handling

An interrupt is an event which may occur to receive some service by a device or due to some error. After an interrupt occurs, it causes multiple tasks in the software as well as in the hardware.

Figure 7.13 represents the series of hardware tasks which occurred in order to I/O operation using an I/O device. The complete lists of tasks required for interrupt processing are as follows:

1. An I/O device sends an interrupt signal to processor.
2. The processor completes the execution of currently running instruction and next, it checks and takes action accordingly.
3. An acknowledgement signal is also sent by the processor to the corresponding I/O device.
4. The processor saves the context of the program that is being executed currently. It saves the following registers:
 - (a) The status of the processor: program status word (PSW) register
 - (b) The next instruction to be executed: program counter (PC) register.
5. In addition to the contents of PC & PSW for the interrupted program, it is required to store the contents of the registers of processor on the stack. Figure 7.14 depicts an example of interrupt occurrence. According to the Figure 7.14, a user running program is interrupted after the execution of the instruction at location N. To handle the interrupt, the processor saves the contents of all general-purpose registers, PSW, PC along with the address of the next instruction at location N+1.
6. The processor loads the PC with the address of first location of the interrupt and then, the processor starts executing instruction from the interrupt-handler program. It means that the interrupt handler handles the interrupt.
7. When the processing of the interrupt is completed. The context of the previous program is restored i.e. contents stored at stack are restored in the processor registers, which is shown in Figure 7.15.
8. Finally, the values of PC and PSW are retrieved from the stack and restored on the corresponding registers. It leads to execution of the instruction which was interrupted from the previously interrupted program.

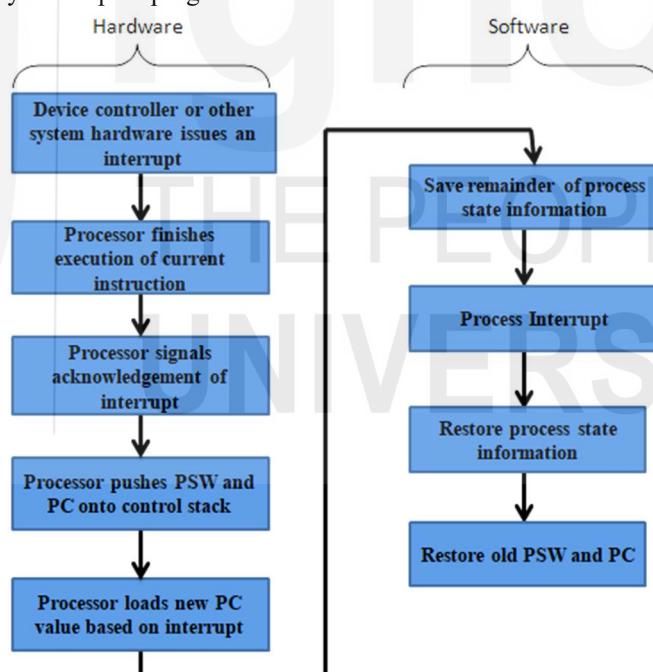


Figure 7.13: The sequence of steps in Interrupt-Processing

In simple words, the interrupt handling consists of the following:

- i. Interrupt the current program
- ii. Saves the context of the interrupted program
- iii. Transfer control to interrupt servicing program
- iv. Execute the interrupt servicing program
- v. Restore the context of interrupted program
- vi. Resume interrupted program from the point where it was interrupting.

Design Issues

To realize the interrupt-driven I/O, there exist two design issues given as follows:

- 1) How does the processor identify the I/O device issuing the interrupt?
- 2) How does the processor handle multiple interrupts occurred simultaneously?

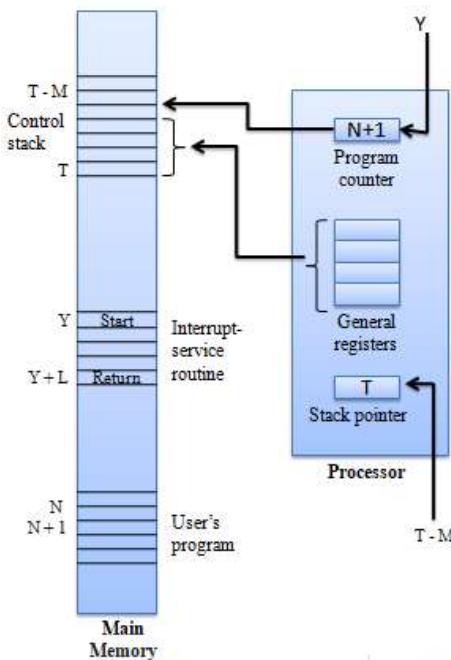


Figure 7.14: Interrupt occur for instruction at memory location N

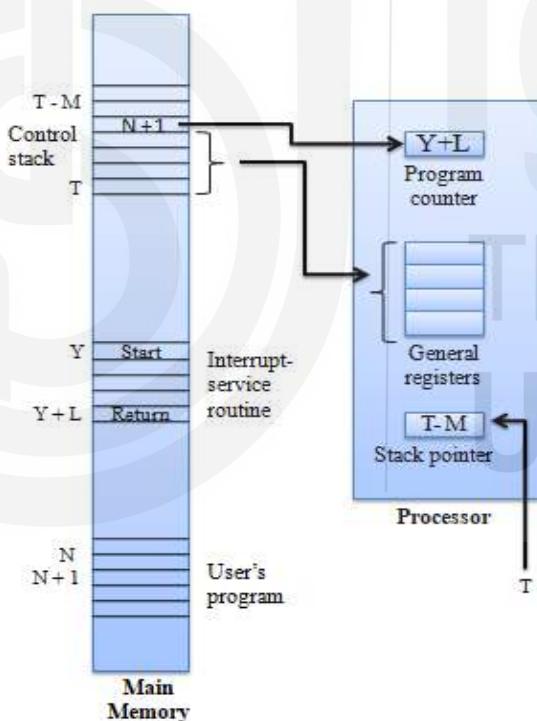


Figure 7.15: Return after handling an Interrupt

The following four general techniques can be used to solve the above design issues:

- Multiple Interrupt Lines:** Multiple interrupt lines can be used to handle multiple interrupts. In this technique, the priorities are assigned to various interrupts. Whenever an interrupt occurs, the highest priority interrupt will be handled first among all interrupts. However, the practical realization of this technique is difficult because computer system provides only a few lines for the interrupt.
- Software Poll:** The processor jumps to an interrupt service routine after an interrupt has occurred. The processor polls at I/O interface by observing the status register in order to identify the I/O interface which caused the interrupt. The software poll technique is time consuming due to polling process.
- Daisy Chain or Hardware Poll:** In this technique, the interrupt request line is shared by all I/O interfaces. Whenever an interrupt occurs, the processor sends an

acknowledgement of interrupt. This acknowledgement goes to all the I/O devices through I/O interface. When it reaches to the I/O device which sends interrupt, the I/O device sends a response by specifying an address or unique identifier through the data bus. This unique identifier helps in deciding the appropriate interrupt servicing program. The hardware poll consists of an in-built priority mechanism and this priority is based on the sequence of devices on interrupt acknowledge line.

- iv. **Bus Arbitration:** The bus arbitration technique allows only one I/O interface to control the bus and this I/O interface is known as bus master. It means only one interrupt request can be fulfilled at the same time. After acknowledgement of interrupt by processor, the I/O interface sends the interrupt vector to processor through data lines. The interrupt is handled according to the number in interrupt vector.

7.7.4 Direct Memory Access (DMA)

The programmed I/O as well as interrupt-driven I/O techniques require the processor's interventions for data transfer to/from the main memory. Since processor may involve in the execution of multiple programs due to multiprogramming, which restricts the processor time for testing the I/O device and servicing the I/O request by transferring I/O data over the system bus. Is it possible to store/retrieve data to/from main memory without involving the processor in the data transfer? Yes! There exists an alternative approach which is referred to as **direct memory access (DMA)**. In DMA, the main memory and I/O interface exchange data directly without the active involvement of processor. Figure 7.16 depicts the block diagram of direct memory access (DMA) technique. Now, an obvious question arises: What is the use of DMA interface? The DMA interface is mainly used to transfer a large quantity of data which is exchanged between I/O device and main memory.

Figure 7.16 depicts the block diagram of DMA technique. The steps of the DMA technique are given as follows:

- i. Processor issues read/write command to DMA module for transferring the block of data
- ii. The processor sends the following information to DMA interface-
 - a. To perform read/write operation, the Read /Write signal is sent using control lines
 - b. I/O Device address is communicated through a data bus
 - c. Beginning address of memory block using data bus
 - d. "The number of words to be transferred" is communicated through data bus. This information is stored in a count register.
- iii. The processor executes some other program.
- iv. Now, DMA controller performs the data transfer
- v. When DMA controller finishes the data transfer, it sends an interrupt signal to processor

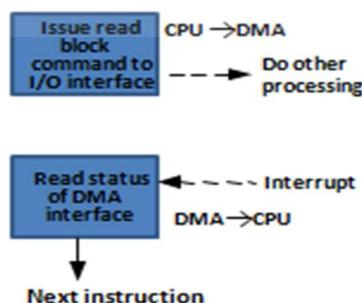


Figure 7.16: DMA Technique

Figure 7.17 shows the block diagram of DMA module along with its different signal lines. The DMA module handles data transfers to send the entire block of data and one word at a time is transferred directly to or from memory without going through the registers of processor. After transferring the entire block, an interrupt signal is sent to processor by DMA module. Therefore, the processor involvement is limited only at the start and end of the I/O operation.

In DMA, the processor intervention is minimized but it must use the path through system bus. Thus, DMA interface requests for system bus to transfer one data word at a time and the control of the system bus is returned to the processor after transferring one word of data. This process is known as **cycle stealing**. The CPU cycle stealing causes the delay to the operation

of processor for one memory cycle. In DMA technique, the active involvement of processor can be restricted at the beginning and at the completion of the I/O operation.

Figure 7.18 depicts the five cycles of typical instruction execution. In Figure 7.18, three points are marked where a processor can respond to DMA request, and; a point is also marked where the processor can respond to interrupt request. The point at which the interrupt request can be acknowledged is called the interrupt cycle.

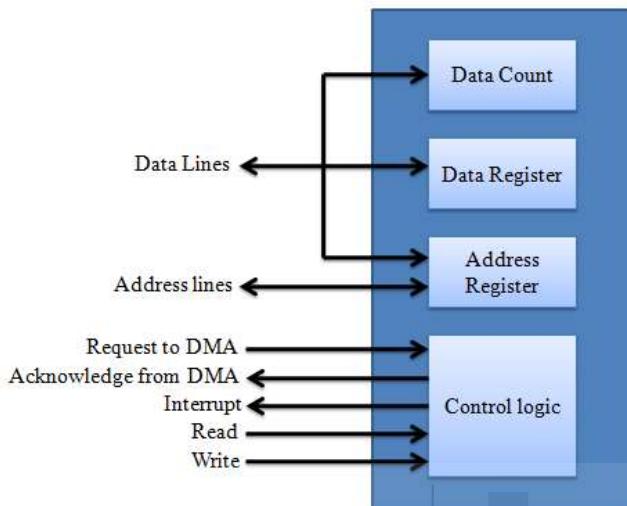


Figure 7.17: DMA's Block Diagram

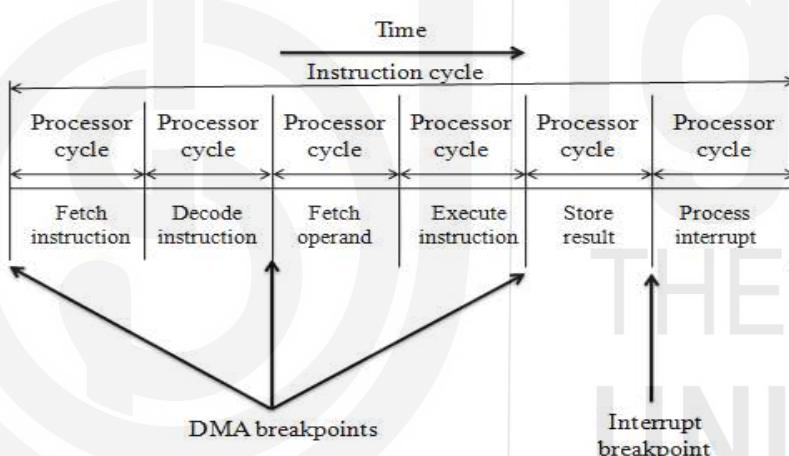
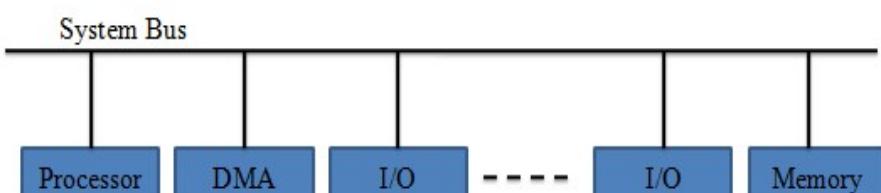
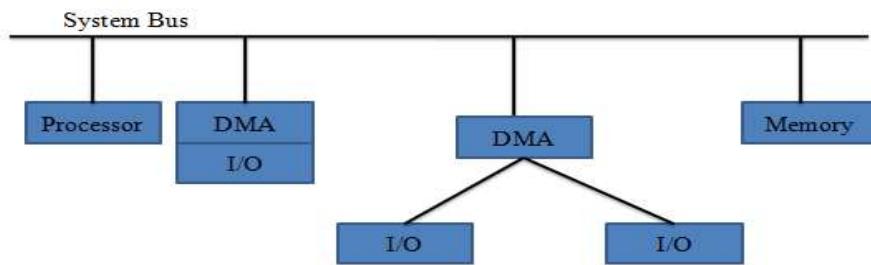


Figure 7.18: Breakpoints for DMA and Interrupt in an Instruction Cycle

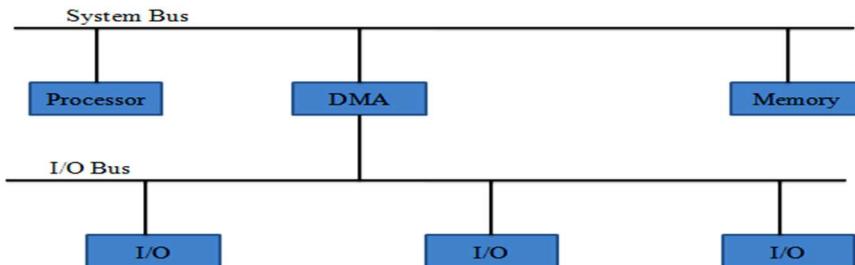
There exist different configurations to realize the DMA mechanism. Figure 7.19(a) shows one of the several configurations. In this configuration, the system bus is shared by all interfaces. The DMA works as a supportive component and may employ programmed I/O for data transfer between memory and I/O interface using DMA interface. The advantage of this configuration and programmed I/O is that DMA does not require extra system bus cycles to transfer information from DMA to/from I/O interface as well as from main memory to/from DMA.



(a) Single Shared System Bus with detached DMA



(b) Single Shared System Bus with integrated DMA-I/O



(c) I/O bus

Figure 7.19: Three Different DMA Configurations

Figure 7.19(b) depicts one more DMA configuration which has some advantages over the first configuration shown in Figure 7.19(a). In this configuration, a direct path is provided between the DMA interface and I/O interface and this path is different than the system bus. Furthermore, DMA logic may act as a constituent of I/O interface and single or multiple I/O interfaces may be controlled by it. One more flexible configuration is shown in Figure 7.19(c) in which the DMA interface is connected to I/O bus. This configuration is extendable in an easy manner. The additional benefit in both configurations is that the data transfer can be performed from DMA interface to I/O interface without the involvement of the system bus.

Check Your Progress 2

- What do you understand by Programmed I/O technique?
-
.....
.....

- Explain interrupt? What processing is performed on the occurrence of an interrupt?
-
.....
.....

- Why is DMA needed? What are its advantages and disadvantages?
-
.....
.....

7.8 INPUT-OUTPUT PROCESSOR (IOP)

The evolution of I/O techniques passes through many development stages and this evolution can be helpful to understand the idea of I/O processors. Various stages of evolution are summarized as follows:

- Direct Control: The processor directly controls I/O device.
- Control using I/O controllers without interrupts: In this configuration, processor and I/O interfaces are different with each other and programmed I/O without interrupts was employed by I/O interface for I/O data transfer.

3. Control using I/O controllers with interrupts: In this configuration, the processor need not to wait for an I/O operation to be completed which leads to improved efficiency of the processor.
4. Control transferred to DMA: In this configuration, the involvement of the processor is restricted at the beginning and on completion of DMA operation. The I/O interface and DMA module control the data transfer directly without the involvement of the processor.
5. Control transferred to I/O processor: On the request of main processor, the I/O processor takes control of I/O operation and performs I/O operation with the intervention of the main processor. This approach allows controlling various I/O devices with minimum involvement of processor.

Through various evolution stages, it is evident that the responsibility of I/O tasks has been shifted from CPU to I/O-processor and it leads to improved performance of the processor.

It is also evident from the recent developments that a major shift occurs due to the introduction of I/O interface which is capable to execute the I/O instructions. Hence, the '*I/O interface*' is often called an **I/O processor** or **I/O channel**.

Input Output Processor (IOP)

An IOP is a processor that handles only the I/O processing. The block diagram of a computer system with input output processor is shown in Figure 7.20. The computer system with I/O processor consists of three major components: (a) *a memory unit* (ii) *the processor* and (iii) *the IOP*. The IOP controls and manages the input-output tasks. On the command of CPU, the IOP can fetch, decode and execute I/O instructions which are loaded to perform the I/O operations.

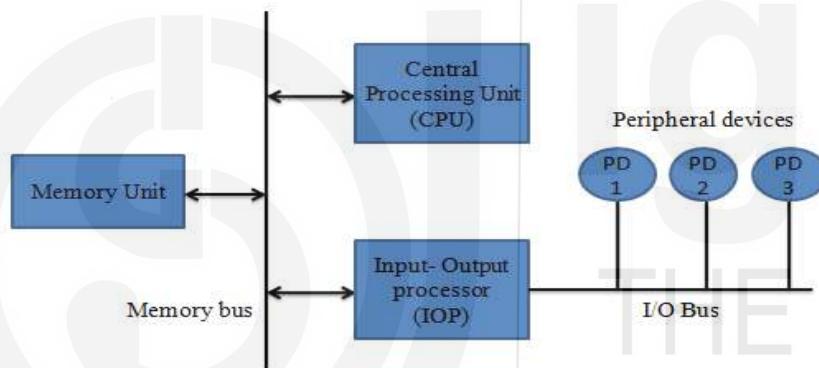


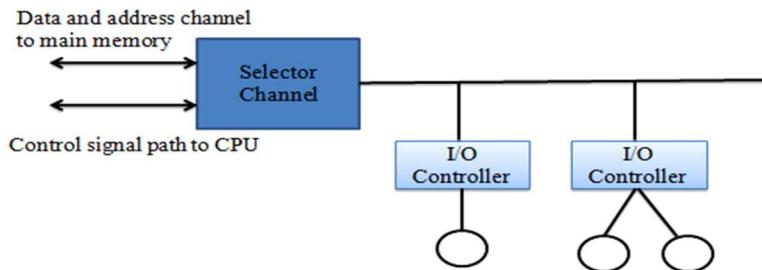
Figure 7.20: Block Diagram of a Computer System with I/O Processor

7.8.1 I/O Processor's Characteristics

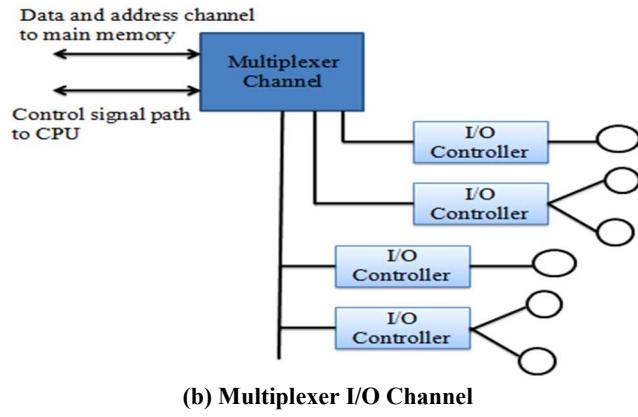
In simple words, an I/O processor (or I/O channel) is the extension of the concept of DMA technique. An I/O channel holds complete control of the I/O operation and is also capable to execute I/O instructions. The CPU initiates the I/O operations and handing over the execution of I/O commands to I/O processor. In response, the I/O processor executes the I/O instructions stored into the main memory.

The I/O channels can be divided into two categories:

- i. Selector I/O channels
- ii. Multiplexer I/O channels.



(a) Selector I/O Channel



(b) Multiplexer I/O Channel

Figure 7.21: Structures of I/O Channels

- A. **Selector I/O channel:** The selector I/O channel is a dedicated channel to perform data transfer exclusively with a high speed I/O device connected to it. Each I/O controller may control a single or multiple I/O devices. Thus, I/O channels reduce the burden of the CPU by controlling the I/O controllers. The selector I/O channel is shown in Figure 7.21(a).
- B. **Multiplexer I/O channel:** Figure 7.21(b) depicts the multiplexer I/O channel. This I/O channel is capable of handling I/O operations with more than one I/O device altogether. For example-if, there are three slow devices interested to send the following bytes given as:

I st I/O device:	A1	A2	A3	A4	A5
II nd I/O device:	B1	B2	B3	B4	B5
III rd I/O device:	C1	C2	C3	C4	C5

Then, the byte multiplexer I/O channel may send the bytes as follows:

C1 B1 A1 C2 B2 A2 C3 B3 A3.....

For high-speed devices, block multiplexer I/O channels can be used which transfers data in form of blocks.

7.9 EXTERNAL COMMUNICATION INTERFACES

The external communication interface is used to provide an interface between the peripheral devices and I/O interface. There exist two main categories of external communication interfaces:

- (a) Serial Communication Interface
- (b) Parallel Communication Interface

The serial communication interface employs only one line for transferring one bit of data at a time. Serial interfaces are used for *terminals and printers*.

The parallel communication interface can transfer several bits at the same time. The parallel interfaces are usually utilized for I/O devices high-speed. For instance, *disks* and *tapes* use parallel interfaces. The communication that takes place across the communication interface includes the exchange of data and control information.

In both serial and parallel communication, the I/O interface must engage in communication with the peripheral. The communication steps for the read/write operation are given as follows:

- The I/O interface sends a control signal to I/O device to request permission to receive or send data.
- The I/O device accepts the Read/Write request and sends an acknowledgement.
- The data transfer operation either from I/O device to I/O interface (Read operation) or from I/O interface to I/O device (Write operation) is performed.
- The I/O device sends an acknowledgement signal after the data transfer operation.

Point-to-Point and Multipoint Configuration

There are two types of connection between the external I/O devices and I/O interface in a computer system: (i) point-to-point (ii) multipoint. In a point-to-point interface, a dedicated line is provided between the external device and I/O interface in the computer system. The

examples of use of point-to-point interfaces point are: keyboard, printer and external modems. Some common examples of point-to-point interfaces are **EAI-232** and **RS-232C**. On the other hand, the multipoint external interfaces are used to support external multimedia devices (such as audio, video, CD-ROM) and mass storage devices (such as tape drives and disk). Some common examples of multipoint external interfaces are **Infini Band** and **FireWire**.

Check Your Progress 3

1. List major characteristics of I/O processor.

.....
.....
.....

2. Differentiate selector and multiplexer I/O channels.

.....
.....
.....

3. Compare serial and parallel external interfaces.

.....
.....
.....

7.10 SUMMARY

This unit explains exclusively the I/O organisation of a computer system. This unit presents the description of I/O devices, the concept of I/O interface, the description and structure of device controllers, and asynchronous data transfer modes. It also discusses three I/O techniques i.e., programmed I/O, interrupt-driven I/O, and direct memory access (DMA). The three I/O techniques involve processor at different levels and help processor to behave differently. The interrupt processing is also discussed in detail. The evolution of I/O processor is also discussed along with the brief explanation of the input/output processor as well as the external communication interfaces. The I/O processors are the most recent I/O interfaces that are capable to execute the I/O instructions without the involvement of the processor.

7.11 SOLUTIONS /ANSWERS

Check Your Progress 1

1. An I/O interface acts as a bridge between the I/O devices and processor. It provides a communication interface to connect the computer with the external environment. The I/O interfaces are used to send/receive data from the external environment using external device or peripheral. The I/O interface offers the following major functions:
 - Synchronization of control and Timing signals
 - Establish communication between peripheral devices and processor
 - Data buffering due to the speed mismatch between memory and processor
 - A mechanism for error detection
2. Device controllers work as a connecting medium to connect the peripheral devices to a computer system rather than connecting them directly to the system bus. On device controller, the I/O device is connected at one end while it is connected with the system bus on another end. Thus, a device controller works as an interface between the system bus and the I/O device. The major benefits for connecting I/O devices to a computer system using device controllers are as follows:

- A single device controller can be used to connect multiple I/O devices with the computer system simultaneously.
 - The device controllers allow I/O devices to upgrade/change without any update/change required in the current configuration of computer system.
 - The device controllers allow connecting the I/O devices with different configurations/manufacturers with the computer system. This feature offers more flexibility to the computer users in order to buy I/O devices of different configurations/manufacturers
3. A device driver is a software module which manages the communication with a specific I/O device. It provides a software interface to hardware devices. Some examples of different device drivers are printer driver, sound card driver, graphics driver, network card driver, USB driver etc.

Check Your Progress 2

1. Programmed I/O technique is used to perform the I/O operation and it does not need an interrupt for I/O operation. The main purpose of the programmed I/O is to perform data transfer between processor and external environment. This technique is very inefficient, especially when multiprogramming environment is used, as Programmed I/O requires continuous involvement of the processor for the I/O to complete. This wasted time could have been used for execution of other programs.
2. An interrupt is an event which may occur to receive some service by a device or due to some error. After an interrupt occurs, it causes multiple tasks in the software as well in the hardware. The abstract steps for the interrupt handling are as follows:
 - Interrupt the current program
 - Saves the context of the interrupted program
 - Transfer control to interrupt servicing program
 - Execute the interrupt servicing program
 - Restore the context of the interrupted program
 - Resume interrupted program from the point of interruption.
3. DMA is needed to enable a device to transfer data without exposing the CPU which results in much less CPU overhead. The advantages of DMA are:
 - DMA allows a peripheral device to read and write to/from memory without passing through the CPU.
 - DMA allows for faster processing as the processor can be working on something else while the peripheral can perform memory related work.

The disadvantages of DMA are:

- requires a DMA controller to carry out the operation, which increases the cost of the system
- problem of cache coherence

Check Your Progress 3

1. Some of the characteristics of I/O channels are:
 - An I/O processor (I/O channel) is the extension of the DMA concept.
 - An I/O channel holds complete control of the I/O operation and is also capable to execute I/O instructions.
 - The CPU initiates the I/O operations and handovers the execution I/O commands to I/O channel. In response, the I/O channel executes the I/O instructions stored into the main memory.
2. The difference between Selector Channel and Multiplexer Channel is that the Selector channel communicates the data with only one dedicated high-speed device at one time while the multiplexer channel is capable to handle I/O with multiple devices at the same time.
3. The difference between serial and parallel interfaces is given below:
 - The serial communication interface employs only one line to transfer one bit of data at a time. The parallel communication interface can transfer several bits at the same time and is usually utilized for I/O devices high-speed.
 - Serial interfaces are used for terminals and printers while high-speed peripherals such as disks and tapes use parallel interfaces.

UNIT 8 I/O TECHNOLOGY

Structure

- 8.0 Introduction
- 8.1 Objectives
- 8.2 Mouse
 - 8.2.1 Classifications of Mice
- 8.3 Keyboard
 - 8.3.1 Features of Keyboard
- 8.4 Monitors
 - 8.4.1 Cathode Ray Tube
 - 8.4.2 Liquid Crystal Display
 - 8.4.3 Light-Emitting diode
- 8.5 Video Cards
 - 8.5.1 Resolution
 - 8.5.2 Color Depth
 - 8.5.3 Video Memory
 - 8.5.4 Refresh Rates
 - 8.5.5 Graphic Accelerators
 - 8.5.6 Video Card Interfaces
- 8.6 Sound Cards
- 8.7 Digital Camera
 - 8.7.1 Webcam
- 8.8 Voice Based Input Devices
 - 8.8.1 Siri
 - 8.8.2 Alexa
- 8.9 Printers
 - 8.9.1 Impact Printers
 - 8.9.2 Non-impact Printers
- 8.10 Scanners
 - 8.10.1 Resolution
- 8.11 Modems
- 8.12 Summary
- 8.13 Solutions /Answers
- References

ignou
THE PEOPLE'S
UNIVERSITY

8.0 INTRODUCTION

In the previous unit, you have studied the concept of input/output interfaces and I/O techniques. The previous unit discussed three I/O techniques i.e., programmed I/O, Interrupt-driven I/O and DMA were discussed along with the evolution of I/O processor. A computer supports a number of I/O devices in order to perform data transfer with external environment. This unit provides a brief introduction to the various I/O devices such as mouse, keyboard, monitor, printer, scanner, video & sound cards etc. It also discusses the modern voice-based input devices. The unit does not attempt to provide all the details of these devices, but attempts to introduce you the characteristics, basic functions and use of the devices in the context of the processor.

8.1 OBJECTIVES

After study of this unit, the students ought to be able to:

- Explain the features of mouse and its classifications;
- List the basic characteristics, functioning and interfacing requirements of keyboard;
- Explain different types of monitors
- Explain video Cards, sound cards, and digital camera
- Explain different types of printers;
- Explain the basic characteristics of Modems and scanners;
- Explain the concept of voice-based input

8.2 MOUSE

Douglas C. Engelbart at Stanford Research Institute (now SRI International) proposed the basic concept of mouse in order to use it with computer system. Xerox Corporation is first organization which developed the first Mouse. It is hand-held hardware input pointing device, which gives user a cursor (pointing mark) on monitor screen and this cursor is used to send the input to computer system. The purpose of mouse is to detect two-dimensional movement relative to surface. Typically, mouse is available with two or three buttons but a single button is sufficient to control the movement of cursor. There exist different types of mice namely Wired, Wireless, Bluetooth, Trackball, Optical, Laser, Magic, USB etc.

The unit of mouse resolution is Counts Per Inch (CPI) which represents *the number of signals per inch of physical travel of mouse*. The value of CPI may range from 400 to 1600. The mouse also sends CPI data to computer with some frequency which is known as polling rate. The polling rate may range from 60 Hz to 1000 Hz. The large value of CPI will result in faster movement of cursor which requires sending much data to computer demanding high polling rate. Therefore, it will be difficult to control the accuracy for large value of CPI.

8.2.1 Classifications of Mice

The classifications of mice are based on connectors, number of buttons and position sensing technologies. Two classifications are discussed-

1. **Connectors:** This category deals with categorization of computer mice based on ports/physical channels which are used to connect the mouse and computers.
 - a) **Bus Mouse**-Bus was used to connect the first mouse with PC. Thus, it has been called as the bus mouse. It was used with IBM-compatible personal computers in its early days. A specialized bus interface was used to connect them with PC which was implemented via an ISA add-in card.
 - b) **Serial Mouse**-In Serial mouse, serial port was used for connection. It is basically an interface present physically on computer for communication. Bit by bit information goes in and taken out of the computer through this port. It is a male port of D-type having 9 pin (DB9M) which is found at the back of the motherboard. However, this category of mice is no longer in use.
 - c) **PS/2 Mouse**-The green colored PS/2 port is used to connect the mouse. Introduced in 1987, PS/2 uses 6-pin mini-din connector. It is the successor of serial connectors. PS/2 ports were first used in the PS/2 systems and they are still being used in modern designs. Green color of PS/2 port is for mouse and purple colored is for keyboard.
 - d) **USB Mouse**-USB mouse are same in terms of shape and appearance but the difference lies in terms of connector. They are connected to a USB port. USB stands

for universal serial bus has superseded the PS/2 ports, though some of the computers still have the PS/2 ports. This standard defines the cables, connectors and communication protocols for connection and communication between computers and attached peripheral devices. The objective of this standard was to standardize computer devices connection.

- e) **Wireless Mouse**-These are the modern mouse that does not require any cable for connection. Eliminating the clutter of cables, it provides a neat type of mouse to use. Some of its key features are- comfortable ergonomic design, improved battery life, Plug-and-Play, multi-function and wide compatibility
2. **Sensing Technology:** There are two types of mice based on sensing technologies i.e., mechanical mouse and optical mouse.
- a) Mechanical mouse has a rubber or metal ball in middle, which is used to control the movement of cursor. The sensors inside the ball detect the rotation of ball. When the ball rolls with the movement of mouse, it causes sensors to detect the rotation of ball along the two axes which consequently send signals to monitor screen. Figure 8.1(a) depicts the mechanical mouse.
 - b) Optical mice use light emitting diodes (LEDs), optical sensors and digital image processing. The optical mouse detects by sensing the changes in the reflected light. The change in reflected light is measured by analyzing the images and the cursor moves on screen accordingly. Figure 8.1(b) shows the optical mouse.

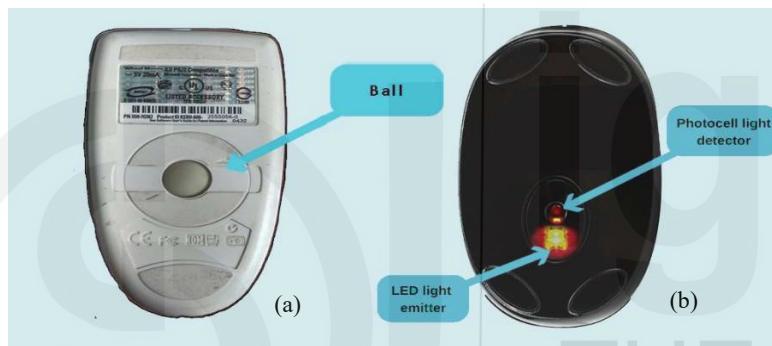


Figure-8.1: Difference between (a) Mechanical Mouse and (b) Optical Mouse

8.3 KEYBOARD

A keyboard is an input device, which is used since the inception of the computer systems. The keyboard allows manual input of alphabets, numbers, special characters, which are available as keys on a board. Figure 8.2 depicts a keyboard. In general, users use a keyboard to transfer a meaningful sequence of characters or numbers to a computer. Thus, a keyboard can be used to send input data into a computer from the external world.

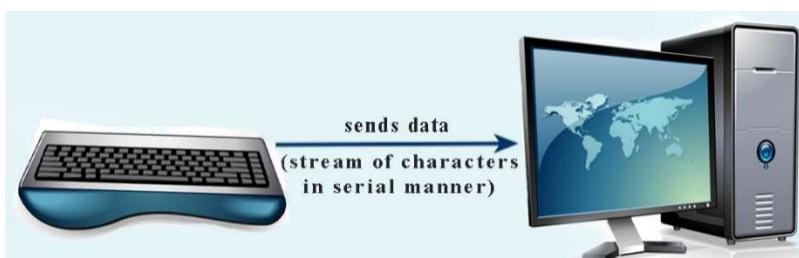


Figure-8.2: Overview of how Keyboard transfers data to the computer

8.3.1 Features of Keyboard

Some of the basic characteristics of keyboard are given as follows.

- **Keyboard Layout**

The layout defines the arrangement of the keys on the keyboard. This arrangement is mostly influenced by the typewriter. The keyboard layout is now available for many different languages of the world. A good layout is the one, which allows faster data input. Thus, in most cases, computer keyboard layout is identical to typewriter key layout, which was designed for enhancing the speed of data input. The standard keyboard layout for English is called QWERTY layout. QWERTY layout stands for the six top alphabets in the keyboard. The

QWERTY arrangement was created by Sholes, who invented typewriter. However, a computer in addition to alphabets is required to input numbers, special characters, and several shortcut commands. Therefore, keyboard has a very detailed arrangement of keys.

One popular keyboard designed by IBM for the personal computer had 101 keys. This keyboard's key layout is shown in Figure 8.3. This keyboard consists of key sets for alphabets in the middle, numeric key pad in the right for easy entry of numbers, function keys at the top name F1, F2, which are used as shortcut to various functions in different software, and a set of cursor keys to move the cursor on the screen. Later, a set of windows function keys were added to this design.

Another keyboard layout was designed by A Dvorak and William Dealey in 1936. Their layout was primarily designed for people who find typing with all fingers difficult and would like to type using only two fingers. It was expected that typing would be faster by using Dvorak-Dealey keyboard, as you can use both the hands while typing. You can find more details on this keyboard in the further readings. With the availability of newer mobile smart devices, there are many possibilities of designing new keyboards for specific areas, including better designed layout for regional scripts.

For the languages of our nation, Indic keyboard layout is a standard layout. This layout supports 12 Indian scripts.



Figure-8.3: IBM 101-key Keyboard layout

- **Keyboard Touch**

In addition to layout, the other important characteristic of a keyboard is the keyboard touch. The keys should be sensitive enough to capture the data being entered by the user. A good keyboard must be able to send data with speed. These days, in addition, to physical keyboards, touch screen keyboards are also available. Most of these keyboards provide features of predictive text and autocorrect, which facilitate data entry by the user.

- **Scan Codes**

When a key is pressed on a keyboard, it transfers the scan code relating to those keys to the processor. Scan code of every key is unique. The scan codes are used to communicate the desired data or action to the processor. A keyboard of processor is connected through interrupt driven I/O mechanism. Therefore, when a key or several keys are pressed together on the keyboard, it interrupts the processor, provided processor has enabled interrupts. The processor receives the scan code/codes and identifies the key or keys that were pressed using the scan code table stored in the ROM BIOS. In addition, the status byte that is associated with the keyboard informs the processor about the status of keys that are used as toggles, like, Caps lock, Num Lock, etc. But how does a keyboard identify that more than one keys are pressed together, such as CTRL & ALT & DEL. Interestingly, a keyboard sends two scan codes to the processor - one when key is pressed and second key is released, which were called Make and Break scan codes respectively. Thus, by knowing the timing of these make and break scan codes, processor determines, which keys are pressed together. A detailed discussion on scan codes is beyond the scope of this Unit.

8.4 MONITORS

A monitor is an output display device connected to processor and it displays the vision into the brain of the processor. It allows a user to graphically interact with the processor which is helpful to send output as well as to receive input to/from the user. Technically speaking, it is display device which provides a graphical vision by converting the digital/analog signals into the visual form.

The monitor looks like a television set but both the devices are different with each other. The monitors have greater sharpness, lower input lag, higher refresh rates, color purity, and operate at higher frequencies in comparison to TV sets. The TV set consists of tuner or demodulator circuit to convert the signals.

Whenever users are interested to buy a monitor, they search for the better configurations in minimum possible budget. The configuration of monitors consists of display size, resolution, supported frequencies, the size of the picture tube and the type of connector used to connect to the computer. Monitors are manufactured by many manufacturers like LG, Samsung, Acer, Dell, HP, Lenovo, Sony, Asus, BenQ, etc. The monitors are available with different sizes i.e., 14'', 15'', 17'', 19'', 21.9'', 24'' or even higher. The monitors are also available with different screen form factor i.e., flat and curved screens. The monitors can be categorized into three categories based on the design technology. These categories are discussed in next sub-sections.

8.4.1 Cathode Ray Tubes

The cathode ray tubes monitors and television sets are based on the technology of Cathode ray tube (CRT). A CRT is a partly empty glass tube which consists of inert gas at low pressure. A negatively charged electrode which is known as Cathode/Electron gun is used to shoot beams of electrons at high speed towards a positively charged electrode (anode). The high-speed electrons impinge on the small phosphor coated screen. The screen consists of dots with three primary colors i.e., Red, Green and Blue. Indeed, there exists either one electron gun for the three colors (Red, Green and Blue) or one different electron gun for each color. Figure-8.4 depicts the cathode ray tube (CRT). The quality of image on CRT screen is influenced by following four factors:

1. **Phosphor coating:** The monitor screen is coated with phosphor (fluorescent material) which emits light when bombarded by electron gun. The phosphor coating is provided in inner surface of cathode ray tube. The coating affects the color and the persistence. The term persistence in the context of monitor is defined as the time for which the effect of a single hit on a dot on the monitor surface lasts.
2. **Shadow Mask/Aperture Grill:** It is the manufacturing technology for CRT monitors to produce clear and focused color images. It determines the resolution of the screen in color monitors. In shadow mask CRT, each pixel position consists of 3 phosphor color dots one for each red, green and blue. The Triad and inline arrangements are used for the alignment of color dots to produce good quality images. Another technology for same purpose is the aperture grille
3. **Electron Gun:** The electron gun must be efficient in its working. The high-quality electron gun affects the quality/sharpness of the image.
4. The screen glare and lighting of the monitor are also major factors to influence the quality of the image.

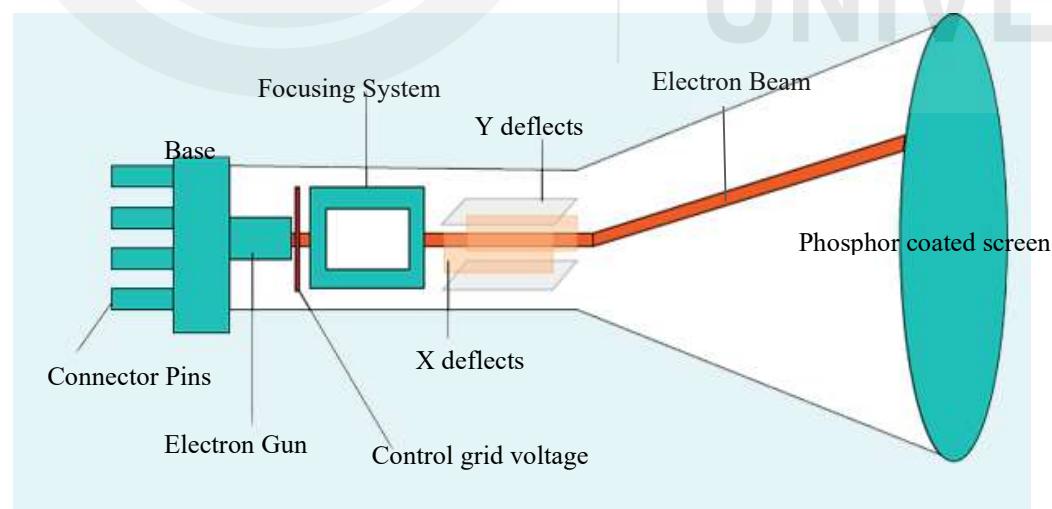


Figure-8.4: Cathode Ray Tube

8.4.2 LIQUID CRYSTAL DISPLAYS (LCDs)

LCDs were developed by the company RCA in the year 1960. An LCD is an electronically modulated optical device which employs light-modulating properties of liquid crystals combined with polarizers. The light is not emitted directly from the liquid crystals rather a

reflector is used to produce images in color or monochrome. An LCD blocks the light to display patterns. LCDs are lightweight screens and are mainly used for portable computers. They are known for low power consumption, good resolution and bright colors. The LCDs can be divided into following three categories based on display generation techniques.

1. **Reflective LCDs:** The display is generated by selectively blocking reflected light.
2. **Backlit LCDs:** The display is generated due to a light source behind LCD panel.
3. **Edgelite LCDs:** The display is generated due to a light source that is adjacent to LCD panel.

LCD Technology

To manufacture the LCD screens, Nematic technology is used. The molecules of liquid crystals (rod-shaped crystals) which are known as Nematic cells are used. Figure 8.5 depicts Nematic cells. The Nematic cells are packed (sandwich) between two thin plastic membranes. The Nematic cells have special properties i.e., these cells can change the polarity and bend of the light. The electric current is used to control these properties by applying the electric on grooves in the plastic membranes.

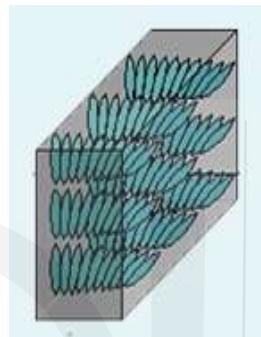


Figure-8.5: Nematic Cells

There exist two types of LCDs i.e., Passive matrix and Active matrix.

1. **Passive Matrix-** The passive matrix arrangement is most widely used technology due to low weight, high image quality, low cost and high response time. LCD panel consists of a grid of horizontal and vertical conductors. The conductors consist of Indium Tin Oxide to create a picture. Each pixel is located at the intersection of two conductors in the grid. Whenever current is passed through a pixel, it becomes dark.
2. **Active Matrix-** It employs Thin Film Transistors (TFT) and that's why known as TFT technology. In active-matrix arrangement, TFTs are arranged in a matrix on a glass surface and these TFTs are considered as pixels. These TFTs receives a small amount of white light, which is then enhanced by TFT to activate a pixel. The advantage of using TFTs is that they have faster response times, as they use smaller amount of light. However, the disadvantage of using TFTs are that they are difficult to fabricate, therefore, are costly. TFT LCD Display Technology is shown in the below Figure-8.6.

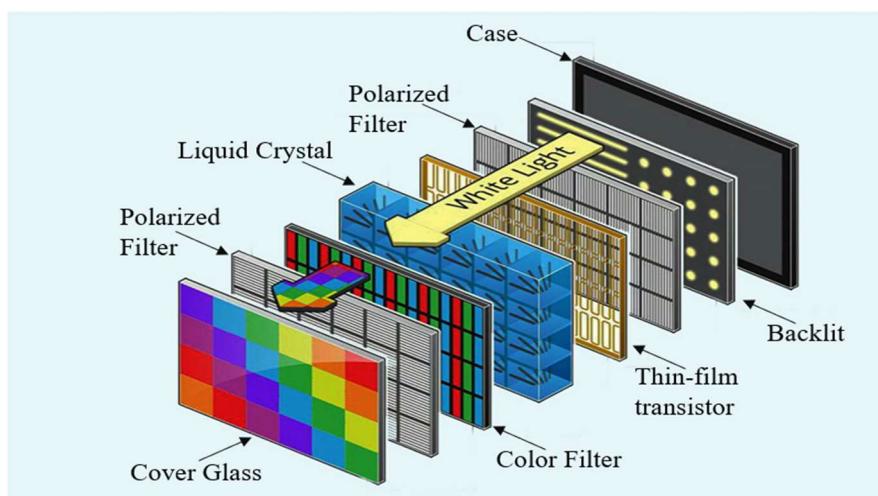


Figure-8.6: TFT LCD Display Technology

8.4.3 Light Emitting Diodes (LED)

The light-emitting diode (LED) monitors display is a flat screen, flat-panel computer monitors or television. It uses array of LEDs as pixels for displaying the videos. It is light weight and has a very short depth. LCD monitors and LED monitors differ only in terms of backlighting; typical LCD monitors uses fluorescent backlights whereas an LED monitor uses light-emitting diodes. The earlier LCD monitors used CCFL instead of LEDs to illuminate the screen. LED monitors offer many features/benefits namely slim design, flicker-free & brighter images, longer lifespan, broader dimming range, low power consumption, better color and picture quality etc. Figure 8.7 lists the benefits of LED monitors.



Figure-8.7 Benefits offered by LED monitors

Check Your Progress 1

1. Explain mechanical and optical mice.
-
-
-

2. Discuss scan codes.
-
-
-

3. What are the differences between LCDs and LEDs?
-
-
-

8.5 VIDEO CARDS

First, this section discusses a brief overview of graphic display technology with the primary focus on CRT monitors, before jumping to video hardware. The graphic display system is responsible for displaying bit-mapped graphics on monitor. Every image is formed using small dots which are known as ***picture elements*** or ***pixels***. Figure 8.8 shows the pixels of an image. The description of each pixel is stored in the memory which is taken care by video system.

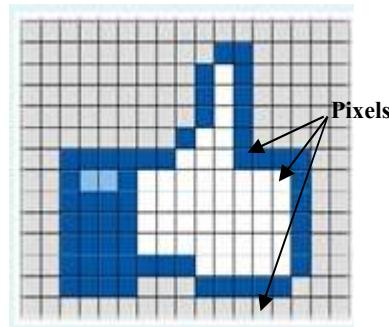


Figure-8.8 Bit-mapped Graphics Image

The display memory which is used to store the data for images is known as **frame buffer**. At any moment, the frame buffer consists of data for bit-map representation of current image on screen and the next image. The frames are read dozens of times per second and sent to the monitor using a cable in serial manner. Upon receiving the stream of data, the monitor forms and displays it on the screen by scanning raster movement from first up to down one row at a time. Based on this raster movement CRT, the monitor will illuminate its small phosphor dots. It is shown in Figure 8.9 and Figure 8.10.

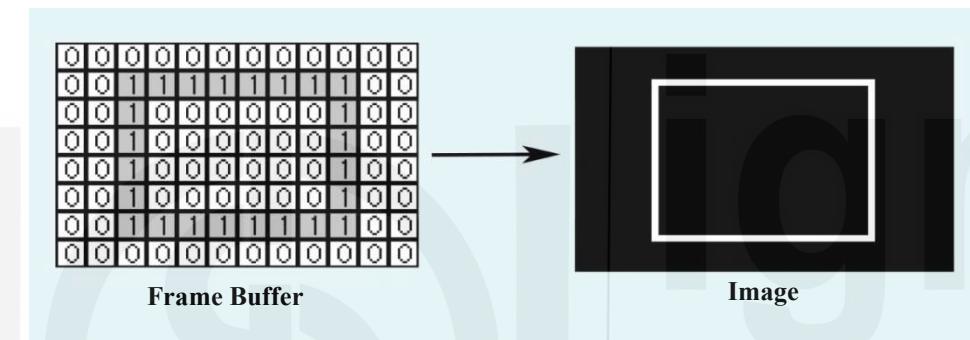


Figure-8.9: Frame Buffer and the corresponding image displayed on the system

The greater number of dots leads to better resolution of the image as well as the sharper the picture. The number of dots directly correspond to the richness of the image (or gray levels for a monochrome display) displayed by the system. The higher the number of colors, the more is the information required for each dot. Therefore, higher resolution and color depth of the system required bigger memory storage by the system to store the frame buffers.

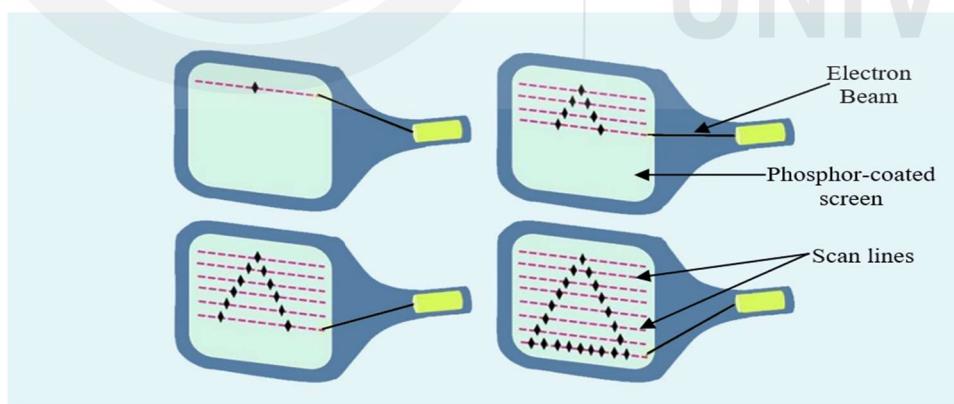


Figure 8.10: Raster Display

8.5.1 Resolution

The resolution is defined as the possible *sharpness or clarity of an image*. The resolution does not depend upon the physical characteristics of the monitor. It is measured in terms of number of pixels on a monitor. For instance, a standard VGA graphic display with resolution 640×480 consists of 640 and 480 pixels on horizontal and vertical axes respectively. In order to construct an image, different numbers of pixels are spread across both the axes of monitor screen. Higher is the resolution, sharper is the image due to large number of pixels.

The sharpness of an image on actual live-screen does not depend only on resolution but it is measured in the unit of dots-per-inch. These dots-per-inch are dependent on (i) size of the

image and (ii) resolution of the image. An image will be sharper on a smaller screen in comparison to bigger screen. For instance, an image may appear sharp on a 15" monitor and may be a little jagged on a 12" monitor display. Figure 8.11 shows a circle with different sharpness on different size monitor screens.

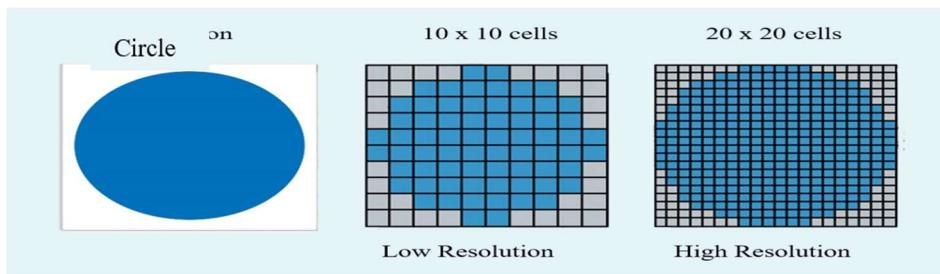


Figure-8.11: Circle with low and high resolutions

8.5.2 Color Depth

The image is constructed using stream of pixels. If the value of pixel is ‘ON’ and ‘OFF’, the pixel will be displayed in image on the screen as a pure black and white respectively. If single bit is assigned to a pixel, the image will be black and white. This system is known as **two-color system**. The pure black and white images can be converted to gray levels, which are different levels between white and black. This requires a greater number of bits to code each pixel. For instance-if you assign two bits to each pixel, four color levels are possible: White, Light Gray, Dark Gray and Black. In general, you need more than one bit to describe a pixel. Hence, one bit per pixel implies 2 colors or 2 gray-levels, 2 bits per pixel implies 4 colors or 4 gray-levels, and 3 bits per pixel implies 8 colors and so on. It means n bits per pixel imply 2^n gray-levels. For colored images color codes for the intensity of the three primary colors, viz. Red, Green and Blue, for each pixel are stored.

Color Depth can be understood as *the number of bits allocated to every pixel in order to store color code information*. Since every bit of a pixel corresponds to a specific color i.e., all bits at the same position for all pixels corresponds to the same color. Thus, the bits corresponding to same color can be regarded to form a plane and these planes are known as color planes. It is considered the color planes are stacked on top of each other which are helpful in deciding final color at each pixel. Thus, depending upon number of bits required for each pixel, there are 3 *Color Planes* (one each for Red, Green and Blue). Figure-8.12 depicts the 3-bits color display and 3 color planes.

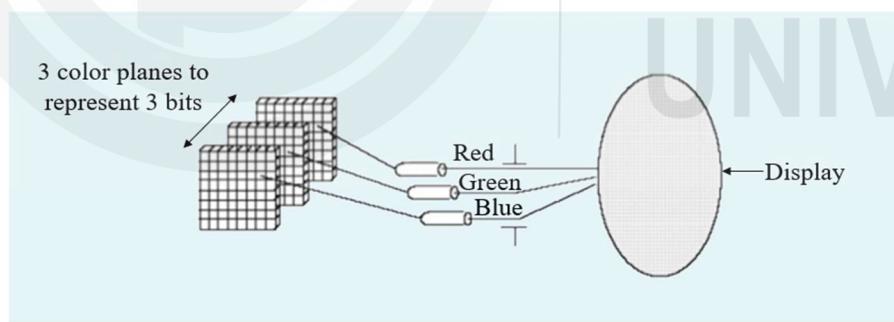


Figure-8.12: 3-bits Color Display with 3 color planes

The computer system with a 3-bit RGB color planes utilized 1 bit for each of the red, green and blue color components. Therefore, every color component can exist only in “ON” or “OFF” state. The three-bit RGB ‘ON’ or ‘OFF’ color components result in 8-colors consisting of three primary RGB colors i.e., red, green and blue; two pure colors i.e., white and black; and three complementary colors i.e., magenta, cyan, and yellow colors. The RGB values (ON” or “OFF”) of 3-bits color are given in Table 1 and the colors are displayed in Figure 8.13.



Figure-8.13: 8 possible colors for 3-bits Color Display

Table-1: 3-bit Color Display RGB values

Bit-values ('ON' or 'OFF')			Color
0	0	0	Black
0	0	1	Blue
0	1	0	Green
0	1	1	Cyan
1	0	0	Red
1	0	1	Magenta
1	1	0	Yellow
1	1	1	White

What Color depths are practically used?

If n is the color depth representing the total number of bits used to store one pixel, the number of colors will be 2^n (2 to the power n). Table 2 puts the most popular color modes.

Table 2: Popular Color Depths

Sr. No.	Color Depth(bits/pixel)	Color Mode
1	1	Monochrome
2	4	16-Colours
3	8	256-Colours
4	16	High Color
5	24	True Color

Human & Color Depth of Monitor Screen

The modern monitors can display up to a maximum of 262,144 (2^{18}) colors for 18 bits/pixel *Color Depth*. If different tones (color pitch) are allowed and eight bits are used for each RGB color, a total of 16.77 million colors (256 tones (R) x 256 tones (G) x 256 tones (B) = 16,777,216 colors) can be generated. On the other hand, human eyes are capable to distinguish maximum few million different colors. Thus, even if the monitor screens display colors more than few million, they would not be distinguished by humans. It also implies that the practical upper limit the 24-bit per pixel color depth. Since the number of possible colors produced by this color depth is more than the colors that could be distinguished by human eye, these colors are called the true colors. However, 24-bits color or true color systems have more color than possibly useful; the extra 8-bits are used by designers to store special effects information of the image. These extra bits formed a channel which is known as Alpha channel.

8.5.3 Video Memory

The video memory/frame buffer is used to store the video to be displayed. The quality of video display depends on the efficiency of the video system i.e., how quickly the frames are accessed and updated. Initially, a fixed area of RAM is allocated to the video memory. Later, video RAM along with video cards was introduced and it can be increased by placing additional video RAM under the unified memory architecture (UMA). UMA is helpful in reducing the cost of computer system. In UMA supported systems, an area of main memory is used as frame buffer/video memory which results in elimination of bus for video processing. Therefore, the computer system with UMA may be less costly.

Basically, UMA comes with on-board video card in the modern low-cost motherboards. The required resolution and color-depths are the deciding factor for the size of video memory/frame buffer. The minimum size of video memory can be calculated simply by multiplying the *Color Depths* and resolution of the monitor screen.

Let us solve a simple exercise. Assume a standard VGA monitor screen with resolution 800×600 and color depth value 8.

Number of Pixels	= $800 \times 600 = 480000$
Color Depth (8 or 2^3)	= 3-bits
Minimum Memory required	= 1,440,000 bits (180,000 bytes)
	= 180 KB

It implies that minimum RAM required for resolution 800×600 and 8 color depth is 180 KB. But the memory is available in exponential power of 2 and the next minimum size of memory is 256 KB. It implies that minimum size RAM required for resolution 800×600 and 8 color depth is 256 KB.

Now-a-days, a very odd-looking resolution i.e., 1152×864 has become popular. Could you guess why this is so? The following are the reasons behind its popularity. There are nearly one million (9,95,328) pixels for VGA with 1152×864 resolution. For color depth value 8, nearly 8 million bits or 1 MB memory is required. Further, human eyes perceive only a few million colors and this resolution is more suitable. In addition, a square pixel that has a ratio of 4: 3 allows easier programming.

Please note that the calculations shown above are not applicable for 3-D displays, which requires more memory due to the issues like “Double Buffering” and/or “Z-Buffering”.

8.5.4 Refresh Rates

The Video Controller (a special circuit) scans the frame buffer and reads rows one by one followed by sending this data in serial manner. On monitor screen, the electron beam starts scanning one-line at a time from left to right direction in order to create images. The *horizontal refresh rate or horizontal frequency* is the rate at which horizontal sweeps take place, while *vertical refresh rate or vertical frequency* is the rate at which vertical sweeps take place. The vertical frequency is also known as *refresh rate or frame rate*, as during a vertical sweep one complete frame is displayed. There exist several hundred rows in each frame and thus, horizontal frequency is hundreds of times higher than vertical frequency. The unit of horizontal frequency is KHz while the unit of vertical frequency is Hz.

Note: It is necessary to maintain the same frequencies between the monitor and video system for better quality of images. The compatible refresh rates are provided with the manual of the monitor.

8.5.5 Graphic Accelerators

An important chip associated with video card is known as Graphic Accelerator which is the replacement older technology known as *Graphic Co-Processor*. The graphic accelerator chip is a dedicated unit that executes in-built video functions of image construction and rendering, thus, releasing the microprocessor (main processor) from this work. The accelerator chips are optional but they are required due to noticeable impact on the performance of the computer, especially in graphics-intensive tasks such as- Rendering of 3D models and images, Video editing and Gaming. The graphic accelerator are needed if you need the following:

- Good support to 3-D graphics.
- Better resolution of graphics.
- Larger size of memory in the frame buffer.
- Better speed of display of drop-down menu.
- Good quality video playback.

The Graphics accelerators are widely used in industries such as- Motion pictures for special effects, Computer-aided design (CAD), Video games, 3D-effect etc.

What is a 3-D Accelerator?

The accelerator chip that has built-in ability to perform the mathematical calculations and execute the algorithms required for 3-D image generation and rendering, are called 3-D Accelerator. A 3-D image is just an illusion for human eyes which basically represents a projection of 3-D images/videos on 2-D monitor screens. This conversion takes place by projection and transparency effects, perspective effects, color depth and lighting effects. In addition, the following techniques can be used for creating 3-D images on 2-D screens: (i) Ray-Tracing, which traces the path of light rays emitted by a light source; (ii) Z-buffering, which uses a buffer to store the third axis, i.e., Z-axis positions and (iii) Double-Buffering, which uses two buffers in place of a single buffer.

8.5.6 Video Card Interfaces

A video card interface connects the video display to the computer system in order to improve the performance of the visual data you see on your screen. The video card can either be a separate component which is plugged into a slot on the motherboard of the computer or it may be integrated into the motherboard known as “onboard”. For isolated video cards, the connection is realized using either *Peripheral Connect Interface (PCI)* or *Accelerated Graphics Port (AGP)* bus.

- **PCI-** It is introduced by Intel and also known as *Peripheral Component Interconnect*. It is a high-speed common bus which is used to attach the computer peripherals to the motherboard. It is used to attach sound cards, network cards and video cards. The computers may use now some modern technologies like PCI-Express (PCIe), USB and AGP.
- **AGP-** It is also known as *Advanced Graphics Port*. It is a standard connector port used to connect the video card with the microprocessor and the main memory. It is a dedicated high-speed connection interface which is used by only graphics subsystem. AGP employs pipelining, isolated data and address buses and high-speed mode to improve the performance of graphics card.
In specific computers, the video card is directly connected with the microprocessor and may use direct memory access (DMA) I/O technique to send data from main memory to frame buffer.

8.6 SOUND CARDS

Multimedia has become an indispensable component of personnel computers to play different music files like MP3, MP4, WAV (Waveform audio file), WMA (Windows media audio), AAC (Advanced audio coding), FALC (Free lossless audio codec), OGG (The latest Free Sound format standard) etc.

The Sound card can either integrated into motherboard (built-in sound card) or connected through expansion slot. As you may study in computer networks, the analog sound waves could be converted into electrical form using electrical signals, which is used to compute the strength of sound. Usually, the analog audio signal is converted into digital audio (or digital signals) in the form of bits using sampling process. The microprocessor manipulates the digital audio bits and this data is sent to the sound card. The sound card converts this data into analog audio in order to play back through the speakers or headphones. The major functions of a modern sound card are as follows:

1. Conversion from digital sound signals to analog form to play back the sound.
2. Amplifiers to augment the strength of sound signals
3. Sound recording.
4. Sound synthesis.
5. Mixing of sound from various resources.

The three basic issues relating to sound cards are - Compatibility, Connections and Quality.

- **Compatibility:** Sound cards must be compatible for hardware as well as for software to meet the current industry standards/protocols. Some specific software like games need sound cards to be compatible with industry standards. You may refer to further readings to know about these standards.
- **Connections:** The sound card must provide different connections in order to perform various functions. It should provide MIDI port (Musical Instrument Device Interface) which allows user to produce music directly by using synthesizer circuit in the sound card. It also allows connecting a Piano keyboard to the computer system.
- **Quality:** There exist different sound cards which provide sounds with different qualities. The quality of sound differs due to the noise control, digital quality and the ranges of frequency supported by the sound card.

Check Your Progress 2

1. Explain the concept of a frame buffer in the context of Video Card interfaces.

.....
.....
.....

2. What do you understand by horizontal and vertical frequencies?

.....
.....
.....

3. Compute the minimum required video memory for 16 color depth and a monitor screen with the highest possible resolution 7680x4320.

4. Explain sound card. What are the functions of sound card?

8.7 DIGITAL CAMERA

The first digital camera was invented in the year 1975 by Steven Sasson at Eastman Kodak. Digital camera is a hardware device that takes images or record videos and stores them on memory as digital data on memory card instead of on photographic film in analog camera. In digital Camera, the images are stored in digital form and thus they can be reused later for different purposes like printing, editing etc. Since the digital camera takes images (input) and sends them to computer (output), it is considered as input as well as output device.

Figure 8.14 depicts the digital camera which is taking an image of subject (scene) under consideration. A digital camera consists of a sequence of lenses which focuses light on to a semiconductor device to create an image of a scene under consideration. The semiconductor device, in turn, records this light as digital images by using an in-built processor.

The semiconductor device is known as an Image sensor which converts light into electrical charges. Two types of Image sensors exist: Charge Coupled Devices (CCD) and Complementary Metal Oxide Semiconductor (CMOS). CCD is more popular and powerful kind of sensor in comparison to CMOS image sensors.

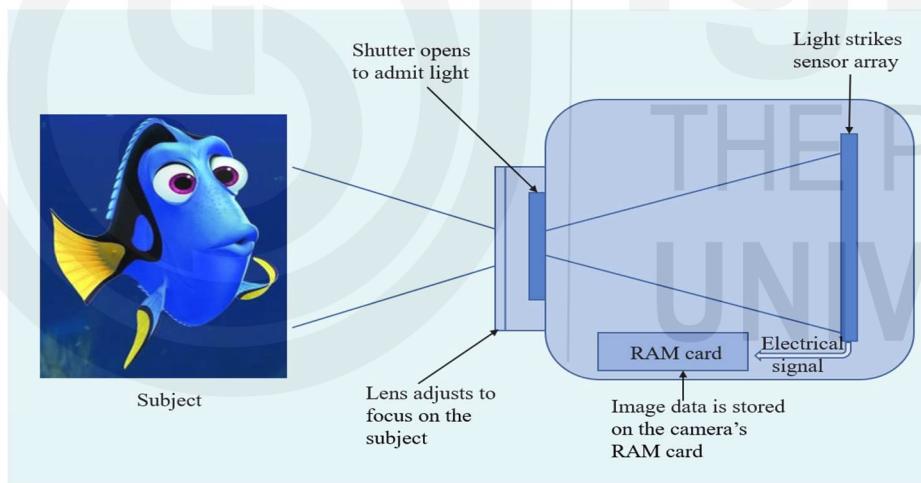


Figure-8.14: Working of a Digital Camera

The resolution (pixels) of digital camera is major deciding factor for the quality images. The higher the resolution, the better the digital camera is. The major benefits of a digital camera are as follows

- Allows user to see the videos and images immediately
- Allows user to store thousands of images/videos due to in-built memory
- Digital cameras are portable
- Allows user to edit images directly
- Allows flexibility in printing of desired images

8.7.1 Webcam

A digital camera without storage connected to computer system or network is referred as Webcam. In modern computers, the webcam can either be a separate component which is plugged into the computer or be an in-built integrated camera. In order to use webcams, it is necessary to install the required software. A webcam is an input device which is used to capture the images/videos and then send it to the computer. Webcams are used for videoconference or video calling or online meeting using Google Meet, Zoom, MS Team and others services.

8.8 Voice Based Input Device

Modern devices are capable to take human voice as input using speech recognition processes and execute applications accordingly. These devices are known as Voice Based Input devices. As compared to microphone, the speech recognition process of these devices recognizes human voice; converts it into machine-language and execute programs/applications accordingly. Figure 8.15 two devices which use speech recognition process to recognize human voice.



Figure-8.15: (a) Siri (b) Echo Dot 3 Smart Speaker with Alexa

The Voice Based Input Devices can recognize spoken words in two ways. The spoken words can either be recognized from a pre-defined vocabulary or be recognized from a known speaker after training of the input device. Whenever speaker utters a word from the pre-defined vocabulary, the Voice Based Input device may display the characters of monitor screen for verification by the speaker. However, some of these devices may process the speech without verification from the speaker. The process of speech recognition compares each uttered word with the words stored in pre-defined vocabulary table.

8.8.1 Siri

Apple Inc. offers a built-in, voice-controlled virtual assistant with most of products i.e., iPhone, iPad, Apple Watch or Mac (macOS Sierra and later) etc. This voice-controlled personal assistant is known as Siri. The users may talk to Siri as they talk to their friends. Siri allows a seamless interaction with Apple devices such that user speaks to Siri and Siri speaks to user. Siri helps users to get their job done after receiving user commands. Siri can help to open a file, send messages, open a web browser, open a website, booking a ticket, watch movies, and many other activities.

Siri works based on the Artificial Intelligence and Natural Language Processing fields. It consists of three components -Conversational interface, personal context awareness and service delegation systems. The conversational interface understands the user word-for-word manner and the semantic of text is produced using personal context awareness which is based on habit and language of the user. The service delegation helps to deliver services using built-in apps and their inner workings.

8.8.2 Alexa

Amazon offers virtual interactive voice-based AI powered digital assistant known as Alexa. This device has been designed in association with Alexa Voice Service (AVS) in order to simulate real conversations. “Alexa” is basically the “wake word” which is used to alert the device to start listening the voice to perform some tasks. Alexa employs intuitive voice commands to provide services to perform some specific tasks. Figure 8.15(b) depicts the Amazon Alexa. It is available as Echo speakers, smart thermostats, lamps and lights, and right on your phone through the Alexa app. Alexa can do quick math, play music, check news and weather updates, read emails and control many of the smart products.

Alexa also works based on the Artificial Intelligence and Natural Language Processing fields. It Alexa consists of speakers, microphone and a processor which is used to activate the device. It receives input and sends it to cloud where Alexa Voice services (AVS) interprets and understands the user input. Accordingly, AVS sends the appropriate output back to user device. The internet connection is the basic requirement to use Alexa.

8.9 PRINTERS

Printers are devices that accept textual and graphical contents as output from a computer system and print contents on paper in a controlled manner. The text and photographic images are produced by printers. Printers differ in technology used, memory, speed, resolution, color supported, size, hardware compatibility, cost and others factors. The present-day printer technologies include the dot matrix printer, Inkjet or tank printers, Laser Printers etc. to serve different needs. The available printers can be divided into two classifications-Impact and Non-impact printers. Figure 8.16 shows the classifications of printers.

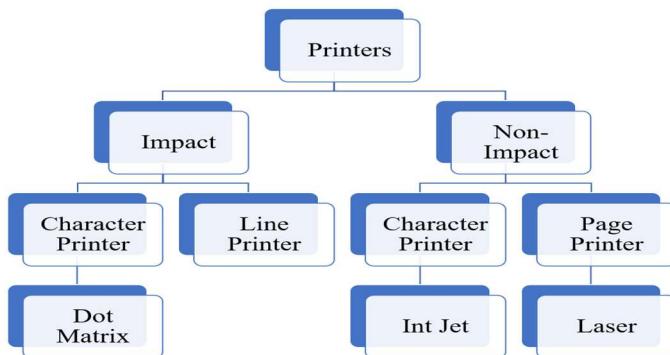


Figure-8.16: Classification of Printers

8.9.1 Impact Printers

Impact printer uses mechanical components for printing i.e., physical contact between printing head and paper. In order to print, the characters and graphics are produced on a paper by striking. They produce banging noise during printing. Impact printers can be divided further into two categories-Character and line printers. The character printers print only one at a time by striking on ribbon whereas line printers print one line at time. The line printers are fast and costly printer in comparison to character printers. Different types of impact printers are line printers, dot-matrix printers and, daisy-wheel printers.



(a)



(b)

Figure-8.17: (a) Dot-matrix printer (b) Local Railway ticket

Dot Matrix Printer

A Dot Matrix Printer (DMP) employs fixed number of pins to print on paper. The print head with many pins runs back and forth on the page and prints by striking against a sooted cloth ribbon in order to make a mark on paper. In DMP, the characters are using matrix of dots and the printed character is basically an accumulation of several dots on the paper. Therefore, the arbitrary font or graphics will be generated in each printing. Figure 8.17 shows a DMP and the local railway ticket printed by DMP.

8.9.2 Non-Impact Printers

In non-impact printer, no mechanical moving component is employed for printing. These printers don't strike or impact the ribbon for print. The technologies used by non-impact printers are chemical, inkjet, electrostatic, xerographic and laser. These printers work silently. Non-impact printers can be further divided into two categories-Character and Page printers. The non-impact character printers spray tiny drops of ink on to the without striking/physical contact with paper. The page printers print one full page at once. Different types of non-impact printers are inkjet, photo, and laser printers.

Laser Printer

Laser printers are very common page printers and print one page at once. Laser printers employ a focused light beam to transfer image or text onto paper. The modern laser printer use Resolution Enhancement Technology (RET) which is introduced by Hewlett-Packard. This technology smoothens the edges of character, diagonal lines etc. to produce better quality printouts. To produce high quality print, the basic requirement is the memory which increases as a square of resolution i.e., dots per inch (DPI). For 600 dpi, approximately 3.5 MB (600x600 bits) memory is required whereas 14 MB (1200x1200 bits) is required for 1200 dpi. Figure 8.18 depicts a single function monochrome laser printer.



Figure-8.18: Single function Monochrome Laser Printer

8.10 SCANNERS

A scanner is an electronic device which is used to capture images from tangible sources like photographic images, paper, posters, slides and others. The scanner converts the captured images into electronic form and stores them in computer memory in order to view/modify later. The scanner employs light sensors arranged in the form of an array in scan-able area. The light sensors detect differences in brightness of reflections from an image and then scan the source.

The existing scanners differ in many factors such as compatibility, resolution, support for different media and interfaces, etc. Two popular types of scanners are - *Flatbed Scanners* and *Handheld Scanners*.

Flatbed Scanners are used to scan high-resolution tangible images into detailed and sharp electronic images. The images are placed on a flat glass tray and movable sensors are used to scan the images. Figure 8.19(a) shows a flatbed scanner. Handheld scanners are used to scan the physical documents, and require good hand control for high quality scanning. These are the most portable and cheapest scanners and shown in Figure-8.19(b).

Scanning is used for many different applications. The scanners are used as Magnetic Ink Character Recognition (MICR) scanner in order to scan cheques and Bar-Code readers to identify different objects. One more application is *Optical Recognition of Characters (OCR)*. The *OCR* software use character/pattern matching algorithms to recognize characters and converts the scanned text to a text file. The *OCR* technology is very much useful in digitizing the ancient text written in old scripts.



Figure-8.19: (a) Flatbed Scanner (b) Handheld Scanner

8.10.1 Resolution

The resolution of scanner is the quality of image achieved by scanner. It is measured in *dots per inch (dpi)* and it indicates the number of dots per inch scanned horizontally and vertically.

It implies that the more is the dpi of a scanner, the more details a scanned electronic image will have. The scanned file size increases with increased resolution. There are various ways to measure the resolution.

Optical Resolution - The upper resolution limit of a scanner which is used to scan the images is known as optical resolution (hardware resolution). For example- if the optical resolution of a scanner is 300 dpi, it means 90000 (300x300) pixels per square inch can be captured by the scanner. The scanners may be available with optical resolutions of 300, 600, 1200, 2400 dpi or even more.

Interpolated Resolution- The resolution of image can be augmented using interpolation algorithms and this resolution is known as Interpolation resolution. The interpolation technique employing complex algorithms is used to add intermediate pixels based on the properties of surrounding pixels. The interpolation technique results in increased size of scanned images but it provides smoother and high-quality images without adding any additional information. For instance- if the optical and interpolated resolutions are 300x300 dpi and 4800x4800 dpi respectively. This implies 90000 pixels per square inch can be captured by the scanner while the interpolation algorithm can add 15 pixels between every pair of pixels to increase the dpi of image.

8.11 MODEMS

Modem (i.e., modulator-demodulator) is a device that connects two computers using telephone lines in order to exchange data with each other. The modem receives digital signals from computer, puts them into analog circuit by modifying a constant wave (known as carriers) and then analog signals are transmitted over the telephone lines. This process is known as modulation. It occurs whenever user connects to the Internet. Demodulation is the inverse process of modulation in which the digital signals are derived from the modulated wave. It occurs whenever user receives data from a website, which is then displayed by your browser. Figure 8.20 shows the process of modulation and demodulation performed by the modem. You may refer to further readings for more details on modulation and demodulation techniques.

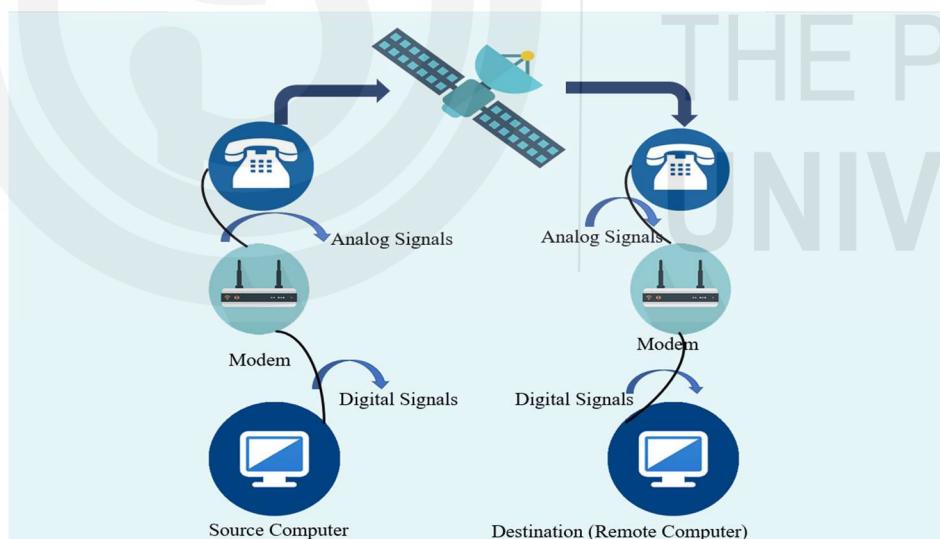


Figure-8.20: Modulation and Demodulation process by Modem

Check Your Progress 3

1. Explain webcam and its major benefits.

.....

.....

.....

2. How characters are recognized by voice-based input devices?

.....
.....
.....

3. Compare impact and non-impact printers.

.....
.....
.....

4. Explain Interpolated resolution.

.....
.....
.....

5. How many pixels can be captured by a scanner with 600 dpi optical resolution?

.....
.....
.....

8.12 SUMMARY

This unit discussed several input/output devices and the technologies behind them. This unit covers different input device along with different components or types or features. It is discussed Mouse and classifications of Mice, Keyboard along with its features, voice-based input devices, scanners and webcam its different types. This unit also discussed output devices along with other components or types or features. It discussed computer monitors with three different categories i.e., CRT, LCD and TFT screens. The printers with their different categories have also been discussed. The video cards have also been discussed with their characteristics like resolution, color depth, video memory, refresh rates, graphic accelerators and video card interfaces. It also discussed sound cards with its functions and different characteristics. At last, the modem device is discussed in brief manner.

8.13 SOLUTIONS/ ANSWERS

Check Your Progress 1

1. Mechanical mouse has a rubber or metal ball in middle, which is used to control the movement of cursor. The sensors inside the ball detect the rotation of ball. When the ball rolls with the movement of mouse, it causes sensors to detect the rotation of ball along the two axes which consequently send signals to monitor screen. Figure 8.1(a) depicts the mechanical mouse.

Optical mice use light emitting diodes (LEDs), optical sensors and digital image processing. The optical mouse detects by sensing the changes in the reflected light. The change in reflected light is measured by analyzing the images and the cursor moves on screen accordingly. Figure 8.1(b) shows the optical mouse.

2. Scan Codes-When a key is pressed on a keyboard, it transfers the scan code relating to those keys to the processor. Scan code of every key is unique. The scan codes are used to communicate the desired data or action to the processor. A keyboard of processor is connected through interrupt driven I/O mechanism. Therefore, when a key or several keys are pressed together on the keyboard, it interrupts the processor, provided processor has enabled interrupts. The processor receives the scan code/codes and identifies the key or keys that were pressed using the scan code table stored in the ROM BIOS.
3. LCD monitors and LED monitors differ only in terms of backlighting; typical LCD monitors uses fluorescent backlights whereas an LED monitor uses light-emitting diodes. The earlier LCD monitors used CCFL instead of LEDs to illuminate the screen.

Check Your Progress 2

1. The display memory which is used to store the data for images is known as frame buffer. At any moment, the frame buffer consists of data for bit-map representation of current image on screen and the next image. The frames are read dozens of times per second and sent to the monitor using a cable in serial manner. Upon receiving the stream of data, the monitor forms and displays it on the screen by scanning raster movement from first up to down one row at a time. Based on this raster movement CRT, the monitor will illuminate its small phosphor dots.
2. Refer text 8.5.4
3. The minimum required video memory is computed as follows-

Number of Pixels	$= 7680 \times 4320 = 33,177,600$
Color Depth (16-colours = 2^4)	= 4-bits
Minimum Memory	= 132,710,400 bits (16,588,800 bytes)
	= 16,200 KB = 15.82 MB

4. Sound card is used to convert digital audio data into analog audio in order to play back through the speakers or headphones. The Sound card can either integrated into motherboard (built-in sound card) or connected through expansion slot. The major functions of a modern sound card are as follows:
 - a) Conversion from digital sound signals to analog form to play back the sound.
 - b) Amplifiers to augment the strength of sound signals
 - c) Sound recording.
 - d) Built-in synthesizer
 - e) Sound mixer circuits.

Check Your Progress 3

1. A digital camera without storage connected to computer system or network is referred as Webcam. In modern computers, the webcam can either be a separate component which is plugged into the computer or be an in-built integrated camera. The webcam can be used for video conference or video calling or online meeting using Google Meet, Zoom, MS Team and others services.
2. These devices recognize spoken words in two ways. The spoken words can either be recognized from a pre-defined vocabulary or be recognized from a known speaker after training of the input device. Whenever speaker utters a word from the pre-defined vocabulary, the Voice Based Input device may display the characters on monitor screen for verification by the speaker. However, some of these devices may process the speech without verification from the speaker.
3. Impact printer uses mechanical components for printing i.e., the characters and graphics are produced on a paper by striking whereas non-impact printer don't strike or impact the ribbon to print on paper. Impact printers produce banging noise during printing while non-impact printers work silently.
4. Refer text 8.10.1
5. The scanner can capture 360000 pixels per square inch

UNIT 9 INSTRUCTION SET ARCHITECTURE

- 9.0 Introduction
- 9.1 Objectives
- 9.2 Instruction Set Characteristics and Design Considerations
 - 9.2.1 Operand Data Types
 - 9.2.2 Types of Instructions
 - 9.2.3 Stored Program Organization
- 9.3 Number of Addresses and Instruction size
- 9.4 Instruction Set and Format Design Issues
- 9.5 Addressing Schemes
 - 9.5.1 Immediate Addressing
 - 9.5.2 Direct Addressing
 - 9.5.3 Indirect Addressing
 - 9.5.4 Register Addressing
 - 9.5.5 Register Indirect Addressing
 - 9.5.6 Relative Addressing Scheme
 - 9.5.7 Base Register Addressing
 - 9.5.8 Indexed Addressing Scheme
 - 9.5.9 Stack Addressing
- 9.6 Summary
- 9.7 Answers/Solutions

9.0 INTRODUCTION

In the previous two blocks, you have learnt the concepts of data representation, memory organization and Input/output organisation. This Unit discusses about one of the most fundamental aspect of a general-purpose computer – the instruction. A computer can do general purpose or specific tasks using the instructions. Instructions are the mediator between the programmer and hardware. Hardware is the computer architecture, and software is the instruction set architecture. Instructions set architecture is the only way you can interact with the computer machines. An instruction set architecture consists of a complete set of instruction to do a task on a specific computer system.

In this unit, details of instructions format, operands data types, instruction types and various addressing modes have been discussed.

9.1 OBJECTIVES

After going through this unit, you will be able to:

- ... Define various characteristics of instruction set of a computer
- ... Appreciate various components of instructions
- ... Explain the design of different instruction
- ... Define the term instruction format
- ... Explain various addressing modes used in an instruction.

9.2 INSTRUCTION SET CHARACTERISTICS AND DESIGN CONSIDERATIONS

A programmer writes the program instructions in assembly language, which are easily understandable for the humans. But the assembly language is not understandable by the computer machines, as these computer machines are made of digital modules, which understand only binary logic. Hence, during execution, program is converted into binary codes which is understandable by the computer machines. A binary coded instruction is in format, which is interpreted and executed by the machine. Instruction format is discussed next.

Instruction format

A simple instruction format is shown in Figure 9.1. It consists of three components: the operand address, the operation code or opcode and the mode. The number of bits is allocated to each component. Basic definition of these components are:

Opcode: In instruction, opcode decides the operation to be performed on the data.

Operand: In Instruction, operands are the data on which operation is to be performed.

Effective address: In instruction, it is the memory address where the data/operand is stored. For example, if data is stored in register, register is the effective address. Effective address is decided by the addressing schemes, which are discussed in details in section 9.5. The mode field of the instruction determines, how the effective address would be computed in a machine.

31	30	24	23	0
I	Opcode			Operand

Figure 9.1: Instruction format components

The total number of operations, which can be performed by a computer machine is decided by the bits allocated for the operation code of an instruction. The n-bits operation code can be used to code 2^n distinct operations. For example, the 7 bits opcode, as given in Figure 9.1, can be used to code $2^7 = 128$ distinct operations. The computer can be designed using these opcodes, such that a typical opcode may represent, a typical operation. An example opcode can be designed as:

ADD operation: 1100100 (7-bit opcode)

The number of bits allocated to operand field, of the instruction in Figure 9.1, depends on the memory address size or the data bits. In Figure 9.1, the 31st bit “I” decides the mode if address is in direct or indirect mode. In direct mode, the address given in the instruction is the effective address of the operand. In indirect mode, the instruction holds the memory address where the effective address of operand is saved. It may be noted that this instruction format just allows direct and indirect addressing mode and a single operand in each instruction. However, different instruction formats may support different types of addressing modes and different number of operands.

A point that can be noted for the instruction format of Figure 9.1 is that size of operand filed of instruction is 24 bits. In case, this operand is a direct operand, then the size of the main memory addresses that can be supported by the machine having instruction format, as given in Figure 9.1 is $2^{24} = 16$ M memory words.

9.2.1 Operand Data Types

In computer, operands are the data bits on which operation is to be performed. As shown in **Figure 9.1**, data types can be categorised as logical, integer and characters. Logical data is Boolean which may be 1/0 or true/false. Integer data may be further divided into: decimal, fixed point, and floating point. And the character data have mnemonics such as: ASCII, UNICODE etc.

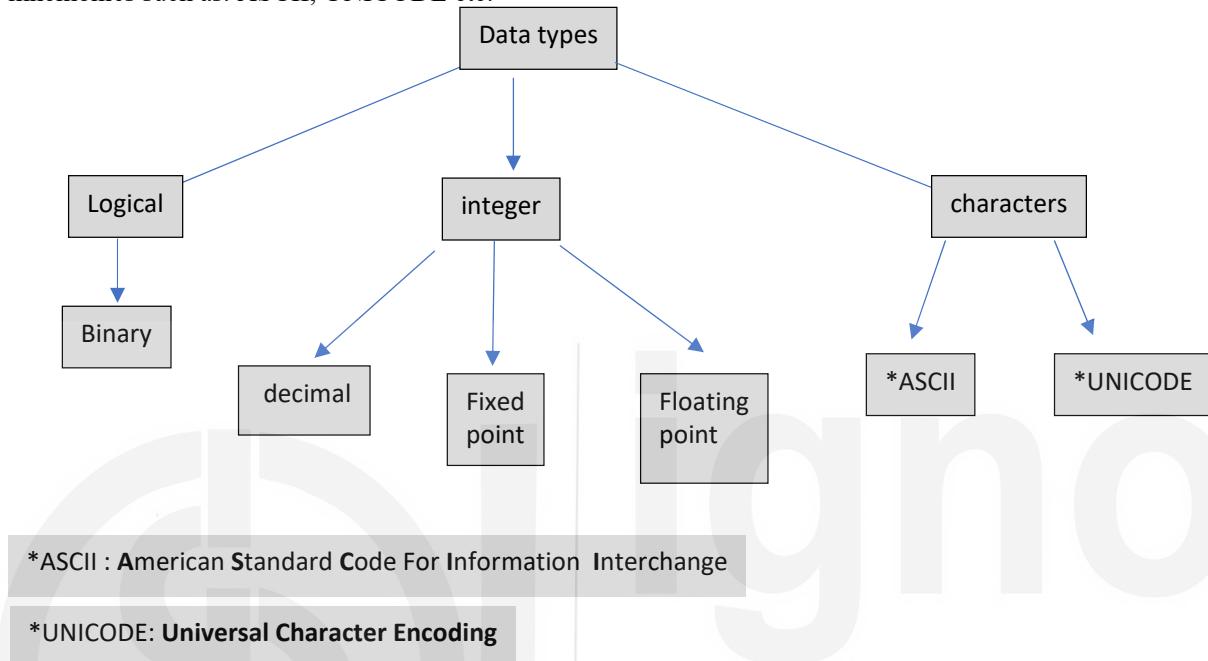


Figure 9.1: operand data types

9.2.2 Types of Instructions

To evaluate any computable function, a computer must have a group of instructions created by the designer in machine language. The set of instructions include various types such as: arithmetic instructions, logical instructions, shift instructions, instructions to transfer content from memory to processor register, program control instructions and input/output instructions. Broadly, instruction can be categorized as below:

1. Instructions to transfer the data
2. Instruction to manipulate the data
3. Instructions for program control

Instructions to transfer the data: In computer machines data transfer takes place between processor registers, between memory and processor register, between processor register and input or output interface. The instructions with data in the central processor register are faster than that of instructions with data in memory. Each instruction has a mnemonic which can be used as part of the assembly language. Please note that these mnemonics may vary for different machines. Data transfer instructions are listed below:

Table 9.1: Some Data transfer instructions of computer

<i>Instruction</i>	<i>Mnemonics</i>	<i>Remarks</i>
Load	LD	To load data from memory to accumulator register
Store	ST	To store data from processor register to memory
Move	MOV	To transfer data between processor registers
Exchange	XCH	Exchange data between processor registers or between a register and memory
Input	IN	Data transfer between processor register and input terminal
Output	OUT	Data transfer between processor register and output terminal
Push	PUSH	Data transfer from register to memory stack
Pop	POP	Data transfer from memory stack to register

Instruction to manipulate the data: Data manipulation instructions are categorized based on the type of operation they perform on the data. The following are the basic categories of data manipulation instructions:

- ... Arithmetic
- ... Logical and bit manipulation
- ... Shift instructions

Arithmetic: These instructions perform arithmetic operations on the data. Various arithmetic operations are: addition, subtraction, multiplication, division etc.

Logical and bit manipulation: The logical and bit manipulation functions are used to perform logical operations or manipulation by setting/resetting a single data bit. Some of the logical operations are -AND, OR, XOR etc.

Bit manipulation - selective set and mask: Let's assume A= 1010, and you need to set the least significant bit (LSB), keeping all the remaining bits unchanged. Since you just need to set the LSB, therefore, the second operand for selective set would be B=0001 and you will use OR operator, as shown below:

$$\begin{array}{ll}
 A & 1010 \\
 B & 0001 \\
 A \text{ OR } B & 1011
 \end{array}$$

You may please notice that upper three bits in the result (A OR B) remains unchanged, whereas, the LSB is set to 1. Hence, least significant bit of A is set to 1 by performing the OR operation. Similarly, AND function can be used to clear the selective bit. For example, if for the given A value (1011), you just want to use the

LSB or in other words you would like to make the upper three bits as 0's. This is performed with the help of AND. For this example, the value of B would be selected as 0001. This operation is also called the mask operation and is shown below:

A	1011
B	0001
A AND B	0001

Shift instructions: Shift instructions are to shift the register data bits in left or right direction. In *shl R*, data bits of the register are shifted left where data bit input takes place at the rightmost bit and leftmost bit is lost. Whereas in *shr R*, data bits are shifted right and data is input from the leftmost bit. In *shl R*, rightmost bit is lost. In case of circular shift no bit is lost. In *cil R*, data bits move to the right and rightmost bits is stored in the leftmost bit. In *cir R*, data bits move to the left and leftmost bit is stored in the rightmost bit. The Table 9.2 shows these two types of shift operations.

Table 9.2: Shift instructions

Symbolic representation	Description	Functioning	Remarks
$R \leftarrow \text{shl } R$	Shift left register R		Left most bit is lost Answer is 1100
$R \leftarrow \text{shr } R$	Shift right register R		Right most bit is lost Answer is 0011
$R \leftarrow \text{cil } R$	Register circular shift left		Left most bit is moved to rightmost bit, no bit loss. Answer is 1110
$R \leftarrow \text{cir } R$	Register circular shift right		Right most bit is moved to left most bit, no bit loss Answer is 1011

Program control instructions: The address of the next executable instruction is stored in the program counter register. Program control instructions contains the condition, which may cause address alteration in the program counter. Hence, the execution of

program control instruction results into change in program counter address breaking the sequence.

Table 9.3 : Program control instructions

Instruction	Mnemonics	Function	Remarks
Branch	BR	To branch at particular address	conditional or unconditional
Jump	JMP	Jump to a specific address	unconditional
Skip	SKP	To skip the next instruction	conditional
Call	CALL	Call is used with subroutines	To call a function
Return	RET	To return back to sequence after function call	To return after executive a function

Branch (BR) and Jump (JMP): Branch and jump instructions are used to change the flow of control of a program to a new instruction address specified as the target of branch and jump instruction. These instructions may be conditional or unconditional. For example,

JMP: is an unconditional branch to an instruction address. It may be used to implement simple loops.

JNE: (jump not equal) is a conditional branch instruction. This instruction checks the zero-flag register to determine, if two operands are equal or not (How the flags would be set? This will be explained in Block 4). The jump to the specified address of instruction will take place, only if the zero flag is not set.

Some of the conditional branch instructions are:

BRP X: branch to memory location X if the result of most recent operation is positive
 BRN X: branch to memory location X if the result of most recent operation is negative

In the following example, conditional branch has been utilized in which, PC will be loaded with memory location 707 if the content of accumulator register (AC) is zero. And an unconditional JUMP instruction is to execute the program from a particular memory location 901.

```

DR ← 0          ; Assign 0 to the memory branch register DR
AC ← M[X]      ; Read a value from memory location X and store in AC
BRZ 707         ; Branch to location 707 if AC is zero (Conditional branch)
ADD AC, DR      ; Add the content of DR to AC and store result to AC
...
JUMP 901        ; jump to memory location 901 for further processing.
    
```

SKIP: The SKIP instruction skips the next instruction to be executed in sequence. Hence, it increments the value of PC by one instruction length. The SKIP can also be conditional. For example, the instruction ISZ skips the next instruction only if the result of the most recent operation is zero.

Subroutine call and return instruction : The subroutine call instruction holds the opcode and address of the start of subroutine. On execution of a subroutine call instruction, first the return address, which is the address of the next instruction to subroutine call stored in PC is moved to the memory, and then PC is loaded with the subroutine address from the subroutine call instruction. Once subroutine execution is completed, program has to return back to the calling program to execute the next instruction after the subroutine call instruction, which was stored as return address. Therefore, all the subroutines end with the return instruction. A subroutine call is implemented as below:

$SP \leftarrow SP - 1$	stack pointer is decreased by one
$M[SP] \leftarrow PC$	PC content is pushed onto the stack to store return address
$PC \leftarrow \text{effective address}$	From the subroutine call instruction the effective address is moved to the program counter (PC) register.

While the return instruction would be executed as:

$PC \leftarrow M[SP]$	The stored return address is assigned to PC
$SP \leftarrow SP + 1$	stack pointer is increased by one

In subroutine call, the return address is saved at one location. Once call function execution is over, PC is loaded with the return address. A typical example of subroutine call and return in the context of 8086 microprocessor is explained in Block 4 of this course.

9.2.3 Stored Program Organization

A computer is organized to have processor registers and memory unit. Memory holds the program instructions as well as the operands/data on which operation is to be performed. For example, in a 16-bit memory of size 4096 words, as shown in Figure 9.3, an instruction of size one word includes an opcode and one operand address. Since the size of the memory is 2^{12} or 4096 words, the operand address would be 12-bits and the remaining 4-bits would represent an opcode, which defines the operation to be performed. Operand address defined in instruction is the memory location which holds 16-bit data.

Accumulator is the processor register. The operations are performed on the memory content and the accumulator data. A detailed and enhanced example of this structure is given in Block 4, which provides details on 8086 microprocessor.

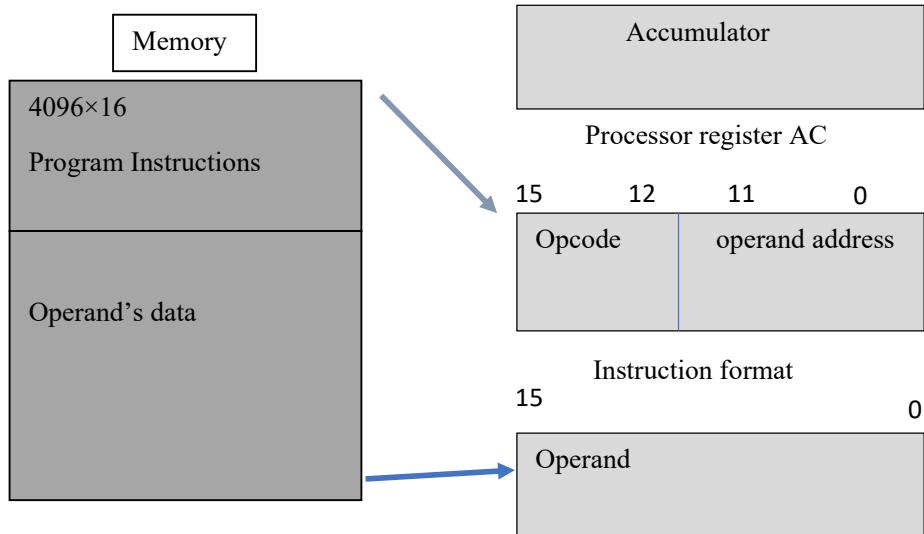


Figure 9.2: Schematic representation of stored program

Check Your Progress – 1

1. What operation can be performed on $A = 0011\ 1001$ to get a
 - a. $1001\ 1100$
 - b. $0100\ 1110$
 - c. $0010\ 0111$

2. Define opcode and operand.

3. Consider that A contains $0101\ 1110$ and B contains $0000\ 1111$, then what would be the value of
 - (a) Selective Set of A using B
 - (b) Masking of A by B

9.3 NUMBER OF ADDRESSES AND INSTRUCTION SIZE

There are instructions set with zero address, one address, two address, three address, and RISC (Reduced Instruction Set Computers). What is the impact of number of

addresses in an instruction on instruction size and program? It will be discussed with the help of program that evaluates the arithmetic function:

$$A = (W+X) \times (Y+Z)$$

Zero address instruction: The instructions used for stack organized computers do not have the address field in the instruction. As you can see in the following instructions that no address is assigned in the instruction except for the PUSH and POP commands. The operands are implicit and are taken from the top of the stack. Hence, these are called Zero address instructions. The program to evaluate the value of A is given in Table 9.4.

Table 9.4: Zero Address instructions

PUSH W	The stack top position $\leftarrow W$
PUSH X	The stack top position $\leftarrow X$
ADD	The stack top position $\leftarrow W+X$
PUSH Y	The stack top position $\leftarrow Y$
PUSH Z	The stack top position $\leftarrow Z$
ADD	The stack top position $\leftarrow Y+Z$
MUL	The stack top position $\leftarrow (W+X) \times (Y+Z)$
POP A	$A \leftarrow$ The stack top position

One address instruction: in these instructions, only one address field is present while the second operand is an implicit register operand - the AC accumulator register. The set of instructions that can solve the stated mathematical operation are discussed in

Table 9.5: One address instructions

Instruction	Function	Discussion
LOAD W	$AC \leftarrow M[W]$	Data of Memory Address W is loaded in accumulator register
ADD X	$AC \leftarrow AC + M[X]$	Data of Memory Address X is added with accumulator data
STORE A	$M[A] \leftarrow AC$	Accumulator data containing $(W+X)$ is stored in Memory Address A
LOAD Y	$AC \leftarrow M[Y]$	Data from Memory Address Y is loaded in accumulator
ADD Z	$AC \leftarrow AC + M[Z]$	Data Memory Address Z is added with accumulator data
MUL A	$AC \leftarrow AC * M[A]$	Memory address A is loaded in accumulator
STORE A	$M[A] \leftarrow AC$	Accumulator data is stored in Memory address A

Table 9.5. You can observe that all the instructions in the Table 9.5 have just one operand address specified in the instruction. We can see, instructions are only with one address.

Two addresses instruction: Here, one instruction holds two addresses which may be memory address or processor register. As listed in the following table, all the instructions hold two addresses one of these is a processor register and the other is the memory location.

Table 9.6: Two-addresses instructions

Instruction	Function	Discussion
MOV R1, W	$R1 \leftarrow M[W]$	Data of Memory Address W is moved to register R1
ADD R1, X	$R1 \leftarrow R1 + M[X]$	Data of Memory Address X is added with data of register R1 and result is saved in R1
MOV R2, Y	$R2 \leftarrow M[Y]$	Data of Memory Address Y is moved to register R2
ADD R2, Z	$R2 \leftarrow R2 + M[Z]$	Data of Memory Address z is added with data of register R2 and result is saved in R2
MUL R1, R2	$R1 \leftarrow R1 * R2$	Data of registers R1 and R2 is multiplied and saved in R1
MOV A, R1	$M[A] \leftarrow R1$	Results of R1 is moved to Memory address A

Three addresses Instructions: These instructions hold three addresses which may be processor register and/or memory. As shown in the following table, all the instructions hold three addresses.

Table 9.7: 3-address instructions

Instruction	Function	Discussion
ADD R1, W, X	$R1 \leftarrow M[W] + M[X]$	Data of memory location W and X are added and result is stored in processor register R1
ADD R2, Y, Z	$R2 \leftarrow M[Y] + M[Z]$	Data of memory location Y and Z are added, and result is stored in processor register R2
MUL A, R1, R2	$M[A] \leftarrow R1 * R2$	Data of processor registers R1 and R2 is multiplied and saved in memory location A .

Advantage of increasing the number of addresses: As you can observe from Tables 9.4, 9.5, 9.6 and 9.7 that number of instructions that are required to perform arithmetic operation decreases.

Disadvantage of increasing the number of addresses: The number of bits required to define an instruction keeps on increasing, thus, increasing the length of an instruction.

Reduced Instruction Set computers (RISC): RISC instructions use three processor registers for operations that perform computations, whereas one memory address and one processor register for the input/output instructions, such as load and store instructions. Instructions performing computational operation do not use memory. The following table illustrate a program based on RISC machine.

Table 9.8: reduced instruction set

Instruction	Function	Working
LOAD R1, W	$R1 \leftarrow M[W]$	Data of Memory Address W is loaded in processor register R1
LOAD R2, X	$R2 \leftarrow M[X]$	Data of Memory Address X is loaded in processor register R2
LOAD R3, Y	$R3 \leftarrow M[Y]$	Data of Memory Address Y is loaded in processor register R3
LOAD R4, Z	$R4 \leftarrow M[Z]$	Data of Memory Address Z is loaded in processor register R4
ADD R1, R1, R2	$R1 \leftarrow R1 + R2$	Data of processor register R1 and R2 are added and result is stored in R1
ADD R3, R3, R2	$R3 \leftarrow R3 + R4$	Data of processor register R3 and R4 are added and result is stored in R3
MUL R1, R1, R3	$R1 \leftarrow R1 * R3$	Data of processor register R1 and R3 is multiplied added and result is stored in R1
STORE A, R1	$M[A] \leftarrow R1$	Final result is stored from register R1 to memory address A

*LOAD: to load data from memory to processor register

*STORE: To store data from processor register to memory

Advantages of RISC instruction set:

- ... Memory access is limited to load and store instruction only
- ... Even though three address instructions are used, however as most of the operands are registers, therefore instruction length is small.
- ... Single cycle executable instructions
- ... All operations are performed within processor registers

9.4 INSTRUCTION SET AND FORMAT DESIGN ISSUES

In Figure 9.1 of Section 9.2 an example instruction format is shown. This instruction format stores only one operand address, which is stored in the lowest 24 bits (bit 0 to bit 23) for the instruction. In case, this address is a direct operand address, then such an instruction format may support only $2^{24} = 16$ M words memory. This computation assumes that each memory address is an address of one memory word. The opcode size is 7 bits (bit 24 to bit 30). Therefore, in general, there would be $2^7 = 128$ possible operation codes for this machine. The most significant bit (bit 31) is an addressing mode bit, which in this instruction format specifies the direct or indirect memory addressing mode. An instruction of a computer can have several addressing modes, which are explained in the next section. Thus, in an instruction format, there are three components: opcode, operand and the addressing mode. Hence, instruction length depends on the number of bits allocated to each component. The following are the issues relating to the instruction format design:

Instruction Length: Instruction length is critical in instruction format. There is trade off in smaller vs longer instructions.

- a) More operands in one instruction will result into smaller programs, as discussed three address instructions have smaller program in comparison to instructions having lesser number of addresses. However, it may increase the length of an instruction.
- b) Addressing modes: Its length is very crucial, as addressing modes give the flexibility of implementing different function.

Bits allocation to operand and opcode: Bits allocated to operand depends on the addressing mode used. For example, register addressing would require lesser number of bits than a memory address. Even the number of opcode depends on the flexibility, for example, an instruction set may have many different add operation codes for different addition operations. For example, a machine may have several addition operations like memory and one immediate operand, one memory and one register operand, two memory operand etc. each being assigned a different opcode. The addressing mode bits can facilitate bringing down such instructions. RISC computers use only register addressing (except for memory read and write instruction), thus, may simplify the instruction format.

Addressing mode: The bits allocated to addressing mode depends on the number of addressing modes used in the instruction set. The number of bits allocated will be high if number of addressing modes been used are high.

Variable length instruction: Computers use several different types of instruction formats. In certain machines these formats can be of different length. For example, an instruction format using one register operand may be a smaller format, whereas an instruction format using one direct memory operand would be of different instruction length. Complex Instruction sets comes under variable length instructions, abbreviated as CISC.

☞ Check Your Progress – 2

1. A computer with memory unit 512K words of 32 bits each. The computer has 32 registers. A binary instruction code is stored in one word memory, where instructions consist of an indirect bit, an opcode, a memory operand address and a register address.

- ... Find the bits of opcode, register address, and memory address.
 - ... Draw the instruction word format.
 - ... Find out the bits required for the address and data input of the memory?
-
.....

2. Give one instruction for each of the following:

- i. Zero address
 - ii. One address
 - iii. Two address
 - iv. Three address
-
.....

9.5 ADDRESSING SCHEMES

There is concept of addressing schemes which can be used by the programmer to get flexibility to do task. There are a number of addressing modes. A machine can support a number of these addressing modes.

Implied mode: In this mode, the operand is an implied operand for the given instruction. For example, CPLA is an instruction, which complements the accumulator register. The instruction does not contain the address of the operand register; however, it has one implied operand, i.e. accumulator register. Few other examples of implied addressing mode instructions are: INCA and DECA, where accumulator content is processed.

INCA	Increments accumulator register content by one
DECA	Decrements accumulator register content by one

9.5.1 Immediate Addressing

In this, the instruction holds the operand on which operation is to be operated, is called direct immediate operand. For example, the following instruction loads the value 20 in the accumulator register. The # sign in this instruction indicates that value 20 is an immediate operand. Thus, in immediate addressing mode an operand is part of the instruction. Such addressing modes are very useful, when you want to initialize a register to a constant value.

Load AC, #20	Loads an immediate value 20 in accumulator register.
Load R, #30	Loads an immediate value 30 in processor register R.

9.5.2 Direct Addressing

In this, the instruction specifies the memory address, where the operand is stored. This is one of the most fundamental addressing modes and is present in most of the machines, including RISC machine. For example, the following example, the content of memory location 200 would be loaded in the accumulator register.

Load AC, 200

In this instruction, memory address 200 holds the operand which is stored in the processor register accumulator. Therefore, instruction address is the effective address. Figure 9.4 illustrates this instruction at location 20. As per Figure 9.4, the location 200 contains the operand 350.

Thus, Effective Address of operand = 200
The value of operand which would be loaded in AC = 350.

Similarly, for an instruction Load R, 350 in Figure 9.4, the effective address would be 350 and the operand value 10 that is stored in the memory address 300 would be loaded in processor register R.

9.5.3 Indirect Addressing

In this addressing mode, the address given in the instruction is the memory address where effective address of operands is stored. For example, consider following instruction in the Figure 9.4:

Load AC (200)

This instruction is stored at memory location 21, as shown in Figure 9.4. The instruction address 200 contains 350 which is the effective address of the operand. The operand stored at 350 is 10. Thus, in the indirect addressing mode for the given example,

Effective Address of operand = Content of location (200) = 350

The value of operand which would be loaded in AC = 10.

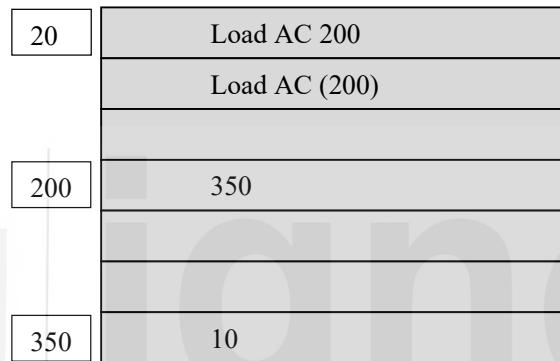


Figure 9.4: Schematic representation of Direct and Indirect addressing modes

9.5.4 Register Addressing

Operands are in register that reside within CPU. For example, consider the instruction:

LD R1

Which loads the content of register R1 to the accumulator register (AC). In this instruction, R1 is a processor register. In case, this instruction is executed on the machine shown in Figure 9.5, the effective address in this case is the address of register R1 itself. On execution of this instruction the content of R1, which is 201, will be loaded in the AC. Therefore, the content of the AC would be 201, after execution of this instruction.

9.5.5 Register Indirect Addressing

In the register indirect addressing, the instruction holds the processor register which carries the address of the operand in memory. For example, the instruction

LD (R1)

In this instruction the operands from memory location whose address is in R1 register is loaded in the accumulator.

$AC \leftarrow M[R1]$

For example, given the values stored in different registers and memory locations, as shown in Figure 9.5, the instruction LD(R1), which is a register indirect addressing mode will be interpreted as follows:

- ... The register R1 contains the value 201, therefore, the effective address of operand is memory address 201, i.e. EA=201
- ... The operand value stored in memory address 201 is 175. Hence, content of memory location 201, which is 175 is loaded in AC register.

Therefore, content of the AC after execution of this instruction would be 175.

9.5.6 Relative Addressing Scheme

In this mode, the operands are stored at the memory address obtained by adding address given in the instruction with the current program counter (PC) location. For example, the instruction:

LD \$ADR ; ADR is the value of the operand address field of instruction.

This instruction results in the operation: $AC \leftarrow M[PC+ADR]$, which means that the operands are located at memory address (effective address) $PC+ADR$. Assuming this is the instruction given in location 100 of the Figure 9.5, which can be written as:

LD \$100 ; Please note ADR=100 in this instruction.

The effective address is calculated as: content of PC + ADR of the instruction. Since the current instruction is at PC value 100, on fetch of this instruction PC would be incremented to the address of next instruction, which is 102. Thus, the effective address would be $102+100 = 202$, which means that operand is at the memory location 202.

EA = 202

Therefore, on execution of this instruction, the content of memory location 202, which is 130, would be loaded in the accumulator.

Therefore, content of the AC after execution of this instruction would be 130.

9.5.7 Base Register Addressing

In the base register addressing scheme a base register and an offset from this register is specified in the instruction. Effective address is found by adding the base register content with the address part of the instruction. For example,

LD BR, ADR

The instruction, as given above, assumes that the instruction specifies the base register (BR in this instruction) and offset from the base register in the address part of the instruction (ADR). Assuming that the instruction at the location 100 is using base register addressing, the instruction can be written as:

LD BR, 100

Since, the value of the base register BR=100 and ADR=100. The effective address would be:

$$EA = BR + ADR = 100 + 100 = 200$$

Therefore, on execution of the instruction the content of the memory location 200, which is 170, would be loaded in the accumulator.

AC is loaded with 170

Therefore, content of the AC after execution of this instruction would be 170.

9.5.8 Indexed Addressing Scheme

In the indexed addressing scheme an address of a memory location is defined and the value of an index register, which represents an offset, is added to it to find the effective address of an operand. This addressing scheme is very useful for accessing an array, where an index register points to offset in an array of values. For example, the following instruction uses the index register XR with the original offset of data is stored in ADR, which is the operand field of the instruction.

LD ADR(XR)

The following operation defines the operation performed by the instruction.

$$AC \leftarrow M[ADR+XR]$$

For example, as ADR portion of the instruction is 100 and if the index register is referring to 200th element of an array that is XR=200, then effective address would be:

$$EA = 100 + 200 = 300$$

Therefore, on execution of this instruction, the content of Memory location 300 would be loaded in the accumulator. Thus, AC will be loaded with 140 (Refer to Figure 9.5).

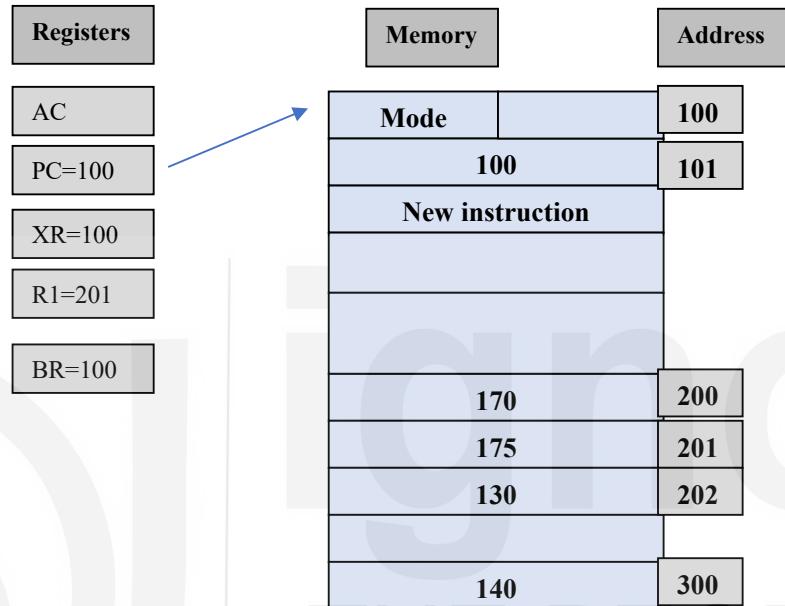


Figure 9.5: An example for addressing modes

9.5.9 Stack Addressing

Stack organization operated on Last in First out (LIFO), e.g. as you arrange books one above other in a bookshelf, while taking out the books last placed book will be lifted first. A stack is collection of finite number of words, as shown in Figure 9.6. It is a 64-word stack of memory locations. A register called Stack Pointer (SP) is used to hold the word address of top of the stack. Data register DR holds the data to be written or read from the stack. Where FULL and EMPTY are one-bit registers to find out if the stack is full or empty. Where data bit FULL is 1 when stack is full, and EMPTY bit is 1 when stack is empty. A new data is inserted in the stack using PUSH, when stack is not full, i.e. FULL=0, the PUSH operation is shown below:

$$\begin{array}{ll} SP \leftarrow SP+1 & \text{Stack pointer is incremented to an empty location} \\ M[SP] \leftarrow DR & \text{Data is written into stack top} \end{array}$$

Whereas, POP deletes a data item from the top of the stack and puts it in DR register:

$$\begin{array}{ll} DR \leftarrow M[SP] & \text{data of top-most stack location is read to DR} \\ SP \leftarrow SP-1 & \text{Stack pointer is decremented by 1} \end{array}$$

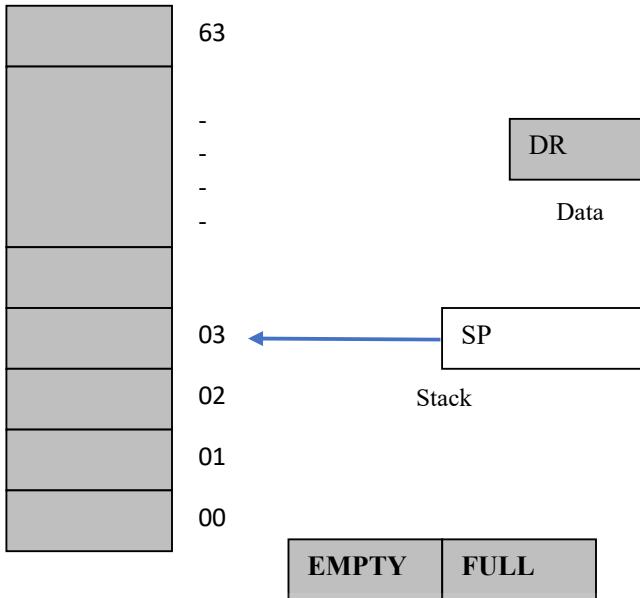


Figure 9.6: Stack organization

Check Your Progress 3:

- Find the effective address in case of each addressing modes using the Figure 9.7.

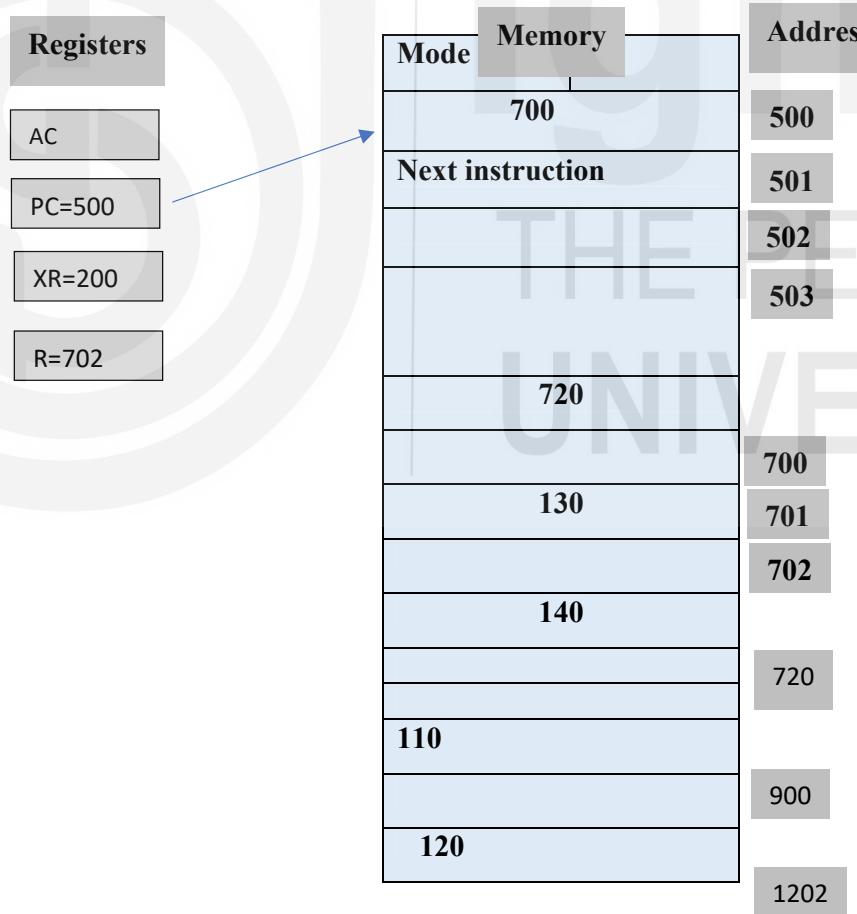


Figure 9.7: The Example Data

You may assume that the instruction is at location 500 and 501. Please note AC is accumulator register; PC is Program counter register; XR is index

register; and R is any processor register. The address of the instruction to be executed, is stored in processor register program counter.

Q 2: which instruction set is with zero address?

9.6 SUMMARY

In this unit, basics of instructions such as operand data types, instruction length, and instruction set has been discussed. The instruction format with details of operands and opcode had been explained. In addition to that, various addressing modes, and related instruction size (w.r.t. number of addresses) has been discussed. Depending on the addressing modes, the number of addresses in one instruction and the program complexity has been explained. The flexibility provided to the programmer because of the addressing modes has also been deliberated.

9.7 ANSWER TO CHECK YOUR PROGRESS

Check Your Progress 1

1. a) cir r 1-time
- b) cir r 2-time
- c) cir r 3-times
- 2.

... *Opcode*: opcode is to decides the operation to be performed on the data.

... *Operand*: operands are the data on which operation is to be performed

3. Consider that A contains 0101 1110 and B contains 0000 1111

Selective set : A OR B 0101 1111 (to set LSB 4)

Masking A AND B 0000 1110 (masked MSB 4
bits)

Check Your Progress 2

- 1 Memory unit =512K

$$= 2^{19}$$

Hence, size of memory address =19 bits

Register address for selecting one of 32 registers = 2^5
= 5 bits

Indirect bit =1

Opcode is : $32-(19+5+1)=7$ bits

Indirect bit	Opcode	Register	Address
31	30 24	23 19	18 0

1. The instructions are as follows:

- i. Zero address: In stack organised computers after two PUSH instructions, which will put two operands on a stack, the zero address ADD instruction will add the content of the top two operands.
- ii. One address: In an accumulator-based machine, which uses one of the operands as AC and stores result in AC. The one address instruction: ADD X will add the content of memory location X to the AC register and result of addition will be in AC.
- iii. Two address: An instruction like ADD R1, R2 would add registers R1 and R2 and store the result in R1 register.
- iv. Three address: An instruction like ADD R3, R1, R2 would add registers R1 and R2 and put the result in R3 register.

Check Your Progress 3

Q1:

Addressing mode	Effective address	Accumulator content	Remarks
Immediate	500	700	Instructions holds the operand data
Direct	700	720	Instruction holds the memory address where operand is stored
Indirect	720	140	Instruction holds the memory address where the operand address is stored
Relative Address	1202	120	Relative address holds the operands at: PC+ instruction content ($502+700$)
Index Address	900	110	Index address holds the operands at : $XR+700$ (Index register is added with instruction content)
Register	-	702	Operand is stored in CPU Register

Register indirect	702	130	Register holds the Memory address where operand is stored
----------------------	-----	-----	--

Question 2: Stack organization instructions PUSH and POP



UNIT 10 REGISTERS, MICRO-OPERATIONS AND INSTRUCTION EXECUTION

Structure	Page No.
10.0 Introduction	
10.1 Objectives	
10.2 Register Organization	
10.2.1 Programmer Visible Registers	
10.2.2 Status and Control Registers	
10.3 Micro-operation concepts	
10.3.1 Register Transfer Language, Bus and Memory Transfers	
10.3.2 Register Transfer Micro-Operation	
10.3.3 Arithmetic Micro-operations	
10.3.4 Logic Micro-operations	
10.3.5 Shift Micro-operations	
10.4 Instruction Execution and Micro-operations	
10.4.1 Relationship of Timing and Control, Memory reference instructions and Input-output Interrupts to instruction execution	
10.4.2 Instruction Cycle	
10.4.3 Interrupt Cycle	
10.5 Instruction Pipelining	
10.6 ALU Organization	
10.6.1 A Simple ALU Organisation	
10.6.2 A Simple ALU Design	
10.7 Arithmetic Processor	
10.8 Summary	
10.9 Solutions/ Answers	

10.0 INTRODUCTION

In the previous unit, you have gone through the concepts relating to various types of instructions and operands that a computer can have. The main task performed by the CPU is the execution of the instructions. This Unit focusses on the process of execution of these instructions by the CPU. This Units tries to answers the following two questions regarding instruction execution.

What are the steps required for the execution of an instruction? And

How are these steps performed by the CPU?

Execution of an instruction can be divided into sequence of steps, together they constitute an instruction execution sequence, called instruction cycle. Each of these steps can be termed as a micro-operation. A micro-operation is the smallest operation performed by the CPU. These operations put together execute an instruction.

For answering the second question, you may recall the basic structure of a computer. The CPU of a computer consists of an Arithmetic Logic Unit (ALU), the Control Unit (CU) and operational registers. We will be discussing the register organisation and ALU in this unit, whereas the control unit organisation is discussed in next unit.

In this unit we will first discuss the basic CPU structure and the register organisation in general. This is followed by a discussion on micro-operations that include register-transfer, arithmetic, logic and shift micro-operation and their implementation, which

forms the basis of design of a ALU. The discussion on micro-operations will gradually lead us towards the discussion of an ALU structure. The unit will also discuss about the arithmetic processor, which are commonly used for floating point computations.

10.1 OBJECTIVES

After going through this unit, you should be able to:

- ... describe the register organisation of the CPU;
- ... define what is a micro-operation;
- ... discuss an instruction execution using the micro-operations; and
- ... describe the basic organisation of ALU;
- ... discuss the requirements of a floating point ALU;
- ... create simple arithmetic logic circuits.

10.2 REGISTER ORGANISATION

The number and the nature of registers is a key factor that differentiates among computers. For example, Intel Pentium has about 32 registers. Some of these registers are special registers and others are general-purpose registers. Some of the basic registers in a machine are:

- ... All von-Neumann machines have a program counter (PC) (or instruction counter IC), which is a register that contains the address of the instruction that is expected to be executed next.
- ... Most computers use special registers to hold the instruction(s) currently being executed. They are called instruction register (IR).
- ... There are a number of general-purpose registers, which can be used for arithmetic computations or any other purpose.
- ... Memory-address register (MAR) holds the address of next memory operation (load or store).
- ... Memory-buffer register (MBR) or Memory data Register (MDR) holds the content of memory operation (load or store).
- ... Processor status bits indicate the current status of the processor. Sometimes it is combined with the other processor status bits and is called the program status word (PSW). Some processors also use flags register, which store different flags set by the processor like carry flag, overflow flag, zero flag etc.

The CPU registers have the following characteristics:

- ... CPU can access registers faster than it can access main memory.
- ... Register addressing requires less bits in the instructions for addressing than that of memory addressing. For example, for addressing 256 registers you just need 8 bits, whereas the memory size of 1MB would require 20 address bits, a difference of 60%.
- ... Compilers tend to use a small number of registers, as large numbers of registers are difficult to use effectively. A general good number of registers is 32 in a general machine.
- ... Registers are more expensive than memory but far less in number.

From a user's point of view, computers have two different kinds of registers. These are:

Programmer Visible Registers: These registers can be used by machine or assembly language programmers while programming. A good program minimizes 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 chip designer companies use some of these registers interchangeably; therefore, you should not stick to these definitions rigidly. Yet this categorization will help in better understanding of register sets of a machine. Therefore, let us discuss more about these categories.

10.2.1 Programmer Visible Registers

These registers can be accessed using machine language. In general, there are four types of programmer visible registers.

- ... General Purpose Registers
- ... Data Registers
- ... Address Registers
- ... Condition Codes Registers.

A comprehensive example of registers of 8086 is given in Unit 1 Block 4.

The general-purpose registers as the name suggests can be used for various functions. For example, they may contain operands or can be used for calculation of address of operand etc. However, to simplify the task of programmers and computers dedicated registers can be used. For example, registers may be dedicated to floating point operations. Such dedication may lead to design of data and address registers.

The data registers are used only for storing intermediate results or data and not for operand address calculation.

The address registers are used for address computation. Some dedicated address registers are:

- Segment Pointer : Used to point out a segment of memory.
Index Register : These are used for index addressing scheme.
Stack Pointer : Points to top of the stack when programmer visible stack addressing is used.

One of the basic issues with register design is the number of general-purpose registers or data and address registers to be provided in a microprocessor. 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, the optimum number of registers in a CPU is in the range 16 to 32. In case registers fall below the range then more memory reference per instruction on an average will be needed, as some of the intermediate results then must be stored in the memory. On the other hand, if the number of registers goes 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. These are called Reduced Instruction Set Computers (RISC) and they exhibit this property. RISC computers are discussed in a later unit.

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: An 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 addresses. If the size of memory is 1 MB than a minimum of 20 bits are required to store an instruction address. Please note how this requirement can be optimized in 8086 in the block 4. Similarly, the length of data register should be long enough to hold the data type it is supposed to hold. If the length of a data register is half of the size of data, then it is possible that two consecutive registers, rather than on single register, are used to store the data.

10.2.2 Status and Control Registers

For control of various operations several registers are used. These registers cannot be used in data manipulation; however, the content of some of these registers can be used by the programmer. Almost all the CPUs have a status register, a part of which may be programmer 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:

Flag	Comments
Sign flag	This indicates whether the sign of previous arithmetic operation was positive (0) or negative (1).
Zero flag	This flag bit will be set (contain a value 1) if the result of the last arithmetic operation was zero.
Carry flag	This flag is set, if a carry results from the addition of the highest order bits or borrow is taken on subtraction of highest order bit.
Equal flag	This bit flag will be set if a logic comparison operation finds out that both of its operands are equal.
Overflow flag	This flag is used to indicate the condition of arithmetic overflow.
Interrupt	This flag is used for enabling or disabling interrupts. Enable/disable flag.
Supervisor flag	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 supervisor mode it will be allowed to execute certain privileged instructions.

Figure 10.1: Flags or conditional codes

These flags are set by the CPU hardware while performing an operation. For example, an addition operation may set the carry flag and zero flag; or on a division by 0 the overflow flag can be set etc. These flags or conditional codes are tested by a program while performing typical operations like conditional branch operation. The condition codes are collected in one or more registers. RISC machines have several sets 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.

The flag register is often known as Program Status Word (PSW). It contains condition code plus other status information. There can be several other status and control registers such as interrupt vector register in the machines using vectored interrupt etc.

Check Your Progress 1

- What is an address register?

.....
.....
.....

- A machine has 20 general-purpose registers. How many bits will be needed for register address of this machine?

.....
.....
.....

- What is the advantage of having independent set of conditional codes?

.....
.....
.....

- Can you store status and control information in the memory?

.....
.....
.....

10.3 MICRO-OPERATION CONCEPTS

We have discussed the general introduction to register set. A computer executes an instruction using the arithmetic logic unit and control unit in several steps. Each of these micro steps of instruction execution is elementary in nature and is termed as a micro-operation. In general, a micro-operation should be executed in a single clock pulse by the ALU or bus transfer unit using the data stored mostly in registers.

A machine instruction is equivalent to an assembly language instruction, with the main difference being that assembly instructions use mnemonics, while machine instructions use opcode, operand addresses etc. Thus, for simplicity of explanation, we will use assembly language instructions rather than machine instructions. It may be noted that a high-level language program, first is converted to machine instructions and is executed thereafter. For example, a C programming language expression $A=A+B$; may require the following sequence of machine/assembly instructions on a hypothetical machine that uses AC and DR registers:

LDA AC, A ; Load the content of memory location A to AC register.
LDA DR, B ; Load the memory operand B to DR register
ADD AC, DR ; The content of AC and DR is added and stored in AC
STR A, AC ; Store the content of AC to memory location A.

Each of the above instructions is required to be executed by the computer. Consider the instruction LDA AC, A how will it be executed? A von Neumann machine may execute instructions in the following steps:

Step 1: Get the instruction from memory to the Instruction Register (IR): This step itself will consist of several micro-steps, which will require transfer of content among registers and using the bus.

Step 2: Decode the instruction: This will be the job of control unit (CU) and it will issue the necessary set of control signals. This step will be discussed in Unit 11.

Step 3: Fetch the operands, if needed, in the processing unit registers.

Step 4: Execute the instructions using arithmetic logic unit (ALU) and store the results back to processing unit registers.

Step 5: Store the result back to memory, if needed.

Please note that each of these steps may require several micro-steps and those micro-steps are the micro-operations. Many of these steps are due to the specific functions of registers, for example, the Program counter (PC) register stores the address of instruction that is to be executed next. Thus, in order to get the next instruction from the memory, you are required to transfer this information to the register that is used as an memory address register. Similarly, instruction once fetched from memory may be in a data register, since IR is used as input for decode operation, the instruction must be sent to IR register. The micro-operations that are used for representing data transfer between two registers or one register and one memory location using the bus are termed as register transfer micro-operations.

In addition, to register transfer, instruction execution also involves the use of arithmetic logic unit. Some of these operations which are performed by ALU on register data are add, subtract, increment, decrement etc. These are called arithmetic micro-operations. In addition, ALU can also be used to perform the logic operations, like AND, OR, NOT etc., on the data of registers. These are called the logic micro-operations. Further, shifting of data to left or right by one bit are used for many useful operations like multiplication or division. Such micro-operations may be termed as shift micro-operations.

These microoperations can be written using a register/memory transfer language which is described next.

10.3.1 Register Transfer Language, Bus and Memory Transfers

Register transfer language can be used to represent various micro-operations. In addition, this language also can be used to represent bus and memory transfer. In this language the symbol \leftarrow is used to indicate transfer of content. For example, if two registers are named R1 and R2, then a register transfer micro-operation $R1 \leftarrow R2$ implies that all the bits of register R2 are to be transferred to register R1. This operation will be feasible only if both the registers are connected through a data path and consists of same number of bits. Some of the basic rules/convention of register transfer language are listed below:

1. Naming of Register: Register would be named using capital alphabets and digits. The first character of the name should be alphabet. Figure 10.2 shows the bit organization and naming of register parts with the help of an example register IR, which is having 16 bits. A register can be partitioned in bytes and each byte may be assigned a name. For example, Figure 10.2 represent IR register of 16 bits, which can have two 8-bit partitions. You can refer to each of these bytes separately if so desired. For example, the lower byte of IR can be referred to as IR(L) and higher byte of IR can be referred to as IR(H) (Please refer to Figure 10.2)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IR(H)								IR(L)							
IR															

Figure 10.2: Register Naming and bit ordering

2. Information transfer from one register to another is designated by the symbol \leftarrow , which replaces the content of the destination register by the content of the source register, as explained earlier. The content of source register remains unchanged.
3. The control signal which enables the process of register transfer is shown with the help of a Boolean control function. This feature is very useful as operation can be controlled. For example, if c is a Boolean control variable that is it can have a value of 0 or 1, which controls the transfer of content from R2 to R1, then it will be represented as:

$c: R1 \leftarrow R2$; Please note this transfer will occur only if $c = 1$.

4. The micro-operations written on the same line are executed in the same clock time. However, such micro-operations should be free from conflict. The following example depicts a conflict in micro-operations; thus, these micro-operations should not be executed in parallel. What is a conflict in micro-operations? It is explained with the help of following example.

Example: Consider that a register R1 is to be incremented and it is also to be loaded with the content of IR register, then if you represent the following parallel micro-operations would be in conflict:

$c: Rn \leftarrow Rn+1, Rn \leftarrow IR$

These two micro-operations would update the register Rn at the same time, so one of these updated values would be lost.

5. All the register transfers occur during the falling edge transition of the clock. For more details, you may refer to further readings.

Use of Bus in register transfer

As explained earlier, a computer system may have about 16-32 or even more (in case of RISC) registers. A computer system, in general, uses an internal Bus, which consists of bus lines or wires, to transfer register data bits between two registers at a time. The control unit selects these two registers – one as the source register and second as the destination register. Thus, the register transfer operation can be expanded as follows:

$c: R1 \leftarrow BUS \leftarrow R2$

The control unit causes the register R2 to put its content (one bit on one bus line). In addition, it also enables the load operation on the R1 register. Thus, the data bits of R2 are transferred to data bits of R1.

Memory Transfer

A von Neumann machine stores the program and data in the main memory of a computer. Therefore, instruction execution requires reading and writing operations from the memory. The main memory and the processing units of a computer are connected through the system bus, which includes address, data and control bus. The address bus is used to select the specific RAM word from the memory, which in turn is transferred over the data bus. The control unit controls the entire process of data transfer by sending control signals through control bus. The two basic operations performed on the memory are Read and Write operations.

Memory Read: Read operation on the memory requires the information about the location of the memory, which is to be read. The processor decides which data word or instruction word from the memory is to be read. The address of that memory word is put in an address register (AR) and then applied on the address bus, simultaneously enabling the memory read operation using the control bus. This causes the selected word from the memory unit to be placed on the data bus part of system bus. Control unit also enables data register (DR) to accept the input from the data bus. Thus,

completing the memory read operation. The following micro-operation shows this memory read operation (please note the use of [] symbol to represent memory):

mr: DR ← [AR] ; Content of memory addressed by AR is send to DR register

Memory Write: Write operation on the memory requires – the location of the memory, which is to be written and the content, which is to be written. The processor puts the address of memory word, which is to be written, to an address register (AR) and then applies this address on the address bus, simultaneously loading the content of data to be written, which may be stored in a data register (DR) and enabling the memory read operation using the control bus. This causes the selected word on the memory unit to accept the data bus. control unit also enables data register (DR) to accept the data from the data bus. Thus, completing the memory write operation. This operation can be represented using following micro-operation:

mw: [AR] ← DR ; Write the content of memory addressed by AR register by DR

Please note, in this micro-operation the memory location, as addressed by AR, is written into. The content of AR and DR registers remains unchanged

10.3.2 Register Transfer Micro-operations

Register transfer micro-operation is one of the basic micro-operations. Two basic requirements for such a transfer are:

- ... There should exist a direct path, such as internal bus, from sender register to receiver register. It may be noted that number of bus lines and the sizes of sender and receiver register should be the same.
- ... Since a micro-operation is proposed to be completed in a single clock pulse, therefore, all the data bits on the receiver register should be loaded at the same time. Thus, the receiver register must support parallel loading of bits (Refer to Unit 4 Block1).

Following are some of the examples of register transfer micro-operations:

R1 ← R2 ; Transfer the content of R2 register to R1 register
AR ← PC ; Transfer the content of PC register to AR register

10.3.3 Arithmetic Micro-operations

Arithmetic micro-operations are performed by the arithmetic logic unit on the data stored in the processor registers. The output is also saved in a processor register. Following are some common arithmetic micro-operations:

AC ← AC + DR ; Addition of AC and DR, result in AC

AC ← AC - DR ; Subtraction of DR from AC, result in AC

AC ← AC + 1 ; Increment AC, result in AC

AC ← AC - 1 ; Decrement AC, result in AC

An adder subtractor circuit as shown in Unit 3 of Block 1 may be used to perform subtraction in machines through complement and add operations. It is represented as:

R3 ← R1 0 R2

Please note that this micro-operation can also be represented as:

R3 ← R1 + R2' + 1

Why? R2' is complement of R2, on adding 1 to it you get 2's complement of R2. Thus, both the above micro-operations are equivalent.

An addition and subtraction micro-operations can be implemented using an ALU that supports simple arithmetic operations, as discussed in section 10.4. Assuming that this ALU does implement the addition, subtraction, simple logic and shift micro-operation of fixed-point numbers, how will this machine implement multiplication and division operations on fixed point and floating-point numbers? In such a machine these operations can be implemented with the help of programs, which may use micro-operations like addition, shift and so. This kind of implementation requires that operations like fixed point multiplication and division be implemented using several micro-operation steps using several micro-instructions. Thus, fixed point multiplication and division and floating point arithmetic instructions cannot be considered as micro operations for this kind of machine.

10.3.4 Logic Micro-operations

A computer system is a binary device. It performs binary operations using bitwise Boolean operations. These bitwise Boolean operations that are implemented in the ALU forms the logic micro-operations. Three most common logic micro-operations are AND(), OR(+) and NOT(~). Other logic micro-operations, which may be implemented in different computers can be XOR, NAND and NOR. For example, you can perform bit wise AND of two registers R1 and R2 using the AND micro-operation, which can be represented as:

$$R1 \leftarrow R1.R2$$

The result of the micro-operation, as given above, will be stored in the R1 register. A typical use of this micro-operation is shown in the following example.

Example 1: Assume that two four-bit registers R1 and R2 contains the data 1100 and 1010 respectively. What would be the output if following micro-operations are performed on these two registers:

- i. $R1 \leftarrow R1 . R2$
- ii. $R1 \leftarrow R1 + R2$
- iii. $R1 \leftarrow \sim R1$

Solution:

- i. $1100 . 1010 = 1000$
- ii. $1100 + 1010 = 1110$
- iii. $\sim 1100 = 0011$

Example 2: Consider a register A containing an 8-bit value 01010011. Find the value of register B and micro-operation, which can be used to set the upper four bits of the register A, while the lower four bit remains unchanged.

Solution: To set the upper four bits irrespective of the values of A while keeping the lower four bits unchanged, the register B can consist of value 11110000 and the micro-operation OR can be used, as shown below:

Register A	0	1	0	1	0	0	1	1
Register B	1	1	1	1	0	0	0	0
A OR B	1	1	1	1	0	0	1	1

Thus, in the output the upper four bits contains value 1, while lower four bits are same as that of register A.

Example 3: Use the same value of register A, as used in example 2. Find the value of register B and micro-operation, which can be used to clear the upper four bits of the register A, while the lower four bit remains unchanged.

Solution: To clear the upper four bits irrespective of the value of A while keeping the lower four bits unchanged, the register B can consist of value 00001111 and the micro-operation AND can be used, as shown below:

Register A	0	1	0	1	0	0	1	1
Register B	0	0	0	0	1	1	1	1
A OR B	0	0	0	0	0	0	1	1

Thus, in the output the upper four bits contains value 0, while lower four bits are same as that of register A. This operation is sometimes also referred to as mask operation, where the upper four bits of the register A are masked out.

Example 4: Use the same value of register A, as used in example 2. Find the value of register B and micro-operation, which can be used to clear all the bits of a register.

Solution: To clear the entire register A, you can use register B same as that of register A and use XOR micro-operation, as shown below:

Register A	0	1	0	1	0	0	1	1
Register B	0	1	0	1	0	0	1	1
A XOR B	0	0	0	0	0	0	0	0

Example 5: Use the same value of register A, as used in example 2. Find the value of register B and micro-operation, which can be used to clear all the bits of a register.

Solution: One of the simplest ways to complement a register is to perform NOT micro-operation on register itself. An alternative native method would be to perform XOR with register B containing all 1's, as shown below:

Register A	0	1	0	1	0	0	1	1
Register B	1	1	1	1	1	1	1	1
A XOR B	1	0	1	0	1	1	0	0

10.3.5 Shift Micro-operations

Shifting of bits of a register can be used for several useful functions in a computer, such as serial transfer of data, multiplication operation, division operation. A register consists of a linear sequence of bits, which can either be shifted towards the left direction or right direction. Shifting by one bit, irrespective of left or right shift, will result in one bit to move out of the shift register from one end and one bit will be input to the register over the other end. Shift operations have been discussed in the Unit 9. Shift operations are of three basic types:

1. Logical Shift: In the logical shift, the input bit is kept as 0 and output bit is discarded.
2. Arithmetic shift: In arithmetic shift the sign of the number is kept the same
3. In circular shift the output bit is circulated to the input.

Check Your Progress 2

1. Explain how the memory read and memory write micro-operations can be performed in a von Neumann machine

.....
.....
.....

2. A simple ALU just performs the addition operation, logic operations and shift operations. Will multiplication on this machine be implemented as a micro-operation? Justify your answer.

3. Consider a register R1 contains 00010011. Select a suitable register R2 and sequence of logic micro-operations that can perform the following tasks:

- (i) Reset the complete register R1, you must use this using Mask micro-operation and not using XOR.
- (ii) Insert the 11001000 data into the register R1. You may use more than one micro-operation to do this task.

10.4 INSTRUCTION EXECUTIONS AND MICRO - OPERATIONS

To execute an instruction, a sequence of micro-operations needs to be executed. This sequence of micro-operations is essentially controlled by using a timing signal. In addition, an instruction execution involves operands. These operands can be stored either in the registers or the memory of a computer. In other words, every computer has register reference and/or memory reference instructions. Finally, in general, input-output on a computer, as discussed in Block 2, may use input-output interrupts. The next sub-section discusses the importance of these concepts in the context of instruction execution. This is followed by discussion on instruction cycle and interrupt cycle.

10.4.1 Relationship of Timing and Control, Memory reference instructions and Input-output Interrupts to instruction execution

Timing and Control: The basic task of a computer system is to execute instructions. Each instruction is executed as a sequence of steps. The execution of instructions follows a typical sequence, which is termed as instruction cycle, which is discussed next sub-section. Thus, each instruction cycle consists of several steps, which are executed in a sequential order to execute the instruction correctly. How will computer ensure that the exact sequence of instruction is followed?

As discussed in Unit 4, a computer system uses a clock for synchronization. The role of a clock is to continuously emit a sequence of clock pulses. In addition, Unit 4 also highlighted few important sequential circuits like registers, counters, multiplexer etc. Most of the sequential circuits change their state on the falling edge of the clock pulse. Thus, a particular action like loading of a register will be complete at the falling edge of the clock pulse. The next sequential action will be completed in the subsequent clock pulse and so on. For example, loading of a register may require one clock pulse, which may be followed by some other operation on the register in the next clock pulse. The counter is the circuit, which can be used to count the sequence of clock pulses. Thus, timing sequences are utilized to control the sequence of operations in a computer system. You may refer to the further readings for more details on these topics.

Memory Reference Instructions: The memory reference instructions access its operands from the memory. In general, these operands may use direct or indirect addressing schemes. In the direct memory access scheme, the address part of the instruction contains the direct reference of a memory operand, or in other words the

address part of the instruction contains the address of the operand. Thus, in order to process such instructions these operands are to be fetched from the memory to some data register of the processor. In the indirect memory reference, the instruction address is the operand address. Thus, two memory accesses would be needed to fetch such operands. First to get the address of the operand and second to get the operand from the memory.

It may be noted that a memory reference, in general, is slower than the reference to a register, as main memory is somewhat slower than the registers. However, with the use of fast cache memories, the difference in speed of the memory access vis-à-vis register access is being reduced. Many memory organizations have also been designed to reduce this speed gap. You may refer to the further readings for more details on these topics.

Input-output Interrupts: Interrupt is a condition, which may result in break in the sequence of instruction execution. Interrupts are used in computers for input-output as well as recovering from a condition that may cause an error, for example, division by zero. An Input-Output interrupt can be used for Input or output of data. For example, a program, while it is being executed by the processor, may have a command for reading of data from a keyboard. The processor would display the prompt for input of data for this program and may go on to execute other programs, while suspending this program as it needs the input. However, as soon as you enter the input from the keyboard, it may issue an interrupt to the processor, which in turn will process the input. More details on Interrupts are already given in Block 2. You may also refer to further readings for more details on interrupts.

Next section discusses the instruction cycle.

10.4.2 Instruction Cycle

As discussed in Unit 1, a von-Neumann machine having a basic set of registers executes instruction. These instructions consist of the following steps of execution:

Step 1: Get the Instruction from the memory to IR, the instruction register, let us call this operation as Fetch the Instruction (FI).

Step 2: Decode the Instruction, which is in the IR register, let us call this operation as DI.

Step 3: In this step operand address is converted to a direct address. are fetched, let us call this step as fetch the direct operand address (FoA).

Step 4: Perform the execution of instruction, let us call this step as execute instruction (ExI)

Step 5: Once execution of instruction is complete, check and acknowledge interrupt if an interrupt has occurred, let us call this step as check and acknowledge interrupt (CaI)

How are these steps can be translated to micro-operations? This section uses the register set as given in Section 10.2. In addition, it assumes the instruction format as given in Unit 9, where an instruction has three fields, viz. Indirect bit, opcode of 7 bits and one operand address of 24 bits. It may please be noted that the assumed instruction set has only direct memory and indirect memory addressing schemes. Further, only for the purpose of simplifying the discussion, it is assumed that memory reference and register reference is performed in almost equal time.

Step 1 (FI cycle): An instruction is available in the memory and program counter register (PC) points to this instruction, which is to be executed. Thus, in order to get the instruction from the memory, bus would be used along with memory address register (MAR). Thus, a sequence of operations would be needed to perform this operation. This sequence is represented with the help of a timing sequence using the timing control T1, T2, etc. Please note the micro-operation with timing control T1 will

be performed prior to micro-operations with timing control T2 and so on. Figure 10.4 lists the micro-operation sequence of FI cycle.

<ul style="list-style-type: none"> ... Transfer the content of PC to MAR. ... Apply MAR on address BUS; Control unit enables the memory Read operation and DR is enabled to receive content on the data BUS. Thus, the content of memory location pointed by MAR is read to DR. ... Perform the following two operations in parallel at time T3: <ul style="list-style-type: none"> o Increment PC so that it points to the next instruction to be executed. (PC is incremented by one memory word length, as it is assumed that each instruction is just one word long and memory address is a word address). o The instruction in DR is sent to IR to complete the FI cycle. 	T1: MAR \leftarrow PC T2: DR \leftarrow (MAR) T3: PC \leftarrow PC +1 : IR \leftarrow DR
--	---

Figure 10.4: Fetch cycle

Step 2: DI: Control unit performs the decoding of instruction. It identifies the two important things; first what operation is to be performed and second what addressing modes are used by the instruction. The addressing modes are to be decoded so that the data is brought in the ALU registers for executing the decoded operation. Since decode operation is performed by the control unit, more details related to this operation would be discussed in Unit 11.

Step 3: FoA: In this step the operand address is converted to the direct operand address and is stored in the address part of instruction. This address is used in the next step for instruction execution. In the case of the present instruction format only two possible addressing types – direct and indirect. In the case of direct addressing the address of the operand is already in the operand address part of instruction, thus, no additional micro-operation is needed. However, in case of indirect addressing, the address of the operand is to be fetched using the address portion of the instruction. This fetched address should replace the current address portion of the instruction. This step is shown in Figure 10.5:

When Direct Addressing is used: ... No action is needed.	IR (Address) contains the direct address of the operand.
When Indirect Addressing is used: ... Transfer the operand address portion of instruction to MAR. ... Read the memory using MAR and bring operand address in DR ... Transfer address from DR to IR. Now, IR has a direct address.	T1: MAR \leftarrow IR(Address) T2: DR \leftarrow (MAR) T3: IR(Address) \leftarrow DR(Address)

Figure 10.5: An Indirect cycle

Thus, the IR now contains the direct address of the operand.

Step 4: ExI: The instruction execution is also performed with the help of micro-operations. The first step of instruction execution will require the operand to be brought from the address of main memory to the processor register. This is followed by the arithmetic micro-operation as per the requirements of instruction opcode. The following examples explain the micro-operations required to execute certain instructions.

- (1) The step required to execute a simple addition instruction of the form (assuming that indirect bit is set to 0, i.e., it is a direct instruction):
ADD ADR

The following sequence of micro-operations, as given in Figure 10.6, would execute this instruction (after FI cycle):

.. Transfer the ADR in instruction to the MAR.	T1: MAR \leftarrow IR(ADR)
.. Load DR by reading the memory location referred to by ADR.	T2: DR \leftarrow (MAR)
.. Control units then enables addition of AC and DR using ALU. Result is put in AC.	T3: R1 \leftarrow R1 + DR

Figure 10.6: Execution cycle of Add instruction

- (2) Micro-operations for execution of a conditional jump instruction, which skips the next instruction, if on incrementing the operand results in zero. The incremented value is stored back to the operand location.

INCSKP ADR

The sequence of micro-operations required for this instruction execution are given in Figure 10.7.

Step 1: Bring the operand stored in location ADR to a processor register AC. AC is incremented. ... Transfer the ADR of IR to the MAR. ... Read ADR to DR ... Transfer DR to AC, as increment operation can be performed in AC (assumption). ... Increment the AC. This will set the Flag register.	T1: MAR \leftarrow IR (ADR) T2: DR \leftarrow (MAR) T3: AC \leftarrow DR T4: AC \leftarrow AC + 1
Step 2: Store the AC to ADR ... Transfer the content of AC to DR. ... Store DR into ADR using MAR and If the content of AC is zero then the next instruction is not to be executed or skipped, to do so increment the value of PC. The control unit checks the zero flag and increments PC if condition is fulfilled.	T5: DR \leftarrow R1 T6: (MAR) \leftarrow DR T6: If AC = 0 then PC \leftarrow PC + 1

Figure 10.7: Execution cycle of increment and skip instruction

- (3) This example shows the sequence of micro-operations required for a branching operation, using a subroutine call instruction. A subroutine call instruction is required to store the return address and then start execution of the subroutine, which will require the change in the value of the PC register. The return address, in general, is stored on a stack, however, for this example, it is assumed that the return address is stored in the subroutine address (ADR) specified in the call instruction. Assume the following is a subroutine call instruction:

SUBCALL ADR

The sequence of micro-operations to execute this instruction are given in Figure 10.8.

<ul style="list-style-type: none"> .. Transfer ADR to MAR and the return address, which is in PC is put in the DR. .. To branch to the subroutine, the ADR should be moved to PC. Further, at this address (ADR), the return address is to be put. Please note that ADR is already in MAR at time T1. .. The first instruction of the subroutine starts at the ADR plus one, thus, increment PC. 	T1: MAR \leftarrow IR(ADR) T1: DR \leftarrow PC T2: PC \leftarrow IR (ADR) T2: (MAR) \leftarrow DR T3: PC \leftarrow PC + 1
---	---

Figure 10.8: Execution cycle of subroutine call instruction

It may be noted that the sequence of micro-operations required to perform instruction execution, viz. FI, DI, FoA and ExI are machine dependent. In this section, we have presented a very simple example for the same.

10.4.3 Interrupt Cycle

Interrupt Processing: In addition to execution of the instruction, the processor should also respond to the interrupt, which may have occurred while instructions of a program are getting executed. It may be noted that programs may have priority and may allow only some of the interrupts, while they are executing. Thus, every computer has provision of enabling interrupts as per their priority. Now, the question is how the processor will check if an enabled interrupt has occurred? One of the solutions to this problem is to keep a control line for interrupt signal and check it after each instruction cycle. Next question is how to acknowledge an interrupt? To acknowledge an interrupt, processor may perform an interrupt cycle, as given in Figure 10.9. The interrupt cycle has been explained assuming that a stack is used to store the address of the instruction, from where the program will be restarted once the interrupt is processed.

<ul style="list-style-type: none"> .. The return address is in the PC register, which is to be stored on a stack, named STACK, with stack top pointed by stack pointer register (SP). It is being represented as STACK[SP]. DR register is to be used for this operation. .. Increment the stack pointer to next location and start executing the interrupt servicing program by transferring the address of its first instruction, say ISRAddress, to PC. 	T1: DR \leftarrow PC T2: STACK[SP] \leftarrow DR T3: SP \leftarrow SP+1 T3: PC \leftarrow ISRAddress
--	---

Figure 10.9: Interrupt cycle using a stack

It may be noted that even interrupt servicing programs are a sequence of instructions. Each of these instructions are executed as per instruction cycle. You may refer to the further readings for more details on instruction cycle.

10.5 INSTRUCTION PIPELINING

In the previous section, you have gone through various steps of instruction execution. Can these steps be performed in parallel to execute an instruction? The answer is NO, as to execute an instruction these steps are to be performed in sequence. However, can the steps of different instruction be performed in parallel to each other or can be executed in an overlapped manner. Execution of several instructions in parallel will require several processing elements, rather breaking the execution of instruction into steps and executing instruction in an overlapped manner may be useful. This is the principle of instruction pipelining. Thus, a simple instruction pipeline would require to execute instructions in an overlapped way, which will facilitate and reduce the time of

overall instruction execution. This would require that instruction cycle should be divided into equal parts which can be executed in parallel for different stages of instructions. One such decomposition of instruction cycle stages, also called pipeline stages are -fetch the instruction (FI), decode the instruction (DI), fetch operand address(FoA) and execute the instruction (ExI). A new stagehas been added here that allows storage of result back to the memory location, let us call it StR.

Figure 10.10 shows the overlapped execution of seven instructions using a five stage instruction pipelining.

Time Slot	1	2	3	4	5	6	7	8	9	10	11
Instruction 1	FI	DI	FoA	ExI	StR						
Instruction 2		FI	DI	FoA	ExI	StR					
Instruction 3			FI	DI	FoA	ExI	StR				
Instruction 4				FI	DI	FoA	ExI	StR			
Instruction 5					FI	DI	FoA	ExI	StR		
Instruction 6						FI	DI	FoA	ExI	StR	
Instruction 7							FI	DI	FoA	ExI	StR

Figure 10.10: Instruction Pipeline

In Figure 10.10, you may notice that at time slot time slot 5 the pipeline is executing 5 instructions simultaneously, though in different stages. At the end of time slot 5, execution of the first instruction is completed, thereafter, at the end of each time slot a new instruction would be completed or in other words the first instruction gets completed at the end of time slot 5, the second instruction gets completed at the end of time slot 6, the third instruction gets completed at the end of time slot 7 and so on. Thus, the execution of instruction in an overlapped fashion has resulted in almost one instruction execution in one time slot. However, the instruction pipelines suffer from the problem of resource conflicts. For example, in the 5th time slotThe first instruction is storing the result, therefore, would require memory reference; at the same time slot the second instruction is in the execution stage, thus, it will fetch the operand using memory reference and use the ALU to execute this instruction; at the same time the third instruction is in operand fetch stage, which also requires memory reference; the fourth instruction is in the decode stage; and the fifth instruction is in the fetch instruction stage, which also requires memory reference. Thus, the pipelined processor should allow each of these instructions to reference memory simultaneously through different paths so that there is no conflict, otherwise the instructions cannot be executed in the pipeline fashion.

The Problems relating to Pipelined execution:

- ... Pipelined execution may suffer from resource conflict as explained above.
- ... The pipelined execution seems good for execution of sequence of instruction, but instructions that require transfer of control, like conditional branch instruction, may cause disruption of pipeline sequence, as the decision whether a branch will be taken or not, can occur only at the execution stage. In case a branch is to be taken then all the subsequent instructions, which were already fetched in the pipelineare to be removed from the pipeline.

Some solutions for problems related to pipelined execution:

The branch penalty can be minimized using any of the following schemes:

- ... Predicting, if branch will be taken or not and accordingly fetching the next instruction.
- ... Making provision of pre-fetching those instructions, which may be executed because of a branch.
- ... Not allowing fetching of the next instruction to pipeline till the branch decision is made.

Check Your Progress 3

- 1) What is the need of the indirect cycle? Will indirect cycle be needed even if an instruction use register addressing schemes? Justify your answer.

.....
.....
.....

- 2) What is fetch cycle? Do the present-day machines also have this cycle?

.....
.....
.....

3. What is the role of Interrupt cycle?

.....
.....
.....

10.6 ALU ORGANISATION

An ALU performs simple arithmetic-logic and shift operations. The complexity of an ALU depends on the type of instruction set, which has been realized for it. A simple ALUs can be constructed for performing computation on fixed-point numbers. An ALU for floating-point arithmetic implementation requires more complex control logic and data processing capabilities. Several micro-processor families utilize only fixed-point arithmetic capabilities in the ALUs. For floating point arithmetic or other complex functions, they may utilize an auxiliary special purpose unit. This unit is called arithmetic processors. Let us discuss all these issues in greater detail in this section.

10.6.1 A Simple ALU Organisation

An ALU consists of circuits that perform data processing micro-operations. Figure 10.11 shows the organisation of a fixed point ALU was suggested by John von Neumann in his IAS computer design.

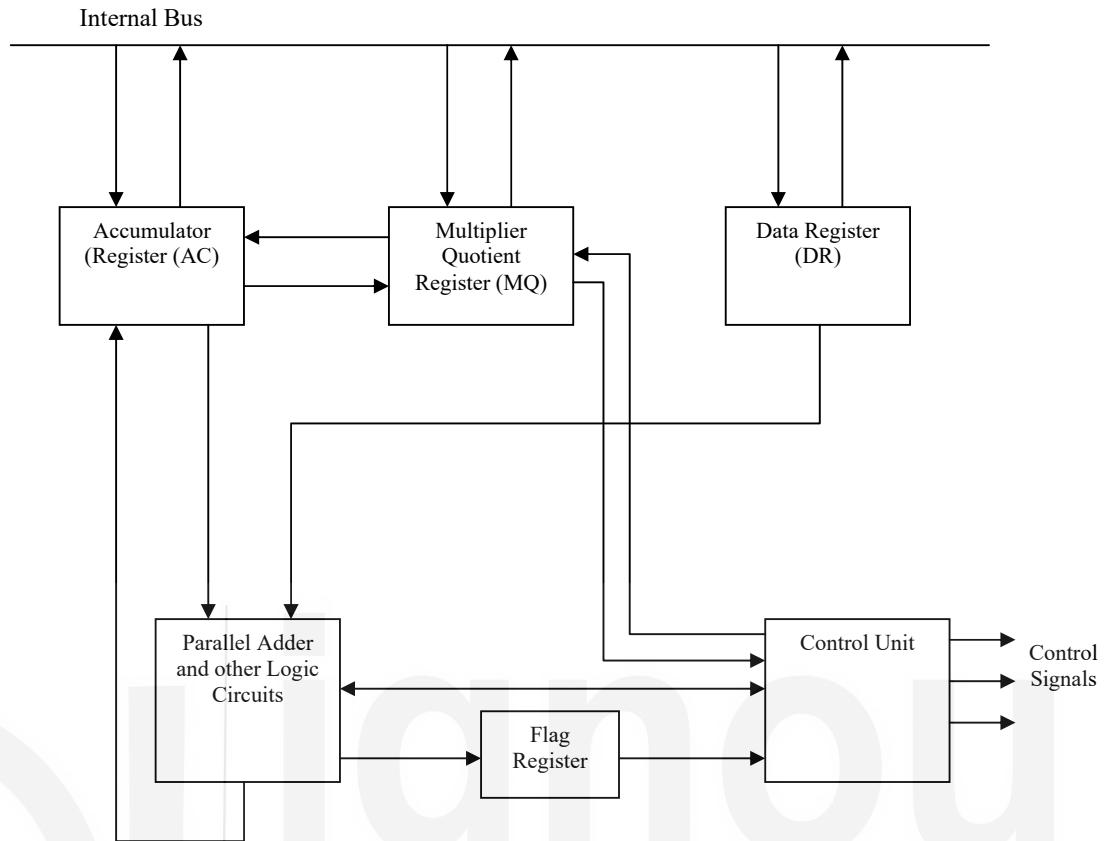


Figure 10.11: Structure of a Fixed-point Arithmetic logic unit

The above structure consists of three registers AC, MQ and DR, with the assumed size of one word each. Please note that the Parallel adders and other logic circuits (these are the arithmetic, logic circuits) this von Neumann machine can have at most two inputs and one output. In other words, it implies that any ALU operation at most can have two input values and will generate single output along with the other status bits. In the Figure 10.11, the two inputs are AC and DR registers, while output is AC register. AC and MQ registers are generally used as a single AC.MQ register. This register is capable of left or right shift operations. Some of the micro-operations that can be defined on this ALU 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 AC'

In this ALU organisation multiplication and division were implemented using shift-add/subtract operations. The MQ (Multiplier-Quotient register) is a special register used for implementation of multiplication and division instructions. Please note that in the ALU shown in Figure 10.11, the multiplication and division instructions are not implemented directly using the logic circuits. For more details on these algorithms please refer to further readings. One such algorithm is Booth's algorithm and you must refer to it in further readings.

For multiplication or division operations DR register stores the multiplicand or divisor respectively. The result of multiplication or division on applying certain algorithm can finally be obtained in AC.MQ register combination. Please note that these are not micro-operations for the given ALU organization, as execution of these two instructions would require a series of shift-add operations.

DR is another important register, which is used for storing second operand. In fact, it acts as a buffer register and stores the data brought from the memory. In machines where we have general purpose registers any of the registers can be utilized as AC, MQ and DR. For more details on ALUs, you can go through the further readings.

10.6.2 A Sample ALU Design

ALU consists of circuits that executes the micro-operations. The data is input to ALU through registers and the output of ALU is also stored in an output register. For performing the input/output of data to ALU, a BUS is used. So, let us first explain how an internal bus can be used for Data transfer.

A computer processor has large number of registers. These registers are used to store data that is required to be processed by the ALU. But, how is the data communicated among these registers? One possibility is to create separate data paths from every register to all other registers, however, this connection structure would waste large amount of processor resources. Therefore a shared media call internal BUS. A BUS consists of shared data lines, which are then connected to every register. Using these shared lines any two registers can communicate with each other, at a time. The number of lines in the shared BUS is kept same as the size of registers.

A register is selected for the transfer of data through bus with the help of control signals. The common data transfer path, that is the bus lines, are made using the multiplexercircuit. Figure 10.12 shows an example of 2-bit data bus using 2×1 multiplexers. Please note the size of the registers is also of two bits.

The construction of a bus system for two 2-bit registers using two 2×1 multiplexers is shown in the Figure 10.12. Each register has two bits, viz. Bit 1 and Bit 0. Each multiplexer has 2-bit data input, numbered 0 and 1, and one control or selection lines, C_0 . The circuit assumes no enable bits. The 0^{th} data input of MUX 0 is connected to the corresponding Bit 0 of Register A and the 1^{st} data input of MUX 0 is connected to Bit 0 of Register B. Similarly, the Bit 1 of Register A and Bit 1 of Register B are connected to 0^{th} data input and 1^{st} data input of MUX 1 respectively. There is just 1 selection line S, when S is 0, then 0^{th} input values for MUX 0 and MUX 1 are transferred to the output, that is the content of Register A is transferred on the bus, whereas, when S is 1 bits of Register B are selected for transmission on the bus.

Register A

Register B

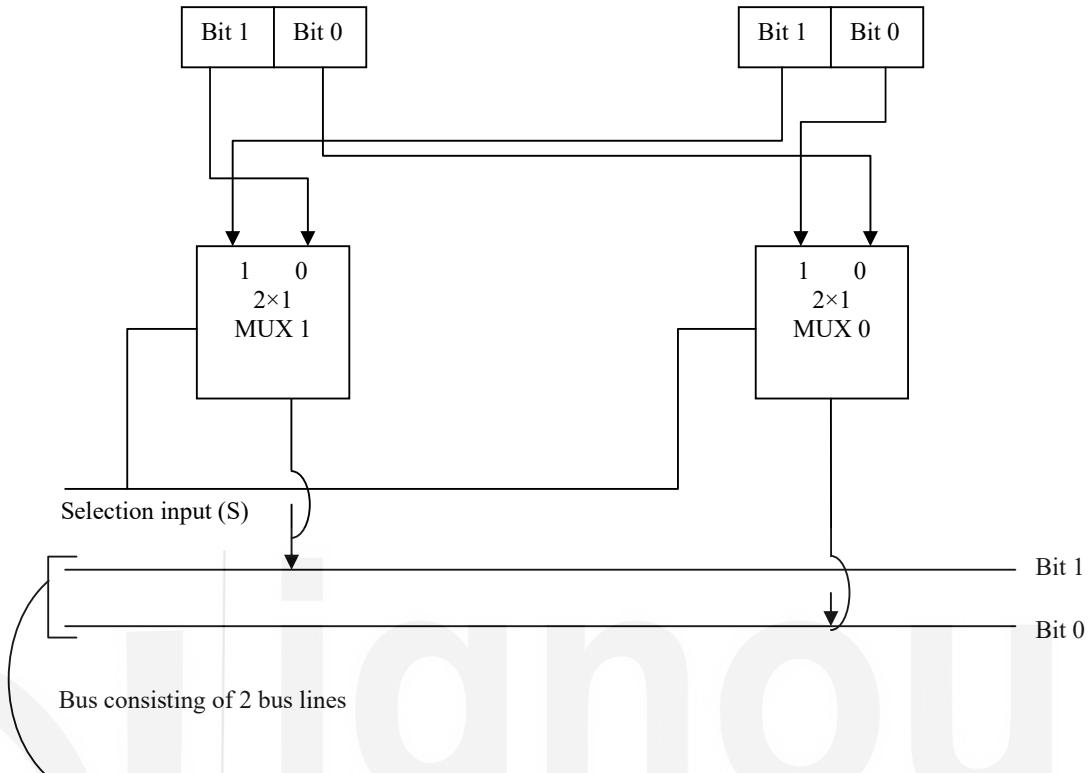


Figure 10.12: Implementation of a 2-bit BUS

The Figure 10.13 lists the selection of data based on the selection input to the multiplexers.

S	MUX 1	MUX 0	Comments
0	Bit 1 of Register A	Bit 0 of Register A	Content of Register A is put on the bus lines.
1	Bit 1 of Register B	Bit 0 of Register B	Content of Register B is put on the bus lines.

Figure 10.13: Bus Line Selection

Thus, to construct a bus for 2 registers of 2-bits each, you would require two 2×1 multiplexers. Similarly, to construct a bus for 8 registers of sixteen bits each, you would require sixteen 8×1 multiplexers, which will have 3 selection input. Please note one multiplexer is needed for transfer of one bit. Since sixteen bits are to be transferred, therefore, sixteen multiplexers would be needed. Further, one of the 8 registers would be selected to transfer data on the BUS, therefore, 3 selection input would be needed, as $2^3 = 8$.

Implementation of Arithmetic Circuits for Arithmetic Micro-operation

An arithmetic circuit can be implemented using a number of full adder circuits or parallel adder circuits, one such circuit is shown in Unit 3 of Block 1. Figure 10.14 shows a simple circuit for a 2-bit arithmetic circuit. The circuit is constructed by using 2 full adders and two 2×1 multiplexers, which requires just one selection input. Please recollect that a full adder circuit adds two input bits and one carry-in bit to produce one sum-bit and one carry-out-bit.

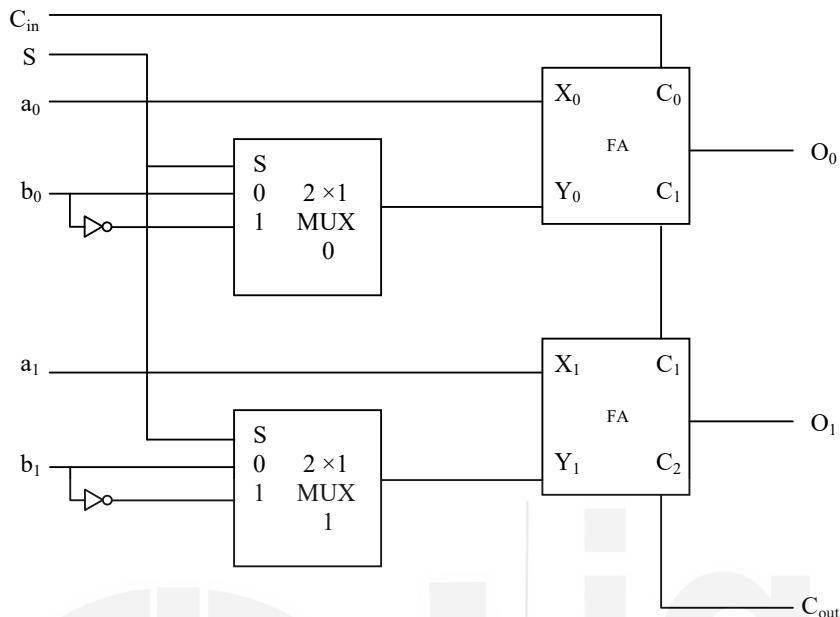


Figure 10.14: A two-bit arithmetic circuit (adder-subtractor)

The multiplexer controls one of the input to the circuit resulting in a set of micro-operations. Let us find out how the multiplexer control lines will change one of the Inputs for Adder circuit. Figure 10.15 shows the two inputs that are possible in the Figure 10.14. (Please note the convention used in this table, viz. uppercase alphabet indicates a 2-bit data word, whereas the lowercase alphabet indicates a bit.)

Control Input	Output of 4×1 Multiplexers		Y input to Adder	Comments
	MUX 0	MUX 1		
0	b ₀	b ₁	B	The data word B is input to Full Adders
1	Complement of b ₀	Complement of b ₁	B'	1's complement of B is input to Full Adders

Figure 10.15: Input to full adders using the multiplexers in Figure 10.14

Now let us discuss how by using the carry-in-bit (C_{in}) and these input values, you can obtain various micro-operations.

Input to Circuits

- ... Register A bits as a_0 and a_1 , are input to X_0 and X_1 bits of the Full Adders (FA).
- ... Register B bits are input as given in the Figure 10.15 to form the Y input to FA.
- ... Please note that each bit of register A and register B is fed to different full adder unit.
- ... Please also note that the A input directly goes to adder but B input can be manipulated through the Multiplexers to create different input values, as shown in Figure 10.15. The B input is controlled by the selection line S.
- ... The input carry C_{in} , which can be equal to 0 or 1, is input to the full adder that adds the least significant bits. The carry out of this full adder is then fed to the

full adder of the next higher bit and so on. The carry out of the most significant bit full adder is the output of the circuit. Logically it is the same as that of addition operation performed by us. We do pass the carry of lower digits addition to higher digits. The following Boolean function represents the output of this adder circuit:

$$O = X + Y + C_{in}$$

Please note that in Figure 10.15, the value of X is a direct input, but the value of Y is input through the multiplexer using the selection input S. In addition, the value of C_{in} is another input. The arithmetic micro-operations that can be implemented using Figure 10.15 are given in Figure 10.16.

S	C_{in}	Y	$O = X+Y+C_{in}$	Equivalent Micro-Operation	Micro-Operation Name
0	0	B	$O = A + B$	$R \leftarrow R1 + R2$	Add
0	1	B	$O = A + B + 1$	$R \leftarrow R1 + R2 + 1$	Add with carry
1	0	B'	$O = A+B'$	$R \leftarrow R1 + R2'$	Subtract with borrow
1	1	B'	$O = A+ B' + 1$	$R \leftarrow R1 + 2's complement of R2$	Subtract

Figure 10.16: Arithmetic Micro-operations implemented using Figure 10.15

Let us refer to some of the cases of the Figure 10.16.

When $S= 0$, input line B is applied directly to the Y inputs of the full adder. Now,

If input carry $C_{in}= 0$, the output will be $O = A + B$
 If input carry $C_{in}= 1$, the output will be $O = A + B + 1$.

If you choose $S= 1$, then B' forms the Y input to the full adder. So,
 If $C_{in}= 1$, then output $D = A + B' + 1$. This is called subtractmicro-operation. (Why?)

Reason: Please observe the following example, where $A = 0111$ and $B=0110$, then $B'=1001$. The sum will be calculated as:

$$\begin{array}{r}
 0111 \quad (\text{Value of } A) \\
 + 1001 \quad (\text{Complement of } B) \\
 \hline
 1\ 0000 + (\text{ignore the carry out bit and Add Carry in } = 1) \\
 = 0001
 \end{array}$$

Thus, it is a subtract micro-operation.

If $C_{in}= 0$, then $D = A + B'$. This is called subtract with borrow micro-operation. (Why?). Let us look into the same addition as above:

$$\begin{array}{r}
 0111 \quad (\text{Value of } A) \\
 1001 \quad (\text{Complement of } B) \\
 \hline
 1\ 0000 + (\text{Carry in } = 0) = 0000
 \end{array}$$

This operation, thus, is equivalent to:

$$\begin{aligned}
 O &= A + B' \\
 O &= (A - 1) + (B' + 1) \\
 \Rightarrow O &= (A - 1) + 2's \text{ complement of } B \\
 \Rightarrow O &= A - (B+1) \quad \text{Thus, is the name subtract with borrow}
 \end{aligned}$$

Two special cases:

When $S= 0$, $C_{in}=1$ and Y input is ZERO.

$$\text{The output } O = A + 0 + C_{in} \Rightarrow O = A + 1$$

This micro-operation is an increment micro-operation.

When $S = 1, C_{in} = 0$ and input word Y has all 1's.

Output $O = A + All(1s) + C_{in} \Rightarrow D = A - 1$ (How? Let us explain with the help of the following example).

This is a decrement micro-operation.

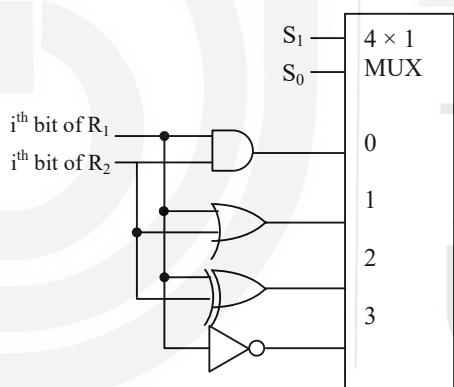
Example: Let us assume that the Register A is of 4 bits and contains the value 0101 and it is added to an all (1) value as:

$$\begin{array}{r} 0101 \\ + 1111 \\ \hline 1 0100 \end{array}$$

The 1 is carry out and is discarded. Thus, on addition with all (1's) the number has actually got decremented by one.

Implementation of Logic Micro-operations

In many computers only four logic micro-operations, viz. AND, OR, XOR and NOT logicmicro-operations, are implemented. The other logic micro-operations can be derived from these four micro-operations. Figure 10.17 shows one bit, which is the i^{th} bit stage of the four logic operations. Please note that the circuit consists of 4 gates and a 4×1 MUX. The i^{th} bits of Register R1 and R2 are passed through the circuit. On the basis of selection inputs S_0 and S_1 the desired micro-operation is obtained.



(a) Logic Diagram

S₁	S₀	Output	The Operation
0	0	$F = R_1 \wedge R_2$	AND Operation
0	1	$F = R_1 \vee R_2$	OR Operation
1	0	$F = R_1 \oplus R_2$	XOR Operation
1	1	$F = R_1'$	Complement of Register R ₁

(b) Functional representation

Figure 10.17: Logic diagram of one stage of logic circuit

Arithmetic, Logic and Shift Unit

So, by now we have discussed how the arithmetic and logic micro-operations can be implemented individually. If we combine these two circuits along with shifting logic then we can have a possible 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 shift registers but sometimes it can be made as a part of overall ALU. More details on ALU can be studied from the further readings.

10.7 ARITHMETIC PROCESSORS

Arithmetic processors were needed in the older computer processors to perform arithmetic processing, especially the floating-point arithmetic, as those processors did not have the required logic circuits to directly handle such processing. They were also called co-processor, as they were used as an additional processor of some main processor. However, in this era of ultra-large-scale integration and beyond, all the fixed point and floating-point computational capabilities are built in the processor. The concept relating to arithmetic processor is explained below.

A typical processor 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 need not be included in the instruction set of a processor. In such systems, floating-point operations were 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 Integrated Circuit. Thus, a special purpose arithmetic processor, for performing only the arithmetic operations, can be constructed. This processor physically may be separate yet can be utilized by the processor to execute complex arithmetic instructions. Please note in the absence of arithmetic processors, these instructions may be executed using the slower software routines by the processor itself. Thus, this auxiliary processor enhances the speed of execution of programs having a lot of complex arithmetic computations.

An arithmetic processor also helps in reducing program complexity, as it provides a richer instruction set for a machine. Some of the instructions that can be assigned to arithmetic processors can be related to the addition, subtraction, multiplication, and division of floating-point numbers, exponentiation, logarithms and other trigonometric functions.

How can this arithmetic processor be connected to the CPU?

If an arithmetic processor is treated as one of the Input / Output or peripheral units then it is termed as a peripheral processor. The CPU sends data and instructions to the peripheral processor, which performs the required operations on the data and communicates the results back to the CPU. A peripheral processor has 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 type of connection is called loosely coupled.

If the arithmetic processor has a register and instruction set which can be considered an extension of the CPU registers and instruction set, then it is called a tightly coupled processor. Here the CPU reserves a special subset of code for arithmetic processor. In such a system the instructions meant for arithmetic processor are fetched by CPU and decoded jointly by CPU and the arithmetic processor, and finally executed by arithmetic processor. Thus, these processors can be considered a logical extension of the CPU. Such attached arithmetic processors were termed as co-processors.

These days floating point units are implemented as a part of the processor itself. More details on these can be found in further readings.

Check Your Progress 4

1. Explain the implementation of ALU.

.....
.....
.....

Instruction fetch: fetching the

2. What is an Arithmetic Processor?

.....
.....
.....

10.8 SUMMARY

This unit discusses the concept of instruction execution for a hypothetical machine with the help of micro-operations. It also describes very simplified view of implementation of micro-operations using combinational and sequential circuits. The idea is to give you a basic information about the implementation of a computer system based on its instruction set. The unit also discusses the concept of register transfer language for representing the micro-operations. The unit also defined the concept of Instruction Pipeline. The unit also discussed the hardware implementation of micro-operations. The unit shows a simple implementation of bus, which is the backbone for any register transfer operation. This is followed by a discussion on arithmetic circuit and micro-operation there on using full adder circuits. The logic micro-operation implementation has also been discussed. Finally, the unit also discussed the arithmetic processors.

You may refer to the further readings for more details on micro-operation concept and instruction cycle.

10.9 SOLUTIONS / ANSWERS

Check Your Progress 1

1. An address register is used to store memory address or can be used to compute memory address of instructions or operands.
2. To address 20 registers, you may require address field of length 5 bits, as $2^5 = 32$, thus, about 12 addresses are unused.
3. Independent set of conditional codes can help in parallel checking of conditions.
4. Yes. Several operating system allocate memory space for storing such information for later use.

Check Your Progress 2

1. Memory read operation requires the address of the location to be read. This address is first applied on the address BUS and the control unit enables memory read. Thus, the content of the addressed location is put in the data BUS. At the same time a data register is enabled to store the data on the data BUS. These operations can be represented as: Address BUS \leftarrow MAR ; DR \leftarrow Data BUS. The overall operation can be represented as: DR \leftarrow (MAR)
In memory write operation the memory address, where data is to be written, is applied on address BUS and the data that is to be stored/written is put on the data BUS. At the same time memory write operation is enabled by the control unit. This operation can be represented as: (MAR) \leftarrow DR
2. No, as multiplication operation will be implemented using addition and shift micro-operations.
3. (i) AND with R2 containing 0000 0000
(ii) Initially XOR of R1 with R1 will clear R1, then perform OR with R2 having value 11001000

Check Your Progress 3

1. Indirect cycle is needed, when indirect memory addressing is used by an instruction. It converts an indirect address to a direct address. If an instruction is using register addressing scheme, then indirect cycle may be required only if instruction is using register indirect addressing. The indirect cycle in this case would require simple register transfer micro-operation.
2. Fetch cycle is primarily used for fetching instruction that is to be executed from the memory. The present-day machines may use instruction cache, instruction prefetch buffer etc., yet this instruction is needed to be moved to the processor, which may execute it. Thus, the nature of the fetch cycle may change but it may be required.
3. Interrupt cycle is responsible for acknowledging an interrupt. When an interrupt occurs in a computer, it is acknowledged, when the instruction execution gets completed. The processor checks, if an interrupt has occurred, if it is then an interrupt cycle is performed.

Check Your Progress 4

1. The implementation of ALU can be done with the help of combinational and sequential circuits. The combinational circuits perform the computations. The results of these computations are stored in sequential circuits. The internal register bus is used for input and output to ALU. Arithmetic circuit like adders may be used to perform arithmetic micro-operations, logic gates may be used to perform logic micro-operations and shift registers may be used to perform shift operations.
2. Arithmetic processor performs arithmetic computation. These may be support processors to a computer.

UNIT 11 THE CONTROL UNIT

Structure	Page No.
11.0 Introduction	
11.1 Objectives	
11.2 The Control Unit	
11.3 The Hardwired Control	
11.4 Wilkes Control	
11.5 The Micro-Programmed Control	
11.6 The Micro-Instructions	
11.6.1 Types of Micro-Instructions	
11.6.2 Control Memory Organisation	
11.6.3 Micro-Instruction Formats	
11.7 The Execution of Micro-Program	
11.7.1 Micro-instruction Encoding	
11.7.2 Micro-instruction Sequencing	
11.8 Design Issues of control Unit	
11.9 Summary	
11.10 Solutions/ Answers	

11.0 INTRODUCTION

In the previous units of this block, instruction set architecture and concept of micro-operations were discussed. The micro-operations are used to provide execution environment for the instruction set in a computer system. The micro-operations are implemented as a part of the ALU and processor internal BUS.

The control unit is responsible for issuing the control signals, as per the micro-operations requirements of the instructions of a computer. In addition, it controls all the other units of a computer system. In this unit we are going to discuss the functions of a control unit and its implementation mechanisms like hardwired control unit and micro-programmed control unit. The micro-programmed control unit is popular amongst the Intel computer architecture due to its flexibility and legacy requirements. The hardwired control unit and other computer logic circuitry can be designed using Hardware Description Languages (HDL). The input to these languages are the electronic circuit structure and expected behaviour. Some of the specialized HDL programming languages are VHDL, Verilog etc. Discussion on these languages is beyond the scope of this course.

The unit discusses the basic requirements of a control unit, followed by the hardwired control unit and Wilkes control unit. Finally, we will discuss the micro-programmed control.

11.1 OBJECTIVES

After going through this unit, you will be able to:

- (a) define what is a control unit and its function;
- (b) describe a simple control unit organization;
- (c) define a hardwired control unit;
- (d) define the micro-programmed control unit;
- (e) define the term micro-instruction; and
- (f) identify types and formats of micro-instruction.

11.2 THE CONTROL UNIT

The processor of a computer consists of three basic components – a set of registers, the arithmetic logic unit and the control unit. All these components are connected through an internal BUS. The role of a control unit is to ensure that an instruction gets executed, as per the sequence of micro-operations for that instruction. This process also involves decoding the instruction that is to be executed. The control unit issues control signals so that these instructions are executed correctly. The basic responsibilities of the control unit are to control:

- a) the data exchange of the processor with the memory or I/O modules or interfaces.
- b) the internal operations in the processor, which may include:
 - moving data between registers using internal BUS or moving data from/to memory location using system BUS (register transfer micro-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 next sections.

Functional Requirements of Control Unit

Let us first try to define the functions, which a control unit must perform to cause instructions execution. But 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:

- (a) Basic components of the processor
- (b) Micro-operation this processor can perform

The processor of a computer consists of the following basic functional components:

- **The Arithmetic Logic Unit (ALU)**, which performs the basic arithmetic and logical operations.
- **Registers** which are used for information storage within the processor.
- **Internal Data Paths:** These paths are useful for moving the data between two registers or between a register and ALU.
- **External Data Paths:** The roles of these data paths are normally to link the processor registers with the memory or I/O interfaces. This role is normally fulfilled by the system bus.
- **The Control Unit:** This causes all the operations to happen in the processor.

The micro-operations performed by the processor can be classified as:

- Micro-operations for data transfer from register-register, register-memory, I/O-register etc.
- Micro-operations for performing arithmetic, logic and shift operations. These micro-operations involve use of registers for input and output.

The basic responsibility of the control unit lies in the fact that the control unit must be able to guide various components of processor 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 processor 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 the two tasks stated above. But how are these control signals generated? We will answer this question in later sections. First let us discuss a simple structure of control unit.

Structure of Control Unit

A control unit has a set of input values based on 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 control unit is shown in Figure 11.1.

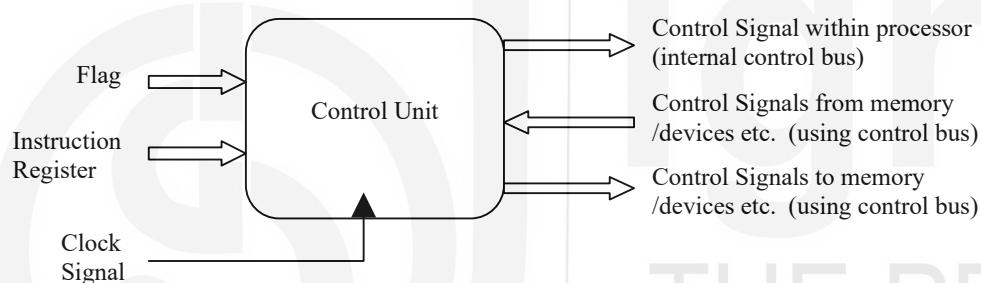


Figure 11.1: A General Model of Control Unit

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 sequence. 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 termed as processor cycle time or the clock cycle time in some machines.
- **The Instruction Register:** It contains the operation code (opcode) and addressing mode bits of the instruction. It helps in determining various instruction cycles, as given in Unit 10, that are to be performed for a specific instruction and hence determines the related micro-operations.
- **Flags:** Flags represent the conditional codes that can be used in decision making. Flags are set by the ALU operations. For example, a zero flag, if set, communicates to the control unit that the result of last ALU operations was zero. Thus, if processor was executing the ISZ instruction (skip the next instruction if zero flag is set), the next instruction should be skipped. This action is initiated by the control unit, which would increment PC by one program instruction length, thus skipping the next instruction.

- **Control Signals from Control Bus:** Some of the control signals are provided to the control unit through the control bus. These signals are issued from outside the processor. Some of these signals are interrupt signals and acknowledgement signals.

Based on the input signals the control unit activates certain output control signals, which in turn are responsible for execution of an instruction. These output control signals are:

- **Control signals, which are required within the processor:** These control signals cause two types of micro-operations, viz., for data transfer from one register to another; and for performing an arithmetic, logic and shift operation using ALU.
- **Control signals to control bus:** These control signals transfer data from or to processor register to or from memory or I/O interface. These control signals are issued on the control bus to activate a data path on the data / address bus etc.

A control unit must know how all the instructions would be executed. It should also know about the nature of the results and the indication of possible errors. All this is achieved with the help of flags, opcodes, clock and control signals.

A control unit contains a clock portion that provides clock-pulses. This clock signal is used for determining the timing sequence of the micro-operations. In general, the timing signals from control unit are kept sufficiently long to accommodate the propagational delays of signals within the processor along various data paths. Since 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 further readings.

How are these control signals applied to realise micro-operations? *The control signals are applied directly as the binary inputs to the logic gates of the logic circuits that are responsible for implementing micro-operations.* All these inputs are the control signals, which are applied to select a circuit (for example, select or enable input) or a path (for example, multiplexers) or any other operation in the logic circuits.

11.3 THE HARDWIRED CONTROL

In the last section, we have discussed the control unit in terms of its inputs, output and functions. A variety of techniques have been used to organize a control unit. Most of them fall into two major categories:

1. Hardwired control organization
2. Microprogrammed control organization.

In the hardwired organization, the control unit is designed as a combinational circuit or a sequential circuit. In other words, control units are implemented using logic gates, flip-flops, counter circuits, decoder circuit, select and enable input to various logic circuits etc. Since the hardwired control unit is made up of fast logic circuits, it can be optimised for fast operations.

The block diagram of a hardwired control unit is shown in Figure 11.2. The major input to the circuit are instruction register, the clock, and the flags. The control unit uses the opcode of instruction stored in the IR register to perform different actions. The opcode can be used to decode a typical sequence of control signals that are

responsible for execution of that instruction. Selecting separate instruction line for each instruction simplifies the control logic. This control line selection can be performed by a decoder. Decoder are explained in the Unit 3, it uses n input line to select one output line out of 2^n lines.

The clocksignal is input to sequencing logic andforms one of the input of control unit. The sequencing logic issues a repetitive sequence of pulses for the execution of micro-operation(s). These timing signals control the sequence of execution of instruction and determine what control signal needs to be applied at what time for instruction execution. Please note a typical example, of timing sequence for execution of micro-operations of different sub-cyclesof an instruction are given in Unit 10

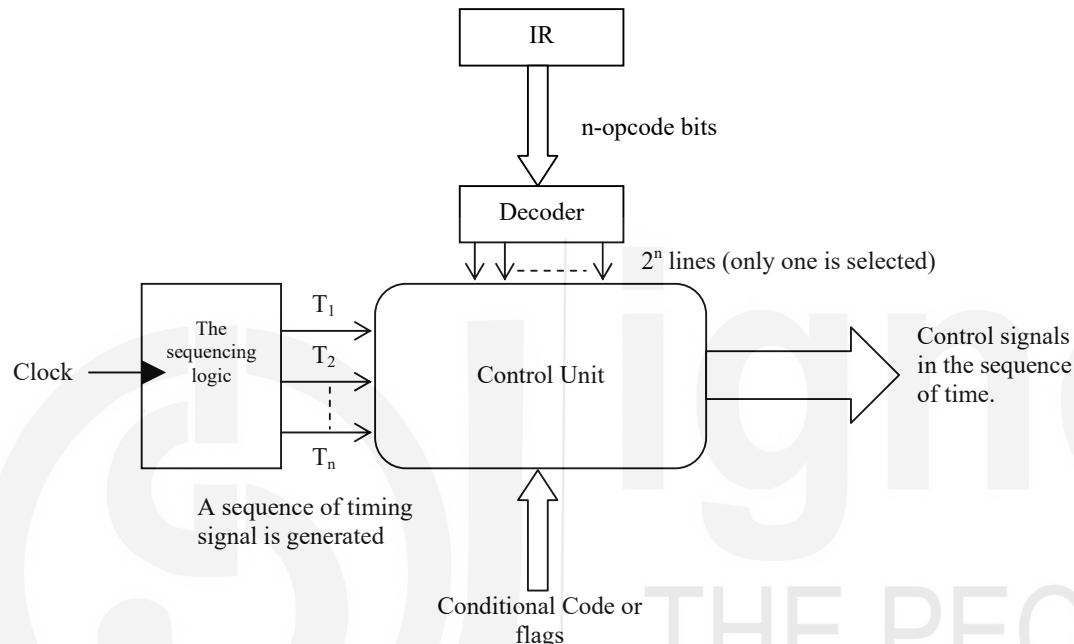


Figure 11.2: Block Diagram of Hardwired Control Unit

Check Your Progress 1

- What are the inputs to control unit?

.....

- How does a control unit control the instruction cycle?

.....

- What is a hardwired control unit?

.....

11.4 WILKES CONTROL

Prof. M. V. Wilkes of the Cambridge University Mathematical Laboratory coined the term microprogramming in 1951. He provided a systematic alternative procedure for designing the control unit of a digital computer. During instruction executing a machine instruction, a sequence of data transformations due to arithmetic, logic and shift micro-operations and transfer of information from one register in the processor due to register transfer micro-operations take place. **Because of the analogy between the execution of individual steps in a machine instruction to the execution of the individual instruction in a program, Wilkes introduced the concept of microprogramming.** The Wilkes control unit replaces the sequential and combinational circuits of hardwired control unit by a simple control unit in conjunction with a storage unit that stores the sequence of steps of instruction that is a micro-program.

In Wilkes microinstruction has two major components:

- Control field which indicates the control lines which are to be activated and
- Address field, which provides the address of the next microinstruction to be executed.

Figure 11.3 is an example of Wilkes control unit design.

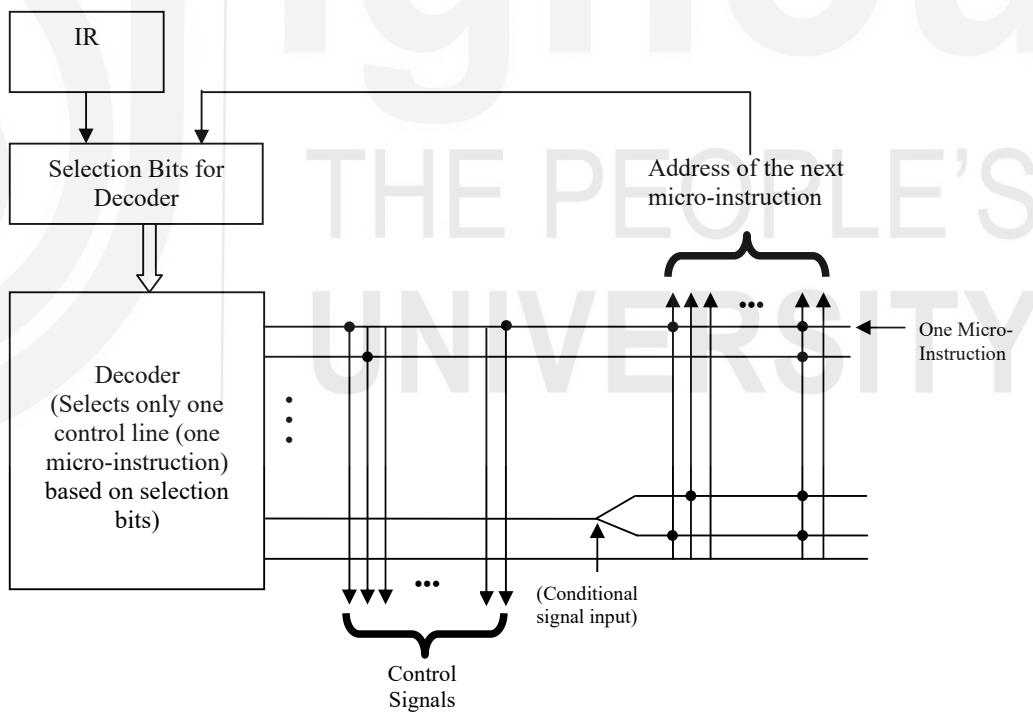


Figure 11.3: Wilkes Control Unit

The control memory in Wilkes control was organized, as a PLA's like matrix made of diodes. Each horizontal line on this matrix consists of two components, viz. the control signals and the address of the next micro-instruction. The next micro-instruction register stores the address of the next micro-instruction to be loaded. Please note that this register should be loaded at the falling edge of clock, that is once the previous micro-instruction completes its execution. The next micro-instruction to be executed is either specified by IR, after the instruction has been decoded or by the address of the next micro-instruction as specified in the micro-instruction itself. The

register input passes through the address decoder and the decoded line of the control matrix is selected for generating the control signals for the processor. The Wilkes control unit also provides handling of conditions. For example, a condition like zero flag can be attached to the conditional signal input, which determines the micro-instruction to be executed next. More details on the Wilkes control unit may be studied from the further readings.

11.5 THE MICRO-PROGRAMMED CONTROL

An alternative to a hardwired control unit is a micro-programmed control unit. Wilke's control unit is one of the examples of micro-program control unit. A micro-program is also called firmware (midway between the hardware and the software). It consists of:

- (a) One or more micro-operations to be executed; and
- (b) The information about the micro-instruction to be executed next.

The general configuration of a micro-programmed control unit is shown in Figure 11.4.

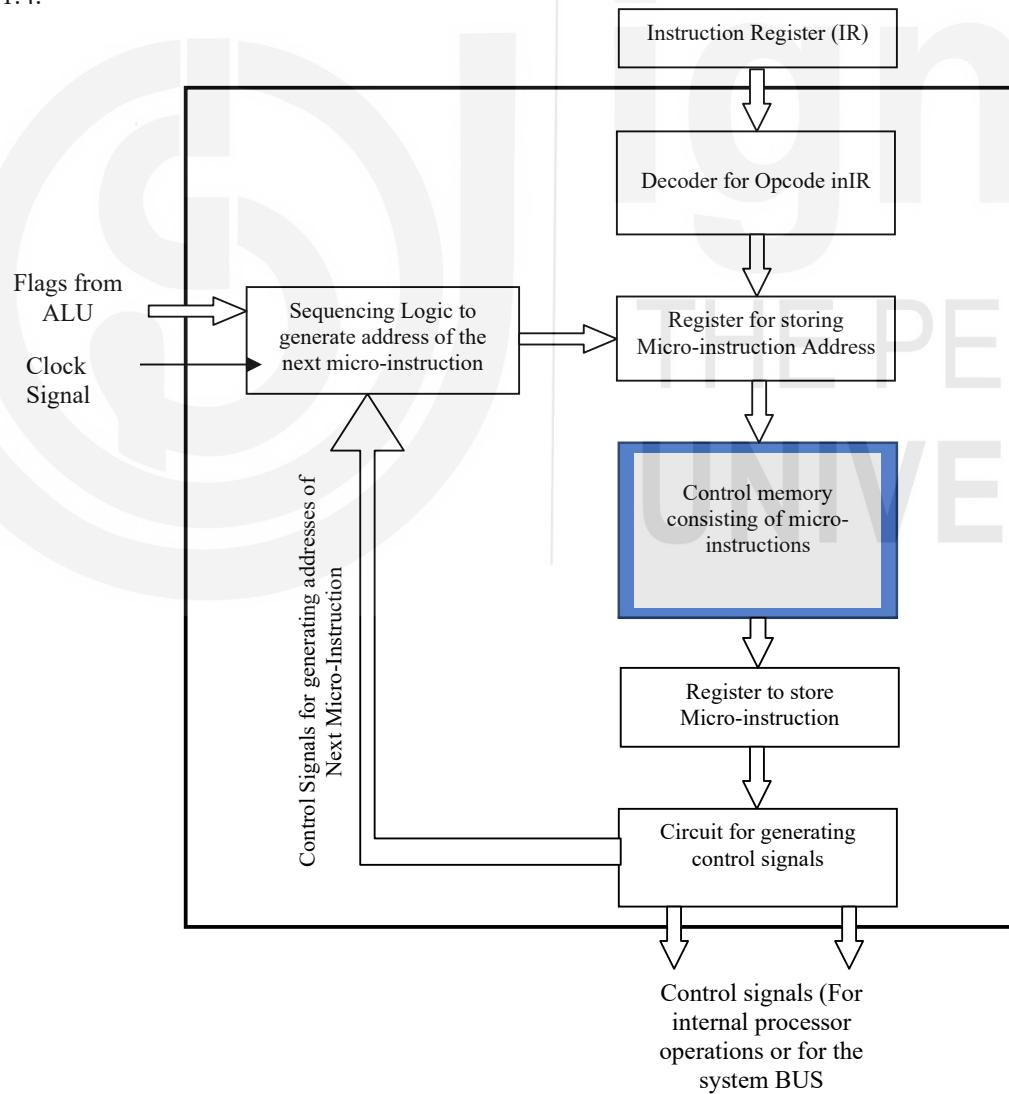


Figure 11.4: Operation of Micro-Programmed Control Unit

The control memory of the micro-programmed control unit stores the micro-instructions. The micro-instruction address register stores the address of the micro-instruction, which should be used to generate the control signal and the address of next micro-instruction. The micro-instruction register stores the last read micro-instruction, which is used to generate the control signals for performing micro-operations and control signals for generating address of the next micro-instruction. . A micro-instruction execution primarily involves the generation of desired control signals and signals used to determine the next micro-instruction to be executed. The sequencing logic of this control unit loads the micro-instruction address register. It also issues a read command to control memory, which stores the micro-instrucitons. The following functions are performed by the micro-programmed control unit:

1. The sequence logic unit specifies the address of the control memory word which contains the micro-instruction that is to be read, in the micro-instruction address register of the Control Memory. It also issues the READ signal to the control memory, so that the desired micro-instruction can be read.
2. The desired control memory word containing the desired micro-instruction is read into the micro-instruction register.
3. The micro-instruction register forms the input to the logic circuit that generates the control signals based on the current micro-instruction. Further, this circuitry also generates the control signals that can be used by sequencing logic to generate the address of micro-instructionin the control memory that is to be executed next.
4. The sequencing logic uses the control signals, as stated above, and flag register to computethe address of the micro-instruction that is to be executed next.

As we have discussed earlier, the execute cycle steps of micro-operations are different for all instructions in addition the addressing mode may be different. All such information generally is stored is coded in the instruction, which is stored in the instruction Register (IR). The IR input to Micro-Instruction Address Register for Control Memory is used for determining the micro-instruction, which performs the execute cycle of the instruction. The decoder after IR uses the IR register to generate the address of the first micro-instruction in control memory for the specified instruction in IR (Refer to Figure 11.4).

■ Check Your Progress 2

1. What is firmware? How is it different from software?
-

2. State True or False

(a) A micro-instruction can initiate only one micro-operation at a time.

(b) A control word is equal to a memory word.

(c) Micro-programmed control is faster than hardwired control.

(d) Wilkes's control does not provide a branching micro-instruction.

3. What will be the control signals and address of the next micro-instruction in the Wilkes control example of Figure 11.3, if the entry address for a machine instruction selects the branching control lineand the conditional bit value for branch is true (assume that out of the two branching lines the bottom line is selected when condition is true)?
-
-
-

11.6 THE MICRO-INSTRUCTIONS

A micro-instruction, as defined earlier, is an instruction of a micro-program. It specifies one or more micro-operations, which can be executed simultaneously. On executing a micro-instruction, a set of control signals are generated which in turn cause the desired micro-operation to happen.

11.6.1 Types of Micro-instructions

In general, the micro-instruction can be categorised into two general types. These are branching and non-branching. After execution of a non-branching micro-instruction the next micro-instruction is the one following the current micro-instruction. However, the sequences of micro-instructions are relatively small and last only for 3 or 4 micro-instructions.

A conditional branching micro-instruction tests conditional variable, or a flag generated by an ALU operation. Normally, the branch address is contained in the micro-instruction itself.

11.6.2 Control Memory Organisation

The next important question about the micro-instruction is: how are they organized in the control memory? One of the simplest ways to organize control memory is to arrange micro-instructions for various sub cycles of the machine instruction in the memory. The Figure 11.5 shows such an organisation.

00h	Micro-instructions for fetch cycle	
01h	...	
02h	...	
03h	Jump to Indirect or Execute Cycle using the addressing mode bits of the instruction	Fetch cycle
04h	Micro-instruction for Indirect cycle	Indirect cycle
05h	...	
06h	...	
07h	Jump to Execute cycle	
08h	Micro-instruction for interrupt initiation	Interrupt cycle
09h	...	
0Ah	...	
0Bh	Jump to fetch cycle	
0Ch	Compute address of micro-instruction based on opcode	Execute cycle
0Dh	...	
0Eh	...	
0Fh	Jump to the micro-instruction of opcode	
10h	Micro-instructions for opcode 0 (Let us say LOAD)	Micro-instructions for opcode 0
11h	...	
12h	...	
13h	Jump to fetch or interrupt cycle	
14h	Micro-instruction for opcode 1 (Let us say ADD)	Micro-instructions for opcode 1
15h	...	
16h	...	
17h	Jump to fetch or interrupt cycle	
...
F8h	Micro-instruction for last opcode (Let us say ISZ)	Micro-instructions for opcode ISZ
F9h	...	
FAh	...	
FBh	...	
FCh	...	
FDh	...	
FEh	...	
FFh	Jump to fetch or interrupt cycle	

Figure 11.5: An example of Control Memory Organisation

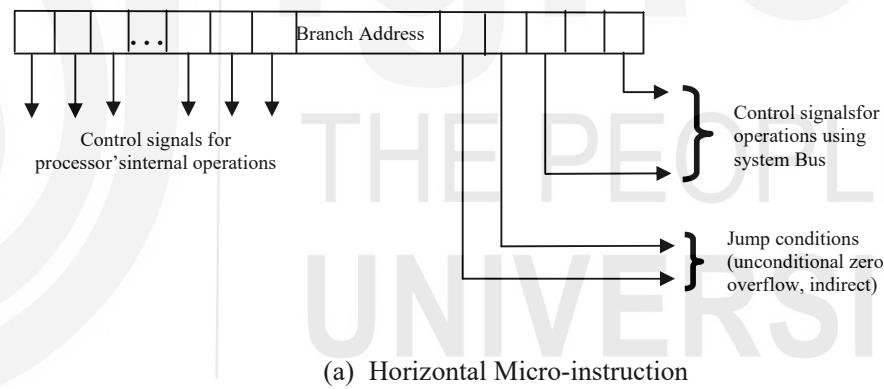
Let us give an example of control memory organization. Let us take a machine instruction: Branch on zero. This instruction causes a branch to a specified main memory address in case the result of the last ALU operation is zero, that is, the zero flag is set. The pseudocode of the micro-program for this instruction may be written as:

Test "Zero flag"; if SET branch to micro-code at label ZERO
 Unconditional branch to micro-code of label NON-ZERO
ZERO: Microcode of sequence of micro-operations required to be executed to replace the PC by the effective address of the instruction operand.
NON-ZERO: *Branch to Interrupt or Fetch cycle.*

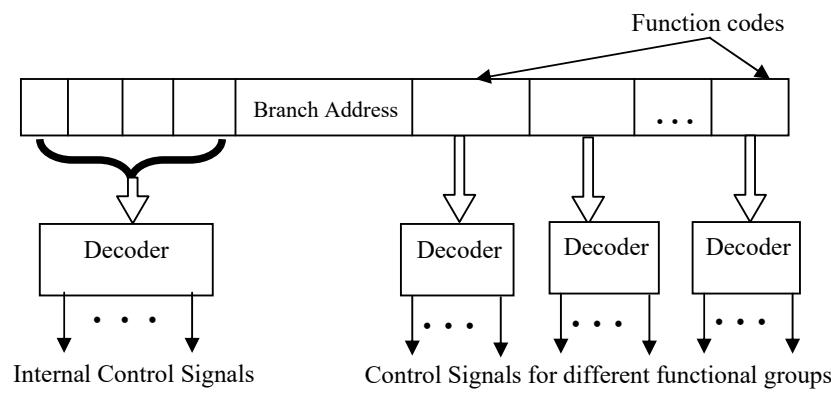
Please note that in case the Zero flag is not SET, then no operation is needed, as next instruction in sequence is to be executed, whose address is already in PC. Thus, next instruction should be fetched.

11.6.3 Micro-instruction Formats

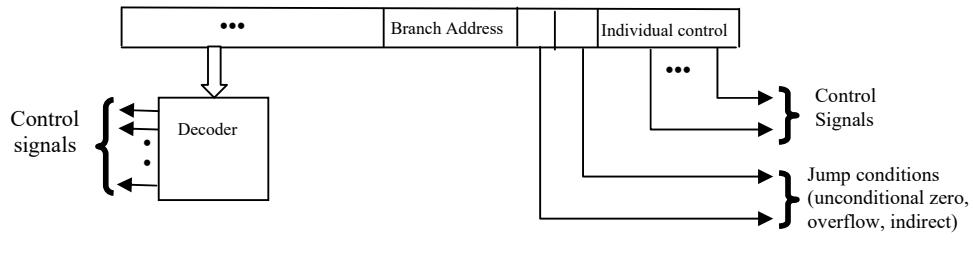
Now let us focus on the format of a micro-instruction. The two widely used formats used for micro-instructions are horizontal and vertical. In the horizontal micro-instruction, each bit of the micro-instruction represents a control signal, which directly controls a single bus line or sometimes a gate in the machine. However, the length of such a micro-instruction may be hundreds of bits. A typical horizontal micro-instruction with its related fields is shown in Figure 11(a).



(a) Horizontal Micro-instruction



(b) Vertical Micro-instruction



(c) A Realistic Micro-instruction
Figure 11.6: Micro-instruction Formats

In a vertical micro-instruction, many similar control signals can be encoded into a few micro-instruction bits. For example, for 16 ALU operations, which may require 16 individual control bits in horizontal micro-instruction, only 4 encoded bits are needed in vertical micro-instruction. Similarly, in a vertical micro-instruction only 3 bits are needed to select one of the eight registers. However, these encoded bits need to be passed from the respective decoders to get the individual control signals. This is shown in Figure 11.6(b).

In general, a horizontal control unit is faster, yet requires wider instruction words, whereas vertical control units, although require a decoder, are shorter in length. Most of the systems use neither purely horizontal nor purely vertical micro-instructions Figure 11.6(c).

11.7 THE EXECUTION OF MICRO-PROGRAM

The micro-instruction cycle can consist of two basic cycles: the fetch and the execute. Here, in the fetch cycle the address of the micro-instruction is generated and this micro-instruction is put in a register used for the address of a micro-instruction for execution. The execution of a micro-instruction simply means generation of control signals. These control signals may drive the processor (internal control signals) or the system bus. The format of micro-instruction and its contents determine the complexity of a logic module, which executes a micro-instruction. These logic module depends on the encoding of micro-instructions, which is discussed next.

11.7.1 Micro-instruction Encoding

One of the key features incorporated in a micro-instruction is the encoding of micro-instructions. What is encoding of micro-instruction? For answering this question let us recall the Wilkes control unit. In Wilkes control unit, each bit of information either generates a control signal or form a bit of next instruction address. Now, let us assume that a machine needs N total number of control signals. If you are using the Wilkes scheme you require N bits for the control signals, one for each control signal in the control unit. In addition, Wilkes control unit also stores the address of the next micro-instruction using address bits.

Since we are dealing with binary control signals, therefore, a ' N ' bit micro-instruction can represent 2^N combinations of control signals.

The question is 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 a source and a destination at the same time. Thus, such a combination of control signals is redundant.
 3. You can provide only one pattern of control signals at a time to ALU, making some of the combinations redundant.
 4. You can provide only one pattern of control signals at a time to the external control bus also.

Therefore, you do not need 2^N combinations. Suppose you only need 2^K (where $K < N$) combinations, then you need only K encoded bits instead of N control signals. The K bit micro-instruction is an extreme encoded micro-instruction. Let us touch upon the characteristics of the extreme encoded and unencoded micro-instructions:

Unencoded micro-instructions

- One bit is needed for each control signal; therefore, the number of bits required in a micro-instruction is high.
- It presents a detailed hardware view, as control signal need can be determined.
- Since each of the control signals can be controlled individually, therefore these micro-instructions are difficult to program. However, concurrency can be exploited easily.
- Almost no control logic is needed to decode the instruction as there is one to one mapping of control signals to a bit of micro-instruction. Thus, execution of micro-instruction and hence the micro-program is faster.
- The unencoded micro-instruction aims at optimising the performance of a machine.

Highly Encoded micro-instructions

- The encoded bits needed in micro-instructions are smaller in size than that of unencoded micro-instructions.
- It provided an aggregated view that is a higher view of the processor as only an encoded sequence can be used for micro-programming.
- The encoding helps in reduction in programming burden; however, the concurrency may not be exploited to the fullest.
- Complex control logic is needed, as decoding is a must. Thus, the execution of a micro-instruction can have propagation delay through gates. Therefore, the execution of micro-program takes longer time than that of an unencoded micro-instruction.
- The highly encoded micro-instructions are aimed at optimizing programming effort.

In general, the micro-programmed control unit designs are neither completely unencoded nor highly encoded. They are slightly coded. This reduces the width of control memory and micro-programming efforts. As shown in Figure 11.7, some of the bits of micro-instructions are unencoded, therefore, can be used to generate the control signals directly. Some of the control bits are organised as fields and can be directly used as input to decoder to generate control signals. Further, a combination of decoded control signals can be passed through a second decoder to generate control signals. These decoding operations are shown in Figure 11.7.

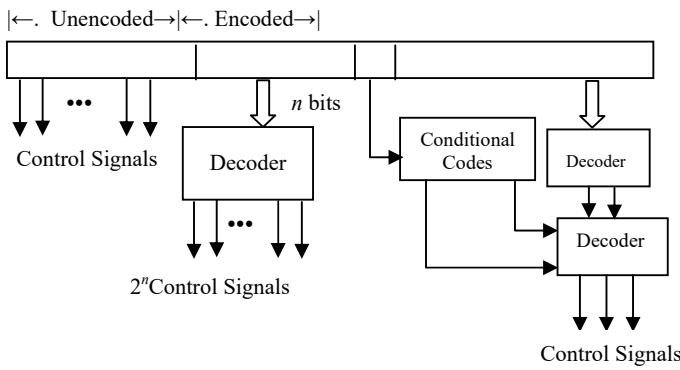


Figure 11.7: Decoding of Micro-instructions

11.7.2 Micro-instruction Sequencing

Another aspect of micro-instruction execution is the micro-instruction sequencing that involves address calculation of the next micro-instruction. In general, the next micro-instruction can be one of the following (refer Figure 11.5):

- Next micro-instruction in sequence
- Calculated on the basis of opcode
- Branch address (conditional or unconditional).

Next instruction in sequence: Figure 11.5 shows one example of control memory. This control organisation has micro-instructions for fetch, indirect and interrupt cycles followed by the execute cycle. You may recall the instruction cycle, as given in Unit 10, the fetchinstruction cycle consists of the following sequence of micro-operations:

T1: MAR \leftarrow PC
 T2: DR \leftarrow (MAR)
 T3: PC \leftarrow PC +1; IR \leftarrow DR

One micro-instruction can be created for each timing sequence. For example, for the stated sequence of micro-operations, three micro-instructions, one each for timing T1, T2 and T3 would be created. These micro-instructions would be stored at address 00h, 01h and 02h. The micro-instruction at 03h would be a conditional branch instruction to the indirect or execute cycle. Please also note that at time T3, two micro-operations are to be performed in parallel. Thus, micro-instruction at 02h should generate control signals, which result in the related units of the processor to perform the increment operation on PC and transfer of DR to IR simultaneously.

The micro-instruction at 00h to 03h are to be executed in a sequence to perform the desired operation of instruction fetch.

Branch address (conditional or unconditional): You may please note that the last micro-instruction for instruction fetch is the conditional branch instruction. Please also note that in Figure 11.5, the indirect cycle starts at an address 04h and the execution cycle starts at 0Ch. Thus, the micro-instruction at address 03h would be a conditional branch instruction, having the condition, if the indirect bit is set or not. This conditional branch would be taken to address 0Ch (the start of execution cycle) in case the indirect bit is CLEAR. Thus, if indirect bit is SET, then the next micro-instruction in sequence would be executed, which will be the starting micro-instruction of the indirect cycle that will convert the indirect operand to direct operand. Please also note in the Figure 11.5, that the last micro-instruction of the indirect cycle is an unconditional jump instruction to the execute cycle.

Calculated on the basis of opcode: The opcode of the Instruction Register is used to decode the operation that is to be performed on the operands. The control unit supports this decode operation. In the case of micro-programmed control unit, this opcode can be used to compute the address of the first micro-instruction to be executed to perform the operation. In Figure 11.5, the execute cycle contains the

micro-instructions that perform jump to the micro-instruction address of the desired operation.

We will explain it with the help of an example, assume that in Figure 11.5, the micro-instructions related to opcodes start from micro-instruction address 00h instead of 10h and the micro-instructions of fetch, indirect, interrupt and execute cycles starts at micro-instruction address F0h, F4h, F8h and FCh respectively. Further, we assume that operation of each opcode is performed using just four micro-instructions. Since the control memory has addresses from 00h to FFh, out of which 00h to EFh (a total of F0h) addresses are for storing micro-instructions of various opcodes. Therefore, this control memory can contain micro-instructions for $F0h/4h = 3Ch$ opcodes. Thus, the possible opcodes for such a machine would be 0000000_2 to 00111011_2 , or 000000_2 to 111011_2 . How these opcodes would be mapped to the related micro-instruction start address. The following table shows these mapping:

Opcode	Starting address of related Micro-instructions	
Binary	Binary	Hexadecimal
0000 00	0000 0000	00
0000 01	0000 0100	04
0000 10	0000 1000	08
0000 11	0000 1000	0C
0001 00	0001 0000	10
0001 01	0001 0100	14
0001 10	0001 1000	18
0001 11	0001 1000	1C
...	...	
1001 00	1001 0000	90
1001 00	1001 0100	94
...	...	
1110 00	1110 0000	E0
1110 01	1110 0100	E4
1110 10	1110 1000	E8
1110 11	1110 1000	EC

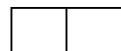
Figure 11.8: Mapping of opcode to micro-instruction address

The interesting part of this example is the mapping from the opcode to the micro-instruction address. In Figure 11.8, an opcode is of 6-bit length. To map an opcode to the first related micro-instruction, you just need to append two 0 bits at the least significant position. For example, an opcode 1001 01 will be mapped to a micro-instruction address 1001 0100 or 94h.

Please note different kind of control memory and opcode organisation will make this computational logic a complex one. In any case, you can design the related logic circuit for calculating the micro-instruction address for a given opcode.

You must refer to further readings for more detailed information on Micro-programmed Control Unit Design.

Check Your Progress 3



1. State True or False

- a) A branch micro-instruction can have only an unconditional jump.
- b) Control store stores opcode-based micro-programs.
- c) A true horizontal micro-instruction requires one bit for every control signal.

- d) A decoder is needed to find a branch address in the vertical micro-instruction.
 - e) One of the responsibilities of sequencing logic (Refer Figure 11.4) is to cause reading of micro-instruction addressed by a micro-program address register.
 - f) Status bits supplied from ALU to sequencing logic have no role to play with the sequencing of micro-instruction.
2. What are the possibilities for the next instruction address?

.....
.....
.....
.....

3. How many address fields are there in Wilkes Control Unit?

.....
.....
.....

4. Compare and contrast unencoded and highly encoded micro-instructions.

.....
.....
.....

11.8 SUMMARY

In this unit we have discussed the organization of control units. Hardwired, Wilkes and micro-programmed control units are also discussed. The key to such control units are micro-instruction, which can be briefly (that is types and formats) described in this unit. The function of a micro-programmed unit, that is, micro-programmed execution, has also been discussed. The control unit is the key for the optimised performance of a computer. The information given in this unit can be further appended by going through further readings.

11.9 SOLUTIONS/ANSWERS

Check Your Progress 1

1. IR, Timing Signal, Flags Register
2. The control unit issues control signals that cause execution of micro-operations in a pre-determined sequence. This enables execution sequence of an instruction.
3. A logic circuit-based implementation of control unit.

Check Your Progress 2

1. Firmware is basically micro-programs, which are used in a micro-programmed control unit. Firmware are more difficult to write than software.

2. (a) False (b) False (C) False (d) False
3. Please check the Figure 11.3 from left to right and select the bottom branch line.
The control signals would be 000...00
Address of next micro-instruction would be: 100...10

Check Your Progress 3

1. (a) False (b) False (c) True (d) False (e) True (f) False.
2. The address of the next micro-instruction can be one of the following:
 - the address of the next micro-instruction in sequence.
 - determined by opcode using mapping or any other method.
 - branch address supplied on the internal address bus.
3. Wilkes control typically has one address field. However, for a conditional branching micro-instruction, it contains two addresses. The Wilkes control, in fact, is a hardware representation of a micro-programmed control unit.
- 4.

Unencoded Micro instructions	Highly encoded
<ul style="list-style-type: none">• Large number of bits• Difficult to program• No decoding logic• Optimizes machine performances• Detailed hardware view	<ul style="list-style-type: none">Relatively less bitsEasy to programNeed decoding logicOptimizes programming effortAggregated view

THE PEOPLE'S
UNIVERSITY

UNIT 12 REDUCED INSTRUCTION SET COMPUTER ARCHITECTURE

Structure	Page No.
12.0 Introduction	
12.1 Objectives	
12.2 Introduction to RISC	
12.2.1 Importance of RISC Processors	
12.2.2 Reasons for Increased Complexity	
12.2.3 High Level Language Program Characteristics	
12.3 RISC Architecture	
12.4 The Use of Large Register File	
12.5 Comments on RISC	
12.6 RISC Pipelining	
12.7 Summary	
12.8 Solutions/ Answers	

12.0 INTRODUCTION

In the previous units, we have discussed the instruction set, register organization and pipelining, and control unit organization. The trend of those years was to have a large instruction set, a large number of addressing modes and about 16 –32 registers.

However, there existed a pool of thought which was in favour of having simplicity in instruction set. This logic was mainly based on the type of the programs, which were being written for various machines. This led to the development of a new type of computers called Reduced Instruction Set Computer (RISC). In this unit, we will discuss about the RISC machines. Our emphasis will be on discussing the basic principles of RISC and its pipeline. You may refer to further readings for more details on this architecture.

12.1 OBJECTIVES

After going through this unit, you should be able to:

- define the reason of increasing complexity of instructions;
 - explain the reasons for developing RISC;
 - define the basic principles of RISC;
 - describe the importance of having large register file;
 - describe RISC pipelining.
-

12.2 INTRODUCTION TO RISC

Reduced Instruction set computer architectures, were initially designed to reduce the complexity of the instruction sets, which included very large number of instructions. The purpose was to design an instruction set that could be designed for better performance with no additional cost of the processor. In fact, the aim of computer processor chip architects has been to design processor chips, which are more powerful than their predecessors, yet are not expensive. Thus, the processor chip designer would like to:

- Optimise the hardware manufacturing cost.
- Optimises the cost for programming scalable/ portable architectures that require low costs for debugging the initial hardware and subsequent programs.

If you review the history of computer families, you will find that the most common architectural change is the trend towards even more complex machines.

12.2.1 Importance of RISC Processors

Reduced Instruction Set Computers recognise a relatively limited number of instructions in comparison to the complex instruction set computers. In addition, a RISC processor also has a limited number of addressing modes with most of the instruction using the register operands. Thus, the instructions of RISC processor are simpler, therefore, can be executed faster. Another advantage is that RISC chips are cheaper to design and produce.

In general, an instruction on a RISC machine can be executed in one processor cycle, as RISC uses instruction pipeline. As discussed in Unit 11, an instruction pipeline enhances the speed of instruction execution. In addition, the control unit of the RISC processor is simpler and smaller than Complex Instruction Set Computers (CISC). This saved space can be used for building additional registers in the processor. This further enhances the processing capabilities of the RISC processor. Most RISC processor uses the register operands, this necessitates that the memory to register “LOAD” and “STORE” are created as separate independent instructions, which use memory reference operation..

Various RISC Processors

RISC has fewer design bugs; its simple instructions reduce design time. Thus, because of all the above important reasons RISC processors have become popular. Some of the uses of RISC processors are in the mobile processors, desktops, workstation and embedded devices. One of the open instruction set architecture for RISC instruction set is RISC V. This architecture uses the principles of RISC, which are discussed in this unit. For more details, you may refer to the further readings.

12.2.2 Reasons for Increased Complexity

The complexity of computer chips kept growing with the advancement in technology. In this section, we discuss the reasons for increased complexity of instruction set of computers.

Speed of Memory Versus Speed of CPU

In the past, there existed a large gap between the speed of a processor and memory. Thus, an execution of instruction using a program, for example floating point addition, may have to follow a lengthy instruction sequence. The question is; if we make it a machine instruction then only one instruction fetch will be required and rest will be done with control unit sequences. Thus, a “higher level” instruction can be added to machines in an attempt to improve performance.

However, this assumption is not very valid in the present era where the Main memory is supported with Cache technology. Cache memories have reduced the difference between the CPU and the memory speed and, therefore, an instruction execution through a subroutine step may not be that difficult. Let us explain it with the help of an example:

Suppose the floating-point operation ADD A, B requires the following steps (assuming the machine does not have floating point registers) and the registers being used for exponent are E1, E2, and EO (output); for mantissa M1, M2 and MO (output):

- Load the exponent of A in E1

- Load the mantissa of A in M1
- Load the exponent of B in E2
- Load the mantissa of B in M2
- Compare E1 and E2
 - If $E1 = E2$ then $MO \leftarrow M1 + M2$ and $EO \leftarrow E1$
 - Normalise MO and adjust EO
 - Result will be contained in MO, E1
 - else if $E1 < E2$ then find the difference = $E2 - E1$
 - Shift Right M1, by difference
 - $MO \leftarrow M1 + M2$ and $EO \leftarrow E2$
 - Normalise MO and adjust EO
 - Result is contained in MO, EO
 - else $E2 < E1$, if so find the difference = $E1 - E2$
 - Shift Right M2 by difference above
 - $MO \leftarrow M1 + M2$ and $EO \leftarrow E1$
 - Normalise MO and adjust E1 into EO
 - Result is contained in MO, EO
- Move the mantissa and exponent of the results to A
- Checks overflow underflow if any.

If all these steps are coded as one machine instruction, then this simple instruction will require several instruction cycles. If this instruction is made a part of the machine instruction set architecture as an instruction: ADDF A, B (Add floating point numbers A and B and store results in A), then it will just be a single machine instruction. All the above steps required will then be coded with the help of micro-operations in the form of Control Unit Micro-Program. Thus, just one instruction cycle (although a long one) may be needed. This cycle will require just one instruction fetch. Whereas in the program memory instructions will be fetched.

However, faster cache memory for Instruction and data stored in registers can create an almost similar instruction execution environment. Pipelining can further enhance such speed. Thus, creating an instruction as above may not result in faster execution.

Microcode and VLSI Technology

It is considered that the control unit of a computer be constructed using two ways; create micro-program that execute micro-instructions or build circuits for each instruction execution. Micro-programmed control allows the implementation of complex architectures more cost effective than hardwired control as the cost to expand an instruction set is very small, only a few more micro-instructions for the control store. Thus, it may be reasoned that moving subroutines like string editing, integer to floating point number conversion and mathematical evaluations such as polynomial evaluation to control unit micro-program is more cost effective. However, such a mechanism may result in slightly slower execution of commonly used instructions.

Code Density and Smaller Faster Programs

The memory was very expensive in the older computer. Thus, there was a need of less memory utilization, that is, it was cost effective to have smaller compact programs. Thus, more complex instruction sets were designed, so that programs are smaller. However, increased complexity of instruction sets had resulted in instruction sets and addressing modes requiring more bits to represent them. It is stated that the code compaction is important, but the cost of 10 percent more memory is often far less than the cost of reducing code by 10 percent out of the CPU architecture innovations.

The smaller programs are advantageous because they require smaller RAM space. Fewer instructions mean fewer instruction bytes to be fetched. But this does not ensure that program written for CISC machines would be smaller in size than that of

programs written for RISC machine. It may be possible that a CISC program is smaller in number of instructions, yet the overall size, in terms of number of bytes, may not be small. This may result from the reason that in RISC we use register addressing and less instruction, which require fewer bits in general. In addition, you may please note that even the compliers on CISC machine favours simpler instructions. Let us explain this with the help of the following example:

Assume a CISC machine has a 4GB byte addressable RAM (2^{32}) and 32 registers (2^5). A machine instruction consists of two operands, one of which should be a register operand. Almost similar RISC machine having the same size of RAM and active registers. Further, the CISC machine uses 16 bit to represent opcode and addressing modes and the RISC machine uses 8 bit to represent opcode and addressing modes. Figure 12.1(a) shows ADD and MOV instructions for the CISC machine. On the other hand, RISC machine would have at least two instruction formats (First to load the data from RAM to register or store the register to memory; or addition operation on register. Figure 12.2 (b) shows these two instruction formats for RISC machine.

	16	5	32	
ADD	R1		A	; R1 \leftarrow R1 + [A]
ADD	A		R1	; [A] \leftarrow R1 + [A]
MOV	R1		A	; R1 \leftarrow [A]
MOV	A		R1	; [A] \leftarrow R1

(a) Sample instructions of CISC

	8	5	32	
LDA	R1		A	; R1 \leftarrow [A]
STR	R1		A	; [A] \leftarrow R1

	8	5	5	
ADD	R1	R2		; R1 \leftarrow R1 + R2

(b) A sample Load, Store and ADD instruction of RISC

Figure 12.1: Sample machine instructions

Figure 12.1 shows the instructions for a CISC and RISC machine. The size of CISC ADD instruction is 53 bits, therefore, it will be stored in 7 bytes or 56 bits. The load (LDA) and store (STR) instructions of RISC machines are 45 bits long, so would be stored in 6 bytes. In addition, in RISC machine the ADD instruction would use 18 bits or 3 bytes. Consider the following sequence of operations on these two machines:

$$C = A + B$$

	16	5	32	
MOV	R1		A	; R1 \leftarrow [A]
ADD	R1		B	; R1 \leftarrow R1 + [A]
MOV	C		R1	; [C] \leftarrow R1

Program segment size = $7 \times 3 = 21$ Bytes

Size in bits = $53 \times 3 = 159$ bits

(a) Segment to compute $C = A + B$ using sample CISC ISA

	8	5	32	
LDA	R1		A	; R1 \leftarrow [A]
LDA	R2		B	; R2 \leftarrow [B]
ADD	R1	R2		; R1 \leftarrow R1 + R2 (18 bits)
STR	R1		C	; [C] \leftarrow R1

Program segment size = $6 \times 3 + 3 = 21$ Bytes

Size in bits = $45 \times 3 + 18 = 153$ bits

(b) Segment to compute $C = A + B$ using sample RISC ISA

Figure 12.2: Execution of $C = A + B$ on hypothetical machines

So, the expectation that a CISC will produce smaller programs may not be correct. In addition, in the present time memory is inexpensive, this potential advantage of smaller programs is not so compelling these days.

Support for High-Level Language

With the increasing use of more and higher-level languages, manufacturers had provided more powerful instructions to support them. It was argued that a stronger instruction set would reduce the software crisis and would simplify the compilers. Another important reason for such a movement was the desire to improve performance.

However, even though the instructions that were closer to the high-level languages were implemented in Complex Instruction Set Computers (CISCs), still it was hard to exploit these instructions since the compilers were needed to find those conditions that exactly fit those constructs. In addition, the task of optimising the generated code to minimise code size, reduce instruction execution count, and enhance pipelining is much more difficult with such a complex instruction set.

Another motivation for increasingly complex instruction sets was that the complex HLL operation would execute more quickly as a single machine instruction rather than as a series of more primitive instructions. However, because of the bias of programmers towards the use of simpler instructions, it may turn out otherwise. CISC makes the more complex control unit with larger microprogram control store to accommodate a richer instruction set. This increases the execution time for simpler instructions.

Thus, it is far from clear that the trend to complex instruction sets is appropriate. This has led a number of groups to pursue the opposite path.

12.2.3 High Level Language Program Characteristics

The new architectures should support high-level language programming. A high-level language system can be implemented mostly by hardware or mostly by software, provided the system hides any lower-level details from the programmer. Thus, a cost-effective system can be built by deciding what pieces of the system should be in hardware and what pieces in software.

To ascertain the above, it may be a good idea to find program characteristics on general computers. Some of the basic findings about the program characteristics were:

Variables	Operations	Procedure Calls
Integral Constants 15-25%	Simple assignment 35-45%	Most time-consuming operation.
Scalar Variables 50-60%	Looping 2-6%	FACTS: Most of the procedures are called with fewer than 6 arguments.
Array/ structure 20-30%	Procedure call 10-15% IF 35-45% GOTO FEW Others 1-5%	Most of these have fewer than 6 local variables

Figure 12.3: Typical Program Characteristics

Observations

- Integer constants appeared almost as frequently as arrays or structures.
- Most of the scalars were found to be local variables, whereas most of the arrays or structures were global variables.
- Most of the dynamically called procedures pass fewer than six arguments.
- The numbers of scalar variables are less than six.
- A good machine design should attempt to optimize the performance of most time-consuming features of high-level programs.
- Performance can be improved by more register references rather than having more memory references.
- There should be an optimized instructional pipeline such that any change in flow of execution is taken care of.

The Origin of RISC

In the 1980s, a new philosophy evolved having optimizing compilers that could be used to compile “normal” programming languages down to instructions that were as simple as equivalent micro-operations in a large virtual address space. This made the instruction cycle time as fast as the technology would permit. These machines should have simple instructions such that it can harness the potential of simple instruction execution in one cycle – thus, having reduced instruction sets – hence the reduced instruction set computers (RISCs).

Check Your Progress 1

1. List the reasons of increased complexity.

.....
.....
.....

2. State True or False

T	F
---	---

- a) The instruction cycle time for RISC is equivalent to CISC.
- b) CISC yields smaller programs than RISC, which improves its performance; therefore, it is very superior to RISC.
- c) CISC emphasizes optional use of register while RISC does not.

12.3 RISC ARCHITECTURE

Let us first list some important considerations of RISC architecture:

1. The RISC functions are kept simple unless there is a very good reason to do otherwise. A new operation that increases execution time of an instruction by 10 per cent can be added only if it reduces the size of the code by at least 10 per cent. Even greater reductions might be necessary if the extra modification necessitates a change in design.
2. Micro-instructions stored in the control unit cannot be faster than simple instructions, as the cache is built from the same technology as writable control unit store, a simple instruction may be executed at the same speed as that of a micro-instruction.
3. Microcode is not magic. Moving software into microcode does not make it better; it just makes it harder to change. The runtime library of RISC has all the characteristics of functions in microcode, except that it is easier to change.

4. Simple decoding and pipelined execution are more important than program size. Pipelined execution gives a peak performance of one instruction every step. The longest step determines the performance rate of the pipelined machine, so ideally each pipeline step should take the same amount of time.
5. Compiler should simplify instructions rather than generate complex instructions. RISC compilers try to remove as much work as possible during compile time so that simple instructions can be used. For example, RISC compilers try to keep operands in registers so that simple register-to-register instructions can be used. RISC compilers keep operands that will be reused in registers, rather than repeating a memory access or a calculation. They use LOADs and STOREs to access memory so that operands are not implicitly discarded after being fetched. (Refer to Figure 12.1(b) for a simple illustration).

Thus, the RISC were designed having the following:

- **One instruction per cycle:** A machine cycle is the time taken to fetch two operands from registers, perform the ALU operation on them and store the result in a register. Thus, RISC instruction execution takes about the same time as the micro-instructions on CISC machines. With such simple instruction execution rather than micro-instructions, it can use fast logic circuits for control unit, thus increasing the execution efficiency further.
- **Register-to-register operands:** In RISC machines the operation that access memories are LOAD and STORE. All other operands are kept in registers. This design feature simplifies the instruction set and, therefore, simplifies the control unit. For example, a RISC instruction set may include only one or two ADD instructions (e.g. integer add and add with carry); on the other hand a CISC machine can have 25 add instructions involving different addressing modes. Another benefit is that RISC encourages the optimization of register use, so that frequently used operands remain in registers.
- **Simple addressing modes:** Another characteristic is the use of simple addressing modes. The RISC machines use simple register addressing having displacement and PC relative modes. More complex modes are synthesized in software from these simple ones. Again, this feature also simplifies the instruction set and the control unit.
- **Simple instruction formats:** RISC uses simple instruction formats. Generally, only one or a few instruction formats are used. In such machines the instruction length is fixed and aligned on word boundaries. In addition, the field locations can also be fixed. Such an instruction format has a number of benefits. With fixed fields, opcode decoding and register operand accessing can occur in parallel. Such a design has many advantages. These are:
 - It simplifies the control unit
 - Simple fetching as memory words of equal size are to be fetched
 - Instructions are not across page boundaries.

Thus, RISC is potentially a very strong architecture. It has high performance potential and can support VLSI implementation. Let us discuss these points in more detail.

- **Performance using optimizing compilers:** As the instructions are simple the compilers can be developed for efficient code organization also maximizing register utilization etc. Sometimes even the part of the complex instruction can be executed during the compile time.
- **High performance of Instruction execution:** While mapping of HLL to machine instruction the compiler favours relatively simple instructions. In addition, the control unit design is simple and it uses little or no micro-

instructions, thus could execute simple instructions faster than a comparable CISC. Simple instructions support better possibilities of using instruction pipelining.

- **VLSI Implementation of Control Unit:** A major potential benefit of RISC is the VLSI implementation of microprocessor. The VLSI Technology has reduced the delays of transfer of information among CPU components that resulted in a microprocessor. The delays across chips are higher than delay within a chip; thus, it may be a good idea to have the rare functions built on a separate chip. RISC chips are designed with this consideration. In general, a typical microprocessor dedicates about half of its area to the control store in a micro-programmed control unit. The RISC chip devotes only about 6% of its area to the control unit. Another related issue is the time taken to design and implement a processor. A VLSI processor is difficult to develop, as the designer must perform circuit design, layout, and modeling at the device level. With reduced instruction set architecture, this processor is far easier to build.

12.4 THE USE OF LARGE REGISTER FILE

In general, the register storage is faster than the main memory and the cache. Also the register addressing uses much shorter addresses than the addresses for main memory and the cache. However, the numbers of registers in a machine are less as generally the same chip contains the ALU and control unit. Thus, a strategy is needed that will optimize the register use and, thus, allow the most frequently accessed operands to be kept in registers in order to minimize register-memory operations.

Such optimisation can either be entrusted to an optimising compiler, which requires techniques for program analysis; or we can follow some hardware related techniques. The hardware approach will require the use of more registers so that more variables can be held in registers for longer periods of time. This technique is used in RISC machines.

It may seem that a large number of registers would lead to fewer memory accesses, however in general, about 32 registers were considered optimum. So how does this large register file further optimize the program execution?

Since most operand references are to local variables of a function in C they are the obvious choice for storing in registers. Some registers can also be used for global variables. However, the problem here is that the program follows function call - return so the local variables are related to most recent local function, in addition this call - return expects saving the context of calling program and return address. This also requires parameter passing on call. On return, from a call the variables of the calling program must be restored and the results must be passed back to the calling program.

RISC register file provides a support for such call-returns with the help of register windows. Register files are broken into an overlapping set of smaller group of registers, as shown in Figure 12.4. Each of these register set can be used for different function/subroutine. A function call automatically changes the set being used. The use from one fixed size window of registers to another, rather than saving registers in memory as done in CISC. Windows for adjacent procedures are overlapped. This feature allows parameter passing without moving the variables at all. The following figure tries to explain this concept:

Assumptions:

Register file contains 138 registers. Let them be called by register number 0 – 137. Further, a program has three functions, viz. main, sorting and Xchange. The operating

system calls function main (f_{main}) which calls function sorting ($f_{sorting}$) and function sorting calls function Xchage (f_{Xchage}).

Registers Nos.	Used for	Function main	Function sorting	Function Xchage
0 – 9	Global variables required by f_{main} , $f_{sorting}$, and f_{Xchage}			
10 – 83	Unused			
84 – 89 (6 Registers)	Used by parameters of f_{Xchage} that may be passed to next call			Temporary variables of function Xchage
90 – 99 (10 Registers)	Used for local variable of f_{Xchage}			Local variables of function Xchage
100 – 105 (6 Registers)	Used by parameters that were passed from $f_{sorting} \rightarrow f_{Xchage}$		Temporary variables of function sorting	Parameters of function Xchage
106 – 115 (10 Registers)	Local variables of $f_{sorting}$		Local variables of function sorting	
116 – 121 (6 Registers)	Parameters that were passed from f_{main} to $f_{sorting}$	Temporary variables of function main	Parameters of function sorting	
122 – 131 (10 Registers)	Local variable of f_{main}	Local variables of function main		
132 – 138 (6 Registers)	Parameter passed to f_{main}	Parameters of function main		

Figure 12.4: Use of three Overlapped Register Windows

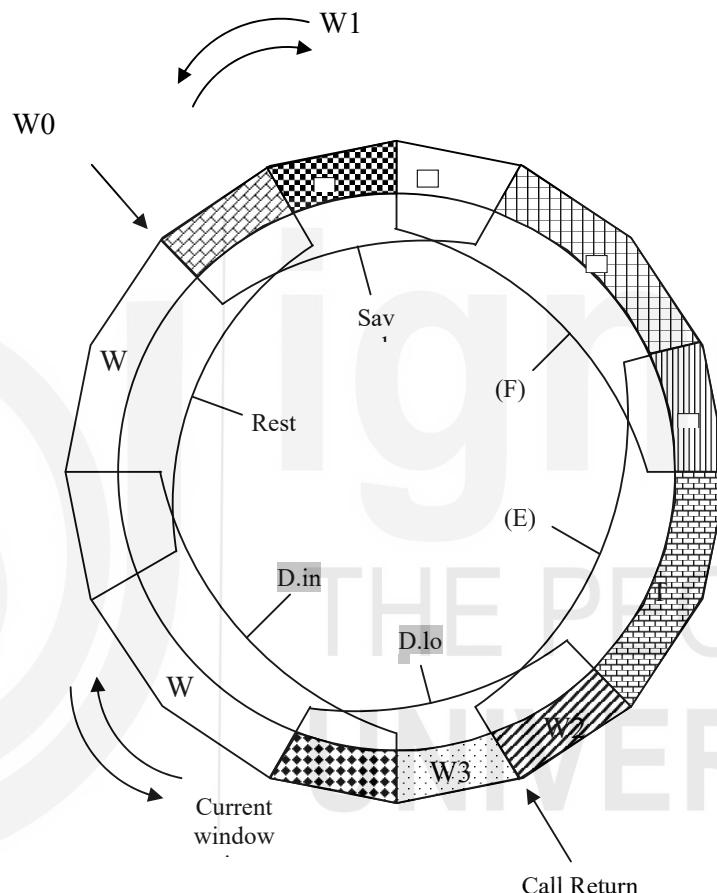
Functioning of the registers: at any point of time the global registers and one set of register being used for a specific function would be active for execution of the program. Thus, for programming purpose there may be only 32 registers. Window in the above example although has a total of 138 registers. This window consists of:

- Global registers which are shareable by all functions.
- Parameters registers for holding parameters passed from the previous function to the current function. They also hold the results that are to be passed back.
- Local registers that hold the local variables, as assigned by the compiler.
- Temporary registers: The temporary registers are used to pass parameters to the function that is called by the presently executing function. The parameters to be passed are stored in these temporary registers and passed as parameters to the function that is called by the presently executing function.

But what is the maximum function calls nesting can be allowed through RISC? Let us describe it with the help of a circular buffer diagram, technically the registers as above have to be circular in the call return hierarchy.

This organization is shown in the following figure. The register buffer is filled as function A called function B, function B called function C, function C called function

D. The function D is the current function. The current window pointer (CWP) points to the register window of the most recent function (function D in this case). Any register references by a machine instruction is added with the contents of CWP to compute the address of the registers holding the operands for the executing function. The other register, i.e., the saved window pointer registers points to the most recent register window that has been saved in the memory of the computer. This action will be needed if a further call is made and there is no space for that call. If function D now calls function E arguments for function E are placed in D's temporary registers indicated by D.temp and the CWP is advanced by one window.



A.in:	Input register parameters/argument of function A
A.loc:	Local variables of function A
B.in or A.temp	Parameters with which function B is to be called B.in. It is same as A.temp which are parameters passed by function A to function B

Figure 12.5: Circular-Buffer Organization of Overlapped Windows

Assume that now the function E calls function F. This call cannot be serviced as the circular buffer already has allowed number of function call, unless you free space equivalent to exactly one window. This condition can easily be determined as current window pointer on incrementing will be equal to saved window pointer. Now, we need to create space; how can we do it? The simplest way will be to swap F_A register to memory and use that space. Thus, an N window register file can support $N - 1$ level of function calls.

Thus, the register file, organized in the form as above, is a small fast register buffer that holds most of the variables that are likely to be used heavily. From this point of view the register file acts almost like a cache memory. So let us find how the two approaches are different.

Characteristics of large-register-file and cache organizations

Large Register File	Cache
Hold local variables for almost all functions. This saves time.	Recently used local variables are fetched from main memory for any further use. Dynamic use optimises memory.
The variables are individual.	The transfer from memory is block wise.
Global variables are assigned by the compilers.	It stores recently used variables. It cannot keep track of future use.
Save/restore needed only after the maximum call nesting is over (that is $n - 1$ open windows) .	Save/restore based on cache replacement algorithms.
It follows faster register addressing.	It is memory addressing.

The basic difference is due to addressing overhead of the two approaches. Small Register (R) address are smaller than the Cache reference, which are generated from a long memory address. Thus, for simple variables access register file is superior to cache memory. However, even in RISC computer, performance can be enhanced by the addition of instruction cache.

Check Your Progress 2

- State True or False in the context of RISC architecture:

A.in

 - RISC has a large register file so that more variables can be stored in register or longer periods of time.
 - Only global variables are stored in registers.
 - Variables are passed as parameters in registers using temporary registers in a window.
 - Cache is superior to a large register file as it stores most recently used local scalars.
 - An overlapped register window RISC machine is having 32 registers. Suppose 8 of these registers are dedicated to global variables and the remaining 24 are split for incoming parameters, local and scalar variables and outgoing parameters. What are the ways of allocating these 24 registers in the three categories?
-
-
-

12.5 COMMENTS ON RISC

Let us now try and answer some of the comments that are asked for RISC architectures. Let us provide our suggestions on those.

CISCs provide better support for high-level languages as they include high-level language constructs such as CASE, CALL etc.

Yes CISC architecture tries to narrow the gap between assembly and High-Level Language (HLL); however, this support comes at a cost. If the architect provides a feature that looks like the HLL construct but runs slowly, or has many options, the compiler writer may omit the feature, or even, the HLL programmer may avoid the construct, as it is slow and cumbersome. Thus, the comment above does not hold.

It is more difficult to write a compiler for a RISC than a CISC.

The studies have shown that it is not so due to the following reasons:

If an instruction can be executed in more ways than one, then more cases must be considered. For it the compiler writer needed to balance the speed of the compilers to get good code. In CISCs compilers need to analyze the potential usage of all available instruction, which is time consuming. Thus, it is recommended that there is at least one good way to do something. In RISC, there are few choices; for example, if an operand is in memory, it must first be loaded into a register. Thus, RISC requires simple case analysis, which means a simple compiler, although more machine instructions will be generated in each case.

RISC is tailored for C language and will not work well with other high level languages.

But the studies of other high-level languages found that the most frequently executed operations in other languages are also the same as simple HLL constructs found in C, for which RISC has been optimized. Unless a HLL changes the paradigm of programming you may get similar result.

The good performance is due to the overlapped register windows; the reduced instruction set has nothing to do with it.

Certainly, a major portion of the speed is due to the overlapped register windows of the RISC that provide support for function calls. However, please note this register windows are possible due to reduction in control unit size. In addition, the control is simple in RISC than CISC, thus further helping the simple instructions to execute faster.

12.6 RISC PIPELINING

Instruction pipelining is often used to enhance performance. A RISC machine, in general, consists of two types of instructions:

- a) The memory reference instructions, which are load and store instructions (Figure 12.1(b)). These instructions are used to bring/send data in registers from/to memory.
- b) Data processing instructions (ADD instruction in Figure 12.1(b)), which perform the operation on register operands.

A memory reference instruction may be divided into the following pipeline stages:

FI: Fetch the Instruction from a memory address.

EI: Compute the effective address of the operand in the memory using the addressing modes. This may be similar to execution of an instruction.

TD: Transfer data from/to register to/from memory location

The data processing instruction would just require two pipeline stages:

FI: Fetch the Instruction from a memory address.

EI: Execute the instruction on register operands, result is stored in register.

Let us explain pipelining in RISC with an example program that uses instruction set given in Figure 12.1(b), with few additional instruction MUL, which use the same format as ADD instruction. The program segment implements the following expression:

$$Z = (A + B) \times C$$

- | | | |
|-----|------------|-------------------------------------|
| (1) | LDA R1, A | (Load memory location A to R1) |
| (2) | LDA R2, B | (Load memory location B to R2) |
| (3) | ADD R1, R2 | (R1 \leftarrow R1 + R2) |
| (4) | LDA R2, C | (Load memory location C to R2) |
| (5) | MUL R1, R2 | (R1 \leftarrow R1 \times R2) |
| (6) | STR R1, Z | (Store result in memory location Z) |

As discussed earlier, each of the instructions (1), (2), (4) and (6) will be processed in three stages and each of the instructions (3) and (5) will be processed in two stages. Assuming that one stage is executed in one clock cycle, a total of $4 \times 3 + 2 \times 2 = 16$ clock cycles would be required if these instructions are not executed without using an instruction pipeline. However, if a pipeline that allows various overlapping stages to execute in parallel would result in execution of these instructions in only 8 clock cycles (Please refer to Figure 12.6)

	(1)	LDA R1, A	FI	EI	TD			
	(2)	LDA R2, B		FI	EI	TD		
	(3)	ADD R1, R2			FI		EI	
	(4)	LDA R2, C			FI	EI	TD	
	(5)	MUL R1, R2				FI		EI
	(6)	STR R1, Z					FI	EI
	Clock Cycles	1	2	3	4	5	6	7
		Time = 8 units						

Figure 12.6: RISC Pipelining

You may please note that the pipeline as shown above suffers from data dependencies at instruction (3) and (5). In both these cases, the preceding data access instruction must complete to allow the execution of these instructions. In addition, an instruction pipeline may suffer due to presence of branch instruction penalties. Next, we discuss about how such problems can be minimized

Optimization of Pipelining

RISC machines can employ a very efficient pipeline scheme because of the simple and regular instructions. Like all other instruction pipelines RISC pipeline suffer from the problems of data dependencies and branching instructions. The data dependency problem can be handled using an optimizing compiler, which can reschedule some of the instructions. For example, in the given program segment the, following changes may minimize the data dependency:

Interchange instruction (3) and instruction (4), in addition, instead of loading memory location C in R2 use R3 register. Accordingly, in instruction (5) change R2 to R3. The new instruction pipeline is shown in Figure 12.7.

(1)	LDA R1, A	FI	EI	TD				
(2)	LDA R2, B		FI	EI	TD			
(4)	<i>LDA R3, C</i>			FI	EI	TD		
(3)	<i>ADD R1, R2</i>				FI	EI		
(5)	<i>MUL R1, R3</i>					FI	EI	
(6)	STR R1, Z						FI	EI
	Clock Cycles	1	2	3	4	5	6	7
								8
								Time = 8 units

Figure 12.7: Pipeline without data dependencies

The second problem of Branch instruction penalty can be optimised in RISC by using several techniques. For example, consider that a conditional branch instruction: “In case after the computation, the value R1 register is zero, then instruction 6 is not executed and the program jumps to instruction 7” exists after instruction 5. This modified instruction sequence is shown in Figure 12.8.

(1)	LDA R1, A	FI	EI	TD				
(2)	LDA R2, B		FI	EI	TD			
(4)	<i>LDA R3, C</i>			FI	EI	TD		
(3)	<i>ADD R1, R2</i>				FI	EI		
(5.a)	<i>MUL R1, R3</i>					FI	EI	
(5.b)	If R1=0 JMP to (7)						FI	EI
(6)	STR R1, Z						FI	EI
(7)	<i>ADD R2, R3</i>						FI	EI
	Clock Cycles	1	2	3	4	5	6	7
							8	9
								Time = 9 units

Figure 12.8: Pipeline with Brach Instruction

The problem with this instruction cycle is that the instruction 6 has already been fetched to the pipeline, therefore, in case R1 has zero value the pipeline should be emptied, i.e., instruction 6 will be removed from the pipeline. After that the instruction (7) should be stated again. There are two possible solutions to this problem. First the fetch of instruction (6) may be delayed for a cycle, so that the decision, whether the branch is to be taken or not would be made. Based on that the next instruction would be fetched. This is shown in Figure 12.9.

(1)	LDA R1, A	FI	EI	TD				
(2)	LDA R2, B		FI	EI	TD			
(4)	<i>LDA R3, C</i>			FI	EI	TD		
(3)	<i>ADD R1, R2</i>				FI	EI		
(5.a)	<i>MUL R1, R3</i>					FI	EI	
(5.b)	If R1=0 JMP to (7)						FI	EI
(5.c)	<i>DO NOTHING instruction</i>						FI	EI
(6) or (7)	STR R1, Z <i>ADD R2, R3</i>						FI	EI
	Clock Cycles	1	2	3	4	5	6	7
							8	9
								TD

Figure 12.9: Pipeline with reduced Brach penalty

Another way to handle the branch penalty is by moving the branch instruction, so that the branch decision is known prior to the execution of next instruction after the branch instruction. Figure 12.10 shows this solution.

(1)	LDA R1, A	FI	EI	TD					
(2)	LDA R2, B		FI	EI	TD				
(4)	LDA R3, C			FI	EI	TD			
(3)	ADD R1, R2				FI	EI			
(5.b)	If $R1=0$ or $R3 = 0$ <i>JMP to (7)</i>					FI	EI		
(5.a)	MUL R1, R3						FI	EI	
(6) or (7)	STR R1, Z ADD R2, R3						FI	EI	TD
	Clock Cycles	1	2	3	4	5	6	7	8
									9

Figure 12.10: Pipeline with optimised Branch penalty

Please note that the instruction at (5.b) shows a hypothetical instruction that checks two conditions at a time. The purpose here is to demonstrate the concept, therefore, such instruction has been shown. Please also note the change in the sequence of instructions (5.a) and (5.b). Please also note that decision to take the branch has been taken at clock cycle 6, therefore, at lock cycle 7, it will be known, which of the two instructions (6) or (7) is to be fetched.

Finally, let us summarize the basic differences between CISC and RISC architecture. The following table lists these differences:

CISC	RISC
1. In general, large number of instructions	1. Relatively fewer instructions than CISC
2. Employs a variety of data types and a large number of addressing modes.	2. Relatively fewer addressing modes.
3. Variable-length instruction formats.	3. Fixed-length instructions, easy to decode instruction format.
4. Instructions manipulate operands residing in memory.	4. Mostly register-register operations. The only memory access is through explicit load and store instructions.
5. Number of Cycles Per Instruction varies from 1-20 depending upon the instruction.	5. Number of cycles per instruction is one as it uses pipelining. Pipeline in RISC is optimised because of simple instructions and instruction formats.
6. About 32 general purpose register, but no support is available for the parameter passing and function calls.	6. Large number of registers, which are used as Global registers and as a register based procedural call and parameter passing.
7. Microprogrammed Control Unit.	7. Hardwired Control Unit.

Check Your Progress 3

- What are the problems, which prevent RISC pipelining to achieve maximum speed?

.....
.....
.....

- How can the above problems be handled?

3. What are the problems of RISC architecture? How are these problems compensated such that there is no reduction in performance?
-
-
-

12.7 SUMMARY

RISC represents new styles of computers that take less time to build yet provide a higher performance. While traditional machines support HLLs with instruction that look like HLL constructs, this machine supports the use of HLLs with instructions that HLL compilers can use efficiently. The loss of complexity has not reduced RISC's functionality; the chosen subset, especially when combined with the register window scheme, emulates more complex machines. Thus, we see that because of all the features discussed above, the RISC architecture should prove to better for certain applications.

In this unit we have also covered the details of the pipelined features of the RISC architecture, which support this architecture to show better performance.

12.8 SOLUTIONS/ ANSWERS

Check Your Progress 1

1.
 - Speed of memory is slower than the speed of CPU.
 - Microcode implementation is cost effective and easy.
 - The intention of reducing code size.
 - For providing support for high-level language.

2.
 - a) False
 - b) False
 - c) False

Check Your Progress 2

1.
 - (a) True
 - (b) False
 - (c) True
 - (d) False

2. Assume that the number of incoming parameters is equal to the number of outgoing parameters.

Therefore, Number of locals = $24 - (2 \times \text{Number of incoming parameters})$

Return address is also counted as a parameter, therefore, number of incoming parameters is more than or equal to 1 or in other terms the possible combination, are:

Incoming Parameter Registers	Outgoing Parameter Registers	No. of Local Registers
1	1	22
2	2	20
3	3	18
4	4	16
5	5	14
6	6	12
7	7	10
8	8	8
9	9	6
10	10	4
11	11	2
12	12	0

Check Your Progress 3

1. The following are the problems:
 - Branch instruction
 - The data dependencies between the instructions
2. It can be improved by:
 - Changing the sequence of some instruction
 - causing optimized/ delayed jumps/loads etc.
3. The problems of RISC architecture are:
 - More instructions to achieve the same amount of work as CISC.
 - Higher instruction traffic
 - However, the cycle time of one instruction per cycle and instruction cache in the chip may compensate for these problems.

Indira Gandhi
National Open University
School of Computer and
Information Sciences

Block

4

Microprocessor and Advanced Architectures

UNIT 13

Microprocessor Architecture

UNIT 14

Introduction to Assembly Language Programming

UNIT 15

Assembly Language Programming

UNIT 16

Advanced Architectures

FACULTY OF THE SCHOOL

Prof P. V. Suresh, Director
Dr Akshay Kumar
Dr Sudhansh Sharma

Prof. V. V. Subrahmanyam
Mr Mangala Prasad Mishra

PROGRAMME/COURSEDESIGN COMMITTEE

Shri Sanjeev Thakur
Amity School of Computer Sciences,
Noida Shri Amrit Nath Thulal
Amity School of Engineering and Technology
New Delhi
Dr. Om Vikas(Retd),
Ministry of ICT, Delhi
Shri Vishwakarma
Amity School of Engineering and
Technology New Delhi
Prof (Retd) S. K. Gupta, IIT Delhi
Prof. T.V. Vijay Kumar, SC&SS, JNU,
New Delhi
Prof. Ela Kumar, CSE, IGDTUW, Delhi

Prof. Gayatri Dhingra, GVMITM, Sonipat
Sh. Milind Mahajani
Impressico Business Solutions, Noida, UP
Prof. V. V. Subrahmanyam
SOCIS, New Delhi
Prof. P. V. Suresh
SOCIS, IGNOU, New Delhi
Dr. Shashi Bhushan
SOCIS, IGNOU, New Delhi
Shri Akshay Kumar,
SOCIS, IGNOU, New Delhi
Shri M. P. Mishra,
SOCIS, IGNOU, New Delhi
Dr. Sudhansh Sharma,
SOCIS. IGNOU. New Delhi

BLOCK PREPARATION TEAM

Dr Zahid Raja (*Content Editor*)
Jawaharlal Nehru University
New Delhi

Dr Akshay Kumar (*Course Writer – Unit 13*)
SOCIS, IGNOU
Delhi

Dr Gaurav Verma (*Course Writer - Unit 16*)
Electronics and Communication Engineering,
NIT, Kurukshetra

*Unit 14 and Unit 15 has been
adopted from MCS012: Block 4
Unit 2, Unit 3 and Unit 4.*

(*Language Editor*) School of Humanities
IGNOU

Course Coordinator: Mr Akshay Kumar

PRINTPRODUCTION

August, 2021

© Indira Gandhi National Open University, 2021

All rights reserved. No part of this work may be reproduced in any form, by mimeograph or any other means, without permission in writing from the Indira Gandhi National Open University.

Further information on the Indira Gandhi National Open University courses may be obtained from the University's office at Maidan Garhi, New Delhi-110068.

Printed at :

BLOCK 4 INTRODUCTION

In the previous blocks, we have discussed about computer organizations, the number systems, memory and input output organization, instruction set of a computer, addressing modes, the micro-operations and the control unit of a computer system.

This block presents an example micro-processor, as an example of computer architecture. We will discuss the microprocessor architecture and its programming. Our main emphasis would be on Intel 8086/ 8088 microprocessor. The newer microprocessors may use the concepts covered for 8086 microprocessors.

This block is divided into four units. We start with the introduction to microprocessors, with special emphasis on 8086 microprocessors. Unit 1 also gives a brief introduction to the Instruction Set and the addressing modes of the 8086 microprocessor. Taking this as the base, in Unit 2 we get on to the Introduction of Assembly Language Programming. In this unit, we will also give a brief account of various tools required to develop and execute an Assembly Language Program. In Unit 3, a detailed study of Assembly Language, its programming techniques along with several examples have been taken up. Unit 4 presents a brief discussion on some of the advanced architectures.

This block gives you only some details about 8086 microprocessor and assembly language programming. For complete details on this Intel series of microprocessors, you may refer to the further readings, given below. You may also study the advanced architectures from the further readings given in Block 1.

FURTHER READINGS FOR THE BLOCK

1. IBM PC Assembly Language and Programming, Fifth Edition, Peter Abel.
2. Douglas V. Hall: Microprocessors and Interfacing – Programming and Hardware by – McGraw Hill – 1986.
3. Peter Norton & John Socha: Assembly Language book for IBM PC-Prentice Hall of India, 1989.
4. Yu-Cheng Liu, A. Gibson: Micro-computer Systems: The 8086/ 8088 family – Prentice Hall of India, 1986.
5. Douglas V. Hall; Microprocessors and Digital Systems 2/e, McGraw Hill 1986.
6. William B. Giles, Assembly Language Programming for the Intel 80xxx family, Maxwell Macmillan International editions 1991.

THE PEOPLE'S
UNIVERSITY

UNIT 13 MICROPROCESSOR ARCHITECTURE

Structure	Page No.
13.0 Introduction	
13.1 Objectives	
13.2 Structure of 8086 CPU	
13.2.1 Bus Interface Unit	
13.2.2 Execution Unit	
13.2.3 Register Set	
13.3 Instruction Set of 8086	
13.3.1 Data Transfer Instructions	
13.3.2 Arithmetic Instructions	
13.3.3 Bit Manipulation Instructions	
13.3.4 Program Execution Transfer Instructions	
13.3.5 String Instructions	
13.3.6 Processor Control Instructions	
13.4 Addressing Modes	
13.4.1 Register Addressing Mode	
13.4.2 Immediate Addressing Mode	
13.4.3 Direct Addressing Mode	
13.4.4 Indirect Addressing Mode	
13.5 Summary	
13.6 Solutions/Answers	

13.0 INTRODUCTION

In the previous three blocks of this course, you have gone through the concept of data representation, logic circuits, memory and I/O organisation, instruction set architecture, micro-operations, control unit etc. of a computer system. The processor of a general purpose computer consists of an instruction set, which uses a set of addressing modes. The control unit of a processor uses a set of registers and arithmetic logic unit to process these instructions. This unit present details of a micro-processor, in the content of all the above concepts. We have selected a simple micro-processors 8086, for the discussion. Although the processor technology is old, all the concepts are valid for higher end Intel processor family, which are commonly referred to as x86 family. This block does not attempt to make you an expert assembly programmer, however, you will be able to write reasonably good assembly programs . This unit discusses the 8086 microprocessor in some detail. This unit will introduce you to block diagram of components of 8086 microprocessor. This is followed by discussion on the register organization for this processor. Some useful instructions and addressing modes of this processor are also discussed in this unit. Please note the concepts discussed in this unit may be useful in writing good Assembly Programs.

13.1 OBJECTIVES

After going through this unit, you should be able to:

- explain the role of various components of 8086 microprocessor;
- illustrate the use of segmentation in 8086 microprocessor;
- use some of the important instruction of 8086 microprocessor
- illustrate the use of different types of addressing modes of 8086 microprocessor.

13.2 STRUCTURE OF 8086 MICROPROCESSOR

A microprocessor contains one or more processing unit on a single chip. Today's processors contain multiple processing cores in a single chip, therefore are called multi-core processors. A computer system consists of a micro-processor, memory unit and input/output interfaces, internal and external connection structure, such as buses; and several input/output devices. The bus size of a processor is a very important design parameter. For example, the address bus of a processor, generally, can determine the size of the physical main memory. The data bus determines the size of the data that can be transferred from the memory to the processor registers.

The size of address bus of 8086 micro-processor is 20 bits and data bus is 16 bits. Thus, 8086 micro-processor has $2^{20} = 1\text{M}$ Byte base memory. From this memory, about 640 KB was part of base RAM and remaining was used as ROM.

8086 micro-processor was designed as a complex instruction set computer with the basic objectives of supporting more instructions, addressing modes and more throughput. Present day multi-core processors are far more powerful than 8086 micro-processor, but objective of this block is to introduce some of the basic features of micro-processor and assembly language programming. For the basic discussion this processor is good example.

A microprocessor executes a sequence of machine instruction, which can be represented as the following notional program:

```
repeat execution of <instruction cycle>
{
    fetch(instruction);
    execute(instruction)
        decode instruction
        fetch operand;
        execute the desired operation on data
    if (interrupt) process it and return to program execution;
}
```

The 8086 microprocessor consists of two independent units (refer to Figure 13.1):

1. The Bus Interface unit, and
2. The Execution unit.

These units can function independently, therefore, they can function as two stages instruction pipeline. Components of these two units as shown in Figure 13.1 are explained in the following sections.

13.3.1 The Bus Interface Unit

The Bus Interface Unit (BIU) is responsible for external communication through the system bus. It has a dedicated address adder circuit, which takes input from segment registers and special registers of the processor. This unit also has an instruction stream queue of 6 byte length and is used to store the instruction, which is to be executed. The main tasks of this unit are:

- Computing the physical address of the instruction or data or input/output port from/ to the information is to be read or written into.
- This unit then reads or writes the information from the physical address as computed above.
 - If an instruction is fetched, it is stored in the instruction stream queue.

- Data is fetched into a general purpose register.
- In case of writing the data value of a selected register is written into a desired memory location or I/O port .

An instruction queue is useful only if more than one instructions are fetched simultaneously, which may be used for instruction pipelining involving the stages of instruction fetch and instruction execution.

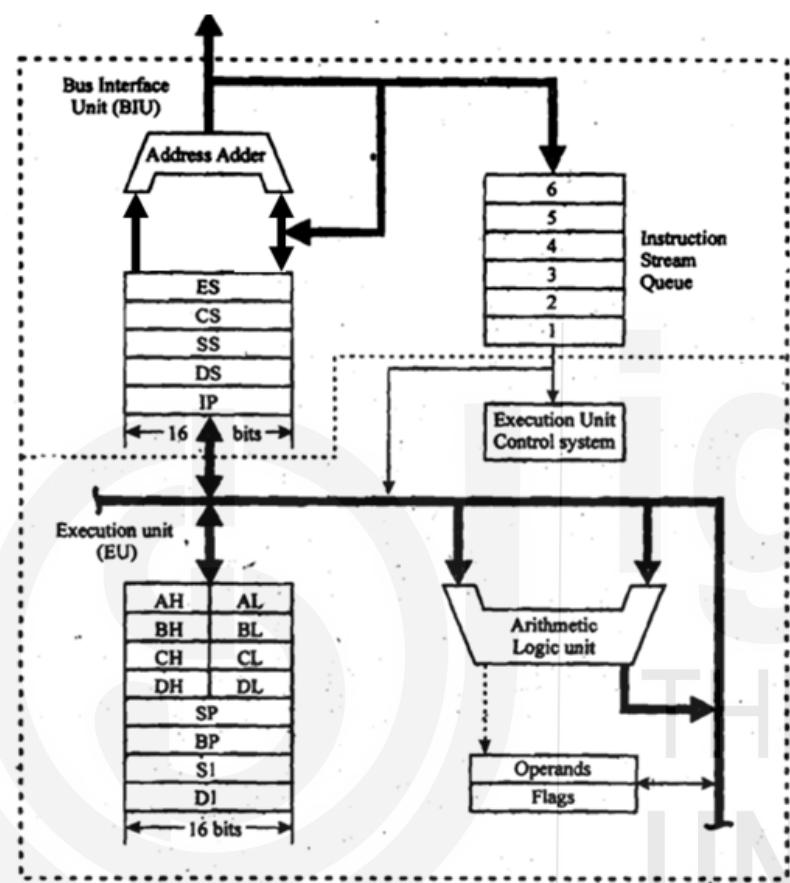


Figure 13.1: Structure of Intel 8086 Microprocessor

The Segment Registers

8086 microprocessor uses a very interesting concept of segmentation. As stated earlier that a 8086 microprocessor has 20 bit address but and 16 bit data bus. Now, assume that direct addressing scheme is used to address an operand in the memory. Since, the size of data is 16 bits, therefore, an efficient system will have a direct memory address of 16 bits, as this is the amount of data that can be fetched in one memory read operation. However, the address of an operand has to be 20 bits long (size of address bus). Thus, the two design options are either fetch two data words, which will be every inefficient; or use a concept of segmentation, which was specifically designed to this microprocessor.

The BIU of this microprocessor has four specific segment registers, namely **CS**: Code Segment register, **DS**: Data Segment register, **SS**: Stack Segment register, and **ES**: Extra Segment register. All of these registers are 16 bit long. Why segmentation? Segmentation divides the 1 MB memory of the computer associated with this microprocessor into logical overlapping segments of 64 KB. A program can have several code, data and stack segments. However, a maximum of four segments, one each of each type, may be available for accessing data and instructions at a specific

time, as there are four segment registers only. Thus, a program can consist of logical segments of code, data, stack etc. Thus, address of a data byte stored in the memory consists of a double *Segment Register (16 bits)*: *Offset Register (16 bits)* Pair. How does this segmented addressing better than fetching two words? In the segmented scheme an address included in an instructions consist of only the 16 bit memory address, thus, a segment can be a maximum size of $2^{16} = 64$ KB only. In addition, as the size of segment register is 16 bits, therefore, there can be $2^{16} = 65536$ number of segments. Please note that these segments will be overlapping as the size of base memory for this processor is only 1 MB. Figure 13.2 shows the memory organisation of 8086 microprocessor. Thus, a segment register is loaded with the address of current segments and offset is used to represent data within that segment. Thus, instruction just needs to store 16 bit address only. The address adder computes the 20 bit physical address from the *Segment Register (16 bits)*: *Offset Register (16 bits)* Pair. Please note all the content in Figure 13.2 is in hexadecimal notation.

Segment Start Address	Offset Address	Physical Memory Address	Byte content of Memory
0000 Offsets in this segment: 0000 to 001F Segment size (in byte) = 0020 in hexadecimal Which is 32 in decimal	0000	00000	52
	0001	00001	AA
	0002	00002	24

	000E	0000E	CF
	000F	0000F	32
	0010	00010	32
	0011	00011	66

	001E	0001E	--
	001F	0001F	--
	0000	00020	49
	0001	00021	23
	0002	00022	1F
0002 Offsets in this segment: 0000 to 00DF Segment size (in byte) = 00E0 in hexadecimal Which is 224 in decimal 0000 to 00DF Segment size = 00E0 in hexadecimal Which is 224 in decimal
	000E	0002E	11
	000F	0002F	42
	0010	00030	34
	0011	00031	CD

	00FE	0011E	AB
	00FF	0011F	AB

	FFFFE	1001E	DD
	FFFFF	1001F	EE

	Unused memory	FFFFC	--
		FFFFD	--
		FFFE	--
		FFFFF	--

Figure 13.2: Memory Organisation of Intel 8086 microprocessor

Figure 13.2 shows two hypothetical segments (just for illustration) in the 1MB memory using hexadecimal notation. Assume that one segment data is of 30 bytes, thus it can be accommodated in the segment of size 32 Bytes. Please note that segment start address for this segment is 0000h and offset of these locations are 0000 to 001Fh. Therefore, the second segment can start from the physical memory address 00020h. The second segment is assumed to be of 64 KB and starting from physical memory 00020h. Therefore, it has segment starting address as 0002h and offset values

from 0000h to FFFFh. An interesting fact about the memory of 8086 processor is that, although a single byte has an address, but in a single memory access two bytes are transferred through data bus. For example, access to an even memory offset 0000h will transfer bytes at offset 0000h and 0001h. However, in case, you try to access an odd memory offset like 0013h, then the bytes 0012h and 0013h would be transferred to the processor.

Now, the question is given the segment starting address of 16 bits and segment offset of 16 bits, how will you compute the physical address? The designers of 8086 used an address adder to compute physical address. The addition is performed as follows:

Given: Segment Address 0002h; and offset say 0001h

Physical address is computed by shifting segment address to left by one hexadecimal digit (appending 0 as the lowest hexadecimal digit and add the offset in the result).

The Segment Address (hexadecimal)	0	0	0	2
Shift left and add zero in least significant digit	0	0	0	2 0
Add the offset		0	0	0 1
Resulting 20 bit physical address	0	0	0	2 1

Given: Segment Address 0002h; and offset say FFFFh. The physical address will be computed as:

The Segment Address (hexadecimal)	0	0	0	2
Shift left and add zero in least significant digit	0	0	0	2 0
Add the offset		F	F	F F
Resulting 20 bit physical address	1	0	0	1 F

Please note that F+2 will be $15+2=17$, so addend is 1 and carry is 1. Also when you add carry 1 to F, it will be 16, which is addend is 0 and carry is 1

What are the advantages of segmentation?

The following are the main advantages of segmentation.

- Since segment can overlap, therefore, in case a segment is smaller than 64 KB, next segment can start almost immediately from where the segment ends, thus, memory wastage is minimized.
- A program stores only the offset within a segment, thus, program code can be relocated to another segment, if the need so be. Thus, segmentation supports writing of reloadable programs.
- Instruction uses only 16 bit address field instead of 20 bits of address bus, thus, reducing the overall size of instructions and program.

Use of Segment Registers

As discussed earlier, 8086 microprocessor has four segment registers, they are used with specialized registers that store to compute physical address. These pairs are:

- (a) Code Segment (CS) register and Instruction Pointer (IP) register, which is the offset of the next instruction to be executed, to compute the address of the next instruction to be fetched. The following example explains their use.

An assumed Code Segment(CS) Address (hexadecimal)	0	1	A	D
Shift left and add zero in least significant digit	0	1	A	D 0
Assume that IP contains an offset 1A3Ch		1	A	3 C
Resulting 20 bit physical address	0	3	5	0 C

Please note $D+3=13+3=16$, therefore, carry = 1 and sum hex digit = 0
 $A+A+carry(1)=21$, so carry = 1 and sum = $21-16=5$

- (b) Stack Segment (SS) register and Stack Pointer (SP) register, which points to the top of the stack in the stack segment, to compute the address of the top of the stack. The following example explains their use.

An assumed Stack Segment(SS) Address (hexadecimal)	F	1	1	D
Shift left and add zero in least significant digit	F	1	1	D 0
Assume that SP contains an offset 0110h		0	1	1 0

Resulting 20 bit physical address	F	1	2	E	0
--	---	---	---	---	---

- (c) Data Segment (DS) register and Offset to compute the address of the data to be fetched. The following example explains their use.

An assumed Data Segment(DS) Address (hexadecimal)	A	5	8	3
Shift left and add zero in least significant digit	A	5	8	3
Assume that data offset is A021h		A	0	2
Resulting 20 bit physical address	A	F	8	5

- (d) Extra Segment (ES) register and offset to compute the address of extra data segment (in case two data segments are used at the same time).

13.3.2 Execution Unit

The execution unit of the 8086 micro-processor consists of a set of general purpose and special purpose registers, an internal data and control bus, arithmetic and logic unit (ALU) and flags registers. The size of ALU of the 8086 processor is 16 bits. The execution unit control system decodes and performs the required operation on the data. It has the following main components:

Control Circuitry for Instruction Decode and operand specification and ALU

The 8086 processor uses a micro-programmed control unit, which decodes the instruction and executes it as per the micro-program stored in the control memory. The control unit is also responsible for generating the control timing sequences. ALU performs the operation on the data as instructed by the control unit.

Registers

8086 has several kinds of registers, which includes general purpose, special purpose registers and a special flag register. The next section explains the role of different registers of 8086 micro-processor.

13.3.3 Register Set

The 8086 registers have five different categories of registers. The following table explains the role of these registers.

Register Category	Register Name and Size	Special Purpose, if any
Segment Registers	CS (16 bits)	For storing the base address of code segment
	DS (16 bits)	For storing the base address of data segment
	SS (16 bits)	For storing the base address of stack segment
	ES (16 bits)	For storing the base address of extra data segment
General Purpose Register: Can be used for any computation, in addition they are used for specific purpose as stated in this table	AX - 16 bits ; it consists of two byte register AH, which stores the higher byte and AL, which stores the lower byte	It is also called accumulator register. It can store the results of addition or subtraction operation; for some instructions like multiplication and division it store one of the operand.
	BX - 16 bits ; it consists of BH and BL	It is also called base register. It stores the base location of a memory array.
	CX - 16 bits ; it consists of CH and CL	It is also called counter register. It can be used for keeping count in looping instructions

	DX - 16 bits ; it consists of DH and DL	This register can be used for I/O operation.
Pointer and Index Registers: These registers can also be used as general purpose registers	BP (16 bits)	Base Pointer register used in stack segment
	SI (16 bits)	Source Index register used in data segment
	DI (16 bits)	Destination Index register used in extra data segment
Special Register	SP	Stack Pointer register, points to the top of the stack.
Flags Register	It consists of 16 flags set by the last ALU operations. Each flag is 1 bit long	Some of the important flags are carry flag (CF), Parity Flag (PF), Auxiliary Flag (AF), Zero flag (ZF), Sign Flag (SF), Overflow Flag (OF), Interrupt Enable flag (IF) and other control flags.

Check Your Progress 1

1. What are the different components of Bus Interface unit and what are their uses?

.....
.....
.....

2. Compute the physical address for a 8086 microprocessor for the following:

- (a) CS:IP = 0111h:0020h
 (b) DS:BX = 0211h:0100h
 (c) SS:SP = 42AAh:0123h

.....
.....
.....

3. What is the role of a flag register in 8086 microprocessor? Can it be used as general purpose register?

.....
.....
.....

13.4 INSTRUCTION SET OF 8086

In the previous sections, the basic structure of 8086 micro-processor was discussed. This section presents the instruction set of this micro-processor. This section presents only few instructions of each type, as idea is to present example of basic instructions, which you may require to write assembly programs in the next two units.

Interestingly, the 8086 instructions can be 1 byte to 6 byte long. A general format of instructions presented here is as under:

Label:	Operation mnemonic	Operand(s)	; Comment
LOOP:	ADD	AX, DX	; AX ← AX + DX

Label is optional and is used to identify an instruction. It may be used if a subgroup of instructions are to be repeated. Operation mnemonic identifies the operation to be performed. These instructions, depending on the operation, may have zero, one or two

operands. It may be noted that in 8086 micro-processor, if an instruction has two operands, then at least one of the operand has to be a register operand; or in other words both the operands cannot be memory operands. This restriction is due to limits on the size of instruction. In addition, an operand address may use several addressing modes, which are discussed in section 1.5. The comments in the instruction are optional and are written after a semi colon (;) symbol. The example of an addition instruction is shown, where LOOP: is the label, ADD operation is to be performed on operand AX an DX. Please note that the comment in this case states the nature of addition instruction in 8086 microprocessor. The 16 bit contents of AX and DX are added and by the ALU and result is stored in AX register. In addition, this operation will also set the flags register.

The following are some of the important functional groups of the 8086 instructions.

13.4.1 Data Transfer Instructions

Data transfer instructions are required for moving the data between a pair of source and destination. Following are some of the more useful 8086 data transfer instructions.

MOV instruction: MOV destination, source (transfers source data to destination)

This instruction transfers the data from source to destination. The source or destination can be a general purpose register, immediate operand, memory location or I/O port. However, it may be noted that both the source and destination cannot be memory location in one instruction.

Example:

To move an immediate operand 2F1Ch into DX register, you can use the following instruction:

MOV DX, 2F1Ch

To move content of DX register into AX register, you can use the following instruction:

MOV AX, DX

PUSH and POP instructions: PUSH operand or POP destination

PUSH and POP instructions are used to transfer a word (2 bytes) to and from the sword stack of 8086 microprocessor. The stack in 8086 microprocessor grows from a higher memory address to lower memory address as shown in Figure 13.3

	Offset	Stack content		Offset	Stack content		Offset	Stack content
	0000			0000			0000	
	0001			0001			0001	
	
	00FC		SP	00FC	FF		00FC	
	00FD			00FD	AA		00FD	
SP	00FE	AB		00FE	AB	SP	00FE	AB
-	00FF	BD		-	BD	-	00FF	BD
Initial Stack State			Let AX = AAFFh After PUSH AX			After POP DX DX = AAFFh		

Figure 13.3: Stack after one PUSH and one POP instructions.

Please note the following about Figure 13.3

- The stack is a word stack and the operands of PUSH and POP instructions are word operands. These two instructions does not affect the flag registers.

- The maximum size of this stack segment is 0100h having offsets 0000h to 00FFh. The stack segment register value is not shown.
- In 8086 microprocessor, the stack grows from higher offset to lower offset. The stack would be empty if SP contains 0100h. Stack is full when SP is 0000h.
- The PUSH instruction causes the decrementing the stack pointer by a value 2 (as stack is a word stack and the offset is an address of a byte), i.e. SP=SP-2, and then the word operand of the PUSH instruction is put in the stack locations pointed to by the SP.
- POP instruction results in moving the content at the stack location into the destination register, specified by the instruction. This is followed by incrementing the stack pointer register value by 2, i.e. SP=SP+2.

PUSHF and POPF instructions: The PUSHF instruction is used for pushing the current flags register on to the stack, while POPF pops the content at the top of the stack to fags register.

Other data transfer instructions

There are a number of other data transfer instructions. These instruction and their purpose is given in the following table:

MNEMONIC	DESCRIPTION
XCHG destination, source	Exchanges bytes of words of source and destination. At least one operand should be a register operand.
XLAT	This is a complex instruction, which translates a byte of AL register using a lookup table. This instruction uses AL register as the operand. An example of this instruction is given in Unit 15.
LEA register, source	This instruction results in loading of 16 bit effective address of source operand to the specified register operand. This instruction is useful for array index manipulation.
IN accumulator, port address	This instruction is used to transfers a byte or word from a specified Input port to accumulator register. The instruction can use DX register as implied operand for port address. The port address can also be an immediate operand.
OUT port address, Accumulator	This instruction can be used to transfer a byte or word, which is in accumulator register to specified output port address of an output devices, such as monitor or printer
LDS/LES	These instructions are used to loading data segment/extras data segment respectively along with one specified registers. Details on these instructions are beyond the scope of this unit.
LAHF/SAHF	The LAHF loads the low byte of flags register to AH register, while SAHF stores value of AH register to low byte of flags register.

13.4.2 Arithmetic Instructions

8086 microprocessor has a large number of arithmetic instructions. These instructions are explained below:

ADD and ADC instructions: ADD destination, source and ADC destination, source

The purpose of ADD instruction is to simply add the two operands and the result of addition is stored in *destination*. The source operand can be a general purpose register, immediate operand, memory location etc. the destination may be register operand or memory location. Also both the operands should not be memory operand in an

instruction. It may be noted that both source and destination operand should either be byte operands or word operands. This instruction causes changes in several flags of the flags register. Some of these flags are: carry flag, overflow flag, sign flag, zero flag etc. ADC instruction in addition to adding the source and destination also adds the Carry flag of the flags register.

.Example:

To add an immediate operand 2F1Ch into DX register, you can use the following instruction:

ADD DX, 2F1Ch

To add content of BL register into AL register, you can use the following instruction:

MOV AL, BL ; the result of addition will be in AL register

INC and DEC instructions: INC destination and DEC destination

The purpose of INC instruction to increment the destination operand by 1, while DEC instruction decrements the destination operand by 1. The operand can a memory or register operand of byte or word type. the two operands and the result of addition is stored in *destination*. This instruction causes changes in several flags of the flags register. Some of these flags are: overflow flag, sign flag, zero flag etc. ADC instruction in addition to adding the source and destination also adds the Carry flag of the flags register.

Example:

To increment the AL register content by a value 1, you may use the instruction:

INC AL

To decrement the value of BL register by 1, you may use the instruction:

DEC BL

AAA and DAA instructions:

The AAA instruction performs ASCII adjust after addition, whereas DAA instruction performs decimal, i.e. binary coded decimal, adjust after addition. These two instructions does not have any operand expect the implied operand, which is AL register. 8086 microprocessor allows you to add two decimal digits (0 to 9) stored in ASCII format, unpacked BCD format (which consists of single BCD digit in a byte) or packed BCD format (which consists of two BCD digits in a byte). AAA is used to adjust the results, if you have added ASCII digits or unpacked BCD. DAA is used to adjust the results, if you have added two packed BCD numbers. The following examples explains these two instructions.

Example:

Consider the AL register has ASCII digit '7' and BL contains ASCII '6'. You want to add these two values to get an answer 13 in decimal. One of the way would be to convert these operand into binary and perform the addition and convert the results back to desired format. Other way will be to use AAA as follows:

```
ADD AL, BL      ; Given AL = 001101112 and BL = 001101102
                  ; Result would be 011011012 (incorrect sum)
AAA             ; Result would be adjusted by adding 0110 in lower
                  ; 4 bits and setting the AF and CF flags as AL is greater than
                  ; 9. The upper four bits will be made 000000112. The CF
                  ; indicates that the result is 13.
```

Example for DAA

```
ADD AL, BL      ; Given AL = 000101112 (packed BCD 17)
                  ; and   BL = 010101102 (packed BCD 56)
DAA             ; Result      011011012 (sum 6D is incorrect)
                  ; As lower 4 bits > 9, so adjust lower 4 bits by adding 0110
```

MUL, DIV and IDIV instructions: MUL source, DIV source and IDIV source

MUL and DIV instructions are unsigned multiplication and unsigned division instructions respectively. IDIV is a signed division instruction. The source can be a memory or register operand, which contains either byte data or word data. For these instructions one of the operand is assumed to be AL register (if data is of byte type) or AX register (if data is of word type). The result of MUL instruction is stored in AX register (if data is of one byte) or DX and AX pair (if data is of word type). Thus, symbolically the MUL instructions can be represented as:

$$\begin{aligned} \text{AX} &\leftarrow \text{AL} \times \text{source} \text{ (if source is 8 bit data)} \\ \text{DX, AX} &\leftarrow \text{AX, sources} \text{ (if source is 16 bit data)} \end{aligned}$$

In case in this instruction, if most significant bit of the result is 0, then carry and overflow flags are set to 0. In case a byte is to be multiplied with a word operand, then you must first convert the byte operand to a word operand using instructions like CBW given later in the unit.

The result of DIV and IDIV instructions for byte operands is stored as AH stores remainder and AL stores quotient of division, or for word operands DX stores the remainder and AX stores the quotient. in AH register (if data is of one byte) or DX and AX pair (if data is of word type). Thus, symbolically the MUL instructions can be represented as:

$$\begin{aligned} \text{AH (Remainder) AL (Quotient)} &\leftarrow \text{AL / source} \text{ (if source is 8 bit data)} \\ \text{DX (Remainder) AX (Quotient)} &\leftarrow \text{AX / source} \text{ (if source is 16 bit data)} \end{aligned}$$

In the division operation a 0 value in the source register will result in run time error.

Example:

Assume that AL register contains 11h and BL register contains 02h.
 Multiplication and division instructions will give following results:
 MUL BL ; Result 11h × 02h = 22h; The AH = 00h and AL=22h
 DIV BL ; Result 11h / 02h = Remainder in AH= 01h and
 ; Quotient in AL 08h

CMP instructions: compares destination and source operands

This is a very interesting instruction used for comparing two operands. This instruction only sets the flag by subtracting source from the destination operand (both byte or both word). Both the source and destination operands cannot be memory operands at the same time. This operation may set carry flag zero flag, sign flag etc. The following example explains how flags may be set by this operand. This instruction only changes the flags, no operand value is changed.

Example:

Instruction	Flags if AX= CX	Flags if AX > CX	Flags if AX < CX
CMP AX, CX	CF=0; ZF=1; SF=0	CF=0; ZF=0; SF=0	CF=1; ZF=0; SF=1

Other arithmetic instructions

Some of the other instructions are given below:

SUB destination, source	This instruction subtract source from destination. The carry flag in subtraction is a borrow flag.
SUB destination, source	Subtracts with previous borrow, if any.
NEG source	Creates the 2's complement of source number.

AAS, DAS	Works in a similar manner as AAA and DAA, except they operate after subtraction operation.
AAM, AAD	Works in a similar manner as AAA, except the operation is multiplication and division respectively.
CBW, CWD	These instructions convert byte to word or word to double word respectively. The value of sign bit is filled in the upper byte or word as the case may be. For CBW operand is in AL register and resulting word is in AX register; whereas for CWD the operand is in AX register and the double word is in DX, AX pair.

13.4.3 Bit Manipulation Instructions

The Bit manipulation instructions are used to manipulate the bit wise data. These instructions are very useful in performing logical operation on the data. The following are some of the bit manipulation instructions:

NOT, AND, OR, XOR instructions:

These are logical instructions of 8086 microprocessor. NOT instruction takes only one operand, while all other instruction have destination and source operands. The operands can be memory or register operands and both the operands cannot be memory operands in a single instruction.

Example:

Let AL= 00111010₂ and BL=11011100₂

NOT AL ; the result in AL would be 11000101₂

AND AL, BL ; the result in AL would be 00011000₂

OR AL, BL ; the result in AL would be 11111110₂

XOR AL, BL ; the result in AL would be 11100110₂

TEST destination, source instruction:

This instruction performs the AND operation on the two operands, but does not changes the operands value. This instruction clears the carry and overflow flags and sign flag and zero flags are set as per the operand.

SHL and SHR; SAL and SAR; ROL and ROR; RCL and RCR instructions:

All these eight instructions are shift instructions with small difference. All these instructions take two operands *destination and count*. The count specifies the count of times the bits of the destination operand are to be shifted. The alphabet L or R at the end of instruction mnemonic specifies Left shift or Right shift respectively. Count sometimes can be stored in CL register.

Following diagram and example explains these shift operations, assuming that data is of byte type and the count of shift is by one bit:

CF	AL Register Value								SHL is shift left; SAL is arithmetic Shift left.
0	1	0	0	0	0	0	1	1	Initial Value
1	0	0	0	0	0	1	1	0	After execution of SHL AL or SAL AL
←									Direction of shift

In SHL or SAL instruction 0 is put at the least significant bit (shown in green colour) and all the bits of the operand are shifted towards the left by 1 bit. The most significant bit is shifted to carry flag.

CF	AL Register Value								SHR is logical shift right.
1	1	1	0	0	0	0	1	1	Initial Value

1	0	1	1	0	0	0	1	After execution of SHR AL
	→							

All the bits of the byte are shifted towards the right. The most significant bit gets the value 0 and least significant bit is pushed to the carry flag (shown in green colour).

CF	AL Register Value								SAR is arithmetic shift right;
0	1 1 0 0 0 0 1 1								Initial Value
1	1 1 1 0 0 0 0 1								After execution of SAR AL
	→								

In the arithmetic shift right, all the bits are shifted towards the right. The most significant bit, which is a sign bit retains the same sign (please see the 1's in the left most position in the diagram above) and least significant bit is pushed to the carry flag (shown in green colour).

CF	AL Register Value								ROL is rotate left;
0	1 0 0 0 0 0 1 1								Initial Value
1	0 0 0 0 0 1 1 1								After execution of ROL AL
	←								

In the Rotate shift left, all the bits are shifted towards the left. The most significant bit is shifted to CF as well as rotated to least significant bit, as shown above (in green colour).

CF	AL Register Value								ROR is rotate right;
0	1 0 0 0 0 0 1 1								Initial Value
1	1 1 0 0 0 0 0 1								After execution of ROR AL
	→								

In the Rotate shift right, all the bits are shifted towards the right. The least significant bit is shifted to CF as well as rotated to most significant bit, as shown above (in green colour).

CF	AL Register Value								RCL is rotate left with carry;
0	1 0 0 0 0 0 1 1								Initial Value
1	0 0 0 0 0 1 1 0								After execution of RCL AL
	←								

In the Rotate shift left with carry, all the bits are shifted towards the left. The most significant bit is shifted to CF, and the CF is rotated to the least significant bit (shown in blue colour)

CF	AL Register Value								RCR is rotate right with carry;
0	1 0 0 0 0 0 1 1								Initial Value
1	0 1 0 0 0 0 0 1								After execution of RCR AL
	→								

In the Rotate shift right with carry, all the bits are shifted towards the right. The least significant bit is rotated to CF (shown in blue colour), and the CF is shifted to the most significant bit.

Check Your Progress 2

1. Point out the error/ errors in the following 8086 assembly instruction (if any)?
 - a. POPF DX
 - b. MOV BX
 - c. XCHG MEM_BYT1, MEM_BYT2

- d. DAA BL, CL
 - e. DIV AX, CH
2. Compare the different types of shift instructions of 8086 micro-processor.

.....
.....
.....

3. How can you check two operands are equal or not?

.....
.....
.....

13.4.4 Program Execution Transfer Instructions

In general, the execution of instructions of a program is sequential. However, there are certain instructions that results in execution of a different set of instructions. Some of these instructions including call to a procedure, return from a procedure, unconditional and conditional jump instructions etc. All these instructions contain an operand, which is the address of the next instruction which is to be executed as a consequence of execution of this instruction. The conditional jump uses flags register to determine if the jump is to be performed or not. Subroutine call instruction stores the return address. This section explains some of the important program execution transfer instructions.

CALL and RET instructions.

Call and return instructions are used form calling a procedure and once execution of the execution of the procedure is over RET instruction brings the control to the next instruction after the CALL instruction. In 8086 microprocessor there are two types of calls, viz. NEAR call and FAR call. The near call is within the same segment, whereas FAR call is to a different segment. A call instruction has the following basic format:

CALL <address of procedure>

Now, the question is how to recognize, if it is a NEAR or FAR procedure call? This is resolved by the assembler from the declaration of the procedure, which is created as a NEAR or FAR procedure. An example, explaining this is discussed in Unit 15 of this Block. A call to the procedure can be made using the CALL instruction. For example, if the name of a procedure in a separate code segment is procedure1, then the following call instruction will be used:

CALL procedure1 ;

This instruction will cause the execution for following sequence of operations:

1. If, it is a FAR procedure, then, present CS and IP should be saved as return address on the top of the stack, otherwise only IP will be stored on the stack.
SP=SP-2; SS[SP]←CS; // This step will not be required in NEAR procedure
SP=SP-2; SS[SP]←IP;
2. The CS will be loaded with the code segment address of *procedure1* and IP will be loaded with the offset of *procedure1*.
CS= CS of *procedure1*; // This step will not be required in NEAR procedure
IP = Offset of first instruction of *procedure1*;
3. The next instruction as per CS:IP value updated in step 2 will be executed next.

A procedure ends in a return instruction (RET). It causes the called procedure to return to the calling program. The following sequence of actions are performed by the RET instruction.

1. Perform the following actions:..

- ```
CS ← SS[SP] ; SP=SP+2; // NOT performed in NEAR procedure
IP ← SS[SP] ; SP=SP+2;
2. The next instruction as per CS:IP value updated in step 1 will be executed
next.
```

*Jump instructions:*

8086 micro-processor have instructions for unconditional and conditional jump instructions. The unconditional jump can be to NEAR or FAR label. It only requires one operand, which is the address, specified using a Label, of the next instruction to be executed. The format of this instruction is given below:

JMP Label

There are number of unconditional jump instructions. An unconditional jump instruction checks various flag register to determine, if the jump is to be taken or not. One of the most common instruction to set flags prior to conditional jump instruction is the CMP instruction, which has been explained in section 13.4.2. Following table lists some of the conditional jump instructions along with the condition, when the jump will be taken.

| Instruction | Condition if the prior instruction is<br><b>CMP AX, BX or any other arithmetic instruction</b> |
|-------------|------------------------------------------------------------------------------------------------|
| JA/JNBE     | Jump if AX > BX                                                                                |
| JAE/JNB     | Jump if AX >= BX                                                                               |
| JB/JNAE     | Jump if AX < BX                                                                                |
| JBE/JNA     | Jump if AX <= BX                                                                               |
| JC          | Jump if carry flag is set                                                                      |
| JE/JZ       | Jump if AX = BX                                                                                |
| JNC         | Jump if no carry                                                                               |
| JNE/JNZ     | Jump if AX ≠ BX                                                                                |
| JO          | Jump if overflow flag is set                                                                   |
| JNO         | Jump if overflow flag is not set                                                               |
| JP/JPE      | Jump if parity flag is set ; Jump if parity is even                                            |
| JNP/JPO     | Jump if parity flag is not set ; Jump if parity is odd                                         |
| JG/JNLE     | Jump if AX > BX                                                                                |
| JA/JNL      | Jump if AX > BX                                                                                |
| JL/JNGE     | Jump if AX < BX                                                                                |
| JLE/JNG     | Jump if AX <= BX                                                                               |
| JS          | Jump if sign flag is set                                                                       |
| JNS         | Jump if sign flag is not set                                                                   |
| JCXZ        | Jump to specified address if CX =0                                                             |

The instruction JCXZ is a very useful instruction, when CX register is used as a counter.

*Loop instructions:*

A loop instruction (LOOP label) uses *CX register as a counter register*. The label in the loop instruction should be in the range -128 to +127. Prior to a loop instruction, the looping count value should be moved to CX register. The Loop instruction decrements the CX register and checks if CX register has zero value. If CX is not zero, then loop instruction takes the program back to the instruction, which is specified by the label of that instruction. In case, CX is zero then the loop is terminated, i.e., the next instruction after the loop instruction is executed in sequence. 8086 micro-processor has a number of loop instruction, which differ in condition the condition of loop termination. The following table lists some of these instructions, which may be used

later Units. There are many other such instructions for looping, a discussion on them is beyond the scope of this unit.

| Instruction         | Loop Termination                                                        |
|---------------------|-------------------------------------------------------------------------|
| LOOP label          | When CX register is zero.                                               |
| LOOPE/ LOOPZ label  | When a value being checked is unequal OR when CX register becomes zero. |
| LOOPNE/LOOPNZ label | When the value being checked becomes equal or CX register become zero.  |

Example: Let us assume you have byte array of 40h bytes. Write an assembly program segment that check if each of these elements have a value 00F0h.

Solution: Please note the two conditions - the first condition is that each element should be equal to 00F0h and the second condition is loop is to be executed 40h times. Thus, LOOPE instruction would be used, but prior to that you need to set different registers. The program segment for looping is shown below:

```
; Assume that the name of the array is BYTECOST
MOV BX, OFFSET BYTECOST ; This instruction will cause the BX register to
; point at the first element of byte array BYTECOST.
DEC BX ; Decrementing the value of BX register by one.
; This will cause BX to point to one byte prior to
; BYTECOST array. Why is this instruction?
; This is due to specific loop instructions below.
; Initialise the loop counter to size of array
L1: MOV CX,40h ; Move to the next element in the array.
 INC BX ; Compare the array element to 0F0h
 CMP [BX],0F0h ; Loop if the present array element is equal to
 LOOPE L1 ; 0F0h as per CMP instruction and CX is not zero.
```

It may be noted that LOOPE instruction will automatically decrement the value of the counter CX register.

In addition to the program execution control transfer, there are string instructions which are useful for string matching. Such instructions were specially designed for 8086 microprocessor, so that it can perform faster string comparisons. Some of these instructions are discussed in the next section.

### 13.4.5 String Instructions

String based instructions in 8086 were added to allow faster processing of strings based operations. A string is a sequence of ASCII characters in a 8086 microprocessor. Most of the string instructions use a subscript B to indicate that each character in the string is of byte type and a subscript W indicates that each character of the string is of the size of a word (16 bits). Following are the some of the string instructions.

*REP, REPE/REPZ and REPNE/REPNZ:*

These keywords are used before any string instructions to repeat the following instruction to a number of times as specified in CX register. REP prefix repeats the instruction and decrements the CX register by 1, till CX becomes zero. The REPE/REPZ prefix repeats the instruction till either CX becomes zero or ZF becomes 0, whereas REPNE/REPNZ prefix repeats the instruction till either CX becomes zero or ZF becomes 1.

*MOVS/MOVSB/MOVSW instruction:*

This instruction moves data from one byte string to another byte string. This string operator uses several registers implicitly. The source string is assumed to be in data segment, indexed by SI register, whereas the destination string is assumed to be extra data segment indexed by DI register. CX is used as counter register. On transfer of one byte data from sources string to destination, automatically results in increment of SI and DI registers, and decrement of CX register.

Example: Assume that both data segment and extra data segment registers start from segment address 00FFh and a byte string of length 0100h starting at an offset 0400h is to be copied at an offset 0600h. Write the program segment to show this transfer.

; Assuming data segment and extra data segments registered are already initialised.

```
MOV CX, 0100h ; Initialize counter CX
MOV SI, 0400h ; Initialise SI
MOV DI, 0600h ; Initialise DI
REP MOVS ; Will perform transfer till CX is 0.
; SI and DI will be incremented after one byte is transferred
```

*Other string instructions:*

The following table shows other string instructions.

| Instruction          | Purpose                                                                                                                                                                                                                                           |
|----------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| CMPS/CMPSB/<br>CMPSW | This instruction compares two byte or word strings, use of CX, SI and DI remains the same as MOVS. It is recommended to use REPE in this case.                                                                                                    |
| SCAS/SCASB/<br>SCASW | This instruction compares a string with a value in AL or AX register for a byte or word string respectively. The string to be scanned is assumed to be in extra data segment. This instruction uses CX and DI registers, when REP prefix is used. |
| LODS/LODSB/<br>LODSW | This instruction is used to load a byte or word of a string pointed to by SI register into AL or AX registers respectively.                                                                                                                       |
| STOS/STOSB/<br>STOSW | This instruction is used to store a byte or word from AL or AX registers respectively into a location pointed by DI register.                                                                                                                     |

### 13.4.6 Processor Control Instructions

These instructions are used to change certain parameters that are under the control of the programmer. You can control some of the flags, which may alter the conditional jump and direction of string manipulation. The following table briefly lists some of the most used processor control instructions. You may refer to the further readings for more details on such instructions.

| Instruction | Purpose                                                                                            |
|-------------|----------------------------------------------------------------------------------------------------|
| STC         | This instruction sets the carry flag.                                                              |
| CLC         | This instruction clears the carry flag.                                                            |
| CMC         | This instruction complements or inverts the state of the carry flag.                               |
| STD         | This instruction sets the direction flag (DF=1), so the SI and DI are decremented automatically.   |
| CLD         | This instruction clears the direction flag (DF=0), so the SI and DI are incremented automatically. |

There are many other process control instructions. You may refer to further readings to know more about these instructions.

---

## 13.5 ADDRESSING MODES

---

The 8086 micro-processor supports many operating modes to address the operands. These are – Immediate addressing mode, Register addressing mode, direct addressing mode and Indirect addressing modes. These addressing modes are explained in the following sections.

### 13.5.1 Immediate Addressing Mode

Immediate addressing allows an operand to be part of the instruction. The 8086 assembly language allows you to even use expressions as part of the instructions; however, these expressions should be computable at assembly time to produce a constant value. Some of the examples of immediate source operand are given below:

```
MOV AL, 45h ; move immediate value 45h to AL
MOV AL, 'a' ; move immediate ASCII character value 'a' to AL
MOV AX, 'ba' ; move immediate ASCII characters values to AX register
MOV AL, (5+3)*2 ; move 16 to AL register.
```

### 13.5.2 Register Addressing Mode

A register addressing mode allows any of the register of 8086 to be made as an Operand. However, some special registers cannot be used in every instruction. 8086 microprocessor may allow two register operands in a single instruction. Register operands can be 16 bit registers or 8 bit registers as shown below:

16 bit Register operands: AX,BX,CX,DX,SI,DI,BP,IP,CS,DS,ES,SS

8 bit Register operands: AH, AL, BH, BL, CH, CL, DH, DL

Register operand, in general, allows faster execution of instructions. Some example of instructions using register operands is given below:

```
MOV AL, BL ; Move the content of BL register to AL register
 ; Both the register operands are of 8 bits
MOV DS, AX ; loads data segment register using 16 bit AX register
```

### 13.5.3 Direct Addressing Mode

A direct addressing mode, in general, specifies a memory location as an operand in an instruction. An 8086 instruction can have a maximum of one memory operand. Interestingly, the 8086 memory address is of 16 bits only and contains the offset of an address; therefore, these addresses are called relocatable addresses. If a program is to be reloaded in a different memory segment, it will just require to change the segment register and not the offset. Thus, programs can be relocated to any segment, without changing the instructions. Following are some of the examples of direct addressing mode:

```
MOV CL, LoopCount; loads the content of LoopCount location to CL register.
 ; Segment register used will be data segment (DS)
JMP Loop ; Jump to address specified by the label loop.
 ; Please note that segment register for Loop
 ; would be the code segment.
```

### 13.5.4 Indirect Addressing Mode

In indirect addressing modes, operands use registers to point to locations in memory. So it is actually a register indirect addressing mode. This is a useful mode for handling strings and arrays. For this mode two types of registers are used. These are:

- Base register BX, BP
- Index register SI, DI

BX register contains the offset of the location in Data Segment, whereas BP register points to the base of the stack segment register. The index registers SI and DI also contains offset in the Data Segment and Extra data Segment respectively.

These registers can be combined to create several indirect addressing modes. These are:

*Register indirect:* In this addressing mode the register contains the address of the data. In general, the type of register as stated above determines the segment in which the data is to be accessed. Examples of this mode are:

MOV AL, [DI] ; Move the byte at the memory location ES:DI to AL.  
MOV AL, [BX]; Move the byte at the memory location DS:BX to AL.

*Based indirect:* In this addressing mode a base register and a displacement are added to compute the offset of address of data in the related segment. Example of this mode are:

MOV AL, [BX+2] ; Move the byte at the memory location DS:BX+2 to AL.

*Indexed indirect:* In this addressing mode an index register and a displacement are added to compute the offset of address of data in the related segment. Example of this mode are:

MOV AL, [DI+2] ; Move the byte at the memory location ES:DI+2 to AL.

There are two more such indirect addressing modes, viz. Based Indexed and Based Indexed with displacement, however, they are rarely used and are not explained in this Unit.

### **Check Your Progress 3**

1. Why are CALL and RET statements used?

.....

.....

2. What are the different types of Jump instructions? Why are they needed?

.....

.....

.....

3. What are the different implicit operations of LOOP instruction?

.....

.....

.....

4. Why do you need the string instructions?

.....

.....

.....

5. Give one example each of each type of addressing mode of 8086 micro-processor.

.....

.....

.....

---

## **13.6 SUMMARY**

---

In this unit, you have gone through the basic architecture of 8086 microprocessor. This architecture was a creative design and used many interesting concepts related to enhancing the speed of instruction processing. First of these is the concept of use of segment registers to reduce 20 bit physical address to a 16 bit offset address, reducing the size of instruction using direct addressing, second faster string processing by using two separate segments to speed up string operations such as matching, third use of pipelining by designing two sections in CPU, fourth use of instruction queue for pre-fetching instructions and so on. 8086 assembly language forms the basis of Intel instruction sets of advanced processors and may help you appreciate the assembly language of those processors.

Some of the key features of this processor include:

- It has 20 bit address bus, therefore, base memory is 1 MB
- It has 16 bit data bus, thus can fetch two bytes simultaneously
- It has four segment registers that along with other pointer registers converts 16 bit offsets to 20 bit physical address.
- It has large number of instructions of different types, which allows writing of powerful assembly programs.

Please refer to the further reading for more details on 8086 assembly language programming.

## 13.7 SOLUTIONS/ANSWERS

### Check Your Progress 1

1. 8086 microprocessor has a Bus interface unit, which is a dedicated unit to compute memory address for reading/ writing a byte or word from/to the memory. It consists of a dedicated adder circuit, which converts 16 bit offset and content of 16 bit segment register to a 20 bit physical address. It also has a 6 byte instruction queue, which can store more than one instruction at a time. The execution unit performs the arithmetic, logical, shift, call, test and many other operations on data. It also contains registers, which store data and temporary results.
2. (a)

|                                       |          |          |          |          |
|---------------------------------------|----------|----------|----------|----------|
| CS (in hexadecimal)                   | 0        | 1        | 1        | 1        |
| Shift left by one Hexadecimal digit   | 0        | 1        | 1        | 0        |
| IP (in hexadecimal)                   | 0        | 0        | 2        | 0        |
| <b>Physical address (Hexadecimal)</b> | <b>0</b> | <b>1</b> | <b>1</b> | <b>0</b> |

(b)

|                                       |          |          |          |          |
|---------------------------------------|----------|----------|----------|----------|
| DS (in hexadecimal)                   | 0        | 2        | 1        | 1        |
| Shift left by one Hexadecimal digit   | 0        | 2        | 1        | 0        |
| BX (in hexadecimal)                   | 0        | 1        | 0        | 0        |
| <b>Physical address (Hexadecimal)</b> | <b>0</b> | <b>2</b> | <b>2</b> | <b>1</b> |

(c)

|                                       |          |          |          |          |
|---------------------------------------|----------|----------|----------|----------|
| SS (in hexadecimal)                   | 4        | 2        | A        | A        |
| Shift left by one Hexadecimal digit   | 4        | 2        | A        | 0        |
| SP (in hexadecimal)                   | 0        | 1        | 2        | 3        |
| <b>Physical address (Hexadecimal)</b> | <b>4</b> | <b>2</b> | <b>B</b> | <b>C</b> |

3. Flag register is used to store all the flag bits, which are generated as a result of last instruction. Some of these flags are sign flag, carry flag, overflow flag etc.  
Flag register cannot be used as a general purpose register.

### **Check Your Progress 2**

1. (a) POPF instruction does not take any explicit operand.  
 (b) Move instruction has a source and destination  
 (c) An instruction cannot have two memory operands in 8086 microprocessor  
 (d) DAA instruction has an implicit operand only. No explicit operand is to specified.  
 (e) In DIV instruction you need to specify one operand only. The other operand is explicit.
2. SHL is shift left instruction and identical to arithmetic shift left instruction.  
 Compare the different types of shift instructions of 8086 micro-processor.  
 However, SHR and SAL differ and different input is added to the left most bit.  
 Rotate instruction ROL and ROR just rotates the word/byte, whereas RCL and RCR also rotate the sign bit. (Please refer to section 13.4.3).
3. Perform test instruction on the operands (please make sure both the operands are not memory operand). If it sets the zero flag, then both the operands are same; otherwise they are different.

### **Check Your Progress 3**

1. CALL statement calls a subroutine, i.e. the next instruction to be executed by the processor should be the first instruction of the subroutine. Since on completion of the subroutine execution the next instruction of the calling program is to execute therefore the return address is stored by the CALL instruction. RET instruction just brings the control back the the next instruction after CALL instruction in the calling program.
2. There are primarily two types of jump instructions: unconditional jump and conditional jumps. The unconditional jump instruction causes a compulsory jump to specified label. There are a number of conditional jump instructions, where a jump is taken if the related condition is fulfilled; else next instruction in sequence is executed.
3. Loop instruction in each iteration decrements CX register, and checks the value of CX. In case it is not zero, you go back to the Label from where the loop started. However, if the CX register is zero, the next instruction in sequence is executed.
4. String instructions in 8086 microprocessor are specially designed for efficient execution of string operations. For example, to match two strings, one string each be put in DS and ES with DS:SI pointing to first string and ES:DI pointing to second string. String length is put in CX register. The string matching instruction on using REPE command will compare the first byte and will increment SI and DI; and decrement CX. Thus, you do not need to write lengthy program for string matching, which includes all the operation as given above.
5. Immediate Operand

MOV AL, (9+7)\*2 ; move 32 to AL register.

Register Addressing

MOV AL, DL ; move DL to AL register.

Direct Addressing

MOV AL, X ; move content of byte location X to AL register.

Register Addressing

MOV AH, [BX] ; move content of location, whose address is ; DS:BX to AL register.

---

# **UNIT 14 INTRODUCTION TO ASSEMBLY LANGUAGE PROGRAMMING**

---

| <b>Structure</b>                               | <b>Page No.</b> |
|------------------------------------------------|-----------------|
| 14.0 Introduction                              |                 |
| 14.1 Objectives                                |                 |
| 14.2 The Need and Use of the Assembly Language |                 |
| 14.3 Assembly Program Execution                |                 |
| 14.4 An Assembly Program and its Components    |                 |
| 14.4.1 The Program Annotation                  |                 |
| 14.4.2 Directives                              |                 |
| 14.5 Input Output in Assembly Program          |                 |
| 14.5.1 Interrupts                              |                 |
| 14.5.2 DOS Function Calls (Using INT 21H)      |                 |
| 14.6 The Types of Assembly Programs            |                 |
| 14.6.1 COM Programs                            |                 |
| 14.6.2 EXE Programs                            |                 |
| 14.7 How to Write Good Assembly Programs       |                 |
| 14.8 Summary                                   |                 |
| 14.9 Solutions/Answers                         |                 |
| 14.10 Further Readings                         |                 |

---

## **14.0 INTRODUCTION**

---

In the previous unit, you have gone through the basic concepts of 8086 microprocessor, which included the 8086 structure, segmentation, register set, instructions and addressing modes. This unit present a basic framework for writing assembly language programs for 8086 microprocessor. In this unit, you will learn about the importance, basic components and development tools of assembly language programming. The Input/Output to an assembly language program is a complex process. This unit discusses the Input/Output to assembly program by using interrupts. This unit also discussed about different kinds of Assembly programs, viz. COM programs and EXE programs. Finally, the unit presents an example assembly program. An assembly program consists of assembler directives and instructions of 8086 microprocessor. This program is assembled using an assembler program. Several such assembler programs exist, which use different assembler directives. We have used the assembler directives, as used in Microsoft Assembler (MASM). However, these directives may be different for different assemblers. Therefore, before running an assembly program you must consult the reference manuals of the assembler you are using and change directives accordingly.

---

## **14.1 OBJECTIVES**

---

After going through this unit, you should be able to:

- define the need and importance of an assembly program;
- use basic directives for writing an assembly program;
- use interrupts to perform input/ output in an assembly program;
- define and differentiate between COM and EXE programs.

## 14.2 THE NEED AND USE OF THE ASSEMBLY LANGUAGE

The computer instructions are a sequence of 0's and 1's. These sequences consist of instruction operation code, addressing modes and operand addresses. The instructions of the programs written in the machine language are directly decoded by processing unit. However, you may have to face the following problems, if you program using machine language:

- Machine Language depends on machine instruction set and is difficult for most people to write in 0-1 forms.
- Debugging or correcting a machine language program is difficult.
- Deciphering the machine code is very difficult. Thus, program logic of programs written in machine language will be difficult to understand.

To overcome these difficulties computer manufacturers have devised English-like words to represent the binary instructions of a machine. This symbolic code for each instruction is called a mnemonic. The mnemonic for a particular instruction consists of letters that suggest the operation to be performed by that instruction. For example, ADD mnemonic is used for adding two numbers. Using these mnemonics machine language instructions can be written in symbolic form with each machine instruction represented by one equivalent symbolic instruction. This is called an assembly language.

### Pros and Cons of Assembly Language

The following are some of the advantages / disadvantages of using assembly language:

- Assembly Language provides better control over handling particular hardware and software, as it allows you to study the instructions set, addressing modes, interrupts etc.
- Assembly Programming generates smaller, more compact executable execution module. An assembly instruction is directly translated to a machine instruction, therefore, assembly programs are highly optimized. This results in faster execution of programs.

Assembly language programs are at least 30% denser than the same programs written in high-level language. The reason for this is that the compilers produce a long list of code for every instruction as compared to assembly language, which produces single line code for a single instruction. Further, complex instructions of a computer, like string instruction of 8086 highly optimized, can be used while writing an assembly program, making program faster. On the other hand, unlike high level languages, assembly language is machine dependent. Each microprocessor has its own set of instructions. Thus, assembly programs are not portable.

Assembly language has very few restrictions or rules; nearly everything is left to the discretion of the programmer. This gives lots of freedom to programmers to write good logic of a program

### Uses of Assembly Language

Assembly language is used primarily for writing short, specific, efficient interfacing modules/ subroutines. The basic idea of using assembly is to support the High level languages with some highly efficient but non-portable routines. It will be worth mentioning here that UNIX is mostly written in C, a high-level language, but it has

## 14.3 ASSEMBLY PROGRAM EXECUTION

An assembly program is written according to a strict set of rules. An editor or word processor is used for keying an assembly program into the computer, as a file, and then an assembler program is used to translate the assembly language program into machine code. The symbolic instructions that you code in assembly language is known as - Source program. An assembler program translates the source program into machine code, which is known as object program (refer to Figure 14.1).

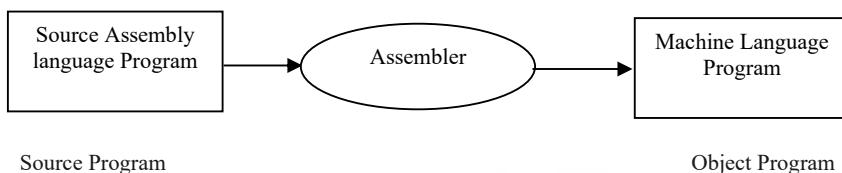


Figure 14.1: Use of assembler

Subsequently the object program is linked and an executable program is created. These steps are explained below:

Step 1: The assembly step involves translating the source code into object code and generating an intermediate .OBJ (object file) or module. The assembler also generates a header in front of the .OBJ module; part of the header contains information about incomplete addresses in the object module. The .OBJ module is not an executable form.

Step 2: The link step involves converting the .OBJ module to an .EXE machine code module. The linker completes any address left open by the assembler and combines separately assembled programs into one executable module. In addition, it also initializes the .EXE module with special instructions to facilitate its subsequent loading of the .EXE program into the computer memory for execution.

Step 3: The last step is to load the program for execution. Because the loader knows where the program is going to be loaded in the memory, it is able to resolve all the remaining incomplete addresses in the header. The loader drops the header and creates a program segment prefix just before the program is loaded in memory.

These steps are shown in Figure 14.2

## Assembly Language Programming

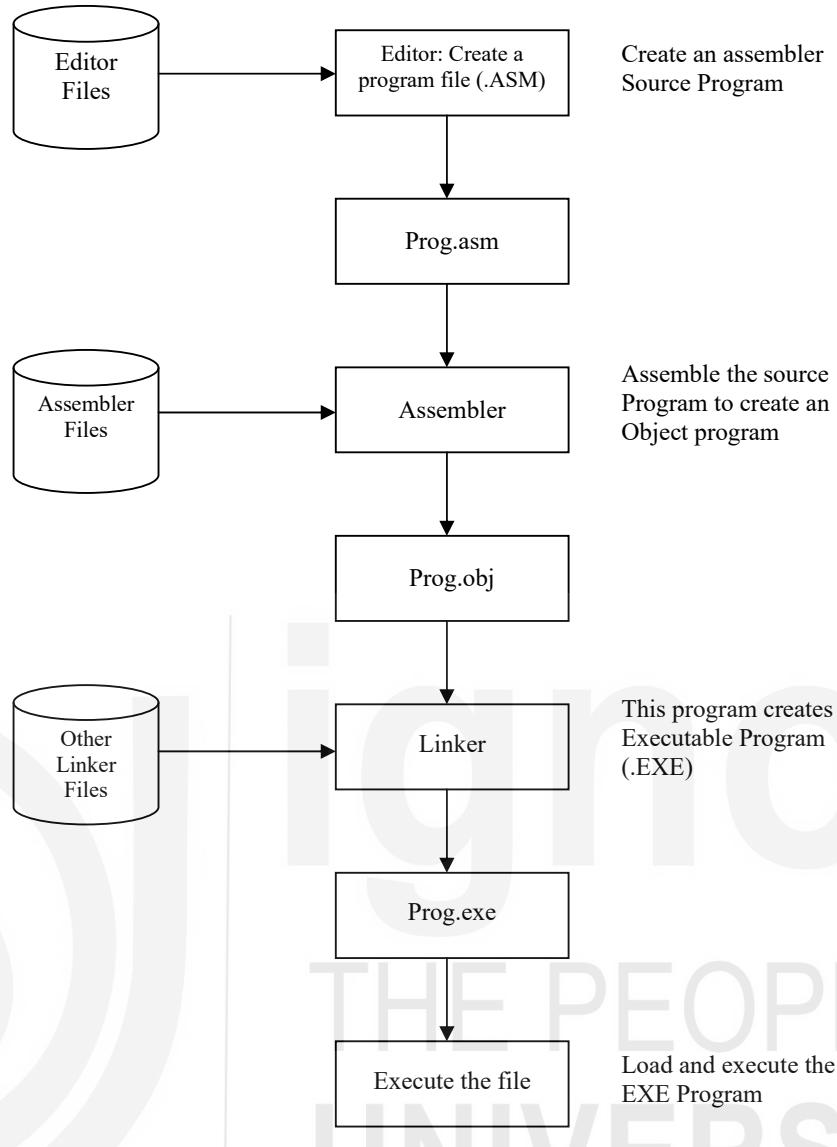


Figure 14.2: Program Assembly

**Tools required for assembly language programming:** Following are some of the basic tools needed to create assembly program. A modern-day assembler may contain several of these tools.

**Editor:** The editor is a program that allows the user to enter, modify, and store a group of instructions or text under a file name. The editor program creates an ASCII file. A common line editor program is NOTEPAD in Windows; vi editor in UNIX etc. An editor program may be part of assembler itself. You should use proper syntax of the assembly instructions to create an 8086-assembly program.

**Assembler:** An assembly program consists of assembly language instructions, which consists of assembly language mnemonics. The editor program, as defined above, is used to input the assembly language program and save it as a text file. An assembler is a program that converts the assembly language program, stored in a text file, into an equivalent machine language program. In general, this conversion is performed in two phases: first the assembler reads the assembly language file to collect various symbols used by the program along with their offsets in symbol table. On the second pass, it produces binary code for each instruction of the program and assigns an offset to all the symbols in the symbol table with respect to the segment base.

The assembler generates three files when your program gets successfully assembled with no errors. These three files are the object file, the list file and cross reference file. The object file contains the binary code for each instruction in the program. The errors that are detected by the assembler are called the symbol errors. For example, in the following statement the mnemonic MOVE is compared by assembler to all the mnemonics of the mnemonic set. It fails to get a match following which it assumes MOVE to be an identifier and looks for its entry in the symbol table. It does not find it there too, therefore gives an error “undeclared identifier”.

```
MOVE AX1, ZX1 ;
```

List file is optional and contains the source code, the binary equivalent of each instruction, and the offsets of the symbols in the program. This file is for purely documentation purposes. Some of the historical assemblers available on PCs are MASM, TURBO assembler etc.

**Linker:** For better modularity programs are broken into several sub routines. It is even better to design common routine, like reading a hexadecimal number, writing hexadecimal number etc., which could be used by a lot of other programs. These common routines can be put into files and assembled separately. After each file has been successfully assembled, they can be linked together to form a large file, which constitutes your complete program. The file containing the common routines can be linked to your other program also. The program that links your program is called the linker.

The linker produces a linked file, which contains the binary code for all component modules. The linker also produces link map, which contains the address information about the linked files. The linker, however, does not assign absolute or physical addresses to your program. It only assigns continuous relative addresses to all the modules linked starting from the zero. Since these programs uses just relative addresses, they can be loaded in any physical memory address. Thus, these programs are called relocatable programs.

**Loader:** The basic purpose of the loader program is to convert the logical or relative addresses assigned by linker to absolute or physical memory addresses. This task is performed, while loader loads the linked program into the physical memory for execution. The linked program is brought from the secondary memory, like disk, to the computer memory for execution. The file name extension of the files for loading is .EXE or .COM, which after loading can be executed by the CPU.

**Debugger:** The debugger is a program that allows the user to test and debug the object file. The user can employ this program to perform the following functions:

- Make changes in the object code.
- Examine and modify the contents of memory.
- Set breakpoints, execute a segment of the program and display contents of the register after the execution.
- It traces the execution of the specified segment of the program and displays the register and memory contents after the execution of each instruction.
- Disassemble a section of the program, i.e., converts the object code into the source code or mnemonics.

In summary, to run an assembly program you may require your computer:

- A word processor like notepad
- MASM, TASM, Emulator of 8086, or any other assembler
- Linker, which may be included in the assembler
- Debugger for debugging, if the need so be.

## Errors

Two possible kinds of errors can occur in assembly programs:

- Programming errors: They are the errors you can encounter in the course of executing a program written in any high-level language, like syntax errors and semantic error
- System errors: These are unique to assembly language that permit low-level operations. A system error is one that corrupts or destroys the system under which the program is running - In assembly language there is no supervising interpreter or compiler to prevent a program from erasing itself or even from erasing the computer operating system. Therefore, assembly program should be tested in a safe mode.

---

## 14.4 AN ASSEMBLY PROGRAM AND ITS COMPONENTS

---

In this section, a simple assembly program is shown and its various components are explained. Consider the following program:

| <i>Line Numbers</i> | <i>Offset</i> | <i>Source Code</i>                           |                              |
|---------------------|---------------|----------------------------------------------|------------------------------|
| 0001                |               | DATA SEGMENT                                 |                              |
| 0002                | 0000          | MESSAGE DB "Assembly Language Programming\$" |                              |
| 0003                |               | DATA ENDS                                    |                              |
| 0004                |               | STACK SEGMENT                                |                              |
| 0005                |               | STACK 0400H                                  |                              |
| 0006                |               | STACK ENDS                                   |                              |
| 0007                |               | CODE SEGMENT                                 |                              |
| 0008                |               | ASSUME CS: CODE, DS: DATA SS: STACK          |                              |
| 0009; Offset        |               | <i>MachineCode</i>                           | <i>Assembly Instructions</i> |
| 0010                | 0000          | B8XXXX                                       | MOV AX, DATA                 |
| 0011                | 0003          | 8ED8                                         | MOV DS, AX                   |
| 0012                | 0005          | BAXXXX                                       | MOV DX, OFFSET MESSAGE       |
| 0013                | 0008          | B409                                         | MOV AH, 09H                  |
| 0014                | 000A          | CD21                                         | INT 21H                      |
| 0015                | 000C          | B8004C                                       | MOV AX, 4C00H                |
| 0016                | 000F          | CD21                                         | INT 21H                      |
| 0017                |               | CODE ENDS                                    |                              |
| 0018                |               | END                                          |                              |

Program1: A simple 8086 assembly language program

The details of this program are explained in section 14.4.1.

### 14.4.1 The Program Annotation

The program annotation consists of 3 columns of data: line numbers, offset and machine code.

- The assembler assigns line numbers to the statements in the source file sequentially. If the assembler issues an error message; the message will contain a reference to one of these line numbers.
- The second column from the left contains offsets. Each offset indicates the address of an instruction or a datum as an offset from the base of its logical segment, e.g., the statement at line 0010 produces machine language at offset

0000H of the CODE SEGMENT and the statement at line number 0002 produces machine language at offset 0000H of the DATA SEGMENT.

- The third column in the annotation displays the machine language produce by code instruction in the program.

**Segment names:** Segment name is specified by the programmer. It allows programs written in 8086 assembly language to be relocatable. They can be loaded practically anywhere in memory and run just as well. Program1 has to store the message “Assembly Language Programming\$” somewhere in memory. It is located in the DATA SEGMENT. Since the characters are stored in ASCII, therefore it will occupy 30 bytes (please note each blank is also a character) in the DATA SEGMENT.

**Missing offset:** The XXXX in the machine language for the instruction at line 0010 is there because the assembler does not know the DATA segment location that will be determined at loading time. The loader must supply that value.

### Program Source Code

Each assembly language statement appears as:

{identifier} Keyword {{parameter},} {;comment}.

The element of a statement must appear in the appropriate order, but significance is attached to the column in which an element begins. Each statement must end with a new line character.

**Keyword:** A keyword is a statement that defines the nature of that statement. If the statement is a directive, then the keyword will be the title of that directive; if the statement is a data-allocation statement the keyword will be a data definition type. Some examples of the keywords are: SEGMENT (directive), MOV (statement) etc.

**Identifiers:** An identifier is the name of an item, given by you, in your program that you expect to reference. The two types of identifiers are name and label.

1. Name refers to the address of a data item such as counter, arr etc.
2. Label refers to the address of our instruction, process or segment. For example

the statement  
A20: BL,45 ; defines a label A20.

Identifier can use alphabet, digit or special character. It always starts with an alphabet.

**Parameters:** A parameter extends and refines the meaning that the assembler attributes to the keyword in a statement. The number of parameters is dependent on the Statement.

**Comments:** A comment is a string of a text that serves only as internal document action for a program. A semicolon identifies all subsequent text in a statement as a comment.

### 14.4.2 Directives

Assembly languages support a number of directive statements. Directives enable you to control the way in which a source program assembles and lists. Directives act only when the assembly is in progress and generate no machine-executable code. Let us discuss some common directives.

1. **HEX:** The HEX directive facilitates the coding of hexadecimal values in the body of the program. This statement directs the assembler to treat related tokens in the source file as numeric constants in hexadecimal notation.
2. **PROC Directive:** PROC directive can be used in the code segment of an assembly program to create a procedure. You can use more than one PROC directives in a code segment. A PROC directive marks the start of a procedure. The end of a procedure is marked by the ENDP directive. The following example shows the start and end of a procedure named CheckZero.  
CheckZero PROC NEAR ; Beginning of a Procedure in same segment  
...  
CheckZero ENDP NEAR ; Marks the end of the Procedure CheckZero
3. **END DIRECTIVE:** There are three different END directives. These are:
  - (i) ENDS Directive: This directive marks the completion of a segment. Thus, every segment used by you must have an ENDS directive.
  - (ii) ENDP directive: As stated in point 2 it is used to mark the end of a procedure.
  - (iii) END directive: It marks the end of the entire program. Any statement after this directive is ignored by the assembler.
4. **ASSUME Directive:** The purpose of assume directive is to identify the possible use of various segments defined in an assembly program. For example, if in your assembly program you have defined segments named STRING, CODE and STACK, which are to be used as data segment, code segment and stack segment respectively, then you can use the following ASSUME directive statement  
`ASSUME CS: CODE, DS: STRING, SS: STACK`
5. **SEGMENT Directive:** The segment directive defines the segment name. A segment directive makes it possible to set a segment register to address the base address of a segment register (Please refer to discussion on segment register in Unit 13). All the offsets in a segment are computed from a base address of a segment. A segment directive indicates to assemble all statements following it in a single source file until an ENDS directive.

### CODE SEGMENT

The code segment contains the code of the program, which may include procedures and sometimes other segments too. Linker marks the code segment in a program in a header. This header is used by the operating system when it invokes the loader to load an executable file of the program into memory. The loader reads this header for setting the CS register. A physical memory address is represented as CS: xxxx, where xxxx represents the offset in the code segment. In general, the first instruction of the code segment is assumed as the first instruction to be executed, therefore, is put at an offset of 0000H. The instruction pointer (IP) register is used to mark the offset of an instruction in code segment. The CS: IP pair is thus used to specify physical address of an instruction in a program that is being executed.

### STACK SEGMENT

8086Microprocessor supports the **Word stack**. The stack segment parameters tell the assembler to alert the linker that this segment statement defines the program stack area.

A program must have a stack area in that the computer is continuously carrying on several background operations that are completely transparent, even to an assembly language programmer, for example, a real time clock issues a real time clock interrupts after every 55 milliseconds. Every 55 ms the CPU is

interrupted. The CPU records the state of its registers and then goes about updating the system clock. When it finishes servicing the system clock, it has to restore the registers and go back to doing whatever it was doing before the occurrence of interrupt. All such information gets recorded in the stack. Please note if you have not specified the stack segment it is automatically created. Why is stack segment essential? Consider your program is being executed by CPU, and a clock pulse need service, then if the system has no stack, then your CPU will not be able to return to your program again after serving of the clock pulse.

## DATA SEGMENT

It contains the data allocation statements for a program. This segment is very useful as it shows the data organization.

### Defining Types of Data

The following format is used for defining data definition:

*Format for data definition:*

{Name} <Directive><expression>

Name - a program references the data item through the name although it is optional.

Directive: Specifies the data type to assemble.

Expression: Represent a value or evaluated to value.

The list of directives are given below:

| Directive | Description        | Number of Bytes |
|-----------|--------------------|-----------------|
| DB        | Define byte        | 1               |
| DW        | Define word        | 2               |
| DD        | Define double word | 4               |
| DQ        | Define Quad word   | 8               |
| DT        | Define 10 bytes    | 10              |

**DUP** Directive is used to duplicate the basic data definition to 'n' number of times

ARRAY            DB            10 DUP (0)

In the above statement ARRAY is the name of the data item, which is of byte type (DB). This array contains 10 duplicate zero values; i.e. 10 zero values.

**EQU** directive is used to define a name to a constant

CONST            EQU            20

The above statement defines a name CONST to a value 20.

Type of number used in data statements can be octal, binary, hexadecimal, decimal and ASCII.

Some other examples of using these directives are:

|               |                                                         |
|---------------|---------------------------------------------------------|
| BINDB0111001B | ; Binary value in byte operand named temp               |
| OCT DW 7341Q  | ; Octal value assigned to word variable named VALI      |
| DECDB49       | ; Decimal value 49 contained in byte variable names DEC |
| HEXDW03B2AH   | ; Hexadecimal value a is stored in a word operand HEX.  |

**☛ Check Your Progress 1**

1. Why should we learn assembly language?

.....  
.....  
.....

2. What is a segment? Write all four main segment names.

.....  
.....  
.....

3. State True or False.

|   |   |
|---|---|
| T | F |
|---|---|

- (a) The directive DT defines a quadword in the memory
- (b) DUP directive is used to indicate if a same memory location is used by two different variables name.
- (c) EQU directive assign a name to a constant value.
- (d) The maximum number of active segments at a time in 8086 can be four.
- (e) ASSUME directive specifies the physical address for the data values of instruction.
- (f) A statement after the END directive is ignored by the assembler.

---

## 14.5 INPUT OUTPUT IN ASSEMBLY PROGRAM

---

A software interrupt is a call to an Interrupt servicing program located in the operating system. Usually the input-output routine in 8086 is performed using these interrupts.

### 14.5.1 Interrupts

An **interrupt** causes interruption of an ongoing program. Some of the common interrupts are caused by devices like keyboard, printer, monitor, an error condition, etc.

8086 recognizes two kinds of interrupts: **Hardware** interrupts and **Software** interrupts.

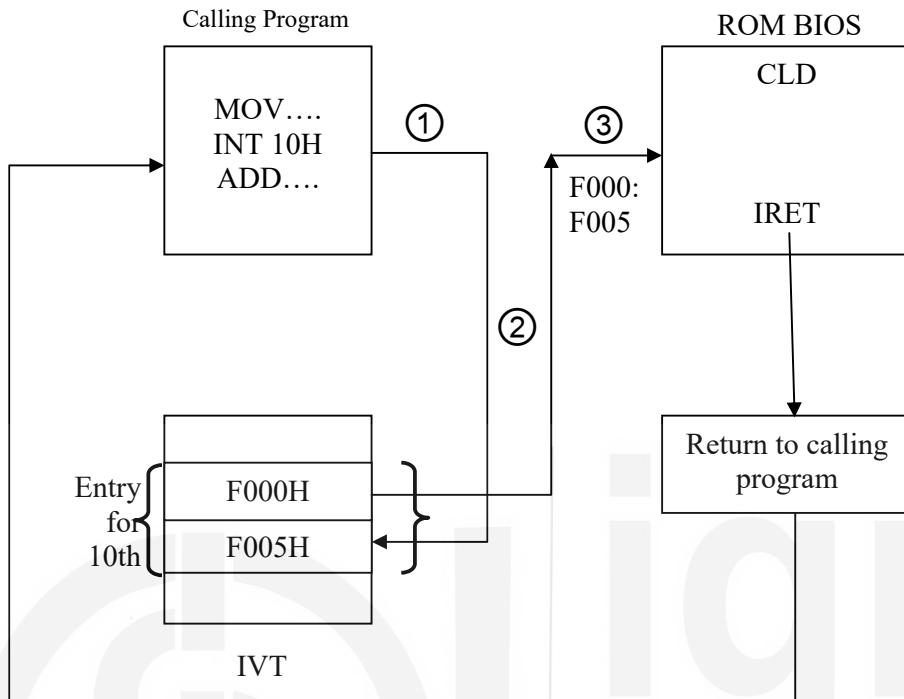
Hardware interrupts are generated by a device that requests for some service. A software interrupt causes a call to the operating system. It usually is the **input-output** routine.

In 8086 software interrupts can be used for input-output of data. A software interrupt is initiated using the following statements:

INT      number

In 8086, this interrupt instruction is processing using an **interrupt vector table (IVT)**. The IVT is located in the first 1K bytes of memory, and has a total of 256 entries,

each of 4 bytes. The entry stores the address of the operating system subroutine that is used to process the interrupt. This address may be different for different machines. Figure 14.3 shows the processing of an interrupt.



**Figure 14.3: Processing of an Interrupt**

The interrupt is processed as:

**Step 1:** The “number” field in INT instruction is multiplied by 4 to find its entry in the interrupt vector table. For example, the IVT entry for instruction INT 10H would be found at IVT at an address 40H. Similarly, the entry of INT 3H will be at an address 0CH.

**Step 2:** The CPU locates the interrupt servicing routine (ISR) whose address is stored at IVT entry of the interrupt. For example, in the figure above the ISR of INT 10H is stored at address (CS: IP) as F000h:F065h

**Step 3:** The CPU loads the CS register and the IP register, with this new address in the IVT, and starts instruction execution process for that instruction.

**Step 4:** IRET (interrupt return) causes the program to resume execution of the next instruction of the program, which was being executed prior to interrupt servicing.

### Keyboard Input and Video output

A Keystroke read from the keyboard is called a console input and a character displayed on the video screen is called a console output. In assembly language, reading and displaying a character is a difficult program. However, these tasks were simplified by the architecture of the 8086. This architecture provides a pack of software interrupt vectors beginning at address 0000h:0000h, i.e., start of IVT.

The advantage of this type of call is that it appears static to a programmer but flexible to a system design engineer. For example, INT 00H is a special system level vector

that points to the “recovery from division by zero” subroutine. If new designer come and want to move interrupt location in memory, it adjusts the entry in the IVT vector of interrupt 00H to a new location. Thus, from the system programmer point of view, it is relatively easy to change the vectors under program control.

One of the commonly used Interrupts for Input /Output is called DOS function call. Let us discuss more about it in the next subsection:

#### 14.5.2 DOS Function Calls (Using INT 21H)

INT 21H supports many different functions. A function is identified by putting the function number value in the AH register. For example, if you want to call function number 01H, then you place this value 01h in AH register first by using MOV instruction, then you may call INT 21H:

Some important DOS function calls are given in the following table:

| DOS Function Call | Purpose and Example                                                                                                                                                                                                                                                                                                                                                                                                                         |
|-------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| AH = 01H          | <p>This function called is used for reading a single character from keyboard and displaying it on monitor. The input value is put in AL register. For example, to read a character in a memory location X, you may use the following code fragment:</p> <pre>MOV AH, 01H ; load AH register with the function value 01h INT 21H; call the interrupt to read a character in AL MOV X, AL; Load the read character in memory location X</pre> |
| AH = 02H          | <p>This function prints an 8-bit data (normally ASCII), which is stored in DL register, on the screen. For example, to print a character ‘?’ on the monitor, you may write the following code fragment:</p> <pre>MOV AH, 02H ; load AH register with the function value 02h MOV DL, '?' ; Move the character to be displayed in DL INT 21H; call the interrupt to display the character in DL</pre>                                         |
| AH = 08H          | <p>This function is also used to input a single character into AL register, except that the character does not get displayed on the monitor. For example, to read a character in a memory location X, you may use the following code fragment:</p> <pre>MOV AH, 08H ; load AH register with the function value 08h INT 21H; call the interrupt to read a character in AL MOV X, AL; Load the read character in memory location X</pre>      |
| AH = 09H          | <p>This function outputs a string whose offset is stored in DX register. The string is terminated by using a \$ character. You can use this function to print newline character, tab character etc. For example, to print a string “Hello World”, you may use the following code fragment:</p> <pre>DATA SEGMENT     STRING DB 'HELLO WORLD', '\$' DATA ENDS CODE SEGMENT     ... END</pre>                                                 |

|          |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |    |   |   |   |   |   |     |   |     |   |
|----------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----|---|---|---|---|---|-----|---|-----|---|
|          | <pre> MOV AX, DATA; Put offset of Data Segment to AX. MOV DS, AX; Initialize data segment register using AX MOV AH, 09H; load AH with the function value 09h MOV DX, OFFSET STRING; Store the offset of STRING in DX INT 21H ; Call interrupt 21H to display the STRING ... </pre>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |    |   |   |   |   |   |     |   |     |   |
| AH = 0AH | <p>For input of string up to 255 characters. The string is stored in a buffer. For example, the following data and code fragment will input a string having a maximum length of 50 bytes. First, you need to define these parameters in the data segment, as given below:</p> <pre> DATA SEGMENT     BUFFDB50     DB?     DB 50 DUP (0) DATA ENDS </pre> <p>The name of the data segment, as given above, is DATA. It consists of total 52 bytes locations named BUFF. The first location of BUFF stores the decimal value 50, which is the maximum size of the string that can be stored in this buffer. The second location, marked with '?', will be used to store the actual size of the string, once it is read in the buffer. The remaining 50 bytes at present are initialized as 0. These bytes will contain the string once it is read. The code segment, which will perform the string read operation is given below:</p> <pre> CODE SEGMENT     ...     MOV AH, 0AH ; Move 0A to AH register     MOV DX, OFFSET BUFF; DX contains offset of BUFF     INT 21H ; Call interrupt 21h     ... CODE ENDS </pre> <p>For the given code (complete the other necessary directives and statements) and data segment, if you input a value “Parv”, then it will be stored in the BUFF as given below:</p> <table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>50</td> <td>4</td> <td>P</td> <td>a</td> <td>r</td> <td>v</td> <td>0</td> <td>0</td> <td>...</td> <td>0</td> </tr> </table> | 50 | 4 | P | a | r | v | 0   | 0 | ... | 0 |
| 50       | 4                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 | P  | a | r | v | 0 | 0 | ... | 0 |     |   |
| AH = 4CH | This function call returns control back to the operating system.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |    |   |   |   |   |   |     |   |     |   |

### Some examples of Input

- (i) **Input a single ASCII character to BL register, without displaying it on screen**

```

CODE SEGMENT
:
MOV AH, 08H ; AH is loaded with function 08H

```

```
INT 21H ; Issue the Interrupt 21h
MOV BL, AL ; Transfer the input obtained in AL to BL
:
CODE ENDS
```

**(ii) Input a Single Digit for example (0,1, 2, 3, 4, 5, 6, 7, 8, 9)**

```
CODE SEGMENT
:
MOV AH, 01H
INT 21H
MOV BL, AL
SUB BL, '0' ; or use SUBBL, 30H
:
CODE ENDS.
```

**Description:** First you move the value of function 01H to AH register, next you call the Interrupt 21H. Execution of these two statements will result in input of a single character in AL register and display of that character on the monitor. You expect this character to be any digit amongst 0 to 9 (you can check this with additional assembly code, if needed.). The ASCII equivalent values of these digits ‘0’ to digit ‘9’ are 30H to 39H. The input value first is moved from AL to BL register and thereafter you either subtract ‘0’, which is ASCII value of digit 0. Alternatively, can subtract 30H from BL, which is same as the hexadecimal value of digit ‘0’. This subtraction will result in binary equivalent value of the input digit in the BL register. For example, if you had input digit ‘8’ as input, then BL register will contain: 38H-30H= 08H. This binary value then can be used for arithmetic operations.

**(iii) Input numbers like (10, 11.....99)**

One of the methods to input two-digit number would be to input two single digit numbers and using the place value of these digits convert them to a two-digit binary equivalent number. For example, to input a two-digit number 48, you will first input digit 4, convert it to binary and then input digit 8, which is also converted to binary. Now, perform the following computation to get an equivalent binary number into the specific register:

$$4*10+8 = 48$$

The assembly code for the same is given below:

```
CODE SEGMENT
:
MOV AH, 08H ; Function 08H
INT 21H ; Interrupt 21H
MOV BL, AL ; Move the value to BL register
SUB BL, '0' ; Subtract '0' to get binary in BL
MOV AH, 08H ; Now, start input of second digit
INT 21H ;
MOV DL, AL ; Store AL in DL register
SUB DL, '0' ; get the second binary digit in DL
MOV AL, 0AH ; Move value 10 in AL register
MUL BL ; Multiply AL by BL.
ADD BL, DL
:
CODE ENDS.
```

**Description:** The code first input the two digits in BL and DL registers respectively, for example, an input 4 will be moved to BL and 8 will be moved to DL register, where they are converted to equivalent binary using subtraction. Next, BL is multiplied by 10, which is moved to AL register and added with the

value of DL, resulting in  $4*10+8$ . This output is stored as binary value in BL register.

### **Examples of output on Display unit**

#### **(i) Displaying a single character**

The following code displays the ASCII equivalent character of the data stored in BL register.

```
CODE SEGMENT
:
MOV AH, 02H ; The single character output function
MOV DL, BL ; Move the character to DL
INT 21H
:
CODE ENDS .
```

#### **(ii) Displaying a single digit (0 to 9)**

The following program first converts the binary value to equivalent decimal digit and then outputs it on the monitor. For example, if BL register contains a value ‘0000 0111’, which is 07H, it will be converted to digit 7 by adding 30H or ‘0’ and moved to DL register so that interrupt instruction displays this value on the monitor. The following code will perform the display as stated above:

```
CODE SEGMENT
:
ADD BL, '0'
MOV DL, BL
MOV AH, 02H
INT 21H
:
CODE ENDS .
```

#### **(iii) Displaying a number (10 to 99)**

Assuming that the two-digit number 48 is stored as number 4 in BH and number 8 in BL. Each of these digits is converted to its equivalent ASCII digit by adding 30H or ‘0’. This is followed by displaying the equivalent ASCII of the digits respectively to output the given number.

```
CODE SEGMENT
:
ADD BH, '0'
ADD BL, '0'
MOV AH, 02H
MOV DL, BH
INT 21H
MOV DL, BH
INT 21H
:
CODE ENDS
```

#### **(iv) Displaying a string**

Assume that a string is stored in the data segment with the label ST1. To display this string the following code can be used:

```
DATA SEGMENT
ST1 DB "Output a string$"
```

```
DATA ENDS
CODE SEGMENT
:
MOV DX, OFFSET ST1
MOV AH, 09H
INT 21H
:
CODE ENDS.
```

**An example with Input and output both:**

To write a program that accepts an input character from the keyboard and respond.  
“The input is \_”.

```
DATA SEGMENT
MESSAGE DB "The input is$"
DATA ENDS
CODE SEGMENT
MOV AX, DATA; Move data segment address to AX
MOV DS, AX ; Initialize DS register
MOV AH, 08H ; Set function for character read
INT 21H ; Read character in AL
MOV BL, AL ; Move input to BL
MOV AH, 09H ; Function to display strings
MOV DX, OFFSET MESSAGE ; Move offset of string to DX
INT 21 H ; Display string named MESSAGE
MOV AH, 02H; Function to display character
MOV DL, BL ; Move character to DL
INT 21 H ; Display character
MOV AX, 4C00H ; Move 4CH to AH (DOS function call)
INT 21H ; Exit to DOS
CODE ENDS
END.
```

**Check Your Progress 2:**

Q1: List the interrupts that can be used to input one character.

---

---

Q2: What is the output of following code segment, assume that BL register contains the binary value 0000 0010<sub>2</sub>

```
CODE SEGMENT
:
ADD BL, '0'
MOV DL, BL
MOV AH, 02H
INT 21H
:
CODE ENDS.
```

---

---

Q3: Name the interrupt used to exit to operation to operating system.

## 14.6 THE TYPES OF ASSEMBLY PROGRAMS

Assembly language programs can be written in two ways:

- COM Programs: A .COM program has only one physical segment, which includes all the different segments.
- EXE Program: An EXE program consists of separate segments.

Let us look into brief details of these programs.

### 14.6.1 COM Programs

A COM (Command) program is the binary image of a machine language program. It is loaded in the memory at the lowest available segment address. In 8086 micro-processor, a COM program code begins at an offset 100h, as the first 1K locations are occupied by the interrupt vector table (IVT).

All the segments of a COM program are kept in the same segment, i.e., its code segment, data segment and stack segments are within the same segment. Since the offsets in a physical segment can be of size 16 bits, therefore the size of COM program is limited to  $2^{16} = 64\text{K}$  which includes code segment, data segment and stack segment. The following program is a COM program, which adds two numbers. The program stores the result of addition and carry bit of addition in memory variables.

```
CSEG SEGMENT
 ASSUME CS:CSEG, DS:CSEG, SS:CSEG
 ORG 100h ; Segment starts at address 0100h
START: MOV AX, CSEG ; Move the segment address to AX
 MOV DS, AX ; Initialize Data segment using AX
 MOV AL, NUM1 ; Transfer first operand to AL
 ADD AL, NUM2 ; Add second operand to AL
 MOV RES, AL ; Store the result in AL to location RES
 RCL AL, 01 ; Rotate AL by 1 bit to get carry into LSB
 AND AL, 00000001B ; Mask out all bits except the LSB
 MOV CARRY, AL ; Store the carry bit into location CARRY
 MOV AX, 4C00h
 INT 21h
 NUM1DB 15h ; The first operand
 NUM2 DB 20h ; The Second operand
 RES DB ? ; Stores the sum
 CARRY DB ? ; Stores the carry bit
CSEG ENDS
END START
```

The program initializes the data segment CSEG, which is also the code and stack segment too. The two operands are stored in memory locations NUM1 and NUM2 are added and the result is stored in the location RES. In order to store the carry bit first a rotate with carry instruction is executed, followed by masking out the upper 7 bits. This causes only the carry bit to remain in the AL register. This carry bit is then moved to the location CARRY. Finally, the program exits to DOS using Interrupt 21H.

The COM programs are stored on a disk with an extension **.com**. A COM program uses less disk space in comparison to an equivalent EXE program. At run-time the

COM program places the stack automatically at the end of the segment, so they use at least one complete segment.

## 14.6.2 EXE Programs

An EXE program is stored on the disk with extension **.exe**. EXE programs are longer than the equivalent COM programs, as each EXE program is associated with an EXE header of 256 bytes followed by a load module containing the program itself. The EXE header contains information for the operating system to compute the address of various segments and other components related to offsets. A detailed discussion on segment header is beyond the scope of this Unit.

The load module of EXE program consists of several segments of length up to 64K. It may be noted that in 8086 microprocessor a maximum of four segments may be active at any time. These segments can be of variable sizes, with the maximum size being 64K.

In the subsequent Units, you will be learning to write EXE programs only as:

- EXE programs are better suited for debugging.
- Assembled EXE programs can be easily linked to subroutines of high-level languages.
- EXE programs are easily to relocate in the memory, as they do not contain any ORG statement. It may be noted that ORG statement forces a program to be loaded from a specific memory address.
- To fully use multitasking operating system, programs must be able to share computer memory and resources. An EXE program is easily able to do this.

An example of EXE program, which is equivalent to the COM program given in the previous section is given below:

```
DATA SEGMENT
 NUM1DB 15h ; The first operand
 NUM2 DB 20h ; The Second operand
 RES DB ? ; Stores the sum
 CARRY DB ? ; Stores the carry bit
DATA END
CODE SEGMENT
 ASSUME CS:CODE, DS:DATA
START: MOV AX, DATA ;Move the segment address to AX
 MOV DS, AX ; Initialize Data segment using AX
 MOV AL, NUM1 ; Transfer first operand to AL
 ADD AL, NUM2 ; Add second operand to AL
 MOV RES, AL ; Store the result in AL to location RES
 RCL AL, 01 ; Rotate AL by 1 bit to get carry into LSB
 AND AL, 00000001B ; Mask out all bits except the LSB
 MOV CARRY, AL ; Store the carry bit into location CARRY
 MOV AX, 4C00h
 INT 21h
CODE ENDS
END START
```

## 14.7 HOW TO WRITE GOOD ASSEMBLY PROGRAMS

This section defines the art of writing good assembly programs. A good program requires a clear description or problem and the algorithm that is being used for solving the problem. The following are some of the advices, which may help you in writing good assembly programs.

1. Write an algorithm for your program closer to assembly language. For example, the algorithm for preceding program would be:
  - Assuming that both the numbers, NUM1 and NUM2 are in the memory.
    - Put first number from memory to AL
    - Add second number from memory to AL
    - Store the result in some memory location
  - Position carry bit in Least significant bit (LSB) of a byte
    - mask off upper seven bits
    - store the result in the CARRY location.
2. Specify the input and output of the program.

Input: Two 8-bit numbers, in two different memory locations

Output: An 8-bit result and an 8-bit carry in memory locations

3. Study the instruction set carefully: Study the available set of instructions, their format and their limitations. For example, the limitation of the move instruction is that it cannot move an immediate operand to a segment register. Thus, the segment address is first moved to a register, say AX, which is then used to initialize the segment register. cannot be directly initialized by a memory variable.
4. You can exit to DOS, by using interrupt routine 21h, function 4Ch. Therefore, 04CH is placed in AH register followed by INT 21H instruction. This will result in exit to DOS.
5. It is a nice practice to first code your program on paper, and use comments liberally. This makes programming easier, and also helps you understand your program later. Please note that the number of comments does not affect the size of your program.
6. You may assemble your program using an assembler, which helps you in removing the syntax errors. It also helps in creating an .exe file for execution.

### Check Your Progress 3

Q1: When would you use .com program?

---



---



---



---

Q2: Why are EXE programs preferred over .COM programs?

---



---



---



---

Q3: State True or False

|   |   |
|---|---|
| T | F |
|---|---|

- (i) Input/output on Intel 8086/8088 machine running on DOS require special functions to be written by the assembly programmers.
- (ii) Intel 8086 processor recognizes only the software interrupts.

**Assembly Language Programming**

- |                                                                                                            |                          |
|------------------------------------------------------------------------------------------------------------|--------------------------|
| (iii) INT instruction in effect calls a subroutine, which is identified by a number.                       | <input type="checkbox"/> |
| (iv) Interrupt vector table IVT stores the interrupt handling programs.                                    | <input type="checkbox"/> |
| (v) INT 21H is a DOS function call.                                                                        | <input type="checkbox"/> |
| (vi) INT 21H will output a character on the monitor if AH register contains 02.                            | <input type="checkbox"/> |
| (vii) String input and output can be achieved using INT 21H with function number 09h and 0Ah respectively. | <input type="checkbox"/> |
| (viii) To perform final exit to DOS we must use function 4CH with the INT 21H.                             | <input type="checkbox"/> |
| (ix) Notepad can be used as an editor package.                                                             | <input type="checkbox"/> |
| (x) Linking is required to link several segments of a single assembly program.                             | <input type="checkbox"/> |
| (xi) Debugger helps in removing the syntax errors of a program.                                            | <input type="checkbox"/> |
| (xii) COM program is loaded at the 0 <sup>th</sup> location in the memory.                                 | <input type="checkbox"/> |
| (xiii) The size of COM program should not exceed 64K.                                                      | <input type="checkbox"/> |
| (xiv) A COM program is longer than an EXE program.                                                         | <input type="checkbox"/> |
| (xv) STACK of a COM program is kept at the end of the occupied segment by the program.                     | <input type="checkbox"/> |
| (xvi) EXE program contains a header module, which is used by DOS for calculating segment addresses.        | <input type="checkbox"/> |
| (xvii) EXE program cannot be easily debugged in comparison to COM programs.                                | <input type="checkbox"/> |
| (xviii) EXE programs are more easily relocatable than COM programs.                                        | <input type="checkbox"/> |

---

## 14.8 SUMMARY

---

This unit introduces you to some of the basic concept of 8086 programming, especially input/output. 8086 microprocessor uses an interrupt vector table (IVT) that points to the address of the interrupt servicing programs of 8086 micro-processor. One of the most important interrupts being interrupt 21H, which is used for input/output and several different functions. An IVT provides a flexible design environment, as you can change the interrupt service program without much efforts. This unit discusses some of the important functions of INT 21H. This unit also differentiates between COM & EXE program that are used in 8086 micro-processor.

## 14.9 SOLUTIONS/ ANSWERS

### Check Your Progress 1

1. (a) It helps in better understanding of computer architecture and machine language.  
(b) Results in smaller machine level code, thus result in efficient execution of programs.  
(c) Flexibility of use as very few restrictions exist.
2. A segment identifies a group of instructions or data value. We have four segments.  
1. Data segment 2. Code segment 3. Stack segment 4. Extra Segment
3. (a) False  
(b) False  
(c) True  
(d) True  
(e) False  
(f) True

### Check Your Progress 2

Q1: Interrupt 21H with functions 01H, 08H and 0AH

Q2: Output will be the digit 2.

Q3: Interrupt 21H with function 4C

### Check Your Progress 3

Q1: The COM programs are of size less than 64K. When you require small fast functions, you may use COM programs.

Q2: Due to better structure and possibility of linking with the high level language programs. In addition, if your environment supports multiprogramming then EXE programs can be easily relocated

Q3:

- (i) False
- (ii) False
- (iii) True
- (iv) False
- (v) True
- (vi) True
- (vii) True
- (viii) True
- (ix) True
- (x) False
- (xi) False
- (xii) False
- (xiii) True
- (xiv) False
- (xv) True
- (xvi) True
- (xvii) False
- (xviii) True

## 14.10 FURTHER READINGS

1. Yu-Cheng Lin, Genn. A. Gibson, “*Microcomputer System the 8086/8088 Family*” 2<sup>nd</sup> Edition, PHI.
2. Peter Abel, “*IBM PC Assembly Language and Programming*”, 5<sup>th</sup> Edition, PHI.
3. Douglas, V. Hall, “*Microprocessors and Interfacing*”, 2<sup>nd</sup> edition, Tata McGraw-Hill Edition.
4. Richard Tropper, “*Assembly Programming 8086*”, Tata McGraw-Hill Edition.
5. M. Rafiquzzaman, “*Microprocessors, Theory and Applications: Intel and Motorola*”, PHI.



---

# **UNIT 15 ASSEMBLY LANGUAGE PROGRAMMING**

---

## **Structure**

- 15.0 Introduction
- 15.1 Objectives
- 15.2 Simple Assembly Programs
  - 15.2.1 Data Transfer
  - 15.2.2 Simple Arithmetic Application
  - 15.2.3 Application Using Shift Operations
  - 15.2.4 Larger of the Two Numbers
- 15.3 Programming With Loops and Comparisons
  - 15.3.1 Simple Program Loops
  - 15.3.2 Find the Largest and the Smallest Array Values
  - 15.3.3 Character Coded Data
  - 15.3.4 Code Conversion
- 15.4 Programming for Arithmetic and String Operations
  - 15.4.1 String Processing
  - 15.4.2 Some More Arithmetic Problems
- 15.5 Modular Programming
  - 15.5.1 The Stack
  - 15.5.2 FAR and NEAR Procedures
  - 15.5.3 Parameter Passing
  - 15.5.4 External Procedure
- 15.6 Summary
- 15.7 Solutions/ Answers

---

## **15.0 INTRODUCTION**

---

After discussing about the directives, program developmental tools and Input / Output in assembly language programming, let us discuss more about assembly language programs. In this unit, we will first discuss the simple assembly programs, which performs simple tasks such as data transfer, arithmetic operations, and shift operations. A key example would be to find the larger of two numbers. Thereafter, you will go through more complex programs showing how loops and various comparisons are used to implement tasks like code conversion, coding characters, finding largest in array etc. This unit also discusses more complex arithmetic and string operations and modular programming. You must refer to further readings for more details on these programming concepts.

---

## **15.1 OBJECTIVES**

---

After going through this unit, you should be able to:

- write assembly programs with simple arithmetic logical and shift operations;
- implement loops;
- use comparisons for implementing various comparison functions;
- write simple assembly programs for code conversion;
- write simple assembly programs for implementing arrays;
- explain the use of stack in parameter passing; and
- use modular programming in assembly language

## 15.2 SIMPLE ASSEMBLY PROGRAMS

In this unit, first simple assembly programs are discussed and later more complex programs are written. In this section several simple assembly programs are explained.

### 15.2.1 Data Transfer

Data transfer is one of the most fundamental operations. 8086 microprocessor has two basic data transfer instructions, viz. MOV and XCHG. These instructions are explained with the help of simple example.

**Program 1:** Write a program using 8086 assembly language to exchange a data word stored in a memory location with the value stored in BX register and interchanges the value of AH and AL registers.

| Directives Statement                                                                                                                                                                                                                 | Discussion                                                                                                                                                                                                                                                                                                                            |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>DATA SEGMENT     VAL DW4321H DATA ENDS</pre>                                                                                                                                                                                    | The data segment stores a variable VAL, which stores a data word.                                                                                                                                                                                                                                                                     |
| <pre>CODE SEGMENT     ASSUME CS:CODE, DS:DATA MAINP:MOV AX, DATA         MOV DS, AX         MOVAX, 8765H         XCHG AH, AL         MOVBX, 8765H         XCHG BX, VAL         MOVAX, 4C00H         INT21H CODE ENDS END MAINP</pre> | <p>Use of assume directive to correlate segment registers with segment names and explicitly initialize data segment register.</p> <p>Move 8765H to AX register.<br/>Result: AX=6587H</p> <p>Move 8765H to BX register.<br/>Result: BX=4321H and VAL=8765H</p> <p>Return to operating system using Interrupt 21h with function 4C.</p> |

**Program 2:** Write an 8086-assembly program that interchanges the values of two Memory locations.

| Directives Statement                                                                                                                                                               | Discussion                                                                                                                                                                                                                                                                                                                     |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>DATA SEGMENT     VAL1 DB 25h     VAL2 DB 65h DATA ENDS</pre>                                                                                                                  | Define two variables VAL1 and VAL2 consisting of byte values.                                                                                                                                                                                                                                                                  |
| <pre>CODE SEGMENT     ASSUMECS:CODE, DS:DATA     MOV AX, DATA     MOV DS, AX     MOV AL, VAL1     XCHG VAL2, AL     MOV VAL1, AL     MOV AX, 4C00H     INT 21h CODE ENDS END</pre> | <p>Use of assume directive to correlate segment registers with segment names and explicitly initialize data segment register.</p> <p>Load the variable VAL1 into AL</p> <p>Exchange AL with variable VAL2</p> <p>Now, move the AL to variable VAL1</p> <p>Return to operating system using Interrupt 21h with function 4C.</p> |

In Program 2, why have we not used XCHG VAL1, VAL2 instruction directly? To answer this question,you should look into the constraints for the MOV instructions, which are given below:

- CS and IP may never be destination operands in MOV;
- Immediate data value and memory variables may not be moved to segment registers;
- The source and the destination operands should be of the same size;
- **Both the operands cannot be memory locations;**
- If the source is immediate data, it must not exceed 255 (FFh) for an 8-bit destination or 65,535 (FFFFh) for a 16-bit destination.
- The statement MOV AL, VAL1, copies the VAL1 that is 25h to the AL register;
- The instruction, XCHG AL, VAL2exchanges the value of AL register (25h) with VAL2(65h). Thus, after the execution of this instruction AL will contain 65h and VAL2 will contain 25h. VAL1 at this time will also contain 25h only.
- Finally, the instruction MOV VALUE1, AL will put the value 65h into VAL1.

### 15.2.2 Simple Arithmetic Application

Let us discuss an example that uses simple arithmetic operations.

**Program 3:** Find the average of two-bytevalues stored in the memory locations named as FIRST and SECOND. The result of the operation can be stored in a third memory location named MEAN.

**Discussion:** The program should have two memory variables stored in memory locations FIRST and SECOND and a third location for storing the mean value.An add instruction cannot add two memory locations directly, so you are required to move a single value in AL first and then add the second value to it.In addition, on adding the two-byte values, there is a possibility of a carry bit. Assuming that problem is addressing two unsigned binary numbers, the problem is how to put the carry bit into the AH register such that the AX(AH:AL) reflects the added byte values. This is done using ADC instruction.The ADC AH,00h instruction will add the immediate number 00h to the contents of the carry flag and the contents of the AH register. The result will be left in the AH register. Since we had cleared AH to all zeros, before the addition, we really are adding 00h + 00h + CF. The result of all this is that the carry flag bit is put in the AH register, which was desired by us.Finally, to get the mean value, you can divide the sum given in AX by 2. After the division, the 8-bit quotient will be left in the AL register, which can then be copied into the memory location named AVGE.

| <b>Directives</b><br><b>Statement</b>                                                    | <b>Discussion</b> |
|------------------------------------------------------------------------------------------|-------------------|
| <pre>DATA SEGMENT     FIRST DB 90h     SECOND DB 78h     MEAN    DB ? DATA    ENDS</pre> | Three variables   |
| <pre>CODE    SEGMENT     ASSUME CS:CODE, DS:DATA</pre>                                   |                   |

|                        |                                                                                                                                                                  |                                                                                                                                                                                                                                                                               |
|------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| START:                 | MOV AX, DATA<br>MOV DS, AX<br>MOV AL, FIRST<br>ADD AL, SECOND<br>MOV AH, 00h<br>ADC AH, 00h<br>MOV BL, 02h<br>DIV BL<br>MOV MEAN, AL<br>MOV AX, 4C00H<br>INT 21H | Initialize data segment<br>Get FIRST number in AL<br>Add SECOND number to AL<br>Clear AH register<br>Put carry in AH<br>Load divisor (2) in BL register<br>Divide AX by BL. Quotient in AL and remainder in AH; and copy the result to memory and return to operating system. |
| CODE ENDS<br>END START |                                                                                                                                                                  |                                                                                                                                                                                                                                                                               |

### 15.2.3 Application Using Shift Operations

Shift and rotate instructions are useful even for multiplication and division. These operations are not generally available in high-level languages, so assembly language may be an absolute necessity in certain types of applications.

**Program 4:** Write a program in 8086 assembly language to convert a two-digit ASCII code to an equivalent packed BCD number. You may assume that these two ASCII digits are stored in AL and BL registers.

*Discussion:* to its BCD equivalent. An ASCII digit is of 8 bit length. The lower four bits of an ASCII digit represents the equivalent BCD value of the ASCII digit. For example, ASCII digit ‘3’ is 00110011<sub>2</sub>, so if we replace the upper four bits by 0s, you will get 00000011<sub>2</sub>, which is equal to BCD digit 03. The number so obtained is called unpacked BCD number. The upper four bits of this byte is zero. So, the upper four bits can be used to store another BCD digit. The byte thus obtained is called packed BCD number. For example, an unpacked BCD number 39 is 00000011 00001001, that is, 03 09. The packed BCD will be 0011 1001, that is 39. Thus, the algorithm to convert two ASCII digits to packed BCD can be stated as:

*Input:* The two-digit ASCII number

*Output:* Two-digit packed BCD number

*Process:*

Convert the ASCII digits to unpacked BCD. An example is given in the table below:

| Digit | ASCII    | Unpacked BCD |
|-------|----------|--------------|
| 3     | 00110011 | 00000011     |
| 9     | 00111001 | 00001001     |

Move most significant BCD digit to upper four bit positions in byte by using rotate instruction as shown below:

|           |                                     |
|-----------|-------------------------------------|
| 0000 0011 | This is a unpacked BCD of digit 3.  |
| 00110000  | Use Rotate Instructions to get this |

Pack the bits of rotated BCD digit with the least significant BCD digit bits, as shown below to create a packed two-digit BCD number in a byte.

|                          |           |
|--------------------------|-----------|
| Rotated value of digit 3 | 0011 0000 |
| The unpacked digit 9     | 0000 1001 |
| Use OR operator          | 0011 1001 |

Display or store the results in a desired location or register.

The assembly language program for the above can be written in the following manner.

| <b>Directives</b> | <b>Statement</b>                                                                                                                                                                                                                                                                                                            | <b>Discussion</b>                                                                                                                                                                                                                                                                                                           |
|-------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|                   | <pre>DATA SEGMENT     HighDigit DB '3'     LowDigit DB '9'     PackedBCD DB ? DATA ENDS</pre>                                                                                                                                                                                                                               | The data segment stores the ASCII value of the two digits, the assumed digits are '3' and '9'.                                                                                                                                                                                                                              |
|                   | <pre>CODE SEGMENT     ASSUME CS:CODE, DS: DATA START: MOV AX, DATA         MOV DS, AX         MOVBL, HighDigit         MOVAL, LowDigit         ANDBL, 0Fh         ANDAL, 0Fh         MOVCL, 04h         ROLBL, CL         ORAL, BL         MOV PackedBCD, AL         MOV AX 4C00H         INT 21H CODE ENDS END START</pre> | Initialize data segment register<br>Move the Higher digit (3) in BL<br>Move the lower digit (9) to AL<br>Mask upper 4 bits of BL<br>Mask upper 4 bits of AL<br>Move the rotate count to CL<br>Rotate BL register using CL<br>OR to get the packed BCD in AL<br>Store the result in Packed BCD<br>Return to Operating system |

#### Discussion on Program 4:

8086 does not have any instruction to swap upper and lower four bits in a byte, therefore you need to use the rotate instruction 4 times. You can choose any of the two rotate instructions, ROL and RCL. In this example, we have chosen ROL, as it rotates the byte left by one or more positions, on the other hand RCL moves the MSB into the carry flag and brings the original carry flag into the LSB position, which is not what we want. Rest of the program proceeds as per the algorithm.

**Program 5:** Write a program using 8086 assembly language that adds two binary numbers (assume the numbers are of byte type) stored in consecutive memory locations. The result of the addition and carry, if any are also stored in the memory locations.

| <b>Directives</b> | <b>Statement</b>                                                                                                                                                                                                                                                                 | <b>Discussion</b>                                                                                                                                                                                                                                                                                                                                             |
|-------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|                   | <pre>DATA SEGMENT     NUM1 DB 25h     NUM2 DB 80h     RES DB ?     CARY DB ? DATA ENDS</pre>                                                                                                                                                                                     | First number contains 25h<br>Second number contains 80h<br>Will store sum of the two numbers<br>Will store carry bit, if any                                                                                                                                                                                                                                  |
|                   | <pre>CODE SEGMENT     ASSUME CS:CODE, DS:DATA START: MOV AX, DATA         MOV DS, AX         AL, NUM1         ADD AL, NUM2         MOV RES, AL         AL, 01         00000001B         AND AL,         MOV CARY, AL         MOV AH, 4CH         INT 21H CODEENDS ENDSTART</pre> | Initialize data segment register<br>Load the first number in AL, add the second number in AL and store the result into RES<br>Rotate AL with carry, to bring carry bit to the least significant bit, AND it with 00000001 <sub>2</sub> to mask out all bits except the least significant bit. Store the carry bit to CARY and return to the operating system. |

**Discussion:**

RCL instruction brings the carry into the least significant bit position of the AL register. The AND instruction is used for masking higher order bits of AL.

### 15.2.4 Larger of the Two Numbers

How are the comparisons done in 8086 assembly language? There exists a compare instruction CMP destination, source. However, this instruction only sets the flags on comparing two operands both of which should be either of 8 bits or 16 bits. Compare instruction just subtracts the value of the source operand from the destination operand without storing the result, but setting the flag during the process. In general, the following three comparisons may be able to address most of the comparison operations:

Instruction: CMP destination, source

| Result of comparison | Flag(s) affected    |
|----------------------|---------------------|
| destination < source | Carry flag = 1      |
| destination = source | Zero flag = 1       |
| destination > source | Carry = 0, Zero = 0 |

The following examples show how the flags are set when the numbers are compared.

**Example 1:**

```
MOV BL, 02h ; Move 02h to BL
CMP BL, 10h ; Compare BL with 10h. Sets carry flag = 1
As the value of BL is less than 10h, the carry flag would be set as borrow
would be needed to subtract 10h from BL.
```

**Example 2:**

```
MOV AX, F0F0h ; Same value is moved to AX
MOV DX, F0F0h ; and BX
CMP AX, DX ; On comparison, it sets Zero flag = 1
The zero flag is set as both the operands are same.
```

**Example 3:**

```
MOV BX, 200H
CMP BX, 0 ; Zero and Carry flags = 0
The destination register (BX) contains a value greater than the source (0), so
both the zero and the carry flags are cleared.
```

In the following section we will discuss an example that uses the flags set by CMP instruction.

### Check Your Progress 1

State True or False with respect to 8086/8088 assembly languages.

|   |   |
|---|---|
| T | F |
|---|---|

1. In a MOV instruction, the immediate operand value for 8-bit destination cannot exceed F0h.
2. XCHG VALUE1, VALUE2 is a valid instruction.
3. In the example given in section 15.2.2 you can change instruction DIV BL with a shift instruction.
4. A single instruction cannot swap the upper and lower four of a byte register.

5. An unpacked BCD number requires 8 bits of storage, however, two unpacked BCD numbers can be packed in a single byte register.
6. If AL = 05 and BL = 06 then CMP AL, BL instruction will clear the zero and carry flags.

## 15.3 PROGRAMMING WITH LOOPS AND COMPARISONS

Let us now discuss a few examples which are slightly more advanced than what we have been doing till now. This section deals with more practical examples using loops, comparison and shift instructions.

### 15.3.1 Simple Program Loops

The loops in assembly can be implemented using:

- Unconditional jump instructions such as JMP, or
- Conditional jump instructions such as JC, JNC, JZ, JNZ etc. and
- Loop instructions.

Let us consider some examples, explaining the use of conditional jumps.

#### Example 4:

```
CMP AX, BX ; compare instruction: sets flags
JE THERE ; if equal then skip the ADD instruction
ADD AX, 02 ; add 02 to AX
...
THERE: MOV CL, 07 ; load 07 to CL
```

In the example given above the control of the program will directly transfer to the label THERE, if the value stored in AX register is equal to that of the register BX. The same example can be rewritten in the following manner, using different jumps.

#### Example 5:

```
CMP AX, BX ; compare instruction: sets flags
JNE FIX ; if not equal do addition
JMP THERE ; if equal skip next instruction
FIX: ADD AX, 02 ; add 02 to AX
...
THERE: MOV CL, 07
```

The assembly code given above is not efficient, but suggests that there are many ways through which a conditional jump can be implemented. You should select the most optimum way based on your program requirements.

#### Example 6:

```
CMP DX, 00 ; checks if DX is zero.
JE Label1 ; if yes, jump to Label1 i.e., if ZF=1
...
Label1: other instructions ; control comes here if DX=0
```

**Example 7:**

```

MOV AL, 10 ; moves 10 to AL
CMP AL, 20 ; checks if AL < 20 i.e., CF=1
JL Label1 ; carry flag = 1 then jump to Lab1
...
Label1: other instructions ; control comes here if condition is satisfied

```

**LOOPING**

**Program 6:** Write a program using 8086 assembly language that computes the new prices from series of prices data stored in the memory. You may assume a constant inflation factor that is added to each old price value. Also assume that all the prices are given in the BCD form.

**Discussion:**

*Input:* A list of prices stored in the memory and a constant inflation factor

*Output:* The new prices

*Process:*

Repeat the following steps

    Read a price (in BCD) from the input array

    Add inflation factor

    Adjust result to correct BCD

    Put result back in the same array

Until all prices are converted to new price

| Directives Statement                                                                                                                                                                                                                 | Discussion                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre> ARRAYSEGMENT     PRICES DB 25h, 35h, 45h, 65h, 75h ARRAYS ENDS </pre>                                                                                                                                                          | The data segment is named ARRAYS and consist of a list of 5 PRICES.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
| <pre> CODESEGMENT ASSUME CS:CODE, DS:ARRAYS START:MOVAX, ARRAYS MOVDS, AX LEA BX, PRICES MOV CX, 0005h  DO_NEXT:MOVAL, [BX] ADD AL, 0Ah  DAA MOV [BX], AL  INC BX DEC CX  JNZ DO_NEXT MOV AH, 4CH INT 21H CODE ENDS END START </pre> | <p>Initialize data segment. Please note the use of name ARRAYS</p> <p>Move address of variable PRICES to BX register and move 5 to CX as PRICES has 5 values.</p> <p>Load the first value from array to AL and add constant 0Ah, which is assumed as inflation factor</p> <p>Since input is BCD, DAA adjusts the addition, and results are stored in the PRICES again.</p> <p>BX is incremented to point to next value and counter CX is decremented</p> <p>If the decrement operation does not result in zero, then jump is taken to DO_NEXT label, else all the values of PRICES has been processed, so program exits to Operating system.</p> |

**Discussion:**

Please note the use of instruction LEA BX, PRICES It will load the BX register with the offset of the array PRICES in the data segment named ARRAYS. [BX] is an

indirection through BX and points to the value stored at that element of array named PRICES. BX is incremented to point to the next element of the array. CX register acts as a loop counter and is decremented by one to keep a check of the bounds of the array. Once the CX register becomes zero, zero flag is set to 1. The JNZ instruction keeps track of the value of CX, and the loop terminates when zero flag is 1 because JNZ does not loop back.

The same program can be written using the LOOP instruction, in such case, DEC CX and JNZ DO\_NEXT instructions are replaced by LOOP DO\_NEXT instruction. LOOP decrements the value of CX and jumps to the given label, only if CX is not equal to zero. The LOOP instruction is demonstrated with the help of following program:

**Program 7:** Write a program using 8086 assembly language that prints the alphabets A-Z on the screen. This program is written using LOOP statement.

| Directives<br>Statement                                                                                                                                                    | Discussion                                                                                                                                                                                                                                                                                                                                                                                                                          |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| CODE SEGMENT<br>ASSUMECS:CODE<br>MAINP:MOV CX, 1AH<br>MOV DL, 41H<br>NEXTC: MOV AH, 02H<br>INT 21H<br>INC DL<br>LOOP NEXTC<br>MOV AX, 4C00H<br>INT21H<br>ENDS<br>END MAINP | 1AH=26 (number of alphabets to be displayed)<br>41H is hexadecimal equivalent of ASCII ‘A’.<br>Function 02H of Interrupt 21h is used to display the character stored in DL.<br>Increment DL to next alphabet value<br>Loop instruction will decrement CX by 1 and check if CX=0, if not it loops to label NEXTC to print remaining characters.<br>Once all the characters are printed, the program returns to the operating system. |

Let us now discuss slightly more complex program in the next section.

### 15.3.2 Find the Largest and the Smallest Array Values

Let us now put together whatever we have done in the preceding sections and write a program to find the largest and the smallest numbers from the numbers stored in an array. This program uses the JGE (jump greater than or equal to) instruction, because we have assumed the array values as signed numbers. We have not used the JAE instruction, which works correctly for unsigned numbers.

**Program 8:** Write a program using 8086 assembly language to find the largest and smallest numbers in an array.

*Discussion:* Initialize the **SMALL** and the **LARGE** variables as the first number in the array. They are then compared with the other array values one by one. If the value happens to be smaller than the assumed smallest number or larger than the assumed largest value, the **SMALL** and the **LARGE** variables are changed by this new value respectively. Let us use register DI to point the current array value and LOOP instruction for looping.

| Directives<br>Statement                                                                | Discussion                                                                                                                                                                                      |
|----------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| DATA SEGMENT<br>ARRAY DW -1, 2000,<br>-4000, 32767, 500, 0<br>LARGE DW ?<br>SMALL DW ? | Data segment includes a total of 6 signed values. You need to find the largest and the smallest among these values. The largest and smallest values will be stored in variables LARGE and SMALL |

|                                                                                                                                                             |                                                                                                                                                                                                                                                                                                                                                                                                  |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| DATA ENDS                                                                                                                                                   | respectively.                                                                                                                                                                                                                                                                                                                                                                                    |
| CODESEGMENT<br>ASSUME CS:CODE, DS:DATA<br>START:MOV AX, DATA<br>MOV DS, AX<br>MOV DI, OFFSET ARRAY<br>MOV AX, [DI]<br>MOV DX, AX<br>MOV BX, AX<br>MOV CX, 6 | Initialize the data segment register using AX<br>Move offset of ARRAY of data segment to DI and move the array element pointed by DI to AX register. DX and BX registers are used to store the largest and smallest respectively. The first value of array is moved in both these registers. Since the size of array is 6, so move 6 to CX register.                                             |
| A1: MOV AX, [DI]<br>CMP AX, BX<br>JGE A2                                                                                                                    | The element pointed to by DI is moved to AX, which is compared with BX. In case, AX is greater than or equal to BX, which means AX is not the smallest. The program jumps to label A2.                                                                                                                                                                                                           |
| MOV BX, AX<br>JMP A3                                                                                                                                        | This instruction will be executed, if the condition as above is false, which means AX is smallest, in this case the value of AX will be moved to BX, which contains new smallest value now. The program will then jump to label A3.                                                                                                                                                              |
| A2: CMP AX, DX<br>JLE A3<br>MOV DX, AX                                                                                                                      | This statement will be executed, if jump is taken to label A2. The value in AX will be compared to largest in DX, if it is less or equal to the program will jump to label A3, otherwise the AX is largest value, so it will be moved to DX register.                                                                                                                                            |
| A3: ADD DI, 2<br>LOOP A1<br>MOV LARGE, DX<br>MOV SMALL, BX<br><br>MOV AX, 4C00h<br>INT 21h<br>CODE ENDS<br>END START                                        | Next, DI is incremented by 2, so that it points to the next data word in the memory. The LOOP instruction decrements CX and checks if it is zero, if not then the steps from label A1 are repeated. Once the CX becomes zero, the DX is moved to LARGE and BX is moved to SMALL, as they contain the largest and smallest values respectively. Finally, program returns to the operating system. |

**Point to Note:** Since the data is word type that is equal to 2 bytes and memory organisation is byte wise, to point to next array value DI is incremented by 2.

### 15.3.3 Character Coded Data

The input/output takes place in the form of ASCII data. These ASCII characters are entered as a string of data. For example, to get two numbers from console, you may enter the numbers as:

|                     |      |
|---------------------|------|
| Enter first number  | 1234 |
| Enter second number | 3210 |

As each digit is input, you would store its ASCII code in a memory byte. After the first number was input, the number would be stored as follows:

The number is stored as:

|    |    |    |    |                                     |
|----|----|----|----|-------------------------------------|
| 31 | 32 | 33 | 34 | hexadecimal values stored in memory |
| 1  | 2  | 3  | 4  | equivalent ASCII digits             |

Each of these numbers will be input as equivalent ASCII digits and need to be converted either to digit string to a 16-bit binary value that can be used for computation or the ASCII digits themselves can be added which can be followed by instruction that adjust the sum to binary.

Another important data format is packed decimal numbers (packed BCD). A packed BCD contains two decimal digits per byte. Packed BCD format has the following advantages:

- The BCD numbers allow accurate calculations for almost any number of significant digits.
- Conversion of packed BCD numbers to ASCII (and vice versa) is relatively fast.
- An implicit decimal point may be used for keeping track of its position in a separate variable.

The instructions DAA (decimal adjust after addition) and DAS (decimal adjust after subtraction) are used for adjusting the result of an addition or subtraction operation on packed decimal numbers. However, no such instruction exists for multiplication and division. For the cases of multiplication and division the number must be unpacked. First, multiplied or divided and packed again.

Let us discuss the process of conversion of ASCII digits to equivalent binary number, which can be used for computation.

#### 15.3.4 Code Conversion

The conversion of data from one form to another is required in programs, which involve input of numbers. Therefore, in this section we will discuss an example, for converting a hexadecimal digit obtained in ASCII form to its equivalent binary form. An example showing ASCII to BCD conversion has already been explained as part of this unit.

**Program 9:** Write a program in 8086 assembly language to convert an ASCII input to equivalent hexadecimal value that it represents. The valid ASCII digits for this conversion are the digits 0 to 9 and alphabets A to F. The program assumes that the ASCII digit is read from a location in memory called ASCII. The hexadecimal or binary result is left in the AL. Since the program converts only one digit number the AL is sufficient for the results. The result in AL is made FF if the character in ASCII is not the proper hex digit.

Algorithm

Input: An ASCII digit

Output: The hexadecimal equivalent of number, if it is valid

Process:

If ASCII digit is in the range 30h to 39h it represents hex digit 0 to 9.

If ASCII digit is in the range 41h to 46h it represents hex digit A to F.

For any other value of ASCII, it does not represent a hex digit.

| Directives<br><br>Statement              | Discussion                              |
|------------------------------------------|-----------------------------------------|
| DATASEGMENT<br>ASCII DB 39h<br>DATA ENDS | ASCII variable contains an ASCII digit. |
| CODESEGMENT<br>ASSUME CS:CODE, DS:DATA   |                                         |

|  |                      |                                                                                |                                                                                                                                                                                                                                                                                        |
|--|----------------------|--------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|  | START:               | MOV AX, DATA<br>MOV DS, AX<br>MOV AL, ASCII<br>CMP AL, 30h<br>JB ERROR         | Initialize data segment using AX<br><br>Get the ASCII digits in AL register and compare it with 30h. If it is less than 30h, it is not a valid digit. So go to label ERROR                                                                                                             |
|  |                      | CMP AL, 3Ah<br>JAE ATOF<br>SUB AL, 30h<br>JMP CONVERTED                        | These instructions will be executed only if the ASCII digit is 30h or more. In case the ASCII digit is equal to or above 3Ah, you jump to ATOF, otherwise the ASCII is in range 30h to 39h. So convert it and move to label CONVERTED.                                                 |
|  | ATOF:                | CMP AL, 41h<br>JB ERROR<br>CMP AL, 46h<br>JA ERROR<br>SUB AL, 37h<br>CONVERTED | You will be here if ASCII is greater than or equal to 3Ah. Check if it is below 41h, if yes go to label ERROR. Next, check if the ASCII digit is above 46h, if yes go to label ERROR. Otherwise convert the ASCII to hex digit equivalent by subtracting 37h. Next, jump to CONVERTED. |
|  | ERROR:               | MOV AL, 0FFh                                                                   | Error is detected when AL has FF, which is moved to it.                                                                                                                                                                                                                                |
|  | CONVERTED:           | MOV AX, 4C00h<br>INT 21h                                                       | Otherwise, AL contains the converted hex digit so the program returns to operating system.                                                                                                                                                                                             |
|  | CODEENDS<br>ENDSTART |                                                                                |                                                                                                                                                                                                                                                                                        |

### Discussions:

The above program demonstrates a conversion of a single ASCII character to equivalent hexadecimal digit represented by that ASCII character. The above programs can be extended to take more ASCII values and convert them into a 16-bit binary number.

### ☛ Check Your Progress 2

1. Write the code sequence in assembly for performing following operation:

$$Z = ((A - B) / 10 * C) * * 2$$

.....

2. Write an assembly code sequence for adding an array of binary numbers.
- .....

3. An assembly program is to be written for inputting two 4 digits decimal numbers from console, adding them up and putting back the results. Will you prefer BCD addition for such numbers? Why?
- .....

4. How can you implement nested loops, such as given below?

```
for (i = 1 to 10, step 1)
 { for (j = 1 to 5, step 1)
 add 1 to AX}
```

in assembly language?

.....

.....

## **15.4 PROGRAMMING FOR ARITHMETIC AND STRING OPERATIONS**

Let us discuss some more advanced features of assembly language programming in this section. Some of these features give assembly an edge over the high-level language (HLL) programming as far as efficiency is concerned. One such instruction is for string processing. The object code generated after compiling the HLL program containing string instruction is much longer than the same program written in assembly language. The following section discuss a program of string processing:

### **15.4.1 String Processing**

Let us write a program for comparing two strings.

**Program 10:** Write a Program using 8086 assembly language to match two strings of same length stored in two separate memory locations.

| <b>Directives Statement</b>                                                                                                               | <b>Discussion</b>                                                                                                                                                                                                                                                                                                                                                                              |
|-------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>DATASEGMENT PASSWORDDB 'FAILSAFE' DESTSTR DB 'FEELSAFE' MESSAGEDB 'String are equal\$' DATA ENDS</pre>                               | The source string<br>The destination string<br>The message to be displayed if strings are the same                                                                                                                                                                                                                                                                                             |
| <pre>CODESEGMENT ASSUMECS:CODE, DS:DATA, ES:DATA MOV AX, DATA MOV DS, AX MOV ES, AX  LEASI, PASSWORD LEA DI, DESTSTR MOV CX, 08 CLD</pre> | The string matching requires two segments for data, viz. data segment for source string and extra data segment for destination string. Thus, DS and ES are initialized using AX.<br><br>The offset of PASSWORD and DESTSTR are stored in SI and DI respectively. As both the strings are of 8 bytes, CX=8. The direction flag, which determines the direction of string processing is cleared. |
| <pre>REPECMPSB JNE NTEQ MOV AH, 09     MOV DX, OFFSET MESSAGE INT 21h NTEQ: MOV AH 4CH     INT 21H CODE ENDS END</pre>                    | Repeat the compare string operation byte by byte and move to label NTEQ, if they are not equal at any character. The branch will not be taken, if the strings are equal. In that case the MESSAGE is printed on the screen. Finally, the program returns to the operating system.                                                                                                              |

#### **Discussion:**

In the program given above, the instruction CMPSB compares the two strings pointed by SI in Data Segment and DI register in extra data segment. The strings are compared byte by byte and then the pointers SI and DI are incremented to next byte. Please note the last letter B in the instruction indicates a byte. If it is W, that is if instruction is CMPSW, then comparison is done word by word and SI and DI are incremented by 2, that is to the next word respectively. The REPE prefix in front of

the instruction tells the 8086 to decrement the CX register by one, and continue to execute the CMPSB instruction, until the counter (CX) becomes zero. Thus, the code size is substantially reduced, when string instructions are used.

Thus, you can write efficient programs for moving one string to another, using MOVS, and scanning a string for a character using SCAS.

### 15.4.2 Some More Arithmetic Problems

Let us now take up some more practical arithmetic problems.

#### Use of array in assembly

An array is referenced using a base array value and an index. To facilitate addressing in arrays, 8086 has provided two index registers for mathematical computations, viz. BX and BP. In addition, two index registers are also provided for processing, viz. SI and DI. You can also use any general-purpose register for indexing.

Let us write a program to add two 5-byte numbers stored in an array. For example, two numbers in hex can be:

|           |    |    |    |    |       |
|-----------|----|----|----|----|-------|
| Carry in  | 0  | 0  | 0  | 1  | 0     |
|           | 20 | 11 | 01 | 10 | FF    |
|           | FF | 40 | 30 | 20 | 10    |
|           | 1  | 1F | 51 | 31 | 31 1F |
| Carry out |    |    |    |    |       |

Let us also assume that the numbers are represented as the lowest significant byte first and put in memory in two arrays. The result is stored in the third array SUM. The SUM also contains the carry out information, thus would be 1 byte longer than number arrays.

**Program 11:** Write a program in 8086 assembly language to add two five-byte numbers using arrays.

Algorithm:

Input: two arrays of 5 bytes each.

Output: an array of sum of size 6 bytes

Process:

Repeat the following steps till all the elements of array (5) are added

    Load the byte of first array in AL

    Add the corresponding byte of second array in AL with carry

    Store the result in a memory array

    Increment to next bytes

    Rotate carry into LSB of accumulator

    Mask all but LSB of accumulator

    Store carry result in memory

| Directives<br>Statement                                                                                | Discussion                                                                       |
|--------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------|
| <pre>DATASEGMENT NUM1DB0FFh,10h,01h,11h,20h NUM2DB10h,20h,30h,40h,0FFh SUMDB    6DUP(0) DATAENDS</pre> | Two arrays of size 5 each, SUM will store the addition and overall carry out bit |

|                                                                                                                                                                                                                                                                                                                                                                  |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre> CODESEGMENT ASSUME CS:CODE, DS:DATA START: MOVAX, DATA         MOVDS, AX         MOVS1, 00         MOVCX, 05h CLC AGAIN: MOV AL, NUM1[SI]         ADC AL, NUM2[SI]         MOV SUM[SI], AL         INC SI         LOOPAGAIN         RCL AL, 01h         AND AL, 01h         MOVSUM[SI], AL         MOV AX, 4C00h         INT 21h CODE ENDS ENDSTART </pre> | <p>Initialize segment register</p> <p>SI register is being used as index register, in the array, therefore, is initialized to 0. CX is initialized to the size of arrays and CLC clears the carry bit</p> <p>First an element of array NUM1 is moved to AL and then the corresponding element of NUM2 and previous carry is added in AL. The result is stored in the memory and SI is incremented to the next element of the arrays. These operations are repeated for all the elements of arrays.</p> <p>Next, the final carry is rotated to LSB of AL and the higher bits are masked out. This final carry is also stored in the 6<sup>th</sup> element of SUM. Finally, program returns to operating system.</p> |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

A good example of code conversion involving arithmetic is discussed next.

**Program 12:** Write a program using 8086 assembly programming language to convert a 4-digit BCD number into its binary equivalent. The BCD number can be stored as a word in memory location called BCD. The result is to be stored in location HEX.

The procedure to perform this number is explained with the help of following example.

Assume a 4-digit BCD number, say 4567, which is stored in a word in memory. To convert this number, you should extract each BCD digit separately and perform the following operation:

The binary number =  $4 \times (1000 \text{ or } 3E8h) + 5 \times (100 \text{ or } 64h) + 6 \times (10 \text{ or } Ah) + 7$

| Directives<br>Statement                                                                                                                                                                                             | Operation                                                                                                                                                                                                                                                                                                                |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| THOU EQU 3E8h<br>DATASEGMENT<br>BCDDW4567h<br>HEXDW?<br>DATAENDS                                                                                                                                                    | Constant THOU is equal to 1000, i.e. 3E8h                                                                                                                                                                                                                                                                                |
| CODESEGMENT<br>ASSUME CS:CODE, DS:DATA<br>START: MOVAX, DATA<br>MOVDS, AX<br>MOVAX, BCD<br>MOVBX, AX<br>MOVAL, AH<br>MOVBH, BL<br>MOVCL, 04<br>RORAH, CL<br>RORBH, CL<br>ANDAX, 0F0FH<br>ANDBX, 0F0FH<br>MOV CX, AX | Initialize data segment Register.<br><br>AX = 4567<br>BX = AX = 4567<br>AL = AH = 45<br>BH = BL = 67<br>CL = 4, as 4-bit rotation will be used<br>AH = 54 due to 4-bit rotation<br>BH = 76 due to 4-bit rotation<br>AX=5445 AND 0F0Fh = 0405<br>BX=7667 AND 0F0Fh = 0607<br>AX is moved to CX so that AX can be used for |

|                                                                                                                                                                                                                                                |                                                                                                                                                                                                                                                                                                                                               |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre> MOV AX, 0000H MOVAL, CH      MOVDI, THOU      MULDI           MOVDH, 00H           MOVDL, BL ADD DX, AX MOVAX, 0064h MULCL ADDDX, AX      MOVAX, 000Ah           MULBH ADDDX, AX MOVHEX, DX MOVAX, 4C00h INT21h CODEENDS ENDSTART </pre> | other operations. CX = AX = 0405<br>AX=0<br>AL=CH=04<br>DI=1000<br>AX= 04×1000 = 0FA0h<br>DH=0<br>DL=BL=07, thus DX= 0007<br>DX=0FA0h+0007h=0FA7h<br>AX=0064h<br>CL=05; AX=5×100 = 01F4h<br>DX= 0FA7h+01F4h= 119Bh<br>AX=000A<br>BH=6; AX=6×10=003Ch<br>DX=119Bh+003Ch=11D7h<br>Move this value to location HEX<br>Return to operating system |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

### ☞ Check Your Progress 3

1. Why should we perform string processing in assembly language in 8086 and not in high-level language?

.....  
.....  
.....

2. What is the function of direction flag?

.....  
.....  
.....

3. What is the function of REPE statement?

.....  
.....  
.....

---

## 15.5 MODULAR PROGRAMMING

---

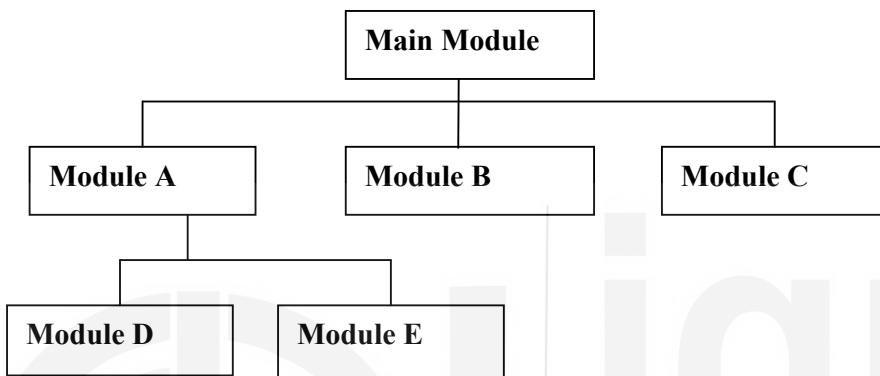
Modular programming refers to the practice of writing a program as a series of independently assembled source files. Each source file is a modular program designed to be assembled into a separate object file. Each object file constitutes a module. The linker collects the modules of a program into a coherent whole.

There are several reasons a programmer might choose to modularize a program.

1. Modular programming permits breaking a large program into a number of smaller modules each of more manageable size.
2. Modular programming makes it possible to link source code written in two separate languages. A hybrid program written partly in assembly language and

- partly in higher level language necessarily involves at least one module for each language.
3. Modular programming allows for the creation, maintenance and reuse of a library of commonly used modules.
  4. Modules are easy to comprehend.
  5. Different modules can be assigned to different programs.
  6. Debugging and testing can be done in a more orderly fashion.
  7. Document action can be easily understood.
  8. Modifications may be localized to a module.

A modular program can be represented using hierarchical diagram:



You can divide a program into subroutines or procedures. You need to CALL the procedures whenever needed. A subroutine call instruction transfers the control to subroutine instructions and the return statement brings the control back to the calling program.

### 15.5.1 The Stack

A procedure call is supported by a stack. Stack is a Last In First Out (LIFO) data structure. A stack in assembly language can be used for storing the return address of procedures, for parameter passing and for storing the value returned by the procedure.

In 8086 microprocessor a stack is created in the stack segment. The SS register stores the base of stack segment and SP register stores the position of the top of the stack. A value is pushed in to top of the stack or taken out (poped) from the top of the stack. The stack of 8086 is a word stack. In order to use stack, first the stack segment register is initialized, as given below:

|                                                                                                                                                                                                               |                                                                                                                                                                                                                                                                                                                                        |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre> STACK SEGMENT STACK     DW 100 DUP (0)     TOS LABEL WORD STACK SEG ENDS  CODE SEGMENT ASSUME CS:CODE, SS:STACK SEG     MOV AX, STACK SEG     MOV SS, AX     LEA SP, TOP     ... CODE ENDS END   </pre> | <p>Declaration of the stack segment.<br/>Assign 100-word locations to stack<br/>TOS is a word label to the top of stack.<br/>End of stack segment</p> <p>Just like a data segment, the SS register is initialized to the base of stack segment.</p> <p>The SP register is loaded with the maximum offset of the stack, represented</p> |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

The directive STACK SEGMENT STACK declares the logical segment for the stack segment. DW 100 DUP(0) assigns an actual size of the stack to 100 words. All locations of this stack are initialized to zero. The label TOS defines the initial top of this empty stack. Please note that the stack in 8086 is a WORD stack. The stack grows from a higher offset to a lower offset. The top position of stack uses an indirect addressing mechanism through a special register called the stack pointer (SP). SP initially is made to point to a label TOS. SP is automatically decremented when an item is put on the stack (called PUSH operation) and incremented as an item is retrieved from the stack (called POP operation). SP points to the address of the last element pushed on to the stack. The following table explains the PUSH and POP instructions of 8086 microprocessor

| Name                                   | Mnemonics | Description                                                                                                                             |
|----------------------------------------|-----------|-----------------------------------------------------------------------------------------------------------------------------------------|
| Push a word value SRC onto the stack   | PUSH SRC  | $SP \leftarrow SP - 2$ ; (decrement SP to the next word)<br>Put the word SRC into word pointed to by present value of SP and $SP + 1$ ; |
| Pop a word value from the stack in DST | POP DST   | Retrieve the word stored on stack top to DST ;<br>$SP \leftarrow SP + 2$ ;                                                              |

### 15.5.2 FAR and NEAR Procedures

Procedure provides the primary means of breaking the code in a program into modules. Procedures have one major disadvantage, i.e., they require extra code to join them together in such a way that they can communicate with each other. This extra code is sometimes referred to as linkage overhead.

A procedure call involves:

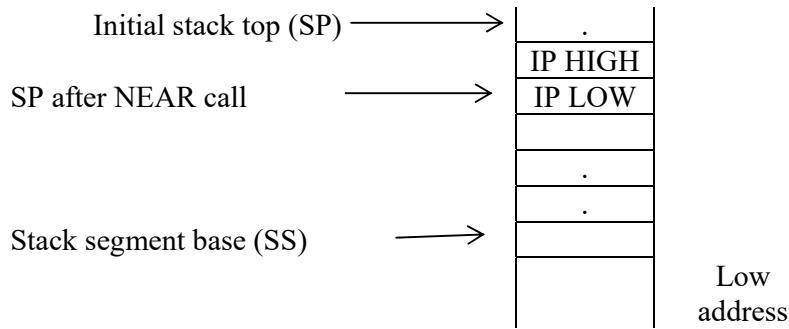
1. Unlike branch instructions, a procedure call must save the address of the next instruction to be executed of the calling program, so that after completion of execution of the procedure, the procedure can return the control to the calling program.
2. The registers used by the procedures need to be saved before their contents are changed by the procedure. These saved register values are used to restore the contents of the registers when the execution returns to the calling program.
3. A procedure must have means of communicating or sharing data with the procedures that call it, that is parameter passing.

#### Calls, Returns and Procedures definitions in 8086

The 8086 microprocessor supports CALL and RET instructions for procedure call.

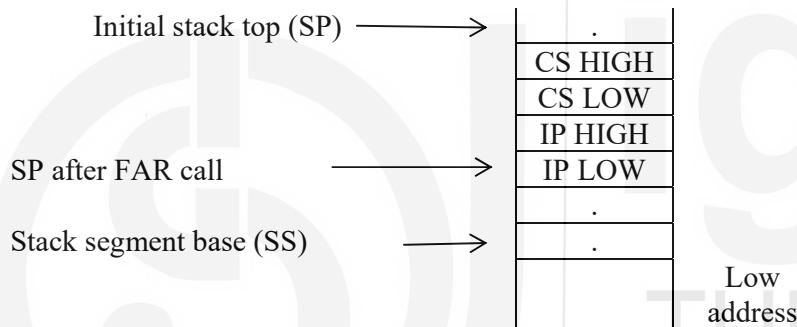
The CALL instruction not only branches to the indicated address, but also pushes the return address onto the stack. In addition, it also initializes IP with the address of the first instruction of the procedure. The RET instructions simply pops the return address from the stack. 8086 supports two kinds of procedure calls namely FAR and NEAR calls.

The NEAR procedure call is also known as Intra-segment call as the called procedure is in the same segment from which call has been made. Thus, only IP is stored as the return address on the top of the stack. The IP can be stored on the stack as:



Please note the growth of stack is towards stack segment base register. So, stack becomes full on an offset 0000h. Also, for push operation you decrement SP by 2 as stack is a word stack (word size in 8086 = 16 bits) while memory is byte organized memory.

FAR procedure call, also known as intersegment call, is a call made to separate code segment. Thus, the control will be transferred outside the current segment. Therefore, both CS and IP need to be stored as the return address. These values on the stack after the calls look like:



When the 8086 executes the FAR call, it first stores the contents of the code segment register followed by the contents of IP on to the stack. A RET from the NEAR procedure. Pops the two bytes into IP. The RET from the FAR procedure pops four bytes from the stack.

Procedure is defined within the source code by placing a directive of the form:

**<Procedure name> PROC <Attribute>**

A procedure is terminated using:

**<Procedure name> ENDP**

The <procedure name> is the identifier used for calling the procedure and the <attribute> is either NEAR or FAR. A procedure can be defined in:

1. The same code segment as the statement that calls it.
2. A code segment that is different from the segment containing the statement that calls it, but in the same source module as the calling statement.
3. A different source module and segment from the calling statement.

In the first case the <attribute> code NEAR should be used as the procedure and code are in the same segment. For the latter two cases the <attribute> must be FAR.

### 15.5.3 Parameter Passing

Parameter passing is a very important concept in assembly language. It makes the assembly procedures more general. Parameter can be passed to and from the main procedures. The parameters can be passed in the following ways to a procedure:

1. Parameters passing through registers
2. Parameters passing through dedicated memory location accessed by name
3. Parameters passing through pointers passed in registers
4. Parameters passing using stack.

However, in this Unit we will discuss parameter passing using a stack with the help of an example.

### **Passing Parameters Through Stack**

The best technique for parameter passing is through stack. It is also a standard technique for passing parameters when the assembly language is interfaced with any high-level language. Parameters are pushed on the stack and are referenced using BP register in the called procedure. One important issue for parameter passing through stack is to keep track of the stack overflow and underflow to keep a check on errors.

**Program 13:** Write a program using 8086 assembly language, which has a procedure to convert a two-digit packed BCD number to an equivalent binary number. Use stack for parameter passing.

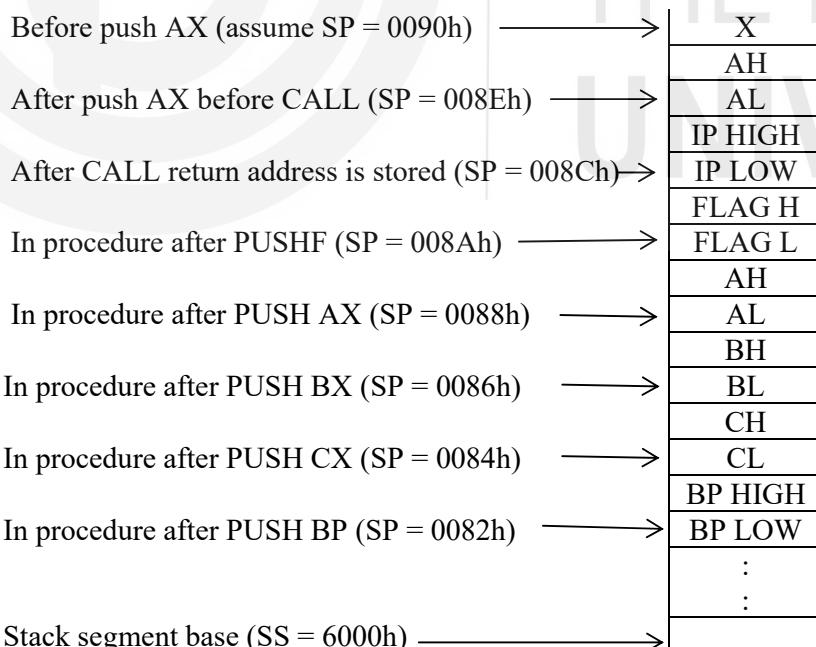
**Discussion:** The logic of conversion of packed BCD number to binary can be done in two simple steps. First convert the packed BCD digits to unpacked BCD digits and then multiply each digit with place value.

| Directives Statement                                                                                                                                                                                                                                                                                                                                         | Discussion                                                                                                                                                                                                                                                                                                                                                                                                                        |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| DATA SEGMENT<br>BCDDB25h<br>BINDB?<br>DATA SEGENDS                                                                                                                                                                                                                                                                                                           | Storage for BCD value<br>Storage for binary value                                                                                                                                                                                                                                                                                                                                                                                 |
| STACK SEGMENT STACK<br>DW100 DUP(0)<br>TOP_STACKLABELWORD<br>STACK SEGENDS                                                                                                                                                                                                                                                                                   | Stack of 100 words<br>Label for stack top                                                                                                                                                                                                                                                                                                                                                                                         |
| CODE SEGMENT<br>ASSUME CS:CODE_SEG,<br>DS:DATA_SEG, SS:STACK_SEG<br>START:MOVAX, DATA<br>MOVDS, AX<br>MOVAX, STACK-SEG<br>MOV SS, AX<br>MOV SP, OFFSET TOP_STACK<br>MOVAL, BCD<br>PUSH AX<br><br>CALL BCD_BINARY<br>POPAX<br>MOV BIN, AL<br>MOV AH, 4CH<br>INT 21H<br><br>; PROCEDURE : BCD_BINARY<br>BCD_BINARY PROC NEAR<br>; Store the registers<br>PUSHF | Initialize data segment<br>Initialize stack segment<br>Initialize stack pointer<br>Move BCD value into AL and push it onto word stack as parameter and call the procedure. The procedure returns binary value in AX register, which is moved to AL and the program returns to operating system.<br>The procedure to convert BCD value received in AX register to binary value. But, first all the registers used by the procedure |

|                               |                                                                                                  |
|-------------------------------|--------------------------------------------------------------------------------------------------|
| PUSHAX                        | and flags register is pushed in the stack. Next, the value of stack top is moved to BP register. |
| PUSHBX                        |                                                                                                  |
| PUSHCX                        |                                                                                                  |
| PUSHBP                        |                                                                                                  |
| MOVBP, SP                     |                                                                                                  |
| MOVAX, [BP+ 12]               | The stack location [BP+12] contains the BCD value, which is moved to AX =0025h.<br>BL=AL=25h     |
| MOVBL, AL                     | BL = 25h AND 0Fh = 05h                                                                           |
| ANDBL, 0Fh                    | AL = 25h AND F0h = 20h                                                                           |
| ANDAL, F0H                    | CL=04                                                                                            |
| MOVCL, 04                     | AL= 02h                                                                                          |
| RORAL, CL                     | BH=0Ah (or 10)                                                                                   |
| MOV BH, 0Ah                   | AX= 02×10 = 0014h                                                                                |
| MULBH                         | AL=14h+05h=19h                                                                                   |
| ADDAL, BL                     | Move this binary value to stack                                                                  |
| MOV [BP + 12], AX             |                                                                                                  |
| ; Restore flags and registers |                                                                                                  |
| POPBP                         | Restore all the registers to their original content, restore is in reverse order of storage and  |
| POPCX                         |                                                                                                  |
| POPBX                         |                                                                                                  |
| POPAX                         |                                                                                                  |
| POPF                          |                                                                                                  |
| RET                           | return to the calling program                                                                    |
| BCD_BINARY ENDP               | End of procedure                                                                                 |
| CODE SEG ENDS                 | End of code segment                                                                              |
| END START                     | End of the file                                                                                  |

### Discussion:

The parameter is pushed on the stack before the procedure call. The procedure call causes the current instruction pointer to be pushed on to the stack. In the procedure flags, AX, BX, CX and BP registers are also pushed in that order. Thus, the stack would be as follows:



The instruction MOV BP, SP transfers the contents of the SP to the BP register. Now BP is used to access any location in the stack, by adding appropriate offset to it. For example, MOV AX, [BP + 12] instruction transfers the word beginning at the 12th byte from the top of the stack to AX register. It does not change the contents of the BP

register or the top of the stack. Since the BP contains SP value, which is 0082h, therefore, BP+12 would be 0082h + 000Ch = 008Eh. This address contains the value of AX, which was pushed prior to call to the procedure. Please recall this pushed value was the BCD value, which is to be converted (this is the parameter value). Thus, this instruction copies the BCD parameter value at offset 008Eh into the AX register in the procedure. This instruction is not equivalent to POP instruction.

Stacks are useful for writing procedures for multi-user system programs or recursive procedures. It is a good practice to make a stack diagram as above while using procedure call through stacks. This helps in reducing errors in programming.

### 15.5.4 External Procedures

These procedures are written and assembled in separate assembly modules, and later are linked together with the main program to form a bigger module. Since the addresses of the variables are defined in another module, you need segment combination and global identifier directives to write such programs. Let us discuss them briefly.

#### Concept of Segment Combination

In 8086 assembler provides a means for combining the segments declared in different modules. Some typical combine types are:

1. PUBLIC: This combine directive combines all the segments having the same name (in different modules) as a single combined segment.
2. COMMON: If the segments in different object modules have the same name and the COMMON combine type then they have the same beginning address. During execution these segments overlay each other.
3. STACK: If the segments in different object modules have the same name and the combine type is STACK, then they become one segment having the length, as the sum of the lengths of individual segments.

For more details, you may refer to the further readings.

#### Use of Identifiers

- a) **Access to External Identifiers:** An external identifier is one that is referred in one module but defined in another. You can declare an identifier to be external by including it on as EXTRN in the modules in which it is to be referred. This tells the assembler to leave the address of the variable unresolved. The linker looks for the address of this variable in the module where it is defined to be PUBLIC.
- b) **Public Identifiers:** A public identifier is one that is defined within one module of a program but potentially accessible by all of the other modules in that program. You can declare an identifier to be public by including it on a PUBLIC directive in the module in which it is defined.

The following example explains the use of external procedures in 8086 microprocessor.

**Program 14:** Write a program using 8086 assembly procedure that divides a 32-bit number by a 16-bit number. The procedure should be defined in one module, and other modules should be able to call this procedure.

The procedure is named a SMART\_DIV procedure and first the calling program to this external procedure is given below:

| Directives Statement | Discussion |
|----------------------|------------|
|----------------------|------------|

|                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre> DATA SEGMENT WORD PUBLIC DIVIDENDDW2345h, 89ABh DIVISORDW5678h MESSAGEDB 'INVALID', '\$' DATA SEGENDS  MORE _ DATASEGMENTWORD QUOTIENT DW2DUP (0)     REMAINDERDW 0 MORE _ DATA ENDS  STACK SEGSEGMENTSTACK     DW100 DUP (0) TOP_STACKLABEL WORD STACK_SEG ENDS  PUBLICDIVISOR </pre>                                                                                                                                                                                                                                                                                                                                                                                                           | <p>Public data segment<br/>32-bit dividend<br/>16-bit divisor<br/>Message in case division is invalid</p> <p>This segment is valid only for this module as it is not shared.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
| <pre> PROCEDURESSEGMENTPUBLIC EXTRN SMART_DIV: FAR  PROCEDURESENDSD </pre>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             | <p>PROCEDURES segment is a PUBLIC division, it contains an external FAR procedure</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
| <pre> CODE SEGMENTWORDPUBLIC ASSUME CS:CODE_SEG DS:DATA_SEG, SS:STACK SEG START: MOVAX, DATA_SEG         MOVDS, AX         MOVAX, STACK_SEG         MOVSS, AX         MOVSP, OFFSET TOP_STACK         MOVAX, DIVIDEND         MOV DX, DIVIDEND + 2         MOVCX, DIVISOR          CALL SMART_DIV         JNC SAVE_ALL          JMP STOP  ASSUME DS:MORE_DATA SAVE_ALL:PUSH DS         MOV BX, MORE_DATA         MOV DS, BX         MOV QUOTIENT, AX         MOV QUOTIENT + 2, DX         MOV REMAINDER, CX  ASSUMEDS:DATA SEG         POP DS         JMP ENDING  STOP:  MOV DL, OFFSET MESSAGE         MOV AH, 09H         INT 21H  ENDING: MOV AH, 4Ch         INT 21H  CODE SEGENDS ENDSTART </pre> | <p>It declares the code segment as PUBLIC so that it can be merged with other PUBLIC segments.<br/>Initially, DATA_SEG is used as the data segment<br/>There is only one stack segment for this main program</p> <p>AX is loaded with lower word (2345h), DX with higher word (89ABh) of the DIVIDEND; and DIVISOR is loaded in CX.<br/>Procedure is Called<br/>In case Carry flag is NOT set all the values are saved<br/>Unconditional jump to label STOP is executed.<br/>New data segment is assumed and initialized. The old DS is pushed to stack.</p> <p>The values of QUOTIENT and REMAINDER are saved.</p> <p>After saving the data segment is restored and unconditional jump is taken to end of program<br/>This code will be executed in case of Carry Flag is set, it will show the message INVALID to show that division was invalid.</p> <p>Finally, program will terminate</p> |

### Discussion on the calling program:

The linker appends all the segments having the same name and PUBLIC directive with segment name into one segment. Their contents are pulled together into consecutive memory locations. The statement to be noted is PUBLIC DIVISOR. It tells the assembler and the linker that this variable can be legally accessed by other assembly modules. The statement EXTRN SMART\_DIV:FAR tells the assembler that this module will access a label or a procedure of type FAR in some assembly module. Please also note that the EXTRN definition is enclosed within the PROCEDURES SEGMENT PUBLIC and PROCEDURES ENDS, to tell the assembler and linker that the procedure SMART\_DIV is located within the segment PROCEDURES and all such PROCEDURES segments need to be combined in one. Please also note that in case the procedure SMART\_DIV encounters an error, such as division by zero, it sets carry flag, which is checked in the calling program to put the results in the memory or display an error message.

Let us now define the PROCEDURE module:

| Directives Statement                                                                                                                                                                                                                                                                                                                                 | Discussion                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Input:</b><br>Dividend is 2 words input. The low word is input in AX and high word is input in DX register<br>The divisor is input in CX register.<br><b>Output:</b><br>The Quotient is returned in DX:AX pair and remainder is returned in CX register. In case, divisor is zero, then Carry Flag is set to indicate that division is incorrect. |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
| DATA SEGMENT PUBLIC<br>EXTRN DIVISOR:WORD<br>DATA SEG ENDS<br><br>PUBLIC SMART_DIV                                                                                                                                                                                                                                                                   | This declaration informs assembler that DIVISOR is a word variable and is external to this procedure. It also indicates that DIVISOR can be found in public segment DATA_SEG<br><br>The SMART_DIV defined in this module is PUBLIC, i.e. it is available to other modules also.                                                                                                                                                                                                                                                                                                                                        |
| PROCEDURES SEGMENT PUBLIC<br>SMART_DIV PROC FAR<br>ASSUME CS:PROCEDURES,<br>DS:DATA_SEG<br><br>CMP DIVISOR, 0<br>JE ERROR_EXIT<br><br>MOV BX, AX<br>MOV AX, DX<br>MOV DX, 0000h<br>DIV CX<br><br>MOV BP, AX<br>MOV AX, BX                                                                                                                            | It declares the PROCEDURES segment as PUBLIC so that it can be merged with other PUBLIC segments with the same name.<br><br>The divisor is compared to 0, to check division by 0. You can also check it using CX. In case, it is same then jump to label ERROR_EXIT.<br><br>AX containing lower dividend word is moved to BX and higher divided word is moved from DX to AX. The DX is emptied. The DX: AX now contains (0000h:89ABh). This is divided by divisor in CX. Leaving remainder in DX and quotient in AX. The quotient of higher word division is moved to BP and AX is loaded with lower word of dividend. |
|                                                                                                                                                                                                                                                                                                                                                      | The DX:AX pair is divided by CX.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |

|                          |                                                                                           |                                                                                                                                                                                                                                                                                                                                             |
|--------------------------|-------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| DIVCX                    |                                                                                           | Please note DX already contained the remainder of earlier division. The quotient of this division is stored in AX and remainder in DX<br>The final remainder is transferred from DX to CX and the higher word division quotient saved in BO is moved to DX. Thus, DX:AX contains the quotient and CX remainder of this SMART_DIV procedure. |
| MOV CX, DX<br>MOV DX, BP | CLC<br>JMPEXIT<br><br>ERROR_EXIT: STC<br><br>EXIT: RET<br>SMART_DIV ENDP<br>PROCEDUREENDS | The carry flag is cleared and control jumps to label EXIT<br>This code is executed in case divisor was zero, the STC will set the Carry flag.<br>The procedure returns.                                                                                                                                                                     |

#### Discussion:

The procedure accesses the data item named DIVISOR, which is defined in the calling program, therefore the statement EXTRN DIVISOR:WORD is necessary for informing assembler that this data name is found in some other segment. The data type is defined to be of word type. Please note that the DIVISOR is enclosed in the same segment name as that of calling program that is DATA\_SEG and the procedure SMART\_DIV is in a PUBLIC PROCEDURES segment.

#### Check Your Progress 4

T  F

1. State True or False
  - (a) A NEAR procedure can be called only in the segment it is defined.
  - (b) A FAR call uses one word in the stack for storing the return address.
  - (c) While making a call to a procedure, the nature of procedure that is NEAR or FAR must be specified.
  - (d) Parameter passing through stack is used whenever assembly language programs are interfaced with any high level language programs.
  - (e) A segment if declared PUBLIC informs the linker to append all the segments with same name into one.
2. Show the stack if the following statements are encountered in sequence.
  - a) Call to a NEAR procedure FIRST at 20A2h:0050h
  - b) Call to a FAR procedure SECOND at location 3000h:5055h
  - c) RETURN from Procedure FIRST.

## 15.6 SUMMARY

This Unit presents some programs written in 806 assembly language. The programs cover elementary arithmetic problems, code conversion problems, use of arrays and

jump statements in assembly, the use of near and far procedure, highlighting the use of stack in procedure calls. Some of the important points presented in this unit are:

- An algorithm should precede your program. It is a good programming practice. This not only increases the readability of the program, but also makes your program less prone to logical errors.
- Use comments liberally. You will appreciate them later.
- Study the instructions, assembler directives and addressing modes carefully, before starting to code your program.
- Some instructions are very specific to the type of operand they are being used with, example signed numbers and unsigned numbers, byte operands and word operands, so be careful !!
- Certain instructions requires you to initialize certain registers prior to their use in program, for example, LOOP expects the counter value to be contained in CX register, string instructions expect DS:SI to be initialized by the segment and the offset of the string instructions, and ES:DI to be with the destination strings, INT 21h expects AH register to contain the function number of the operation to be carried out etc. Study such requirements carefully and do the needful. In case you miss out on something, in most of the cases, you will not get an error message, instead the 8086 will proceed to execute the instruction, with whatever junk is lying in those registers.

In spite of all these complications, assembly languages is still an indispensable part of programming, as it gives you an access to most of the hardware features of the machine, which might not be possible with high level language. Secondly, as you have also seen some assembly programs can be very efficient in comparison of HLL programs, for example, the assembly programs of string processing are very efficient. You should write assembly programs from the further readings.

---

## 15.7 SOLUTIONS/ ANSWERS

---

### Check Your Progress 1

1. False 2. False 3. True 4. True 5. True 6. False

### Check Your Progress 2

1. 

|      |           |                                                     |
|------|-----------|-----------------------------------------------------|
| MOV  | AX, A     | ; bring A in AX                                     |
| SUB  | AX, B     | ; subtract B                                        |
| MOV  | DX, 0000h | ; move 0 to DX as it will be used for word division |
| MOV  | BX, 0Ah   | ; move dividend to BX                               |
| IDIV | BX        | ; divide DX:AX by BX. The quotient will be in AX    |
| IMUL | C         | ; ((A-B) / 10 * C) in AX                            |
| IMUL | AX        | ; square AX to get (A-B/10 * C) ** 2                |

2. Assuming that each array element is a word variable and is stored in data segment.

|        |                   |                                                                                   |
|--------|-------------------|-----------------------------------------------------------------------------------|
| MOV    | CX, COUNT         | ; put the number of elements of the array in<br>; CX register                     |
| MOV    | AX, 0000h         | ; zero SI and AX                                                                  |
| MOV    | SI, AX            | ; add the elements of array in AX repeatedly                                      |
| AGAIN: | ADD AX, ARRAY[SI] | ; another way of handling array                                                   |
| ADD    | SI, 2             | ; select the next element of the array                                            |
| LOOP   | AGAIN             | ; add all the elements of the array. It will<br>; terminate when CX becomes zero. |

MOV TOTAL, AX ; store the results in TOTAL.

3. Yes, because the conversion efforts are less.
4. You may use two nested loop instructions in assembly also. However, as both the loop instructions use CX, therefore every time before we are entering inner loop you must push CX of outer loop in the stack and reinitialize CX to the inner loop requirements.

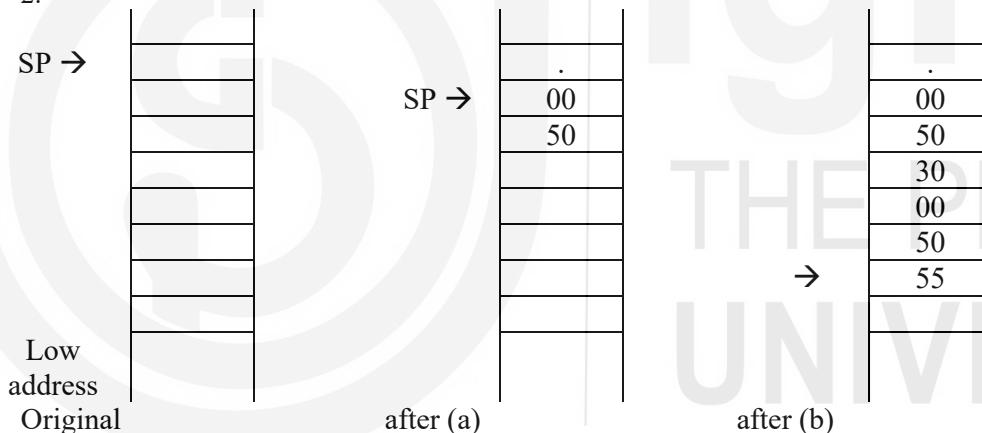
### Check Your Progress 3

1. The object code generated on compiling high level languages for string processing commands is, in general, found to be long and contains several redundant instructions. However, you can perform string processing very efficiently in 8086 assembly language.
2. Direction flag if clear will cause REPE statement to perform in forward direction, i.e. comparison would be from first element to last.
3. It repeats the instruction after this instruction.

### Check Your Progress 4

1. (a) True (b) False (c) False (d) True (e) True.

2.



- (c) The return for FIRST can occur only after return of SECOND. Therefore, the stack will be back in Original state.

---

# **UNIT 16 ADVANCED ARCHITECTURES**

---

| <b>Structure</b>                                            | <b>Page Nos.</b> |
|-------------------------------------------------------------|------------------|
| 16.0 Introduction                                           |                  |
| 16.1 Objectives                                             |                  |
| 16.2 Need of Advanced Architectures and Parallel Processing |                  |
| 16.3 Parallelism in Uni-Processor Systems                   |                  |
| 16.3.1 Arithmetic Pipeline                                  |                  |
| 16.3.2 Instruction Pipeline                                 |                  |
| 16.4 Parallelism through Hardware and Software              |                  |
| 16.4.1 Vector Processing                                    |                  |
| 16.4.2 Array Processing                                     |                  |
| 16.5 Multiprocessors                                        |                  |
| 16.5.1 Characteristics of Multiprocessors                   |                  |
| 16.5.2 Interconnection Structures                           |                  |
| 16.5.3 Inter-processor Arbitration                          |                  |
| 16.6 Inter-Processor Communication and Synchronization      |                  |
| 16.7 Cache Coherence                                        |                  |
| 16.8 Multi-core Processors                                  |                  |
| 16.9 Summary                                                |                  |
| 16.10 Solutions/Answers                                     |                  |

---

## **16.0 INTRODUCTION**

---

The previous Units of this course discuss about the basic computer architecture of a computer system, including the assembly language. Architecture design is the first step in life cycle of a processor. It is a very crucial step as the performance of processor majorly depends on the design chosen. For instance, if you choose a non-pipelined architecture for your processor, you are compromising its performance for simplicity. On the other hand, if you choose an architecture involving multiple cores, you can increase your processor performance exponentially but at the same time its complexity increases drastically. So, choosing an architecture that suits our application is a must. For that you should have an understanding of various types of architectures and their implementations in detail.

In this Unit, we will discuss some advanced architectures and design methodologies used to achieve higher performance of a computer system.

---

## **16.1 OBJECTIVES**

---

After going through this Unit, you will be able to:

- define the concept of parallelism in both uniprocessor and multi-processor system.
- define various techniques used to implement pipelining in a processor.
- explain the concept of multiprocessor systems and interrelated details.
- define how processors communicate with each other.
- define the concept of Cache Coherence and
- define various types of multi-core processors.

## 16.2 NEED OF ADVANCED ARCHITECTURES AND PARALLEL PROCESSING

The fast-paced IT industry demands high-end architectural designs to support its high-performance applications. To meet these needs computer architectures are going through numerous changes constantly. One such implementation is parallel processing which is used to achieve higher speeds.

Parallel Processing includes a set of techniques that can be used to increase processing speed and latency, which is defined for the data rate of data transfer, of a computational system. These techniques enable simultaneous processing of data as opposed to the conventional sequential processing. Due to parallel execution of instructions, there is a significant increase in throughput. *Throughput* is the measure of computation and is defined in terms of number of instructions that can be executed by a given processor in a given interval of time. Increase in throughput implies to increase in speed of operation. To this extent it must be noted that the need of parallel processing is to increase the performance of a system. The techniques involved in inducing parallel processing in a system widely vary in methodology and resources used. However, the aim of all these techniques is to process data concurrently. For example, in a processor, one instruction can be fetched from memory and another instruction can be executed by the ALU in the same clock cycle. Another example can be a system using two processors for parallel execution of instructions. In the former example, the Processor is using Pipelining as a technique to speed up the execution of instructions. We will discuss pipelining in the next sections in detail. The later example of two processors is used to achieve parallel processing. Here, performance of the system increases at the expense of increased complexity and cost.

## 16.3 PARALLELISM IN UNI-PROCESSOR SYSTEMS

Parallel execution of several programs can be done on a single processor system. Such parallelism can be implemented using the hardware as well as software. In this unit, we will focus only on the hardware related parallelism in a uni-processor system.

As far as hardware-based parallelism is concerned several techniques are used on uni-processor system to increase the throughput of instruction execution. Some of these techniques are:

- Several processors contain multiple units to perform various arithmetic functions, such as multiple adder circuits unit as designed in Block 1. These units' speedup the execution of various functions that can be done in parallel.
- Using the memory organizations, such as interleaved memory, to speed up the data access operations. In addition, the processor operations can be overlapped with memory operations, for example, 8086 micro-processor
- Use of pipelining in the processor. Which is explained in detail in this section.

The concept of pipelining is one of the major aspects of Parallelism in Uniprocessors, let us first answer the question “What is Pipelining?”.

Pipelining is a technique in which we divide the whole task into subtasks and execute them concurrently. Each subtask is processed in different segment of the processor. These segments are interconnected to each other in such a way that the result of one segment is passed to another segment. Output is obtained after the data is processed through all the segments. The characteristic feature of pipelining is that several processes can be running in different segments at the same time. Typical related example of pipelining can be an assembly line in industries, such as automobile manufacturing, which fabricate automobile in a step-by-step manner in different segments. In the following sub-section, the concepts of arithmetic and instruction pipeline are explained.

### 16.3.1 Arithmetic Pipeline

In Arithmetic Pipelining an arithmetic operation is divided into a sequence of segments and computations of several arithmetic operations can be done concurrently. Therefore, in this technique, there may be a possibility that a segment has produced data for the next segment, which is still processing the data of earlier operation. This will result in overlapping of data in different segments. To overcome this problem, you can use registers between segments to store data while next segment is still computing. Adding registers between stages can increase complexity but this step improves the reliability of the system.

Arithmetic pipelining is preferred in those systems in which same operation is to be performed on different data sets multiple times. The arithmetic pipeline can be used in computers used for high-end scientific computations to implement the floating point number arithmetic etc.

To describe Arithmetic Pipelining, let us consider a pipeline that adds two floating-point numbers. Let us consider two numbers represented in floating point format.

$$A = X \times 10^x$$

$$B = Y \times 10^y$$

Here, X, Y are the mantissa of numbers A, B respectively and x,y are the exponents of A and B respectively. In order to find the sum of the two floating point numbers A and B, the following sequence of steps are required to be computed:

**Step1:** Compare the exponents of the given floating point numbers A and B, i.e. compare the values of x and y.

**Step2:** Align the mantissa of the number having smaller exponent. It may be noted that this process may result in unnormalized mantissa representation.

**Step3:** Perform the addition of mantissas to obtain resultant mantissa, while exponent of the bigger number is chosen as the exponent of the result.

**Step4:** If the resultant mantissa is not normalized, then normalize the result.

The following example explains the process of addition of floating-point numbers.

**Example:** Add the following two floating point numbers using the process as explained above:

$$A = 0.5565 \times 10^4 \text{ and } B = 0.166 \times 10^3$$

**Step1:** The exponent of A, which is 4, is greater than that of exponent of B, which is 3.

**Step2:** In this step, you should align the mantissa of the smaller numbers by modifying the B as follows (please note that A remains unchanged). Please note after alignment exponent of both A and B is same, however, the mantissa of B is unnormalized.

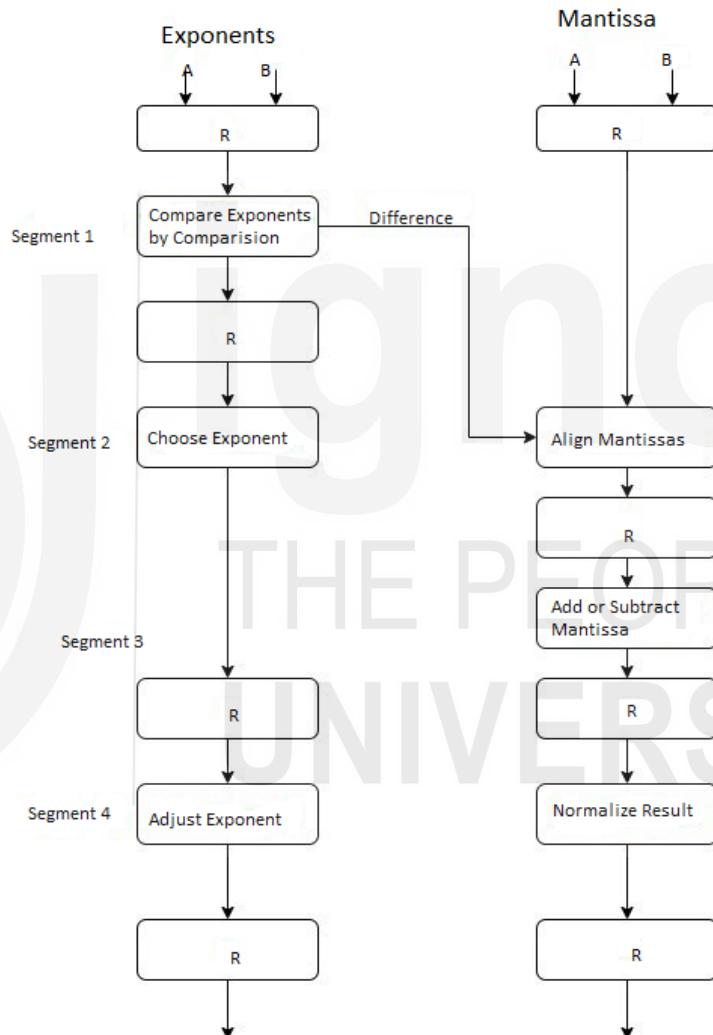
$$B = 0.0166 \times 10^4$$

**Step3:**Add the two numbers as follows:

$$\text{Result} = A + B = 0.5565 \times 10^4 + 0.0166 \times 10^4 = 0.5731 \times 10^4$$

**Step4:**Since the result is already normalized, this step will not perform any operation. Thus, the final result of the addition operation is  $0.5731 \times 10^4$ .

Registers are present in between all these steps to store intermediate values from one step to another step and prevent overlapping of data. The following diagram illustrates this process of addition or subtraction of two floating point numbers.



**Figure 16.1:** An arithmetic pipeline organization with intermediate registers for addition or subtraction of two floating point numbers.

### 16.3.2 Instruction Pipeline

In a processor when instructions are executed in various pipeline stages concurrently, then it is called instruction pipelining. The execution of an instruction is performed in a sequence of stages. For example, in a 3-stage pipelined processor every instruction undergoes three stages namely Fetch, Decode, Execute.

**Fetch:** In the fetch stage, the instruction is fetched from instruction memory and stored in Instruction Register (IR).

**Decode:** In this stage the instruction is decoded for information like operation to be performed, source operand address, destination operand address etc., the source operand and destination operand may be stored in temporary registers for further processing.

**Execute:** In this stage the ALU performs the operation specified by the instruction on the operands in the temporary registers.

In the pipelined processor as stated above, when the first instruction is in decode stage the second instruction enters into fetch stage. When first instruction enters execute stage, second instruction enters decode stage and third instruction enters fetch stage. The process continues until all the instructions are out of the execution stage.

Assuming that one stage is performed in one clock cycle, then this processor will take  $(k+n-1)$  clock cycles to execute  $n$  instructions in a  $k$  stage pipeline. In the present case,  $k= 3$ , as we have used a three-stage instruction pipeline. Thus, using this instruction pipeline  $n$  instructions, in an ideal condition, would be executed in  $n+2$  clock cycle.

If you are executing 100 instructions, then this processor will take 102 clock cycles to execute all of them. A Non-pipelined processor would have taken 300 clock cycles to complete the same task i.e.,  $n \times k$ .

We could see that there is a speedup ratio ( $S$ ) =  $\frac{(n \times k)t}{(k+n-1)t}$

| Instruction 1 | FE | DE | EX |    |    |    |    |
|---------------|----|----|----|----|----|----|----|
| Instruction 2 |    | FE | DE | EX |    |    |    |
| Instruction 3 |    |    | FE | DE | EX |    |    |
| Instruction 4 |    |    |    | FE | DE | EX |    |
| Instruction 5 |    |    |    |    | FE | DE | EX |

**Figure 16.2: An Instruction pipeline demonstrating execution of five instructions**

The computation of speed up due to pipeline though looks very promising, however, the pipelined execution suffers from several problems. The first problem is due to limitation of resources of the processing unit. For example, the system bus is one of the resources required by several units. The second problem may be related to data dependencies among consecutive instructions, for example, an instruction may produce a result, which may be required by the immediate next instruction. This may sometimes cause delay in execution of this immediate next instruction. Finally, in general, the decision to take a jump in conditional jump instructions is known only when the EX phase of that instruction is performed. This may result in emptying the pipeline. For example, consider instruction 1 is a conditional jump instruction, which causes jump to instruction 4 in case the condition is TRUE. The pipeline will execute as shown in the following diagram, assuming that the condition is evaluated to be TRUE.

|                                       |           |           |           |           |           |           |           |
|---------------------------------------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|
| <b>Conditional Jump Instruction 1</b> | <b>FE</b> | <b>DE</b> | <b>EX</b> |           |           |           |           |
| <b>Instruction 2</b>                  |           | <b>FE</b> | <b>DE</b> | -         |           |           |           |
| <b>Instruction 3</b>                  |           |           | <b>FE</b> | -         | -         |           |           |
| <b>Instruction 4</b>                  |           |           |           | <b>FE</b> | <b>DE</b> | <b>EX</b> |           |
| <b>Instruction 5</b>                  |           |           |           |           | <b>FE</b> | <b>DE</b> | <b>EX</b> |

**Figure 16.3: An Instruction pipeline with conditional jump**

Please note that the instruction 2 and instruction 3 are not to be executed, still they will be fetched. There are a number of methods such as branch prediction, which can be used to handle such problem. A detailed discussion on these problems are beyond the scope of this Unit.

### Check Your Progress-1:

- 1) **State True or False**
    - i) A Non-pipelined processor is faster than a pipelined processor
    - ii) Implementing parallelism in a system may lead to increased complexity of the system
    - iii) Throughput is the measure of area of the chip
    - iv) Modern day processors rely mostly on sequential data processing to improve their performance
    - v) A Non-pipelined processor takes  $n \times t_n$  clock cycles to complete n tasks with a clock period of  $t_n$  secs.
  - 2) Calculate the number of clock cycles a processor takes to complete 180 instructions in a 7-stage pipeline.
- .....  
.....  
.....  
.....

- 3) Where can you use Arithmetic Pipelining?
- .....  
.....  
.....  
.....

---

## **16.4 PARALLELISM THROUGH HARDWARE AND SOFTWARE**

---

As we know parallelism is the concept of processing multiple tasks concurrently. If this parallelism is achieved through hardware, then it is called Hardware parallelism. Similarly, if parallelism is achieved through software, then it is called Software parallelism.

### **Hardware Parallelism:**

- Hardware parallelism is implemented in a system at the architectural level by implementing hardware multiplicity or by modifying machine architecture.
- While implementing hardware parallelism one should keep in mind that it is a tradeoff between cost and performance.
- For example, using multiple adders can speed up a system which is initially working with single adder. But there is an overhead of cost of extra adders implemented in the system. Depending on the application you should choose between cost and performance.

### **Software Parallelism:**

- Software parallelism is achieved by modifying programs for data dependencies and control.
- Parallelism in software varies during the time of execution.

In the next section we discuss vector processing, which propose software instructions to exploit parallelism. This would be followed by a discussion of array processor, which is a hardware to exploit the parallelism.

### **16.4.1 Vector Processing**

The capabilities of a conventional computer are limited to computations that are simple and not as complex as vector or matrix arithmetic. But some modern-day applications need more complex calculations to be performed on huge amount of data. In this scenario, performing computations on scalars one by one is not recommended. To overcome this limitation, computers with vector processing are preferred to do faster vector calculations. There are many scientific problems and real-time computations that could take many weeks on a conventional computer to complete. Vector processors do the same problem in a much faster way. But how do they do that? The following description and example attempt to answer this question.

Instruction sets in vector processors contain instructions that operate on vector data, such as one-dimensional array. In scalar processors the instructions operate on a single data element at a time.

A scalar subtraction operation can be represented as  $C = A - B$ , where the scalar value  $B$  is subtracted from scalar value  $A$  in a single step.

However, if both  $A$  and  $B$  are vector operands, you may perform the subtraction operation, as shown below:

$$C_i = A_i - B_i, \text{ where } i=1,2,3, 4, \dots, n \text{ and } n \text{ is the number of elements in the vector}$$

If you follow the above procedure, then you will subtract vector  $B$  from vector  $A$  in  $n$  steps.

In vector processing vectors are subtracted in a single step. This procedure saves (n-1) clock cycles, which makes a huge difference when processing large arrays of data.

Vector instructions and scalar instructions are specified in a different way. For example, a typical scalar instruction having three operands can be defined as follows:

**Scalar instruction format:**

|        |                     |                     |                        |
|--------|---------------------|---------------------|------------------------|
| Opcode | Address of source 1 | Address of source 2 | Address of destination |
|--------|---------------------|---------------------|------------------------|

Whereas, a typical vector instruction would require to specify the base addresses of the operands and the length of the vector, which specifies the number of elements in those vectors. A typical vector instruction can be represented as follows:

**Vector Instruction Format:**

|               |                                 |                                 |                                     |                  |
|---------------|---------------------------------|---------------------------------|-------------------------------------|------------------|
| Vector Opcode | Base address of source operand1 | Base address of source operand2 | Base address of destination operand | Length of vector |
|---------------|---------------------------------|---------------------------------|-------------------------------------|------------------|

**Example:** SUB X,Y,Z,10

Here opcode is SUB, this specifies that the operation to be performed is subtraction. Base address of source operands are X and Y respectively and the Base address of destination operand is Z. The length of vector is 10.

The scalar version of the program may be written as follows:

for i = 1 to 10 SUB X[i], Y[i], Z[i]

The advantage of the vector instruction over scalar instruction is that the scalar version of instruction would require repeated fetch and decode of the subtraction instruction, whereas, the vector instruction would be fetched and decoded only once. The efficiency of execution of vector instructions can be further enhanced by using an array processor, which is discussed next.

### 16.4.2 Array Processing

Array processing is a technique in which arithmetic operations are done on large arrays of data. Array processing also works on vectors of data, as explained in the previous section. In general, to support array processing array processors are used in a computer. Array processors perform operations on vectors, but they operate on large data sets at the same time.

Array processors are of two types:

- 1) Attached array processors
- 2) SIMD array processors

Let us discuss these two array processors in detail:

#### 1) Attached Array Processors:

Attached array processors are used to aid host computer as an auxiliary processor or co-processor to perform array operations. Host computer can be a general-purpose computer that deals with simple computations and operations. When this general-purpose computer needs to operate on complex data like arrays, an array processor comes into play.

An array processor is interfaced with its host computer as follows:

- 1) I/O interface with host.
- 2) Interfaced to main memory of host.

Since array processor is attached to host computer through I/O interface, host computer treats it as a peripheral device.

## 2) SIMD array processors:

**SIMD** stands for Single Instruction Multiple Data,i.e. in a single instruction they process multiple data. This is achieved by having multiple processing elements working in parallel.All the processing elements are controlled using a single controller. Thus, a single instruction is sent to all the processing units, which are fed with different data by their local memory.

Every processing unit contains three elements:

- a) Arithmetic and Logical Unit (ALU)
- b) Floating point arithmetic unit
- c) Working registers

Control unit stores the instructions while processing units store operands.

**Example:**A vector operation  $C_i = A_i \times B_i$ ,where  $i=1,2,3\dots$  is to be performed using SIMD Array processor.

The array processor will perform the following steps, for execution of the instruction as given above:

**Step1:**Control unit checks whether the data is scalar or vector.If data is scalar, then control unit performs the operations directly in the master processing unit.If data is vector it sends data to Processing Units (PUs) and the following steps are performed:

**Step2:** Control unit sends data to processing units such that  $PU_1$  stores operands  $A_1$  and  $B_1$ ,  $PU_2$  stores operands  $A_2$  and  $B_2$  and so on.

**Step3:**Finally, control unit send instruction to multiply the operands to all the PUs.Thus, all the data is processed concurrently in the array processors.

Depending on the length of the vector, the control unit decides the number of PUs that should be activated. For example, if the length of vector to be processed is 32, then only 32 PUs are activated, and all other PUs are masked. In case, the length of a vector is more than the number of PUs, then all the PUs are activated and control unit uses these units as per requirements.

## Check Your Progress – 2

- 1) State True or False T/F
  - i) An Array processor performs operations on vector data
  - ii) SIMDarray processors contains a single processing unit and multiple control
  - iii) Masking is a technique of activating Processor Units (PU) in SMID Array Processor.
  - iv) Throughput of Vector Processor is higher than Scalar Processor
- 2) What are the components present in Processor Unit in an SIMD Array Processor?

- 3) Where can you use Vector Processing?

---

## 16.5 MULTI PROCESSORS

---

Multiprocessor systems refer to those systems that use multiple processors. These processors execute multiple tasks by sharing them on multiple processors simultaneously. You can think of multiprocessor as a system consisting of different processing units working together.

At the operating system level, multiprocessing can be referred as processing multiple processes simultaneously, with each process or task being executed by different cores. This is different from single processor execution flow. Major difference between multiprocessor and single processor flow comes from the fact that, in multiprocessor systems single task is executed by multiple cores, whereas, in single processor system multiple tasks are executed concurrently over a single core. This is called as multi-tasking. Sometimes multi-tasking is being confused with multi-processor, it may be noted that multi-tasking uses single processor but switches among the tasks on completion of a time slice allotted to a specific task. The Figure 16.4 describes the use of multi processors (CPU) while using the common shared memory.

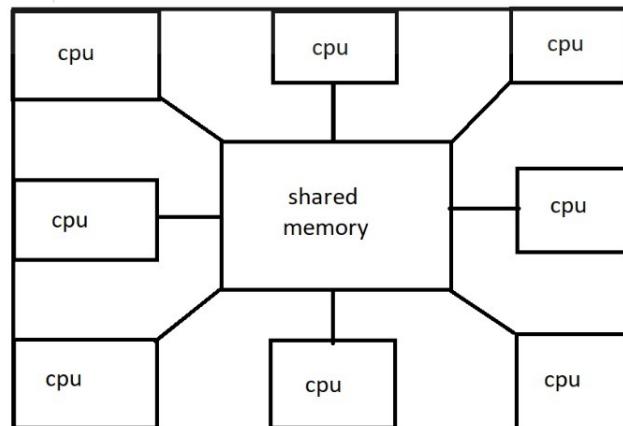


Figure 16.4: Multiprocessors with common shared memory

### 16.5.1 Characteristics of Multiprocessor

Multi-processor is a tightly coupled system with more than one CPU sharing same memory and input-output resources. Some of the major characteristics of multiprocessor are as discussed below:

- Processors used in multiprocessor system can be either Central Processing Units (CPUs) or Input-Output Processors (IOPs).
- They are categorized as Multiple Instruction Multiple Data (MIMD) systems.
- Major difference between a multiprocessor and a multicomputer system is, in multi computer system are connected with one another to form a network and they may or may not communicate information among them.
- A multiprocessor system may be designed to have a single master Operating System. This operating system may be responsible for communication between the processors, other peripherals and shared memory of the system.
- Multiprocessing can improve the reliability of the system such that when a failure happens, it affects only that processor, therefore, it has very less impact on rest of the processors.
- If one of the processors has failed, then some other processor can take up the duties of the faulty processor.
- Multiprocessing will enhance performance by breaking down a program into parallel executable tasks. Multiprocessors are flexible, i.e. the user can explicitly declare which processes are to be executed in parallel.
- Other effective way to improve performance of the system is to use a compiler which can identify parallelism in a process automatically.
- Compiler also tries to debug for data dependency in the program, which may cause issues while executing a task in parallel.
- Two threads of a task, which use different data can run concurrently.
- Multiprocessor systems can be implemented as ~~barely~~ loosely coupled systems as well. Each element of a processor in ~~barely~~ loosely coupled system has its own independent local memory.

### 16.5.2 Inter-Connection Structures

Every processor in a multi-processor system has a set of components, namely:

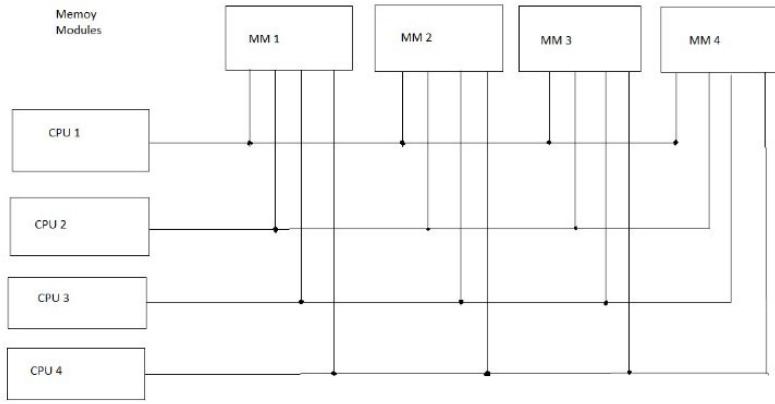
- CPU
- Shared Memory
- I/O

These components should communicate with each other for proper execution of programs. These components communicate among themselves through some interconnect paths; these paths which connect these modules or components are called as interconnection structures.

Let's discuss various interconnection structures in the detail.

#### Multi-port memory Organization

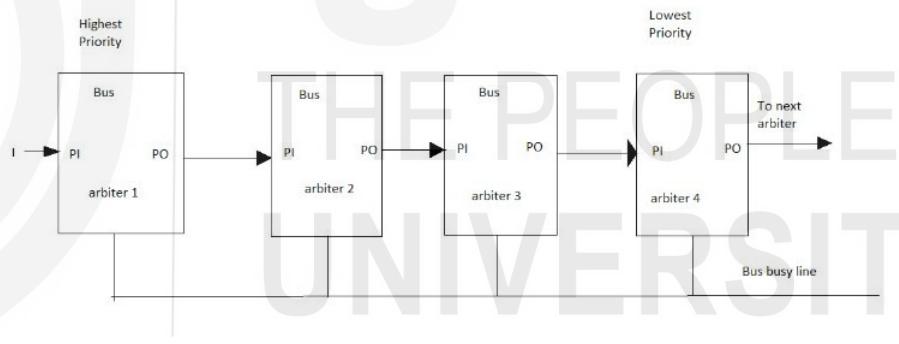
The multiport memory organization is illustrated in Figure 16.5. The multi-core processors have been using the shared memory resource modules (MM1, MM2, MM3, MM4) with the bus interconnection, which is used as a path for communicating among them. In this interconnection structure every CPU can communicate with every memory module. The advantage of such kind of connection is that, in case every CPU is trying to access a different memory module, they will be allowed access in parallel. However, there can be conflict, if two processors are trying to access the same memory module at the same time. A detailed discussion on this topic is beyond the scope of this Unit.



**Figure 16.5: Multi-port memory Organization**

### Bus Interconnection Structure

A bus is used as a communication path for connecting two or more processors or devices. Bus is a shared transmission medium; this is a major advantage of bus interconnects. Bus has already been explained in the Block 2 of this course. In this section, let us discuss the serial arbitration procedure used in the buses, in the context of priority scheme for bus allocation. Figure 16.6 shows the serial arbitration process.



**Figure 16.6: Serial Arbitration Procedure**

In the above figure, Serial Arbitration procedure is illustrated. The arbiters are sharing a common bus busy line which is synchronized to maintain the synchronous communication. The input (I) is serially transmitted through the arbiters with the enabling of bus busy line. It might depend on the clock edge occurrence of bus busy line to transmit the input data through the arbiters.

---

## 16.6 INTER PROCESSOR COMMUNICATION AND SYNCHRONIZATION

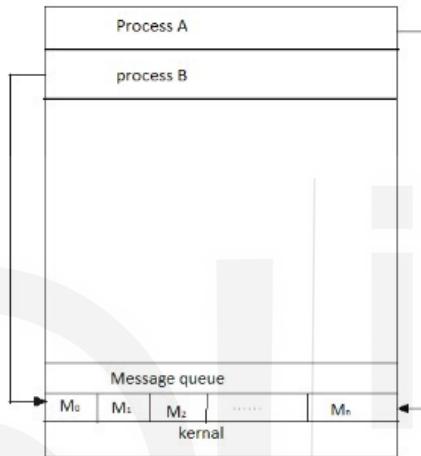
---

In this section we will be discussing about Inter process communication facilities. Various applications of inter-process and inter-thread communication facilities, which uses data transfer are:

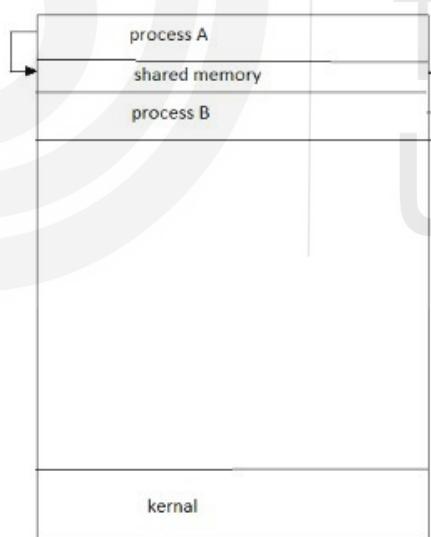
- Pipes (named, dynamic – shell or process generated)

- TCP/IP socket communication (named, dynamic - loop back interface or network interface)
- D-Bus is an IPC mechanism offering one to many broadcast and subscription facilities between processes.
- Shared memory
- Between Processes
- Between Threads (global memory)

Figure 16.7 shows and Figure 16.8 shows two simple mechanisms of inter-processor communication: message passing and shared memory.



**Figure 16.7: Message Passing**



**Figure 16.8: Shared Memory**

The above figure illustrates the Inter-processor communication in two possible ways. The first one is with respect to two processes A and B trying to pass messages to the message queue. It depends on the algorithm used, which decides on which process gets to pass the message first.

The second figure indicates the two processes A and B trying to use the common memory and ultimately depends on the algorithm designed, which decides how the processes share the memory.

When two processes need multiple shared resources at the same time in order to proceed further a **Deadlock** condition occurs.

To understand this condition, let us consider a scenario in which Thread X is expecting data from Thread Y for further processing and Thread Y is expecting data from Thread X for further processing of data. This scenario is called deadlock and no further processing can be done between the two threads.

Operating System provides synchronization and communications between processes sharing resources and prevents them from facing potential Inter process communication problems.

## 16.7 CACHE COHERENCE

In multi-processor system, while processing the shared data, various processors can store this shared data in multiple local caches for faster local processing. Therefore, there is a chance of having caches with incoherent data. This can lead to non-uniformity in shared resources. This problem is predominant in the case of multi-processors that use multiple processors and have shared data.

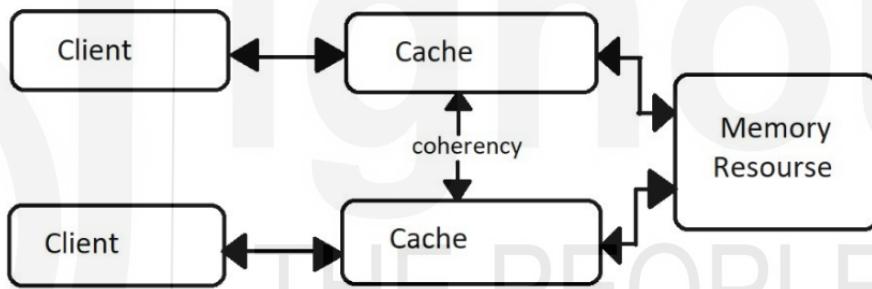
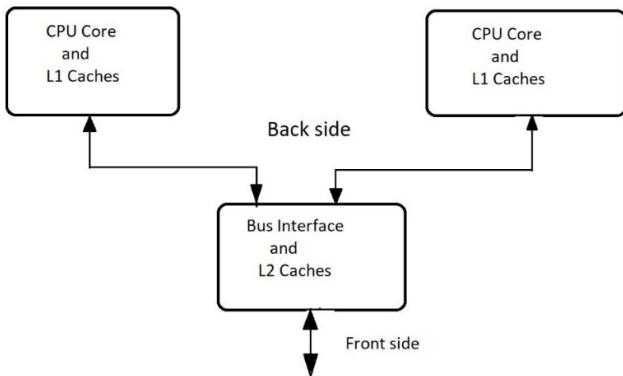


Figure 16.9: Cache Coherence

In the Figure 16.9, there are two client processors having their own local cache memory, which are initially coherent. Next, one of the client processors updates its cache, which in turn changes the data in the shared memory. The other client processor should also update its cache memory to reflect the change made by the first client processor. This is how cache coherence works. However, what happens if the other processor is not able to update its cache. In this case, cache coherence is not present. Thus, the update of data made by one processor though is reflected in its cache and shared memory, but the other client processor has no clue about it. There will be data conflict between both these client processors. To overcome such conflicts and synchronize data between multiple caches, Cache Coherence protocols are used. The detailed discussion on these protocols is beyond the scope of this Unit.

## 16.8 MULTI-CORE PROCESSORS

Multi-core processor is a design in which a number of cores are integrated on a single chip. These cores can be treated as processing units. They can process data and execute programs, similar to that of any other processor. It can be considered as a system having multiple processors. This system executes normal instructions such as add, subtract, and branch instructions. But these instructions can be executed on various cores at the same time. This results in drastic increase in speed by giving scope to multithreading and parallel computing.



**Figure 16.10: Multi-core Processor block diagram**

Figure 16.10 shows the basic architecture of cores in a multi-core processor. Applications of Multi-core processors are in various domains such as embedded systems, networks, Digital Signal Processing (DSP) and other domains.

Performance of Multi-core processors depends mostly on algorithms and methodologies used to implement them. In general, a good multi-core processor requires a strong support from its operating system so that all the cores of the processor are efficiently used. This Software parallelization is one of the areas of the present research.

### ☛ Check Your Progress – 3

1. Explain Inter-processor Communication with various techniques.

.....  
 .....  
 .....  
 .....

2. Explain about Cache Coherence

.....  
 .....  
 .....

3. Give some examples on Multi core processors

.....  
 .....  
 .....

---

## 16.9 SUMMARY

---

This Unit provides a basic introduction pf the concepts of pipelining and multiprocessor. The two pipelining techniques discussed in this Unit are – arithmetic pipeline and instruction pipeline. Various Parallelism techniques such as Vector and array processing are also discussed in this unit, and further topics like Multiprocessors are demonstrated with diagrams and explained in brief.

The information given on various topics such as pipelining, both arithmetic and Instruction pipeline, is introductoryand can be supplemented with additional reading. In fact, a course in an area of computer must be supplemented by further reading to keep your knowledge up to date, as the computer world is changing with leaps and bounds. In addition to further readings the student is advised to study several Indian Journals on computers to enhance her/hisknowledge.

---

## **16.10 SOLUTIONS**

---

### **Check your progress 1:**

1.

- i) False
- ii) True
- iii) False
- iv) False
- v) True

2. Number of stages in Pipeline is ‘k’ =7

Number of instructions is ‘n’ = 180

If each clock cycle is  $tp$

Then the number of clock cycles taken by the processer to process 180 tasks in a

$$7\text{-segment pipeline is } = (k+n-1)tp$$

$$= (7+(180-1))tp$$

$$= 186tp$$

Number of clock cycles for pipelined processor = 186

3.

- In high-speed Computers
- For doing floating point arithmetic
- For processing data of scientific problems

### **Check your progress – 2**

1.

- i) True
- ii) False
- iii) False
- iv) True

2. Processing Unit contains the following components:

- ALU
- Floating Point Unit
- Registers

3. Vector processing is used in following fields:

- Weather forecasting
- Artificial Intelligence (AI)
- Image Processing
- Healthcare and diagnosis

### **Check your progress- 3**

1. The inter processor communication can be achieved either through a shared memory area or through a queue. However, in both the cases software algorithms must control the sequence and synchronization of such communications.

2. In the case of multi-processors having shared memory and local caches, it is possible that a shared data item may be present in several local cache of the processors. In case, any of these processor changes the value of the shared data in its local cache, then it should result in update of this data in the shared memory and all other local caches. The coherence of cache is a required feature to keep the computation error free.

3. These days due to ULSI technologies most of the processors are multi-core processors. The Intel latest processors, AMD processor, processors of mobiles etc. all have multi-cores.