
UNIT 1 THE BASIC COMPUTER

Structure	Page Nos.
1.0 Introduction	5
1.1 Objectives	5
1.2 The von Neumann Architecture	5
1.3 Instruction Execution: An Example	9
1.4 Instruction Cycle	12
1.4.1 Interrupts	
1.4.2 Interrupts and Instruction Cycle	
1.5 Computers: Then and Now	18
1.5.1 The Beginning	
1.5.2 First Generation Computers	
1.5.3 Second Generation Computers	
1.5.4 Third Generation Computers	
1.5.5 Later Generations	
1.6 Summary	29
1.7 Solutions/Answers	29

1.0 INTRODUCTION

The use of Information Technology (IT) is well recognised. IT has become a must for the survival of all business houses with the growing information technology trends. Computer is the main component of an Information Technology network. Today, computer technology has permeated every sphere of existence of modern man. From railway reservations to medical diagnosis, from TV programmes to satellite launching, from matchmaking to criminal catching — everywhere we witness the elegance, sophistication and efficiency possible only with the help of computers.

In this unit, you will be introduced to one of the important computer system structures: the von Neumann Architecture. In addition, you will be introduced to the concepts of a simple model of Instruction execution. This model will be enhanced in the later blocks of this course. More details on these terms can be obtained from further reading. We have also discussed about the main developments during the various periods of computer history. Finally, we will discuss about the basic components of microprocessors and their uses.

1.1 OBJECTIVES

After going through this unit you will be able to:

- define the logical structure of the computer;
 - define the instruction cycle;
 - define the concept or Interrupt;
 - discuss the basic features of computers; and
 - define the various components of a modern computer and their usage.
-

1.2 THE VON NEUMANN ARCHITECTURE

The von Neumann architecture was the first major proposed structure for a general-purpose computer. However, before describing the main components of von Neumann

architecture, let us first define the term ‘computer’ as this will help us in discussing about von Neumann architecture in logical detail.

Computer is defined in the Oxford dictionary as “An automatic electronic apparatus for making calculations or controlling operations that are expressible in numerical or logical terms”.

The definition clearly categorises computer as an electronic apparatus although the first computers were mechanical and electro-mechanical apparatuses. The definition also points towards the two major areas of computer applications viz., data processing’s and computer assisted controls/operations. Another important aspect of the definition is the fact that the computer can perform only those operations/calculations, which can be expressed in Logical or Numerical terms.

Some of the basic questions that arise from above definition are:

How are the data processing and control operations performed by an electronic device like the computer?

Well, electronic components are used for creating basic logic circuits that are used to perform calculations. These components are further discussed in the later units. However, for the present discussion, it would be sufficient to say that there must be a certain unit that will perform the task of data processing and control.

What is the basic function performed by a computer? The basic function performed by a computer is the execution of the program. A program is a sequence of instructions, which operates on data, to perform certain tasks such as finding a prime number. The computer controls the execution of the program.

What is data in computers? In modern digital computers data is represented in binary form by using two symbols 0 and 1. These are called **binary** digits or bits. But the data which we deal with consists of numeric data and characters such as decimal digits 0 to 9, alphabets A to Z, arithmetic operators (e.g. +, -, etc.), relations operators (e.g. =, >, etc.), and many other special characters (e.g. ;, @, {, }, etc.). Therefore, there has to be a mechanism for data representation. Old computers use eight bits to represent a character. This allows up to $2^8 = 256$ different items to be represented uniquely. This collection of eight bits is called a byte. Thus, one byte is used to represent one character internally. Most computers use two bytes or four bytes to represent numbers (positive and negative) internally. The data also includes the operational data such as integer, decimal number etc. We will discuss more about data representation in the next unit.

Thus, the prime task of a computer is to perform instruction execution. The key questions, which can be asked in this respect, are: (a) how are the instructions supplied to the computer? and (b) how are the instructions interpreted and executed?

Let us answer the second question first. All computers have a Unit that performs the arithmetic and logical functions. This Unit is referred to as the Arithmetic Logic Unit (ALU). But how will the computer determine what operation is to be performed by ALU or in other words who will interpret the operation that is to be performed by ALU?

This interpretation is done by the Control Unit of the computer. The control unit accepts the binary form of instruction and interprets the instruction to generate control signals. These control signals then direct the ALU to perform a specified arithmetic or logic function on the data. Therefore, by changing the control signal the desired function can be performed on data. Or conversely, the operations that need to be performed on the data can be obtained by providing a set of control signals. Thus, for a new operation one only needs to change the set of control signals.

The unit that interprets a code (a machine instruction) to generate respective control signals is termed as Control Unit (CU). A program now consists of a sequence of codes. Each code is, in effect, an instruction, for the computer. The hardware

interprets each of these instructions and generates respective control signals such that the desired operation is performed on the data.

The Arithmetic Logic Unit (ALU) and the Control Unit (CU) together are termed as the Central Processing Unit (CPU). The CPU is the most important component of a computer's hardware.

All these arithmetic and logical Operations are performed in the CPU in special storage areas called registers. The size of the register is one of the important considerations in determining the processing capabilities of the CPU. Register size refers to the amount of information that can be held in a register at a time for processing. The larger the register size, the faster may be the speed of processing.

But, how can the instructions and data be put into the computers? The instructions and data to a computer are supplied by external environment; it implies that input devices are needed in the computer. The main responsibility of input devices will be to put the data in the form of signals that can be recognised by the system. Similarly, we need another component, which will report the results in proper format. This component is called output device. These components together are referred to as input/output (I/O) devices.

In addition, to transfer the information, the computer system internally needs the system interconnections. At present we will not discuss about Input/Output devices and system interconnections in details, except the information that most common input/output devices are keyboard, monitor and printer, and the most common interconnection structure is the Bus structure. These concepts are detailed in the later blocks.

Input devices can bring instructions or data only sequentially, however, a program may not be executed sequentially as jump, looping, decision-making instructions are normally encountered in programming. In addition, more than one data element may be required at a time. Therefore, a temporary storage area is needed in a computer to store temporarily the instructions and the data. This component is referred to as memory.

The memory unit stores all the information in a group of memory cells such as a group of 8 **binary digits** (that is a byte) or 16 bits or 32 bits etc. These groups of memory cells or bits are called memory locations. Each memory location has a unique address and can be addressed independently. The contents of the desired memory locations are provided to the CPU by referring to the address of the memory location. The amount of information that can be held in the main memory is known as memory capacity. The capacity of the main memory is measured in Mega Bytes (MB) or Giga Bytes (GB). One-kilo byte stands for 2^{10} bytes, which are 1024 bytes (or approximately 1000 bytes). A Mega byte stands for 2^{20} bytes, which is approximately a little over one million bytes, a giga byte is 2^{30} bytes.

Let us now define the key features of von Neumann Architecture:

- The most basic function performed by a computer is the execution of a program, which involves:
 - the execution of an instruction, which supplies the information about an operation, and
 - the data on which the operation is to be performed.

The control unit (CU) interprets each of these instructions and generates respective control signals.

- The Arithmetic Logic Unit (ALU) performs the arithmetic and logical Operations in special storage areas called registers as per the instructions of control unit. The size of the register is one of the important considerations in determining the processing capabilities of the CPU. Register size refers to the

amount of information that can be held in a register at a time for processing. The larger the register size, the faster may be the speed of processing.

- An Input/ Output system involving I/O devices allows data input and reporting of the results in proper form and format. For transfer of information a computer system internally needs the system interconnections. One such interconnection structure is BUS interconnection.
- Main Memory is needed in a computer to store instructions and the data at the time of Program execution. Memory to CPU is an important data transfer path. The amount of information, which can be transferred between CPU and memory, depends on the size of BUS connecting the two.
- It was pointed out by von-Neumann that the same memory can be used for Storing data and instructions. In such a case the data can be treated as data on which processing can be performed, while instructions can be treated as data, which can be used for the generation of control signals.
- The von Neumann machine uses **stored program concept**, i.e., the program and data are stored in the same memory unit for execution. The computers prior to this idea used to store programs and data on separate memories. Entering and modifying these programs was very difficult as they were entered manually by setting switches, plugging, and unplugging.
- Execution of instructions in von Neumann machine is carried out in a sequential fashion (unless explicitly altered by the program itself) from one instruction to the next.

Figure 1 shows the basic structure of a conventional von Neumann machine

Figure 1: Structure of a Computer

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

Check Your Progress 1

1) State True or False

T/F

- a) A byte is equal to 8 bits and can represent a character internally. ☐
- b) von Neumann architecture specifies different memory for data and instructions. The memory, which stores data, is called data memory and the memory, which stores instructions, is called instruction memory. ☐
- c) In von Neumann architecture each bit of memory can be accessed independently. ☐
- d) A program is a sequence of instructions designed for achieving a task/goal. ☐

- e) One MB is equal to 1024KB. ☐
- f) von Neumann machine has one path between memory and control unit. ☐
- This is the bottleneck of von Neumann machines.

2) What is von Neumann Architecture?

.....

.....

.....

3) Why is memory needed in a computer?

.....

.....

.....

1.3 INSTRUCTION EXECUTION: AN EXAMPLE

After discussing about the basic structure of the computer, let us now try to answer the basic question: “How does the Computer execute a Program?” Let us explain this with the help of an example from higher level language domain.

Problem: Write a program to add two numbers.

A sample C program (Assuming two fixed values of numbers as a = 5 and b = 2)

```
1.  #include <stdio.h>
2.  main ()
3.  {
4.  int a =5, b=2, c;
5.  c= a+b;
6.  printf (“\n The added value is: % d”, c);
7.  }
```

The program at line 4 declares variables that will be equivalent to 3 memory locations namely a, b and c. At line 5 these variables are added and at line 6 the value of c is printed.

But, how will these instructions be executed by CPU?

First you need to compile this program to convert it to machine language. But how will the machine instructions look like?

Let us assume a hypothetical instruction set of a machines of a size of 16 binary digits (bits) instructions and data. Each instruction of the machine consists of two components: (a) Operation code that specifies the operation that is to be performed by the instruction, and (b) Address of the operand in memory on which the given operation is to be performed.

Let us further assume that the size of operation code is assumed to be of six bits; therefore, rest 10 bits are for the address of the operand. Also the memory word size is assumed to be of 16 bits. Figure 2 shows the instruction and data formats for this machine. However, to simplify our discussion, let us present the operation code using Pnemonics like LOAD, ADD, STORE and decimal values of operand addresses and signed decimal values for data.

Figure 2: Instruction and data format of an assumed machine

The instruction execution is performed in the CPU registers. But before we define the process of instruction execution let us first give details on Registers, the temporary storage location in CPU for program execution. Let us define the minimum set of registers required for von Neumann machines:

Accumulator Register (AC): This register is used to store data temporarily for computation by ALU. AC is considered to contain one of the operands. The result of computation by ALU is also stored back to AC. It implies that the operand value is over-written by the result.

Memory Address Register (MAR): It specifies the address of memory location from which data or instruction is to be accessed (read operation) or to which the data is to be stored (write operation). Refer to figure 3.

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

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

Instruction Register (IR): Here the instructions are loaded prior to execution.

Comments on figure 3 are as follows:

- All representation are in decimals. (In actual machines the representations are in Binary).
- The Number of Memory Locations = 16
- Size of each memory location = 16 bits = 2 Bytes (Compare with contemporary machines word size of 16,32, 64 bits)
- Thus, size of this sample memory = 16 words (Compare it with actual memory size, which is 128 MB, 256 MB, 512 MB, or more).
- In the diagram MAR is pointing to location 10.
- The last operation performed was “read memory location 10” which is 65 in this. Thus, the contents of MBR is also 65.

The role of PC and IR will be explained later.

Now let us define several operation codes required for this machine, so that we can translate the High level language instructions to assembly/machine instructions.

Operation Code		Definition/Operation (please note that the address part in the Instruction format specifies the Location of the Operand on whom operation is to be performed)
LOAD	as	“Load the accumulator with the content of memory”
STORE	as	“Store the current value of Accumulator in the memory”
ADD	as	“Add the value from memory to the Accumulator”

A sample machine instructions for the assumed system for line 5 that is $c = a + b$ in the program would be:

LOAD	A	; Load the contents of memory location A to Accumulator register
ADD	B	; Add the contents of B to contents of Accumulator and store result in Accumulator.
STORE	C	; Store the content into location C

Please note that a simple one line statement in ‘C’ program has been translated to three machine instructions as above. Please also note that these translated instructions are machine dependent.

Now, how will these instructions execute?

Let us assume that the above machine instructions are stored in three consecutive memory locations 1, 2 and 3 and the PC contains a value (1), which in turn is address of first of these instructions. (Please refer to figure 4 (a)).

Figure 4: Memory and Registers Content on execution of the three given Consecutive Instructions (All notations in Decimals)

Then the execution of the instructions will be as follows:

Fetch First Instruction into CPU:

Step 1: Find or calculate the address of the first instruction in memory: In this machine example, the next instruction address is contained in PC register. It contains 1, which is the address of first instruction to be executed. (figure 4 a).

Step 2: Bring the binary instruction to IR register. This step requires:

- Passing the content of PC to Memory Address Registers so that the instruction pointed to by PC is fetched. That is location 1’s content is fetched.
- CPU issues “Memory read” operation, thus, brings contents of location pointed by MAR (1 in this case) to the MBR register.
- Content of MBR is transferred to IR. In addition PC is incremented to point to next instruction in sequence (2 in this case).

Execute the Instruction

- Step 3: The IR has the instruction LOAD A, which is decoded as “Load the content of address A in the accumulator register”.
- Step 4: The address of operand that is 13, that is A, is transferred to MAR register.
- Step 5: The content of memory location (specified by MAR that is location 13) is transferred to MBR.
- Step 6: The content of MBR is transferred to Accumulator Register.

Thus, the accumulator register is loaded with the content of location A, which is 5. Now the instruction 1 execution is complete, and the next instruction that is 2 (indicated by PC) is fetched and PC is incremented to 3. This instruction is ADD B, which instruct CPU to add the contents of memory location B to the accumulator. On execution of this instruction the accumulator will contain the sum of its earlier value that is A and the value stored in memory location B.

On execution of the instruction at memory location 3, PC becomes 4; the accumulator results are stored in location C, that is 15, and IR still contains the third instruction. This state is shown in figure 4 (C).

Please note that the execution of the instructions in the above example is quite simple and requires only data transfer and data processing operations in each instruction. Also these instructions require one memory reference during its execution.

Some of the problems/limitations of the example shown above are?

1. The size of memory shown in 16 words, whereas, the instruction is capable of addressing $2^{10} = 1$ K words of Memory. But why 2^{10} , because 10 bits are reserved for address in the machine instruction format.
2. The instructions shown are sequential in nature, however, a machine instruction can also be a branch instruction that causes change in the sequence of instruction execution.
3. When does the CPU stop executing a program? A program execution is normally completed at the end of a program or it can be terminated due to an error in program execution or sometimes all running programs will be terminated due to catastrophic failures such as power failure.

1.4 INSTRUCTION CYCLE

We have discussed the instruction execution in the previous section, now let us discuss more about various types of instruction execution.

What are the various types of operations that may be required by computer for execution of instruction? The following are the possible steps:

S.No.	Step to be performed	How is it done	Who does it
1	Calculate the address of next instruction to be executed	The Program Counter (PC) register stores the address of next instruction.	Control Unit (CU).
2.	Get the instruction in the CPU register	The memory is accessed and the desired instruction is brought to register (IR) in CPU	Memory Read operation is done. Size of instruction is important. In addition, PC is incremented to point to next instruction in sequence.
3.	Decode the instruction	The control Unit issues necessary control signals	CU.

4.	Evaluate the operand address	CPU evaluates the address based on the addressing mode specified.	CPU under the control of CU
5.	Fetch the operand	The memory is accessed and the desired operands brought into the CPU Registers	Memory Read
Repeat steps 4 and 5 if instruction has more than one operands.			
6.	Perform the operation as decoded in steps3.	The ALU does evaluation of arithmetic or logic, instruction or the transfer of control operations.	ALU/CU
7.	Store the results in memory	The value is written to desired memory location	Memory write

Figure 5: Instruction Cycle

Thus, in general, the execution cycle for a particular instruction may involve more than one stage and memory references. In addition, an instruction may ask for an I/O operation. Considering the steps above, let us work out a more detailed view of instruction cycle. Figure 5 gives a diagram of an instruction cycle.

Please note that in the preceding diagram some steps may be bypassed while some may be visited more than once. The instruction cycle shown in figure 5 consists of following states/stages:

- First the address of the next instruction is calculated, based on the size of instruction and memory organisation. For example, if in a computer an instruction is of 16 bits and if memory is organized as 16-bits words, then the address of the next instruction is evaluated by adding one in the address of the current instruction. In case, the memory is organized as bytes, which can be addressed individually, then we need to add two in the current instruction address to get the address of the next instruction to be executed in sequence.
- Now, the next instruction is fetched from a memory location to the CPU registers such as Instruction register.
- The next state decodes the instruction to determine the type of operation desired and the operands to be used.
- In case the operands need to be fetched from memory or via Input devices, then the address of the memory location or Input device is calculated.
- Next, the operand is fetched (or operands are fetched one by one) from the memory or read from the Input devices.
- Now, the operation, asked by the instruction is performed.
- Finally, the results are written back to memory or Output devices, wherever desired by first calculating the address of the operand and then transferring the values to desired destination.

Please note that multiple operands and multiple results are allowed in many computers. An example of such a case may be an instruction ADD A, B. This instruction requires operand A and B to be fetched.

In certain machines a single instruction can trigger an operation to be performed on an array of numbers or a string of characters. Such an operation involves repeated fetch for the operands without fetching the instruction again, that is, the instruction cycle loops at operand fetch.

Thus, a Program is executed as per the instruction cycle of figure 5. But what happens when you want the program to terminate in between? At what point of time is an interruption to a program execution allowed? To answer these questions, let us discuss the process used in computer that is called interrupt handling.

1.4.1 Interrupts

The term interrupt is an exceptional event that causes CPU to temporarily transfer its control from currently executing program to a different program which provides service to the exceptional event. It is like you asking a question in a class. When you ask a question in a class by raising hands, the teacher who is explaining some point may respond to your request only after completion of his/her point. Similarly, an interrupt is acknowledged by the CPU when it has completed the currently executing instruction. An interrupt may be generated by a number of sources, which may be either internal or external to the CPU.

Some of the basic issues of interrupt are:

- What are the different kinds of interrupts?
- What are the advantages of having an interruption mechanism?
- How is the CPU informed about the occurrence of an interrupt?
- What does the CPU do on occurrence of an interrupt?

Figure 6 Gives the list of some common interrupts and events that cause the occurrence of those interrupts.

Interrupt Condition	Occurrence of Event
Interrupt are generated by executing program itself (also called traps)	<input type="checkbox"/> Division by Zero <input type="checkbox"/> The number exceeds the maximum allowed. <input type="checkbox"/> Attempt of executing an illegal/privileged instruction. <input type="checkbox"/> Trying to reference memory location other than allowed for that program.
Interrupt generated by clock in the processor	Generally used on expiry of time allocated for a program, in multiprogramming operating systems.
Interrupts generated by I/O devices and their interfaces	<input type="checkbox"/> Request of starting an Input/Output operation. <input type="checkbox"/> Normal completion of an Input/Output operation. <input type="checkbox"/> Occurrence of an error in Input/Output operation.
Interrupts on Hardware failure	<input type="checkbox"/> Power failure <input type="checkbox"/> Memory parity error.

Figure 6: Various classes of Interrupts

Interrupts are a useful mechanism. They are useful in improving the efficiency of processing. How? This is to the fact that almost all the external devices are slower than the processor, therefore, in a typical system, a processor has to continually test whether an input value has arrived or a printout has been completed, in turn wasting a lot of CPU time. With the interrupt facility CPU is freed from the task of testing status of Input/Output devices and can do useful processing during this time, thus increasing the processing efficiency.

How does the CPU know that an interrupt has occurred?

There needs to be a line or a register or status word in CPU that can be raised on occurrence of interrupt condition.

Once a CPU knows that an interrupt has occurred then what?

First the condition is to be checked as to why the interrupt has occurred. That includes not only the device but also why that device has raised the interrupt. Once the

interrupt condition is determined the necessary program called ISRs (Interrupt servicing routines) must be executed such that the CPU can resume further operations.

For example, assume that the interrupt occurs due to an attempt by an executing program for execution of an illegal or privileged instruction, then ISR for such interrupt may terminate the execution of the program that has caused this condition. Thus, on occurrence of an Interrupt the related ISR is executed by the CPU. The ISRs are pre-defined programs written for specific interrupt conditions.

Considering these requirements let us work out the steps, which CPU must perform on the occurrence of an interrupt.

- The CPU must find out the source of the interrupt, as this will determine which interrupt service routine is to be executed.
- The CPU then acquires the address of the interrupt service routine, which are stored in the memory (in general).
- What happens to the program the CPU was executing before the interrupt? This program needs to be interrupted till the CPU executes the Interrupt service program. Do we need to do something for this program? Well the context of this program is to be saved. We will discuss this a bit later.
- Finally, the CPU executes the interrupt service routine till the completion of the routine. A RETURN statement marks the end of this routine. After that, the control is passed back to the interrupted program.

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

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

1.4.2 Interrupts and Instruction Cycle

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

Thus, an interrupt can be considered as the interruption of the execution of an ongoing user program. The execution of user program resumes as soon as the interrupt processing is completed. Therefore, the user program does not contain any code for interrupt handling. This job is to be performed by the processor and the operating system, which in turn are also responsible for suspending the execution of the user program, and later after interrupt handling, resumes the user program from the point of interruption.

Figure 7: Instruction Cycle with Interrupt Cycle

But when can a user program execution be interrupted?

It will not be desirable to interrupt a program while an instruction is getting executed and is in a state like instruction decode. The most desirable place for program

interruption would be when it has completed the previous instruction and is about to start a new instruction. Figure 7 shows instruction execution cycle with interrupt cycle, where the interrupt condition is acknowledged. Please note that even interrupt service routine is also a program and after acknowledging interrupt the next instruction executed through instruction cycle is the first instruction of interrupt servicing routine.

In the interrupt cycle, the responsibility of the CPU/Processor is to check whether any interrupts have occurred checking the presence of the interrupt signal. In case no interrupt needs service, the processor proceeds to the next instruction of the current program. In case an interrupt needs servicing then the interrupt is processed as per the following.

- Suspend the execution of current program and save its context.
- Set the Program counter to the starting address of the interrupt service routine of the interrupt acknowledged.
- The processor then executes the instructions in the interrupt-servicing program. The interrupt servicing programs are normally part of the operating system.
- After completing the interrupt servicing program the CPU can resume the program it has suspended in step 1 above.

Check Your Progress 2

1) **State True or False**

T/F

- i) The value of PC will be incremented by 1 after fetching each instruction if the memory word is of one byte and an instruction is 16 bits long. ☐
- ii) MAR and MBR both are needed to fetch the data /instruction from the memory. ☐
- iii) A clock may generate an interrupt. ☐
- iv) Context switching is not desired before interrupt processing. ☐
- v) In case multiple interrupts occur at the same time, then only one of the interrupt will be acknowledged and rest will be lost. ☐

2) What is an interrupt?

.....

.....

.....

.....

3) What happens on the occurrence of an interrupt?

.....

.....

.....

.....

1.5 COMPUTERS: THEN AND NOW

Let us now discuss the history of computers because this will give the basic information about the technological development trends in computer in the past and its projections for the future. If we want to know about computers completely, then we

must look at the history of computers and look into the details of various technological and intellectual breakthroughs. These are essential to give us the feel of how much work and effort has been done in the past to bring the computer to this shape. Our effort in this section will be to describe the conceptual breakthroughs in the past.

The ancestors of modern age computer were the mechanical and electromechanical devices. This ancestry can be traced as far back as the 17th Century, when the first machine capable of performing four mathematical operations, viz. addition, subtraction, division and multiplication, appeared. In the subsequent subsection we present a very brief account of Mechanical Computers.

1.5.1 The Beginning

Blaise Pascal made the very first attempt towards automatic computing. He invented a device, which consisted of lots of gears and chains which used to perform repeated additions and subtractions. This device was called Pascaline. Later many attempts were made in this direction.

Charles Babbage, the grandfather of the modern computer, had designed two computers:

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

The Analytical Engine by Babbage: It was a general purpose-computing device, which could be used for performing any mathematical operation automatically. The basic features of this analytical engine were:

- It was a general-purpose programmable machine.
- It had the provision of automatic sequence control, thus, enabling programs to alter its sequence of operations.
- The provision of sign checking of result existed.
- A mechanism for advancing or reversing of control card was permitted thus enabling execution of any desired instruction. In other words, Babbage had devised the conditional and branching instructions. The Babbage's machine was fundamentally the same as the modern computer. Unfortunately, Babbage work could not be completed. But as a tribute to Charles Babbage his Analytical Engine was completed in the last decade of the 20th century and is now on display at the Science Museum at London.

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

Harvard Mark-I and the Bug

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

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

The basic drawbacks of these mechanical and electromechanical computers were:

- Friction/inertia of moving components limited the speed.
- The data movement using gears and liners was quite difficult and unreliable.
- The change was to have a switching and storing mechanism with no moving parts. The electronic switching device “triode” vacuum tubes were used and hence the first electronic computer was born.

1.5.2 First Generation Computers

It is indeed ironic that scientific inventions of great significance have often been linked with supporting a very sad and undesirable aspect of civilization, that is, fighting wars. Nuclear energy would not have been developed as fast, if colossal efforts were not spent towards devising nuclear bombs. Similarly, the origin of the first truly general-purpose computer was also designed to meet the requirement of World War II. The ENIAC (the Electronic Numerical Integrator And Calculator) was designed in 1945 at the University of Pennsylvania to calculate figures for thousands of gunnery tables required by the US army for accuracy in artillery fire. The ENIAC ushered in the era of what is known as first generation computers. It could perform 5000 additions or 500 multiplications per minute. It was, however, a monstrous installation. It used thousands of vacuum tubes (18000), weighed 30 tons, occupied a number of rooms, needed a great amount of electricity and emitted excessive heat.

The main features of ENIAC can be summarised as:

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

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

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

1.5.3 Second Generation Computers

Silicon brought the advent of the second generation computers. A two state device called a transistor was made from silicon. Transistor was cheaper, smaller and dissipated less heat than vacuum tube, but could be utilised in a similar way to vacuum tubes. A transistor is called a solid state device as it is not created from wires, metal glass capsule and vacuum which was used in vacuum tubes. The transistors were invented in 1947 and launched the electronic revolution in 1950.

But how do we characterise the future generation of computers?

The generations of computers are basically differentiated by the fundamental hardware technology. The advancement in technology led to greater speed, large memory capacity and smaller size in various generations. Thus, second generation computers were more advanced in terms of arithmetic and logic unit and control unit than their counterparts of the first generation and thus, computationally more powerful. On the software front at the same time use of high level languages started and the developments were made for creating better Operating System and other system software.

One of the main computer series during this time was the IBM 700 series. Each successful member of this series showed increased performance and capacity and reduced cost. In these series two main concepts, I/O channels - an independent processor for Input/Output, and Multiplexer - a useful routing device, were used. These two concepts are defined in the later units.

1.5.4 Third Generation Computers

The third generation has the basic hardware technology: the Integrated Circuits (ICs). But what are integrated circuits? Let us first define a term called discrete components. A single self-contained transistor is called discrete component. The discrete components such as transistors, capacitors, resistors were manufactured separately and were soldered on circuit boards, to create electronic components or computer cards. All these cards/components then were put together to make a computer. Since a computer can contain around 10,000 of these transistors, therefore, the entire mechanism was cumbersome. The basic idea of integrated circuit was to create electronic components and later the whole CPU on a single Integrated chip. This was made possible by the era of microelectronics (small electronics) with the invention of Integrated Circuits (ICs).

In an integrated circuit technology the components such as transistors, resistors and conductors are fabricated on a semiconductor material such as silicon. Thus, a desired circuit can be fabricated in a tiny piece of silicon. Since, the size of these components is very small in silicon, thus, hundreds or even thousands of transistors could be fabricated on a single wafer of silicon. These fabricated transistors are connected with a process of metalisation, thus, creating logic circuits on the chip.

An integrated circuit is constructed on a thin wafer of silicon, which is divided into a matrix of small areas (size of the order of a few millimeter squares). An identical circuit pattern is fabricated in a dust free environment on each of these areas and the wafer is converted into chips. (Refer figure 8). A chip consists of several gates, which are made using transistors. A chip also has a number of input and output connection points. A chip then is packaged separately in a housing to protect it. This housing provides a number of pins for connecting this chip with other devices or circuits. For example, if you see a microprocessor, what you are looking and touching is its housing and huge number of pins.

Different circuits can be constructed on different wafers. All these packaged circuit chips then can be interconnected on a Printed-circuit board (for example, a motherboard of computer) to produce several complex electronic circuits such as in a computer.

The Integration Levels:

Initially, only a few gates were integrated reliably on a chip. This initial integration was referred to as small-scale integration (SSI).

With the advances in microelectronics technologies the SSI gave way to Medium Scale Integration where 100's of gates were fabricated on a chip.

Next stage was Large Scale Integration (1,000 gates) and very large integration (VLSI 1000,000 gates on a single chip). Presently, we are in the era of Ultra Large Scale Integration (ULSI) where 100,000,000 or even more components may be fabricated on a single chip.

What are the advantages of having densely packed Integrated Circuits? These are:

- **Reliability:** The integrated circuit interconnections are much more reliable than soldered connections. In addition, densely packed integrated circuits enable fewer inter-chip connections. Thus, the computers are more reliable. In fact, the two unreliable extremes are when the chips are in low-level integration or extremely high level of integration almost closer to maximum limits of integration.
- **Low cost:** The cost of a chip has remained almost constant while the chip density (number of gates per chip) is ever increasing. It implies that the cost of computer logic and memory circuitry has been reducing rapidly.
- **Greater Operating Speed:** The more is the density, the closer are the logic or memory elements, which implies shorter electrical paths and hence higher operating speed.
- **Smaller computers provide better portability**
- **Reduction in power and cooling requirements.**

The third generation computers mainly used SSI chips. One of the key concept which was brought forward during this time was the concept of the family of compatible computers. IBM mainly started this concept with its system/360 family.

A family of computers consists of several models. Each model is assigned a model number, for example, the IBM system/360 family have, Model 30,40, 50,65 and 75. The memory capacity, processing speed and cost increases as we go up the ladder. However, a lower model is compatible to higher model, that is, program written on a lower model can be executed on a higher model without any change. Only the time of execution is reduced as we go towards higher model and also a higher model has more

number of instructions. The biggest advantage of this family system was the flexibility in selection of model.

For example, if you had a limited budget and processing requirements you could possibly start with a relatively moderate model. As your business grows and your processing requirements increase, you can upgrade your computer with subsequent models depending on your need. However, please note that as you have gone for the computer of the same family, you will not be sacrificing investment on the already developed software as they can still be used on newer machines also.

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

S.No.	Feature	Characteristics while moving from lower member to higher member
1.	Instruction set	<ul style="list-style-type: none"> ❑ Similar instructions. ❑ Normally, the instructions set on a lower and member is a subset of higher end member. A program written on lower end member can be executed on a higher end member, but program written on higher end member may or may not get executed on lower end members.
2	Operating System	<ul style="list-style-type: none"> ❑ Same may have some additional features added in the operating system for the higher end members.
3	Speed of instruction execution	<ul style="list-style-type: none"> ❑ Increases
4	Number of I/O ports	<ul style="list-style-type: none"> ❑ Increases
5	Memory size	<ul style="list-style-type: none"> ❑ Increases
6	Cost	<ul style="list-style-type: none"> ❑ Increases

Figure 9: Characteristics of computer families

But how was the family concept implemented? Well, there were three main features of implementation. These were:

- Increased complexity of arithmetic logic unit;
- Increase in memory - CPU data paths; and
- Simultaneous access of data in higher end members.

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

- Application of IC circuits in the computer hardware replacing the discrete transistor component circuits. Thus, computers became small in physical size and less expensive.
- Use of Semiconductor (Integrated Circuit) memories as main memory replacing earlier technologies.
- The CPU design was made simple and CPU was made more flexible using a technique called microprogramming (will be discussed in later Blocks).
- Certain new techniques were introduced to increase the effective speed of program execution. These techniques were pipelining and multiprocessing. The details on these concepts can be found in the further readings.
- The operating system of computers was incorporated with efficient methods of sharing the facilities or resources such as processor and memory space automatically. These concepts are called multiprogramming and will be discussed in the course on operating systems.

1.5.5 Later Generations

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

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

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

However, let us discuss some of the important breakthroughs of VLSI technologies in this subsection:

Semiconductor Memories

Initially the IC technology was used for constructing processor, but soon it was realised that the same technology can be used for construction of memory. The first memory chip was constructed in 1970 and could hold 256 bits. The cost of this first chip was high. The cost of semiconductor memory has gone down gradually and presently the IC RAM's are quite cheap. Although the cost has gone down, the memory capacity per chip has increased. At present, we have reached the 1 Gbits on a single memory chip. Many new RAM technologies are available presently. We will give more details on these technologies later in Block 2.

Microprocessors

Keeping pace with electronics as more and more components were fabricated on a single chip, fewer chips were needed to construct a single processor. Intel in 1971 achieved the breakthrough of putting all the components on a single chip. The single chip processor is known as a microprocessor. The Intel 4004 was the first microprocessor. It was a primitive microprocessor designed for a specific application. Intel 8080, which came in 1974, was the first general-purpose microprocessor. This microprocessor was meant to be used for writing programs that can be used for general purpose computing. It was an 8-bit microprocessor. Motorola is another manufacturer in this area. At present 32 and 64 bit general-purpose microprocessors are already in the market. Let us look into the development of two most important series of microprocessors.

S.No.	Processor	Year	Memory size	Bus width	Comment
1	4004	1971	640 bytes	4 bits	Processor for specific applications
2.	8080	1974	64 KB	8 bits	First general-purpose micro-processor. It was used in development of first personal computer
3.	8086	1978	1 MB	16 bits	<input type="checkbox"/> Supported instruction cache memory or queue <input type="checkbox"/> Was the first powerful machine

4	80386	1985-1988 various versions.	4 G Byte Processor	32 bits	<input type="checkbox"/> First 32 bit <input type="checkbox"/> The processor supports multitasking
5	80486	1989-1991	4 g Byte	32 bits	<input type="checkbox"/> Use of powerful cache technology. <input type="checkbox"/> Supports pipeline based instruction execution <input type="checkbox"/> Contains built-in facility in the term of built-in math co-processor for floating point instructions
6	Pentium	1993-1995	64 G Bytes	32-bits and 64 bits	Uses superscalar techniques, that is execution of multiple instructions in parallel.
7	Pentium II	1997	64 G Bytes	64 bits	Contains instruction for handling processing of video, audio, graphics etc. efficiently. This technology was called MMX technology.
8	Pentium III	1999	64 B bytes	64 bits	Supports 3 D graphics software.
9	Pentium IV	2000	64 G Bytes	64 bits	Contains instructions for enhancement of multimedia. A very powerful processor
10	Itaium	2001	64 G bytes	64 bits	Supports massively parallel computing architecture.
11	Xeon	2001	64 G bytes	64 bits	Support hyper threading explained after this diagrams Outstanding performance and dependability: ideal for low cost servers

Figure 10: Some Important Developments in Intel Family of Microprocessors

Hyper-threading:

Nonthreaded program instructions are executed in a single order at a time, till the program completion. Suppose a program have 4 tasks namely A, B, C, D. Assume that each task consist of 10 instructions including few I/O instructions. A simple sequential execution would require $A \rightarrow B \rightarrow C \rightarrow D$ sequence.

In a threaded system these tasks of a single process/program can be executed in parallel provided is no data dependency. Since, there is only one processor these tasks will be executed in threaded system as interleaved threads, for example, 2 instructions of A 3 instruction of B, 1 instruction of C, 4 instruction of D, 2 instruction of C etc. till completion of the threads.

Hyper-threading allows 2 threads A & B to execute at the same time. How? Some of the more important parts of the CPU are duplicated. Thus, there exists 2 executing threads in the CPU at the exact same time. Please note that both these sections of the CPU works on the same memory space (as threads are the same program). Eventually dual CPUs will allow the computer to execute two threads in two separate programs at the same time.

Thus, Hyper-threading technology allows a single microprocessor to act like two separate threaded processors to the operating system and the application program that use it.

Hyper-threading requires software that has multiple threads and optimises speed of execution. A threaded program executes faster on hyper threaded machine. However, it should be noted that not all programs can be threaded.

The other architecture that has gained popularity over the last decade is the power PC family. These machines are reduced set instruction computer (RISC) based technologies. RISC technologies and are finding their application because of simplicity of Instructions. You will learn more about RISC in Block 3 of this course.

The IBM made an alliance with Motorola and Apple who has used Motorola 68000 chips in their Macintosh computer to create a POWER PC architecture. Some of the processors in this family are:

S.No.	Processor	Year	Bus Width	Comment
1	601	1993	32 bits	The first chip in power PC
2	603/603e	1994	32 bits	❑ Low cost machine, intended for low cost desktop
3	604/604e	1997	64 bits	❑ Low end server having superscalar architecture
4	G3	1997	64 bits	❑ Contains two levels of cache ❑ Shows good performance
5	G4	1999	64 bits	Increased speeds & parallelism of instruction execution
6	G6	2003	64 bits	Extremely fast multimedia capability rated very highly.

Figure 11: Power PC Family

The VLSI technology is still evolving. More and more powerful microprocessors and more storage space now is being put in a single chip. One question which we have still not answered, is: Is there any classification of computers? Well-for quite sometime computers have been classified under the following categories:

- Micro-controllers
- Micro-computers
- Engineering workstations
- Mini computers
- Mainframes
- Super computers
- Network computers.

Micro-controllers: These are specialised device controlling computers that contains all the functions of computers on a single chip. The chip includes provision for processing, data input and output display. These chips then can be embedded into various devices to make them more intelligent. Today this technology has reached

great heights. In fact it has been stated that embedded technology computing power available even in a car today is much more than what was available in the system on first lunar mission”.

Microcomputers

A microcomputer's CPU is a microprocessor. They are typically used as single user computer although present day microcomputers are very powerful. They support highly interactive environment specially like graphical user interface like windows. These computers are popular for home and business applications. The microcomputer originated in late 1970's. The first microcomputers were built around 8-bit microprocessor chips. What do we mean by an 8-bit chip? It means that the chip can retrieve instructions/data from storage, manipulate, and process an 8-bit data at a time or we can say that the chip has a built-in 8-bit data transfer path.

An improvement on 8-bit chip technology was seen in early 1980s, when a series of 16-bit chips namely 8086 and 8088 were introduced by Intel Corporation, each one with an advancement over the other.

8088 was an 8/16 bit chip i.e. an 8-bit path is used to move data between chip and primary storage (external path), but processing was done within the chip using a 16-bit path (internal path) at a time. 8086 was a 16/16-bit chip i.e. the internal and external paths both were 16 bits wide. Both these chips could support a primary basic memory of storage capacity of 1 Mega Byte (MB).

Similar to Intel's chip series exists another popular chip series of Motorola. The first 16-bit microprocessor of this series was MC 68000. It was a 16/32-bit chip and could support up to 16 MB of primary storage. Advancement over the 16/32 bit chips was the 32/32 chips. Some of the popular 32-bit chips were Intel's 80486 and MC 68020 chip.

Most of the popular microcomputers were developed around Intel's chips, while most of the minis and super minis were built around Motorola's 68000 series chips. With the advancement of display and VLSI technology a microcomputer was available in very small size. Some of these are laptops, note book computers etc. Most of these are of the size of a small notebook but equivalent capacity of an older mainframe.

Workstations

The workstations are used for engineering applications such as CAD/CAM or any other types of applications that require a moderate computing power and relatively high quality graphics capabilities. Workstations generally are required with high resolution graphics screen, large RAM, network support, a graphical user interface, and mass storage device. Some special type of workstation comes, without a disk. These are called diskless terminals/ workstations. Workstations are typically linked together to form a network. The most common operating systems for workstations are UNIX, Windows 2003 Server, and Solaris etc.

Please note that networking workstation means any computer connected to a local area network although it could be a workstation or a personal computer.

Workstations may be a client to server Computers. Server is a computer that is optimised to provide services to other connected computers through a network. Servers usually have powerful processors, huge memory and large secondary storage space.

Minicomputer

The term minicomputer originated in 1960s when it was realised that many computing tasks do not require an expensive contemporary mainframe computers but can be solved by a small, inexpensive computer.

The mini computers support multi-user environment with CPU time being shared among multiple users. The main emphasis in such computer is on the processing power and less for interaction. Most of the present day mini computers have proprietary CPU and operating system. Some common examples of a mini-computer are IBM AS/400 and Digital VAX. The major use of a minicomputer is in data processing application pertaining to departments/companies.

Mainframes

Mainframe computers are generally 32-bit machines or higher. These are suited to big organisations, to manage high volume applications. Few of the popular mainframe series were DEC, IBM, HP, ICL, etc. Mainframes are also used as central host computers in distributed systems. Libraries of application programs developed for mainframe computers are much larger than those of the micro or minicomputers because of their evolution over several decades as families of computing. All these factors and many more make the mainframe computers indispensable even with the popularity of microcomputers.

Supercomputers

The upper end of the state of the art mainframe machine are the supercomputers. These are amongst the fastest machines in terms of processing speed and use multiprocessing techniques, where a number of processors are used to solve a problem. There are a number of manufacturers who dominate the market of supercomputers-CRAY, IBM 3090 (with vector), NEC Fujitsu, PARAM by C-DEC are some of them. Lately, a range of parallel computing products, which are multiprocessors sharing common buses, have been in use in combination with the mainframe supercomputers. The supercomputers are reaching upto speeds well over 25000 million arithmetic operations per second. India has also announced its indigenous supercomputer. They support solutions to number crunching problems.

Supercomputers are mainly being used for weather forecasting, computational fluid dynamics, remote sensing, image processing, biomedical applications, etc. In India, we have one such mainframe supercomputer system-CRAY XMP-14, which is at present, being used by Meteorological Department.

Let us discuss about PARAM Super computer in more details

PARAM is a high-performances, scalable, industry standard computer. It has evolved from the concepts of distributes scalable computers supporting massive parallel processing in cluster of networked of computers. The PARAM's main advantages is its Scalability. PARAM can be constructed to perform Tera-floating point operations per second. It is a cost effective computer. It supports a number of application software.

PARAM is made using standard available components. It supports Sun's Ultra SPARC series servers and Solaris Operating System. It is based on open environments and standard protocols. It can execute any standard application available on Sun Solaris System.

Some of the applications that have been designed to run in parallel computational mode on PARAM include numerical weather forecasting, seismic data processing, Molecular modelling, finite element analysis, quantum chemistry.

It also supports many languages and Software Development platforms such as:

Solaris 2.5.1 Operating system on I/O and Server nodes, FORTRAN 77, FORTRAN 90, C and C++ language compilers, and tools for parallel program debugging, Visualisation and parallel libraries, Distributed Computing Environment, Data warehousing tools etc.

Check Your Progress 3

- 1) What is a general purpose machine?

.....

.....

.....

- 2) List the advantages of IC technology over discrete components.

.....

.....

.....

- 3) What is a family of computers? What are its characteristics?

.....

.....

1.6 SUMMARY

This completes our discussion on the introductory concepts of computer architecture. The von-Neumann architecture discussed in the unit is not the only architecture but many new architectures have come up which you will find in further readings. The information given on various topics such as interrupts, classification, history of computer although is exhaustive yet 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 his knowledge.

1.7 SOLUTIONS / ANSWERS**Check Your Progress 1**

1.
 - a) True
 - b) False
 - c) False
 - d) True
 - e) True
 - f) True
2. von Neumann architecture defines the basic architectural (logical) components of computer and their features. As per von Neumann the basic components of a computer are CPU (ALU+CU + Registers), I/O Devices, Memory and interconnection structures. von Neumann machine follows stored program concept that is, a program is loaded in the memory of computer prior to its execution.
3.
 - The instructions are not executed in sequence
 - More than one data items may be required during a single instruction execution.
 - Speed of CPU is very fast in comparison to I/O devices.

Check Your Progress 2

1.
 - i) False
 - ii) True
 - iii) True
 - iv) False
 - v) False, they may be acknowledged as per priority.
2. An interrupt is an external signal that occurs due to an exceptional event. It causes interruption in the execution of current program of CPU.
3. An interrupt is acknowledged by the CPU, which executes an interrupt cycle which causes interruption of currently executing program, and execution of interrupt servicing routine (ISR) for that interrupt.

Check Your Progress 3

1. A machine, which can be used for variety of applications and is not modeled only for specific applications. von Neumann machines are general-purpose machines since they can be programmed for any general application, while microprocessor based control systems are not general-purpose machines as they are specifically modeled as control systems.
2.
 - Low cost
 - Increased operating speed
 - Reduction in size of the computers
 - Reduction in power and cooling requirements
 - More reliable
3. The concept of the family of computer was floated by IBM 360 series where the features and cost increase from lower end members to higher end members.

UNIT 2 DATA REPRESENTATION

Structure	Page Nos.
2.0 Introduction	31
2.1 Objectives	31
2.2 Data Representation	31
2.3 Number Systems: A Look Back	32
2.4 Decimal Representation in Computers	36
2.5 Alphanumeric Representation	37
2.6 Data Representation For Computation	39
2.6.1 Fixed Point Representation	
2.6.2 Decimal Fixed Point Representation	
2.6.3 Floating Point Representation	
2.6.4 Error Detection And Correction Codes	
2.7 Summary	56
2.8 Solutions/ Answers	56

2.0 INTRODUCTION

In the previous Unit, you have been introduced to the basic configuration of the Computer system, its components and working. The concept of instructions and their execution was also explained. In this Unit, we will describe various types of binary notations that are used in contemporary computers for storage and processing of data. As far as instructions and their execution is concerned it will be discussed in detailed in the later blocks.

The Computer System is based on the binary system; therefore, we will be devoting this complete unit to the concepts of binary Data Representation in the Computer System. This unit will re-introduce you to the number system concepts. The number systems defined in this Unit include the Binary, Octal, and Hexadecimal notations. In addition, details of various number representations such as floating-point representation, BCD representation and character-based representations have been described in this Unit. Finally the Error detection and correction codes have been described in the Unit.

2.1 OBJECTIVES

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

- Use binary, octal and hexadecimal numbers;
 - Convert decimal numbers to other systems and vice versa;
 - Describe the character representation in computers;
 - Create fixed and floating point number formats;
 - Demonstrate use of fixed and floating point numbers in performing arithmetic operations; and
 - Describe the data error checking mechanism and error detection and correction codes.
-

2.2 DATA REPRESENTATION

The basic nature of a Computer is as an information transformer. Thus, a computer must be able to take input, process it and produce output. The key questions here are:

How is the Information represented in a computer?

Well, it is in the form of **Binary Digit** popularly called **Bit**.

How is the input and output presented in a form that is understood by us?

One of the minimum requirements in this case may be to have a representation for characters. Thus, a mechanism that fulfils such requirement is needed. In Computers information is represented in digital form, therefore, to represent characters in computer we need codes. Some common character codes are ASCII, EBCDIC, ISCII etc. These character codes are discussed in the subsequent sections.

How are the arithmetic calculations performed through these bits?

We need to represent numbers in binary and should be able to perform operations on these numbers.

Let us try to answer these questions, in the following sections. Let us first recapitulate some of the age-old concepts of the number system.

2.3 NUMBER SYSTEMS: A LOOK BACK

Number system is used to represent information in quantitative form. Some of the common number systems are binary, octal, decimal and hexadecimal.

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

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

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

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

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

For example, 101010 is a valid binary number.

Decimal equivalent of a binary number:

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

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

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

Octal number $(23.4)_8$.

(Please note the subscript 8 indicates it is an octal number, similarly, a subscript 2 will indicate binary, 10 will indicate decimal and H will indicate Hexadecimal number, in case no subscript is specified then number should be treated as decimal number or else whatever number system is specified before it.)

Decimal equivalent of Octal Number:

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

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

$F \times 16^1 + 2 \times 16^0$
$= (15 \times 16) + 2 \quad // \text{ (As F is equivalent to 15 for decimal)}$
$= 240 + 2$
$= (242)_{10}$

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

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

Solution:

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

Integer Part	Quotient on division by 2	Remainder on division by 2
43	21	1
21	10	1
10	05	0
05	02	1
02	01	0
01	00	1

↑
Read

Please note in the figure above that:

- The equivalent binary to the Integer part of the number is $(101011)_2$
- You will get the Integer part of the number, if you READ the remainder in the direction of the Arrow.

Fraction	On Multiplication by 2	Integer part after Multiplication	Read ↓
0.125	0.250	0	
0.250	0.500	0	
0.500	1.000	1	

Please note in the figure above that:

- The equivalent binary to the Fractional part of the number is 001.
- You will get the fractional part of the number, if you READ the Integer part of the number in the direction of the Arrow.

Thus, the number $(101011.001)_2$ is equivalent to $(43.125)_{10}$.

You can cross check it as follows:

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

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

2^6	2^5	2^4	2^3	2^2	2^1	2^0
64	32	16	8	4	2	1

- Step 1: Take the integer part e.g. 43, find the next lower or equal binary place value number, in this example it is 32. Place 1 at 32.
- Step 2: Subtract the place value from the number, in this case subtract 32 from 43, which is 11.
- Step 3: Repeat the two steps above till you get 0 at step 2.
- Step 4: On getting a 0 put 0 at all other place values.

These steps are shown as:

32	16	8	4	2	1	
32	16	8	4	2	1	
1	-	-	-	-	-1	$43 - 32 = 11$
1	-	1	-	-	-	$11 - 8 = 3$
1	-	1	-	1	-	$3 - 2 = 1$
1	-	1	-	1	1	$1 - 1 = 0$
1	0	1	0	1	1	is the required number.

You can extend this logic to fractional part also but in reverse order. Try this method with several numbers. It is fast and you will soon be accustomed to it and can do the whole operation in single iteration.

Conversion of Binary to Octal and Hexadecimal: The rules for these conversions are straightforward. For converting binary to octal, the binary number is divided into

groups of three, which are then combined by place value to generate equivalent octal. For example the binary number 1101011.00101 can be converted to Octal as:

1	101	011	.	001	01
001	101	011	.	001	010
1	5	3	.	1	2

(Please note the number is unchanged even though we have added 0 to complete the grouping. Also note the style of grouping before and after decimal. We count three numbers from right to left while after the decimal from left to right.)

Thus, the octal number equivalent to the binary number 1101011.00101 is $(153.12)_8$.

Similarly by grouping four binary digits and finding equivalent hexadecimal digits for it can make the hexadecimal conversion. For example the same number will be equivalent to $(6B.28)_H$.

110	1011	.	0010	1
0110	1011	.	0010	1000
6	11	.	2	8
6	B	.	2	8
(11 in hexadecimal is B)				
Thus equivalent hexadecimal number is $(6B.28)_H$				

Conversely, we can conclude that a hexadecimal digit can be broken down into a string of binary having 4 places and an octal can be broken down into string of binary having 3 place values. Figure 1 gives the binary equivalents of octal and hexadecimal numbers.

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

Figure 1: Binary equivalent of octal and hexadecimal digits

Check Your Progress 1

- 1) Convert the following binary numbers to decimal.
 - i) 1100.1101
 - ii) 10101010

.....

.....

.....

.....
- 2) Convert the following decimal numbers to binary.
 - i) 23
 - ii) 49.25
 - iii) 892

.....

.....

.....

.....
- 3) Convert the numbers given in question 2 to hexadecimal from decimal or from the binary.

.....

.....

.....

.....

2.4 DECIMAL REPRESENTATION IN COMPUTERS

The binary number system is most natural for computer because of the two stable states of its components. But, unfortunately, this is not a very natural system for us as we work with decimal number system. So, how does the computer perform the arithmetic? One solution that is followed in most of the computers is to convert all input values to binary. Then the computer performs arithmetic operations and finally converts the results back to the decimal number so that we can interpret it easily. Is there any alternative to this scheme? Yes, there exists an alternative way of performing computation in decimal form but it requires that the decimal numbers should be coded suitably before performing these computations. Normally, the decimal digits are coded in 7-8 bits as alphanumeric characters but for the purpose of arithmetic calculations the decimal digits are treated as four bit binary code. As we know 2 binary bits can represent $2^2 = 4$ different combinations, 3 bits can represent $2^3 = 8$ combinations, and similarly, 4 bits can represent $2^4 = 16$ combinations. To represent decimal digits into binary form we require 10 combinations, but we need to have a 4-digit code. One such simple representation may be to use first ten binary combinations to represent the ten decimal digits. These are popularly known as Binary Coded Decimals (BCD). Figure 2 shows the binary coded decimal numbers.

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

Figure 2: Binary Coded Decimals (BCD)

Let us represent 43.125 in BCD.

4	3	.	1	2	5
0100	0011	.	0001	0010	0101

Compare the equivalent BCD with equivalent binary value. Both are different.

2.5 ALPHANUMERIC REPRESENTATION

But what about alphabets and special characters like +, -, * etc.? How do we represent these in a computer? A set containing alphabets (in both cases), the decimal digits (10 in number) and special characters (roughly 10-15 in numbers) consist of at least 70-80 elements.

ASCII

One such standard code that allows the language encoding that is popularly used is ASCII (American Standard Code for Information Interchange). This code uses 7 bits

to represent 128 characters, which include 32 non-printing control characters, alphabets in lower and upper case, decimal digits, and other printable characters that are available on your keyboard. Later as there was need for additional characters to be represented such as graphics characters, additional special characters etc., ASCII was extended to 8 bits to represent 256 characters (called Extended ASCII codes). There are many variants of ASCII, they follow different code pages for language encoding, however, having the same format. You can refer to the complete set of ASCII characters on the web. The extended ASCII codes are the codes used in most of the Microcomputers.

The major strength of ASCII is that it is quite elegant in the way it represents characters. It is easy to write a code to manipulate upper/lowercase ASCII characters and check for valid data ranges because of the way of representation of characters.

In the original ASCII the 8th bit (the most significant bit) was used for the purpose of error checking as a check bit. We will discuss more about the check bits later in the Unit.

EBCDIC

Extended Binary Coded Decimal Interchange Code (EBCDIC) is a character-encoding format used by IBM mainframes. It is an 8-bit code and is NOT Compatible to ASCII. It had been designed primarily for ease of use of punched cards. This was primarily used on IBM mainframes and midrange systems such as the AS/400. Another strength of EBCDIC was the availability of wider range of control characters for ASCII. The character coding in this set is based on binary coded decimal, that is, the contiguous characters in the alphanumeric range are represented in blocks of 10 starting from 0000 binary to 1001 binary. Other characters fill in the rest of the range. There are four main blocks in the EBCDIC code:

0000 0000 to 0011 1111	Used for control characters
0100 0000 to 0111 1111	Punctuation characters
1000 0000 to 1011 1111	Lowercase characters
1100 0000 to 1111 1111	Uppercase characters and numbers.

There are several different variants of EBCDIC. Most of these differ in the punctuation coding. More details on EBCDIC codes can be obtained from further reading and web pages on EBCDIC.

Comparison of ASCII and EBCDIC

EBCDIC is an easier to use code on punched cards because of BCD compatibility. However, ASCII has some of the major advantages on EBCDIC. These are: While writing a code, since EDCDIC is not contiguous on alphabets, data comparison to continuous character blocks is not easy. For example, if you want to check whether a character is an uppercase alphabet, you need to test it in range A to Z for ASCII as they are contiguous, whereas, since they are not contiguous range in EDCDIC these may have to be compared in the ranges A to I, J to R, and S to Z which are the contiguous blocks in EDCDIC.

Some of the characters such as `[]\{}^~|` are missing in EBCDIC. In addition, missing control characters may cause some incompatibility problems.

UNICODE

This is a newer International standard for character representation. Unicode provides a unique code for every character, irrespective of the platform, Program and Language. Unicode Standard has been adopted by the Industry. The key players that have adopted Unicode include Apple, HP, IBM, Microsoft, Oracle, SAP, Sun, Sybase, Unisys and many other companies. Unicode has been implemented in most of the

latest client server software. Unicode is required by modern standards such as XML, Java, JavaScript, CORBA 3.0, etc. It is supported in many operating systems, and almost all modern web browsers. Unicode includes character set of Dev Nagari. The emergence of the Unicode Standard, and the availability of tools supporting it, is among the most significant recent global software technology trends.

One of the major advantages of Unicode in the client-server or multi-tiered applications and websites is the cost saving over the use of legacy character sets that results in targeting website and software products across multiple platforms, languages and countries without re-engineering. Thus, it helps in data transfer through many different systems without any compatibility problems. In India the suitability of Unicode to implement Indian languages is still being worked out.

Indian Standard Code for information interchange (ISCII)

The ISCII is an eight-bit code that contains the standard ASCII values till 127 from 128-225 it contains the characters required in the ten Brahmi-based Indian scripts. It is defined in IS 13194:1991 BIS standard. It supports INSCRIPT keyboard which provides a logical arrangement of vowels and consonants based on the phonetic properties and usage frequencies of the letters of Bramhi-scripts. Thus, allowing use of existing English keyboard for Indian language input. Any software that uses ISCII codes can be used in any Indian Script, enhancing its commercial viability. It also allows transliteration between different Indian scripts through change of display mode.

2.6 DATA REPRESENTATION FOR COMPUTATION

As discussed earlier, binary codes exist for any basic representation. Binary codes can be formulated for any set of discrete elements e.g. colours, the spectrum, the musical notes, chessboard positions etc. In addition these binary codes are also used to formulate instructions, which are advanced form of data representation. We will discuss about instructions in more detail in the later blocks. But the basic question which remains to be answered is:

How are these codes actually used to represent data for scientific calculations?

The computer is a discrete digital device and stores information in flip-flops (see Unit 3, 4 of this Block for more details), which are two state devices, in binary form. Basic requirements of the computational data representation in binary form are:

- Representation of sign
- Representation of Magnitude
- If the number is fractional then binary or decimal point, and
- Exponent

The solution to sign representation is easy, because sign can be either positive or negative, therefore, one bit can be used to represent sign. By default it should be the left most bit (in most of the machines it is the Most Significant Bit).

Thus, a number of n bits can be represented as $n+1$ bit number, where $n+1^{\text{th}}$ bit is the sign bit and rest n bits represent its magnitude (Please refer to Figure 3).

The decimal position can be represented by a position between the flip-flops (storage cells in computer). But, how can one determine this decimal position? Well to simplify the representation aspect two methods were suggested: (1) Fixed point representation where the binary decimal position is assumed either at the beginning or at the end of a number; and (2) Floating point representation where a second register is used to keep the value of exponent that determines the position of the binary or decimal point in the number.

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

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

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

Solution:

9's complement: The 9's complement is obtained by subtracting each digit of the number from 9 (the highest digit value). Let us assume that we want to represent a maximum of four decimal digit number range. 9's complement can be used for BCD numbers.

9's complement of 256	9	9	9	9
	-0	-2	-5	-6
	9	7	4	3

Similarly, for obtaining 1's complement for a binary number we have to subtract each binary digit of the number from the digit 1.

10's complement: Adding 1 in the 9's complement produces the 10's complement.
 $10's \text{ complement of } 0256 = 9743 + 1 = 9744$

Please note on adding the number and its 9's complement we get 9999 (the maximum possible number that can be represented in the four decimal digit number range) while on adding the number and its 10's complement we get 10000 (The number just higher than the range. This number cannot be represented in four digit representation.)

Example3: Find 1's and 2's complement of 1010 using only four-digit representation.

Solution:

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

1	1	1	1
-1	-0	-1	-0
0	1	0	1

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

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

2's complement: Adding 1 in 1's complement will generate the 2's complement

The number is	1	0	1	0
The 1's complement is	0	1	0	1
For 2's complement add 1 in 1's complement	-	-	-	1
Please note that $1+1 = 1\ 0$ in binary	0	1	1	0

↑
↑
 Most Significant bit Least significant bit

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

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

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

The number is	0	0	1	0	0	1	0	0
The 2's complement is	1	1	0	1	1	1	0	0

└───┘
 No change in these bits

The number is	1	0	0	0	0	0	0	0
The 2's complement is	1	0	0	0	0	0	0	0

└──────────┘
 No change in number and its 2's Complement, a special case

The number is	0	0	1	0	1	0	0	1
The 2's complement is	1	1	0	1	0	1	1	1

└─┘
 No change in this bit only

2.6.1 Fixed Point Representation

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

- Signed magnitude representation

- Signed 1's complement representation, or
 - Signed 2's complement representation.
- (Assumption: size of register = 8 bits including the sign bit)

Signed Magnitude Representation

Decimal Number	Representation (8 bits)	
	Sign Bit	Magnitude (7 bits)
+6	0	000 0110
-6	1	000 0110
No change in the Magnitude, only sign bit changes		

Signed 1's Complement Representation

Decimal Number	Representation (8 bits)	
	Sign Bit	Magnitude/ 1's complement for negative number (7 bits)
+6	0	000 0110
-6	1	111 1001
For negative number take 1's complement of all the bits (including sign bit) of the positive number		

Signed 2's Complement Representation

Decimal Number	Representation (8 bits)	
	Sign Bit	Magnitude/ 1's complement for negative number (7 bits)
+6	0	000 0110
-6	1	111 1010
For negative number take 2's complement of all the bits (including sign bit) of the positive number		

Arithmetic addition

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

Example 4: Add 25 and -30 in binary using 8 bit registers, using:

- Signed magnitude representation
- Signed 1's complement
- Signed 2's complement

Solution:

Number	Signed Magnitude Representation	
	Sign Bit	Magnitude
+25	0	001 1001
-25	1	001 1001
+30	0	001 1110
-30	1	001 1110

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

Number	Signed Magnitude Representation	
	Sign Bit	Magnitude
+25	0	001 1001
-25	1	110 0111
+30	0	001 1110
-30	1	110 0010

Now let us perform addition using signed 2's complement notation:

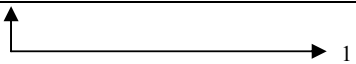
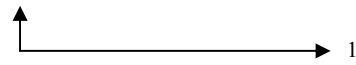
Operation	Decimal equivalent number	Signed 2's complement representation				Comments
		Carry out	Sign out	Magnitude		
Addition of two positive number	+25	-	0	001	1001	Simple binary addition. There is no carry out of sign bit
	+30	-	0	001	1110	
	+55	0	0	011	0111	
Addition of smaller Positive and larger negative Number	+25	-	0	001	1001	Perform simple binary addition. No carry in to the sign bit and no carry out of the sign bit
	-30	-	1	110	0010	
	-05	0	1	111	1011	
Positive value of result	+05	0	1	000	0101	2's complement of above result
Addition of larger Positive and smaller negative Number	-25	-	1	110	0111	Perform simple binary addition. No carry in to the sign bit and carry out of the sign bit
	+30	-	1	001	1110	
	+05	1	0	000	0101	
		↑ Discard the carry out bit				
Addition of two negative Numbers	-25	-	1	110	0111	Perform simple binary addition. There is carry in to the sign bit and carry out of the sign bit No overflow
	-30	-	1	110	0010	
	-55	1	1	110	1001	
		↑ Discard the carry out bit				
Positive value of result	+55	-	0	011	0111	2's complemnt of above result

Please note how easy it is to add two numbers using signed 2's Complement. This procedure requires only one control decision and only one circuit for adding the two numbers. But it puts on additional condition that the negative numbers should be stored in signed 2's complement notation in the registers. This can be achieved by

complementing the positive number bit by bit and then incrementing the resultant by 1 to get signed 2's complement.

Signed 1's complement representation

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

Operation	Decimal equivalent number	Signed 1's complement representation				Comments
		Carry out	Sign out	Magnitude		
Addition of two positive number	+25	-	0	001	1001	Simple binary addition. There is no carry out of sign bit
	+30	-	0	001	1110	
	+55	0	0	001	0111	
Addition of smaller Positive and larger negative Number	+25	-	0	001	1001	Perform simple binary addition. No carry in to the sign bit and no carry out of the sign bit
	-30	-	1	110	0001	
	-05	0	1	111	1011	
Positive value of result	+05	-	0	000	0101	1's complement of above result
Addition of larger Positive and smaller negative Number	-25	-	1	110	0111	There is carry in to the sign bit and carry out of the sign bit. The carry out is added it to the Sum bit and then discard no overflow.
	+30	-	0	001	1110	
		1	0	000	0101	
	Add carry to Sum and discard it					
	+05	-	0	000	0101	
Addition of two negative Numbers	-25	-	1	110	0111	Perform simple binary addition. There is carry in to the sign bit and carry out of the sign bit No overflow
	-30	-	1	110	0010	
	-55	1	1	100	0111	
	Add carry to sum and discard it					
		-	1	100	1000	
Positive value of result	+55	-	0	011	0111	1's complemnt of above result

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

Representation	+ 0			-0
Signed magnitude	0	000 0000	1	000 0000
Signed 1's complement	0	000 0000	1	111 1111

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

+0 in 2's Complement Notation: 0 000 0000

-0 in 1's complement notation: 1 111 1111

Add 1 for 2's complement: 1

Discard the Carry Out 1 0 000 0000

Thus, -0 in 2's complement notation is same as +0 and is equal to 0 000 0000. Thus, both +0 and -0 are same in 2's complement notation. This is an added advantage in favour of 2's complement notation.

The highest number that can be accommodated in a register, also depends on the type of representation. In general in an 8 bit register 1 bit is used as sign, therefore, the rest 7 bits can be used for representing the value. The highest and the lowest numbers that can be represented are:

For signed magnitude representation

	$(2^7 - 1)$ to $-(2^7 - 1)$
	$= (128 - 1)$ to $-(128 - 1)$
	$= 127$ to -127

For signed 1's complement 127 to -127

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

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

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

Overflow: An overflow is said to have occurred when the sum of two n digits number occupies $n+1$ digits. This definition is valid for both binary as well as decimal digits.

What is the significance of overflow for binary numbers?

Well, the overflow results in errors during binary arithmetic as the numbers are represented using a fixed number of digits also called the size of the number. Any value that results from computation must be less than the maximum of the allowed value as per the size of the number. In case, a result of computation exceeds the maximum size, the computer will not be able to represent the number correctly, or in other words the number has overflowed. Every computer employs a limit for representing numbers e.g. in our examples we are using 8 bit registers for calculating the sum. But what will happen if the sum of the two numbers can be accommodated in 9 bits? Where are we going to store the 9th bit, The problem will be better understood by the following example.

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

65	0	100 0001
75	0	100 1011
140	1	000 1100

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

How does the computer know that overflow has occurred?

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

Another simple test of overflow is: if the sign of both the operands is same during addition, then overflow must have occurred if the sign of resultant is different than that of sign of any operand.

For example

Decimal	Carry out	Sign bit	2's Complement Mantissa	Decimal	Carry out	Sign bit	2's Complement Mantissa
-65		1	011 1111	-65		1	011 1111
-15		1	111 0001	-75		1	111 0001
-80	1	1	011 0000	-140	1	0	111 0100

Carry into Sign bit = 1
Carry out of sign bit = 1
Therefore, NO OVERFLOW

Carry into Sign bit = 0
Carry out of Sign bit = 1
Therefore, OVERFLOW

Thus, overflow has occurred, i.e. the arithmetic results so calculated have exceeded the capacity of the representation. This overflow also implies that the calculated results will be erroneous.

2.6.2 Decimal Fixed Point Representation

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

For example, a simple decimal number – 2156 can be represented as:

1101 0010 0001 0101 0110

Sign

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

Check Your Progress 2

- 1) Write the BCD equivalent for the three numbers given below:
 - i) 23
 - ii) 49.25
 - iii) 892

2) Find the 1's and 2's complement of the following fixed-point numbers.

i) 10100010

ii) 00000000

iii) 11001100

3) Add the following numbers in 8-bit register using signed 2's complement notation

i) +50 and -5

ii) +45 and -65

iii) +75 and +85

Also indicate the overflow if any.

2.6.3 Floating Point Representation

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

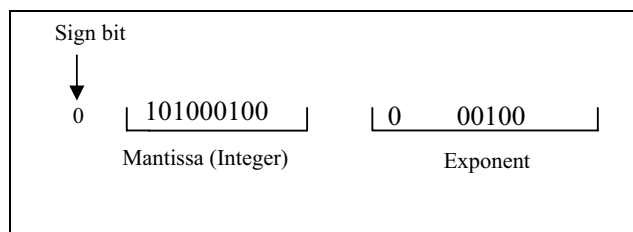
Example 1: A decimal + 12.34 in a typical floating point notation can be represented in any of the following two forms:

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

Floating-point numbers are often represented in normalised forms. A floating point number whose mantissa does not contain zero as the most significant digit of the number is considered to be in normalised form. For example, a BCD mantissa + 370 which is 0 0011 0111 0000 is in normalised form because these leading zero's are not part of a zero digit. On the other hand a binary number 0 01100 is not in a normalised form. The normalised form of this number is:

0	1100	0100
Sign	Normalised Mantissa	Exponent (assuming fractional Mantissa)

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



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

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

The following figure shows a format of a 32-bit floating-point number.

0	1	8	9	31
Sign	Biased Exponent = 8 bits		Significand = 23 bits	

Figure 4: Floating Point Number Representation

The characteristics of a typical floating-point representation of 32 bits in the above figure are:

- Left-most bit is the sign bit of the number;
- Mantissa or significand should be in normalised form;
- The base of the number is 2, and
- A value of 128 is added to the exponent. (Why?) This is called a bias.

A normal exponent of 8 bits normally can represent exponent values as 0 to 255. However, as we are adding 128 for getting the biased exponent from the actual exponent, the actual exponent values represented in the range will be - 128 to 127.

Now, let us define the range that a normalised mantissa can represent. Let us assume that our present representations has the normalised mantissa, thus, the left most bit

cannot be zero, therefore, it has to be 1. Thus, it is not necessary to store this first bit and it is being assumed **implicitly** for the number. Therefore, a 23-bit mantissa can represent $23 + 1 = 24$ bit mantissa in our representation.

Thus, the smallest mantissa value may be:

The implicit first bit as 1 followed by 23 zero's, that is,

0.1000 0000 0000 0000 0000 0000

Decimal equivalent $= 1 \times 2^{-1} = 0.5$

The Maximum value of the mantissa:

The implicit first bit 1 followed by 23 one's, that is,

0.1111 1111 1111 1111 1111 1111

Decimal equivalent:

For finding binary equivalent let us add 2^{-24} to above mantissa as follows:

Binary: 0.1111 1111 1111 1111 1111 1111

+0.0000 0000 0000 0000 0000 0001 $= 2^{-24}$

1.0000 0000 0000 0000 0000 0000 $= 1$

$= (1 - 2^{-24})$

Therefore, in normalised mantissa and biased exponent form, the floating-point number format as per the above figure, can represent binary floating-point numbers in the range:

Smallest Negative number

Maximum mantissa and maximum exponent

$$= - (1 - 2^{-24}) \times 2^{127}$$

Largest negative number

Minimum mantissa and Minimum exponent

$$= -0.5 \times 2^{-128}$$

Smallest positive number

$$= 0.5 \times 2^{-128}$$

Largest positive number

$$= (1 - 2^{-24}) \times 2^{127}$$

Figure 5: Binary floating-point number range for given 32 bit format

In floating point numbers, the basic trade-off is between the range of the numbers and accuracy, also called the precision of numbers. If we increase the exponent bits in 32-bit format, the range can be increased, however, the accuracy of numbers will go down, as size of mantissa will become smaller. Let us take an example, which will clarify the term precision. Suppose we have one bit binary mantissa then we can represent only 0.10 and 0.11 in the normalised form as given in above example (having an implicit 1). The values such as 0.101, 0.1011 and so on cannot be represented as complete numbers. Either they have to be approximated or truncated and will be represented as either 0.10 or 0.11. Thus, it will create a truncation or round off error. The higher the number of bits in mantissa better will be the precision.

In floating point numbers, for increasing both precision and range more number of bits are needed. This can be achieved by using double precision numbers. A double precision format is normally of 64 bits.

Institute of Electrical and Electronics Engineers (IEEE) is a society, which has created lot of standards regarding various aspects of computer, has created IEEE standard 754 for floating-point representation and arithmetic. The basic objective of developing this standard was to facilitate the portability of programs from one to another computer. This standard has resulted in development of standard numerical capabilities in various microprocessors. This representation is shown in figure 6.

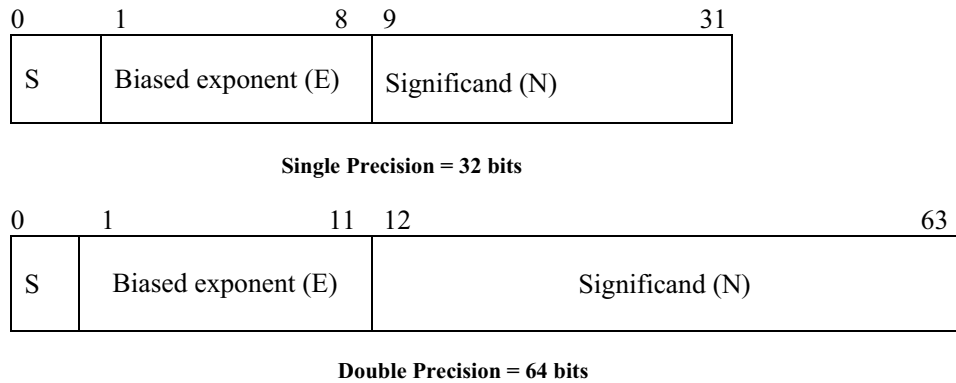


Figure 6: IEEE Standard 754 format

Figure 7 gives the floating-point numbers specified by the IEEE Standard 754.

Single Precision Numbers (32 bits)

Exponent (E)	Significand (N)	Value / Comments
255	Not equal to 0	Do not represent a number
255	0	- or + ∞ depending on sign bit
0 < E < 255	Any	$\pm (1.N) 2^{E-127}$ For example, if S is zero that is positive number. N=101 (rest 20 zeros) and E=207 Then the number is = $+(1.101) 2^{207-127}$ = $+ 1.101 \times 2^{80}$
0	Not equal to 0	$\pm (0.N) 2^{-126}$
0	0	± 0 depending on the sign bit.
Double precision Numbers (64 bits)		
Exponent (E)	Significand (N)	Value / Comments
2047	Not equal to 0	Do not represent a number

2047	0	- or $+\infty$ depending on the sign bit
$0 < E < 2047$	Any	$\pm (1.N) 2^{E-1023}$
0	Not equal to 0	$\pm (0.N) 2^{-1022}$
0	0	± 0 depending on the sign bit

Figure 7: Values of floating point numbers as per IEEE standard 754

Please note that IEEE standard 754 specifies plus zero and minus zero and plus infinity and minus infinity. Floating point arithmetic is more sticky than fixed point arithmetic. For floating point addition and subtraction we have to follow the following steps:

- Check whether a typical operand is zero
- Align the significand such that both the significands have same exponent
- Add or subtract the significand only and finally
- The significand is normalised again

These operations can be represented as

$$x + y = (N_x \times 2^{Ex-Ey} + N_y) \times 2^{Ey}$$

$$\text{and } x - y = (N_x \times 2^{Ex-Ey} - N_y) \times 2^{Ey}$$

Here, the assumption is that exponent of x (E_x) is greater than exponent of y (E_y), N_x and N_y represent significand of x and y respectively.

While for multiplication and division operations the significand need to be multiplied or divided respectively, however, the exponents are to be added or to be subtracted respectively. In case we are using bias of 128 or any other bias for exponents then on addition of exponents since both the exponents have bias, the bias gets doubled. Therefore, we must subtract the bias from the exponent on addition of exponents. However, bias is to be added if we are subtracting the exponents. The division and multiplication operation can be represented as:

$$x \times y = (N_x \times N_y) \times 2^{Ex+Ey}$$

$$x \div y = (N_x \div N_y) \times 2^{Ex-Ey}$$

For more details on floating point arithmetic you can refer to the further readings.

2.6.4 Error Detection and Correction Codes

Before we wind up the data representation in the context of today's computers one must discuss about the code, which helps in recognition and correction of errors. Computer is an electronic media; therefore, there is a possibility of errors during data transmission. Such errors may result from disturbances in transmission media or external environment. But what is an error in binary bit? An error bit changes from 0 to 1 or 1 to 0. One of the simplest error detection codes is called parity bit.

Parity bit: A parity bit is an error detection bit added to binary data such that it makes the total number of 1's in the data either odd or even. For example, in a seven bit data 0110101 an 8th bit, which is a parity bit may be added. If the added parity bit is even parity bit then the value of this parity bit should be zero, as already four 1's exists in the 7-bit number. If we are adding an odd parity bit then it will be 1, since we already have four 1 bits in the number and on adding 8th bit (which is a parity bit) as 1 we are making total number of 1's in the number (which now includes parity bit also) as 5, an odd number.

Similarly in data 0010101 Parity bit for even parity is 1
 Parity bit for odd parity is 0

But how does the parity bit detect an error? We will discuss this issue in general as an error detection and correction system (Refer figure 8).

The error detection mechanism can be defined as follows:

Figure 8: Error detection and correction

The Objective : Data should be transmitted between a source data pair reliably, indicating error, or even correcting it, if possible.

The Process:

- An error detection function is applied on the data available at the source end an error detection code is generated.
- The data and error detection or correction code are stored together at source.
- On receiving the data transmission request, the stored data along with stored error detection or correction code are transmitted to the unit requesting data (Destination).
- On receiving the data and error detection/correction code from source, the destination once again applies same error detection/correction function as has been applied at source on the data received (but not on error detection/correction code received from source) and generates destination error detection/correction code.
- Source and destination error codes are compared to flag or correct an error as the case may be.

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

Hamming Error-Correcting Code: Richard Hamming at Bell Laboratories devised this code. We will just introduce this code with the help of an example for 4 bit data.

Let us assume a four bit number b_4, b_3, b_2, b_1 . In order to build a simple error detection code that detects error in one bit only, we may just add an odd parity bit. However, if we want to find which bit is in error then we may have to use parity bits for various combinations of these 4 bits such that a bit error can be identified uniquely. For example, we may create four parity sets as

Source Parity

Destination Parity

b1, b2, b3	P1	D1
b2, b3, b4	P2	D2
b3, b4, b1	P3	D3
b1, b2, b3, b4	P4	D4

Now, a very interesting phenomena can be noticed in the above parity pairs. Suppose data bit b1 is in error on transmission then, it will cause change in destination parity D1, D3, D4.

**ERROR IN
(one bit only)** **Cause change in Destination Parity**

b1	D1, D3, D4
b2	D1, D2, D4
b3	D1, D2, D3, D4
b4	D2, D3, D4

Figure 9 : The error detection parity code mismatch

Thus, by simply comparing parity bits of source and destination we can identify that which of the four bits is in error. This bit then can be complemented to remove error. Please note that, even the source parity bit can be in error on transmission, however, under the assumption that only one bit (irrespective of data or parity) is in error, it will be detected as only one destination parity will differ.

What should be the length of the error detection code that detects error in one bit? Before answering this question we have to look into the comparison logic of error detection. The error detection is done by comparing the two 'i' bit error detection and correction codes fed to the comparison logic bit by bit (refer to figure 8). Let us have comparison logic, which produces a zero if the compared bits are same or else it produces a one.

Therefore, if similar Position bits are same then we get zero at that bit Position, but if they are different, that is, this bit position may point to some error, then this Particular bit position will be marked as one. This way a matching word is constructed. This matching word is 'i' bit long, therefore, can represent 2^i values or combinations.

For example, a 4-bit matching word can represent $2^4=16$ values, which range from 0 to 15 as:

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

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

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

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

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

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

$$i=4 \quad 2^i - 1 = 2^4 - 1 = 15; \text{ RHS} = 8+4 = 12$$

Therefore, for an eight-bit word we need to have at least four-bit error correction code for detecting and correcting errors in a single bit during transmission.

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

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

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

Let us explain this with the help of an example:

Let us assume 4 bit data as 1010

The logic is shown in the following table:

Source:

Source Data

Odd parity bits at source

b4	b3	b2	b1	P1 (b1, b2, b3)	P2 (b2, b3, b4)	P3 (b3, b4, b1)	P4 (b1, b2, b3, b4)
1	0	1	0	0	1	0	1

This whole information, that is (data and P1 to P4), is transmitted.

Assuming one bit error in data.

Case 1: Data received as 1011 (Error in b1)

b4	b3	b2	b1	D1 (b1, b2, b3)	D2 (b2, b3, b4)	D3 (b3, b4, b1)	D4 (b1, b2, b3, b4)
1	0	1	0	0	1	0	1

Thus, $P1 - D1$, $P3 - D3$, $P4 - D4$ pair differ, thus, as per Figure 9, b1 is in error, so correct it by complementing b1 to get correct data 1010.

Case 2: Data Received as 1000 (Error in b2)

b4	b3	b2	b1	D1 (b1, b2, b3)	D2 (b2, b3, b4)	D3 (b3, b4, b1)	D4 (b1, b2, b3, b4)
1	0	0	0	0	1	0	0

Thus, $P1 - D1$, $P2 - D2$, $P4 - D4$ pair differ, thus, as per figure 9, bit b2 is in error. So correct it by complementing it to get correct data 1010.

Case 3:

Now let us take a case when data received is correct but on receipt one of the parity bit, let us say P4 become 0. Please note in this case since data is 1010 the destination parity bits will be $D1=0$, $D2=1$, $D3=0$, $D4=1$. Thus, $P1 - D1$, $P2 - D2$, $P3 - D3$, will be same but $P4 - D4$ differs. This does not belong to any of the combinations in Figure 9. Thus we conclude that P4 received is wrong.

Please not that all these above cases will fail in case error is in more than one bits. Let us see by extending the above example.

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

Case: 4

Let us assume now that two bit errors occur in data.

Data received:

b4 b3 b2 b1
1 1 0 0

b4	b3	b2	b1	D1 (b1, b2, b3)	D2 (b2, b3, b4)	D3 (b3, b4, b1)	D4 (b1, b2, b3, b4)
1	0	0	0	0	1	0	0

Thus, on -matching we find P3-D3 pair does not match.

However, this information is wrong. Such problems can be identified by adding one more bit to this Single Error Detection Code. This is called Double Error Detection bit (P5, D5).

So our data now is

b4 b3 b2 b1 P1 P2 P3 P4 P5 (Overall parity of whole data)
1 0 1 0 0 1 0 1 1

Data receiving end.

b4 b3 b2 b1 D1 D2 D3 D4 D5
1 1 0 0 0 1 1 1 0

D5–P5 mismatch indicates that there is double bit error, so do not try to correct error, instead asks the sender to send the data again. Thus, the name single error correction, but double error detection, as this code corrects single bit errors but only detects error in two bit.

Check Your Progress 3

- 1) Represent the following numbers in IEEE-754 floating point single precision number format:

- i) 1010. 0001
- ii) –0.0000111

- 2) Find the even and odd parity bits for the following 7-bit data:

- i) 0101010
- ii) 0000000
- iii) 1111111
- iv) 1000100

.....
.....

- 3) Find the length of SEC code and SEC-DED code for a 16-bit word data transfer.

2.7 SUMMARY

This unit provides an in-depth coverage of the data representation in a computer system. We have also covered aspects relating to error detection mechanism. The unit covers number system, conversion of number system, conversion of numbers to a different number system. It introduces the concept of computer arithmetic using 2's complement notation and provides introduction to information representation codes like ASCII, EBCDIC, etc. The concept of floating point numbers has also been covered with the help of a design example and IEEE-754 standard. Finally error detection and correction mechanism is detailed along with an example of SEC & SEC-DED code.

The information given on various topics such as data representation, error detection codes etc. although exhaustive yet 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 by leaps and bounds. In addition to further reading the student is advised to study several Indian Journals on computers to enhance his knowledge.

2.8 SOLUTIONS/ANSWERS

Check Your Progress 1

1.

$$(i) \begin{array}{cccccccc} 2^3 & 2^2 & 2^1 & 2^0 & 2^{-1} & 2^{-2} & 2^{-3} & 2^{-4} \\ 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 \end{array}$$

$$\text{thus; Integer} = (1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0) = (2^3 + 2^2) = (8 + 4) = 12$$

$$\text{Fraction} = (1 \times 2^{-1} + 1 \times 2^{-2} + 0 \times 2^{-3} + 1 \times 2^{-4}) = 2^{-1} + 2^{-2} + 2^{-4} = 0.5 + 0.125 + 0.0625 = 0.6875$$

ii) 10101010

$$\begin{array}{cccccccc} 2^7 & 2^6 & 2^5 & 2^4 & 2^3 & 2^2 & 2^1 & 2^0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 128 & 64 & 32 & 16 & 8 & 4 & 2 & 1 \\ =1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \end{array}$$

The decimal equivalent is

$$= 1 \times 128 + 0 \times 64 + 1 \times 32 + 0 \times 16 + 1 \times 8 + 0 \times 4 + 1 \times 2 + 0 \times 1$$

$$= 128 + 32 + 8 + 2 = 170$$

2.

$$(i) \begin{array}{cccccc} 16 & 8 & 4 & 2 & 1 \\ 1 & 0 & 1 & 1 & 1 \end{array}$$

ii) Integer is 49.

32	16	8	4	2	1
----	----	---	---	---	---

1	1	0	0	0	1
---	---	---	---	---	---

Fraction is 0.25

1/2	1/4	1/8	1/16
-----	-----	-----	------

0	1	0	0
---	---	---	---

The decimal number 49.25 is 110001.010

iii)

512	256	128	64	32	16	8	4	2	1
-----	-----	-----	----	----	----	---	---	---	---

1	1	0	1	1	1	1	1	0	0
---	---	---	---	---	---	---	---	---	---

The decimal number 892 in binary is 1101111100

3)

i) Decimal to Hexadecimal

16) 23 (1

-16

7

Hexadecimal is 17

Binary to Hexadecimal (hex)

= 1 0111 (from answer of 2 (i)

0001	0111
1	7

ii)

49.25 or 110001.010

Decimal to hex

Integer part = 49

16) 49 (3

-48

1

Integer part = 31

Fraction part = .25 × 16

= 4.000 So fraction part = 4

Hex number is 31.4

Binary to hex	11	0001	.	010
=	0011	0001	.	0100
=	3	1	.	4
=	31.4			

iii) 892 or 1101111100

0011	0111	1100
------	------	------

= 3	7	C
-----	---	---

= 37C

Number	Quotient on division by 16	Remainder
--------	----------------------------	-----------

892	55	12=C
55	3	7
3	0	3

So the hex number is : 37C

Check Your Progress 2

1.
 - i) 23 in BCD is 0010 0011
 - ii) 49.25 in BCD is 0100 1001.0010 0101
 - iii) 892 in BCD is 1000 1001 0010
2. 1's complement is obtained by complementing each bit while 2's complement is obtained by leaving the number unchanged till first 1 starting from least significant bit after that complement each bit.

	(i)	(ii)	(iii)
Number	10100010	00000000	11001100
1's complement	01011101	11111111	00110011
2's complement	01011110	00000000	00110100

3. We are using signed 2's complement notation

(i) +50 is 0 0110010
 +5 is 0 0000101
 therefore -5 is 1 1111011

Add+50 = 0 0110010

- 5 1 1111011

1 0 0101101
 ^ ↑

carry out (discard the carry)

Carry in to sign bit = 1

Carry out of sign bit = 1 Therefore, no overflow

The solution is 0010 1101= +45

ii) +45 is 0 0101101
 +65 is 0 1000001
 Therefore, -65 is 1 0111111
 +45 0 0101101
 - 65 1 0111111
 1 1101100

No carry into sign bit, no carry out of sign bit. Therefore, no overflow.

+20 is 0 0010100

Therefore, -20 is 1 1101100

which is the given sum

(iii)	+75	is	0	1001011
	+85	is	0	1010101
				<hr/>
			1	0100000

Carry into sign bit = 1

Carry out of sign bit = 0

Overflow.

Check Your Progress 3

1.

i) 1010.0001

$$= 1.0100001 \times 2^3$$

So, the single precision number is :

Significand = 010 0001 000 0000 0000 0000

Exponent = $3+127 = 130 = 10000010$

Sign=0

So the number is = 0 1000 0010 010 0001 0000 0000 0000 0000

ii) -0.0000111

$$-1.11 \times 2^{-5}$$

Significand = 110 0000 0000 0000 0000 0000

Exponent = $127-5 = 122 = 0111 1010$

Sign = - $\equiv 1$

So the number is

1 0111 1010 110 0000 0000 0000 0000 0000

2.	Data	Even parity bit	Odd parity bit
	0101010	1	0
	0000000	0	1
	1111111	1	0
	1000100	0	1

3. The equation for SEC code is

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

i — Number of bits in SEC code

N — Number of bits in data word

In, this case N = 16
 i = ?

so the equation is

$$2^i - 1 \geq 16 + i$$

at i = 4

$$2^4 - 1 \geq 16 + 4$$

15 \geq 20 Not true.

at i = 5

$$2^5 - 1 \geq 16 + 5$$

31 \geq 21 True the condition is satisfied.

Although, this condition will be true for $i > 5$ also but we want to use only minimum essential correction bits which are 5.

For SEC-DED code we require an additional bit as overall parity. Therefore, the SEC-DED code will be of 6 bits.

UNIT 3 PRINCIPLES OF LOGIC CIRCUITS I

Structure	Page Nos.
3.0 Introduction	60
3.1 Objectives	60
3.2 Logic Gates	60
3.3 Logic Circuits	62
3.4 Combinational Circuits	63
3.4.1 Canonical and Standard Forms	
3.4.2 Minimization of Gates	
3.5 Design of Combinational Circuits	72
3.6 Examples of Logic Combinational Circuits	73
3.6.1 Adders	
3.6.2 Decoders	
3.6.3 Multiplexer	
3.6.4 Encoder	
3.6.5 Programmable Logic Array	
3.6.6 Read Only Memory ROM	
3.7 Summary	82
3.8 Solutions/ Answers	82

3.0 INTRODUCTION

In the previous units, we have discussed the basic configuration of computer system von Neumann architecture, data representation and simple instruction execution paradigm. But ‘How does a computer actually perform computations?’. Now, we will attempt to find answer of this basic query. In this unit, you will be exposed to some of the basic components that form the most essential parts of a computer. You will come across terms like logic gates, binary adders, logic circuits and combinational circuits etc. These circuits are the backbone of any computer system and knowing them is quite essential. The characteristics of integrated digital circuits are also discussed in this unit.

3.1 OBJECTIVES

After going through this unit you will be able to:

- define logic gates;
 - describe the significance of Boolean algebra in digital circuit design;
 - describe the necessity of minimizing the number of gates in design;
 - describe how basic mathematical operations, viz. addition and subtraction, are performed by computer; and
 - define and describe some of the useful circuits of a computer system such as multiplexer, decoders, ROM etc.
-

3.2 LOGIC GATES

A logic gate is an electronic circuit which produces a typical output signal depending on its input signal. The output signal of a gate is a simple Boolean operation of its input signal. Gates are the basic logic elements that produce signals of binary 1 or 0.

We can represent any Boolean function in the form of gates.

In general we can represent each gate through a distinct graphic symbol and its operation can be given by means of algebraic expression. To represent the input-output relationship of binary variables in each gate, truth tables are used. The notations and truth -tables for different logic gates are given in Figure 3.1.

Figure 3.1: Logic Gates

The truth table of NAND and NOR can be made from NOT (A AND B) and NOT (A OR B) respectively. Exclusive OR (XOR) is a special gate whose output is one only if the two inputs are not equal. The inverse of exclusive OR, called as XNOR gate, can be a comparator which will produce a 1 output if two inputs are equal.

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

3.3 LOGIC CIRCUITS

A Boolean function can be implemented into a logic circuit using the basic gates:- AND , OR & NOT. Consider, for example, the Boolean function: -

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

The relationship between this function and its binary variables A, B, C can be represented in a truth table as shown in figure 3.2(a) and figure 3.2(b) shows the corresponding logic circuit.

Inputs			Output
A	B	C	F
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

(a) Truth Table

(b) Logic Circuit

Figure 3.2 : Truth table & logic diagram for $F = \overline{A} B + C$

Thus, in a logic circuit, the variables coming on the left hand side of boolean expression are inputs to circuit and the variable function coming on the right hand side of expression is taken as output.

Here, there is one important point to note i.e. there is only one way to represent the boolean expression in a truth table but can be expressed in variety of logic circuits. How? [try to find the answer]

Check Your Progress 1

- What are the logic gates and which gates are called as Universal gates.
.....
.....
- Simplify the Boolean function: $F = \overline{\left\{ \left(\overline{A + B} \right) + \left(\overline{A + B} \right) \right\}}$
.....
.....
.....
- Draw the logic diagram of the above function.
.....
.....
.....
.....

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

.....
.....

- 5) Show implementation of AND, NOT and OR Operations using NAND gates.

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

3.4 COMBINATIONAL CIRCUIT

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

The output of the combinational circuit is related to its input by a combinational function, which is independent of time. Therefore, for an ideal combinational circuit the output should change instantaneously according to changes in input. But in actual case there is a slight delay. The delay is normally proportional to *depth* or number of levels i.e. the maximum numbers of gates on any path from input to output. For example, the depth of the combinational circuit in figure 3.3 is 2.

Figure 3.3 : A two level AND-OR combinational circuit

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

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

3.4.1 Canonical and Standard Forms

An algebraic expression can exist in two forms :

- Sum of Products (SOP) e.g. $(A \cdot \bar{B}) + (\bar{A} \cdot \bar{B})$
- Product of Sums (POS) e.g. $(\bar{A} + \bar{B}) \cdot (A + B)$

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

$$F(A, B, C) = (A \cdot B \cdot C) + (\bar{A} \cdot \bar{B} \cdot C) + (A \cdot B)$$

We have three product terms namely $A \cdot B \cdot C$, $\bar{A} \cdot \bar{B} \cdot C$ and $A \cdot B$. But only first two of them qualifies to be a minterm, as the third one does not contain variable C or its

complement. In addition, the term $A.B.C$ will be one only if $A = 1$, $B = 1$ and $C = 1$, for any other combination of values of A , B , C the minterm $A.B.C$ will have 0 value. Similarly, the minterm $\bar{A} \bar{B} . C$ will have value 1 only if $\bar{A} = 1$, $\bar{B} = 1$ and $C = 1$ (It implies $A=0$, $B=0$ and $C=1$) for any other combination of values the minterm will have a zero value.

Similar type of term used in POS form is called **Maxterm or Standard Sums** . Maxterm is a term of POS expression, which contains all the variables of the function in true or complemented form. For example, $F(A, B, C) = (A + B + C). (\bar{A} + \bar{B} + C)$ has two maxterms. A maxterm has a value 0, for only one combination of input values.

The maxterm $A + B + C$ will has 0 value only for $A = 0$, $B = 0$ and $C = 0$ for all other combination of values of A , B , C it will have a value 1.

Figure 3.4 indicates the 2^n different minterms and maxterms where n is number of variables.

Variable's Value			Minterm		Maxterm	
a	b	c	Term	Representation	Term	Representation
0	0	0	$\bar{a} \bar{b} \bar{c}$	m_0	$a + b + c$	M_0
0	0	1	$\bar{a} \bar{b} c$	m_1	$a + b + \bar{c}$	M_1
0	1	0	$\bar{a} b \bar{c}$	m_2	$a + \bar{b} + c$	M_2
0	1	1	$\bar{a} b c$	m_3	$a + \bar{b} + \bar{c}$	M_3
1	0	0	$a \bar{b} \bar{c}$	m_4	$\bar{a} + b + c$	M_4
1	0	1	$a \bar{b} c$	m_5	$\bar{a} + b + \bar{c}$	M_5
1	1	0	$a b \bar{c}$	m_6	$\bar{a} + \bar{b} + c$	M_6
1	1	1	$a b c$	m_7	$\bar{a} + \bar{b} + \bar{c}$	M_7

Figure 3.4: Maxterms and Minterms for 3 variables

We can represent any Boolean function algebraically directly in minterm and maxterm form from the truth table. For *minterms*, consider each combination of variables that produces a 1 output in function and then taking OR of all those terms. For example, the function F in figure 3.5 is represented in minterm form by ORing the terms where the output F is 1 i.e. $\bar{a} \bar{b} \bar{c}$, $\bar{a} b \bar{c}$, $a \bar{b} c$ & $a b c$.

a	b	c	F	
0	0	0	0	m_0
0	0	1	1	m_1
0	1	0	1	m_2
0	1	1	1	m_3
1	0	0	0	m_4
1	0	1	0	m_5
1	1	0	1	m_6
1	1	1	1	m_7

Figure 3.5: Function of three variables

$$\begin{aligned}
 \text{Thus, } F(a,b,c) &= \bar{a} \bar{b} \bar{c} + \bar{a} b \bar{c} + \bar{a} b c + a \bar{b} c + a b c \\
 &= m_1 + m_2 + m_3 + m_6 + m_7 \\
 &= \sum (1,2,3,6,7)
 \end{aligned}$$

The complement of function F can be obtained by ORing of the minterms corresponding to the combinations that produce a 0 output in function. Thus,

$$\bar{F}(a, b, c) = \bar{a}\bar{b}\bar{c} + a\bar{b}\bar{c} + a\bar{b}c$$

If we take the complement of \bar{F} , we get the function F in maxterm form.

$$\begin{aligned} F(a, b, c) &= (\bar{F}) = \overline{(\bar{a}\bar{b}\bar{c} + a\bar{b}\bar{c} + a\bar{b}c)} = \overline{(\bar{a}\bar{b}\bar{c})} \cdot \overline{(a\bar{b}\bar{c})} \cdot \overline{(a\bar{b}c)} \\ &= (a + b + c)(\bar{a} + b + c)(\bar{a} + b + \bar{c}) \quad [\text{De Morgan's law}] \\ &= M_0 \cdot M_4 \cdot M_5 \\ &= \Pi(0, 4, 5) \end{aligned}$$

The product symbol Π stands for ANDing the maxterms.

Here, you will appreciate the fact that the terms which were missing in minterm form are present in maxterm form. Thus if any form is known then the other form can be directly formed.

The Boolean function expressed as a sum of minterms or product of maxterms has the property that each and every literal of the function should be present in each and every term in either normal or complemented form.

3.4.2 Minimization of Gates

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

- Algebraic Simplification
- Karnaugh Maps
- Quine McCluskey Method

Algebraic Simplification

We have already discussed algebraic simplification of logic circuit. An algebraic expression can exist in POS or SOP forms. Let us examine the following example to understand how it helps in implementing any logic circuit.

Example : Consider the function $F(a, b, c) = a\bar{b}\bar{c} + a\bar{b}c + a\bar{b}$. The logic circuit implementation of this function is shown in fig 3.6(a).

$$(a) F = a\bar{b}\bar{c} + a\bar{b}c + a\bar{b}$$

$$(b) F = \bar{a}\bar{b}\bar{c} + \bar{a}b\bar{c} + a\bar{b}b$$

Figure 3.6 : Two logic diagrams for same boolean expression

The expression F can be simplified using boolean algebra.

$$\begin{aligned} F(a,b,c) &= \bar{a}\bar{b}\bar{c} + \bar{a}b\bar{c} + a\bar{b}b \\ &= \bar{a}\bar{b}(\bar{c} + c) + a\bar{b}b \quad [\text{as } c + \bar{c} = 1] \\ &= \bar{a}\bar{b} + a\bar{b}b \\ &= a \oplus b \end{aligned}$$

The logic diagram of the simplified expression is drawn in fig 3.6 (b) using NOT, OR and AND gates (the same operation can be performed by using a single XOR gate). Thus the number of gates are reduced to 5 gates (2 inverters, 2 AND gates & 1 OR) instead of 7 gates. (3 inverters, 3 AND & 1 OR gate).

The algebraic function can appear in many different forms although a process of simplification exists yet it is cumbersome because of absence of routes which tell what rule to apply next. The Karnaugh map is a simple direct approach of simplification of logic expressions.

Karnaugh Maps

Karnaugh maps are a convenient way of representing and simplifying Boolean function of 2 to 6 variables. The stepwise procedure for Karnaugh map is.

Step 1: Create a simple map depending on the number of variables in the function. Figure 3.7(a) shows the map of two, three and four variables. A map of 2 variables contains 4 value position or elements, while for 3 variables it has $2^3 = 8$ elements. Similarly for 4 variables it is $2^4 = 16$ elements and so on. Special care is taken to represent variables in the map. The value of only one variable changes in two adjacent columns or rows. The advantage of having change in one variable is that two adjacent columns or rows represent a true or complement form of a single variable.

For example, in figure 3.7(a) the columns which have positive A are adjacent and so are the column for \bar{A} . Please note the adjacency of the corners. The right most column can be considered to be adjacent to the first column since they differ only by one variable and are adjacent. Similarly the top most and bottom most rows are adjacent.

(a) Maps for 2, 3 and 4 variables

(b) Possible adjacencies

Figure 3.7: Maps and their adjacencies

Please note:

- 1) Decimal equivalents of column are given for help in understanding where the position of the respective set lies. It is *not* the value filled in the square. A square can contain one or nothing.
- 2) The 00, 01, 11 etc written on the top implies the value of the respective variables.
- 3) Wherever the value of a variable is 0 it is said to represent its complement form.
- 4) The value of only one variable changes when we move from one row to the next row or one column to the next column.

Step 2: The next step in Karnaugh map is to map the truth table into the map. The mapping is done by putting a 1 in the respective square belonging to the 1 value in the truth table. This mapped map is used to arrive at simplified Boolean expression which then can be used for drawing up the optimal logical circuit. Step 2 will be more clear in the example.

Step 3: Now, create simple algebraic expression from the K-Map. These expressions are created by using adjacency if we have two adjacent 1's then the expression for those can be simplified together since they differ only in 1 variable. Similarly, we search for the adjacent pairs of 4, 8 and so on. A 1 can appear in more than one adjacent pairs. We should search for octets first then quadrets and then for doublets. The following example will clarify the step 3.

Example: Now, let us see how to use K map simplification for finding the Boolean function for the cases whose truth table is given in figure 3.8(a) and 3.8(B) shows the K-Map for this.

Decimal	A	B	C	D	Output F
0	0	0	0	0	1
1	0	0	0	1	1
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	1
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	0
12	1	1	0	0	0
13	1	1	0	1	0
14	1	1	1	0	0
15	1	1	1	1	0

$$\text{Or } F = \sum (0, 1, 2, 6, 8, 9, 10)$$

(a) Truth table

(b) Karnaugh's map

Figure 3.8 : Truth table & K-Map of Function $F = \sum (0, 1, 2, 6, 8, 9, 10)$

Let us see what the pairs which can be considered as adjacent in the Karnaugh's here.

The pairs are:

- 1) The four corners
- 2) The four 1's as in top and bottom in column 00 & 01
- 3) The two 1's in the top two rows of last column.

The corners can be represented by the expressions :

$$\begin{aligned}
 &1) \text{ Four corners} \\
 &= (\bar{A} \bar{B} \bar{C} \bar{D} + \bar{A} \bar{B} C \bar{D}) + (\bar{A} \bar{B} \bar{C} D + \bar{A} \bar{B} C D) \\
 &= \bar{A} \bar{B} \bar{D} (\bar{C} + C) + \bar{A} \bar{B} D (\bar{C} + C) \quad [\text{as } C + \bar{C} = 1] \\
 &= \bar{A} \bar{B} \bar{D} + \bar{A} \bar{B} D \\
 &0 \quad = \bar{B} \bar{D} (\bar{A} + A) \\
 &= \bar{B} \bar{D}
 \end{aligned}$$

2) The four 1's in column 00 and 01 gives the following terms

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

3) The two 1's in the last columns

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

Thus, the Boolean expression derived from this K-Map is

$$F = \bar{B} \bar{D} + \bar{B} \bar{C} + \bar{A} C \bar{D}$$

[Note : This expression can be directly obtained from the K-Map after making quadrets and doublets. Try to find how ?]

The expressions so obtained through K-Maps are in the forms of the sum of the product form i.e. it is expressed as the sum of the products of the variables. This expression can be expressed in product of sum form, but for this special method are required to be used [already discussed in last section].

Let us see how we can modify K-Map simplification to obtain POS form. Suppose in the previous example instead of using 1 we combined the adjacent 0 squares then we will obtain the inverse function and on taking transform of this function we will get the POS form.

Another important aspect about this simple method of digital circuit design is DONOT care conditions. These conditions further simplify the algebraic function. These conditions imply that it does not matter whether the output produced is 0 or 1 for the specific input. These conditions can occur when the combination of the number of inputs are more than needed. For example, calculation through BCD where 4 bits are used to represent a decimal digit implies we can represent $2^4 = 16$ digits but since we have only 10 decimal digits therefore 6 of those input combination values do not matter and are a candidate for DONOT care condition.

For the purpose of exercises you can do the exercise from the reference [1], [2], [3] given in Block introduction.

What will happen if we have more than 4– 6 variables? As the numbers of variables increases K-Maps become more and more cumbersome as the numbers of possible combinations of inputs keep on increasing.

Quine McCluskey Method

A tabular method was suggested to deal with the increasing number of variables known as Quine McCluskey Method. This method is suitable for programming and hence provides a tool for automating design in the form of minimizing Boolean expression.

The basic principle behind the Quine McCluskey Method is to remove the terms, which are redundant and can be obtained by other terms.

To understand Quine - Mc Kluskey method, lets us see following example:-

$$\begin{aligned}
 \text{Given, } F(A,B,C,D,E) &= ABCDE + ABC \bar{D} E + A \bar{B} \bar{C} DE + \bar{A} BCD \bar{E} + \\
 &\quad \bar{A} \bar{B} CDE + A \bar{B} \bar{C} DE + A \bar{B} \bar{C} \bar{D} E + \bar{A} \bar{B} \bar{C} \bar{D} E
 \end{aligned}$$

Step I: The terms of the function are placed in table as follows:

Term/var	A	B	C	D	E	Checked/Unchecked
ABCDE	1	1	1	1	1	✓
ABC \bar{D} E	1	1	1	0	1	✓
A \bar{B} \bar{C} DE	1	0	0	1	1	✓
\bar{A} BCD \bar{E}	0	1	1	1	0	✓
\bar{A} \bar{B} CD \bar{E}	1	0	1	1	0	✓
\bar{A} \bar{B} \bar{C} DE	0	0	0	1	1	✓
A \bar{B} \bar{C} \bar{D} E	1	0	0	0	1	✓
\bar{A} \bar{B} \bar{C} \bar{D} \bar{E}	0	0	0	0	0	✓

Step II: Forming the pairs which differ in only one variable, also put check (v) against the terms selected and finding resultant terms as follows :-

$$\left. \begin{array}{l} ABCDE \\ ABC\bar{D}E \end{array} \right\} \longrightarrow \boxed{ABCE}$$

$$\left. \begin{array}{l} \bar{A}\bar{B}\bar{C}DE \\ \bar{A}\bar{B}CDE \end{array} \right\} \longrightarrow \boxed{\bar{B}\bar{C}DE} \quad \checkmark$$

$$\left. \begin{array}{l} \bar{A}BCD\bar{E} \\ \bar{A}\bar{B}CD\bar{E} \end{array} \right\} \longrightarrow \boxed{\bar{A}CD\bar{E}}$$

$$\left. \begin{array}{l} A\bar{B}\bar{C}\bar{D}E \\ A\bar{B}\bar{C}DE \end{array} \right\} \longrightarrow \boxed{\bar{B}\bar{C}\bar{D}E} \quad \checkmark$$

In the new terms, again find all the terms which differ only in one variable and put a check (x) across those terms i.e.

$$\left. \begin{array}{l} \bar{B}\bar{C}DE \\ \bar{B}\bar{C}\bar{D}E \end{array} \right\} \longrightarrow \boxed{\bar{B}\bar{C}E}$$

Step III: Now, constructing final table as :

	ABCDE	ABC \bar{D} E	\bar{A} \bar{B} \bar{C} DE	A \bar{B} \bar{C} D \bar{E}	\bar{A} BCD \bar{E}	\bar{A} \bar{B} CD \bar{E}	\bar{A} \bar{B} \bar{C} \bar{D} E	\bar{A} \bar{B} \bar{C} \bar{D} \bar{E}
ABCE	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>						
$\bar{A}CD\bar{E}$					<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>		
$\bar{B}\bar{C}E$			<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>			<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>

Thus all columns have mark 'X'. Thus the **final expression** is:

$$F(A,B,C,D,E) = ABCE + \bar{A}CD\bar{E} + \bar{B}\bar{C}E$$

The process can be summarised as follows:-

Step I : Build a table in which each term of the expression is represented in row (Expression should be in SOP form). The terms can be represented in the 0 (Complemented) or 1 (normal) form.

Step II : Check all the terms that differ in only one variable and then combine the pairs by removing the variable that differs in those terms. Thus a new table is formed.

This process is repeated, if necessary, in the new table also until all uncommon terms are left i.e. no matches left in table.

Step III :

- a) Finally, a two dimensional table is formed all terms which are not eliminated in the table form rows and all original terms form the column.
- b) At each intersection of row and column where row term is subset of column term, a 'X' is placed.

Step IV :

- a) Put a square around each 'X' which is alone in column
- b) Put a circle around each 'X' in any row which contains a squared 'X'
- c) If every column has a squared or circled 'X' then the process is complete and the corresponding minimal expression is formed by all row terms which have marked Xs.

Check Your Progress 2

1) Prepare the truth table for the following boolean expressions:

(i) $A \bar{B} \bar{C} + \bar{A} B \bar{C}$

(ii) $(A+B) \cdot (\bar{A} + \bar{B})$

2 Simplify the following functions using algebraic simplification procedures and draw the logic diagram for the simplified function.

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

(ii) $F = (\overline{(A \cdot B)}) \cdot (\bar{A} \bar{B})$

.....

3) Simplify the following boolean functions in SOP and POS forms by means of K-Maps.

Also draw the logic diagram.

$F(A,B,C,D) = \Sigma (0,2,8,9,10,11,14,15)$

.....

3.5 DESIGN OF COMBINATIONAL CIRCUITS

The digital circuits, which we use now-a-days, are constructed with NAND or NOR gates instead of AND–OR–NOT gates. NAND & NOR gates are called *Universal Gates* as we can implement any digital system with these gates. To prove this point we need to only show that the basic gates : AND , OR & NOT, can be implemented with either only NAND or with only NOR gate. This is shown in figure 3.9 below:

Figure 3.9 : Basic Logic Operations with NAND and NOR gates

Any Boolean expression can be implemented with NAND gates, by expressing the function in sum of product form.

Example: Consider the function $F(A, B, C) = \sum (1, 2, 3, 4, 5, 7)$. Firstly bring it in SOP form. Thus, from the K-Map shown in figure 3.10(a), we find

$$F(A, B, C) = C + \bar{A}B + A\bar{B} = \left(\overline{\overline{C + \bar{A}B + A\bar{B}}} \right)$$

$$= \left(\overline{\bar{C} \cdot (\bar{A}B) \cdot (A\bar{B})} \right)$$

Figure 3.10: K-Map & Logic circuit for function $F(A, B, C) = \sum (1, 2, 3, 4, 5, 7)$.

Similarly, any Boolean expression can be implemented with only NOR gate by expressing in POS form. Let us take same example, $F(A, B, C) = \sum (1, 2, 3, 4, 5, 7)$.

As discussed in section 3.4.1, the above function F can be represented in POS form as

$$\begin{aligned} F(A, B, C) &= \prod (0, 6) \\ &= (\overline{A} + \overline{B} + \overline{C})(A + B + \overline{C}) = \overline{(\overline{A} + \overline{B} + \overline{C})}(\overline{A + B + \overline{C}}) \\ &= \overline{(\overline{A} + \overline{B} + \overline{C})} + (A + B + \overline{C}) \end{aligned}$$

Figure 3.11: Logic circuit for function $F(A, B, C) = \sum (1, 2, 3, 4, 5, 7)$ using NOR gates

After discussing so much about the design let us discuss some important combinational circuits. We will not go into the details of their design in this unit.

3.6 EXAMPLES OF COMBINATIONAL CIRCUITS

The design of combinational circuits can be demonstrated with some basic combinational circuits like adders, decoders, multiplexers etc. Let us discuss each of these examples briefly.

3.6.1 Adders

Adders play one of the most important roles in binary arithmetic. In fact fixed point addition is often used as a simple measure to express processor's speed. Addition and subtraction circuit can be used as the basis for implementation of multiplication and division. (we are not giving details of these, you can find it in Suggested Reading).

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

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

In half adder the inputs are:

The augend lets say 'x' and addend 'y' bits.

The outputs are sum 'S' and carry 'C' bits.

The logical relationship between these are given by the truth table as shown in figure 3.12 (a). Carry 'C' can be obtained on applying AND gate on 'x' & 'y' inputs, therefore , $C = x.y$, while S can be found from the Karnaugh Map as shown in figure 3.12(b). The corresponding logic diagram is shown in figure 3.12(c).

Thus, the sum and carry equations of half- adder are:

$$S = x.\bar{y} + \bar{x}.y$$

$$C = x.y$$

(a) Truth table

(b) K- Map for 'S'

(c) Logic Diagram

Figure 3.12: Half – Adder implementation

Let us take the full adder. For this another variable carry from previous bit addition is added let us call it 'p'. The truth table and K-Map for this is shown in figure 3.13.

K=Maps for 'S'

Truth table

K – Maps for 'C'

(d) Logic Diagram

(e) Block Diagram

Figure 3.13 : Full-adder implementation

Three adjacencies marked a,b,c in K-Map of 'C' are

$$\begin{aligned} \text{a) } & x \bar{y} p + x y p \\ &= x p (y + \bar{y}) \\ &= x p \end{aligned}$$

$$\begin{aligned} \text{b) } & x y p + x y \bar{p} \\ &= x y \end{aligned}$$

$$\begin{aligned} \text{c) } & \bar{x} y p + x y p \\ &= y p \end{aligned}$$

Thus, $C = x p + x y + y p$

In case of K-Map for 'S', there are no adjacencies. Therefore,

$$S = \bar{x} \bar{y} \bar{p} + \bar{x} y \bar{p} + x \bar{y} \bar{p} + x y p$$

Till now we have discussed about addition of bit only but what will happen if we are actually adding two numbers. A number in computer can be 4 byte i.e. 32 bit long or even more. Even for these cases the basic unit is the full adder. Let us see (for example) how can we construct an adder which adds two 4 bit numbers. Let us assume that the numbers are: $x_3 x_2 x_1 x_0$ and $y_3 y_2 y_1 y_0$; here x_i and y_i ($i = 0$ to 3) represent a bit. The 4-bit adder is shown in figure 3.14.

Figure 3.14 : 4-bit Adder

The overall sum is represented by $S_3 S_2 S_1 S_0$ and over all carry is C_3 from the 4th bit adder. The main feature of this adder is that carry of each lower bit is fed to the next higher bit addition stage, it implies that addition of the next higher bit has to wait for the previous stage addition. This is called ripple carry adder. The ripple carry becomes time consuming when we are going for addition of say 32 bit. Here the most significant bit i.e. the 32nd bits has to wait till the addition of first 31 bits is complete.

Therefore, a high-speed adder, which generates input carry bit of any stage directly from the input to previous stages was developed. These are called carry lookahead adders. In this adder the carry for various stages can be generated directly by the logic expressions such as:

$$C_0 = x_0 y_0$$
$$C_1 = x_1 y_1 + (x_1 + y_1) C_0$$

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

Adder- subtractor

The subtraction operation on binary numbers can be achieved by sequence of addition operations only i.e. to perform subtraction, $A-B$, we can find 2's complement of B . This can be calculated using 1's complement & then adding 1 to it. Thus, a common circuit can perform the addition and subtraction operation. A 4-bit adder- subtraction circuit is shown in figure 3.15, which is formed by using XOR gate with every full adder. The XOR gate with output 0 is for detecting overflow.

Figure 3.15: 4-bit adder-subtractor circuit

The control input 'x' controls the operations i.e. if $x=0$ then the circuit behaves like an adder and if $x=1$ then circuit behaves like a subtractor. The operation is summarized as :

- a) When $x = 0$, $c = 0$, the output of all XOR gates will be the same as the corresponding input B_i where $i = 0$ to 3. Thus, A_i & B_i are added through full adders giving Sum, S_i & carry C_i

- b) When $x = 1$, the output of all XOR gates will be complement of input B_i where $i = 0$ to 3 , to which carry $C_0 = 1$ is added. Thus, the circuit finds A plus 2 's complement of B , that is equal to $A - B$.

3.6.2 Decoders

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

(a) Block Diagram

(b) Logic Diagram

(c) Truth Table

Figure 3.16 : 3×8 decoder

Please make sure while constructing the logic diagram wherever the values in the truth table are appearing as zero in input and one in output the input should be fed in complemented form e.g. the first 4 entries of truth table contains 0 in I_0 position and hence I_0 value 0 is passed through a NOT gate and fed to AND gates 'a', 'b', 'c' and 'd' which implies that these gates will be activated/selected only if I_0 is 0. If I_0 value is 1 then none of the top 4 AND gates can be activated. Similar type of logic is valid for I_1 . Please note the output line selected is named 000 or 010 or 111 etc. The output value of only one of the lines will be 1. These 000, 010 indicates the label and suggest that if you have these $I_0 I_1 I_2$ input values the labeled line will be selected for the output. The enable line is a good resource for combining two 3×8 decoders to make one 4×16 decoder.

3.6.3 Multiplexer

Multiplexer is one of the basic building units of a computer system which in principle allows sharing of a common line by more than one input lines. It connects multiple input lines to a single output line. At a specific time one of the input lines is selected and the selected input is passed on to the output line. The diagram 4×1 multiplexer (MUX) is given in figure 3.16.

(a) Block diagram

(c) Truth table

(c) Logic diagram

Figure 3.17: 4×1 Multiplexer

But how does the multiplexer know which line to select? This is controlled by the select lines. The select lines provide the communication among the various components of a computer. Now let us see how the multiplexer also known as MUX works, here for simplicity we will take the example of 4×1 MUX i.e. there are 4 input lines connected to 1 output line. For the sake of consistency we will call input line as I, and output line as O and control line a selection line S or enable as E.

Please notice the way in which S_0 and S_1 are connected in the circuit. To the 'a' AND gate S_0 and S_1 are inputted in complement form that means 'a' gate will output I_0 when both the selection lines have a value 0 which implies $\overline{S_0} = 1$ and $\overline{S_1} = 1$, i.e. $S_0 = 0$ and $S_1 = 0$ and hence the first entry in the truth table. Please note that at $S_0 = 0$ and $S_1 = 0$, AND gate 'b', 'c', 'd' will yield 0 output and when all these outputs will pass OR gate 'e' they will yield I_0 as the output for this case. That is for $S_0 = 0$ and $S_1 = 0$ the output becomes I_0 , which in other words can be said as "For $S_0 = 0$ and $S_1 = 0$, I_0 input line is selected by MUX". Similarly other entries in the truth table are corresponding to the logical nature of the diagram. Therefore, by having two control lines we could have a 4×1 MUX. To have 8×1 MUX we must have 3 control lines or with 3 control lines we could make $2^3 = 8$ i.e. 8×1 MUX. Similarly, with 'n' control lines we can have

$2^n \times 1$ MUX. Another parameter which is predominant in MUX design is a number of inputs to AND gate. These inputs are determined by the voltage of the gate, which normally support a maximum of 8 inputs to a gate.

Where can these devices be used in the computer? The multiplexers are used in digital circuits for data and controlled signal routing.

We have seen a concept where out of 'n' input lines, 1 can be selected, can we have a reverse concept i.e. if we have one input line and data is transmitted to one of the possible 2^n lines where 'n' represents the number of selection lines. This operation is called *Demultiplexing*.

3.6.4 Encoders

An Encoder performs the reverse function of the decoder. An encoder has 2^n input lines and 'n' output line. Let us see the 8×3 encoder which encodes 8 bit information and produces 3 outputs corresponding to binary numbers. This type of encoder is also called octal-to-binary encoder. The truth table of encoder is shown in figure 3.17.

(a) Block diagram

I_0	I_1	I_2	I_3	I_4	I_5	I_6	I_7		O_2	O_1	O_0
1	0	0	0	0	0	0	0	D_0	0	0	0
0	1	0	0	0	0	0	0	D_1	0	0	1
0	0	1	0	0	0	0	0	D_2	0	1	0
0	0	0	1	0	0	0	0	D_3	0	1	1
0	0	0	0	1	0	0	0	D_4	1	0	0
0	0	0	0	0	1	0	0	D_5	1	0	1
0	0	0	0	0	0	1	0	D_6	1	1	0
0	0	0	0	0	0	0	1	D_7	1	1	1

(b) Truth Table

Figure 3.18 : Encoder

From the encoder table, it is evident that at any given time only one input is assumed to have 1 value. This is a major limitation of encoder. What will happen when two inputs are together active? The obvious answer is that since the output is not defined the ambiguity exists. To avoid this ambiguity the encoder circuit has input priority so that only one input is encoded. The input with high subscript can be given higher priority. For example, if both D_2 and D_6 are 1 at the same time, then the output will be 110 because D_6 has higher priority than D_2 .

The encoder can be implemented with 3 OR gates whose inputs can be determined from the truth table. The output can be expressed as:

$$O_0 = I_1 + I_3 + I_5 + I_7$$

$$O_1 = I_2 + I_3 + I_6 + I_7$$

$$O_2 = I_4 + I_5 + I_6 + I_7$$

You can draw the K-Maps to determine above functions and draw the related combinational circuit

3.6.5 Programmable Logic Array

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

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

Figure 319: Programmable Logic Array

The figure 3.18(a) shows a PLA of 3 inputs and 2 outputs. Please note the connectivity points, all these points can be connected if desired. Figure 3.18(b) shows an implementation of logic function:

$$O_0 = I_0 \cdot I_1 \cdot I_2 + \bar{I}_0 \cdot \bar{I}_1 \cdot \bar{I}_2 \text{ and } O_1 = \bar{I}_0 \cdot \bar{I}_1 \cdot \bar{I}_2 + \bar{I}_0 \cdot \bar{I}_1$$

3.6.6 Read-only-Memory (ROM)

The read-only-memory is an example of a Programmable Logic Device (PLD) i.e the binary information that is stored within a PLD is specified in some fashion and embedded within the hardware. Thus the information remains even when the power goes.

(a) Block Diagram

b) Logic Diagram of 64-bit ROM

Figure 3.20: ROM Design

Figure 3.19 shows the block diagram of ROM. It consists of 'k' input address lines and 'n' output data lines. An $m \times n$ ROM is an array of binary cell organised into m ($2^k = m$) words of 'n' bits each. The ROM does not have any data input because the write operation is not defined for ROM. ROM is classified as a combinational circuit and constructed internally with decoder and a set of OR gates.

In general, a $m \times n$ ROM (where $m = 2^k$, k = no. of address lines) will have an internal $k \times 2^k$ decoder and 'n' OR gate. Each OR gates has 2^k inputs which are connected to each of the outputs of the decoder.

Check Your Progress 3

- 1) Draw a Karnaugh Map for 5 variables.
.....
.....
.....
- 2) Map the function having 4 variables in a K- Map and draw the truth table. The function is
 $F(A, B, C, D) = (2, 6, 10, 14)$.
.....
.....
.....
- 3) Find the optimal logic expression for the above function. Draw the resultant logic diagram.
.....
.....
.....
- 4) What are the advantages of PLA?
.....
.....
.....
- 5) Can a full adder be constructed using 2 half adders?
.....
.....
.....

3.7 SUMMARY

This unit provides you the information regarding a basis of a computer system. The key elements for the design of a combinational circuit like adders etc. are discussed in this unit. With the advent of PLA's the designing of circuit is changing and now the scenario is moving towards micro processors. With this developing scenario in the forefront and the expectation of Ultra- Large- Integration (ULSI) in view, time is not far of when design of logic circuits will be confined to single microchip components. You can refer to latest trends of design and development including VHDL (a hardware design language) in the further readings.

3.8 SOLUTIONS/ANSWERS

Check Your Progress 1

1. Logic gates produce typical outputs based on input values NAND and NOR are universal gates as they can be used to construct any other logic gate.

2.

$$\begin{aligned}
 F &= \overline{\left\{ \left(\overline{\overline{A} + \overline{B}} \right) + \left(\overline{A + \overline{B}} \right) \right\}} \\
 &= \overline{\overline{\overline{A} + \overline{B}}} \cdot \overline{\overline{A + \overline{B}}} \\
 &= (\overline{A} + \overline{B}) \cdot (A + \overline{B}) \\
 &= (\overline{A} + \overline{B}) \cdot A + (\overline{A} + \overline{B}) \cdot \overline{B} \\
 &= \overline{A} \cdot A + A \cdot \overline{B} + \overline{A} \cdot \overline{B} + \overline{B} \cdot \overline{B} \\
 &= 0 + A \cdot \overline{B} + \overline{A} \cdot \overline{B} + \overline{B} \\
 &= 0 + \overline{B}(A + \overline{A}) + \overline{B} \\
 &= 0 + \overline{B} + \overline{B} = \overline{B}
 \end{aligned}$$

3.

4.

5.

Check Your Progress 2

1 (i):

A	B	C	F = (A $\overline{B}\overline{C}$ + $\overline{A}B\overline{C}$)
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0

**Introduction to Digital
Circuits**

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

(ii)

A	B	$F = (A+B) \cdot (\bar{A} + \bar{B})$
0	0	0
0	1	1
1	0	1
1	1	0

2 (i)

$$\begin{aligned}
 F &= ((\bar{A} \cdot \bar{B}) + B) \\
 &= \bar{A} + \bar{B} + B \\
 &= \bar{A} + 1 \quad (B + \bar{B} \text{ is always } 1) \\
 &= 1
 \end{aligned}$$

(ii)

$$\begin{aligned}
 F &= (\bar{A} \cdot \bar{B}) \cdot (\bar{A} + \bar{B}) \\
 &= (\bar{A} + \bar{B}) \cdot (\bar{A} \cdot \bar{B}) \\
 &= \bar{A} \bar{A} \bar{B} + \bar{A} \bar{B} \bar{B} \\
 &= \bar{A} \bar{B} + \bar{A} \bar{B} \\
 &= \bar{A} \bar{B}
 \end{aligned}$$

3

$$\begin{aligned}
 \bar{F} &= \bar{A}B + B\bar{C} + \bar{A}D \\
 F &= \overline{(\bar{A}B + B\bar{C} + \bar{A}D)} \\
 F &= (\overline{\bar{A}B}) \cdot (\overline{B\bar{C}}) \cdot (\overline{\bar{A}D}) \\
 F &= (A + \bar{B}) \cdot (\bar{B} + C) \cdot (A + \bar{D})
 \end{aligned}$$

Check Your Progress 3 :

1

2 **K-Map**

Truth table

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

**Introduction to Digital
Circuits**

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

3. One adjacency of 4 variables, So
 $F = C.\overline{D}$
4. PLA's are generic chips that can be used to implement a number of SOP logic function
- 5.

UNIT 4 PRINCIPLE OF LOGIC CIRCUITS II

Structure	Page Nos.
4.0 Introduction	87
4.1 Objectives	87
4.2 Sequential Circuits: The Definition	87
4.3 Flip Flops	88
4.3.1 Basic Flip-Flops	
4.3.2 Excitation Tables	
4.3.3 Master Slave Flip Flops	
4.3.4 Edge Triggered Flip-flops	
4.4 Sequential Circuit Design	95
4.5 Examples of Sequential Circuits	98
4.5.1 Registers	
4.5.2 Counters – Asynchronous Counters	
4.5.3 Synchronous Counters	
4.5.4 RAM	
4.6 Design of a Sample Counter	103
4.7 Summary	105
4.8 Solutions/ Answers	105

4.0 INTRODUCTION

By now you are aware of the basic configuration of computer systems, how the data is represented in computer systems, logic gates and combinational circuits. In this unit you will learn how all the computations are performed inside the system. You will come across terms like flip flops, registers, counters, sequential circuits etc. Here, you will also learn how to make circuits using combinational and sequential circuits. These circuit design will help you in performing practicals in MCSL-017 lab course.

4.1 OBJECTIVES

After going through this unit you will be able to:

- define the flip-flops and latch;
- describe behaviour of various flip-flops;
- define significance of excitation tables and state diagrams;
- define some of the useful circuits of a computer system like registers counters etc.; and
- construct logic circuits involving sequential and combinational circuits.

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 time. The block diagram of a sequential circuit is shown in figure 4.1.

As shown in the diagram, the present output depends upon the past Input states.

Figure 4.1: Block Diagram of sequential circuits.

These sequential circuits unlike combinational circuits are time dependent. The sequential circuits are broadly classified, depending upon the time at which these are observed and their internal state changes. The two broad classifications of sequential circuits are:

- Synchronous
- Asynchronous

Synchronous circuits use flip-flops and their status can change only at discrete intervals (Doesn't it seem as good choice for discrete digital devices such as computers?). Asynchronous sequential circuits may be regarded as combinational circuit with feedback path. Since the propagation delays of output to input are small, they may tend to become unstable at times. Thus, complex asynchronous circuits are difficult to design.

The synchronization in a sequential circuit is achieved by a clock pulse generator, which gives continuous clock pulse. 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: - 0 or 1; disabled or active state. The storage elements can change their state only when a clock pulse occurs. Sequential circuits that have clock pulses as input to flip-flops are called clocked sequential circuit.

4.3 FLIP-FLOPS

Let us see flip-flops in detail. A flip-flop is a binary cell, which stores 1-bit of information. It itself is a sequential circuit. By now we know that flip-flop can change its state when clock pulse occurs but when? Generally, a flip-flop can change its state when the clock transitions from 0 to 1 (rising edge) or from 1 to 0 (falling edge) and not when clock is 1. If the storage element changes its state when clock is exactly at 1 then it is called *latch*. In simple words, **flip-flop** is *edge-triggered* and latch is *level-triggered*.

4.3.1 Basic Flip-flops

Let us first see a basic **latch**. A latch or a flip-flop can be constructed using two NOR or NAND gates. Figure 4.3 (a) shows logic diagram for S-R latch using NOR gates. The latch has two inputs S & R for set and reset respectively. When the output is $Q=1$ & $\bar{Q}=0$, the latch is said to be in the set state. When $Q=0$ & $\bar{Q}=1$, it is the reset state. Normally, The outputs Q & \bar{Q} are complement of each other. When both inputs are equal to 1 at the same time, an undefined state results, as both outputs are equal to 0.

(a) Logic Diagram

(b) Truth Table

Figure. 4.3: SR Latch using NOR gates

Figure 4.3 (b) Shows truth table for S-R latch. Let us examine the latch more closely.

- i) Say, initially 1 is applied to S leaving R to 0 at this time. As soon as $S=1$, the output of NOR gate 'b' goes to 0 i.e. \bar{Q} becomes 0 and almost immediately Q becomes 1 as both the inputs (\bar{Q} & R) to NOR gate 'a' become 0. The change in the value of S back to 0 does not change \bar{Q} as the input to NOR gate 'b' now are $\bar{Q} = 1$ & $S=0$. Thus, the flip-flop stays in set state even after S returns to 0.
- ii) If R goes to 1 then latch acquires clear state. On changing R to 1, Q changes to 0 irrespective of the state of flip-flop and as Q is 0 & S is 0 then \bar{Q} becomes 1. Even after R returns to 0, Q remains 0 i.e. latch is in clear state.
- iii) When both S & R go to 1 simultaneously, the two outputs go to 0. This gives undefined state.

Let us try to construct most common flip- flops from this basic latch.

R-S Flip flop - The graphic symbol of S-R flip-flop is shown in Fig 4.4. It has three inputs, S (set), R (reset) and C (for clock). The $Q(t+1)$ is the next state of flip-flop after the occurrence of a clock pulse. $Q(t)$ is the present state, that is present Q value (Set-1 or Reset- 0).

(a) Graphic Symbol

(b) Logic Diagram

(c) Characteristic Table

Figure 4.4: R-S Flip-flop

In figure 4.4 (a), the arrowhead symbol in front of clock pulse C indicates that the flip-flop responds to leading edge (from 0 to 1) of input clock signal.

Operation of R-S flip-flop can be summarised as:

- 1) If no clock signal i.e. $C=0$ then output can not change irrespective of R & S values
- 2) When clock signal changes from 0 to 1 and $S=1$, $R=0$ then output $Q=1$ & $\bar{Q}=0$ (Set)
- 3) If $R=1$ $S=0$ & clock signal C changes from 0 to 1 then output $Q=0$ & $\bar{Q}=1$ (Reset)
- 4) During positive clock transition if both S & R become 1 then output is not defined, as it may become 0 or 1 depending upon internal timing delays occurring in circuit.

D Flip -Flop

The D (data) flip-flop is modification of RS flip-flop. The problem of undefined output in SR flip-flop when both R & S become 1 gets avoided in D flip-flop. The simple solution to avoid such condition is by providing just a single input. Thus, the non-clocked inputs to AND gates (S & R of fig 4.4 (b)) are guaranteed to be opposite of each other by inserting an **inverter** between them. The logic diagram and characteristic table of D flip flop is shown in figure 4.5.

(a) Graphic Symbol

(b) Logic Diagram

(c) Characteristic Table

Figure 4.5: D Flip flop

D flip-flop is also referred as Delay flip-flop because it delays the 0 or 1 applied to its input by a single clock pulse.

J-K flip-flop

The J-K flip-flop is also a modification of SR flip-flop, it has 2 inputs like S & R and all possible inputs combinations are valid in J K flip-flop.

Figure. 4.6 shows implementation of J K flip-flop. The inputs J & K behave exactly like input S & R to set and reset flip-flop, respectively. When J & K are 1, the flip-flop output is complemented with clock transition. [Try this as an exercise]

(a) Graphic Symbol (b) Logic Diagram (c) Characteristic Table

Figure 4.6: J – K Flip flop

T flip-flop

T (Toggle) flip-flop is obtained from JK flip-flop by joining inputs J & K together. The implementation of T flip-flop is shown in figure. 4.7. When T=0, the clock pulse transition does not change the state. When T=1, the clock pulse transition complement the state of the flip-flop.

(a) Graphic Symbol

b) Logic Diagram

(c) Characteristic Table

Figure 4.7: T- Flip flop

4.3.2 Excitation Tables

The characteristic tables of flip-flops provide the next state when inputs and the present state are known. These tables are useful for analysis of sequential circuits. But, during the design process, we know the required transition from present state to next state and wish to find the required flip-flop inputs. Thus comes the need of a table that lists the required input values for given change of state. Such a table is called excitation Table. Fig 4.8 shows excitation tables for all flip-flops.

Q(t)	Q(t+1)	J	K
0	0	0	X
0	1	1	X
1	0	X	1
1	1	X	0

(a) J-K Flip flop

Q(t)	Q(t+1)	S	R
0	0	0	X
0	1	1	0
1	0	0	1
1	1	X	0

(b) S-R 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	1

(d) T Flip flop

Figure 4.8: Excitation Tables for flip-flops

Q(t) & Q(t+1) indicates present and next state for flip a flop, respectively. The symbol X in the table means don't care condition i.e. doesn't matter whether input is 0 or 1.

Let us discuss more deeply, how these excitation tables are formed. For this, we take an example of J-K Flip flop.

- 1) The state transition from present state 0 to next state 0 (Figure 4.8 (a)) can be achieved when

(a) J=0, K=0, then no change in the state of flip flop

(b) J=0, K=1, then flip flop resets i.e. 0

(remember J-K Characteristic table from figure 4.6)

Thus in either case J=0 but K can be 0 or 1 that is represented by don't care condition X.

- 2) The state transition from present state 0 to next state 1 can be achieved when

(a) J=1, K=0, then flip flop is set i.e. 1

(b) J=1, K=1, then flip flop is complemented i.e. change from 0 to 1

Here, also in either case J=1 but K can be 0 or 1 that means again K is represented as a don't care case.

- 3) Similarly, state transition from present state 1 to next state 0 can be achieved when

(a) J=0, K=1, flip flop is reset i.e. 0

(b) J=1, K=1, flip flop is complemented i.e. changes from 1 to 0

This indicates that in either case $K=1$ but J can be either 0 or 1 thus don't care case.

- 4) For state transition from present state 1 to next state 1 can be achieved when
- (a) $J=0, K=0$, no change in flip flop
 - (b) $J=1, K=0$, flip flop is set i.e 1

Thus J is don't care case but $K=0$.

This whole process can be summarized in the table below:

Present State	Next State	Can be achieved
0	0	a) $J=0, K=0$, since $Q(t)=0$ b) $J=0, K=1$, flip flop resets
0	1	a) $J=1, K=0$, flip flop set b) $J=1, K=1$, flip flop complements, $Q(t)=0=Q(t+1)=1$
1	0	a) $J=0, K=1$, flip flop reset b) $J=1, K=1$, complement $Q(t)=\overline{Q(t)}$
1	1	a) $J=0, K=0$, no change b) $J=1, K=0$, flip – flop set

Similarly, the excitation tables for the rest of the flip-flops can be derived (Try to do this as an exercise).

Check Your Progress 1

1. What are sequential circuits?

.....

.....

.....

.....

2. What is flip- flop? How is different from latch?

.....

.....

.....

.....

3. What is the difference between excitation table and characteristic table?

.....

.....

.....

4.3.3 Master-Slave Flip-Flop

The master slave flip-flop consists of two flip-flops. One is the master flip-flop & other is called the slave flip-flop. Fig 4.9 shows implementation of master-slave flip-flop using J-K flip-flop.

Figure 4.9: Master – Slave flip- flop

Note: Master-slave flip-flop can be constructed using D or SR flip-flop in the same manner.

Now, let us summarize the working of this flip-flop:

- (i) When the clock pulse is 0, the master flip-flop is disabled but the slave becomes active and its output Q & \bar{Q} becomes equal to Y and \bar{Y} respectively. Why? Well the possible combination of the value of Y and Y' are either $Y=1, \bar{Y}=0$ or $Y=0, \bar{Y}=1$. Thus, the slave flip-flop can have following combinations: -
 - (a) $J=1, K=0$ which means $Q=1, \bar{Q}=0$ (set flip-flop)
 - (b) $J=0, K=1$ which means $Q=0, \bar{Q}=1$ (clear flip-flop)
- (ii) When inputs are applied at JK and clock pulse becomes 1, only master gets activated resulting in intermediate output Y going to state 0 or 1 depending on the input and previous state. Remember that during this time slave is also maintaining its previous state only. As the clock pulse becomes 0, the master becomes inactive and slave acquires the same state as master as explained in (a) and (b) conditions above.

But why do we require this master-slave combination? To understand this, consider a situation where output of one flip-flop is going to be input of other flip-flop. Here, the assumption is that clock pulse inputs of all flip-flops are synchronized and occur at the same time. The change of state of master occurs when the clock pulse goes to 1 but during that time the output of slave still has not changed, thus the state of the flip-flops in the system can be changed simultaneously during the same clock pulse even though output of flip-flops are connected to the inputs of other flip-flops.

4.3.4 Edge-Triggered flip-flops

An edge-triggered flip-flop is used to synchronize the state change during a clock pulse transition instead of constant level. Some edge-triggered flip-flops trigger on the rising edge (0 to 1 transition) whereas others trigger on the falling edge (1- to 0 transition). Fig 4.10 shows the clock pulse signal in positive & negative edge-triggered flip-flops.

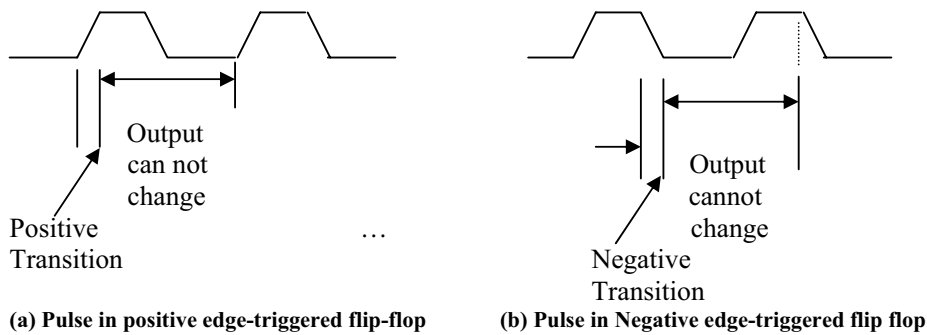


Figure. 4.10: Pulse signal

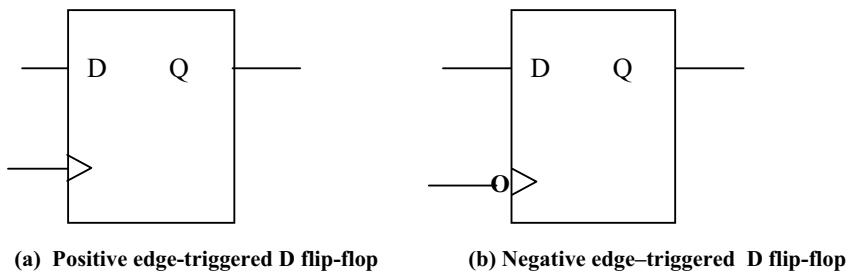


Figure 4.11: Edge triggered D flip- flops

The effective positive clock transition includes a minimum time called **setup time**, for which the D input must be maintained at constant value before the occurrence of clock transition. Similarly, a minimum time called **hold time**, for which the D input must not change after the application of positive transition of the pulse.

Check Your Progress 2

- What are the advantages of master- slave flip-flop?
.....
.....
- What are edge- triggered flip-flops?
.....
.....

4.4 SEQUENTIAL CIRCUIT DESIGN

A sequential circuit is specified by a time sequence of external inputs, external outputs and internal flip-flop binary states. Thus firstly, a *state table and state diagram* is used to describe behaviour of the circuit. Then from the state table, we get information for making logic circuit diagram.

Let us first see what is state table and state diagram. A **state table** includes the functional relationships between the inputs, output and flip-flop states (present and next) of a sequential circuit. A state diagram pictorially describes the state transition. In state diagram, a circle describes a state and directed lines indicate the transition between states. The state of flip-flop is written inside the circle. The directed lines are labelled with two binary numbers separated by a slash. The first one indicates the input value during present state and second number indicates output during present state. The state diagram of a binary counter is given in figure 4.12 (b).

The following is the procedure for design of sequential circuits:

- 1) Draw state table or state diagram from the problem statement, (if state diagram is available, draw state table also)
- 2) Give binary codes to states.
- 3) From state table, make input equation in simplified form. i.e. generating Boolean functions which describes signals for the inputs of flip-flops.
- 4) From state table, derive output equation in simplified form.
- 5) Draw logic diagram with required flip-flops and combinational circuits.

Let us take an example to illustrate the above procedure. Suppose we want to design 2-bit binary counter using D flip-flop. The circuit goes through repeated binary states 00, 01, 10 and 11 when external input $X = 1$ is applied. The state of circuit will not change when $X = 0$. The state table & state diagram for this is shown in figure 4.12. But how do we make this state diagram? Please note the number of flip-flops– 2 in our example as we are designing 2 bits counter. Various states of two bit input would be 00 01 10 and 11. These are shown in circle. The arrow indicate the transitions on an input value X . For example, when the counter is in state 00 and input value $X=0$ occurs, the counter remains in 00 state. Hence the loop back on $X=0$. However, on encountering $X=1$ the counter moves to state 01. Like wise in all other states similar transition occur. For making state table remember the excitation table of D flip-flop given in figure 4.8 (c).

The present state of the two flip-flops and next states of the flip-flops are put into the table along with any input value. For example, if the present state of flip-flops is 01 and input value is 1 then counter will move to state 10. Notice these values in the fourth row of the values in the state table (figure 4.12 (a))

Or we can write as

A	B	→	A (Next)	B (Next)
0	1 $X=1$		1	0

This implies that flip-flop. A has moved from state clear to set. As we are making the counter using D flip-flop, the question is what would be the input D_A value of A flip-flop that allows this transition that is $Q(t) = 0$ to $Q(t+1) = 1$ possible for A flip flop. On checking the excitation table for D Flip-flop, we find the value of D input of A flip-flop (called D_A in this example) would be 1. Similarly, the B flip-flop have a transition $Q(t) = 1$ to $Q(t+1)=0$, thus, D_B , would be 0. Hence notice the values of flip-flop inputs D_A and D_B . (Row 3).

a) State Table

(b) State Diagram

Figure 4.12: Binary Counter Design

Next step indicates simplification of input equation to flip-flop which is done using K-Maps as shown in fig 4.13. But why did we make K-map for D_A or D_B which happens to be flip-flop input values? Please note in sequential circuit design, we are

designing the combinational logic that controls the state transition of flip-flops. Thus, each input to a flip-flop is one output of this combinational logic and the present state of flip-flops and any other input value form the input values to this combinational logic.

Figure 4.13: Maps for combinational circuit of 2-bit counter

Thus, two simplified flip-flop input equations are derived:

$$D_A = A\bar{B} + A\bar{X} + \bar{A}BX$$

$$D_B = \bar{B}X + B\bar{X}$$

The logic diagram is drawn in fig 4.14.

Figure 4.14: Logic diagram for 2-bit Binary Counter

Note: Similarly, the sequential circuits can be designed using any number of flip-flops using state diagrams and combinational circuits design methods.

4.5 EXAMPLES OF SEQUENTIAL CIRCUITS

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

4.5.1 Registers

A register is a group of flip-flops, which store binary information, and gates, which controls when and how information is transferred to the register. An n-bit register has n flip-flops and stores n-bits of binary information. Two basic types of registers are: parallel registers and shift registers.

A parallel register is one of the simplest registers, consisting of a set of flip-flops that can be read or written simultaneously. Fig. 4.15 shows a 4-bit register with parallel input-output. The signal lines I_0 to I_3 inputs to flip-flops, which may be output of other arithmetic circuits like multipliers, so that data from different sources can be loaded into the register. It has one additional line called clear line, which can clear the register completely. This register is called a parallel register as all the bits of the register can be loaded in a single clock pulse.

Figure 4.15: 4-bit parallel register

A shift register is used for shifting the data to the left or right. A shift register operates in serial input-output mode i.e. data is entered in the register one bit at a time from one end of the register and can be read from the other end as one bit at a time. Fig. 4.16 shows a 4-bit right shift register using D logical shift functions.

Figure 4.16: 4-bit right – shift register

Please note that in this register signal shift enable is used instead of clock pulse, why? Because it is not necessary that we want the register to perform shift on each clock pulse.

A register, which shifts data only in one direction, is called **uni-directional shift register** and a register, which can shift data in both directions, is called **bi-directional shift register**. Shift register can be constructed for bi-directional shift with parallel input-output. A general shift register structure may have parallel data transfer to or from the register along with added facility of left or right shift. This structure will require additional control lines for indicating whether parallel or serial output is desired and left or right shift is required. A general symbolic diagram is shown in Fig. 4.17 for this register.

There are 3 main control lines shown in the above figure. If parallel load enable is active, parallel input-output operation is done otherwise serial input- output shift select line for selecting right or left shift. If it has value 0 then right shift is performed and for value 1, left shift is done. Shift enable signal indicates when to start shift.

Figure. 4.17: 4 – bit left shift register with parallel load

4.5.2 Counters Asynchronous Counters

A counter is a register, which goes through a predetermined sequence of states when clock pulse is applied. In principle, the value of counters is incremented by 1 module the capacity of register i.e. when the value stored in a counter reaches its maximum value, the next incremented value becomes zero. The counters are mainly used in circuits of digital systems where sequence and control operations are performed, for example, in CPU we have program counter (PC).

Counters can be classified into two categories, based on the way they operate: Asynchronous and synchronous counters. In Asynchronous counters, the change in state of one flip-flop triggers the other flip-flops. Synchronous counters are relatively faster because the state of all flip-flops can be changed at the same time.

Asynchronous Counters : This is more often referred to as ripple counter, as the change, which occurs in order to increment the counter ripples through it from one end to the other. Fig 4.18 shows an implementation of 4-bit ripple counter using J-K flip-flops. This counter is incremented on the occurrence of each clock pulse and counts from 0000 to 1111 (i.e. 0 to 15).

Figure. 4.18: 4 – bit ripple counter

The input line to J & K of all flip-flops is kept high i.e. logic1. Each time a clock pulse occurs the value of flip-flop is complemented (Refer to characteristic table of J K flip-flop in Figure. 4.6 (c)). Please note that the clock pulse is given only to first flip-flop and second flip-flop onwards, the output of previous flip-flop is fed as clock signal. This implies that these flip-flops will be complemented if the previous flip-flop has a value 1. Thus, the effect of complement will ripple through these flip-flops.

4.5.3 Synchronous Counters

The major disadvantage of ripple counter is the delay in changing the value. How? To understand this, take an instance when the state of ripple counter is 0111. Now the next state will be 1000, which means change in the state of all flip-flops. But will it occur simultaneously in ripple counter? No, first O_0 will change then O_1 , O_2 & lastly O_3 . The delay is proportional to the length of the counter. Therefore, to avoid this disadvantage of ripple counters, synchronous counters are used in which all flip-flops change their states at same time. Fig 4.19 shows 3-bit synchronous counter.

Figure 4.19: Logic diagram of 3-bit synchronous counter

You can understand the working of this counter by analyzing the sequence of states (O_0 , O_1 , O_2) given in Figure 4.20

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

Figure. 4.20 : Truth table for 3 bit synchronous counter

The operation can be summarized as: -

- i) The first flip-flop is complemented in every clock cycle
- ii) The second flip-flop is complemented on occurrence of a clock cycle if the current state of first flip-flop is 1.
- iii) The third flip-flop is fed by an AND gate which is connected with output of first and second flip-flops. It will be complemented only when first & second flip-flops are in Set State.

4.5.4 RAM (Random Access Memory)

Here we will confine our discussion, in general to the RAM only as an example of sequential circuit. A memory unit is a collection of storage cells or flip flops alongwith associated circuits required to transfer information in and out of the device. The access time and cycle time it takes are constant and independent of the location, hence the name random access memory.

(a) Block Diagram

(b) Logic Diagram

Figure 4.21: Binary Cell

RAMs are organized (logically) as words of fixed length. The memory communicates with other devices through data input and output lines, address selection lines and control lines that specify the direction of transfer.

Now, let us try to understand how data is stored in memory. The internal construction of a RAM of 'm' words and 'n' bits per word consists of $m \times n$ binary cells and associated circuits for detecting individual words. Figure 4.21 shows logic diagram and block diagram of a binary cell.

The input is fed to AND gate 'a' in complemented form. The read operation is indicated by 1 on read/ write signal. Therefore during the read operation only the 'AND' gate 'c' becomes active. If the cell has been selected, then the output will become equal to the state of flip flop i.e. the data value stored in flip flop is read. In write operation 'a' & 'b' gates become active and they set or clear the J-K flip flop depending upon the input value. Please note in case input is 0, the flip flop will go to clear state and if input is 1, the flip flop will go to set state. In effect, the input data is

reflected in the state of flip-flop. Thus, we say that input data has been stored in flip-flop or binary cell.

Fig 4.22 is the extension of this binary cell to an IC RAM circuit, where a 2×4 decoder is used to select one of the four words. (For 4 words we need 2 address lines) Please note that each decoder output is connected to a 4bit word and the read/write signal is given to each binary cell. Once the decoder selects the word, the read/write input tells the operation. This is derived using an OR gate, since all the non-selected cells will produce a zero output. When the memory select input to decoder is 0, none of the words is selected and the contents of the cell are unchanged irrespective of read/write input.

Figure 4.22: 4×4 RAM

After discussing so much about combinational circuits and sequential circuits, let us discuss in the next section an example having a combination of both circuits.

4.6 DESIGN OF A SAMPLE COUNTER

Let us design a synchronous BCD counter. A BCD counter follows a sequence of ten states and returns to 0 after the count of 9. These counters are also called **decade counters**. This type of counter is useful in display applications in which BCD is required for conversion to a decimal readout. Fig 4.23 shows the characteristic table for this counter.

Present State				Next State				Flip-Flops Inputs							
A	B	C	D	A	B	C	D	J _A	K _A	J _B	K _B	J _C	K _C	J _D	K _D
0	0	0	0	0	0	0	1	0	X	0	X	0	X	1	X
0	0	0	1	0	0	1	0	0	X	0	X	1	X	X	1
0	0	1	0	0	0	1	1	0	X	0	X	X	0	1	X
0	0	1	1	0	1	0	0	0	X	1	X	X	1	X	1
0	1	0	0	0	1	0	1	0	X	X	0	0	X	1	X
0	1	0	1	0	1	1	0	0	X	X	0	1	X	X	1
0	1	1	0	0	1	1	1	0	X	X	0	X	0	1	X
0	1	1	1	1	0	0	0	1	X	X	1	X	1	X	1
1	0	0	0	1	0	0	1	0	X	0	X	0	X	1	X
1	0	0	1	0	0	0	0	0	X	0	X	0	X	X	1

Figure 4.23: Characteristic table for decade counter

[NOTE : Remember excitation table for J-K flip flop given in fig 4.8]

There are 4 flip-flop inputs for decade counter i.e. A, B, C, D. The next state of flip-flop is given in the table. J_A & K_A indicates the flip flop input corresponding to flip-flop-A. Please note this counter require 4-flip-flops.

From this the flip flop input equations are simplified using K-Maps as shown in figure 4.24. The unused minterms from 1010 through 1111 are taken as don't care conditions.

Figure 4.24: K-maps for Decade counter

Thus, the simplified input equation for BCD counter are:

$$\begin{array}{llll}
 J_A & = & BCD & K_A & = & D \\
 J_B & = & CD & K_B & = & CD \\
 J_C & = & \overline{A} D & K_C & = & D \\
 J_D & = & 1 & K_D & = & 1
 \end{array}$$

The logic circuit can be made with 4 JK flip flops & 3 AND gates

Figure 4.25: Logic Diagram for decade counter.

Check Your Progress 3

- 1) Differentiate between synchronous & asynchronous counters?
.....
.....
.....
- 2) Can ripple counter be constructed from a shift register?
.....
.....
.....
- 3) Can we design a counter with the following repeated binary sequence:
0,1,2,3,4,5,6. If yes, design it using J K flip flop.
.....
.....
.....

4.7 SUMMARY

As told to you earlier this unit provides you information regarding sequential circuits which is the foundation of digital design. Flip-flops are basic storage unit in sequential circuits are derived from the latches. The sequential circuit can be formed using combinational circuits (discussed in the last unit) and flip flops. The behavior of sequential circuit can be analyzed using tables & state diagrams.

Registers, counters etc. are structured sequential blocks. This unit has outlined the construction of registers, counters, RAM etc. Lastly, we discussed how a circuit can be designed using both sequential & combinational circuits. For more details, the students can refer to further reading.

4.8 SOLUTIONS / ANSWERS

Check Your Progress 1

- 1) An interconnection of combinational circuits and flip-flops, used for making different logic devices in computers that involves manipulation and storage of data. Some such circuits are registers, counters etc.
- 2) Flip flop is the basic storage element for synchronous sequential circuits. Whereas latches are bistable devices whose state normally depends upon the asynchronous inputs and are not suitable for use in synchronous sequential circuits using single clock.
- 3) Excitation table indicates that if present and next state are known then what will be inputs whereas a characteristics table indicates just opposite of this i.e. inputs are known the, next state has to be found.

Check Your Progress 2

- 1) The main advantage is that they allow feed back paths
- 2) Edge-Triggered flip-flops are bi-stable devices with synchronous inputs whose state depends on the inputs only at the triggering transition of a clock pulse i.e. changes in output occur only at triggering transition of the clock

Check Your Progress 3

- 1) The main difference is the time when the counter flip-flops change its states. In synchronous counter all the flip flops that need to change; change simultaneously. In asynchronous counter the complement if to be done may ripple through a series of flip-flops.
- 2) Yes, but this: circuit will generate sequence of states where only 1-bit changes at a time i.e. 0000, 1000, 1100, 1110, 1111, 0111, 0011, 0001
- 3) Yes, We require 2^3 i.e. three flip flops for the sequence 0, 1, 2, 3, 4, 5&6.

Present State			Next State			Flip – Flops Inputs					
A	B	C	A	B	C	J _A	K _A	J _B	K _B	J _C	K _C
0	0	0	0	0	1	0	X	0	X	1	X
0	0	1	0	1	0	0	X	1	X	X	1
0	1	0	0	1	1	0	X	X	0	1	X
0	1	1	1	0	0	1	X	X	1	X	1
1	0	0	1	0	1	X	0	0	X	1	X
1	0	1	1	1	0	X	0	1	X	X	1
1	1	0	0	0	0	X	1	X	1	0	X

The state is don't care condition: Make the suitable K-maps. The following are the flip-flop input values:

$$J_A = BC$$

$$K_A = B$$

$$J_B = C$$

$$K_B = C + A$$

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

$$K_C = 1$$

The circuit can be constructed as:

UNIT 1 THE MEMORY SYSTEM

Structure	Page Nos.
1.0 Introduction	5
1.1 Objectives	5
1.2 The Memory Hierarchy	5
1.3 RAM, ROM, DRAM, Flash Memory	7
1.4 Secondary Memory and Characteristics	13
1.4.1 Hard Disk Drives	
1.4.2 Optical Memories	
1.4.3 CCDs, Bubble Memories	
1.5 RAID and its Levels	21
1.6 The Concepts of High Speed Memories	26
1.6.1 Cache Memory	
1.6.2 Cache Organisation	
1.6.3 Memory Interleaving	
1.6.4 Associative Memory	
1.7 Virtual Memory	34
1.8 The Memory System of Micro-Computer	36
1.8.1 SIMM, DIMM, etc., Memory Chips	
1.8.2 SDRAM, RDRAM, Cache RAM Types of Memory	
1.9 Summary	39
1.10 Solutions /Answers	39

1.0 INTRODUCTION

In the previous Block, we have touched upon the basic foundation of computers, which include concepts on von Neumann machine, instruction, execution, the digital data representation and logic circuits. In this Block we will define some of the most important component units of a computer, which are the memory unit and the input-output units. In this unit we will discuss various components of the memory system of a computer system. Computer memory is organised into a hierarchy to minimise cost. Also, it does not compromise the overall speed of access. Memory hierarchy include cache memory, main memory and other secondary storage technologies. In this Unit, we will discuss the main memory, the secondary memory and high-speed memories such as cache memory, and the memory system of microcomputer.

1.1 OBJECTIVES

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

- describe the key characteristics of the memory system;
- distinguish among various types of random access memories;
- describe the latest secondary storage technologies;
- describe the importance of cache memory and other high-speed memories; and
- describe the different memory chips of micro computers.

1.2 THE MEMORY HIERARCHY

Memory in a computer system is required for storage and subsequent retrieval of the instructions and data. A computer system uses a variety of devices for storing these instructions and data that are required for its operation. Normally we classify the information to be stored into two basic categories: Data and Instructions. But what is a memory system?

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

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

But does this kind of technology where there is no speed gap between processor and memory speed exist? The answer is yes, it does. Unfortunately as the access time (time taken by CPU to access a location in memory) becomes less the cost per bit of memory becomes higher. In addition, normally these memories require power supply till the information needs to be stored. Both these things are not very convenient, but on the other hand the memories with smaller cost have very high access time that will result in slower operation of the CPU. Thus, the cost versus access time anomaly has led to a hierarchy of memories where we supplement fast memories with larger, cheaper, slower memories. These memory units may have very different physical and operational characteristics; therefore, the memory system is very diverse in type, cost, organisation, technology and performance. This memory hierarchy will work only if the frequency of access to the slower memories is significantly less than the faster memories. The memory hierarchy system consists of all storage devices employed in a computer system from the slow but high capacity auxiliary memory to a relatively faster main memory, to an even smaller and faster cache memory accessible to the high speed registers and processing logic. Figure 1 illustrates the components of a typical memory system.

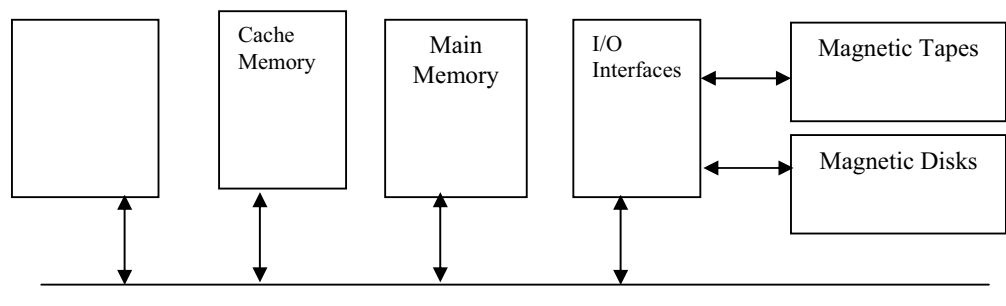


Figure 1: The Memory Hierarchy (Block Diagram)

A typical storage hierarchy is shown in Figure 1 above. Although Figure 1 shows the block diagram, it includes the storage hierarchy:

Register
Cache memory
Main memory
Secondary Storage and
Mass Storage.

As we move up the hierarchy, we encounter storage elements that have faster access time, higher cost per bit stored, and slower access time as a result of moving down the hierarchy. Thus, cache memory generally has the fastest access time, the smallest storage capacity, and the highest cost per bit stored. The primary memory (main memory) falls next in the storage hierarchy list. On-line, direct-access secondary storage devices such as magnetic hard disks make up the level of hierarchy just below the main memory. Off-line, direct-access and sequential access secondary storage devices such as magnetic tape, floppy disk, zip disk, WORM disk, etc. fall next in the storage hierarchy. Mass storage devices, often referred to as archival storage, are at

the bottom of the storage hierarchy. They are cost-effective for the storage of very large quantities of data when fast access time is not necessary.

Please note two important points here:

- The size of the memory increases as we move down the hierarchy.
- The quantum of data that is transferred between two consecutive memory layers at a time also increases as we go from a higher to lower side. For example, from main memory to Cache transfer one or few memory words are accessed at a time, whereas in a hard disk to main memory transfer, a block of about 1 Megabyte is transferred in a single access. You will learn more about this in the later sections of the unit.

Let us now discuss various forms of memories in the memory hierarchy in more details.

1.3 RAM, ROM, DRAM, FLASH MEMORY

RAM (Random Access Memory)

The main memory is a random access memory. It is normally organised as words of fixed length. The length of a word is called word length. Each of these memory words has an independent address and each has the same number of bits. Normally the total numbers of words in memory are some power of 2. Some typical memory word sizes are 8 bits, 16 bits, 32 bits etc. The main memory can be both read and written into, therefore it is called read-write memory.

The access time and cycle time in RAMs are constant and independent of the location accessed. How does this happen? To answer this, let us first discuss how a bit can be stored using a sequential circuit. The logic diagram of a binary cell is shown in Figure 2a:

Figure 2(a) Logic Diagram of RAM cell

The construction shown in Figure 2(a) is made up of one JK flip-flop and 3 AND gates. The two inputs to the system are one input bit and read/write signal. Input is fed in complemented form to AND gate 'a'. The read/write signal has a value of 1 if it is a read operation. Therefore, during the read operation the AND gate 'c' has the read/write input as 1. Since AND gate 'a' and 'b' have 0 read/write input, and if the

chip is selected i.e. this cell is currently being selected, then output will become equal to the state of flip-flop. In other words the data value stored in flip-flop has been read. In write operation only 'a' and 'b' gates get a read/write value of 1 and they set or clear the JK flip-flop depending on the data input value. If the data input is 0, the flip-flop will go to clear state and if data input is 1, the flip-flop will go to set state. In effect, the input data is reflected in the state of the flip-flop. Thus, we say that the input data has been stored in flip-flop or binary cell.

Figure 2(b) Internal Organisation of a 32×4 RAM

A 32×4 RAM means that this RAM has 32 words, 5 address lines ($2^5 = 32$), and 4 bit data word size. Please note that we can represent a RAM using $2^A \times D$, where A is the number of address lines and D is the number of Data lines. Figure 2 (b) is the extension of the binary cell to an integrated 32×4 RAM circuit where a 5×32 bit decoder is used. The 4 bit data inputs come through an input buffer and the 4-bit data output is stored in the output buffer.

A chip select (\overline{CS}) control signal is used as a memory enable input. When $\overline{CS} = 0$ that is $\overline{CS} = 1$, it enables the entire chip for read or write operation. A R/W signal can be used for read or write operation. The word that is selected will determine the overall output. Since all the above is a logic circuit of equal length that can be accessed in equal time, thus, the word RAM.

DRAM (Dynamic Random Access Memory)

RAM technology is divided into two technologies: dynamic and static. A dynamic RAM (DRAM) is made with cells that store data as charge on capacitors. The presence or absence of charge on capacitor is interpreted as binary 1 or 0. Because capacitors have a natural tendency to discharge, dynamic RAM requires periodic charge refreshing to maintain data storage. The term dynamic refers to this tendency of the stored charge to leak away, even with power continuously applied.

Figure 3(a) is a typical DRAM structure for an individual cell that stores one bit. The address line is activated when the bit value from this cell is to be read or written. The transistor acts as

a switch that is closed (allowing current to flow) if a voltage is applied to the address line and open (no current flows) if no voltage is present on the address line.

Figure 3(a): DRAM Cell

Figure 3(b): Typical 16 Megabit DRAM

For the write operation (please refer to Figure 3 (a), a voltage signal is applied to the bit line; a high voltage represents 1, and a low voltage represents 0. A signal is then applied to the address line, allowing a charge to be transferred to the capacitor.

For the read operation, when the address line is selected, the transistor turns on and the charge stored on the capacitor is fed out onto a bit line and to the sense amplifier. The sense amplifier compares the capacitor voltage to a reference value and determines if the cell contains logic 1 or logic 0. The read out from the cell discharges the capacitor, which **must be restored** to complete the operation.

Although the DRAM cell is used to store a single bit (0 or 1), it is essentially an analog device. The capacitor can store any charge value within a range; a threshold value determines whether the charge is interpreted as 1 or 0.

Organisation of DRAM Chip

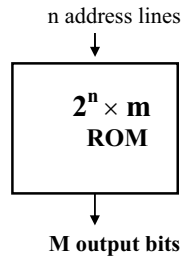
The Figure 3(b) is a typical organisation of 16 mega bit DRAM. It shows a typical organisation of $2048 \times 2048 \times 4$ bit DRAM chip. The memory array in this organisation is a square array that is (2048×2048) words of 4 bits each.

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

- Row address selection specifying the present address values A0 to A10 (11 address lines only). For the rows, it is stored in the row address buffer through decoder.
- Row decoder selects the required row.
- The column address buffer is loaded with the column address values, which are also applied to through A0 to A10 lines only. Please note that these lines should contain values for the column.
- This job will be done through a change in external signal $\overline{\text{RAS}}$ (Row address Strobe) because this signal is high at the rising edge of the clock.
- $\overline{\text{CAS}}$ (Column address Strobe) causes the column address to be loaded with these values.
- Each column is of 4 bits, that is, those require 4 bit data lines from input/output buffer. On memory write operation data in bit lines being activated while on read sense lines being activated.
- This chip requires 11 address lines (instead of 22), 4 data in and out lines and other control lines.
- As there are 11 row address lines and 11 column address lines and each column is of 4 bits, therefore, the size of the chip is $2^{11} \times 2^{11} \times 4 = 2048 \times 2048 \times 4 = 16$ mega bits. On increasing address lines from 11 to 12 we have $2^{12} \times 2^{12} \times 4 = 64$ mega bits, an increase of a factor of 4. Thus, possible sizes of such chips may be 16K, 256K, 1M, 4M, 16M, and so on.
- Refreshing of the chip is done periodically using a refresh counter. One simple technique of refreshing may be to disable read-write for some time and refresh all the rows one by one.

ROM (Read-Only Memory)

A ROM is essentially a memory or storage device in which a fixed set of binary information is stored. A block diagram of ROM is as shown in Figure 4(a). It consists of n input lines and m output lines. Each bit combination of the input variables is called an **address**. Each bit combination that comes out of the output lines is called a **word**. The number of bits per word is equal to the number of output lines m . The number of distinct addresses possible with n input variables is 2^n .



(a) ROM Block diagram

Input		Output	
I_1	I_2	O_1	O_2
0	0	0	1
0	1	1	0
1	0	1	1
1	1	0	0

(b) Truth table

(c) A Sample ROM

Figure 4: ROM

A ROM is characterised by the number of words (2^n) and the number of bits (m) per word. For example, a 32×8 ROM which can be written as $2^5 \times 8$ consists of 32 words of 8 bit each, which means there are 8 output lines and 32 distinct words stored in the unit. There are only 5 input lines because $32 = 2^5$ and with 5 binary variables, we can specify 32 addresses.

A ROM is basically a combinational circuit and can be constructed as shown in Figure 4(c). On applying an Input $I_1 = 0$, $I_2 = 0$, the 00 line of the decoder is selected and we will get $O_1 = 0$ and $O_2 = 1$; on applying $I_1 = 0$ and $I_2 = 1$ we will get $O_1 = 1$ AND $O_2 = 0$. This same logic can be used for constructing larger ROMs.

ROMs are the memories on which it is not possible to write the data when they are on-line to the computer. They can only be read. This is the reason why it is called read-only memory (ROM). Since ROM chips are non-volatile, the data stored inside a ROM are not lost when the power supply is switched off, unlike the case of a volatile RAM chip. ROMs are also known as permanent stores.

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

The ROMs / PROMs can be written just once, but in both the cases whatever is written once cannot be changed. But what about a case where you read mostly but write only very few times? This led to the concepts of read mostly memories and the best example of these are EPROMs (Erasable PROMs) and EEPROMs (Electrically Erasable PROMs).

The EPROMs can be read and written electrically. But, the write operation is not simple. It requires erasure of whole storage cells by exposing the chip to ultra violet light, thus bringing them to the same initial state. Once all the cells have been brought to same initial state, then the EPROM can be written electrically. EEPROMs are becoming increasingly popular, as they do not require prior erasure of previous

contents. However, in EEPROMS the writing time is considerably higher than the reading time. The biggest advantage of EEPROM is that it is non-volatile memory and can be updated easily, while the disadvantages are the high cost and at present they are not completely non-volatile and the write operation takes considerable time. But all these advantages are disappearing with growth in technology. In general, ROMs are made of cheaper and slower technology than RAMs.

Flash Memory

This memory is another form of semiconductor memory, which was first introduced in the mid-1980. These memories can be reprogrammed at high speed and hence the name flash. This is a type of non-volatile, electronic random access memory.

Basically this memory falls in between EPROM and EEPROM. In flash memory the entire memory can be erased in a few seconds by using electric erasing technology. Flash memory is used in many I/O and storage devices. *Flash memory is also used to store data and programming algorithms in cell phones, digital cameras and MP3 music players.*

Flash memory serves as a hard drive for consumer devices. Music, phone lists, applications, operating systems and other data are generally stored on stored on flash chips. *Unlike the computer memory, data are not erased when the device is turned off.*

There are two basic kinds of flash memory:

Code Storage Flash made by Intel, AMD, Atmel, etc. It stores programming algorithms and is largely found in cell phones.

Data Storage Flash made by San Disk, Toshiba, etc. It stores data and comes in digital cameras and MP3 players.

The feature of semiconductor memories are summarised in the Figure 5.

Memory	Category	Erase	Write Mechanism	Volatility
Random-access Memory (RAM)	Read-write memory	Electrically, byte level	Electrically	Volatile
Read-only Memory (ROM)	Read-only memory	Not possible	Masks	Non-volatile
Programmable ROM (PROM)	Read-only memory	Not possible	Electrically	Non-volatile
Erasable PROM (EPROM)	Read-mostly memory	UV light chip level	Electrically	Non-volatile
Electrically Erasable (EEPROM)	Read-mostly memory	Electrically, byte level	Electrically	Non-volatile
Flash memory	Read-mostly memory	Electrically, block level	Electrically	Non-volatile

Figure 5: Features of Semiconductor Memories

1.4 SECONDARY MEMORY AND CHARACTERISTICS

It is desirable that the operating speed of the primary storage of a computer system be as fast as possible because most of the data transfer to and from the processing unit is via the main memory. For this reason, storage devices with fast access times, such as semiconductors, are generally used for the design of primary storage. These high-speed storage devices are expensive and hence the cost per bit of storage is also high for a primary storage. *But the primary memory has the following limitations:*

- a) **Limited capacity:** The storage capacity of the primary storage of today's computers is not sufficient to store the large volume of data handled by most of the data processing organisations.
- b) **Volatile:** The primary storage is volatile and the data stored in it is lost when the electric power is turned off. However, the computer systems need to store data on a permanent basis for several days, months or even several years.

The result is that an additional memory called secondary storage is used with most of the computer systems. Some popular memories are described in this section.

1.4.1 Hard Disk Drive

This is one of the components of today's personal computer, having a capacity of the order of several Giga Bytes and above. A magnetic disk has to be mounted on a disk drive before it can be used for reading or writing of information. A disk drive contains all the mechanical, electrical and electronic components for holding one or more disks and for reading or writing of information on it. That is, it contains the central shaft on which the disks are mounted, the access arms, the read/write head and the motors to rotate the disks and to move the access arms assembly. Now-a-days, the disk drive assembly is packed in very small casing although having very high capacity. Now let us know about what a magnetic disk is.

Magnetic Disk

A disk is circular platter constructed of nonmagnetic material, called the substrate, coated with a magnetisable material. This is used for storing large amount of data. Traditionally, the substrate has been an aluminium or aluminium alloy material; more recently, glass substrates have been introduced. The glass substrate has a number of benefits, including the following:

- Improvement in the uniformity of the magnetic film surface to increase disk reliability.
- A significant reduction in overall surface to help reduce read-write errors.
- Ability to support lower fly heights.
- Better stiffness to reduce disk dynamics.
- Greater ability to withstand shock and damage.

Magnetic Read and Write Mechanisms

Data are recorded on and later retrieved from the disk via a conducting coil named the head; in many systems there are two heads, a read head and a write head. During a read or write operation, the head is stationary while the platter rotates beneath it.

Figure 6: Read /write Heads

The write mechanism is based on the fact that electricity flowing through a coil produces a magnetic field. Pulses are sent to the write head, and magnetic patterns are recorded on the surface below, with different patterns for positive and negative currents. The write head itself is made of easily magnetisable material and is in the shape of a rectangular doughnut with a gap along one side and a few turns of conducting wire along the opposite side (Figure 6). An electric current in the wire induces a magnetic field across the gap, which in turn magnetizes a small area of the recording medium. Reversing the direction of the current reverses the direction of the magnetization on the recording medium.

The traditional read mechanism is based on the fact that a magnetic field moving relative to a coil produces an electrical current in the coil. When the surface of the disk passes under the head, it generates a current of the same polarity as the one already recorded. The structure of the head for reading is in this case essentially the same as for writing and therefore the same head can be used for both. Such single heads are used in floppy disk systems and in older rigid disk systems.

Data Organization and Formatting

The head is a relatively small device capable of reading from or writing to a portion of the platter rotating beneath it. This gives rise to the organization of data on the platter in a concentric set of rings, called tracks; each track is of the same width as the head. There are thousands of tracks per surface.

Figure 7 depicts this data layout. Adjacent tracks are separated by gaps. This prevents, or at least minimizes, errors due to misalignment of the head. Data are transferred to and from the disk in sectors. To identify the sector position normally there may be a starting point of a track and a starting and end point of each sector. But the question is how is a sector of a track recognised? A disk is formatted to record control data on it such that some extra data are stored on it for identical purpose. This control data is

accessible only to the disk drive and not to the user. Please note that in Figure 7 as we move away from the centre of a disk the physical size of the track is increasing. Does it mean we store more data on the outside tracks? No. A disk rotates at a constant angular velocity. But, as we move away from centre the liner velocity is more than the liner velocity nearer to centre. Thus, the density of storage of information decreases as we move away from the centre of the disk. This results in larger physical sector size. Thus, all the sectors in the disk store same amount of data.

An example of disk formatting is shown in Figure 8. In this case, each track contains 30 fixed-length sectors of 600 bytes each. Each sector holds 512 bytes of data plus control information useful to the disk controller. The ID field is a unique identifier or address used to locate a particular sector. The SYNC byte is a special bit pattern that delimits the beginning of the field. The track number identifies a track on a surface. The head number identifies a head, because this disk has multiple surfaces. The ID and data fields each contain an error-detecting code.

Figure 8: A typical Track Format for Winchester Disk

Physical Characteristics

Figure 9 lists the major characteristics that differentiate among the various types of magnetic disks. First, the head may either be fixed or movable either respect to the radial direction of the platter. In a fixed-head disk, there is one read-write head per track. All of the heads are mounted on a rigid arm that extends across all tracks; such systems are rare today. In a movable-head disk, there is only one read-write head. Again, the head is mounted on an arm. Because the head must be able to be positioned above any track, the arm can be extended or retracted for this purpose.

Head Motion	Platters
Fixed head (one per track)	Single platter
Moveable head (one per surface)	Multiple platter
Disk Portability	Head mechanism
Non-removable	Disk Contact (floppy)

Removable disk	Fixed gap
Sides	Aerodynamic gap (Winchester)
Single sided	
Double sided	

Figure 9: Physical characteristics of Disk Systems

The disk itself is mounted in a disk drive, which consists of the arm, a shaft that rotates the disk, and the electronics needed for input and output binary data. A non-removable disk is permanently mounted in the disk drive; the hard disk in a personal computer is a non-removable disk. A removable disk can be removed and replaced with another disk. The advantage of the latter type is that unlimited amounts of data are available with a limited number of disk systems. Furthermore, ZIP cartridge disks are examples of removable disks. Figure 10 shows other components of the disks.

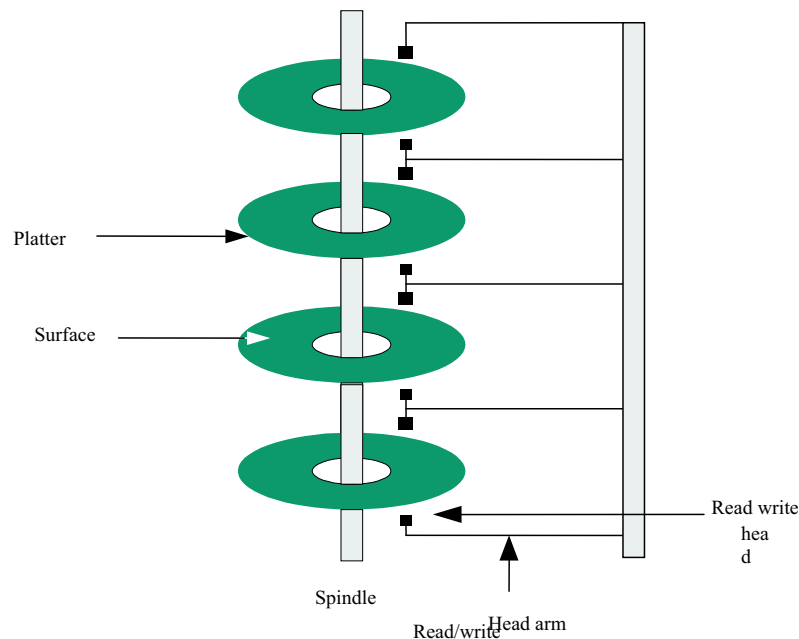


Figure 10: The Disk Components

The head mechanism provides a classification of disks into three types. Traditionally, the read-write head has been positioned at a fixed distance above the platter, allowing an air gap. At the other extreme is a head mechanism that actually comes into physical contact with the medium during a read or write operation. This mechanism is used with the floppy disk, which is a small, flexible platter and the least expensive type of disk.

To understand the third type of disk, we need to comment on the relationship between data density and the distance of head from the surface. The head generates or senses an electromagnetic field of sufficient magnitude to write and read properly. The narrower the head is, the closer it must be to the platter surface to function. A narrower head means narrower tracks and therefore greater data density, which is desirable. However, the closer the head is to the disk, the greater are the risks of errors from impurities or imperfections.

To push the technology further, the Winchester disk was developed. Winchester heads are used in sealed drive assemblies that are almost free of contaminants. They are designed to operate closer to the disk's surface than conventional rigid disk heads, thus allowing greater data density. The head is actually an aerodynamic foil that rests

lightly on the platter's surface when the disk is motionless. The air pressure generated by a spinning disk is enough to make the foil rise above the surface. The resulting non-contact system can be engineered to use narrower heads that operate closer to the platter's surface than conventional rigid disk heads.

Accessing the Disk Data

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

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

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

1.4.2 Optical Memories

In 1983, one of the most successful consumer products of all times was introduced: the compact disk (CD) digital audio system. This CD was a non-erasable disk that could store more than 60 minutes of audio information on one side. The huge commercial success of this CD enabled the development of low-cost optical-disk storage technology that has revolutionised computer data storage. A variety of optical-disk systems has been introduced. We briefly review each of these.

Compact Disk ROM (CD-ROM)

Both the audio CD and the CD-ROM (compact disk read-only memory) share a similar technology. The main difference is that CD-ROM players are more rugged and have error correction devices to ensure that data are properly transferred from disk to computer. Both types of disk are made the same way. The disk is formed from a resin, such as polycarbonate. Digitally recorded information (either music or computer data) is imprinted as a series of microscopic pits on the surface of the polycarbonate. The pitted surface is then coated with a highly reflective surface, usually aluminium. This shiny surface is protected against dust and scratches by a topcoat of clear acrylic. Finally, a label can be silk-screened onto the acrylic.

Figure 11: The CD Surface and Operation

Information is retrieved from a CD or CD-ROM by a low-powered laser housed in an optical-disk player, or drive unit. The laser shines through the clear polycarbonate while a motor spins the disk past it (Figure 11). The intensity of the reflected light of the laser changes as it encounters a pit. Specifically, if the laser beam falls on a pit, the light scatters and a low intensity is reflected back to the source. The areas between pits are called **lands**. A land is a smooth surface, which reflects back at a higher intensity. The change between pits and lands is detected by a **photo sensor** and converted into a digital signal. The sensor tests the surface at regular intervals. The beginning or end of a pit represents a 1; when no change in elevation occurs between intervals, a 0 is recorded.

Data on the CD-ROM are organised as a sequence of blocks. A typical block format is shown in Figure 12 (a). It consists of the following fields:

- **Sync:** The sync field identifies the beginning of a block. It consists of a byte of all 0s, 10 bytes of all 1s, and bytes of all 0s.
- **Header:** The header contains the block address and the mode byte. Mode 0 specifies a blank data field; mode 1 specifies the use of an error-correcting code and 2048 bytes of data; mode 2 specifies 2336 bytes of user data with no error correcting code.
- **Data:** User data.
- **Auxiliary:** Additional user data in mode 2. In mode 1, this is a 288-byte error correcting code.

(a) Block Format

(b) CD-ROM Layout

But what is the Min (Minute), Sec (Second) and Sector fields in the Header field?

The sectors of CD-ROM are not organised like the sectors in hard disks (Please refer Figure 12(b)). Rather, they are all equal length segments. If we rotate the CD drive at constant speed the linear velocity of disk surface movement will be higher at the outer side than that of the centre portions. To offset this linear speed gap, either we store less data on the outer sectors or we reduce the speed of rotation while reading outer tracks. The CD follows the later approach, that is, instead of moving the CD drive at constant velocity, it is rotated at variable velocity. The speed of rotation of disk reduces as we move away from the centre such that the sector's can be read in constant time. This method of reading is called Constant Linear Velocity (CLV).

CD-ROM is appropriate for the distribution of large amounts of data to a large number of users. CD-ROMs are a common medium these days for distributing information. Compared with traditional hard disks, *the CD-ROM has three advantages*:

1. Large data/information storage capability.
2. The optical disk together with information stored on it can be mass replicated inexpensively, unlike a magnetic disk. The database on a magnetic disk has to be reproduced by copying data from one disk to second disk, using two disk drives.
3. The optical disk is removable, allowing the disk itself to be used for archival storage. Most magnetic disks are non-removable. The information on non-removable magnetic disks must first be copied on tape before the disk drive / disk can be used to store new information.

The disadvantages of CD- ROM are as follows:

1. It is read-only and cannot be updated.
2. It has an access time much longer than that of a magnetic disk drive (as it employs CLV), as much as half a second.

Compact Disk Recordable (CD-R)

To accommodate applications in which only one or a small number of copies of a set data is needed, the write-once read-many CD, known as the CD Recordable (CD-R), has been developed. For CD-R a disk is prepared in such a way that it can be subsequently written once with a laser beam of modest intensity. Thus, with a somewhat more expensive disk controller than for CD-ROM, the customer can write once as well as read the disk.

The CD-R medium is similar to but not identical to that of a CD or CD-ROM. For CDs and CD-ROMs, information is recorded by the pitting of the surface of the medium, which changes reflectivity. For a CD-R, the medium includes a dye layer. The resulting disk can be read on a CD-R drive or a CD-ROM drive.

The CD-R optical disk is attractive for archival storage of documents and files. It provides a permanent record of large volumes of user data.

Compact Disk Rewritable (CD-RW)

The CD-RW optical disk can be repeatedly written and overwritten, as with a magnetic disk. Although a number of approaches have been tried, the only pure optical approach that has proved attractive is called phase change. The phase change disk uses a material that has two significantly different reflectivities 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 primary disadvantage of phase change optical disks is that the material eventually and permanently loses its desirable properties. Current materials can be used for between 500,000 and 1,000,000 erase cycles.

The CDRW has the obvious advantage over CD-ROM and CD-R that it can be rewritten and thus used as a true secondary storage. As such, it competes with magnetic disk. A key advantage of the optical disk is that the engineering tolerances for optical disks are much less severe than for high-capacity magnetic disks. Thus, they exhibit higher reliability and longer life.

Digital Versatile Disk (DVD)

With the capacious digital versatile disk (DVD), the electronics industry has at last found an acceptable replacement for the videotape used in videocassette recorders (VCRs) and, more important for this discussion, replace the CD-ROM in personal computers and servers. The DVD has taken video into the digital age. It delivers movies with impressive picture quality, and it can be randomly accessed like audio CDs, which DVD machines can also play. Vast volumes of data can be crammed onto the disk, several times as much as a CD-ROM. With DVD's huge storage capacity and vivid quality, PC games will become more realistic and educational software will incorporate more video.

1.4.3 CCDs, Bubble Memories

Charge-coupled Devices (CCDs)

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

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

Magnetic Bubble Memories

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

In these devices deposition of a soft magnetic material called Perm alloy is made as a predetermined path, thus making a track. Bubbles are forced to move continuously in a fixed direction on these tracks. In these memories the presence of a bubble represents a 1 while absence represents a 0 state. For writing data into a cell, a bubble generator to introduce a bubble or a bubble annihilator to remove a bubble, are required. A bubble detector performs the read operation. Magnetic bubble memories having capacities of 1M or more bits per chip have been manufactured. The cost and performance of these memories fall between semi-conductor RAMs and magnetic disks.

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

Check Your Progress 1

1. State True or False:

- a) Bubble memories are non-volatile.

T/F

- b) The disadvantage of DRAM over static RAM is the need to refresh the capacitor charge every few milliseconds. T/F
- c) Flash memory is a volatile RAM. T/F

2. **Fill in the blanks:**

- a) The EPROM is _____ erasable and _____ programmable.
- b) _____ memory requires a rechargeable cycle in order to retain its information.
- c) Memory elements employed specifically in computer memories are generally _____ circuits.
3. Differentiate among RAM, ROM, PROM and EPROM.
.....
4. What is a flash memory? Give a few of its typical uses.
.....
5. A memory has a capacity of $4K \times 8$
 (a) How many data input and data output lines does it have?
 (b) How many address lines does it have?
 (c) What is the capacity in bytes?

6. Describe the internal architecture of a DRAM that stores 4K bytes chip size and uses a square register array. How many address lines will be needed? Suppose the same configuration exists for an old RAM, then how many address lines will be needed?

7. How many RAM chips of size $256K \times 1$ bit are required to build 1M Byte memory?

1.5 RAID AND ITS LEVELS

Researchers are constantly trying to improve the secondary storage media by increasing their capacity, performance, and reliability. However, any physical media has a limit to its capacity and performance. When it gets harder to make further

improvements in the physical storage media and its drive, researchers normally look for other methods for improvement. Mass storage devices are a result of such improvements, which researchers have resorted to for larger capacity secondary devices. The idea is to use multiple units of the storage media being used as a single secondary storage device. One such attempt in the direction of improving disk performance is to have multiple components in parallel. Some basic questions for such systems are:

How are the disks organised?

May be as an array of disks.

Can separate I/O requests be handled by such a system in parallel?

Yes, but only if the disk accesses are from separate disks.

Can a single I/O request be handled in parallel?

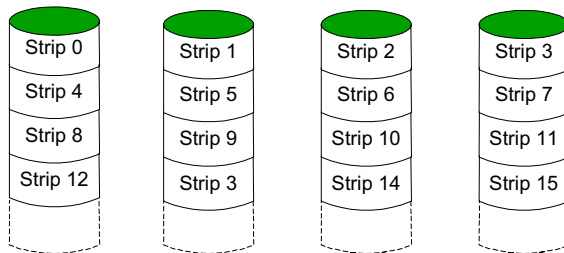
Yes, but the data block requested should be available on separate disks.

Can this array of disks be used for increasing reliability?

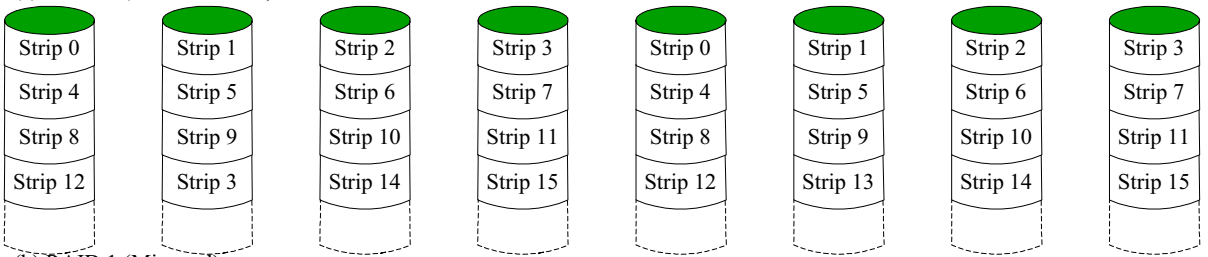
Yes, but for that redundancy of data is essential.

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

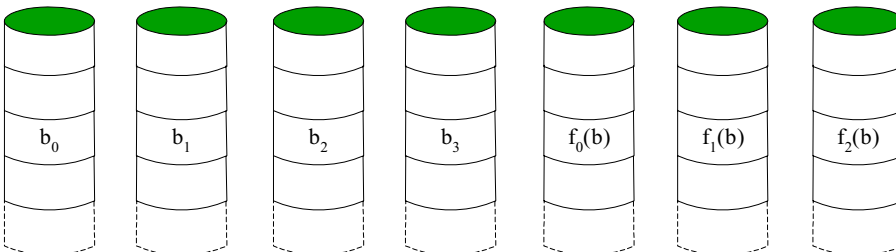
- Operating system considers the physical disks as a single logical drive.
- Data is distributed across the physical disks.
- In case of failure of a disk, the redundant information (for example, the parity bit) kept on redundant disks is used to recover the data.



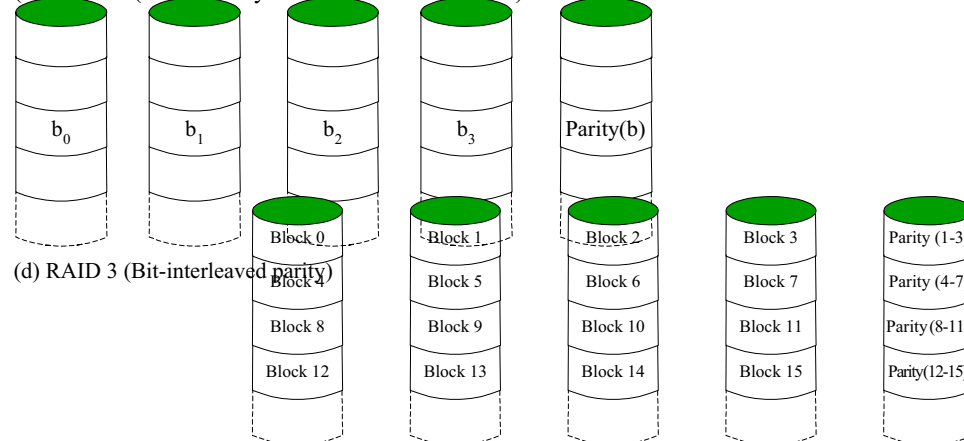
(a) RAID 0 (Non-redundant)



(b) RAID 1 (Mirrored)



(c) RAID 2 (Redundancy through Hamming Code)



(d) RAID 3 (Bit-interleaved parity)

(e) RAID 4 (Block level Parity)

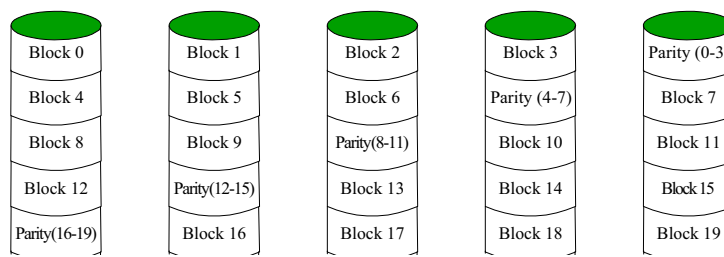


Figure 13: The RAID levels

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

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

The basic strategy used in RAID is to replace the large capacity disk drive with multiple smaller capacity disks. The data on these disks is distributed to allow simultaneous access, thus improving the overall input/output performance. It also allows an easy way of incrementing the capacity of the disk. Please note that one of the main features of the design is to compensate for the increase in probability of failure of multiple disks through the use of parity information. The seven levels of RAID are given in Figure 13 shown above. Please note that levels 2 and 4 are not commercially offered.

In RAID technologies have two important performance considerations:

- The Data Transfer Rate
- Input/Output Request rate

High data transfer rate is dependent on:

- Path between individual disks and Memory.
- Fulfilment of an I/O request by multiple disks of disk array. Thus, increasing the transfer rate.

Input/Output request rate is defined as the time taken to fulfil an I/O request. This is an important consideration while dealing with transaction processing systems like Railway reservation system. A high I/O request rate can be obtained through distributing data requests on multiple disks.

1.6 THE CONCEPTS OF HIGH SPEED MEMORIES

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 strips, maybe a block, a sector or other unit. b) Non-redundant.	Large strips: Excellent	Small strip: Excellent	Applications requiring high performance for non-critical data
1	Mirroring	a) Every disk in the array has a mirror disk that contains the same data. b) Recovery from a failure is simple. When a drive fails, the data may still be recovered from the second drive.	Good / fair	Fair /fair	System drives; critical files
2	Parallel Access	a) All member disks participate in the execution of every I/O request by synchronising the spindles of all the disks to the same position at a time. b) The strips are very small, often a single byte or word. c) Redundancy via hamming code which is able to correct single-bit errors and detect double-bit errors.	Poor	Excellent	Commercially not useful.
3	Parallel Access	a) Employs parallel access as that of level 2, with small data strips. b) A simple parity bit is computed for the set of data instead of an error-correcting code in case a disk fails.	Poor	Excellent	Large I/O request size application, such as imaging CAD
4	Independent access	a) Each member disk operates independently, thus enabling fulfilment of separate input/output requests in parallel. b) Data strip is large and bit by bit parity strip is created for bits of strips of each disk. c) Parity strip is stored on a separate disk.	Excellent/ fair	Fair / poor	Commercially not useful.
5	Independent access	a) Employs independent access as that of level 4 and distributes the parity strips across all disks. b) The distribution of parity strips across all drives avoids the potential input/output bottleneck found in level 4.	Excellent / fair	Fair / poor	High request rate read intensive, data lookup
6	Independent access	a) Also called the P+Q redundancy scheme, is much like level 5, but stores extra redundant information to guard against multiple disk failures. b) P and Q are two different data check algorithms. One of the two is the exclusive-or calculation used in level 4 and 5. The other one is an independent data check algorithm.	Excellent/ poor	Fair / poor	Application requiring extremely high availability

Why are high-speed memories needed? Is the main memory not a high-speed memory? The answer to the second question is definitely “No”, but why so? For this, we have to go to the fundamentals of semiconductor technology, which is beyond the scope of the Unit. Then if the memories are slower, then how slow are they? On an average it has been found that the operating speed of main memories lack by a factor

of 5 to 10 than that of the speed of processors (such as CPU or Input / Output Processors).

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

The question is what can be done to increase this processor-memory interface bandwidth? There are four possible answers to the question. These are:

- a) Decrease the memory access time; use a faster but expensive technology for main memory.
- b) Access more words in a single memory access cycle. That is, instead of accessing one word from the memory in a memory access cycle, access more words.
- c) Insert a high-speed memory termed as Cache between the main memory and processor.
- d) Use associative addressing in place of random access.

Hardware researchers are taking care of the first point. Let us discuss some high speed memories that are in existence at present.

1.6.1 Cache Memory

Cache memory is an extremely fast, small memory between CPU and main memory whose access time is closer to the processing speed of the CPU. It acts as a high-speed buffer between CPU and main memory and is used to temporarily store currently active data and instructions during processing. Since the cache memory is faster than main memory, the processing speed is increased by making data and instructions needed in present processing available in the cache.

The obvious question that arises is how the system can know in advance which data and instruction are needed in present processing so as to make it available beforehand in the cache. The answer to this question comes from a principle known as ***locality of reference***. According to this principle, during the course of execution of most programs, memory references by the processor, for both instructions and data, tend to cluster. That is, if an instruction is executed, there is a likelihood of the nearby instruction being executed soon. Locality of reference is true not only for reference to program instruction but also for references to data. As shown in Figure 14, the cache memory acts as a small, fast-speed buffer between the processor and main memory.

Figure 14: Cache Memory Operation

It contains a copy of a portion of main memory contents. When a program is running and the CPU attempts to read a word of memory (instruction or data), a check is made to determine if the word is in the cache. If so, the word is delivered to the CPU from the cache. If not, a block of main memory, consisting of some fixed number of words including the requested word, is read into the cache and then the requested word is delivered to the CPU. Because of the feature of locality of reference, when a block of memory word is fetched into the cache to satisfy a single memory reference, it is likely that there will soon be references to other words in that block. That is, the next time the CPU attempts to read a word, it is very likely that it finds it in the cache and saves the time needed to read the word from main memory.

Many computer systems are designed to have two separate cache memories called *instruction cache* and *data cache*. The instruction cache is used for storing program instruction and the data cache is used for storing data. This allows faster identification of availability of accessed word in the cache memory and helps in further improving the processor speed. Many computer systems are also designed to have multiple levels of caches (such as level one and level two caches, often referred to as **L1 and L2 caches**). L1 cache is smaller than L2 cache and is used to store more frequently accessed instruction/data as compared to those in the L2 cache.

The use of cache memory requires several design issues to be addressed. Some key design issues are briefly summarised below:

1. **Cache Size:** Cache memory is very expensive as compared to the main memory and hence its size is normally kept very small. It has been found through statistical studies that reasonably small caches can have a significant impact on processor performance. As a typical example of cache size, a system having 1 GB of main memory may have about 1 MB of cache memory. Many of today's personal computers have 64KB, 128KB, 256KB, 512KB, or 1 MB of cache memory.
2. **Block Size:** Block size refers to the unit of data (few memory words) exchanged between cache and main memory. As the block size increases from very small to larger size, the hit ratio (fraction of times that referenced instruction/data is found in cache) will at first increase because of the principle of locality since more and more useful words are brought into the cache. However, the hit ratio will begin to decrease as the block size further increases because the probability of using the newly fetched words becomes less than the probability of reusing the words that must be moved out of the cache to make room for the new block. Based on this fact, the block size is suitably chosen to maximise the hit ratio.
3. **Replacement Policy:** When a new block is to be fetched into the cache, another may have to be replaced to make room for the new block. The replacement policy decides which block to replace in such a situation. Obviously, it will be best to replace a block that is least likely to be needed again in the near future.
4. **Write Policy:** If the contents of a block in the cache are altered, then it is necessary to write it back to main memory before replacing it. The write policy decides when the altered words of a block are written back to main memory. At

one extreme, an updated word of a block is written to the main memory as soon as such updates occur in the block. At the other extreme, all updated words of the block are written to the main memory only when the block is replaced from the cache. The latter policy minimises overheads of memory write operations but temporarily leaves main memory in an inconsistent (obsolete) state.

1.6.2 Cache Organisation

Cache memories are found in almost all latest computers. They are very useful for increasing the speed of access of information from memory. Let us look into their organisation in more detail in this section.

The fundamental idea of cache organisation is that by keeping the most frequently accessed instructions and data in the fast cache memory; hence the average memory access time will approach the access time of the cache.

The basic operation of the cache is as follows. When the CPU needs to access memory, the cache is examined. If the word addressed by the CPU is not found in the cache, the main memory is accessed to read the word. A block of words is then transferred from main memory to cache memory.

The performance of cache memory is frequently measured in terms of a quantity called **hit ratio**. When the CPU refers to the main memory and finds the word in cache, it is said to produce a **hit**. If the word is not found in cache, it is in the main memory and it counts as a **miss**. The ratio of the number of hits divided by the total CPU references to memory is the hit ratio.

The average memory access time of a computer system can be improved considerably by use of a cache. For example, if memory read cycle takes 100 ns and a cache read cycle takes 20 ns, then for four continuous references, the first one brings the main memory contents to cache and the next three from cache.

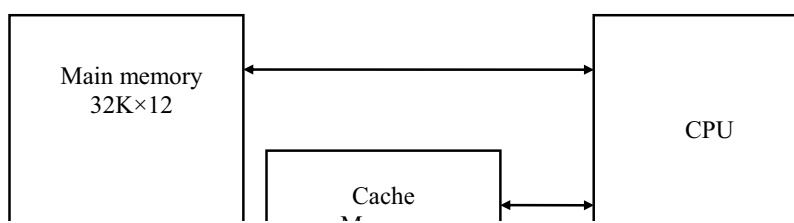
$$\begin{aligned} \text{The time taken with cache} &= (100+20) & + & & (20 \times 3) \\ & \text{(For the first read operation)} & & & \text{(For the last three read operations)} \\ & = 120 & + & & 60 \\ & = 180 \text{ ns} \\ \text{Time taken without cache} &= 100 \times 4 = 400 \text{ ns} \end{aligned}$$

Thus, the closer are the reference, the better is the performance of cache.

The basic characteristic of cache memory is its fast access time. Therefore, very little or no time must be wasted when searching for words in the cache. The transformation of data from main memory to cache memory is referred to as a mapping process. The mapping procedure for the cache organization is of three types:

1. Associative mapping
2. Direct mapping
3. Set-associative mapping

Let us consider an example of a memory organization as shown in Figure 15 in which the main memory can store 32K words of 12 bits each and the cache is capable of storing 512 blocks (each block in the present example is equal to 24 bits, which is equal to two main memory words) at any time.



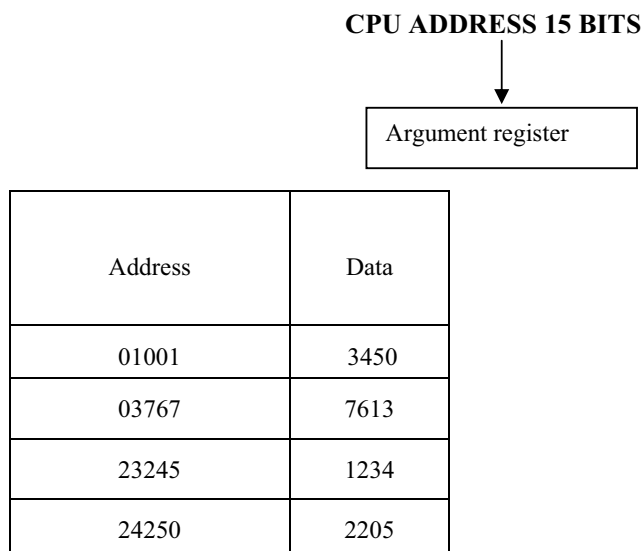
Size of main memory address (Given word size of 12 bits) = 32 K words = 2^{15} words
 \Rightarrow 15 bits are needed for address
Block Size of Cache = 2 Main Memory Words

Figure 15: Cache Memory

For every word stored in cache, there is a duplicate copy in the main memory. The CPU communicates with both memories. It first sends a 15 bits ($32K = 2^5 \times 2^{10} = 2^{15}$) address to cache. If there is a hit, the CPU uses the relevant 12 bits data from 24 bit cache data. If there is a miss, the CPU reads the block containing the relevant word from the main memory. So the key here is that a cache must store the address and data portions of the main memory to ascertain whether the given information is available in the cache or not. However, let us assume the block size as 1 memory word for the following discussions.

Associative Mapping

The most flexible and fastest cache organization uses an associative memory which is shown in Figure 16. The associative memory stores both the address and data of the memory word. This permits any location in cache to store any word from the main memory. The address value of 15 bits is shown as a five-digit octal number and its corresponding 12 bits word is shown as a five digit octal number. A CPU address of 15 bits is placed in the argument register and the associative memory is searched for a matching address. If the address is found, the corresponding 12 bits data is read and sent to the CPU. If no matches are found, the main memory is accessed for the word. The address-data pair is then transferred to the associative cache memory. This address checking is done simultaneously for the complete cache in an associative way.



(All numbers are in octal)

Figure 16: Associative Mapping Cache

Direct Mapping

In the general case, there are 2^k words in cache memory and 2^n words in the main memory. The n -bits memory address is divided into two fields: k bits for the index field and $(n - k)$ bits for the tag field.

The direct mapping cache organization uses the n -bit address to access the main memory and k -bit index to access the cache. The internal organization of the words in the cache memory is as shown in Figure 17. Each word in cache consists of the data word and its associated tag. When a new word is first brought into the cache, the tag bits are stored alongside the data bits. When the CPU generates a memory request, the index field is used for the address to access the cache.

Figure 17: Addressing Relationship for Main Memory and Cache

The tag field of the CPU address is compared with the tag in the word read from the cache. If the two tags match, there is a hit and the desired data word is in cache. If there is no match, there is a miss and the required word is read from the main memory.

Let us consider a numerical example shown in Figure 18. The word at address zero is at present stored in the cache (index = 000, tag = 00, data = 1456). Suppose that the CPU wants to access the word at address 02000. The index address is 000, so it is used to access the cache. The two tags are then compared. The cache tag is 00 but the address tag is 02, which does not produce a match. Therefore, the main memory is accessed and the data word 4254 is transferred to the CPU. The cache word at index address 000 is then replaced with a tag of 02 and data of 4254.

Figure 18: Direct Mapping Cache Organisation

Set-Associative Mapping

A third type of cache organization called set-associative mapping is an improvement on the direct mapping organization in that each word of cache can store two or more words of memory under the same index address. Each data word is stored together with its tag and the number of tag data items in one word of cache is said to form a set.

Let us consider an example of a set-associative cache organization for a set size of two as shown in the Figure 19. Each index address refers to two data words and their associated tags. Each tag requires six bits and each data word has 12 bits, so the word length of cache is $2(6+12) = 36$ bits. An index address of nine bits can accommodate 512 words. Thus, the size of cache memory is 512×36 . In general, a Set-Associative cache of set size K will accommodate K-words of main memory in each word of cache.

Index	Tag	Data	Tag	Data
000	01	3450	02	5670
777	02	6710	00	2340

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

Figure 19: Two-Way Set-Associative Mapping Cache

- The contents of cache and main memory can be altered by more than one device. For example, CPU can write to caches and input/output module can directly write to the main memory. This can result in inconsistencies in the values of the cache and main memory.
- In the case of multiple CPUs with different cache, a word altered in one cache automatically invalidate the word in the other cache.

The suggested techniques for writing in system with caches are:

- (a) **Write through:** Write the data in cache as well as main memory. The other CPUs - Cache combination has to watch with traffic to the main memory and make suitable amendment in the contents of cache. The disadvantage of this technique is that a bottleneck is created due to large number of accesses to the main memory by various CPUs.
- (b) **Write block:** In this method updates are made only in the cache, setting a bit called Update bit. Only those blocks whose update bit is set is replaced in the main memory. But here all the accesses to the main memory, whether from other CPUs or input/output modules, need to be from the cache resulting in complex circuitry.
- (c) **Instruction Cache:** An instruction cache is one which is employed for accessing only the instructions and nothing else. The advantage of such a cache is that as the instructions do not change we need not write the instruction cache back to memory, unlike data storage cache.

1.6.3 Memory Interleaving

In this method, the main memory is divided into 'n' equal-size modules and the CPU has separate Memory Address Register and Memory Base register for each memory module. In addition, the CPU has 'n' instruction register and a memory access system. When a program is loaded into the main memory, its successive instructions are stored in successive memory modules. For example if $n=4$ and the four memory modules are M_1, M_2, M_3 , and M_4 then 1st instruction will be stored in M_1 , 2nd in M_2 , 3rd in M_3 , 4th in M_4 , 5th in M_1 , 6th in M_2 and so on. Now during the execution of the program, when the processor issues a memory fetch command, the memory access system creates n consecutive memory addresses and places them in the Memory Address Register in the right order. A memory read command reads all the 'n' memory modules simultaneously, retrieves the 'n' consecutive instructions, and loads them into the 'n' instruction registers. Thus each fetch for a new instruction results in the loading of 'n' consecutive instructions in the 'n' instruction registers of the CPU.

Since the instructions are normally executed in the sequence in which they are written, the availability of N successive instructions in the CPU avoids memory access after each instruction execution, and the total execution time speeds up. Obviously, the fetch successive instructions are not useful when a branch instruction is encountered during the course of execution. This is because they require the new set of 'n' successive instructions, overwriting the previously stored instructions, which were loaded, but some of which were not executed. The method is quite effective in minimising the memory-processor speed mismatch because branch instructions do not occur frequently in a program.

Figure 20 illustrates the memory interleaving architecture. The Figure shows a 4- way ($n=4$) interleaved memory system.

Figure 20: A 4-way Interleaved Memory

1.6.4 Associative Memory

The time required to find an item stored in memory can be reduced considerably if stored data can be identified for access by the contents of the data itself rather than by an address. A memory unit accessed by content of the data is called an **associative memory** or **content addressable memory (CAM)**. This type of memory is accessed simultaneously and in parallel on the basis of data content rather than by specific address or location. When a word is written in an associative memory, no address is given. The memory is capable of finding an empty unused location to store the word. When a word is to be read from an associative memory, the content of the word, or part of the word, is specified. The memory locates all words, which match the specified content, and marks them for reading.

Because of its organization, the associative memory is uniquely suited to do parallel searches by data association. Moreover, searches can be done on an entire word or on a specific field within a word. An associative memory is more expensive than a random access memory because each cell must have storage capability as well as logic circuits for matching its content with an external argument. For this reason associative memories are used in applications where the search time is very critical and must be very short.

Hardware Organization

The block diagram of an associative memory is shown in Figure 21. It consists of a memory array and logic for m words with n bits per word. The argument register A and key register K each have n bits, one for each bit of a word. The match register M has m bits, one for each memory word. Each word in memory is compared in parallel with the content of the argument register; the words that match the bits of the argument register set a corresponding bit in the match register. After the matching process, those bits in the match register that have been set indicate the fact that their corresponding words have been matched. Reading is accomplished by a sequential access to memory for those words whose corresponding bits in the match register have been set.

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 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.

Figure 21: Associative Memory – Block Diagram

To illustrate with a numerical example, suppose that the argument register A and the key register K have the bit configuration shown below. Only the three leftmost bits of A are compared with memory words because K has 1's on these positions

A	101 111100	
K	111 000000	
Word 1	100 111100	no match
Word 2	101 000001	match

Word 2 matches the unmasked argument field because the three leftmost bits of the argument and the word are equal.

Check Your Progress 2

- What is a RAID? What are the techniques used by RAID for enhancing reliability?
.....
.....
.....
- State True or False:**
 - Interleaved memories are best suited for small loops and large sequential code. T/F
 - The principle of locality of reference justifies the use of cache memory. T/F
 - High-speed memories are needed to bridge the gap of speed between I/O device and memory. T/F
 - Write policy is not needed for instruction cache. T/F
 - A replacement algorithm is needed only for associative and set associative mapping of cache. T/F
- How can the Cache memory and interleaved memory mechanisms be used to improve the overall processing speed of a Computer system?

-
-
-
4. Assume a Computer having 64 word RAM (assume 1 word = 16 bits) and cache memory of 8 blocks (block size = 32 bits). Where can we find Main Memory Location 25 in cache if (a) Associative Mapping (b) Direct mapping and (c) 2 way set associative (2 blocks per set) mapping is used.

-
-
-
5. How is a given memory word address (memory location 25 as above) located to Cache for the example above for (a) Associative (b) Direct and (c) 2 way set associative mapping.

-
-
-
6. A computer system has a 4K-word cache organised in block set associative manner with 4 blocks per set, 64 words per block. What is the number of bits in the Index and Block Offset fields of the main memory address formula?

.....

.....

.....

1.7 VIRTUAL MEMORY

In a memory hierarchy system, programs and data are first stored in auxiliary or secondary memory. The program and its related data are brought into the main memory for execution. What if the size of Memory required for the Program is more than the size of memory? Virtual memory is a concept used in some large computer systems that permit the user to construct programs as though a large memory space were available, equal to the totality of secondary memory. Each address generated by the CPU goes through an address mapping from the so-called virtual address to a physical address in the main memory. Virtual memory is used to give programmers the illusion that they have a very large memory at their disposal, even though the computer actually has a relatively small main memory. A Virtual memory system provides a mechanism for translating program-generated addresses into correct main memory locations. This is done dynamically, while programs are being executed in the CPU. The translation or mapping is handled automatically by the hardware by means of a mapping table.

Address Space and Memory Space

An address used by a programmer will be called a virtual address, and the set of such addresses the address space. An address in the main memory is called a physical address. The set of such locations is called the memory space. Thus, the address space is the set of addresses generated by programs as they reference instructions and data; the memory space consists of the actual main memory locations directly addressable for processing.

Consider a computer with a main-memory capacity of 64K words ($K=1024$). 16-bits are needed to specify a physical address in memory since $64K = 2^{16}$. Suppose that the computer has auxiliary memory for storing information equivalent to the capacity of 16 main memories. Let us denote the address space by N and the memory space by M , we then have for this example $N = 16 \times 64 K = 1024K$ and $M = 64K$.

In a multiprogramming computer system, programs and data are transferred to and from auxiliary memory and main memory based on demands imposed by the CPU. Suppose that program 1 is currently being executed in the CPU. Program 1 and a portion of its associated data are moved from secondary memory into the main memory as shown in Figure 22. Portions of programs and data need not be in contiguous locations in memory since information is being moved in and out, and empty spaces may be available in scattered locations in memory.

Figure 22: Address and Memory Space in Virtual Memory

In our example, the address field of an instruction code will consist of 20 bits but physical memory addresses must be specified with only 16-bits. Thus CPU will reference instructions and data with a 20 bits address, but the information at this address must be taken from physical memory because access to auxiliary storage for individual words will be prohibitively long. A mapping table is then needed, as shown in Figure 23, to map a virtual address of 20 bits to a physical address of 16 bits. The mapping is a dynamic operation, which means that every address is translated immediately as a word is referenced by CPU.

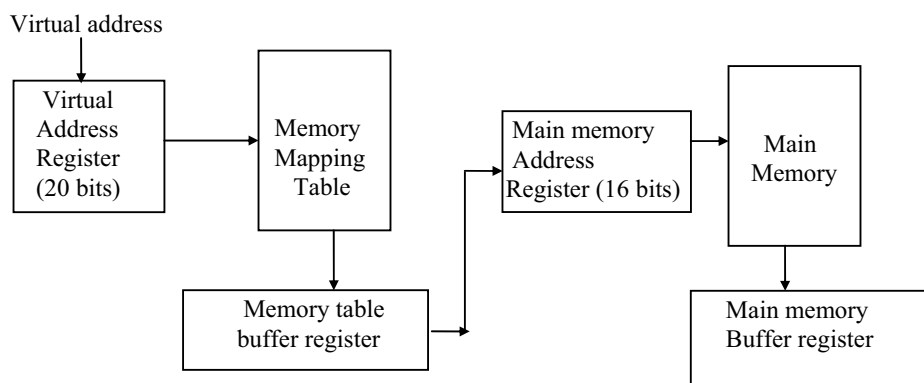


Figure 23: Memory table for mapping a virtual table

1.8 THE MEMORY SYSTEM OF MICROCOMPUTER

Till now we have discussed various memory components. But, how is the memory organised in the physical computer? Let us discuss various kinds of memory technologies used in personal computer.

1.8.1 SIMM (Single In-line Memory Modules), DIMM (Dual In line Memory Modules), etc., Memory Chips

From the early days of semiconductor memory until the early 1990s, memory was manufactured, brought and installed as a single chip. Chip density went from 1K bits to 1M bits and beyond, but each chip was a separate unit. Early PCs often had empty sockets into which additional memory chips could be plugged, if and when the purchaser needed them. At present, a different arrangement is often used called SIMM or DIMM.

A group of chips, typically 8 to 16, is mounted on a tiny printed circuit board and sold as a unit. This unit is called a SIMM or DIMM depending on whether it has a row of connectors on one side or both sides of the board.

A typical SIMM configuration might have 8 chips with 32 megabits (4MB) each on the SIMM. The entire module then holds 32MB. Many computers have room for four modules, giving a total capacity of 128MB when using 32MB SIMMs. The first SIMMs had 30 connectors and delivered 8 bits at a time. The other connectors were addressing and control. A later SIMM had 72 connectors and delivered 32 bits at a time. For a machine like Pentium, which expected 64-bits at once, 72-connectors SIMMs were paired, each one delivering half the bits needed.

A DIMM is capable of delivering 64 data bits at once. Typical DIMM capacities are 64MB and up. Each DIMM has 84 gold patted connectors on each side for a total of 168 connectors. SIMM and DIMM are shown in Figure 24 (a) and (b) respectively. How they are put on a motherboard is shown in Figure 24 (c).

SIMM

DIMM

(C) DIMM on Motherboard**Figure 24: SIMM & DIMM****1.8.2 SDRAM, RDRAM, Cache RAM Types of Memory**

The basic building block of the main memory remains the DRAM chip, as it has for decades. Until recently, there had been no significant changes in DRAM architecture since the early 1970s. The traditional DRAM chip is constrained both by its internal architecture and by its interface to the processor's memory bus. The two schemes that currently dominate the market are SDRAM and RDRAM. A third one, that is Cache RAM, is also very popular.

SDRAM (Synchronous DRAM)

One of the most widely used forms of DRAM is the synchronous DRAM (SDRAM). Unlike the traditional DRAM, which is asynchronous, the SDRAM exchanges data with the processor synchronized to an external clock signal and running at the full speed of the processor /memory bus without imposing wait states.

In a typical DRAM, the processor presents addresses and control levels to the memory, indicating that a set of data at a particular location in memory should be either read from or written into the DRAM. After a delay, the access time, the DRAM either writes or reads the data during the access-time delay. The DRAM performs various internal functions, such as activating the high capacitance of the row and column lines, sensing the data and routing the data out through the output buffers. The processor must simply wait through this delay, slowing system performance.

With synchronous access, the DRAM moves data in and out under control of the system clock. The processor or other master issues the instruction and address information, which is latched on to by the DRAM. The DRAM then responds after a set number of clock cycles. Meanwhile, the master can safely do other tasks while the SDRAM is processing the request.

The SDRAM employs a burst mode to eliminate the address setup time. In burst mode, a series of data bits can be clocked out rapidly after the first bit has been accessed. The mode is useful when all the bits to be accessed are in sequence and in the same row of the array as the initial access. In addition, the SDRAM has a multiple-bank internal architecture that improves opportunities for on-chip parallelism.

The mode register and associated control logic is another key feature differentiating SDRAMs from conventional DRAMs. It provides a mechanism to customize the SDRAM to suit specific system needs. The mode register specifies the burst length, which is the number of separate units of data synchronously fed onto the bus. The register also allows the programmer to adjust the latency between receipt of a read request and the beginning of data transfer.

The SDRAM performs best when it is transferring large blocks of data serially, such as for applications like word processing, spreadsheets, and multimedia.

RDRAM (Rambus DRAM)

RDRAM, developed by Rambus, has been adopted by Intel for its Pentium and Itanium processors. It has become the main competitor to SDRAM. RDRAM chips are vertical packages, with all pins on one side. The chip exchanges data with the processor over 28 wires no more than 12 centimeters long. The bus address up to 320 RDRAM chips and is rated at 1.6 GBps.

The special RDRAM bus delivers address and control information using an asynchronous block-oriented protocol. After an initial 480 ns access time, this produces the 1.6 GBps data rate. The speed of RDRAM is due to its high speed Bus. Rather than being controlled by the explicit RAS CAS R/W, and CE signals used in conventional DRAMs an RDAR gets a memory request over the high-speed bus. This request contains the desired address, the type of operation and the number of bytes in the operation.

CDRAM (Cache DRAM)

Cache DRAM (CDRAM), developed by Mitsubishi, integrates a small SRAM cache (16Kb) onto a generic DRAM chip. The SRAM on the CDRAM can be used in two ways. First, it can be used as true cache consisting of a number of 64-bit line. The cache mode of the CDRAM is effective for ordinary random access to memory.

The SRAM on the CDRAM can also be used as a buffer to support the serial access of a block of data. For example, to refresh a bit-mapped screen, the CDRAM can prefetch the data from the DRAM into the SRAM buffer. Subsequent accesses to chip result in accesses solely to the SRAM.

Check Your Progress 3

1. Difference between
 - a) SDRAM and RDRAM
 - b) SIMM and DIMM

.....

.....

.....

.....
2. A Computer supports a virtual memory address space of 1Giga Words, but a physical Memory size of 64 Mega Words. How many bits are needed to specify an instruction address for this machine?

.....

.....

.....

.....

1.9 SUMMARY

In this unit, we have discussed the details of the memory system of the computer. First we discussed the concept and the need of the memory hierarchy. Memory hierarchy is essential in computers as it provides an optimised low-cost memory system. The unit also covers details on the basic characteristics of RAMs and different kinds of ROMs. These details include the logic diagrams of RAMs and ROMs giving basic functioning through various control signals. We have also discussed the latest secondary storage technologies such as CD-ROM, DVD-ROM, CD-R, CD-RW etc. giving details about their data formats and access mechanisms.

The importance of high-speed memories such as cache memory, interleaved memory and associative memories are also described in detail. The high-speed memory, although small, provides a very good overall speed of the system due to locality of reference. There are several other concepts such as the memory system of the microcomputer which consists of different types of chips such as SIMM, DIMM and different types of memory such as SDRAM, RDRAM also defined in easy way. The unit also contains details on Virtual Memory. For more details on the memory system you can go through further units.

1.10 SOLUTIONS / ANSWERS

Check Your Progress 1

1. a) True
b) True
c) False
2. a) Ultraviolet light, electrically
b) Dynamic
c) Sequential
- 3.

	<u>RAM</u>	<u>ROM and all types of ROM's</u>
a)	Volatile Memory	Non – volatile
b)	Faster access time	Slower than RAM
c)	Higher cost per bit storage	Lower than RAM
d)	Random access	Sequential access
e)	Less storage capacity	Higher storage capacity

4. A type of EPROM called EEPROM is known as flash memory used in many I/O and storage devices. It is commonly used memory in embedded systems.
5. (a) Eight, since the word size is 8.
(b) $4K = 4 \times 1024 = 4096$ words. Hence, there are 4096 memory addresses. Since $4096 = 2^{12}$ it requires 12 bits address code to specify one of 4096 addresses.
(c) The memory has a capacity of 4096 bytes.
6. 4K bytes is actually $4 \times 1024 = 4096$ bytes and the DRAM holds 4096 eight bit words. Each word can be thought of as being stored in an 8 bit register and there are 4096 registers connected to a common data bus internal to the chip. Since $4096 = (64)^2$, the registers are arranged in a 64×64 array, that is there are $64=2^6$ rows and $64=2^6$ columns. This requires a 6×64 decoder to decode six- address inputs for the row select and a second 6×64 decoder to decode six other address

inputs for the column select. Using the structure as shown in Figure 3 (b), it requires only 6 bit address input.

While in the case of an old RAM, the chip requires 12 address lines (Please refer to Figure 2(b)), since $4096 = 2^{12}$ and there are 4096 different addresses.

$$\begin{aligned} 7. \quad 1 \text{ M Bytes} &= 2^{20} \cdot 2^3 \text{ bits} = 2^{23} \\ 256\text{K} \times 1 \text{ bit} &= 2^8 \cdot 2^{10} \text{ bits} = 2^{18} \end{aligned}$$

$$\text{Hence, total number of RAM chips of size } (256\text{K} \times 1) = \frac{2^{23}}{2^{18}} = 2^5 = 32$$

Check Your Progress 2

- 1) A disk array known as RAID systems is a mass storage device that uses a set of hard disks and hard disk drives with a controller mounted in a single box. All the disks of a disk array form a single large storage unit. The RAID systems were developed to provide large secondary storage capacity with enhanced performance and enhanced reliability. The performance is based upon the data transfer rate, which is very high rather than taking an individual disk. The reliability can be achieved by two techniques that is mirroring (the system makes exact copies of files on two hard disks) and stripping (a file is partitioned into smaller parts and different parts of the files are stored on different disks).
2.
 - a) True
 - b) True
 - c) False
 - d) True
 - e) False
3. The Cache memory is a very fast, small memory placed between CPU and main memory whose access time is closer to the processing speed of the CPU. It acts as a high-speed buffer between CPU and main memory and is used to temporarily store data and instruction needed during current processing. In memory interleaving, the main memory is divided into n number of equal size modules. When a program is loaded in to the main memory, its successive instruction in also available for the CPU, thus, it avoids memory access after each instruction execution and the total time speeds up.
4. Main memory Size = 64 Words
Main Memory word size = 16 bits
Cache Memory Size = 8 Blocks
Cache Memory Block size = 32 words
 $\Rightarrow 1 \text{ Block of Cache} = 2 \text{ Words of RAM}$
 $\Rightarrow \text{Memory location address 25 is equivalent to Block address 12.}$
 $\Rightarrow \text{Total number of possible Blocks in Main Memory} = 64/2 = 32 \text{ blocks}$
 - (a) Associative Mapping: The block can be anywhere in the cache.
 - (b) Direct Mapping:
Size of Cache = 8 blocks
Location of Block 12 in Cache = $12 \text{ modulo } 8 = 4$
 - (c) 2 Way set associative mapping:
Number of blocks in a set = 2
Number of sets = $\text{Size of Cache in blocks} / \text{Number of blocks in a set}$
 $= 8 / 2 = 4$
Block 12 will be located anywhere in $(12 \text{ modulo } 4)$ set, that is set 0.

Following figure gives the details of above schemes.

Cache mapping scheme

5. The maximum size of physical memory address in the present machine
= 6 bits (as Memory have 64 words, that is, 2^6)
The format of address mapping diagram for Direct and Set-Associative Mapping:

Physical Memory Word Address		
Block Address		Block Offset
Tag	Index	Block Offset

The address mapping in Direct Mapping:

Memory Address	0	1	1	0	0	1	\Rightarrow Memory Address = 25
Block Address	0	1	1	0	0	1	\Rightarrow Block Address = 12 and . Block offset = 1
Cache Address	0	1	1	0	0	1	\Rightarrow Tag = 1; Index = 4 and . Block offset = 1

Please note that a main memory address 24 would have a block offset as 0.

Address Mapping for 2 way set associative mapping:

Memory Address	0	1	1	0	0	1	\Rightarrow Memory Address = 25
Block Address	0	1	1	0	0	1	\Rightarrow Block Address = 12 and . Block offset = 1
Cache Address	0	1	1	0	0	1	\Rightarrow Tag = 3; Index (Set . Number) = 0 and . Block offset = 1

The Tag is used here to check whether a given address is in a specified set. This cache has 2 blocks per set, thus, the name two way set associative cache. The total number of sets here is $8 / 2 = 4$.

For Associative mapping the Block address is checked directly in all location of cache memory.

6. There are 64 words in a block, therefore 4K cache has $(4 \times 1024) / 64 = 64$ blocks. Since 1 set has 4 blocks, there are 16 sets. 16 sets need 4 bit as $2^4 = 16$

representation. In a set there are 4 blocks. So, the block field needs 2 bits. Each block has 64 words. So the block offset field has 6 bits.
Index Filed is of 4 bits.
Block offset is of 6 bits.

Check Your Progress 3

1. a) SDRAM exchanges data with the processor synchronized to an external clock signal and running at the full speed of the processor/memory bus without imposing wait states.

A RDRAM module sends data to the controller synchronously to the clock to master, and the controller sends data to an RDRAM synchronously with the clock signal in the opposite direction.

b) In SIMM, a group of chips, typically 8 to 16, is mounted on a tiny printed circuit board with one side only. While in DIMM, this can be mounted on both sides of the printed circuit board.

The latest SIMMs and DIMMs are capable of delivering 64 data bits at once. Each DIMM has 84 gold patted connectors on each side for a total of 168 connectors while each SIMM 72 connectors.

2. The virtual address is $1 \text{ GB} = 2^{30}$, thus, 30 bit Virtual address, that will be translated to physical memory address of 26 bits ($64 \text{ Mega words} = 2^{26}$).

UNIT 2 THE INPUT/OUTPUT SYSTEM

Structure	Page No.
2.0 Introduction	43
2.1 Objectives	43
2.2 Input / Output Devices or External or Peripheral Devices	43
2.3 The Input Output Interface	44
2.4 The Device Controllers and its Structure	46
2.4.1 Device Controller	
2.4.2 Structure of an Input /Output Interface	
2.5 Device Drivers	48
2.6 Input Output Techniques	50
2.6.1 Programmed Input /Output	
2.6.2 Interrupt-Driven Input /Output	
2.6.3 Interrupt-Processing	
2.6.4 DMA (Direct Memory Access)	
2.7 Input Output Processors	59
2.8 External Communication Interfaces	61
2.9 Summary	62
2.10 Solutions /Answers	62

2.0 INTRODUCTION

In the previous Unit, we have discussed the memory system for a computer system that contains primary memory, secondary memory, high speed memory and their technologies; the memory system of micro-computers i.e., their chips and types of memory. Another important component in addition to discussing the memory system will be the input/output system. In this unit we will discuss Input /Output controllers, device drivers, the structure of I/O interface, the I/O techniques. We will also discuss about the Input / Output processors which were quite common in mainframe computers.

2.1 OBJECTIVES

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

- identify the structure of input/output interface;
 - identify the concepts of device drivers and device controllers;
 - describe the input/output techniques, i.e., programmed I/O, interrupt-driven I/O and direct memory access;
 - define an input/output processor;
 - describe external communication interfaces such as serial and parallel interfaces; and
 - define interrupt processing.
-

2.2 INPUT/ OUTPUT DEVICES OR EXTERNAL OR PERIPHERAL DEVICES

Before going on to discuss the input/ output sub/systems of a computer, let us discuss how a digital computer system can be implemented by a microcomputer system. A typical microcomputer system consists of a microprocessor plus memory and I/O interface. The various components that form the system are linked through buses that transfer instructions, data, addresses and control information among the components. The block diagram of a microcomputer system is shown in Figure. 1.

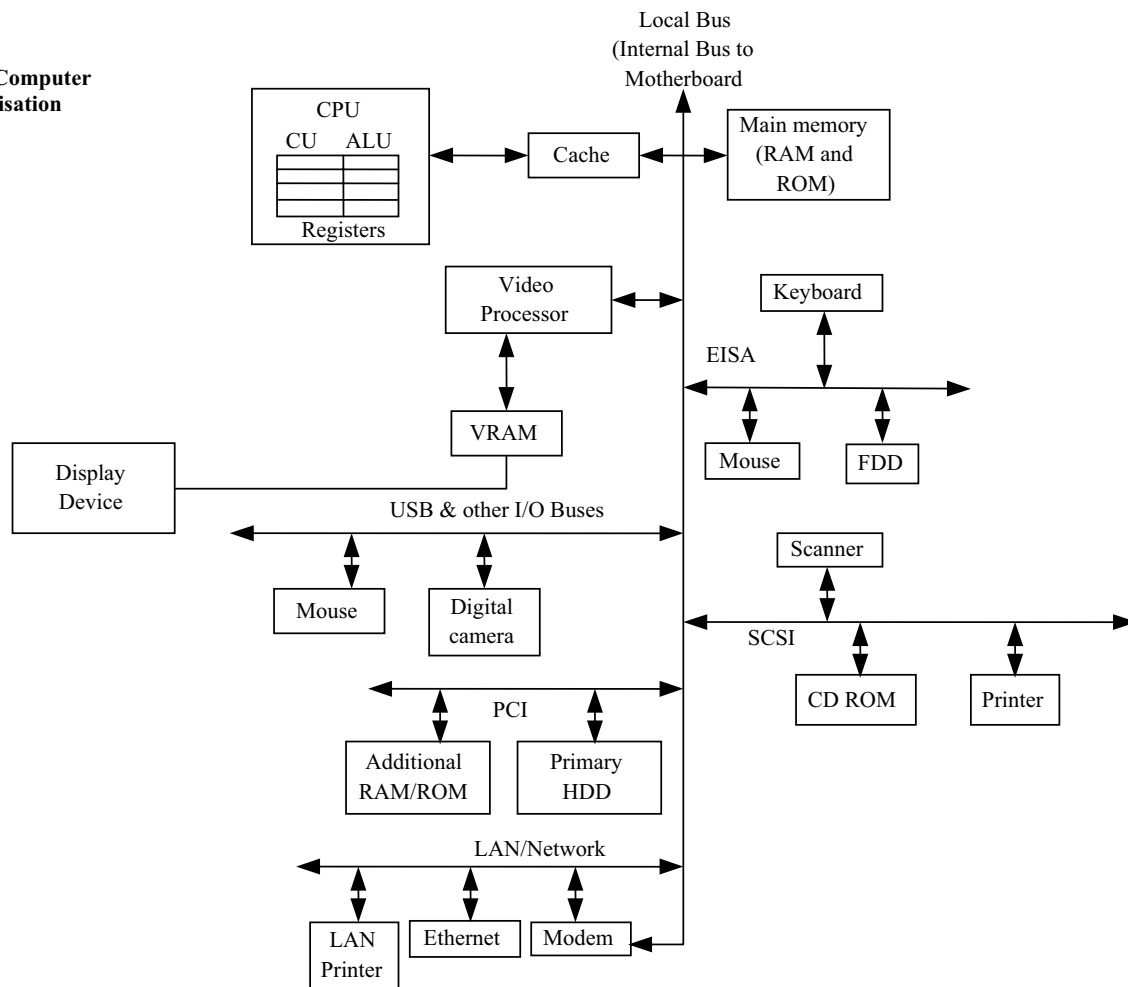


Figure 1: Block Diagram of a Microcomputer System

The microcomputer has a single microprocessor, a number of RAM and ROM chips and an interface units communicates with various external devices through the I/O Bus.

The Input / Output subsystem of a computer, referred to as I/O, provides an efficient mode of communication between the central system and the output environment. External devices that are under the direct control of the computers are said to be connected **on-line**. These devices are designed to read information into or out of the memory unit upon command from the CPU and are considered to be part of the computer system. Input / Output devices attached to the computer are also called peripherals. We can broadly classify peripherals or external devices into 3 categories:

- Human readable: suitable for communicating with the computer user, e.g., video display terminals (VDTs) & printers.
- Machine-readable: suitable for communicating with equipment, e.g., magnetic disks and tape system.
- Communication: suitable for communicating with remote devices, e.g., terminal, a machine-readable device.

2.3 THE INPUT /OUTPUT INTERFACE

The Input /Output interface provides a method for transferring information between internal storage and external I/O devices. Peripherals connected to a computer need special communication links for interfacing them with the CPU. The purpose of the

communication link is to resolve the differences that exist between the central computer and each peripheral. The major differences are:

- Peripherals are electromagnetic and electromechanical devices and their operations are different from the operation of the CPU and the memory, which are electronic devices.
- The data transfer rate of peripherals is usually slower than the transfer rate of the CPU, and consequently a synchronization mechanism may be needed.
- Data codes and formats in peripherals differ from the word format in the CPU and memory.
- The operating modes of peripherals are different from each other and each must be controlled so as not to disturb the operation of other peripherals connected to the CPU.

To resolve these differences, computer systems include special hardware component between the CPU and peripherals to supervise and synchronize all input and output transfers. These components are called *interface* units because they interface between the processor bus and the peripheral device.

Functions of I/O Interface

An I/O interface is bridge between the processor and I/O devices. It controls the data exchange between the external devices and the main memory; or external devices and processor registers. Therefore, an I/O interface provides an interface internal to the computer which connects it to the processor and main memory and an interface external to the computer connecting it to external device or peripheral. The I/O interface should not only communicate the information from processor to main I/O device, but it should also coordinate these two. In addition, since there are speed differences between processor and I/O devices, the I/O interface should have facilities like buffer and error detection mechanism. Therefore, the major functions or requirements of an I/O interface are:

It should be able to provide control and timing signals

The need of I/O from various I/O devices by the processor is quite unpredictable. In fact it depends on I/O needs of particular programs and normally does not follow any pattern. Since, the I/O interface also shares system bus and memory for data input/output, control and timing are needed to coordinate the flow of data from/to external devices to/from processor or memory. For example, the control of the transfer of data from an external device to the processor might involve the following steps:

1. The processor enquires from the I/O interface to check the status of the attached device. The status can be busy, ready or out of order.
2. The I/O interface returns the device status.
3. If the device is operational and ready to transmit, the processor requests the transfer of data by means of a command, which is a binary signal, to the I/O interface.
4. The I/O interface obtains a unit of data (e.g., 8 or 16 bits) from the external device.
5. The data is transferred from the I/O interface to the processor.

It should communicate with the processor

The above example clearly specifies the need of communication between the processor and I/O interface. This communication involves the following steps:

1. Commands such as READ SECTOR, WRITE SECTOR, SEEK track number and SCAN record-id sent over the control bus.
2. Data that are exchanged between the processor and I/O interface sent over the data bus.
3. Status: As peripherals are so slow, it is important to know the status of the I/O interface. The status signals are BUSY or READY or in an error condition from I/O interface.
4. Address recognition as each word of memory has an address, so does each I/O device. Thus an I/O interface must recognize one unique address for each peripheral it controls.

It should communicate with the I/O device

Communication between I/O interface and I/O device is needed to complete the I/O operation. This communication involves commands, status or data.

It should have a provision for data buffering

Data buffering is quite useful for the purpose of smoothing out the gaps in speed of processor and the I/O devices. The data buffers are registers, which hold the I/O information temporarily. The I/O is performed in short bursts in which data are stored in buffer area while the device can take its own time to accept them. In I/O device to processor transfer, data are first transferred to the buffer and then passed on to the processor from these buffer registers. Thus, the I/O operation does not tie up the bus for slower I/O devices.

Error detection mechanism should be in-built

The error detection mechanism may involve checking the mechanical as well as data communication errors. These errors should be reported to the processor. The examples of the kind of mechanical errors that can occur in devices are paper jam in printer, mechanical failure, electrical failure etc. The data communication errors may be checked by using parity bit.

2.4 THE DEVICE CONTROLLERS AND ITS STRUCTURE

All the components of the computer communicate with the processor through the system bus. That means the I/O devices need to be attached to the system bus. However, I/O devices are not connected directly to the computer's system bus. Instead they are connected to an intermediate electronic device interface called a device controller, which in turn is connected to the system bus. Hence a device controller is an interface between an I/O device and the system bus. On one side, it knows how to communicate with the I/O device connected to it, and on the other it knows how to communicate with the computer's CPU or processor and memory through the system bus.

2.4.1 Device Controller

A device controller need not necessarily control a single device. It can usually control multiple I/O devices. It comes in the form of an electronic circuit board that plugs directly into the system bus, and there is a cable from the controller to each device it controls. The cables coming out of the controller are usually terminated at the back panel of the main computer box in the form of connectors known as ports.

The Figure 2 below illustrates how I/O devices are connected to a computer system through device controllers. Please note the following points in the diagram:

- Each I/O device is linked through a hardware interface called I/O Port.
- Single and Multi-port device controls single or multi-devices.
- The communication between I/O controller and Memory is through bus only in case of Direct Memory Access (DMA), whereas the path passes through the CPU for such communication in case of non-DMA.

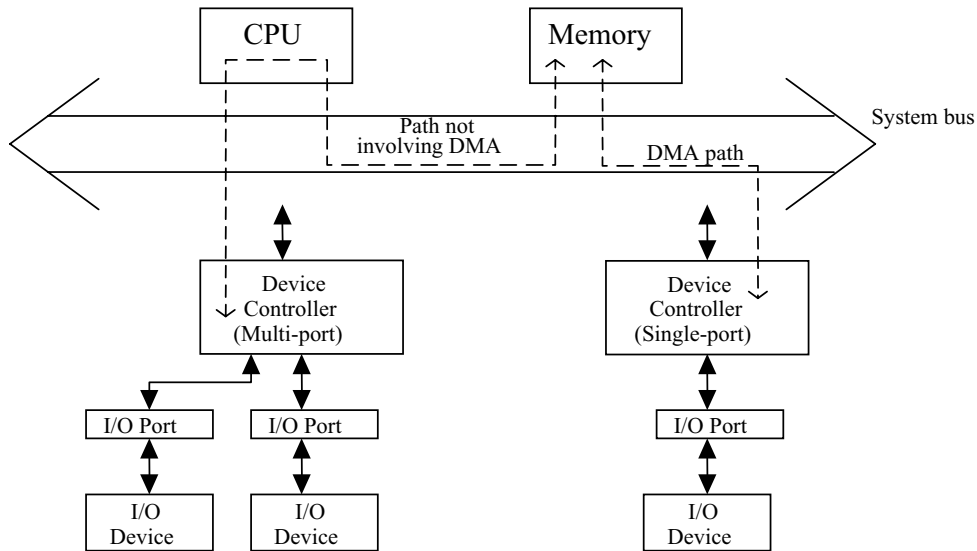


Figure 2: Connecting I/O Devices using Device Controller

Using device controllers for connecting I/O devices to a computer system instead of connecting them directly to the system bus has the following advantages:

- A device controller can be shared among multiple I/O devices allowing many I/O devices to be connected to the system.
- I/O devices can be easily upgraded or changed without any change in the computer system.
- I/O devices of manufacturers other than the computer manufacturer can be easily plugged in to the computer system. This provides more flexibility to the users in buying I/O devices of their choice.

2.4.2 Structure of an I/O Interface

Due to the complexity and the number of external devices that the I/O interface control, there is no standard structure of I/O interface. Let us give a general structure to an I/O interfaces:

- There is a need of I/O logic, which should interpret and execute dialogue between the processor and I/O interface. Therefore, there need to be control lines between processors and I/O interface.
- The data line connecting I/O interface to the system bus must exist. These lines serve the purpose of data transfer.
- Data registers may act as buffer between processor and I/O interface.
- The I/O interface contains logic specific to the interface with each device that it controls.

Figure 3: The General Structure of an I/O

Figure 3 above is a typical diagram of an I/O interface which in addition to all the registers as defined above has status/control registers which are used to pass on the status information or the control information.

2.5 DEVICE DRIVERS

A device driver is software interface which manages the communication with, and the control of, a specific I/O device, or type of device. It is the task of the device driver to convert the logical requests from the user into specific commands directed to the device itself. For example, a user request to write a record to a floppy disk would be realised within the device driver as a series of actions, such as checking for the presence of a disk in the drive, locating the file via the disk directory, positioning the heads, etc.

Device Drivers in UNIX, MS-DOS and Windows System

Although device drivers are in effect add-on modules, they are nevertheless considered to be part of the system since they are closely integrated with the Input/Output Control System, which deals with I/O related system calls.

In UNIX the device drivers are usually linked onto the object code of the kernel (the core of the operating system). This means that when a new device is to be used, which was not included in the original construction of the operating system, the UNIX kernel has to be re-linked with the new device driver object code. This technique has the advantages of run-time efficiency and simplicity, but the disadvantage is that the addition of a new device requires regeneration of the kernel. In UNIX, each entry in the /dev directory is associated with a device driver which manages the communication with the related device. A list of some device names is as shown below:

Device name	Description
/dev/console	system console
/dev/tty01	user terminal 1
/dev/tty02	user terminal 2
/dev/lp	line printer
/dev/dsk/f03h	1.44 MB floppy drive

In MS-DOS, device drivers are installed and loaded dynamically, i.e., they are loaded into memory when the computer is started or re-booted and accessed by the operating system as required. The technique has the advantage that it makes addition of a new driver much simpler, so that it could be done by relatively unskilled users. The additional merit is that only those drivers which are actually required need to be loaded into the main memory. The device drivers to be loaded are defined in a special file called CONFIG.SYS, which must reside in the root directory. This file is automatically read by MS-DOS at start-up of the system, and its contents acted upon. A list of some device name is as shown below:

Device name	Description
con:	keyboard/screen
com1:	serial port1
com2:	serial port2
lpt1:	printer port1
A:	first disk drive
C:	hard disk drive

In the Windows system, device drivers are implemented as dynamic link libraries (DLLs). This technique has the advantages that DLLs contains shareable code which means that only one copy of the code needs to be loaded into memory. Secondly, a driver for a new device can be implemented by a software or hardware vendor without the need to modify or affect the Windows code, and lastly a range of optional drivers can be made available and configured for particular devices.

In the Windows system, the idea of Plug and Play device installation is required to add a new device such as a CD drive, etc. The objective is to make this process largely automatic; the device would be attached and the driver software loaded. Thereafter, the installation would be automatic; the settings would be chosen to suit the host computer configuration.

Check Your Progress 1

- What are the functions of an I/O interface?
.....
.....
.....
- State True or False:**
 - Com1 is a UNIX port. T/F
 - The buffering is done by data register. T/F
 - Device controller is shareable among devices. T/F
 - I/O system is basically needed for better system efficiency T/F
 - Device drives can be provided using software libraries. T/F
 - The devices are normally connected directly to the system bus. T/F
 - Data buffering is helpful for smoothing out the speed differences between CPU and input/output devices. T/F
 - Input/ output module is needed only for slower I/O devices T/F

3. What is a device driver? Differentiate between device controller and device drivers.

.....

.....

.....

.....

.....

2.6 INPUT-OUTPUT TECHNIQUES

After going through the details of the device interfaces, the next point to be discussed is how the interface may be used to support input/output from devices. Binary information received from an external device is usually stored in memory for later processing. Information transferred from the central computer into an external device originates in the memory unit. Data transfer between the central computer and I/O devices may be handled in a variety of modes. Three techniques are possible for I/O operation. These are:

- Programmed input/output
- Interrupt driven input/output
- Direct memory access

Figure 4 gives an overview of these three techniques

	Interrupt Required	I/O interface to/from memory transfer (refer Figure 2)
Programmed I/O	No	Through CPU
Interrupt-driven I/O	Yes	Through CPU
DMA	Yes	Direct to Memory

Figure 4: Overview of the three Input/ Output

In programmed I/O, the I/O operations are completely controlled by the processor. The processor executes a program that initiates, directs and terminate an I/O operation. It requires a little special I/O hardware, but is quite time consuming for the processor since the processor has to wait for slower I/O operations to complete.

With interrupt driven I/O, when the interface determines that the device is ready for data transfer, it generates an interrupt request to the computer. Upon detecting the external interrupt signal, the processor stops the task it is processing, branches to a service program to process the I/O transfer, and then returns to the task it was originally performing which results in the waiting time by the processor being reduced.

With both programmed and interrupt-driven I/O, the processor is responsible for extracting data from the main memory for output and storing data in the main memory during input. What about having an alternative where I/O device may directly store data or retrieve data from memory? This alternative is known as direct memory access (DMA). In this mode, the I/O interface and main memory exchange data directly, without the involvement of processor.

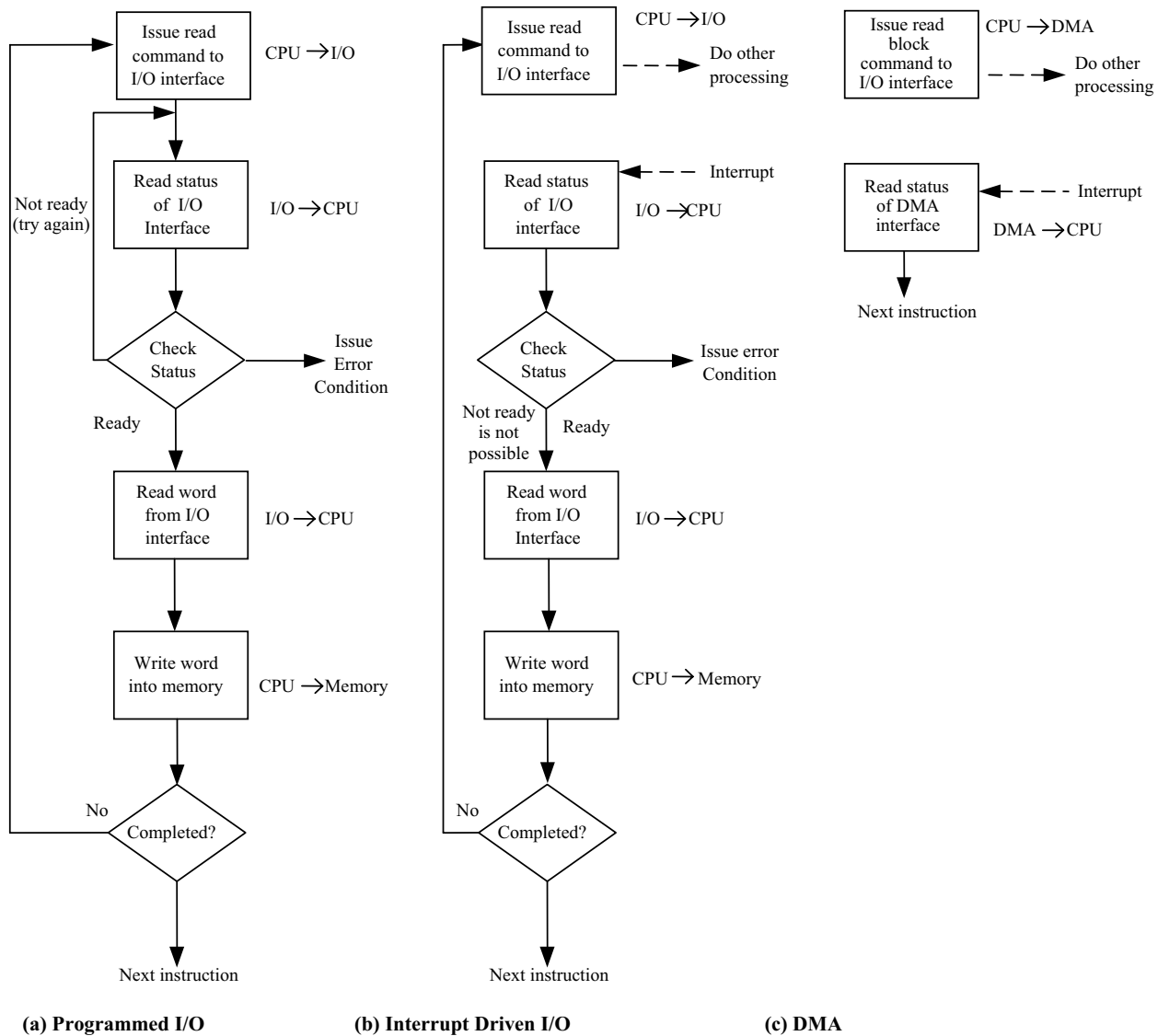


Figure 5: Three techniques of I/O

2.6.1 Programmed Input /Output

Programmed input/output is a useful I/O method for computers where hardware costs need to be minimised. The input or output operation in such cases may involve:

- Transfer of data from I/O device to the processor registers.
- Transfer of data from processor registers to memory.

With the programmed I/O method, the responsibility of the processor is to constantly check the status of the I/O device to check whether it is free or it has finished inputting the data. Thus, this method is very time consuming where the processor wastes a lot of time in checking and verifying the status of an I/O device. Figure 5(a) gives an example of the use of programmed I/O to read in a block of data from a peripheral device into memory.

I/O Commands

There are four types of I/O commands that an I/O interface may receive when it is addressed by a processor:

- **Control:** These commands are device specific and are used to provide specific instructions to the device, e.g. a magnetic tape requiring rewinding and moving forward by a block.
- **Test:** This command checks the status such as if a device is ready or not or is in error condition.
- **Read:** This command is useful for input of data from input device.
- **Write:** this command is used for output of data to output device.

I/O Instructions:

An I/O instruction is stored in the memory of the computer and is fetched and executed by the processor producing an I/O-related command for the I/O interface. With programmed I/O, there is a close correspondence between the I/O-related instructions and the I/O commands that the processor issues to an I/O interface to execute the instructions.

In systems with programmed I/O, the I/O interface, the main memory and the processors normally share the system bus. Thus, each I/O interface should interpret the address lines to determine if the command is for itself. There are two methods for doing so. These are called memory-mapped I/O and isolated I/O.

With memory-mapped I/O, there is a single address space for memory locations and I/O devices. The processor treats the status and data registers of I/O interface as memory locations and uses the same machine instructions to access both memory and I/O devices. For a memory-mapped I/O only a single read and a single write line are needed for memory or I/O interface read or write operations. These lines are activated by the processor for either memory access or I/O device access. Figure 6 shows the memory-mapped I/O system structure.

With isolated I/O, there are separate control lines for both memory and I/O device read or write operations. Thus a memory reference instruction does not affect an I/O device. In isolated I/O, the I/O devices and memory are addressed separately; hence separate input/output instructions are needed which cause data transfer between addressed I/O interface and processor. Figure 7 shows the structure of isolated I/O.

Figure 7: Structure of Isolated I/O

2.6.2 Interrupt-Driven Input/Output

The problem with programmed I/O is that the processor has to wait a long time for the I/O interface to see whether a device is free or wait till the completion of I/O. The result is that the performance of the processor goes down tremendously. What is the solution? What about the processor going back to do other useful work without waiting for the I/O device to complete or get freed up? But how will the processor be intimated about the completion of I/O or a device is ready for I/O? A well-designed mechanism was conceived for this, which is referred to as interrupt-driven I/O. In this mechanism, provision of interruption of processor work, once the device has finished the I/O or when it is ready for the I/O, has been provided.

The interrupt-driven I/O mechanism for transferring a block of data is shown in Figure 5(b). Please note that after issuing a read command (for input) the CPU goes off to do other useful work while I/O interface proceeds to read data from the associated device. On the completion of an instruction cycle, the CPU checks for interrupts (which will occur when data is in data register of I/O interface and it now needs CPU's attention). Now CPU saves the important register and processor status of the executing program in a stack and requests the I/O device to provide its data, which is placed on the data bus by the I/O device. After taking the required action with the data, the CPU can go back to the program it was executing before the interrupt.

2.6.3 Interrupt-Processing

The occurrence of an interrupt fires a numbers of events, both in the processor hardware and software. Figure 8 shows a sequence.

Figure 8: Interrupt-Processing Sequence

When an I/O device completes an I/O operation, the following sequence of hardware events occurs:

1. The device issues an interrupt signal to the processor.
2. The processor finishes execution of the current instruction *before responding* to the interrupt.
3. The processor tests for the interrupts and sends an acknowledgement signal to the device that issued the interrupt.
4. The minimum information required to be stored for the task being currently executed, before the CPU starts executing the interrupt routine (using its registers) are:
 - (a) The status of the processor, which is contained in the register called program status word (PSW), and
 - (b) The location of the next instruction to be executed, of the currently executing program, which is contained in the program counter (PC).

5. The processor now loads the PC with the entry location of the interrupt-handling program that will respond to this interrupting condition. Once the PC has been loaded, the processor proceeds to execute the next instruction, that is the next instruction cycle, which begins with an instruction fetch. Because the instruction fetch is determined by the contents of the PC, the result is that control is transferred to the interrupt-handler program. The execution results in the following operations:
6. The PC & PSW relating to the interrupted program have already been saved on the system stack. In addition, the contents of the processor registers are also needed to be saved on the stack that are used by the called Interrupt Servicing Routine because these registers may be modified by the interrupt-handler. Figure 9(a) shows a simple example. Here a user program is interrupted after the instruction at location N. The contents of all of the registers plus the address of the next instruction (N+1) are pushed on to the stack.
7. The interrupt handler next processes the interrupt. This includes determining of the event that caused the interrupt and also the status information relating to the I/O operation.
8. When interrupt processing is complete, the saved register values are retrieved from the stack and restored to the registers, which are shown in Figure 9(b).
9. The final step is to restore the values of PSW and PC from the stack. As a result, the instruction to be executed will be from the previously interrupted program.

Thus, interrupt handling involves interruption of the currently executing program, execution of interrupt servicing program and restart of interrupted program from the point of interruption.

Design issues: Two design issues arise in implementing interrupt-driven I/O:

- 1) How does the processor determine which device issued the interrupt?
- 2) If multiple interrupts have occurred, how does the processor decide which one to be processed first?

To solve these problems, four general categories of techniques are in common use:

- **Multiple Interrupt Lines:** The simplest solution to the problems above is to provide multiple interrupt lines, which will result in immediate recognition of the interrupting device. Priorities can be assigned to various interrupts and the interrupt with the highest priority should be selected for service in case a multiple interrupt occurs. But providing multiple interrupt lines is an impractical approach because only a few lines of the system bus can be devoted for the interrupt.
- **Software Poll:** In this scheme, on the occurrence of an interrupt, the processor jumps to an interrupt service program or routine whose job it is to poll (roll call) each I/O interface to determine which I/O interface has caused the interrupt. This may be achieved by reading the status register of the I/O interface. Once the correct interface is identified, the processor branches to a device-service routine specific to that device. The disadvantage of the software poll is that it is time consuming.
- **Daisy chain:** This scheme provides a hardware poll. With this technique, an interrupt acknowledge line is chained through various interrupt devices. All I/O interfaces share a common interrupt request line. When the processor senses an interrupt, it sends out an interrupt acknowledgement. This signal passes through all the I/O devices until it gets to the requesting device. The first device which has made the interrupt request thus senses the signal and responds by putting in a word which is normally an address of interrupt servicing program or a unique identifier on the data lines. This word is also referred to as interrupt vector. This address or identifier in turn is used for selecting an appropriate interrupt-servicing program. The daisy chaining has an in-built priority scheme, which is determined by the sequence of devices on interrupt acknowledge line.
- **Bus arbitration:** In this scheme, the I/O interface first needs to control the bus and only after that it can request for an interrupt. In this scheme, since only one of the interfaces can control the bus, therefore only one request can be made at a time. The interrupt request is acknowledged by the CPU on response of which I/O interface places the interrupt vector on the data lines. An interrupt vector normally contains the address of the interrupt serving program.

An example of an interrupt vector can be a personal computer, where there are several IRQs (Interrupt request) for a specific type of interrupt.

2.6.4 DMA (Direct Memory Access)

In both interrupt-driven and programmed I/O, the processor is busy with executing input/output instructions and the I/O transfer rate is limited by the speed with which the processor can test and service a device. What about a technique that requires minimal intervention of the CPU for input/output? These two types of drawbacks can be overcome with a more efficient technique known as DMA, which acts as if it has taken over control from the processor. Hence, the question is: why do we use DMA interface? It is used primarily when a large amount of data is to be transferred from the I/O device to the Memory.

DMA Function

Although the CPU intervention in DMA is minimised, yet it must use the path between interfaces that is the system bus. Thus, DMA involves an additional interface on the system bus. A technique called cycle stealing allows the DMA interface to transfer one data word at a time, after which it must return control of the bus to the processor. The processor merely delays its operation for one memory cycle to allow the directly memory I/O transfer to “steal” one memory cycle. When an I/O is requested, the processor issues a command to the DMA interface by sending to the DMA interface the following information (Figure 10):

- Which operations (read or write) to be performed, using the read or write control lines.
- The address of I/O devices, which is to be used, communicated on the data lines.
- The starting location on the memory where the information will be read or written to be communicated on the data lines and is stored by the DMA interface in its address register.
- The number of words to be read or written is communicated on the data lines and is stored in the data count register.

Figure 10: DMA block diagram

The DMA interface transfers the entire block of data, one word at a time, directly to or from memory, without going through the processor. When the transfer is complete, the DMA interface sends an interrupt signal to the processor. Thus, in DMA the processor involvement can be restricted at the beginning and end of the transfer, which can be shown as in the figure above. But the question is when should the DMA take control of the bus?

For this we will recall the phenomenon of execution of an instruction by the processor. Figure 11 below shows the five cycles for an instruction execution. The Figure also shows the five points where a DMA request can be responded to and a point where the interrupt request can be responded to. Please note that an interrupt request is acknowledged only at one point of an instruction cycle, and that is at the interrupt cycle.

Figure 11: DMA and Interrupt Breakpoints

The DMA mechanism can be configured into a variety of ways. Some possibilities are shown below in Figure 12(a), in which all interfaces share the same system bus. The DMA acts as the supportive processor and can use programmed I/O for exchanging data between memory and I/O interface through DMA interface. But once again this spoils the basic advantage of DMA not using extra cycles for transferring information from memory to/from DMA and DMA from/to I/O interface.

The Figure 12(b) configuration suggests advantages over the one shown above. In these systems a path is provided between I/O interface and DMA interface, which does not include the system bus. The DMA logic may become part of an I/O interface and can control one or more I/O interfaces. In an extended concept an I/O bus can be connected to this DMA interface. Such a configuration (shown in Figure 12 (c)) is quite flexible and can be extended very easily. In both these configurations, the added advantage is that the data between I/O interface and DMA interface is transferred off the system bus, thus eliminating the disadvantage we have witnessed for the first configuration.

Check Your Progress 2

1. Which of the I/O techniques does not require an Interrupt Signal? Is this technique useful in Multiprogramming Operating Systems? Give reason.
.....
.....
.....
2. What are the techniques of identifying the device that has caused the Interrupt?
.....
.....
.....
3. What are the functions of I/O interface? What is DMA?
.....
.....
.....
4. State True or False:
 - a) Daisy chain provides software poll. T/F
 - b) I/O mapped I/O scheme requires no additional lines from CPU to I/O device except for the system bus. T/F
 - c) Most of the I/O processors have their own memory while a DMA module does not have its own memory except for a register or a simple buffer area. T/F
 - d) The advantage of interrupt driven I/O over programmed I/O is that in the first the interrupt mechanisms free I/O devices quickly. T/F

2.7 INPUT-OUTPUT PROCESSORS

Before discussing I/O processors, let us briefly recapitulate the development in the area of input/output functions. These can be summarised as:

1. The CPU directly controls a peripheral device.
2. Addition of I/O controller or I/O interface: The CPU uses programmed I/O without interrupts. CPU was separated from the details of external I/O interfaces.
3. Contained use of I/O controllers but with interrupts: The CPU need not spend time waiting for an I/O operation to be performed, increasing efficiency.
4. Direct access of I/O interface to the memory via DMA: CPU involvement reduced to at the beginning and at the end of DMA operation.

5. The CPU directs the I/O processors to execute an I/O program in memory. The I/O processor fetches and executes these instructions without CPU intervention. This allows the CPU to specify a sequence of I/O activities and to be interrupted only when the entire sequence has been performed. With this architecture, a large set of I/O devices can be controlled, with minimum CPU involvement.

With the last two steps (4 and 5), a major change occurs with the introduction of the concept of an I/O interface capable of executing a program. For steps 5, the I/O interface is often referred to as an I/O channel and I/O processor.

Characteristics of I/O Channels

The I/O channel represents an extension of the DMA concept. An I/O channel has the ability to execute I/O instructions, which gives complete control over the I/O operation. With such devices, the CPU does not execute I/O instructions. Such instructions are stored in the main memory to be executed by a special-purpose processor in the I/O channel itself. Thus, the CPU initiates an I/O transfer by instructing the I/O channel to execute a program in memory. Two types of I/O channels are commonly used which can be seen in Figure 13 (a and b).

(a) Selector Channel

(b) Multiplexer Channel

Figure 13: I/O Channel Structures

A **selector channel** controls multiple high-speed devices and, at any one time, is dedicated to the transfer of data with one of those devices. Each device is handled by a controller or I/O interface. Thus the I/O channel serves in place of the CPU in controlling these I/O controllers.

A **multiplexer channel** can handle I/O with multiple devices at the same time. If the devices are slow then byte multiplexer is used. Let us explain this with an example. If we have three slow devices which need to send individual bytes as:

```
X1 X2 X3 X4 X5 .....
Y1 Y2 Y3 Y4 Y5.....
Z1 Z2 Z3 Z4 Z5.....
```

Then on a byte multiplexer channel they may send the bytes as X1 Y1 Z1 X2 Y2 Z2 X3 Y3 Z3..... For high-speed devices, blocks of data from several devices are interleaved. These devices are called **block multiplexer**.

2.8 EXTERNAL COMMUNICATION INTERFACES

The external interface is the interface between the I/O interface and the peripheral devices. This interface can be characterised into two main categories: (a) parallel interface and (b) serial interface.

In parallel interface multiple bits can be transferred simultaneously. The parallel interface is normally used for high-speed peripherals such as tapes and disks. The dialogues that take place across the interface include the exchange of control information and data.

In serial interface only one line is used to transmit data, therefore only one bit is transferred at a time. Serial printers are used for serial printers and terminals. With a new generation of high-speed serial interfaces, parallel interfaces are becoming less common.

In both cases, the I/O interface must engage in a dialogue with the peripheral. The dialogue for a read or write operation is as follows:

- A control signal is sent by I/O interface to the peripheral requesting the permission to send (for write) or receive (for read) data.
- The peripheral acknowledges the request.
- The data are transferred from I/O interface to peripheral (for write) or from peripheral to I/O interface (for read).
- The peripheral acknowledges receipt of the data.

The connection between an I/O interface in a computer system and external devices can be either point-to-point or multipoint. A point-to-point interface provides a dedicated line between the I/O interface and the external device. For example keyboard, printer and external modems are point-to-point links. The most common serial interfaces are RS-232C and EIA-232.

A multipoint external interface used to support external mass storage devices (such as disk and tape drives) and multimedia devices (such as CD-ROM, video, audio).

Two important examples of external interfaces are FireWire and InfiniBand.

Check Your Progress 3

1. What is the need of I/O channels?
.....
.....
.....
.....
2. What is the need of external Communication Interfaces?
.....
.....
.....
.....

2.9 SUMMARY

This unit is totally devoted to the I/O of computer system. In this unit we have discussed the identification of I/O interface, description of I/O techniques such as programmed I/O, interrupt-driven I/O and direct memory access. These techniques are useful for increasing the efficiency of the input-output transfer process. The concepts of device drivers for all types of operating systems and device controllers are also discussed with this unit. We have also defined an input/output processor, the external communication interfaces such as serial and parallel interfaces and interrupt processing. The I/O processors are the most powerful I/O interfaces that can execute the complete I/O instructions. You can always refer to further reading for detail design.

2.10 SOLUTIONS /ANSWERS

Check Your Progress 1

1. The functions of I/O interfaces are to provide:
 - Timing and control signal.
 - Communication with processor and I/O devices.
 - Support for smoothing the speed gap between CPU and Memory using buffering.
 - Error detection.
2. (a) False (b) True (c) True (d) True (e) True (f) False (g) True (h) False
3. A device driver is a software module which manages the communication with, and the control of, a specific I/O device, or type of device. The difference between device driver and controller are:
 - One device controller can control many devices, whereas drivers are device specific.
 - Device controllers are a more intelligent hardware-software combination than device drivers.
 - I/O controllers allow different types and upgradeability of devices whereas device driver is device specific.

Check Your Progress 2

1. The technique Programmed I/O does not require an Interrupt. It is very inefficient for Multiprogramming environment as the processor is busy waiting for the I/O to complete, while this time would have been used for instruction execution of other programs.
2. The techniques for recognition of interrupting device/conditions can be:
 - Multiple Interrupt Lines: Having separate line for a device, thus direct recognition.
 - Software Poll: A software driven roll call to find from devices whether it has made an interrupt request.
 - Daisy Chain: A hardware driven passing the buck type signal that moves through the devices connected serially. The device on receipt of signal on his turn, if has interrupt informs its address.
 - Bus Arbitration: In this scheme, the I/O interface requests for control of the Bus. This is a common process when I/O processors are used.
3. The functions of I/O interface are:
 - Control and timing signals
 - CPU communications
 - I/O device communication
 - Data buffering
 - In-built error-detection mechanism.

DMA is an I/O technique that minimises the CPU intervention at the beginning and end of a time consuming I/O. One, commonplace where DMA is used is when I/O is required from a Hard Disk, since one single I/O request requires a block of data transfer which on the average may take a few milliseconds. Thus, DMA will free CPU to do other useful tasks while I/O is going on.

4.
 - a) False
 - b) False
 - c) True
 - d) False

Check Your Progress 3

1. The I/O channels were popular in older mainframes, which included many I/O devices and I/O requests from many users. The I/O channel takes control of all I/O instructions from the main processor and controls the I/O requests. It is mainly needed in situations having many I/O devices, which may be shared among multiple users.
2. The external interfaces are the standard interfaces that are used to connect third party or other external devices. The standardization in this area is a must.

UNIT 3 SECONDARY STORAGE TECHNIQUES

Structure	Page No.
3.0 Introduction	64
3.1 Objectives	64
3.2 Secondary Storage Systems	65
3.3 Hard Drives	65
3.3.1 Characteristics: Drive Speed, Access Time, Rotation Speed	
3.3.2 Partitioning & Formatting: FAT, Inode	
3.3.3 Drive Cache	
3.3.4 Hard Drive Interface: IDE, SCSI, EIDE, Ultra DMA & ATA/66	
3.4 Removable Drives	72
3.4.1 Floppy Drives	
3.4.2 CD-ROM & DVD-ROM	
3.5 Removable Storage Options	75
3.5.1 Zip, Jaz & Other Cartridge Drives	
3.5.2 Recordable CDs & DVDs	
3.5.3 CD-R vs CD-RW	
3.5.4 Tape Backup	
3.6 Summary	78
3.7 Solutions /Answers	78

3.0 INTRODUCTION

In the previous units of this block, we have discussed the primary memory system, high speed memories, the memory system of microcomputer, and the input/output interfaces and techniques for a computer. In this unit we will discuss the secondary storage devices such as magnetic tapes, magnetic disks and optical disks, also known as backing storage devices. The main purpose of such a device is that it provides a means of retaining information on a permanent basis. The main discussion provides the characteristics of hard-drives, formatting, drive cache, interfaces, etc. The detailed discussion on storage devices is being presented in the Unit. The storage technologies have moved a dimension from very small storage devices to Huge Giga byte memories. Let us also discuss some of the technological achievements that made such a technology possible.

3.1 OBJECTIVES

Storage is the collection of places where long-term information is kept. At the end of the unit you will be able to:

- describe the characteristics of the different secondary storage drives, i.e., their drive speed, access time, rotation speed, density etc.;
- describe the low-level and high level formatting of a blank disk and also the use of disk partitioning;
- distinguish among the various types of drives, i.e., hard drives , optical drives removable drives and cartridge drive; and
- define different type of disk formats.

3.2 SECONDARY STORAGE SYSTEMS

As discussed in Block 2 Unit 1, there are several limitations of primary memory such as limited capacity, that is, it is not sufficient to store a very large volume of data; and volatility, that is, when the power is turned off the data stored is lost. Thus, the secondary storage system must offer large storage capacities, low cost per bit and medium access times. Magnetic media have been used for such purposes for a long time. Current magnetic data storage devices take the form of floppy disks and hard disks and are used as secondary storage devices. But audio and video media, either in compressed form or uncompressed form, require higher storage capacity than the other media forms and the storage cost for such media is significantly higher.

Optical storage devices offer a higher storage density at a lower cost. CD-ROM can be used as an optical storage device. Many software companies offer both operating system and application software on CD-ROM today. This technology has been the main catalyst for the development of multimedia in computing because it is used in the multimedia external devices such as video recorders and digital recorders (Digital Audio Tape) which can be used for the multimedia systems.

Removable disk, tape cartridges are other forms of secondary storage devices are used for back-up purposes having higher storage density and higher transfer rate.

3.3 HARD DRIVES

The Disks are normally mounted on a disk drive that consists of an arm and a shaft along with the electronic circuitry for read-write of data. The disk rotates along with the shaft. A non-removable disk is permanently mounted on the disk drive. One of the most important examples of a non-removable disk is the hard disk of the PC. The disk is a platter coated with magnetic particles. Early drives were large. Later on, smaller hard (rigid) disk drivers were developed with fixed and removable pack. Each pack held about 30MB of data and became known as the Winchester drive. The storage capacity of today's Winchester disks is usually of the order of a few tens of Megabytes to a few Gigabytes. Most Winchester drives have the following common features:

- the disk and read/write heads are enclosed in a sealed airtight unit;
- the disk(s) spin at a high speed, one such speed may be 7200 revolutions per minute;
- the read/write head do not actually touch the disk surface;
- the disk surface contains a magnetic coating;
- the data on disk surface (platter) are arranged in the series of concentric rings. Each ring is called a track, is subdivided into a number of sectors, each sector holding a specific number of data elements called bytes or characters.
- The smallest unit that can be written to or read from the disk is a sector. The storage capacity of the disk can be determined as the number of tracks, number of sectors, byte per sector and number of read/write heads.

(a) An Open Disk Casing

(b) Tracks and Cylinders

Figure 1: The Hard Disk

3.3.1 Characteristics: Drive Speed, Access Time, Rotation Speed

Tracks and Sectors: The disk is divided into concentric rings called tracks. A track is thus one complete rotation of the disk underneath the read/write head. Each track is subdivided into a number of sectors. Each sector contains a specific number of bytes or characters. Typical sector capacities are 128, 256, 512, 1024 and 4096 bytes.

Bad Blocks: The drive maintains an internal table which holds the sectors or tracks which cannot be read or written to because of surface imperfections. This table is called the bad block table and is created when the disk surface is initially scanned during a low-level format.

Sector Interleave: This refers to the numbering of the sectors located in a track. A one to one interleave has sectors numbered sequentially 0,1,2,3,4 etc. The disk drive rotates at a fixed speed 7200 RPM, which means that there is a fixed time interval between sectors. A slow computer can issue a command to read sector 0, storing it in an internal buffer. While it is doing this, the drive makes available sector 1 but the computer is still busy storing sector 0. Thus the computer will now have to wait one full revolution till sector 1 becomes available again. Renumbering the sectors like 0,8,1,9,2,10,3,11 etc., gives a 2:1 interleave. This means that the sectors are alternated, giving the computer slightly more time to store sectors internally than previously.

Drive Speed: The amount of information that can be transferred in or out of the memory in a second is termed as disk drive speed or data transfer rate. The speed of the disk drive depends on two aspects, bandwidth and latency.

- **Bandwidth:** The bandwidth can be measured in bytes per second. The sustained bandwidth is the average data rate during a large transfer, i.e., the number of bytes divided by the transfer time. The effective bandwidth is the overall data rate provided by the drive. The disk drive bandwidth ranges from less than 0.25 megabytes per second to more than 30 megabytes per second.
- **Access latency:** A disk access simply moves the arm to the selected cylinder and waits for the rotational latency, which may take less than 36ms. The latency

depends upon the rotation speed of the disk which may be anywhere from 300 RPM to 7200 RPM. An average latency of a disk system is equal to half the time taken by the disk to rotate once. Hence, the average latency of a disk system whose rotation speed is 7200 RPM will be $0.5 / 7200 \text{ minutes} = 4.1 \text{ ms}$.

Rotation Speed: This refers to the speed of rotation of the disk. Most hard disks rotate at 7200 RPM (Revolution per Minute). To increase data transfer rates, higher rotation speeds, or multiple read/write heads arranged in parallel or disk arrays are required.

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

The disk access time has two major components:

- **Seek Time:** The seek time is the time for the disk arm to move the heads to the cylinder containing the desired sector.
- **Latency Time:** The latency time is the additional time waiting for the disk to rotate the desired sector to the disk head.

The sums of average seek and latency time is known as the average access time.

3.3.2 Partitioning and Formatting: FAT, Inode

Today the modern PC contains total capacity of approximately 40GB for storage of program and data. Because of this huge capacity, instead of having only one operating system in our PC, partitions are used to provide several separate areas within one disk, each treated as a separate storage device. That is, a disk partition is a sub-division of the disk into one or more areas. Each partition can be used to hold a different operating system. The computer system boots from the active partition and software provided allows the user to select which partition is the active one.

For example, we can run both Windows and Linux operating systems from the same storage of the PC.

A new magnetic disk is just platters of a magnetic recording material. Before a disk can store data, it must be divided into sectors that the disk controller can read and write. This is called low level formatting. Low level formatting fills the disk with a special data structure for each sector, which consists of a header, a data area, and a trailer. The low level formatting is placing track and sector information plus bad block tables and other timing information on the disks. Sector interleave can also be specified at this time.

In any disk system, space at some time in use will become unwanted and hence will be 'free' for another application. The operating system allocates disk space on demand by user programs. Generally, space is allocated in units of fixed size called an allocation unit or a cluster, which is a simple multiple of the disk physical sector size, usually 512 bytes. The DOS operating system forms a cluster by combining two or more sectors so that the smallest unit of data access from a disk becomes a cluster, not a sector. Normally, the size of the cluster can range from 2 to 64 sectors per cluster.

High level formatting involves writing directory structures and a map of free and allocated space (FAT or INODE) to the disk. Often this also means transferring the boot file for the operating system onto the hard disks.

FAT and Inode

The FAT maps the usage of data space of the disk. It contains information about the space used by each individual file, the unused disk space and the space that is unusable due to defects in the disk. Since FAT contains vital information, two copies of FAT are stored on the disk, so that in case one gets destroyed, the other can be used. A FAT entry can contain any of the following:

- unused cluster
- reserved cluster
- bad cluster
- last cluster in file
- next cluster number in the file.

The DOS file system maintains a table of pointers called FAT (File allocation table) which consists of an array of 16-bit values. There is one entry in the FAT for each cluster in the file area, i.e., each entry of the FAT (except the two) corresponds to one cluster of disk space. If the value in the FAT entry doesn't mark an unused, reserved or defective cluster, then the cluster corresponding to the FAT entry is part of a file and the value in the FAT entry would indicate the next cluster in the file.

The first two entries (0 & 1) in FAT are reserved for use by the operating system. Therefore, the cluster number 2 corresponds to the first cluster in the data space of the disk. Prior to any data being written on to the disk, the FAT entries are all set to zero indicating a 'free' cluster. The FAT chain for a file ends with the hexadecimal value, i.e., FFFF. The FAT structure can be shown as in Figure 2 below.

Figure 2: FAT structure

Limitation of FAT16: The DOS designers decided to use clusters with at least four sectors in them (thus a cluster size of at least 2KB) for all FAT16 hard disks. That size suffices for any hard disk with less than a 128MB total capacity. The largest logical disk drives that DOS can handle comfortably have capacities up to 2GB. For such a large volume, the cluster size is 32KB. This means that even if a file contains only a single byte of data, writing it to the disk uses one entire 32KB region of the disk, making that area unavailable for any other file's data storage.

The most recent solution to these large-disk problems was introduced by Microsoft in its OSR2 release of Windows 95 and it was named FAT32. The cluster entry for FAT32 uses 32-bit numbers. The minimum size for a FAT32 volume is 512MB. Microsoft has reserved the top four bits of every cluster number in a FAT32 file

allocation table. That means there are only 28-bits for the cluster number, so the maximum cluster number possible is 268,435,456.

In the UNIX system, the information related to all these fields is stored in an Inode table on the disk. For each file, there is an inode entry in the table. Each entry is made up of 64 bytes and contains the relevant details for that file. These details are:

- a) Owner of the file
- b) Group to which the Owner belongs
- c) File type
- d) File access permissions
- e) Date & time of last access
- f) Date & time of last modification
- g) Size of the file
- h) No. of links
- i) Addresses of blocks where the file is physically present.

3.3.3 Drive Cache

Disk cache may be a portion of RAM, sometimes called soft disk cache that is used to speed up the access time on a disk. In the latest technologies such memory can be a part of disk drive itself. Such memory is sometimes called hard disk cache or buffer.

These hard disk caches are more effective, particularly applicable in multiprogramming machines or in disk file servers, but they are expensive, and therefore are smaller. Almost all-modern disk drives include a small amount of internal cache. The cycle time of cache would be about a tenth of the main memory cycle time and its cost about 10 times the cost per byte of main memory.

The disk caching technique can be used to speed up the performance of the disk drive system. A set (cache) of buffers is allocated to hold a number of disk blocks which have been recently accessed. In effect, the cached blocks are in memory copies of the disk blocks. If the data in a cache buffer memory is modified, only the local copy is updated at that time. Hence processing of the data takes place using the cached data avoiding the need to frequently access the disk itself.

The main disadvantage of the system using disk caching is risking loss of updated information in the event of machine failures such as loss of power. For this reason, the system may periodically flush the cache buffer in order to minimize the amount of loss.

The disk drive cache is essentially two-dimensional-all the bits are out in the open.

3.3.4 Hard Drive Interface: IDE, SCSI, EIDE, Ultra DMA and ATA/66

Secondary storage devices need a controller to act as an intermediary between the device and the rest of the computer system. On some computers, the controller is an integral part of the computer's main motherboard. On others, the controller is an expansion board that connects to the system bus by plugging into one of the computer's expansion slots. In order that devices manufactured by independent vendors can be used with different computer manufacturers, it is important that the controllers follow some drive interfacing standard. Following are the commonly used drive interface standards:

- **IDE (Integrated Disk Electronics) Devices**

IDE devices are connected to the PC motherboard via a 34-wire ribbon cable. The common drive used today for workstations has capacities of 40MB to

1000MB and rotation speed 7200RPM. The controller is embedded on the disk drive itself. It is an interface between the disk controller and an adapter located on the motherboard. It has good access time of 20ms and data transfer rates of about 1Mbps under ideal conditions. Drives are reasonably cheap. The latest version of the IDE specification enables four IDE channels; each one is capable of supporting two IDE devices.

- **SCSI (Small Computer Systems Interface)**

The other popular way is to attach a disk drive to a PC via a SCSI interface. The common drive choice for servers or high-end workstations with drive capacities ranges from 100MB to 20GB and rotation speed 7200RPM. It is a common I/O interface between the adapter and disk drives or any other peripheral, i.e., CD-ROMs drives, tape drives, printers etc.

SCSI (pronounced “scuzzy”) is an interesting and important variation of the separate device controller idea for each device. It uses a generic device controller (called SCSI controller) on the computer system and allows any device with an SCSI interface to be directly connected to the SCSI bus of the SCSI controller. The SCSI interface of a device contains all circuitry that the device needs to operate with the computer system.

As shown in Figure 3, a SCSI controller connects directly to the computer bus on one side and controls another bus (called SCSI bus) on the other side. Since the SCSI controller is connected to the computer’s bus on one side and to the SCSI bus on the other side, it can communicate with the processor and memory and can also control the devices connected to the SCSI bus. The SCSI bus is a bus designed for connecting devices to a computer in a uniform way.

These drives have fast access time and high data rates but are expensive. One advantage of these drives is that a single SCSI controller can communicate simultaneously with up to seven 16-bit SCSI devices or up to 15 Wide or Ultra-Wide devices. Each device must be assigned a unique SCSI identification between 0 and 7 (or 15).

The versions of SCSI:

Figure 3: SCSI Drive Interface Standards

- The SCSI-1 calls for a cable with 8 data wires plus one for parity.
- The SCSI-2 enables the use of multiple cables to support 16- or even 32-bit data transfers in parallel.

- The SCSI-3 enables the use of multiple cables to support 32- or even 64-bit data transfers in parallel.
- With fast SCSI, it is possible to transfer 40MB of data per second on a single SCSI cable.

- **EIDE (Enhanced IDE)**

The principle behind the EIDE interface is the same as in the IDE interface but this drive has capacities ranging from 10.2GB to 20.5GB. The rotation speed is 7200RPM. Its feature include 9.5ms access time, a 2MB buffer and support for the Ultra ATA/66 interface for high speed data throughput and greater data integrity.

Modern EIDE interfaces enable much faster communication. The speed increases due to improvements in the protocol that describes how the clock cycles will be used to address devices and transfer data. The modern EIDE hard drives are Ultra DMA and ATA/66.

- **Ultra DMA or ATA/33 (AT Attachment):** The ATA standard is the formal specification for how IDE and EIDE interfaces are supposed to work with hard drives. The ATA33 enables up to 33.3 million bytes of data to be transferred each second, hence the name ATA33.
- **ATA/66:** The ATA66 enables up to 66.7 millions bytes of data to be transferred each second, hence the name ATA66 doubles the ATA33.

Check Your Progress 1

1. The seek time of a disk is 30ms. It rotates at the rate of 30 rotations per sec. Each track has 300 sectors. What is the access time of the disk?

.....

2. Calculate the number of entries required in the FAT table using the following parameters for an MS-DOS system:

Disk capacity	30MB
Block size	512 bytes
Blocks/cluster	4

.....

3. What are the purposes of using SCSI, EISA, ATA, IDE?

.....

3.4 REMOVABLE DRIVES

A disk drive with removable disks is called a removable drive. A removable disk can be replaced by another similar disk on the same or different computer, thus providing enormous data storage that is not limited by the size of the disk. Examples of removable disks are floppy disks, CDROM, DVDROM, etc.

3.4.1 Floppy Drives

The disks used with a floppy disk drive are small removable disks made up of plastic coated with magnetic recording material. The disk rotates at 360RPM. Floppies can be accessed from both the sides of the disk. Floppy diskette drives are attached to the motherboard via a 34-wire ribbon cable. You can attach zero, one or two floppy drives and how you connect each one determines whether a drive become either A: or B: A typical floppy drive and floppy is as shown in Figure 4.

(a) Floppy Disk

(b) Floppy Drive

Figure 4: The Floppy Disk and Drive

A floppy is about 0.64 mm thick and is available in diameters 5.25 inch and 3.5 inch. The data are organized in the form of tracks and sectors. The tracks are numbered sequentially inwards, with the outermost being 0. The utility of index hole is that when it comes under a photosensor, the system comes to know that the read/write head is now positioned on the first sector of the current track. The write-protect notch is used to protect the floppy against deletion of recorded data by mistake.

The data in a sector are stored as a series of bits. Once the required sector is found, the average data transfer rate in bytes per second can be computed by the formula:

$$\text{Average data transfer rate} = \frac{\text{bytes/sector} * \text{sector/track} * \text{speed in rpm}}{60}$$

Typical values for IBM/PC compatibles are given in the following table:

Size	Capacity	Tracks	Sectors
5.25	360KB	40	9
5.25	1.2MB	80	15
3.5	720KB	40	18

3.5	1.44MB	80	18
-----	--------	----	----

3.4.2 CD-ROM and DVD-ROM

Optical disks use Laser Disk Technology, which is the latest, and the most promising technology for high capacity secondary storage. The advent of the compact disk digital audio system, a non-erasable optical disk, paved the way for the development of a new low-cost storage technology. In optical storage devices the information is written using a laser beam. We will discuss here the use of some optical disks such as CD-ROM and DVD-ROM devices that are now becoming increasingly popular in various computer applications.

Figure 5: CD-ROM and DVD-ROM

1. **CD-ROM (Compact Disk Read Only Memory):** This technology has evolved out of the entertainment electronics market where cassette tapes and long playing records are being replaced by CDs. The term CD used for audio records stands for Compact Disk. The disks used for data storage in digital computers are known as CD-ROM, whose diameter is 5.25 inches. It can store around 650MB. Information in CD-ROM is written by creating pits on the disk surface by shining a laser beam. As the disk rotates the laser beam traces out a continuous spiral. The focused beam creates a circular pit of around 0.8-micrometer diameter wherever a 1 is to be written and no pits (also called a land) if a 0 is to be written. Figure 5 shows the CD-ROM & DVD-ROM.

The CD-ROM with pre-recorded information is read by a CD-ROM reader which uses a laser beam for reading. It is rotated by a motor at a speed of 360 RPM. A laser head moves in and out to the specified position. As the disk rotates the head senses pits and land. This is converted to 1s and 0s by the electronic interface and sent to the computer. The disk speed of CD-ROM is indicated by the notation nx , where n is an integer indicating the factor by which the original speed of 150KB/s is to be multiplied. It is connected to a computer by SCSI and IDE interfaces. The major application of CD-ROM is in distributing large text, audio and video. For example, the entire Encyclopedia could be stored in one CD-ROM. A 640MB CD-ROM can store 74 min. of music.

The main **advantages** of CD-ROMs are:

- large storage capacity
- mass replication is inexpensive and fast
- these are removable disks, thus they are suitable for archival storage.

The main **disadvantages** of CD-ROMs are:

- it is read only, therefore, cannot be updated
- access time is longer than that of a magnetic disks.
- very slow as compared to hard disks, i.e., the normal transfer rate is 300 Mbps for double speed drives and 600 Mbps for quadruple speed drives.

2. **DVD-ROM (Digital Versatile Disk Read Only Memory):** DVD-ROM uses the same principle as a CD-ROM for reading and writing. However, a smaller wavelength laser beam is used. The total capacity of DVD-ROM is 8.5GB. In double-sided DVD-ROM two such disks are stuck back to back which allows recording on both sides. This requires the disk to be reversed to read the reverse side. With both side recording and with each side storing 8.5GB the total capacity is 17GB.

In both CD-ROMs and DVD-ROMs, the density of data stored is constant throughout the spiral track. In order to obtain a constant readout rate the disk must rotate faster, near the center and slower at the outer tracks to maintain a constant linear velocity (CLV) between the head and the CD-ROM/DVD-ROM platter. Thus CLV disks are rotated at variable speed. Compare it with the mechanism of constant angular velocity (CAV) in which disk is rotated at a constant speed. Thus, in CAV the density of information storage on outside sectors is low.

The main advantage of having CAV is that individual blocks of data can be accessed at semi-random mode. Thus the head can be moved from its current location to a desired track and one waits for the specific sector to spin under it.

The main disadvantage of CAV disk is that a lot of storage space is wasted, since the longer outer tracks are storing the data only equal to that of the shorter innermost track. Because of this disadvantage, the CAV method is not recommended for use on CD ROMs and DVD-ROMs.

Comparison of CD-ROM and DVD-ROM

<u>Characteristics</u>	<u>CD-ROM</u>	<u>DVD-ROM</u>
Pit length(micron)	0.834	0.4
Track pitch(micron)	1.6	0.74
Laser beam wave-length(nanometer)	635	780

Capacity	
1 layer/1 side	650MB
4.7GB	
2 layers/1 side	NO
8.5GB	
1 layer/2sides	NO
9.4GB	
2 layers/2 sides	NO
17GB	
Speed 1x	150KB/s
1.38MB/s	

Check Your Progress 2

1. Compare and contrast the difference between CD-ROM and DVD-ROM.

.....

2. List the advantage and disadvantage of CD-ROM.

.....

3.5 REMOVABLE STORAGE OPTIONS

3.5.1 Zip, Jaz and Other Cartridge Drives

Zip Drive: Volumes of data, loads of files have definitely increased the onus on today's computer user, and the protection of this data is what bugs each one of us. The Zip drive is a special high-capacity disk drive that uses a 3.5-inch Zip disk which can store 100MB of data. It allows an easy and rapid shift of the data from desktop to laptop. The user can also connect to notebooks running Windows 95/Windows 98/Windows ME via a Zip cable. The latest addition of Zip drive is Iomega's Zip 250MB drive that can store 250MB of data.

Jaz Drive: The Jaz drive is a popular drive with 2GB and unleashes the creativity of professionals in the graphic design and publishing, software development, 3D CAD/CAM, enterprise management systems and entertainment authorizing markets by giving them unlimited space for dynamic digital content. It has an impressive sustained transfer rate of 8.0 MB/s - fast enough to run applications or deliver full-screen, full-motion video. It is compatible with both Windows (95/98/NT 4.0 & 2000) & MAC OS 8.1 through 9.x.

Cartridge Drive: A cartridge is a protective case or covering, used to hold a disk, magnetic tape, a printer ribbon or toner. The contents are sealed inside a plastic container so that they cannot be damaged.

Disk Cartridges: Removable disk cartridges are an alternative to hard disk units as a form of secondary storage. The cartridge normally contains one or two platters enclosed in a hard plastic case that is inserted into the disk drive much like a music tape cassette. The capacity of these cartridges ranges from 5MB to more than 60MB, somewhat lower than hard disk units but still substantially superior to diskettes. They are handy because they give microcomputer users access to amounts of data limited only by the number of cartridges used.

Quarter Inch Cartridge Tapes (QIC Standard): These tape cartridges record information serially in a track with one head. When the end of the tape is reached the tape is rewound and data is recorder on the next track. There are 9 to 30 tracks. Data bits are serial on a track and blocks of around 6000 bytes are written followed by error-correction code to enable correction of data on reading if any error occurs. The density of data is around 16000 bits per inch in modern tapes. The tapes store around 500 MB. The cassette size is 5.25 inch just like a floppy and mounted in a slot provided on the front panel of a computer. The tape read/write speed is around 120 inch/second and data are transferred at the rate of 240KB/s.

These tapes are normally interfaced to a computer using the SCSI standard. The data formats used in these tapes are called QIC standard.

Tape drive	Capacity MB	Transfer rate(KB/S)	read/write speed(KB/S)	Main application
QIC DEC TZK 10	525	240	120	Backup, archiving
QIC DEC TK50	95	62.5	75	-do-
QIC TS/1000	1000	300	66	-do-
DAI DELTLZ06	4000	366	1GB/Hour	-do-

3.5.2 Recordable CDs and DVDs

The optical disk becomes an attractive alternative for backing up information from hard disk drives, and uses large text, audio and video data. The advent of CD-ROM-R and DVD-ROM-R has broken the monopoly of tapes for backing up files from hard disks with enormous storage capacities in GB range.

3.5.3 CD-R vs CD RW

A CD-R disc looks like a CD. Although all pressed CDs are silver, CD-R discs are gold or silver on their label side and a deep green or cyan on the recordable side. The silver/cyan CD-Rs were created because the green dye used in the original CD-R does not reflect the shorter-wavelength red lasers used in new DVD drives. The cyan dye used in the CD-R will allow complete compatibility with DVD drives. The CD-R disc has four layers instead of three for a CD. At the lowest level, the laser light suffices to detect the presence or absence of pits or marks on the recording surface to read the disc. At the higher level, it can actually burn marks into the surface.

CD-RW is relatively new technology, but it has been gaining market share quite rapidly. The drives cost little more than CD-R drives because they can be used to play audio CDs and CD-ROMs as well as playing and recording CD-RW discs. A CD-RW disc contains two more layers than a CD-R. The difference is that the recordable layer is made of a special material, an alloy of several metals.

Iomega Corporation has announced a CD-RW drive, the **Iomega 48*24*48 USB 2.0 external CD-RW drive**. These drive features buffers under run protection, which list user's record safely, even while multitasking. It offers plug-&-play capability with Microsoft Windows & Mac OS operating systems and its digital audio extraction rate (DAE) of 48x allows users to rep or burn a 60-min CD in under 3 min., while maximum drive speed is attainable only with hi-speed USB 2.0 connections.

3.5.4 Tape Backup

Magnetic tapes are used nowadays in computers for the following purposes:

- Backing up data stored in disks. It is necessary to regularly save data stored on disk in another medium so that if accidentally the data on disk are overwritten or if data get corrupted due to hardware failure, the saved data may be written back on the disk.
- Storing processed data for future use. This is called archiving.
- Program and data interchange between organizations.

Comparative Characteristics of Secondary Memories
(Please note that this may change with technology advancements /time)

Memory Type	Average Capacity in byte	Technology	Average time to access a byte	Permanence of storage	Access mode	Purpose in computer system	Relative cost per byte in units
Hard disk	50 GB	Magnetic surges on hard disks	10 msec	Non-volatile	Direct	Large data files and program overflow from main memory	1/100
Floppy disk	10 MB	Magnetic surges on hard disks	500 msec	Non-volatile	Direct	Data entry. As input unit	1/1000
Main memory	50 MB	Integrated circuits	20 nsec	Volatile	Random	Program and data	1
Cache memory	0.5 MB	High speed integrated circuits	2 nsec	Non-volatile	Direct	Instructions and data to be immediately used	10
CD-ROM	650 MB	Laser Disk	500 msec	Non-volatile	Direct	Store large text, pictures and audio. Software distribution	1/10000
DVD-ROM	8.5 GB	Laser Disk	500 msec	Non-volatile	Direct	Video files	1/100000
Magnetic tape	5 GB	Long ¼"	25 sec	Non-volatile	Sequential	Historical files. Backup for disk	1/1000

Digital Audio Tape (DAT): The most appropriate tape for backing up data from a disk today is Digital Audio Tape (DAT). It uses a 4mm tape enclosed in a cartridge. It uses a helical scan, read after write recording technique, which provides reliable data recording. The head spins at a high speed while the tape moves. Very high recording densities are obtained. It uses a recording format called Digital Data Storage (DDS), which provides three levels of error correcting code to ensure excellent data integrity. The capacity is up to 4GB with a data transfer speed of 366KB/sec. This tape uses SCSI interface.

Check Your Progress 3

1. What is the difference between CD-R and CD-RW?

2. State True or False:

- | | |
|--|-----|
| (a) Zip drive can be used for storing 10 MB data. | T/F |
| (b) QIC standard Cartridges have 40 tracks | T/F |
| (c) DAT is a useful backing store technology | T/F |
| (d) DVD-ROMs are preferred for data storage over CD-ROMs | T/F |
| (e) Magnetic tape is faster than CD-ROM. | T/F |

3.6 SUMMARY

In this unit, we have discussed the characteristics of different secondary storage drives, their drive speed, access time, rotation speed, density, etc. We also describe the low-level and high-level formatting of a blank disk and also the use of disk partitioning. We have also learnt to distinguish among the various types of drives, i.e., hard drives, optical drives, removable drives and cartridge drive, the hard drive interfaces, removable drives and non-removable drives. This unit also described the different types of disk formats. The advanced technologies of optical memories such as CD-ROM, DVD-ROM, CD-R, CD-RW, etc., and backups for storage such as DAT are also discussed in this unit.

3.7 SOLUTIONS /ANSWERS

Check Your Progress 1

1. Access time is 'seek time' plus 'latency time'. Seek time is the time taken by read-write head to get into the right track. Latency time is the time taken by read-write head to position itself in the right sector. Here a track has 300 sectors. So on an average to position in the right word the read-write head should traverse 150 words. Time taken for this will be $150 / (30 \times 300)$ second = 17 ms (approximately). So the access time will be $30 + 17 = 47$ ms.
2. Cluster size is $4 \times 512 = 2048$ bytes
Number of clusters = $30 \times 1,000,000 / 2048 = 14648$ approx.
3. SCSI is a port or rather an I/O Bus that is used for interfacing many devices like disk drives, printers, etc to computer. SCSI interfaces provide data transmission rates up to 80 Mbits per second. It is an ANSI standard also. EISA (**E**xtended **I**ndustry **S**tandard **A**rchitecture) is used for connecting peripherals such as mouse etc. ATA (**A**dvanced **T**echnology **A**ttachment) is a disk drive that integrates the controller on the disk drive itself. IDE (**I**ntegrated **D**rive **E**lectronics) is an interface for mass storage devices that integrates the controller into the disk or CD-ROM drive.

Check Your Progress 2

1. A CD-ROM is a non-erasable disk used for storing computer data. The standard uses 12 cm disk and can hold more than 650 MB.
A DVD-ROM is used for providing digitized compressed representation of video as well as the large volume of digital data. Both 8 and 12 cm diameters are used with a double sided capacity of up to 17GB.
2. The advantages of CD-ROM are:

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

Check Your Progress 3

1. A CD-R is similar to a CD-ROM but the user can write to the disk only once. A CD-RW is also similar to a CD-ROM but the user can erase and rewrite to the disk multiple times.
2. (a) False (b) False (c) True (d) False (e) False.

UNIT 4 I/O TECHNOLOGY

Structure	Page No.
4.0 Introduction	80
4.1 Objectives	81
4.2 Keyboard	81
4.2.1 Keyboard Layout	
4.2.2 Keyboard Touch	
4.2.3 Keyboard Technology	
4.3 Mouse	85
4.4 Video Cards	87
4.4.1 Resolution	
4.4.2 Colour Depth	
4.4.3 Video Memory	
4.4.4 Refresh Rates	
4.4.5 Graphic Accelerators and 3-D Accelerators	
4.4.6 Video Card Interfaces	
4.5 Monitors	92
4.5.1 Cathode Ray Tubes	
4.5.2 Shadow Mask	
4.5.3 Dot Pitch	
4.5.4 Monitor Resolutions	
4.5.5 DPI	
4.5.6 Interlacing	
4.5.7 Bandwidth	
4.6 Liquid Crystal Displays (LCD)	95
4.7 Digital Camera	96
4.8 Sound Cards	96
4.9 Printers	97
4.9.1 Classification of Printers	
4.9.2 Print Resolutions	
4.9.3 Print Speed	
4.9.4 Print Quality	
4.9.5 Colour Management	
4.10 Modems	99
4.11 Scanners	100
4.11.1 Resolution	
4.11.2 Dynamic Range/Colour Depth	
4.11.3 Size and Speed	
4.11.4 Scanning Tips	
4.12 Power Supply	102
SMPS (Switched Mode Power Supply)	
4.13 Summary	104
4.14 Solutions /Answers	104
References	

4.0 INTRODUCTION

In the previous units you have been exposed to Input/Output interfaces, control and techniques etc. This unit covers Input/Output devices and technologies related to them. The basic aspects covered include:

- The characteristics of the Device.
- How does it function?
- How does it relate with the Main computing unit?

4.1 OBJECTIVES

After going through this unit you will be able to:

- describe the characteristics, types, functioning and interfacing of Keyboards;
- describe the characteristics, technology and working of Mice;
- describe characteristics, technology and working of Video Cards including various parameters, Video Memory, interfaces and Graphic accelerators;
- describe the characteristics, technology and working of Monitors;
- describe the characteristics, technology and working of Liquid Crystal Displays (LCDs), and Video Cameras;
- describe the characteristics, technology and working of Sound Cards;
- describe the characteristics, technology and working of Printers;
- describe the characteristics, technology and working of Modems;
- describe the characteristics, technology and working of Scanners; and
- describe the the purpose of Power Supply and explain SMPS.

4.2 KEYBOARD

The keyboard is the main input device for your computer. It is a fast and accurate device. The multiple character keys allow you to send data to your computer as a stream of characters in a serial manner. The keyboard is one device which can be used in public spaces or offices where privacy is not ensured. The keyboard is efficient in jobs like data entry. The keyboard is one device which shall stay on for years to come, probably even after powerful voice-based input devices have been developed.

The precursor of the keyboard was the mechanical typewriter, hence it has inherited many of the properties of the typewriter.

The Keys

A full size keyboard has the distance between the centres of the keycaps (keys) as 19mm (0.75in). The keycaps have a top of about 0.5in (12.5in) which is shaped as a sort of dish to help you place your finger. Most designs have the keys curved in a concave cylindrical shape on the top.

4.2.1 Keyboard Layout

A keyboard layout is the arrangement of the array of keys across the keyboard. There is one keyboard layout that anybody who has worked on a standard keyboard or typewriter is familiar with; that layout is QWERTY. However, there are other less popular layouts also.

QWERTY

q,w,e,r,t,y are the first six letters of the top row of the alphabets of the QWERTY layout. The QWERTY arrangement was given by Sholes, the inventor of the typewriter. The first typewriter that Sholes created had an alphabetic layout of keys. However, very soon Sholes designed QWERTY as a superior arrangement though he gave no record of how he came upon this arrangement.

QWERTY-based keyboards

Besides the standard alphabet keys having the QWERTY arrangement, a computer keyboard also consists of the control (alt, Del, Ctrl etc. keys), the function keys (F1, F2 .. etc.), the numerical keypad etc.

PC 83-key and AT 84-key Keyboards

The PC 83-key was the earliest keyboard offered by IBM with its first Personal Computers (PC). This had 83 keys. Later IBM added one more key with its PC AT computer keyboards to make it a 84-key keyboard. The special feature of these keyboards was that they had function keys in two columns on the left side of the keyboard.

101-key Enhanced Keyboard

With its newer range of PCs IBM introduced the 101-key Enhanced/Advanced keyboard. This keyboard is the basic keyboard behind modern QWERTY keyboards. This has the function keys aligned in a separate row at the top of the PC, to

Figure 1: IBM 101-key Keyboard layout

correspond to the function keys shown by many software on the monitor. However, this has also been criticised at times for having a small enter key and function keys on the top! ! ! .

Windows 104-key keyboard

This is enhancements of the 101-key keyboard with special keys for Windows functions and popup. Individual vendors sometimes make changes to the basic keyboard design, for example by having a larger enter key.

Dvorak-Dealey keyboard

This was one keyboard layout designed to be a challenger to the QWERTY layout. This was designed by August Dvorak and William Dealey after much scientific research in 1936. This layout tries to make typing faster. The basic strategy it tries to incorporate is called *hand alteration*. *Hand alteration* implies that if you press one key with the left hand, the next key is likely to be pressed by the right hand, thus speeding up typing (assuming you type with both hands).

Figure 2: Dvorak keyboard layout

However, the Dvorak has not been able to compete with QWERTY and almost all systems now come with QWERTY 101-key or 104-key based keyboards. Still, there may be a possibility of designing new keyboards for specific areas, say, for Indian scripts.

4.2.2 Keyboard Touch

When using a keyboard, the most important factor is the feel of the keyboard, i.e., how typing feels on that particular keyboard. The keyboard must respond properly to your keypress. This not only means that keys must go down when pressed and then come up but also that there must be a certain feedback to your fingers when a key gets activated. This is necessary for you to develop faith in the keyboard and allow fast, reliable typing.

Linear travel or linear touch keyboards increase resistance linearly with the travel of the key. Therefore, you have to press harder as the key goes lower. There can be audible feedback as a click and visual feedback as the appearance of a character on screen letting you know when a key gets activated. Better keyboards provide tactile feedback (to your fingers) but suddenly reducing resistance when the key gets actuated. This is called an over-center feel. Such keyboards are best for quick touch typing. These were implemented by using springs earlier but now they are usually elastic rubber domes. Keyboards also differ in whether they ‘click’ or not (soundless), on the force required and the key travel distance to actuate a key. The choice is usually an issue of personal liking. Laptops usually have short travel keys to save space which is at a premium in laptops.

4.2.3 Keyboard Technology

Each key of a keyboard is like an electric switch changing the flow of electricity in some way. There are two main types — capacitor-based and contact-based keyboards.

Capacitor-Based Keyboards

These keyboards are based on the concept of Capacitance. A simple capacitor consists of a pair of conductive plates having opposite charges and separated by an insulator. This arrangement generates a field between the plates proportional to the closeness of the plates. Changing the distance between the plates causes current to flow. Capacitive keyboards have etched circuit boards, with tin and nickel-plated copper pads acting as capacitors under each key (a key is technically called a station). Each key press presses a small metal-plastic circle down causing electric flow. These keyboards work

well but have the drawback that they follow an indirect approach though they have a longer life than contact-based keyboards. These keyboards were introduced by IBM.

Contact-Based Keyboards

Contact-based keyboards use switches directly. Though they have a comparatively shorter life, they are the most preferred kind nowadays due to their lower cost. Three such kinds of keyboards have been used in PCs:

1. **Mechanical Switches:** These keyboards use traditional switches with the metal contacts directly touching each other. Springs and other parts are used to control positioning of the keycaps and give the right feel. Overall, this design is not suited to PC keyboards.
2. **Rubber Dome:** In rubber dome keyboards, both contact and positioning are controlled by a puckered sheet of *elastomer*, which is a stretchy, rubber-like synthetic material. This sheet is moulded to have a dimple or dome in each keycap. The dome houses a tab of carbon or other conductive material which serves as a contact. When a key is pressed, the dome presses down to touch another contact and complete the circuit. The elastomer then pushes the key back. This is the most popular PC keyboard design since the domes are inexpensive and proper design can give the keyboards an excellent feel.
3. **Membrane:** These are similar to rubber domes except that they use thin plastic sheets (membranes) with conductive traces on them. The contacts are in the form of dimples which are plucked together when a key is pressed. This design is often used in calculators and printer keyboards due to their low cost and trouble-free life. However, since its contacts require only a slight travel to actuate, it makes for a poor computer keyboard.

Scan Codes

A scan code is the code generated by a microprocessor in the keyboard when a key is pressed and is unique to the key struck. When this code is received by the computer it issues an interrupt and looks up the scan code table in the BIOS and finds out which keys have been pressed and in what combination. Special memory locations called *status bytes* tell the status of the locking and toggle keys, e.g., Caps lock etc. Each keypress generates two different scan codes — one on key-push down called Make code, another on its popping back called Break code. This two-key technique allows the computer to tell when a key is held pressed down, e.g., the ALT key while pressing another key, say, CTRL-ALT-DEL.

There are three standards for scan codes: Mode1 (83-key keyboard PC, PC-XT), Mode2 (84-key AT keyboard), Mode3 (101-key keyboard onwards). In Mode1 Make and Break codes are both single bytes but different for the same key. In Mode2 and Mode3, Make code is a single byte and Break code is two bytes (byte F0(Hex) + the make code).

Interfacing

The keyboard uses a special I/O port that is like a serial port but does not explicitly follow the RS-232 serial port standard. Instead of multiple data and handshaking signals as in RS-232, the keyboard uses only two signals, through which it manages a bi-directional interface with its own set of commands.

Using its elaborate handshaking mechanism, the keyboard and the PC send commands and data to each other. The USB keyboards work differently by using the USB coding and protocol.

OPERATOR

— means dash which is longer.
- means hyphen which is shorter.

Table 1: Some Scan Codes

Mode1				Mode2	Mode 3
Key	KeyNo.	Make	Break	Make	Break
A	31	1E	9E	1C	F0 1C
0	11	0B	8B	45	F0 45
Enter	43	1C	9C	5A	F0 5A
Left Shift	44	2A	AA	12	F0 12
F1	112	3B	BB	07	F0 07

Connections

5-pin DIN connector: This is the connector of the conventional keyboard having 5 pins (2 IN, 2 OUT and one ground pin), used for synchronization and transfer.

PS/2 connector (PS/2 keyboards): These were introduced with IBM's PS/2 computers and hence are called PS/2 connectors. They have 6-pins but in fact their wiring is simply a rearrangement of the 5-pin DIN connector. This connector is smaller in size and quite popular nowadays. Due to the similar wiring, a 5-pin DIN can easily be connected to a PS/2 connector via a simple adapter.

Ergonomic Keyboards

Ergonomics is the study of the environment, conditions and efficiency of workers¹. Ergonomics suggests that the keyboard was not designed with human beings in mind. Indeed, continuous typing can be hazardous to health. This can lead to pain or some ailments like the Carpal Tunnel Syndrome.

For normal typing on a keyboard, you have to place your hands apart, bending them at the wrists and hold this position for a long time. You also have to bend your wrist vertically especially if you elevate your keyboard using the little feet behind the keyboards. This stresses the wrist ligaments and squeezes the nerves running into the hand through the Carpal tunnel, through the wrist bones.

To reduce the stress, keyboards called ergonomic keyboards have been designed. These split the keyboard into two and angle the two halves so as to keep the wrists straight. To reduce vertical stress, many keyboards also provide extended wrist rests. For those who indulge in heavy, regular typing, it is recommended that they use more ergonomics based keyboards and follow ergonomic advice in all aspects of their workplace.

4.3 MOUSE

The idea of the Mouse was developed by Douglas C. Engelbart of Stanford Research institute, and the first Mouse was developed by Xerox corporation. Mouse itself is a device which gives you a pointer on screen and a method of selection of commands through buttons on the top. A single button is usually sufficient (as in Mouse with Apple Macintosh machines) but Mice come with upto 3 buttons.

Types of Mice

Mice can be classified on the basis of the numbers of buttons, position sensing technology or the type of Interface:

¹Oxford Advanced Learner's Dictionary

Sensing Technology

The Mice can be Mechanical or Optical.

Mechanical Mice have a ball made from rough rubbery material, the rotation of which effects sensors that are perpendicular to each other. Thus, the motion of the ball along the two axes is detected and reflected as the motion of the pointer on the screen.

Optical Mice can detect movement without any moving parts like a ball. The typical optical Mouse used to have a pair of LEDs (Light Emitting Diodes) and photo-detectors in each axis and its own Mousepad on which it is slid. However, due to the maintenance needs of the Mousepad, this was not very successful. Recently, optical Mice have made a comeback since they can now operate without a Mousepad.

Interface

Mouse is usually a serial device connected to a serial port(RS232), but these connections can themselves take various forms:

Serial Mouse

Mice that use the standard serial port are called “serial”. Since Serial ports 1 and 4 (COM1, COM4 under DOS, /dev/ttyS0 and /dev/ttyS3 under Unix/GNU-Linux systems) and ports 2 and 3 (COM2, COM3 or /dev/ttyS1/dev/ttyS2) share the same interrupts respectively, one should be careful not to attach the mouse so that it shares the interrupt with another device in operation like a modem.

Bus Mouse

These Mice have a dedicated Mouse card and port to connect to. Recently, USB mouse has become popular.

Proprietary

Mouse ports specific to some PCs e.g., IBM’s PS/2 and some Compaq computers.

Mouse Protocols

The mouse protocol is the digital code to which the signal from the mouse gets converted. There are four major protocols: Microsoft, Mouse Systems Corporation(MSC), Logitech and IBM. Most mice available do support at least the Microsoft protocol or its emulation.

Resolution versus Accuracy

Resolution of mouse is given in CPI(Counts per Inch) i.e. the number of signals per inch of travel. This means the mouse will move faster on the screen but it also means that it will be more difficult to control the accuracy.

Check Your Progress 1

1. Discuss the merits and demerits of Dvorak-Dealey keyboard vs. QWERTY keyboard.

.....

.....

.....

.....

.....

2. Why is keyboard touch important? What kind of touch would you prefer and which kind of keyboard will give that touch?

.....

.....

.....

.....

.....

3. What precautions should be taken while attaching a Serial Mouse?

.....

.....

.....

.....

.....

4. You enter 'a' as left-shift + 'A' ? What will be the scan-code generated in Mode-3 by the keyboard?

- | | |
|-----------------|-----------------|
| a) 2A1E9EAA | b) 1CF01C |
| c) 121CF01CF012 | d) 1CF01C5AF05A |

4.4 VIDEO CARDS

Before discussing in detail video hardware, let us have a brief overview of graphic display technology. The purpose of your graphic display system is to display bit-mapped graphics on your monitor. The image displayed on your system thus consists of small dots called pixels (short for 'picture elements') and your video system contains a description of each of these dots in the memory. At any moment, the display memory contains the exact bit-map representation of your screen image and what is coming next. This is like a time-slice of what you see on your monitor. Therefore, display memory is also called a framebuffer. These frames are read dozens of times a second and sent in a serial manner through a cable to the monitor. The monitor receives the train of data and displays it on the screen. This happens by a scanning raster movement from up to down one row at a time. A CRT (Cathode Ray Tube) based monitor will light its small phosphor dots according to this raster movement. In this respect, it is like a television, which is also a CRT based device.

The more the number of dots, i.e., the higher the resolution of the image, the sharper the picture is. The richness of the image is also dependant on the number of colours (or gray levels for a monochrome display) displayed by the system. The higher the number of colours, the more is the information required for each dot. Hence, the amount of memory (framebuffer) required by a system is directly dependent on the resolution and colour depth required.

4.4.1 Resolution

Resolution is the parameter that defines the possible sharpness or clarity of a video image. Resolution is defined as the number of pixels that make up an image. These pixels are then spread across the width and height of the monitor. Resolution is independent of the physical characteristics of the monitor. The image is generated without considering the ultimate screen it is to be displayed upon. Hence, the unit of resolution is the number of pixels, not the number of pixels per inch. For example, a standard VGA native graphic display mode has a resolution of 640 pixels horizontally by 480 pixels vertically. Higher resolutions mean the image can be sharper because it contains more pixels.

The actual on-screen sharpness is given as dots-per-inch, and this depends on both the resolution and the size of the image. For the same resolution, an image will be sharper on a smaller screen, i.e., an image which may look sharp on a 15" monitor may be a little jagged on a 17" display.

4.4.2 Colour Depth

It is clear that an image consists of an array of pixels. If we tell which pixels are 'on' and which are 'off' to the monitor, it should be able to display the image as a pure black and white image. But what about Colour and Contrast? Clearly, if only a single bit is assigned to a pixel, we cannot give any additional quality to the image. It will look like a black and white line drawing. Such a system is typically called a two-colour system. Such black and white picture can be converted to gray levels by assigning more bits, e.g., with two bits we can get the following levels: White, Light Gray, Dark Gray and Black.

To add colour to an image, we have to store colour of the pixel with each pixel. This is usually stored as intensity measures of the primary light colours — Red, Green and Blue. That means we have to assign more than 1 bit to describe a pixel. Hence, 1 bit per pixel implies 2 colours or 2 gray-levels, 2 bits per pixel 4 colours or 4 gray-levels and so n bits per pixel means a display of colours or gray-levels is possible.

Colour Depth (or the number of *Colour Planes*) is the number of bits assigned to each pixel to code colour information in it. These are also called *Colour Planes* because each bit of a pixel represents a specific colour and the bit at the same position on every pixel represents the same colour. Hence, the bits at the same position can be thought of as forming a plane of a particular colour shade and these planes piled on top of each other give the final colour at each point. Thus, if each pixel is described by 3 bits, one each for red, green and blue colour, then, there are 3 *Colour Planes* (one each for red, green and blue) and 6 colour planes if there are 6 bits — see Figure 4.

What Colour depths are practically used?

Practically, the number of colours are an exponential power of 2, since for *Colour Depth* n , colours can be displayed. The most popular colour modes are given in Table 2.

	Colour Mode	Depth(bits/pixel)
1	Monochrome	1
2	16-Colours	4
3	256-Colours	8
4	High Color	16
5	True Color	24

Table 2: Major Colour Depths

The Bad News

The bad news is that most monitors can only display upto a maximum of 262,144 colours (= i.e. 18 bits/pixel *Colour Depth*). The other bad news is that the human eye can only perceive a few million colours at the most. So, even if you had lots of bits per pixel and very advanced display systems, it would be useless. Maybe, this is good news rather than bad news for the hardware developer!

This also implies that 24-bit colour bit-depth is the practical upper limit. Hence, this depth is also called true colour because with this depth the system stores more colours than can ever be seen by the human eye and, hence, it is a true colour representation of the image. *Though, 24-bit colour or true colour systems have more colour than possibly useful, they are convenient for designers because they assign 1 byte of storage for each of the three additive primary colours (red, green and blue).* Some new systems even have 32 bits per pixel. Why? Actually, the additional bits are not used to hold colours but something called an *Alpha Channel*. This 8-bit *Alpha Channel* stores special effect information for the image.

Why are all resolutions in the ratio of 4:3? The answer you'll find in a later section.

4.4.3 Video Memory

As stated before, video memory is also called framebuffer because it buffers video frames to be displayed. The quality of a video display depends a lot on how quickly can the framebuffer be accessed and be updated by the video system. In early video systems, video memory was just a fixed area of the system RAM. Later, there was video RAM which came with the video cards themselves and could be increased by putting additional video RAM under the UMA (Unified Memory Architecture). Video RAM is again part of the system RAM. UMA is what you get in the modern low-cost motherboards with on-board video and sound cards etc.

The amount of video memory required is dependant on the resolution and colour-depth required of the system. Let us see how to calculate the amount of video memory required. The video memory required is simply the resolution (i.e., the total number of pixels) multiplied by the Colour Depth. Let us do the calculations for a standard VGA graphics screen (640 × 480) using 16 colours.

Total number of Pixels = 640 × 480 =	307200	
Colour Depth (16-colours) =	4	bits
Total minimum Memory =	1,228,800	bits
Total minimum memory (in bytes) =	153,600	bytes
	153 KB	

Minimum Video RAM required and available = 256 KB.

Therefore, 16-colour VGA needs at least 153,600 bytes of memory but memory is only available in exponential powers of 2, hence, the next highest available memory is = 256 KB.

What is a good resolution? Actually, it depends on your hardware. So, it is the maximum your hardware can allow you. However, one odd-looking resolution which has become popular is 1152×864 pixels. Can you judge why this should be so? (Hint: Think of this resolution at 8-bit colour).

If you can't wait any longer, here is the answer: 1152×864 is nearly one million pixels. Since 8-bit colour depth means 8 million bits or 1 MB. This is the highest resolution you can get in 1 MB video memory at 8-bit colour depth, plus this still leaves you square pixels (in the ratio 4: 3) to allow easy programming.

The above calculations hold good for only two-dimensional display systems. This is because 3-D systems require much more memory because of techniques such as "Double Buffering" and "Z-Buffering".

4.4.4 Refresh Rates

A special circuit called the Video Controller scans the video memory one row at a time and reads data value at each address sending the data out in a serial data stream. This data is displayed by a process called *Scanning* where the electron beam is swept across the screen one-line-at-a-time and left-to-right. This is controlled by a vertical and a horizontal field generated by electromagnets — one moving the beam horizontally and another vertically.

The rate at which horizontal sweeps take place is called horizontal frequency or horizontal refresh rate and the rate at which vertical sweeps take place are called vertical frequency or vertical refresh rate or simply refresh rate or frame rate. The term frame rate is used because actually one vertical sweep means display of a single frame. Since each frame contains several hundred rows, horizontal frequency is hundreds of times higher than vertical frequency. Therefore, the unit of horizontal frequency is KHz and that of vertical frequency is Hz.

The most important thing is maintaining the same frequencies between the Video system and monitor. The monitor must support these refresh rates, hence the supported refresh rates are given with the manual of the monitor. More about this topic will be discussed in the section on Monitors.

4.4.5 Graphic Accelerators and 3-D Accelerators

A Graphic Accelerator is actually a chip, in fact the most important chip in your video card. The Graphic Accelerator is actually the modern development of a much older technology called the *Graphic Co-Processor*. The accelerator chip is actually a chip that has built-in video functions. These functions execute the algorithms for image construction and rendering. It does a lot of work which would otherwise have to be done by the microprocessor. Hence, the accelerator chip is actually optional but very important for good graphics performance.

The graphic accelerator determines whether your system can show 3-D graphics, how quickly your system displays a drop-down menu, how good is your video playback, etc. It determines the amount and kind of memory in the framebuffer and also the resolution your PC can display.

The first major graphic accelerators were made by the S3 corporation. Modern Graphic accelerators have internal registers at least 64-bit wide to work on at least 2 pixels at a time. They can use the standard Dynamic RAM (DRAM) or the more expensive but faster dual-ported Video RAM (VRAM). They support at least the standard resolutions up to 1024×768 pixels. They often use RAMDACs for colour support giving full 24-bit or 32-bit colour support. A RAMDAC (Random Access

Memory Digital-to-Analog Converter) is a microchip that converts digital image data into the analog data needed by a computer display. However, the higher the resolution required, the higher is the speed at which the chip has to function. So, for a resolution of 1280×1024 , the chip operates at 100 MHz. At the cutting edge of technology, chips now run even as fast as 180 or 200 Mhz.

What is a 3-D Accelerator?

3-D Accelerator is no magic technology. It is simply an accelerator chip that has built-in ability to carry out the mathematics and the algorithms required for 3-D image generation and rendering. A 3-D imaging is simply an illusion, a projection of 3-D reality on a 2-D screen. These are generated by projection and perspective effects, depth and lighting effects, transparency effects and techniques such as Ray-Tracing (Tracing the path of light rays emitting from a light source), Z-buffering (a buffer storing the Z-axis positions) and Double-Buffering (two buffers instead of one).

4.4.6 Video Card Interfaces

A video interface is the link of the video system to the rest of the PC. To enhance video performance, there is sought to be an intimate connection between the microprocessor and the video system, especially the framebuffer. In modern displays, only in the UMA system is the framebuffer actually a part of the main memory; in the rest the connection is through a bus, which may be PCI or AGP. Let us briefly discuss these interfaces:

PCI

PCI stands for Peripheral Connect Interface. It is the revolutionary high speed expansion bus introduced by Intel. With the growing importance of video, video cards were shifted to PCI from slower interfaces like ISA. The PCI standard has now developed into the even more powerful AGP.

AGP

AGP stands for Advanced (or Accelerated) Graphics Port. It is a connector standard describing a high speed bus connection between the PC video system, the microprocessor and the main memory. It is an advancement of the PCI interface.

AGP uses concepts such as pipelining to allow powerful 3-D graphic accelerators to function when used in conjunction with fast processors. AGP uses three powerful innovations to achieve its performance:

- Pipelined Memory: The use of Pipelining eliminates wait states allowing faster operation.
- Seperate Address and Data Lines.
- High speeds through a special 2X mode that allows running AGP at 133 MHz instead of the default 66 MHz.

Through AGP, the video board has a direct connection to the microprocessor as a dedicated high speed interface for video. The system uses DMA (Direct Memory Access) to move data between main memory and framebuffer. The accelerator chip uses the main memory for execution of high level functions like those used in 3-D rendering.

UMA

UMA stands for Unified Memory Architecture. It is an architecture which reduces the cost of PC construction. In this, a part of the main memory is actually used as framebuffer. Hence, it eliminates the use of a bus for video processing. Therefore, it is less costly. Though it is not supposed to perform as well as AGP etc., in some

cases it may give a better performance than the bus-based systems. It is the interface used nowadays in low-cost motherboards.

Figure 5: AGP Video Architecture and its working

4.5 MONITORS

A Monitor is the television like box connected to your computer and giving you a vision into the mind of your PC. It shows what your computer is thinking. It has a display which is technically defined as the image-producing device, i.e., the screen one sees and a circuitry that converts the signals from your computer (or similar devices) into the proper form for display.

Monitors are or were just like television sets except that television sets have a tuner or demodulator circuit to convert the signals. However, now monitors have branched beyond television. They have greater sharpness and colour purity and operate at higher frequencies.

Generally, when you go to purchase a monitor from the market, you see the following specifications: The maximum Resolution, the Horizontal and Vertical Frequencies supported, the tube size and the connectors to the monitor. There are many vendors on the market like Samsung, LG, Sony etc. Home users generally go in for monitors of size 17", 15" or 14". Monitors are also available as the traditional curved screens, flat screens or LCD. The technology behind Monitors and the above specifications are discussed ahead.

4.5.1 Cathode Ray Tubes

Cathode ray tube is the major technology on which monitors and televisions have been based. CRT is a partially evacuated glass tube filled with inert gas at low pressure. A specially designed Cathode (negatively charged electrode) shoots beams of electrons at high speed towards an anode (positively charged electrode) which impinges on the screen which is coated with small phosphor coated dots of the three primary colours. This cathode is also called an Electron Gun. In fact, there can be three separate guns for the three colours (Red, Green and Blue) or one gun for all three.

Four factors influence the quality of image of the monitor:

1. **The Phosphor coating** : This affects the colour and the persistence (The period the effect of a single hit on a dot lasts).
2. **The Cathode (Electron Gun)** : The sharpness of the image depends on the good functioning of this gun.

3. **Shadow Mask/ Aperture Grill** : This determines the resolution of the screen in colour monitors.
4. The Screen, glare and lighting of the monitor.

4.5.2 Shadow Mask

The Shadow Mask is a metal sheet which has fine perforations (holes) in it and is located a short distance before the phosphor coated screen. The Phosphor dots and the holes in the shadow mask are so arranged that the beams from a particular gun will strike the dots of that colour only. The dots of the other two colours are in the shadow. In an attempt to overcome some shortcomings of Shadow masks due to their round holes, Sony introduced Aperture grills (in their Trinitron technology) which are slots in an array of vertically arranged wires.

Figure 6: Shadow Mask and Aperture

4.5.3 Dot Pitch

Dot Pitch of a CRT is the distance between phosphor dots of the same colour. In Trinitron screens, the term Slot Pitch is used instead of Dot Pitch — this is the distance between two slots of the same colour. Dot Pitch is a very important parameter of monitor quality. For a particular resolution, you can get the minimum dot pitch required by dividing the physical screen size by the number of pixels. Therefore, for smaller screens, you require finer Dot Pitch.

4.5.4 Monitor Resolutions

We have discussed about resolutions and vertical and horizontal refresh rates in the section on Video Cards. Let us refer to them from the monitor point of view. So, we have the following definitions (from the manual of a monitor available in the market):

Horizontal Frequency: The time to scan one line connecting the right edge to the left edge of the screen horizontally is called the Horizontal cycle and the inverse number of the Horizontal cycle is called Horizontal Frequency. The unit is KHz (KiloHertz).

Vertical Frequency: Like a Fluorescent lamp, the screen has to repeat the same image many times per second to display an image to the user. The frequency of this repetition is called Vertical Frequency or Refresh Rate.

If the resolution generated by the video card and the monitor resolution is properly matched, you get a good quality display. However, the actual resolution achieved is a physical quality of the monitor. In colour systems, the resolution is limited by Convergence (Do the beam of the 3 colours converge exactly on the same dot?) and the Dot Pitch. In monochrome monitors, the resolution is only limited by the highest frequency signals the monitor can handle.

4.5.5 DPI

DPI (Dots Per Inch) is a measure for the actual sharpness of the onscreen image. This depends on both the resolution and the size of the image. Practical experience shows that a smaller screen has a sharper image at the same resolution than does a larger screen. This is because it will require more dots per inch to display the same number of pixels. A 15-inch monitor is 12-inches horizontally. A 10-inch monitor is 8 inches horizontally. To display a VGA image (640 480) the 15-inch monitor will require 53DPI and the 10-inch monitor 80 DPI.

4.5.6 Interlacing

Interlacing is a technique in which instead of scanning the image one-line-at-a-time it is scanned alternately, i.e., alternate lines are scanned at each pass. This achieves a doubling of the frame rate with the same amount of signal input. Interlacing is used to keep bandwidth (amount of signal) down. Presently, only the 8514/A display adapters use interlacing. Since Interlaced displays have been reported to be more flickery, with better technology available, most monitors are non-interlaced now.

4.5.7 Bandwidth

Bandwidth is the amount of signal the monitor can handle and it is rated in MegaHertz. This is the most commonly quoted specification of a monitor. The Bandwidth should be enough to address each pixel plus synchronizing signals.

Check Your Progress 2

1. Redraw Figure 4 showing Colour-Planes for a true-colour system.

.....

.....

.....

.....

2. What is a FrameBuffer? Discuss the placement of the FrameBuffer w.r.t. to the different Video Card interfaces.

.....

.....

.....

.....

3. What is the difference between Shadow Mask and Dot Pitch for Trinitron and non-Trinitron monitors?
.....
.....
.....
4. How much Video-RAM would you require for a high-colour (16-bits) Colour-Depth at 1024 × 768 resolution? What would be the size of the corresponding single memory chip you would get from the market?
a) 900KB, 1MB b) 1.6 MB, 4MB
c) 12.6MB, 16MB d) 7.6MB, 8MB
5. There is an image of resolution 1024X768. It has to be displayed on a 15-inch monitor (12-inch horizontal, 9-inch Vertical display). What is the minimum Dot-pitch required for this image? (minimum here means the largest useful dot pitch).
a) 1.4×10^{-4} inches b) 2.8×10^{-4} inches
c) 1.4×10^4 inches d) 1.2×10^{-2} inches

4.6 LIQUID CRYSTAL DISPLAYS (LCD)

LCDs are the screens of choice for portable computers and lightweight screens. They consume very little electricity and have advanced technologically to quite good resolutions and colour support. They were developed by the company RCA in the 1960s. LCDs function simply by blocking available light so as to render display patterns.

LCDs can be of the following types:

1. **Reflective LCDs:** Display is generated by selectively blocking reflected light.
2. **Backlit LCDs** : Display is due to a light source behind LCD panel.
3. **Edgelit LCDs** : Display is due to a light source adjacent to the LCD panel.

LCD Technology

The technology behind LCD is called Nematic Technology because the molecules of the liquid crystals used are nematic i.e. rod-shaped. This liquid is sandwiched between two thin plastic membranes. These crystals have the special property that they can change the polarity and the bend of the light and this can be controlled by grooves in the plastic and by applying electric current.

Passive Matrix

In a passive matrix arrangement, the LCD panel has a grid of horizontal and vertical conductors and each pixel is located at an intersection. When a current is received by the pixel, it becomes dark. This is the technology which is more commonly used.

Active Matrix

This is called TFT (Thin Film Transistor) technology. In this there is a transistor at every pixel acting as a relay, receiving a small amount and making it much higher to activate the pixel. Since the amount is smaller, it can travel faster and hence response times are much faster. However, TFTs are much more difficult to fabricate and are costlier.

4.7 DIGITAL CAMERA

A Digital camera is a camera that captures and stores still images and video (Digital Video Cameras) as digital data instead of on photographic film. The first digital cameras became available in the early 1990s. Since the images are in digital form they can be later fed to a computer or printed on a printer.

Like a conventional camera, a digital camera has a series of lenses that focus light to create an image of a scene. But instead of this light hitting a piece of film, the camera focuses it on to a semiconductor device that records light electronically. An in-built computer then breaks this electronic information down into digital data.

This semiconductor device is called an Image sensor and converts light into electrical charges. There are two main kinds of Image sensors: CCD and CMOS. CCD stands for Charge coupled devices and is the more popular and more powerful kind of sensor. CMOS stands for Complementary Metal oxide semiconductor and this kind of technology is now only used in some lower end cameras. While CMOS sensors may improve and become more popular in the future, they probably won't replace CCD sensors in higher-end digital cameras.

In brief, the CCD is a collection of tiny light-sensitive diodes called photosites, which convert photons (light) into electrons (electrical charge). Each photosite is proportionally sensitive to light – the brighter the light that hits a single photosite, the greater the electrical charge that will accumulate at that site.

A digital Camera is also characterised by its resolution (like monitors and printers) which is measured in pixels. The higher the resolution, the more detail is available in an image.

Mobile Cameras

Mobile cameras are typically low-resolution Digital cameras integrated into the mobile set. The photographs are typically only good enough to show on the low resolution mobile screen. They have become quite popular devices now and the photographs taken can be used for MMS messages or uploading to a Computer.

4.8 SOUND CARDS

Multimedia has become a very important part of today's PC. The home user wants to watch movies and hear songs. The Software developer hacking away at her computer wants to have the computer playing MP3 or OGG (The latest Free Sound format standard) in the background. Thus, the sound system is a very important part of the system.

As you must have read in your high school physics, sound is a longitudinal wave travelling in a medium, usually air in the case of music. Sound can be encoded into electrical form using electrical signals which encode sound strengths. This is called analog audio. This analog audio is converted to digital audio, which is conversion of those signals into bits and bytes through the process called Sampling. In Sampling, analog 'samples' are taken at regular intervals and the amplitude (Voltage) of these samples is encoded to bits. These sounds are manipulated by your PCs microprocessor etc. To play back these digital audio sounds, the data are sent to the Sound card which converts them to analog audio, which is played back through speakers.

The Sound card (The card is often directly built into motherboards nowadays) is a board that has digital to analog sound converter, amplifier, etc., circuitry to play sound and to connect the PC to various audio sources.

A sound card may support the following functions:

1. Convert digital sound to analog form using digital-to-analog converter to play back the sound.
2. May record sound to play back later with analog-to-digital converter.
3. May have built-in Synthesizers to create new sounds.
4. May use various input sources (Microphone, CD, etc.) and mixer circuits to play these sounds together.
5. Amplifiers to amplify the sound signals to nicely audible levels.

Sound cards are described by the Compatibility, Connections and Quality.

Compatibility: Sound cards must be compatible at both hardware and software levels with industry standards. Most software, especially games, require sound cards to be compatible with the two main industry standards: AdLib (A Basic standard) and Sound Blaster (an advanced standard developed by Creative Labs).

Connections: Sound cards should have connections to allow various functions. One of the most important is the MIDI port (MIDI stands for Musical Instrument Device Interface). MIDI port allows you to create music directly with your PC using the Sound Cards synthesizer circuit and even attach a Piano keyboard to your PC.

Quality: Sound Cards vary widely in terms of the quality they give. This ranges from the frequency range support, digital quality and noise control.

4.9 PRINTERS

Printers are devices that put ink on paper in a controlled manner. They manually produce readable text or photographic images. Printers have gone through a large transition in technology. They are still available in a wide range of technology and prices from the dot matrix printer to Inkjet printers to Laser Printers.

4.9.1 Classification of Printers

Printers can be classified on the following bases:

- a) **Impact:** Impact printers print by the impact of hammers on the ribbon (e.g., Dot-Matrix Printers) whereas non-impact printers use other means (e.g., Inkjet, Laser).
- b) **Character formation:** Character formed by Bit-maps or by dots.
- c) **Output :** The quantity of output processed at a time: Serial, Line or Page Printers.

Actually, there are many specifications one has to keep in mind while purchasing a printer. Some of these are Compatibility with other hardware, in-built Memory, maximum supported memory, actual technology, Printer resolution (Colour, BW), PostScript support, output type, Printer speed, Media capacity, Weight, Height and Width of the Printer.

Let us discuss some of these parameters that characterize printers :

4.9.2 Print Resolution

Print Resolution is the detail that a printer can give determined by how many dots a printer can put per inch of paper. Thus, the unit of resolution is *Dots per inch*. This is applicable to both impact and non-impact printer though the actual quality will depend on the technology of the printer.

The required resolution to a great extent determines the quality of the output and the time taken to print it. There is a tradeoff between quality and time. Lower resolution means faster printing and low quality. High resolution means slower printing of a higher quality. There are three readymade resolution modes: draft, near letter quality (NLQ) and letter quality. Draft gives the lower resolution print and letter quality higher resolution. In Inkjet and Laser Printers, the highest mode is often called 'best' quality print.

4.9.3 Print Speed

The speed at which a printer prints is often an important issue. However, the printer has to take a certain time to print. Printing time increases with higher resolution and coloured images. To aid printing, all operating systems have spooling software that accumulates print data and sends it at the speed that the printer can print it.

The measure of speed depends on whether the printer is a **Line Printer** or **Page**

Printer. Let us understand these:

Line Printer: Line Printer processes and prints one line of text at a time.

Page Printer: A page printer processes and prints one full page at a time.

Actually, it rasterizes the full image of the page in its memory and then prints it as one line of dots at a time. For a line printer, the speed is measured in *characters per second* (cps) whereas for page printing, it is *pages per minute* (ppm). Hence, Dot Matrix usually have speeds given in *cps* whereas Lasers have speed in *ppm*. The actual speed may vary from the rating speed given by the manufacturer because, as expected, the printer chooses the more favourable values.

4.9.4 Print Quality

Print quality depends on various factors but ultimately the quality depends on the design of the printer and its mechanical construction.

DotMatrix/InkJet Printers

Three main issues determine the quality of characters produced by DotMatrix/InkJet Printers: - Number of dots in the matrix of each character, the size of the dots and the addressability of the Printer. Denser matrix and smaller dots make better characters.

Addressability is the accuracy with which a dot can be produced (e.g., 1/120 inch means printer can put a dot with 1/120 inch of the required dot). Minimum dot matrix used by general dot matrix printers is 9×9 dots, 18-pin and 24-pin printers use 12 × 24 to 24×24 matrices. Inkjets may even give up to 72×120 dots. Quality of output also depends on the paper used. If the ink of an Inkjet printer gets absorbed by the paper, it spreads and spoils the resolution.

Laser Printer

Laser Printers are page printers. For print quality, they also face the same addressability issues as DMP/InkJet Printers. However, some other techniques are possible to use for better quality here.

One of these is ReT(Resolution Enhancement Technology) introduced by Hewlett-Packard. It prints better at the same resolution by changing the size of the dots at character edges and diagonal lines reducing jagged edges.

A very important requirement for Laser Printers to print at high quality is Memory. Memory increases as a square of resolution, i.e., the Dot density, i.e., the dpi. Therefore, if 3.5 MB is required for a 600 dpi page, approximately 14 MB is required for 1200 dpi. You need even more memory for colour.

For efficient text printing, the Laser printer stores the page image as ASCII characters and fonts and prints them with low memory usage. At higher resolutions, the quality of print toner also becomes important since the resolution is limited by the size of toner particles.

4.9.5 Colour Management

There are three primary colours in pigments - *Red, Yellow and Blue*. There are two ways to produce more colours:

Physical Mixing: Physically mix colours to make a new colour. This is difficult for printers because their colours are quick drying and so colours to be mixed must be applied simultaneously.

Optical Mixing: Mixing to give the illusion of a new colour. This can be done in ways:

- Apply colours one upon another. This is done using inks which are somewhat transparent, as modern inks are.
- Applying dots of different colours so close to one another that the human eye cannot distinguish the difference. This is the theory behind **Dithering**.

3 or 4 colour Printing?

For good printing, printers do not use RBV, instead they use CMYK (Cyan instead of Blue, Magenta instead of Red, Yellow, and a separate Black). A separate Black is required since the 3 colours mixed to produce a black (which is called Composite Black) is often not satisfactory.

What is Dithering?

CMYK gives only 8 colours (C, M, Y, K, Violet= C + M, Orange= M + Y, Green = C + Y, and the colour of the paper itself!). What about other colours? For these, the technique of Dithering is used. Dithering is a method in which instead of being a single colour dot, it is a small matrix of a number of different colour dots. Such pixels are called **Super-pixels**. The dots of a given colour in a Super-pixel decide the intensity of that colour. The problem with dithering is that it reduces the resolution of the image since more dots are taken by a single pixel now.

Monitors versus Printer

Monitor screens and Printers use different colour technologies. The monitor uses RGB and the Printer CMYK. So, how does one know that the colour that is seen is going to be printed. This is where the Printer driver becomes very important, and where many computer models and graphic oriented machine score. For long, a claim to fame of the Apple Macintosh machines has been its very good correspondence between print and screen colours.

4.10 MODEMS

A Modem is one device that most computer users who have surfed the Internet are aware of. A modem is required because though most of the telecommunications have become digital, most telephone connections at the user end are still the analog POTS (Plain Old Telephone Systems/Sets/Service). However, the computer is a digital device and hence another device is needed which can convert the digital signals to analog signals and vice-versa. Such a device is the Modem.

Modem stands for Modulator/Demodulator. Modulation is the process which puts digital information on to the analog circuit by modifying a constant wave (signal) called the Carrier. This is what happens when you press a button to connect to the Internet or to a web site. Demodulation is the reverse process, which derived the digital signal from the modulated wave. This is what happens when you receive data from a website which then gets displayed by your browser.

Discussion of modulation techniques is out of scope here (you can refer to your course on Computer Networks).

Modems are available as the following types:

1. **Internal Modems:** Internal Modems plug into expansion slots in your PC. Internal Modems are cheap and efficient. Internal Modems are bus-specific and hence may not fit universally.
2. **External Modems:** Modems externally connected to PC through a serial or parallel port and into a telephone line at the other end. They can usually connect to any computer with the right port and have a range of indicators for troubleshooting.
3. **Pocket Modems:** Small external Modems used with notebook PCs.
4. **PC-Card Modems:** PC and Modems are read with PCMCIA slots found in notebooks. They are like external Modems which fit into an internal slot. Thus, they give the advantage of both external and internal modems but are more expensive.

Modems come according to CCITT/ITU standards, e.g., V.32, V.32bis, V.42 etc.

Modem Language

Modems understand a set of instructions called *Hayes Command Set* or the *AT Command Set*. These commands are used to communicate with the Modem. Sometimes, when you are in trouble setting up your Modem, it is useful to know some basic commands, e.g., ATDT 17776 will dial the number 17776 across a Tone Phone and ATDP 17776 to the number 17776 if it is a Pulse phone.

4.11 SCANNERS

A Scanner is a device that allows you to capture drawings or photographs or text from tangible sources (paper, slides etc.) into electronic form. Scanners work by detecting differences in brightness of reflections from an image or object using light sensors. These light sensors are arranged in an array across the whole width that is scannable. This packing determines the resolution and details that can be scanned.

Scanners come in various types: *Drum Scanners*, *Flatbed Scanners*, *Hand Scanners* and *Video Scanners*. Drum Scanners use a rotating drum to scan loose paper sheets. Flatbed scanners have movable sensors to scan images placed on a flat glass tray.

These are the most expensive kind. Hand held Scanners are the cheapest and most portable.

They are useful for many applications but are small in size and need good hand control for high quality scanning. Video Scanners use Video technology and Video cameras instead of Scanning technology. Potentially, they can give high resolutions, scanners in the economical range give poor resolutions.

Faltbed Scanner

Hand-held Scanner

Figure 7: Scanners

When you buy a scanner, there are many factors that can be looked at: Compatibility of the Scanner with your Computer, The Technology (Depth, Resolution), the media types supported for scanning, How media can be loaded, Media size supported, Interfaces supported, physical dimensions, style and ease of use of the scanner. One exciting application of Scanners is Optical Recognition of Characters (*OCR*). OCR software tries to recognise characters from their shapes and write out the scanned text as a text file. Though this technology is steadily improving, it is still not completely reliable especially w.r.t. Indian scripts. However, it can be very useful to digitize the ancient texts written in Indian scripts.

Scanning technology is also everpresent nowadays in Bar-Code readers and MICR (Magnetic Ink Character Recognition) cheques. This technology is very useful for automating data at source of origin, thereby avoiding problems like inaccuracies in data entry, etc.

4.11.1 Resolution

Optical Resolution

Optical resolution or hardware resolution is the mechanical limit on resolution of the Scanner. For scanning, the sensor has to advance after each line it scans. The smallness of this advancement step gives the resolution of the Scanner. Typically, Scanners may be available with mechanical resolutions of 300, 600, 1200 or 2400 dpi. Some special scanners even scan at 10,000 dpi.

Interpolated Resolution

Each Scanner is accompanied by a software. This software can increase the apparent resolution of the scan by a technique called Interpolation. By this technique, additional dots are interpolated (added) between existing dots. This gives a higher resolution and smoother picture but without adding any additional information. The added dots will however lead to larger file sizes.

4.11.2 Dynamic Range/Colour Depth

Dynamic Range is the number of colours a colour scan or the number of grays a monochrome scanner can differentiate. The dynamic range is usually given as bit-depth or colour depth. This is simply the number of bits to distinguish the colours. Most scanners can do 256(8-bit), 1024(10-bit) or 4096(12-bit) for each primary colour. This adds up to and is advertised as 24-bit, 30-bit and 36-bit colour scanners. Actually though, to utilise the Colour Depth, the image under scanning must be properly focused upon and properly illuminated by the scanner.

Since the minimum colour range useful for human vision is 24-bits, more bits may seem useful. However, extra bits of scanning give you firm control for filtering the image colour to your requirements.

4.11.3 Size and Speed

Before actual scanning, a quick, low resolution scan called pre-scan is made to preview the image and select scanning area. After this only does the actual scan take place. Early colour scanners used to take three passes for a scan - one pass for each colour. Now, Scanners use just one pass and use photodetectors to detect the colours. Then, they operate as fast as monochrome Scanners. However, other issues are also involved.

High resolution scans of large images result in large file sizes. These can slow down processing since they need Hard Disk I/O for virtual memory. Hence, for large scans, it is necessary to have higher RAM in your PC.

4.11.4 Scanning Tips

- Do not scan at more resolution than required. This saves both time and Disk Space.
- Usually, it is not useful to scan at more than the optical resolution since it adds no new information. Interpolation can be done later with Image processing softwares.
- If scanning photographs for Printers, it is enough to scan at one-third the resolution of printing, since Printers usually use Super-Pixels (Dithering) for printing. Only for other kind of Printers, like continuous tone Printers, do you need to scan at the Printer resolution for best quality.
- For images to be seen only at the Computer Monitor, you may need to only scan so that the image size in pixels is the same as display resolution. That is, Scan resolution = Height of image in pixels divided by the screen size in inches. This may be surprisingly small.

4.12 POWER SUPPLY

Computer operate electronically — either through power supply obtained from your electric plug or batteries as in the case of portable computers. However, the current coming through your electric line is too strong for the delicate computer circuits. Also, electricity is supplied as AC but the computer uses DC. Thus, power supply is that equipment which takes AC from electrical supply and converts it to DC to supply to computer circuits. Early power supplies were linear power supplies and they worked by simply blocking one cycle of the AC current. They were superseded by the SMPS.

SMPS (Switched Mode Power Supply)

SMPS is the unit into which the electric supply from the mains is attached to your PC and this supplies DC to the internal circuits. It is more efficient, less expensive and more complex than linear supplies.

SMPS works in the following way: The electric supply received is sent to a component called **triac** which shifts it from 50 Hz to a much higher frequency (almost 20,000 Hz). At the same time, using a technique called **Pulse Width modulation**, the pulse is varied to the needs of the computer circuit. Shorter pulses give lower output voltage. A transformer then reduces back the voltage to the correct levels and rectifiers and filters generate the pure DC current.

SMPS has two main advantages: They generate less heat since they waste less power, and use less expensive transformers and circuits since they operate at higher frequencies.

The power requirement of a PC depends on the motherboard and the peripherals in your computer. Still, in modern PCs, your requirement may not be more than 150-200 Watts.

Check Your Progress 3

1. In what ways does a digital camera differ from a conventional camera?

.....

.....

.....

.....

2. Explain the term Resolution and how it applies to Monitors, Cameras, Printers, Scanners etc.

.....

.....

.....

.....

3. Explain the process of Colour management in Printers.

.....

.....

.....

.....

.....

4. Compare Laptops using passive matrix and TFT technology. Which are cheaper in price?

.....

.....

.....

.....

.....

In this unit, we discussed various Input/Output devices. We have covered the input devices Keyboard, Mouse and Scanner. Various types of Keyboards, Keyboard layouts (QWERTY, Dvorak) and technologies have been discussed. Various types of mice and their operation have been discussed. Different types of Scanners, the underlying technology and use in applications like OCR have been discussed.

The output devices discussed are Monitor, LCD and Printer. The technologies and specifications behind Monitors, LCD and Printers have been discussed. Colour management has also been discussed. Video cards, which control the display on monitors from the CPU and their system of display have been discussed with their characteristics like depth, resolution and memory. Modem is a communication device and thereby an I/O device. Its functioning has been discussed. The Power supply, and especially, the SMPS, which is actually input of electric power for the computing unit, has also been discussed.

Check Your Progress 1

Check Your Progress 1

- 104

Check Your Progress 2

1. A true-colour system has a depth of 24 bits per pixel. This means that 8 bits each are assigned to R,G and B i.e. there are 8 Colour Planes. Hence, in figure-4 replace 'n' by 8 to draw the new figure.
2. Framebuffer is another name for the Display Memory. This is like a time-slice of what you see on your monitor. Discuss how framebuffer is handled differently in early display systems, PCI, AGP and UMA. (refer text for details).
3. Shadow Mask: Trinitron uses Aperture Grills instead of Shadow Mask, for the same purpose.
Dot Pitch: Similarly, instead of Dot Pitch, there is Slot Pitch.
explain the terms Shadow Mask, Aperture Grill, Dot Pitch and Slot Pitch (refer text).
4. Ans. (b) $1024 \times 768 \times 2\text{Bytes} = 1.6\text{MB}$. RAM is/was available as 1MB, 4MB, 16MB etc.
5. Ans. (a) Total screen size = $12 \times 9 = 108$ inches. image size = $1024 \times 768 = 786432$ pixels. divide 108 inches by 786432.

Check Your Progress 3

1. In a digital camera, photos are stored in digital format. Instead of film, these cameras use Semiconductor devices, called image sensors. There are many other differences regarding quality, resolution etc.
2. Resolution is a generic term the parameter that defines the possible sharpness or clarity of something i.e. how clearly that thing can be resolved. This applies especially to images. See in what different ways it is used for Monitors, Cameras, Printers, Scanners and even Mice.
3. It tells about physical mixing, optical mixing and RGB and CMYK schemes. The technique of dithering is used for rich colour quality. Colours also differ on monitors and printers. To maintain similarity is also an important issue.
4. Compare Laptops made using passive matrix and TFT technology. Which are cheaper in price?
In a Passive matrix arrangement, the LCD has a grid of horizontal and vertical conductors. Each pixel is located at an intersection. When a current is received by the pixel, it becomes dark whereas in Active Matrix, also called TFT (Thin Film Transistor) technology, each pixel is active, working as a relay. Hence, it needs less power and gives better quality display. Passive matrix LCDs are cheaper but now, TFT LCDs are also economically available. (find out the latest from the market).
5. Ans. (d) ATDP 26176661.
6. Ans. (c) 8MB. The memory requirement increases as a square of the resolution(dpi), so an increase of two times in the dpi leads to an increase of four times in the memory requirement.
7. Ans. All of them.

References:

- 1) [http: //whatis.techtarget.com/](http://whatis.techtarget.com/).
- 2) [http: //www.epanorama.net/links/pc/index.htm](http://www.epanorama.net/links/pc/index.htm).
- 3) [http: //www.howstuffworks.com/](http://www.howstuffworks.com/).
- 4) Mark Minasi. *The Complete PC Upgrade and Maintenance Guide*. BPB Publications, New Delhi, 2002.
- 5) Winn L. Rosch. *Hardware Bible*. Techmedia, New Delhi, 1997.

UNIT 1 INSTRUCTION SET ARCHITECTURE

Structure	Page No.
1.0 Introduction	5
1.1 Objectives	5
1.2 Instruction Set Characteristics	6
1.3 Instruction Set Design Considerations	9
1.3.1 Operand Data Types	
1.3.2 Types of Instructions	
1.3.3 Number of Addresses in an Instruction	
1.4 Addressing Schemes	18
1.4.1 Immediate Addressing	
1.4.2 Direct Addressing	
1.4.3 Indirect Addressing	
1.4.4 Register Addressing	
1.4.5 Register Indirect Addressing	
1.4.6 Indexed Addressing Scheme	
1.4.7 Base Register Addressing	
1.4.8 Relative Addressing Scheme	
1.4.9 Stack Addressing	
1.5 Instruction Set and Format Design Issues	26
1.5.1 Instruction Length	
1.5.2 Allocation of Bits Among Opcode and Operand	
1.5.3 Variable Length of Instructions	
1.6 Example of Instruction Format	28
1.7 Summary	29
1.8 Solutions/ Answers	30

1.0 INTRODUCTION

The Instruction Set Architecture (ISA) is the part of the processor that is visible to the programmer or compiler designer. They are the parts of a processor design that need to be understood in order to write assembly language, such as the machine language instructions and registers. Parts of the architecture that are left to the implementation are not part of ISA. The ISA serves as the boundary between software and hardware.

The term instruction will be used in this unit more often. What is an instruction? What are its components? What are different types of instructions? What are the various addressing schemes and their importance? This unit is an attempt to answer these questions. In addition, the unit also discusses the design issues relating to instruction format. We have presented here the instruction set of MIPS (Microprocessor without Interlocked Pipeline Stages) processor (very briefly) as an example.

Other related microprocessors instruction set can be studied from further readings. We will also discuss about the complete instruction set of 8086 micro-processor in unit 1, Block 4 of this course.

1.1 OBJECTIVES

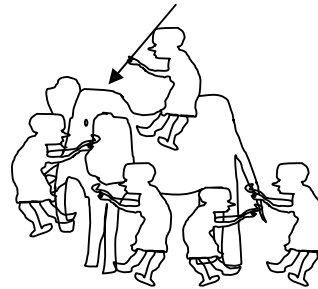
After going through this unit you should be able to:

- describe the characteristics of instruction set;
- discuss various elements of an instruction;
- differentiate various types of operands;

- distinguish various types of instructions and various operations performed by the instructions;
- identify different types of ISAs on the basis of addresses in instruction sets;
- identify various addressing schemes; and
- discuss the instruction format design issues.

1.2 INSTRUCTION SET CHARACTERISTICS

The key role of the Central Processing Unit (CPU) is to perform the calculations, to issue the commands, to coordinate all other hardware components, and executing programs including operating system, application programs etc. on your computer. But CPU is primarily the core hardware component; you must speak to it in the core binary machine language. The words of a machine language are known as *instructions*, and its syntax is known as an *instruction set*.



The Instruction Set Viewpoints

Instruction set is the boundary where the computer designer and the computer programmer see the same computer from different viewpoints. From the designer's point of view, the computer instruction set provides a functional description of a processor, that is:

- (i) A detailed list of the instructions that a processor is capable of processing.
- (ii) A description of the types/ locations/ access methods for operands.

The common goal of computer designers is to build the hardware for implementing the machine's instructions for CPU. From the programmer's point of view, the user must understand machine or assembly language for low-level programming. Moreover, the user must be aware of the register set, instruction types and the function that each instruction performs.

This unit covers both the viewpoints. However, our prime focus is the programmer's viewpoint with the design of instruction set. Now, let us define the instructions, parts of instruction and so on.

What is an Instruction Set?

Instruction set is the collection of machine language instructions that a particular processor understands and executes. In other words, a set of assembly language mnemonics represents the machine code of a particular computer. Therefore, if we define all the instructions of a computer, we can say we have defined the instruction set. It should be noted here that the instructions available in a computer are machine dependent, that is, a different processors have different instruction sets. However, a newer processor that may belong to some family may have a compatible but extended instruction set of an old processor of that family. **Instructions can take different formats.** The instruction format involves:

- the instruction length;

- the type;
- length and position of operation codes in an instruction; and
- the number and length of operand addresses etc.

An interesting question for instruction format may be to have uniform length or variable length instructions.

What are the elements of an instruction?

As the purpose of instruction is to communicate to CPU what to do, it requires a minimum set of communication as:

- What operation to perform?
- On what operands?

Thus, each instruction consists of several fields. The most common fields found in instruction formats are:

Opcode: (What operation to perform?)

- An operation code field termed as **opcode** that specifies the operation to be performed.

Operands: (Where are the operands?)

- An address field of operand on which data processing is to be performed.
- An operand can reside in the memory or a processor register or can be incorporated within the operand field of instruction as an **immediate constant**. Therefore a mode field is needed that specifies the way the operand or its address is to be determined.

A sample instruction format is given in figure 1.

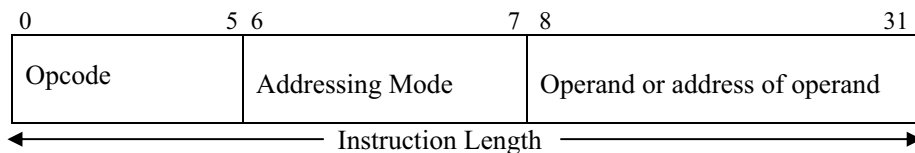


Figure 1: A Hypothetical Instruction Format of 32 bits

Please note the following points in Figure 1:

- The opcode size is 6 bits. So, in general it will have $2^6 = 64$ operations. (However, when you will study more architectures from further readings, you will find even through these bits using special combinations. Instruction set designers have developed much more operations).
- There is only one operand address machine. What is the significance of this? You will find an answer of this question in section 1.3.3 of this unit.
- There are two bits for addressing modes. Therefore, there are $2^2 = 4$ different addressing modes possible for this machine.
- The last field (8 – 31 bits = 24 bits) here is the operand or the address of operand field.

In case of immediate operand the maximum size of the unsigned operand would be 2^{24} .

In case it is an address of operand in memory, then the maximum physical memory size supported by this machine is $2^{24} = 16$ MB.

For this machine there may be two more possible addressing modes in addition to the immediate and direct. However, let us not discuss addressing modes right now. They will be discussed in general, details in section 1.4 of this unit.

The opcode field of an instruction is a group of bits that define various processor operations such as LOAD, STORE, ADD, and SHIFT to be performed on some data stored in registers or memory.

The operand address field can be **data**, or can refer to data – that is address of data, or can be labels, which may be the address of an instruction you want to execute next, such labels are commonly used in Subroutine call instructions. An operand address can be:

- The memory address
- CPU register address
- I/O device address

The mode field of an instruction specifies a variety of alternatives for referring to operands using the given address. **Please note that if the operands are placed in processor registers then an instruction executes faster than that of operands placed in memory, as the registers are very high-speed memory used by the CPU. However, to put the value of a memory operand to a register you will require a register LOAD instruction.**

How is an instruction represented?

Instruction is represented as a sequence of bits. A layout of an instruction is termed as *instruction format*. Instruction formats are primarily machine dependent. A CPU instruction set can use many instruction formats at a time. Even the length of opcode varies in the same processor. However, we will not discuss such details in this block. You can refer to further readings for such details.

How many instructions in a Computer?

A computer can have a large number of instructions and addressing modes. The older computers with the growth of Integrated circuit technology have a very large and complex set of instructions. These are called “complex instruction set computers” (CISC). Examples of CISC architectures are the Digital Equipment Corporation VAX computer and the IBM 370 computer.

However, later it was found in the studies of program style that many complex instructions found CISC are not used by the program. This lead to the idea of making a simple but faster computer, which could execute simple instructions much faster. These computers have simple instructions, registers addressing and move registers. These are called Reduced Instruction Set Computers (RISC). We will study more about RISC in Unit 5 of this Block.

Check Your Progress 1

State True or False.

T	F
---	---

1. An instruction set is a collection of all the instructions a CPU can execute. ☐
2. Instructions can take different formats. ☐
3. The opcode field of an instruction specifies the address field of operand on which data processing is to be performed. ☐

4. The operands placed in processor registers are fetched faster than that of operands placed in memory. ☐
5. Operands must refer to data and cannot be data. ☐

1.3 INSTRUCTION SET DESIGN CONSIDERATIONS

Some of the basic considerations for instruction set design include selection of:

- A set of data types (e.g. integers, long integers, doubles, character strings etc.).
- A set of operations on those data types.
- A set of instruction formats. Includes issues like number of addresses, instruction length etc.
- A set of techniques for addressing data in memory or in registers.
- The number of registers which can be referenced by an instruction and how they are used.

We will discuss the above concepts in more detail in the subsequent sections.

1.3.1 Operand Data Types

Operand is that part of an instruction that specifies the address of the source or result, or the data itself on which the processor is to operate. Operand types usually give operand size implicitly. In general, operand data types can be divided in the following categories. Refer to figure 2:

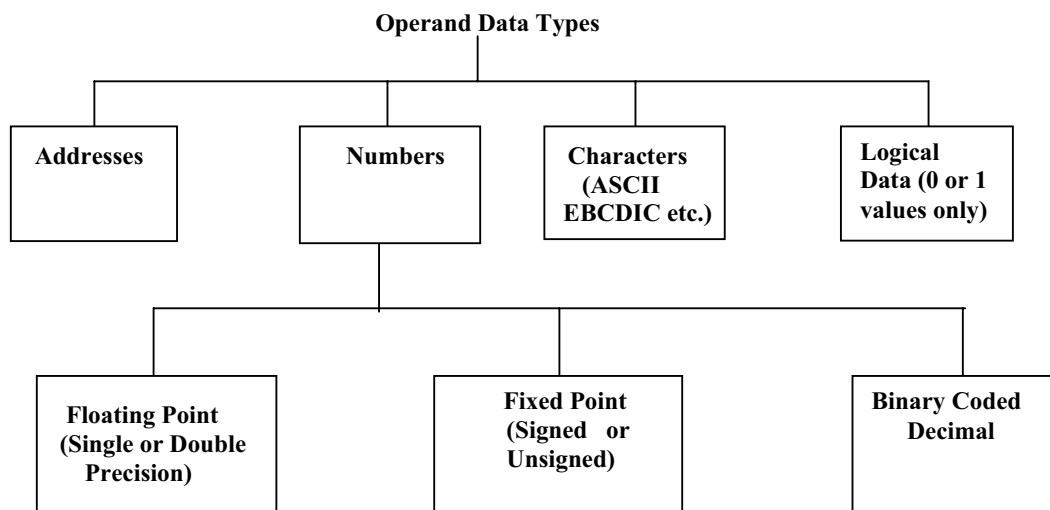


Figure 2: Operand Data Types

- *Addresses*: Operands residing in memory are specified by their memory address and operands residing in registers are specified by a register address. Addresses provided in the instruction are operand references.
- *Numbers*: All machine languages include numeric data types. Numeric data usually use one of three representations:
 - Floating-point numbers-single precision (1 sign bit, 8 exponent bits, 23 mantissa bits) and double precision (1 sign bit, 11 exponent bits, 52 mantissa bits).
 - Fixed point numbers (signed or unsigned).

- Binary Coded Decimal Numbers.

Most of the machines provide instructions for performing arithmetic operations on fixed point and floating-point numbers. However, there is a limit in magnitude of numbers due to underflow and overflow.

- *Characters*: A common form of data is text or character strings. Characters are represented in numeric form, mostly in ASCII (American Standard Code for Information Exchange). Another Code used to encode characters is the Extended Binary Coded Decimal Interchange Code (EBCDIC).
- *Logical data*: Each word or byte is treated as a single unit of data. When an n-bit data unit is considered as consisting of n 1-bit items of data with each item having the value 0 or 1, then they are viewed as logical data. Such bit-oriented data can be used to store an array of Boolean or binary data variables where each variable can take on only the values 1 (true) and 0 (false). One simple application of such a data may be the cases where we manipulate bits of a data item. For example, in floating-point addition we need to shift mantissa bits.

1.3.2 Types of Instructions

Computer instructions are the translation of high level language code to machine level language programs. Thus, from this point of view the machine instructions can be classified under the following categories. Refer to figure 3:

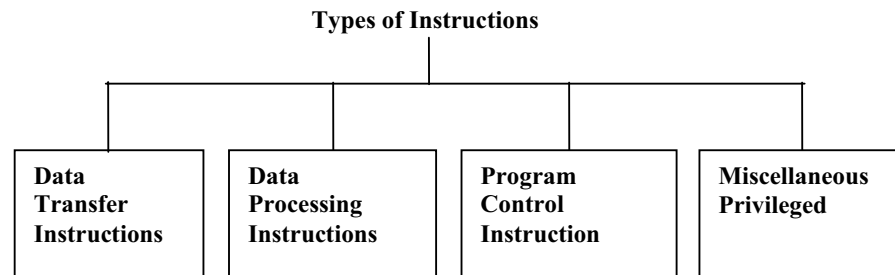


Figure 3: Types of Instructions

Data Transfer Instructions

These instructions transfer data from one location in the computer to another location without changing the data content. The most common transfers are between:

- processor registers and memory,
- processor registers and I/O, and
- processor registers themselves.

These instructions need:

- the location of source and destination operands and
- the mode of addressing for each operand. Given below is a table, which lists eight data transfer instructions with their mnemonic symbols. These symbols are used for understanding purposes only, the actual instructions are binary. Different computers may use different mnemonic for the same instruction.

Operation Name	Mnemonic	Description
Load	LD	Loads the contents from memory to register.
Store	ST	Store information from register to memory location.
Move	MOV	Data Transfer from one register to another or between CPU registers and memory.

Exchange	XCH	Swaps information between two registers or a register and a memory word.
Clear	CLEAR	Causes the specified operand to be replaced by 0's.
Set	SET	Causes the specified operand to be replaced by 1's.
Push	PUSH	Transfers data from a processor register to top of memory stack.
Pop	POP	Transfers data from top of stack to processor register.

Data Processing Instructions

These instructions perform arithmetic and logical operations on data. Data Manipulation Instructions can be divided into three basic types:

Arithmetic: The four basic operations are ADD, SUB, MUL and DIV. An arithmetic instruction may operate on fixed-point data, binary or decimal data etc. The other possible operations include a variety of single-operand instructions, for example ABSOLUTE, NEGATE, INCREMENT, DECREMENT.

The execution of arithmetic instructions requires bringing in the operands in the operational registers so that the data can be processed by ALU. Such functionality is implemented generally within instruction execution steps.

Logical: AND, OR, NOT, XOR operate on binary data stored in registers. For example, if two registers contain the data:

R1 = 1011 0111
R2 = 1111 0000

Then,

$R1 \text{ AND } R2 = 1011 0000$. Thus, the AND operation can be used as a *mask* that selects certain bits in a word and zeros out the remaining bits. With one register is set to all 1's, the XOR operation inverts those bits in R_1 register where R_2 contains 1.

$R_1 \text{ XOR } R_2 = 0100 0111$

Shift: Shift operation is used for transfer of bits either to the left or to the right. It can be used to realize simple arithmetic operation or data communication/recognition etc. Shift operation is of three types:

1. Logical shifts LOGICAL SHIFT LEFT and LOGICAL SHIFT RIGHT insert zeros to the end bit position and the other bits of a word are shifted left or right respectively. The end bit position is the leftmost bit for shift right and the rightmost bit position for the shift left. The bit shifted out is lost.

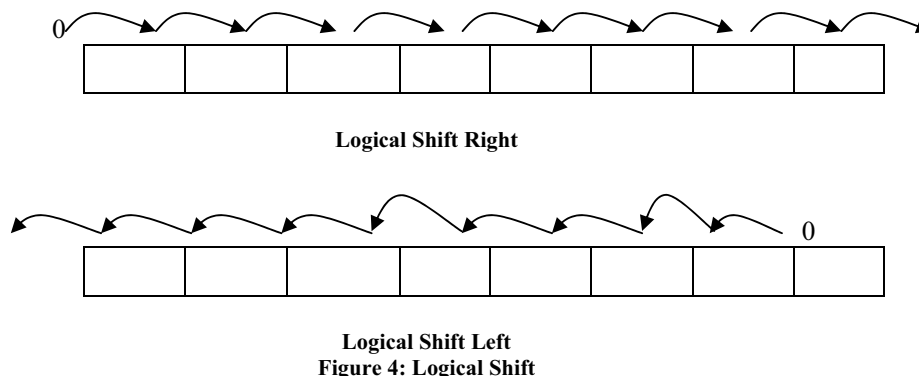


Figure 4: Logical Shift

2. Arithmetic shifts ARITHMETIC SHIFT LEFT and ARITHMETIC SHIFT RIGHT are the same as LOGICAL SHIFT LEFT and LOGICAL SHIFT RIGHT

except that the sign bit it remains unchanged. On an arithmetic shift right, the sign bit is replicated into the bit position to its right. On an arithmetic shift left, a logical shift left is performed on all bits but the sign bit, which is retained.

The arithmetic left shift and a logical left shift when performed on numbers represented in two's complement notation cause multiplication by 2 when there is no overflow. Arithmetic shift right corresponds to a division by 2 provided there is no underflow.

3. Circular shifts ROTATE LEFT and ROTATE RIGHT. Bits shifted out at one end of the word are not lost as in a logical shift but are circulated back into the other end.

Character and String Processing Instructions: String manipulation typically is done in memory. Possible instructions include COMPARE STRING, COMPARE CHARACTER, MOVE STRING and MOVE CHARACTER. While compare character usually is a byte-comparison operation, compare string always involves memory address.

Stack and register manipulation: If we build stacks, stack instructions prove to be useful. LOAD IMMEDIATE is a good example of register manipulation (the value being loaded is part of the instruction). Each CPU has multiple registers, when instruction set is designed; one has to specify which register the instruction is referring to.

No operation (or idle) is needed when there is nothing to run on a computer.

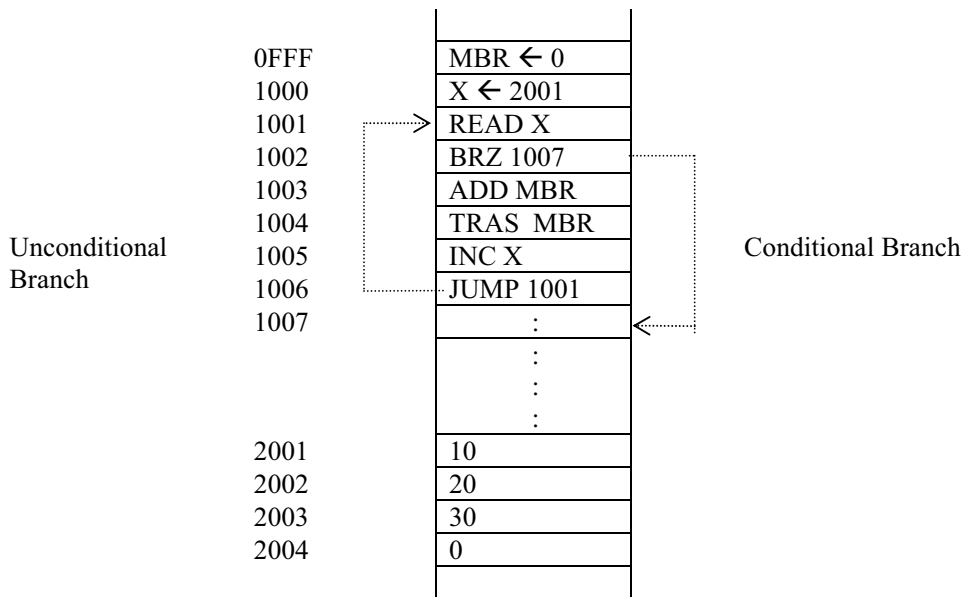
Program Control Instructions

These instructions specify conditions for altering the sequence of program execution or in other words the content of PC (program counter) register. PC points to memory location that holds the next instruction to be executed. The change in value of PC as a result of execution of control instruction like BRANCH or JUMP causes a break in the sequential execution of instructions. The most common control instructions are:

BRANCH and JUMP may be conditional or unconditional. JUMP is an unconditional branch used to implement simple loops. JNE jump not equal is a conditional branch instruction. The conditional branch instructions such as BRP X and BRN X causes a branch to memory location X if the result of most recent operation is positive or negative respectively. If the condition is true, PC is loaded with the branch address X and the next instruction is taken from X, otherwise, PC is not altered and the next instruction is taken from the location pointed by PC. Figure 5 shows an unconditional branch instruction, and a conditional branch instruction if the content of AC is zero.

MBR \leftarrow 0	; Assign 0 to MBR register
X \leftarrow 2001	; Assume X to be an address location 2001
READ X	; Read a value from memory location 2001 into AC
BRZ 1007	; Branch to location 1007 if AC is zero (Conditional branch on zero)
ADD MBR	; Add the content of MBR to AC and store result to AC
TRAS MBR	; Transfer the contents of AC to MBR
INC X	; Increment X to point to next location
JUMP 1001	; Loop back for further processing.

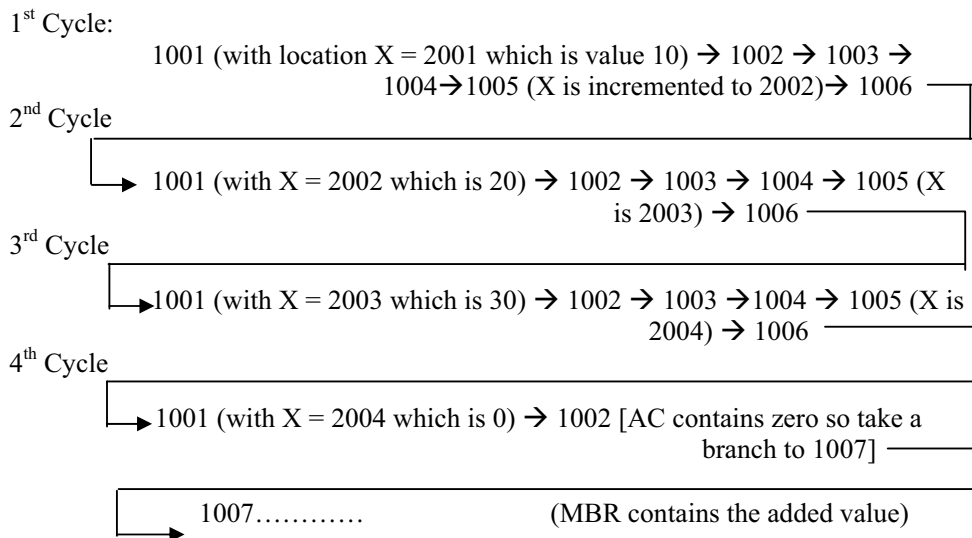
(a) A program on hypothetical machine



(b) The Memory of the hypothetical machine

Figure 5: BRANCH & JUMP Instructions

The program given in figure 5 is a hypothetical program that performs addition of numbers stored from locations 2001 onwards till a zero is encountered. Therefore, X is initialized to 2001, while MBR that stores the result is initialized to zero. We have assumed that in this machine all the operations are performed using CPU. The programs will execute instructions as:



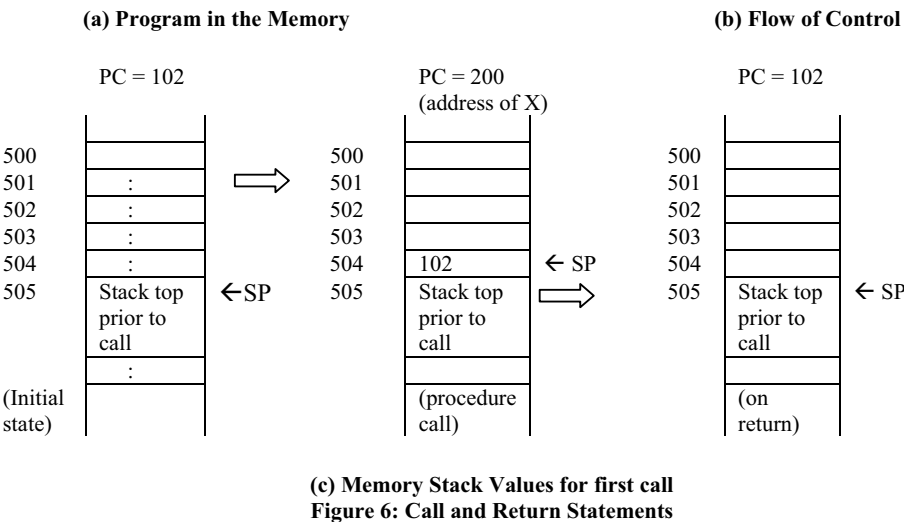
The **SKIP** instruction is a zero-address instruction and skips the next instruction to be executed in sequence. In other words, it increments the value of PC by one instruction length. The SKIP can also be conditional. For example, the instruction ISZ skips the next instruction only if the result of the most recent operation is zero.

CALL and RETN are used for CALLing subprograms and RETurning from them. Assume that a memory stack has been built such that stack pointer points to a non-empty location stack and expand towards zero address.

CALL:

CALL X Procedure Call to function /procedure named X
CALL instruction causes the following to happen:

- 1. Decrement the stack pointer so that we will not overwrite last thing put on stack,
($SP \leftarrow SP - 1$)



- 2. The contents of PC, which is pointing to NEXT instruction, the one just after the CALL is pushed onto the stack, and, $M[SP] \leftarrow PC$.
- 3. JMP to X, the address of the start of the subprogram is put in the PC register; this is all a **jump** does. Thus, we go off to the subprogram, but we have to remember where we were in the calling program, i.e. we must remember where we came from, so that we can get back there again.

$PC \leftarrow X$

RETN :

RETN Return from procedure.

RETN instruction causes the following to happen:

1. Pops the stack, to yield an address/label; if correctly used, the top of the stack will contain the address of the next instruction after the call from which we are returning; it is this instruction with which we want to resume in the *calling* program;
2. Jump to the popped address, i.e., put the address into the PC register.
PC \leftarrow top of stack value; Increment SP.

Miscellaneous and Privileged Instructions: These instructions do not fit in any of the above categories. I/O instructions: start I/O, stop I/O, and test I/O. Typically, I/O destination is specified as an address. Interrupts and state-swapping operations: There are two kinds of exceptions, interrupts that are generated by hardware and traps, which are generated by programs. Upon receiving interrupts, the state of current processes will be saved so that they can be restarted after the interrupt has been taken care of.

Most computer instructions are divided into two categories, privileged and non-privileged. A process running in privileged mode can execute all instructions from the instruction set while a process running in user mode can only execute a sub-set of the instructions. I/O instructions are one example of privileged instruction, clock interrupts are another one.

1.3.3 Number of Addresses in an Instruction

In general, the Instruction Set Architecture (ISA) of a processor can be differentiated using five categories:

- Operand Storage in the CPU - Where are the operands kept other than the memory?
- Number of explicitly named operands - How many operands are named in an instruction?
- Operand location - Can any ALU instruction operand be located in memory? Or must all operands be kept internally in the CPU registers?
- Operations - What operations are provided in the ISA?
- Type and size of operands - What is the type and size of each operand and how is it specified?

As far as operations and type of operands are concerned, we have already discussed about these in the previous subsection. In this section let us look into some of the architectures that are common in contemporary computer. But before we discuss the architectures, let us look into some basic instruction set characteristics:

- The operands can be addressed in memory, registers or I/O device address.
- Instruction having less number of operand addresses in an instruction may require lesser bits in the instruction; however, it also restricts the range of functionality that can be performed by the instructions. This implies that a CPU instruction set having less number of addresses has longer programs, which means longer instruction execution time. On the other hand, having more addresses may lead to more complex decoding and processing circuits.
- Most of the instructions do not require more than three operand addresses. Instructions having fewer addresses than three, use registers implicitly for operand locations because using registers for operand references can result in smaller instructions as only few bits are needed for register addresses as against memory addresses.
- The type of internal storage of operands in the CPU is the most basic differentiation.

The three most common types of ISAs are:

1. **Evaluation Stack:** The operands are implicitly on top of the stack.
2. **Accumulator:** One operand is implicitly the accumulator.
3. **General Purpose Register (GPR):** All operands are explicit, either registers or memory locations.

Evaluation Stack Architecture: A stack is a data structure that implements Last-In-First-Out (LIFO) access policy. You could add an entry to the stack with a PUSH(value) and remove an entry from the stack with a POP(). No explicit operands are there in ALU instructions, but one in PUSH/POP. Examples of such computers are Burroughs B5500/6500, HP 3000/70 etc.

On a stack machine "C = A + B" might be implemented as:

```
PUSH A
PUSH B
```

```
ADD      // operator POP operand(s) and PUSH result(s) (implicit on top of stack)
```

```
POP C
```

Stack Architecture: Pros and Cons

- Small instructions (do not need many bits to specify the operation).
- Compiler is easy to write.
- Lots of memory accesses required - everything that is not on the stack is in memory. Thus, the machine performance is poor.

Accumulator Architecture: An accumulator is a specially designated register that supplies one instruction operand and receives the result. The instructions in such machines are normally one-address instructions. The most popular early architectures were IBM 7090, DEC PDP-8 etc.

On an Accumulator machine "C = A + B" might be implemented as:

```
LOAD A    // Load memory location A into accumulator
ADD B     // Add memory location B to accumulator
STORE C   // Store accumulator value into memory location C
```

Accumulator Architecture: Pros and Cons

- Implicit use of accumulator saves instruction bits.
- Result is ready for immediate reuse, but has to be saved in memory if next computation does not use it right away.
- More memory accesses required than stack. Consider a program to do the expression:
 $A = B * C + D * E$.

Evaluation of Stack Machine		Accumulator Machine	
Program	Comments	Programs	Comments
PUSH B	Push the value B	LOAD B	Load B in AC
PUSH C	Push C	MULT C	Multiply AC with C in AC
MULT	Multiply (B×C) and store result on stack top	STORE T	Store B×C into Temporary T
PUSH D	Push D	LOAD D	Load D in AC
PUSH E	Push E	MULT E	Multiply E in AC

MULT	Multiply D×E and store result on stack top	ADD T	B×C + D×E
ADD	Add the top two values on the stack	STORE A	Store Result in A
POP A	Store the value in A		

General Purpose Register (GPR) Architecture: A register is a word of internal memory like the accumulator. GPR architecture is an extension of the accumulator idea, i.e., use a set of general-purpose registers, which must be explicitly named by the instruction. Registers can be used for anything either holding operands for operations or temporary intermediate values. The dominant architectures are IBM 370, PDP-11 and all Reduced Instruction Set Computer (RISC) machines etc. The major instruction set characteristic whether an ALU instruction has two or more operands divides GPR architectures:

"C = A + B" might be implemented on both the architectures as:

Register - Memory	Load/Store through Registers
LOAD R1, A	LOAD R1, A
ADD R1, B	LOAD R2, B
STORE C, R1	ADD R3, R1, R2
	STORE C, R3

General Purpose Register Architecture: Pros and Cons

- Registers can be used to store variables as it reduces memory traffic and speeds up execution. It also improves code density, as register names are shorter than memory addresses.
- Instructions must include bits to specify which register to operate on, hence large instruction size than accumulator type machines.
- Memory access can be minimized (registers can hold lots of intermediate values).
- Implementation is complicated, as compiler writer has to attempt to maximize register usage.

While most early machines used stack or accumulator architectures, in the last 15 years all CPUs made are GPR processors. The three major reasons are that registers are faster than memory; the more data that can be kept internally in the CPU the faster the program will run. The third reason is that registers are easier for a compiler to use.

But while CPU's with GPR were clearly better than previous stack and accumulator based CPU's yet they were lacking in several areas. The areas being: Instructions were of varying length from 1 byte to 6-8 bytes. This causes problems with the pre-fetching and pipelining of instructions. ALU instructions could have operands that were memory locations because the time to access memory is slower and so does the whole instruction.

Thus in the early 1980s the idea of **RISC** was introduced. RISC stands for Reduced Instruction Set Computer. Unlike CISC, this ISA uses fewer instructions with simple constructs so they can be executed much faster within the CPU without having to use memory as often. The first RISC CPU, the MIPS 2000, has 32 GPRs. MIPS is a load/store architecture, which means that only load and store instructions access memory. All other computational instructions operate only on values stored in registers.

Check Your Progress 2

1. Match the following pairs:

(a) Zero address instruction	(i) Accumulator machines
(b) One address instruction	(ii) General Purpose Register machine
(c) Three address instruction	(iii) Evaluation-Stack machine
2. List the advantages and disadvantages of General Purpose Register machines.
3. Categorize the following operations with the respective instruction types:

(a) MOVE	(i) Data Processing Instructions
(b) DIV	(ii) Data Transfer Instructions
(c) STORE	(iii) Privileged Instructions
(d) XOR	(iv) Program Control Instructions
(e) BRN	
(f) COMPARE	
(g) TRAP	

1.4 ADDRESSING SCHEMES

As discussed earlier, an operation code of an instruction specifies the operation to be performed. This operation is executed on some data stored in register or memory. Operands may be specified in one of the three basic forms i.e., immediate, register, and memory.

But, why addressing schemes? The question of **addressing** is concerned with how operands are interpreted. In other words, the term '**addressing schemes**' refers to the mechanism employed for specifying operands. There are a multitude of addressing schemes and instruction formats. Selecting which schemes are available will impact not only the ease to write the compiler, but will also determine how efficient the architecture can be?

All computers employ more than one addressing schemes to give programming flexibility to the user by providing facilities such as pointers to memory, loop control, indexing of data, program relocation and to reduce the number of bits in the operand field of the instruction. Offering a variety of addressing modes can help reduce instruction counts but having more modes also increases the complexity of the machine and in turn may increase the average Cycles per Instruction (CPI). Before we discuss the addressing modes let us discuss the notations being used in this section.

In the description that follows the symbols A, A1, A2 etc. denote the content of an **operand field**. Thus, A_i may refer to a data or a memory address. In case the operand field is a register address, then the symbols R, R1, R2,... etc., are used. If C denotes the contents (either of an operand field or a register or of a memory location), then (C) denotes the content of the memory location whose address is C.

The symbol EA (Effective Address) refers to a physical address in a non-virtual memory environment and refers to a register in a virtual memory address environment. This register address is then mapped to physical memory address.

What is a virtual address? von Neumann had suggested that the execution of a program is possible only if the program and data are residing in memory. In such a situation the program length along with data and other space needed for execution cannot exceed the total memory. However, it was found that at the time of execution, the complete portion of data and instruction is not needed as most of the time only few areas of the program are being referenced. Keeping this in mind a new idea was put

forward where only a required portion is kept in the memory while the rest of the program and data reside in secondary storage. The data or program portion which are stored on secondary storage are brought to memory whenever needed and the portion of memory which is not needed is returned to the secondary storage. Thus, a program size bigger than the actual physical memory can be executed on that machine. This is called virtual memory. Virtual memory has been discussed in greater details as part of the operating system.

The typicality of virtual addresses is that:

- they are longer than the physical addresses as total addressed memory in virtual memory is more than the actual physical memory.
- if a virtual addressed operand is not in the memory then the operating system brings that operand to the memory.

The symbols D, D1, D2,..., etc. refer to actual operands to be used by instructions for their execution.

Most of the machines employ a set of addressing modes. In this unit, we will describe some very common addressing modes employed in most of the machines. A specific addressing mode example, however, is given in Unit 1 of Block 4.

The following tree shows the common addressing modes:

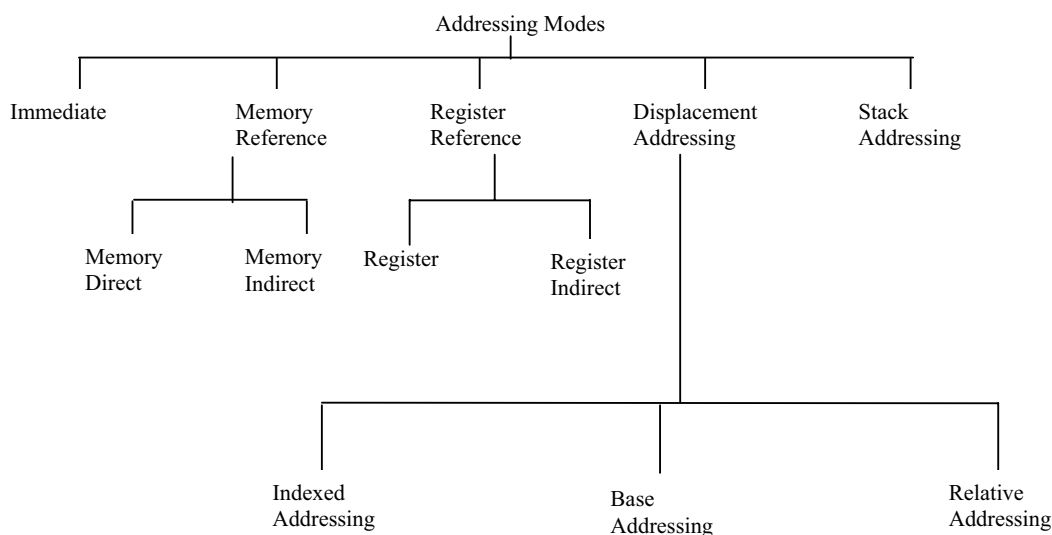


Figure 7: Common Addressing Modes

But what are the uses /applications of these addressing modes?

In general not all of the above modes are used for all applications. However, some of the common areas where compilers of high-level languages use them are:

Addressing Mode	Possible use
Immediate	For moving constants and initialization of variables
Direct	Used for global variables and less often for local variables
Register	Frequently used for storing local variables of procedures
Register Indirect	For holding pointers to structure in programming languages C
Index	To access members of an array
Auto-index mode	For pushing or popping the parameters of procedures
Base Register	Employed to relocate the programs in memory specially in multi-programming systems
Index	Accessing iterative local variables such as arrays
Stack	Used for local variables

1.4.1 Immediate Addressing

When an operand is interpreted as an **immediate** value, e.g. LOAD IMMEDIATE 7, it is the actual value 7 that is put in the CPU register. In this mode the operand is the data in operand address field of the instruction. Since there is no address field at all, and hence no additional memory accesses are required for executing this instruction. In other words, in this addressing scheme, the actual operand D is A, the content of the operand field: i.e. $D = A$. The effective address in this case is not defined.

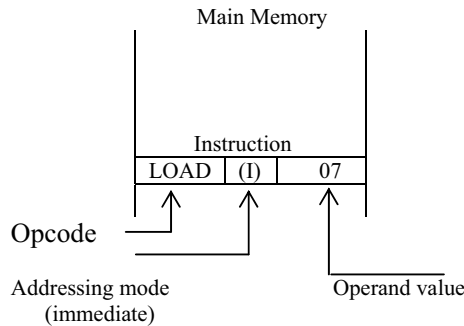


Figure 8: Immediate Addressing

Salient points about the addressing mode are:

- This addressing mode is used to initialise the value of a variable.
- The advantage of this mode is that no additional memory accesses are required for executing the instruction.
- The size of instruction and operand field is limited. Therefore, the type of data specified under this addressing scheme is also restricted. For example, if an instruction of 16 bits uses 6 bits for opcode and 2 bits for addressing mode, then 10 bits can be used to specify an operand. Thus, 2^{10} possible values only can be assigned.

1.4.2 Direct Addressing

In this scheme the operand field of the instruction specifies the **direct address** of the intended operand, e.g., if the instruction LOAD 500 uses direct addressing, then it will result in loading the contents of memory cell 500 into the CPU register. In this mode the intended operand is the address of the data in operation. For example, if memory cell 500 contains 7, as in the diagram below, then the value 7 will be loaded to CPU register.

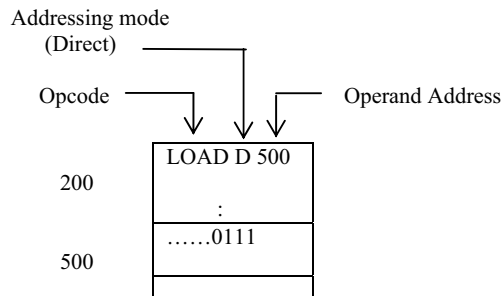


Figure 9: Direct Addressing

Some salient points about this scheme are:

- This scheme provides a limited address space because if the address field has n bits then memory space would contain 2^n memory words or locations. For example, for the example machine of Figure 1, the direct addresses memory space would be 2^{10} .

- The effective address in this scheme is defined as the address of the operand, that is,

$$EA \leftarrow A \quad \text{and} \quad (EA \text{ in the above example will be } 500)$$

$$D = (EA) \quad (D \text{ in the above example will be } 7)$$

The second statement implies that the data is stored in the memory location specified by effective address.

- In this addressing scheme only one memory reference is required to fetch the operand.

1.4.3 Indirect Addressing

In this scheme the operand field of the instruction specifies the **address** of the **address of** intended operand, e.g., if the instruction LOAD I 500 uses indirect addressing scheme, and contains a value 50A, and memory location 50A contains 7, then the value 7 will get loaded in the CPU register.

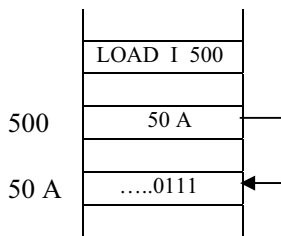


Figure 10: Indirect Addressing

Some salient points about this scheme are:

- In this addressing scheme the effective address EA and the contents of the operand field are related as:
 $EA = (A)$ and (Content of location 500 that is 50A above)
 $D = (EA)$ (Contents of location 50A that is 7)
- The drawback of this scheme is that it requires two memory references to fetch the actual operand. The first memory reference is to fetch the actual address of the operand from the memory and the second to fetch the actual operand using that address.
- In this scheme the word length determines the size of addressable space, as the actual address is stored in a Word. For example, the memory having a word size of 32 bits can have 2^{32} indirect addresses.

1.4.4 Register Addressing

When operands are taken from register(s), implicitly or explicitly, it is called register addressing. These operands are called register operands. If operands are from memory locations, they are called memory operands. In this scheme, a register address is specified in the instruction. That register contains the operand. It is conceptually similar to **direct addressing scheme** except that the register name or number is substituted for memory address. Sometimes the address of register may be assumed implicitly, for example, the Accumulator register in old machines.

Figure 11: Register Addressing

The major advantages of register addressing are:

- Register access is faster than memory access and hence register addressing results in faster instruction execution. However, register obtains operands only from memory; therefore, the operands that should be kept in registers are selected carefully and efficiently. For example, if an operand is moved into a register and processed only once and then returned to memory, then no saving occurs. However if an operand is used repeatedly after bringing into register then we have saved few memory references. Thus, the task of using register efficiently deals with the task of finding what operand values should be kept in registers such that memory references are minimised. Normally, this task is done by a compiler of a high level language while translating the program to machine language. As a thumb rule the frequently used local variables are kept in the registers.
- The size of register address is smaller than the memory address. It reduces the instruction size. For example, for a machine having 32 general purpose registers only 5 bits are needed to address a register.

In this addressing scheme the effective address is calculated as:

$$\begin{aligned}EA &= R \\ D &= (EA)\end{aligned}$$

1.4.5 Register Indirect Addressing

In this addressing scheme, the operand is data in the memory pointed to by a register. In other words, the operand field specifies a register that contains the address of the operand that is stored in memory. This is almost same as indirect addressing scheme except it is much faster than indirect addressing that requires two memory accesses.

Figure 12: Register Indirect Addressing

The effective address of the operand in this scheme is calculated as:

$$EA = (R) \text{ and}$$

$$D = (EA)$$

The address capability of register indirect addressing scheme is determined by the size of the register.

1.4.6 Indexed Addressing Scheme

In this scheme the operand field of the instruction contains an address and an index register, which contains an offset. This addressing scheme is generally used to address the consecutive locations of memory (which may store the elements of an array). The index register is a special CPU register that contains an index value. The contents of the operand field A are taken to be the address of the initial or the reference location (or the first element of array). The index register specifies the distance between the starting address and the address of the operand.

For example, to address of an element $B[i]$ of an array $B[1], B[2], \dots, B[n]$, with each element of the array stored in two consecutive locations, and the starting address of the array is assumed to be 101, the operand field A in the instruction shall contain the number 101 and the index register R will contain the value of the expression $(i - 1) \times 2$.

Thus, for the first element of the array the index register will contain 0. For addressing 5th element of the array, the $A=101$ whereas index register will contain:

$$(5 - 1) \times 2 = 8$$

Therefore, the address of the 5th element of array B is $=101+8=109$. In $B[5]$, however, the element will be stored in location 109 and 110. To address any other element of the array, changing the content of the index register will suffice.

Thus, the effective address in this scheme is calculated as:

$$\begin{aligned} EA &= A + (R) \\ D &= (EA) \\ (\text{DA is Direct address}) \end{aligned}$$

As the index register is used for iterative applications, therefore, the value of index register is incremented or decremented after each reference to it. In several systems this operation is performed automatically during the course of an instruction cycle. This feature is known as auto-indexing. Auto indexing can be auto-incrementing or auto-decrementing. The choice of register to be used as an index register differs from machine to machine. Some machines employ general-purpose registers for this purpose while other machines may specify special purpose registers referred to as index registers.

Figure 13: For Displacement Addressing

1.4.7 Base Register Addressing

An addressing scheme in which the content of an instruction specifies base register is added to the displacement field or address field of the instruction. (Refer to Figure

13). The displacement field is taken to be a positive number. For example, if a displacement field is of 8 bits then a memory region of 256 words beginning at the address pointed to by the base register can be addressed by this mode. This is similar to indexed addressing scheme except that the role of Address field and Register is reversed. In indexing Address field of instruction is fixed and index register value is changed, whereas in Base Register addressing, the Base Register is common and Address field of the instruction in various instructions is changed. In this case:

$$EA = A + (B)$$

$$D = (EA)$$

(B) Refers to the contents of a base register B.

The contents of the base register may be changed in the privileged mode only. No user is allowed to change the contents of the base register. The base-addressing scheme provides protection of users from one another.

This addressing scheme is usually employed to relocate the programs in memory specially in multiprogramming systems in which only the value of base register requires updating to reflect the beginning of a new memory segment.

Like index register a base register may be a general-purpose register or a special register reserved for base addressing.

1.4.8 Relative Addressing Scheme

In this addressing scheme, the register R is the program counter (PC) containing the address of the current instruction being executed. The operand field A contains the displacement (positive or negative) of an instruction or data with respect to the current instruction. This addressing scheme has advantages if the memory references are nearer to the current instruction being executed. (Please refer to the Figure 13).

Let us give an example of Index, Base and Relative addressing schemes.

Example 1: What would be the effective address and operand value for the following LOAD instructions:

- (i) LOAD IA 56 R1 Where IA indicates index addressing, R1 is index register and 56 is the displacement in Hexadecimal.
- (ii) LOAD BA 46 B1 Where BA indicates base addressing, B1 is base register and 46 is the displacement specified in instruction in Hexadecimal notation.
- (iii) LOAD RA 36 Where RA specifies relative addressing.

The values of registers and memory is given below:

Register	Value	Values of Memory Location	
PC	2532 _H	27A8	10 _H
Index Register (R1)	2752 _H		:
			:
Base Register (B1)	2260 _H	2568 _H	70 _H
			:
			:
		22A6 _H	25 _H
			:

The values are shown in the following table:

Addressing Mode	Formulae for addressing mode	EA	Data Value
Index Addressing	$EA = A + (R)$ $D = (EA)$	$56 + 2752 = 27A8_H$	10_H
Base Addressing	$EA = A + (B)$	$46 + 2260 = 22A6_H$	25_H
Relative Addressing	$EA = (PC) + A$	$2532 + 36 = 2568_H$	70_H

1.4.9 Stack Addressing

In this addressing scheme, the operand is implied as top of stack. It is not explicit, but implied. It uses a CPU Register called Stack Pointer (SP). The SP points to the top of the stack i.e. to the memory location where the last value was **pushed**. A stack provides a sort-of indirect addressing and indexed addressing. This is not a very common addressing scheme. The operand is found on the top of a stack. In some machines the top two elements of stack and top of stack pointer is kept in the CPU registers, while the rest of the elements may reside in the memory. Figure 14 shows the stack addressing schemes.

Figure 14: Stack Addressing

Check Your Progress 3

- What are the numbers of memory references required to get the data for the following addressing schemes:
 - Immediate addressing
 - Direct addressing
 - Indirect addressing
 - Register Indirect addressing
 - Stack addressing.

- What are the advantages of Base Register addressing scheme?

- State True or False.

T	F
---	---

- Immediate addressing is best suited for initialization of variables. ☐
- Index addressing is used for accessing global variables. ☐
- Indirect addressing requires fewer memory accesses than that of direct addressing. ☐
- In stack addressing, operand is explicitly specified. ☐

1.5 INSTRUCTION SET AND FORMAT DESIGN ISSUES

Some of the basic issues of concerns for instruction set design are:

Completeness: For an initial design, the primary concern is that the instruction set should be complete which means there is no missing functionality, that is, it should include instructions for the basic operations that can be used for creating any possible execution and control operation.

Orthogonal: The secondary concern is that the instructions be orthogonal, that is, not unnecessarily redundant. For example, integer operation and floating number operation usually are not considered as redundant but different addressing modes may be redundant when there are more instructions than necessary because the CPU takes longer to decode.

An instruction format is used to define the layout of the bits allocated to these elements of instructions. In addition, the instruction format explicitly or implicitly indicates the addressing modes used for each operand in that instruction.

Designing of instruction format it is a complex art. In this section, we will discuss about the design issues for instruction sets of the machines. We will discuss only point wise details of these issues.

1.5.1 Instruction Length

Significance: It is the basic issue of the format design. It determines the richness and flexibility of a machine.

Basic Tradeoff: Smaller instruction (less space) Versus desire for more powerful instruction repertoire.

Normally programmer desire:

- More op-code and operands: as it results in smaller programs
- More addressing modes: for greater flexibility in implementing functions like table manipulations, multiple branching.

However, a 32 bit instruction although will occupy double the space and can be fetched at double the rate of a 16 bit instruction, but can not be doubly useful.

Factors, which must be considered for deciding about instruction length

Memory size	: if larger memory range is to be addressed, then more bits may be required in address field.
Memory organization	: if the addressed memory is virtual memory then memory range which is to be addressed by the instruction is larger than physical memory size.
Memory transfer length	: instruction length should normally be equal to data bus length or multiple of it.
Memory transfer	: the data transfer rate from the memory ideally should be equivalent to the processor speed. It can become a bottleneck if processor executes instructions faster than the rate of fetching the instructions. One solution for such problem is to use cache memory or another solution can be to keep instruction short.

Normally an instruction length is kept as a multiple of length of a character (that is 8 bits), and equal to the length of fixed-point number. The term word is often used in this context. Usually the word size is equal to the length of fixed point number or equal to memory-transfer size. In addition, a word should store integral number of characters. Thus, word size of 16 bit, 32 bit, 64 bit are to be coming very common and hence the similar length of instructions are normally being used.

1.5.2 Allocation of Bits Among Opcode and Operand

The tradeoff here is between the numbers of bits of opcode versus the addressing capabilities. An interesting development in this regard is the development of variable length opcode.

Some of the factors that are considered for selection of addressing bits:

- **Number of addressing modes:** The more are the explicit addressing modes the more bits are needed for mode selection. However, some machines have implicit modes of addressing.
- **Granularity:** As far as memory references are concerned, granularity implies whether an address is referencing a byte or a word at a time. This is more relevant for machines, which have 16 bits, 32 bits and higher bits words. Byte addressing although may be better for character manipulation, however, requires more bits in an address. For example, memory of 4K words (1 word = 16 bit) is to be addressed directly then it requires:

WORD Addressing = 4K words
 = 2^{12} words
 \Rightarrow 12 bits are required for word addressing.

Byte Addressing = 2^{12} words
 = 2^{13} bytes
 \Rightarrow 13 bits are required for byte addressing.

1.5.3 Variable-Length of Instructions

With the better understanding of computer instruction sets, the designers came up with the idea of having a variety of instruction formats of different length. What could be the advantages of such a set? The advantages of such a scheme are:

- Large number of operations can be provided which have different lengths of instructions.
- Flexibility in addressing scheme can be provided efficiently and compactly.

However, the basic disadvantage of such a scheme is to have a complex CPU.

An important aspect about these variables length instructions is: "The CPU is not aware about the length of next instruction which is to be fetched". This problem can be handled if each instruction fetch is made equal to the size of the longest instruction. Thus, sometimes in a single fetch multiple instructions can be fetched.

1.6 EXAMPLE OF INSTRUCTION FORMAT

Let us provide you a basic example by which you may be able to define the concept of instruction format.

MIPS 2000

Let's consider the instruction format of a MIPS computer. **MIPS** is an acronym for **Microprocessor without Interlocked Pipeline Stages**. It is a microprocessor architecture developed by MIPS Computer Systems Inc. most widely known for developing the MIPS architecture. The MIPS CPU family was one of the most successful and flexible CPU designs throughout the 1990s. The MIPS CPU has a five-stage CPU pipeline to execute multiple instructions at the same time. Now what we have introduced is a new term Pipelining. What else: the 5 stage pipeline, let us just introduce it here. It defines the 5 steps of execution of instructions that may be performed in an overlapped fashion. The following diagram will elaborate this concept:

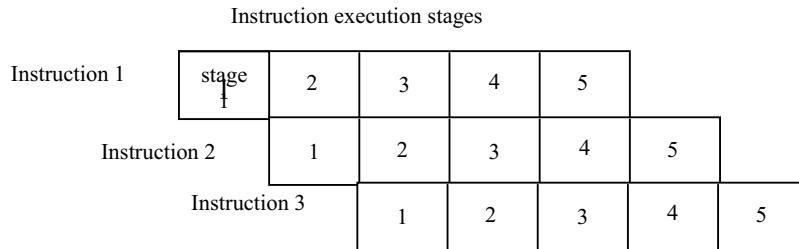


Figure15: Pipeline

Please note that in the above figure:

- All the stages are independent and distinct, that is, the second stage execution of Instruction 1 should not hinder Instruction 2.
- The overall efficiency of the system becomes better.

The early MIPS architectures had 32-bit instructions and later versions have 64-bit implementations.

The first commercial MIPS CPU model, the **R2000**, whose instruction format is discussed below, has thirty-two 32-bit registers and its instructions are 32 bits long.

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	5 bits

Figure 16: A Sample Instruction Format of MIPS instruction

The meaning of each field in MIPS instruction is given below:

- op : operation code or opcode
- rs : The first register source operand
- rt : The second register source operand
- rd : The destination register operand, stores the result of the operation
- shamt : used in case of shift operations
- funct : This field selects the specific variant of the operation in the opcode field, and is sometimes referred to as function code.

All MIPS instructions are of the same length, requiring different kinds of instruction formats for different types of instructions.

Instruction Format

All MIPS instructions are of the same size and are 32 bits long. MIPS designers chose to keep all instructions of the same length, thereby requiring different kinds of instruction formats for different kinds of instructions. For example, R-type (register) or R-format is used for arithmetic instructions (Figure 16). A second type of

instruction format is called i-type or i-format and is used by the data transfer instructions.

Instruction format of I-type instructions is given below:

op	rs	rt	address
6 bits	5 bits	5 bits	16 bits

Figure 17: I-format of RISC

The 16-bit address means a load word instruction can load any word within a region of $+2^{15}$ of the base register **rs**. Consider a load word instruction given below:

The **rt** field specifies the destination register, which receives the result of the load.

MIPS Addressing Modes

MIPS uses various addressing modes:

1. Uses Register and Immediate addressing modes for operations.
2. Immediate and Displacement addressing for Load and Store instructions. In displacement addressing, the operand is at the memory location whose address is the sum of a register.

Check Your Progress 4

1. State True or False.

T	F
---	---

 - (i) Instruction length should normally be equal to data bus length or multiple of it. ☐
 - (ii) A long instruction executes faster than a short instruction. ☐
 - (iii) Memory access is faster than register access. ☐
 - (iv) Large number of opcodes and operands result in bigger program. ☐
 - (v) A machine can use at the most one addressing scheme. ☐
 - (vi) Large number of operations can be provided in the instruction set, which have variable-lengths of instructions. ☐

1.7 SUMMARY

In this unit, we have explained various concepts relating to instructions. We have discussed the significance of instruction set, various elements of an instruction, instruction set design issues, different types of ISAs, various types of instructions and various operations performed by the instructions, various addressing schemes. We have also provided you the instruction format of MIPS machine. Block 4 Unit 1 contains a detailed instruction set of 8086 machine. You can refer to further reading for instruction set of various machines.

1.8 SOLUTIONS/ ANSWERS

Check Your Progress 1

1. True
2. True

3. False
4. True
5. False

Check Your Progress 2

1. (a) - (iii) (b) - (i) (c) - (ii)
2.
 - Speed up of instruction execution as stores temporary results in registers
 - Less code to execute
 - Larger instruction set
 - Difficult for compiler writing
3. (i) - b), d), f) ; (ii) - a), c) ; (iii) - g) ; (iv) - e)

Check Your Progress 3

1.
 - a) Immediate addressing - 0 memory access
 - b) Direct addressing - 1 memory access
 - c) Indirect addressing - 2 memory accesses
 - d) Register Indirect addressing - 1 memory access
 - e) Stack addressing - 1 memory access
2. It allows reallocation of program on reloading. It allows protection of users from one another memory space.
3. (i) True.
(ii) False.
(iii) False.
(iv) False

Check Your Progress 4

1.
 - (i) True.
 - (ii) False.
 - (iii) False.
 - (iv) False.
 - (v) False.
 - (vi) True.

UNIT 2 REGISTERS, MICRO-OPERATIONS AND INSTRUCTION EXECUTION

Structure No.	Page
2.0 Introduction	31
2.1 Objectives	31
2.2 Basic CPU Structure	32
2.3 Register Organization	34
2.3.1 Programmer Visible Registers	
2.3.2 Status and Control Registers	
2.4 General Registers in a Processor	37
2.5 Micro-operation Concepts	38
2.5.1 Register Transfer Micro-operations	
2.5.2 Arithmetic Micro-operations	
2.5.3 Logic Micro-operations	
2.5.4 Shift Micro-operations	
2.6 Instruction Execution and Micro-operations	45
2.7 Instruction Pipelining	49
2.8 Summary	50
2.9 Solutions/ Answers	51

2.0 INTRODUCTION

The main task performed by the CPU is the execution of instructions. In the previous unit, we have discussed about the instruction set of computer system. But, one thing, which remained unanswered is: how these instructions will be executed by the CPU?

The above question can be broken down into two simpler questions. These are:

What are the steps required for the execution of an instruction? How are these steps performed by the CPU?

The answer to the first question lies in the fact that each instruction execution consists of several steps. Together they constitute an instruction cycle. A micro-operation is the smallest operation performed by the CPU. These operations put together execute an instruction.

For answering the second question, we must have an understanding of the basic structure of a computer. As discussed earlier, the CPU consists of an Arithmetic Logic Unit, the control unit and operational registers. We will be discussing the register organisation in this unit, whereas the arithmetic-logic unit and control unit organisation are discussed in subsequent units.

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 and their implementation. The discussion on micro-operations will gradually lead us towards the discussion of a very simple ALU structure. The detail of ALU structure is the topic of the next unit.

2.1 OBJECTIVES

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

- describe the register organisation of the CPU;
- define what is a micro-operation;

- differentiate among various micro-operations;
- discuss an instruction execution using the micro-operations; and
- define the concepts of instruction pipelining.

2.2 BASIC CPU STRUCTURE

A computer manipulates data according to the instructions of a stored program. **Stored program** means the program and data are stored in the same memory unit. The central processing unit, also referred to as CPU, performs the bulk of the data processing operations. It has three main components:

1. A set of registers for holding binary information.
2. An arithmetic and logic unit (ALU) for performing data manipulation, and
3. A control unit that coordinates and controls the various operations and initiates the appropriate sequence of micro-operations for each task.

Computer instructions are normally stored in consecutive memory locations and are executed in sequence one by one. The control unit allows reading of an instruction from a specific address in memory and executes it with the help of ALU and Register.

Instruction Execution and Registers

The basic process of instruction execution is:

1. Instruction is fetched from memory to the CPU registers (called instruction fetch) under the control unit.
2. It is decoded by the control unit and converted into a set of lower level control signals, which cause the functions specified by that instruction to be executed.
3. After the completion of execution of the current instruction, the next instruction fetched is the next instruction in sequence.

This process is repeated for every instruction except for program control instructions, like branch, jump or exception instructions. In this case the next instruction to be fetched from memory is taken from the part of memory specified by the instruction, rather than being the next instruction in sequence.

But why do we need Registers?

If t_{cpu} is the cycle time of CPU that is the time taken by the CPU to execute a well-defined micro-operation using registers, and t_{mem} is the memory cycle time, that is the speed at which the memory can be accessed by the CPU, then $(t_{\text{cpu}}/t_{\text{mem}})$ is in the range of 2 to 10, that is CPU is 2 – 10 times faster than memory. Thus, CPU registers are the fastest temporary storage areas. Thus, the instructions whose operands are stored in the fast CPU registers can be executed rapidly in comparison to the instructions whose operands are in the main memory of a computer. Each instruction must designate the registers it will address. Thus, a machine requires a large number of registers.

Figure 1: CPU with general register organisation

But how do the registers help in instruction execution? We will discuss this with the help of Figure 1.

Step 1:

The first step of instruction execution is to fetch the instruction that is to be executed. To do so we require:

- Address of the “instruction to be fetched”. Normally Program counter (PC) register stores this information.
- Now this address is converted to physical machine address and put on address bus with the help of a buffer register sometimes called Memory Address Register (MAR).
- This, coupled with a request from control unit for reading, fetches the instruction on the data bus, and transfers the instruction to Instruction Register (IR).
- On completion of fetch PC is incremented to point to the next instruction.

In Step 2:

- The IR is decoded; let us assume that Instruction Register contains an instruction. ADD Memory location B with general purpose register R1 and store result in R1, then control unit will first instruct to:
 - Get the data of memory location B to buffer register for data (DR) using buffer address register (MAR) by issuing Memory read operation.
 - This data may be stored in a general purpose register, if so needed let us say R2
 - Now, ALU will perform addition of R1 & R2 under the command of control unit and the result will be put back in R1. The status of ALU

operation for example result in zero/non zero, overflow/no overflow etc. is recorded in the status register.

- Similarly, the other instructions are fetched and executed using ALU and register under the control of the Control Unit.

Thus, for describing instruction execution, we must describe the registers layout, micro-operations, ALU design and finally the control unit organization. We will discuss registers and micro- operation in this unit. ALU and Control Unit are described in Unit 3 and Unit 4 of this Block.

2.3 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 next instruction to be executed.
- 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. With these three kinds of registers, a computer would be able to execute programs.
- Other types of registers:
 - Memory-address register (MAR) holds the address of next memory operation (load or store).
 - Memory-buffer register (MBR) 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).

A few factors to consider when choosing the number of registers in a CPU are:

- CPU can access registers faster than it can access main memory.
- For addressing a register, depending on the number of addressable registers a few bit addresses is needed in an instruction. These address bits are definitely quite less in comparison to a memory address. For example, for addressing 256 registers you just need 8 bits, whereas, the common memory size of 1MB requires 20 address bits, a difference of 60%.
- Compilers tend to use a small number of registers because large numbers of registers are very 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 the register set can be classified under two basic categories.

Programmer Visible Registers: These registers can be used by machine or assembly language programmers to minimize the references to main memory.

Status Control and Registers: These registers cannot be used by the programmers but are used to control the CPU or the execution of a program.

Different vendors have used some of these registers interchangeably; therefore, you should not stick to these definitions rigidly. Yet this categorization will help in better

understanding of register sets of machine. Therefore, let us discuss more about these categories.

2.3.1 Programmer Visible Registers

These registers can be accessed using machine language. In general we encounter 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, in order to simplify the task of programmers and computers dedicated registers can be used. For example, registers may be dedicated to floating point operations. One such common dedication may be the data and address registers.

The data registers are used only for storing intermediate results or data and not for operand address calculation.

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, it has been found that 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 have to 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 later in this 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 then 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. In certain cases two consecutive registers may be used to hold data whose length is double of the register length.

2.3.2 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. One of the control registers for a von-Neumann machine is the Program Counter (PC).

Almost all the CPUs, as discussed earlier, 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 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.

These flags are set by the CPU hardware while performing an operation. For example, an addition operation may set the overflow flag or on a division by 0 the overflow flag can be set etc. These codes may be tested by a program for a typical conditional branch operation. The condition codes are collected in one or more registers. RISC machines have several 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, stack pointer if a stack is used to implement subroutine calls, 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?
.....
.....
.....

3. What is the advantage of having independent set of conditional codes?

.....

.....

.....

3. Can we store status and control information in the memory?

.....

.....

.....

Let us now look into an example register set of MIPS processor.

2.4 GENERAL REGISTERS IN A PROCESSOR

In Block 4 Unit 1, you would be exposed to 8086 registers. In this section we will provide very brief details of registers of a RISC system called MIPS.

MIPS is a register-to-register or load/store architecture and uses three address instructions for data manipulation. It is because of register-register operands that you can have more operands in an instruction of 32 bits, as register address are smaller. The MIPS have 32 addressable registers = $2^5 \Rightarrow 5$ bits register address. The table given below displays the MIPS general purpose registers.

MIPS register names begin with a \$. There are two naming conventions:

- By number:

\$0 \$1 \$2 ... \$31

- By (mostly) two-letter names, such as:

\$a0 - \$a3 \$t0 - \$t7 \$s0 - \$s7 \$gp \$fp \$sp \$ra

Not all of these are general-purpose registers. The following table describes how each general register is treated, and the actions you can take with each register.

Name	Register number	Description	Specify in Expression
ZERO	0	Always has the value 0.	\$zero
AT	1	Reserved for the assembler to handle large constants.	\$at
V0 - V1	2-3	Function value registers. Values for results and expression evaluation.	\$v0 - \$v1
A0 - A3	4-7	Argument registers.	\$a0 - \$a3
T0 - T7	8-15	Temporary registers	\$t0 - \$t7
S0 - S7	16-23	Saved registers	\$s0 - \$s7
T8 - T9	24-25	Temporary registers	\$t8 - \$t9
K0 - K1	26-27	Reserved for the operating system	\$k1 - \$k2

GP	28	Global pointer register	\$gp
SP	29	Stack pointer register	\$sp
FP	30	Frame pointer register	\$fp
RA	31	Return address register	\$ra

You will also study another 8086 based register organization in Block 4 of this course. So, all the computers have a number of registers. But, how exactly is the instruction execution related to registers? To explore this concept, let us first discuss the concept of Micro-operations.

2.5 MICRO-OPERATION CONCEPTS

We have discussed the general architecture and register set of MIPS microprocessor. Our next task is to look at the functionality of ALU, the control unit and how an instruction is executed. In this section, we will define a micro-operation concept, which is the key concept to describe instruction execution.

A micro-operation is an elementary operation performed normally during one clock pulse. On the information stored in one or more registers. The result of the operation may replace the previous content of a register or is transferred to a new register or a memory location.

A digital system performs a sequence of micro-operations on data stored in registers or memory. The specific sequence of micro-operations performed is predetermined for an instruction. Thus, an instruction is a binary code specifying a definite sequence of micro-operations to perform a specific function.

For example, a C program instruction $\text{sum} = \text{sum} + 7$, will first be converted to equivalent assembly program:

- Move data from memory location “sum” to register R1 (LOAD R1, sum)
- Add an immediate operand to register (R1) and store the results in R1 (ADD R1, 7)
- Store data from register R1 to memory location “sum” (STORE sum, R1).

Thus, several machine instructions may be needed (this will vary from machine to machine) to execute a simple C statement. But, how will each of these machine statements be executed with the help of micro-operations? Let us try to elaborate the execution steps:

- Fetch the instructions.
 - Pass the address of Program Counter (PC) to Memory Address Register (MAR).
 - Issue the memory read operation to fetch instruction in the Buffer Register for data, such as M(BR).
 - Increment Program Counter to refer to next instruction in sequence and bring instruction to Instruction Register (IR).
- Execute the instruction
 - Decode the instruction to ascertain operation.
 - As one of the operands is already available in R1 register and the second operand is an immediate operand so fetch operand step is not required. The immediate operand is available in the address part of the instruction.
 - Perform the ALU based addition with R1 and buffer register, store the result in R1.

Thus, we may have to execute the instruction in several steps. For the subsequent discussion, for simplicity, let us assume that each micro-operation can be completed in one clock period, although some micro-operations require memory read/write that may take more time.

Let us first discuss the type of micro-operations. The most common micro-operations performed in a digital computer can be classified into four categories:

- 1) Register transfer micro-operations: simply transfer binary information from one register to another.
- 2) Arithmetic micro-operations: perform simple arithmetic operations on numeric data stored in registers.
- 3) Logic micro-operations: perform bit manipulation (logic) operations on non-numeric data stored in registers.
- 4) Shift micro-operations registers: perform shift operations on data stored in registers.

2.5.1 Register Transfer Micro-operations

These micro-operations, as the name suggests transfer information from one register to another. The information does not change during these micro-operations. A register transfer micro-operation may be designed as: $R1 \leftarrow R2$. The \leftarrow symbol implies that the contents of register R2 are transferred to register R1. R2 here is a source register while R1 is a destination register. We will use this notation throughout this section. Please note the following important points about register transfer micro-operations.

- For a register transfer micro-operation there must be a path for data transfer from the output of the source register to the input of destination register.
- In addition, the destination register should have a parallel load capability, as we expect the register transfer to occur in a predetermined control condition. We will discuss more about the control unit in Unit 4 of this block.
- A common path for connecting various registers is through a common internal data bus of the processor. In general the size of this data bus should be equal to the number of bits in a general register.

The convention used to represent the micro-operations is:

1. Computer register names are designated by capital letters (sometimes followed by numerals) to denote its function. For example, R1, R2 (General Purpose Registers), AR (Address Register), IR (Instruction Register) etc.
2. The individual bits within a register are numbered from 0 (rightmost bit) to n-1 (leftmost bit) as shown in Figure 2b). Common ways of drawing the block diagram of a computer register are shown below. The name of the 16-bit register is IR (Instruction Register) which is partitioned into two subfields in Figure 2d). Bits 0 through 7 are assigned the symbol L (for Low byte) and bits 8 through 15 are assigned the symbol H (for high byte). The symbol IR (L) refers to the low-order byte and IR (H) refers to high-order byte.

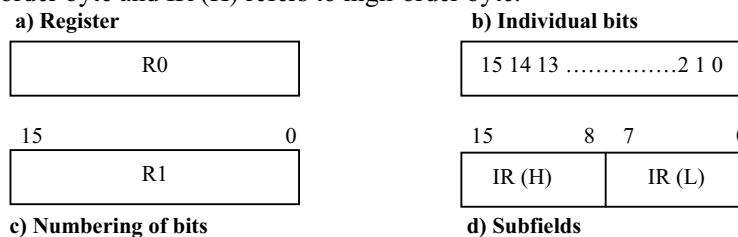


Figure 2: Register Formats

3. Information transfer from one register to another is designated in symbolic notation by a replacement operator. For example, the statement $R2 \leftarrow R1$

denotes a transfer of all bits from the source register R1 to the destination register R2 during one clock pulse and the destination register has a parallel load capacity. However, the contents of register R1 remain unchanged after the register transfer micro-operation. More than one transfer can be shown using a comma operator.

4. If the transfer is to occur only under a predetermined control condition, then this condition can be specified as a control function. For example, if P is a control function then P is a Boolean variable that can have a value of 0 or 1. It is terminated by a colon (:) and placed in front of the actual transfer statement. The operation specified in the statement takes place only when $P = 1$. Consider the statements:

If ($P = 1$) then ($R2 \leftarrow R1$)
or,
P: $R2 \leftarrow R1$,

Where P is a control function that can be either 0 or 1.

5. All micro-operations written on a single line are to be executed at the same time provided the statements or a group of statements to be implemented together are free of conflict. A conflict occurs if two different contents are being transferred to a single register at the same time. For example, the statement: new line X: $R1 \leftarrow R2, R1 \leftarrow R3$ represents a conflict because both R2 and R3 are trying to transfer their contents to R1 at the same time.
6. A clock is not included explicitly in any statements discussed above. However, it is assumed that all transfers occur during the clock edge transition immediately following the period when the control function is 1. All statements imply a hardware construction for implementing the micro-operation statement as shown below:

Implementation of controlled data transfer from R2 to R1 only when $T = 1$
T: $R1 \leftarrow R2$

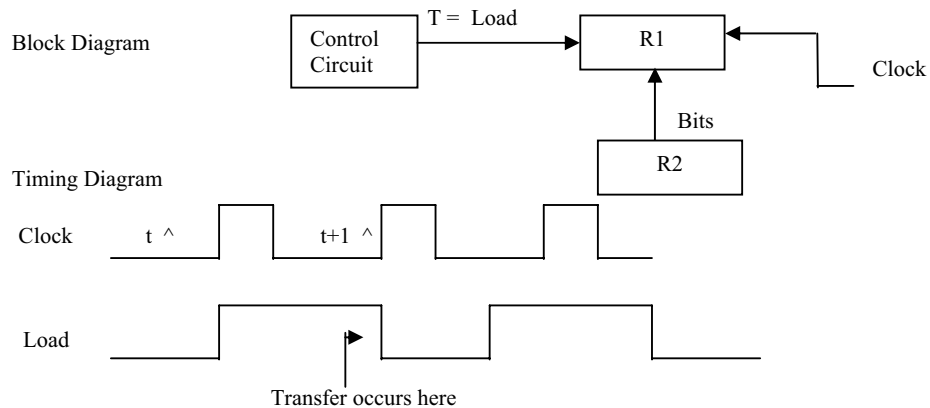


Figure 3: The Register Transfer Time

It is assumed that the control variable is synchronized with the same clock as the one applied to the register. The control function T is activated by the rising edge of the clock pulse at time t. Even though the control variable T becomes active just after time t, the actual transfer does not occur until the register is triggered by the next positive transition of the clock at time t+1. At time t+1, load input is again active and the data inputs of R2 are then loaded into the register R1 in parallel. The transfer occurs with every clock pulse transition while T remains active.

Bus and Memory Transfers

A digital computer has many registers, and rather than connecting wires between all registers to transfer information between them, a common bus is used. Bus is a path

(consists of a group of wires) one for each bit of a register, over which information is transferred, from any of several sources to any of several destinations.

From a register to Bus: $BUS \leftarrow R$. The implementation of bus is explained in Unit 3 of this block.

The transfer from bus to register can be expressed symbolically as:

$$R1 \leftarrow BUS,$$

The content of the selected register is placed on the BUS, and the content of the bus is loaded into register R1 by activating its load control input.

Memory Transfer

The transfer of information from memory to outside world i.e., I/O Interface is called a *read* operation. The transfer of new information to be stored in memory is called a *write* operation. These kinds of transfers are achieved via a system bus. It is necessary to supply the address of the memory location for memory transfer operations.

Memory Read

The memory unit receives the address from a register, called the memory address register designated by MAR. The data is transferred to another register, called the data register designated by DR. The read operation can be stated as:

$$\text{Read: } DR \leftarrow [MAR]$$

Memory Write

The memory write operation transfers the content of a data register to a memory word M selected by the address. Assume that the data of register R1 is to be written to the memory at the address provided in MAR. The write operation can be stated as:

$$\text{Write: } [MAR] \leftarrow R1$$

Please note, it means that the location pointed by MAR will be written and not MAR.

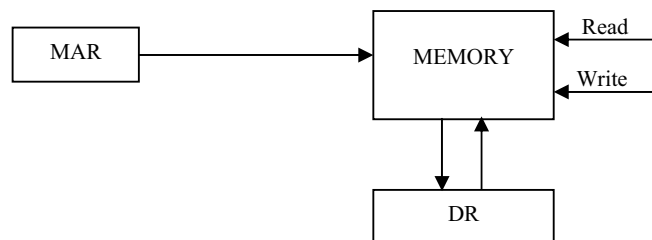


Figure 4: Memory Transfer

2.5.2 Arithmetic Micro-operations

These micro-operations perform simple arithmetic operations on numeric data **stored in registers**. The basic arithmetic micro-operations are addition, subtraction, increment, decrement, and shift.

Addition micro-operation is specified as:

$$R3 \leftarrow R1 + R2$$

It means that the contents of register R1 are added to the contents of register R2 and the sum is transferred to register R3. This operation requires three registers to hold data along with the Binary Adder circuit in the ALU. Binary adder is a digital circuit

that generates the arithmetic sum of two binary numbers of any lengths and is constructed with full-adder circuits connected in cascade. An n-bit binary adder requires n full-adders. Add micro-operation, in accumulator machine, can be performed as:

$$AC \leftarrow AC + DR$$

Subtraction is most often implemented in machines through complement and adds operations. It is specified as:

$$R3 \leftarrow R1 - R2$$

$$R3 \leftarrow R1 + (2\text{'s complement of } R2)$$

$$R3 \leftarrow R1 + (1\text{'s complement of } R2 + 1)$$

$$R3 \leftarrow R1 + \overline{R2} + 1 \quad (\text{The bar on top of } R2 \text{ implies } 1\text{'s complement of } R2 \text{ which is bitwise complement})$$

Adding 1 to the 1's complement produces the 2's complement. Adding the contents of R1 to the 2's complement of R2 is equivalent to subtracting the contents of R2 from R1 and storing the result in R3. We will describe the basic circuit required for these micro-operations in the next unit.

The increment micro-operation adds one to a number in a register. This operation is designated as:

$$R1 \leftarrow R1 + 1$$

This can be implemented in hardware by using a binary-up counter.

The decrement micro-operation subtracts one from a number in a register. This operation is designated as:

$$R1 \leftarrow R1 - 1$$

This can be implemented using binary-down counter.

What about the multiply and division operations? Are not they micro-operations? In most of the older computers multiply and divisions were implemented using add/subtract and shift micro-operations. If a digital system has implemented division and multiplication by means of combinational circuits, then we can call these as the micro-operations for that system.

2.5.3 Logic Micro-operations

Logic operations are basically binary operations, which are performed on the string of bits stored in the registers. For a logic micro-operation each bit of a register is treated as a variable. A logic micro-operation:

$R1 \leftarrow R1.R2$ specifies AND operation to be performed on the contents of R1 and R2 and store the results in R1. For example, if R1 and R2 are 8 bits registers and:

R1 contains 10010011 and

R2 contains 01010101

Then R1 will contain 00010001 after AND operation.

Some of the common logic micro-operations are AND, OR, NOT or Complement, Exclusive OR, NOR, and NAND. In many computers only four: AND, OR, XOR (exclusive OR) and complement micro-operations are implemented.

Let us now discuss how these four micro-operations can be used in implementing some of the important applications of manipulation of bits of a word, such as, changing some bit values or deleting a group of bits. We are assuming that the result

of logic micro-operations go back to Register R1 and R2 contains the second operand.

We will play a trick with the manipulations we are performing. Let us select 1010 as 4 bit data for register R1, and 1100 data for register R2. Why? Because if you see the bit combinations of R2, and R1, they represent the truth table entries (read from right to left and bottom to top) 00, 01, 10 and 11. Thus, the resultant of the logical operation on them will indicate which logic micro-operation is needed to be performed for that data manipulation. The following table gives details on some of these operations:

R1	1	0	1	0
R2	1	1	0	0

Operation name	What is the operation?	Example and Explanation
Selective Set	Sets those bits in Register R1 for which the corresponding R2 bit is 1.	R1 = 1010 R2 = 1100 <div>1110</div> The value 1110 suggests that selective set can be done using logic OR micro-operation. Please note that all those bits of R1, for which we have 0 bit in R2, have remained unchanged. The bits in R1 which need to be set selectively must have the corresponding R2 bits as 1.
Selective Clear	Clear those bits in register R1 for which corresponding R2 bits are 1.	R1 = 1010 R2 = 1100 <div>0010</div> The R1 value after the operation is 0010 which suggests that Corresponding micro-operation is $R1 \text{ AND } \overline{R2}$
Selective Complement	Complement those bits in register R1 for which the corresponding R2 bits are 1.	R1 = 1010 R2 = 1100 <div>0110</div> The R1, value 0110 after the operation suggests that the selective complement can be done using exclusive - OR micro-operation. The bits in R1 which need to be complemented selectively must have the corresponding R2 bits as 1.
Mask Operations	Clears those bits in Register R1 for which the corresponding R2 bits are 0.	R1 = 1010 R2 = 1100 <div>1000</div> The R1 value after the operation is 1000 which suggests that the mask operation can be performed using AND micro-operation. However, the bits in R1 which are cleared or masked correspond to the bits on R2 having a 0 value. The mask operation is preferred over selective clear as most of the computers provide AND micro-operation while the micro-operation required for implementing selective clear is normally not provided in computers
Insert	For inserting a new value in a bit. It is a two-step process: <i>Step 1:</i> Mask out the existing bit value	This is a two-step process. <i>Example:</i> Say contents of R1 = 0011 1011 Suppose, we want to insert 0110 in place of left

	Step 2: Insert the bit using OR micro-operation with the bits which are to be inserted.	most 0011 then: 0011 1011 (R1 before) 0000 1111 (R2 for masking) ————— Perform AND operation (mask) 0000 1011 (R1 after) Now insert: 01100000 (R2 for insertion) ————— Perform OR operation 0110 1011 R1 after insert
Clear	Clear all the bits	R1 = 1101 R2 = 1101 <div style="border: 1px solid black; padding: 2px; display: inline-block;">0000</div> Implemented by taking exclusive OR with the same number. The exclusive OR, thus, can also be used for checking whether two numbers are equal or not.

2.5.4 Shift Micro-operations

Shift is a useful operation, which can be used for serial transfer of data. Shift operations can also be used along with other (arithmetic, logic, etc.) operations. For example, for implementing a multiply operation arithmetic micro-operation (addition) can be used along with shift operation. The shift operation may result in shifting the contents of a register to the left or right. In a shift left operation a bit of data is input at the right most flip-flop while in shift right a bit of data is input at the left most flip-flop. In both the cases a bit of data enters the shift register. Depending on what bit enters the register and where the shift out bit goes, the shifts are classified in three types. These are:

- logical
- arithmetic and
- circular.

In logical shift the data entering by serial input to left most or right most flip-flop (depending on right or left shift operations respectively) is a 0.

If we connect the serial output of a shift register to its serial input then we encounter a circular shift. In circular shift left or circular shift right information is not lost, but is circulated.

In arithmetic shift a signed binary number is shifted to the left or to the right. Thus, an arithmetic shift-left causes a number to be multiplied by 2, on the other hand a shift-right causes a division by 2. But as in division or multiplication by 2 the sign of a number should not be changed, therefore, arithmetic shift must leave the sign bit unchanged. We have already discussed about shift operations in the Unit 1.

Let us summarize micro-operations using the following table:

Sl. No.	Micro-operations	Examples
1.	Register transfer	$R1 \leftarrow R2$ (register transfer) $[MAR] \leftarrow R1$ (Register to memory)
2.	Arithmetic micro-operations	$ADD R1 \leftarrow R1 + R2$ $SUBTRACT R1 \leftarrow R1 + (\overline{R2} + 1)$ $INCREMENT R1 \leftarrow R1 + 1$ $DECREMENT R1 \leftarrow R1 - 1$
3.	Logical micro operations	AND OR COMPLEMENT XOR
4.	Shift	Left or right shift

		<ul style="list-style-type: none"> • Logical • Arithmetic • Circular
--	--	---

Check Your Progress 2

- How does the memory read / operation carried out using system bus?
.....
.....
.....
- Are multiplication and division arithmetic operations micro-operations?
.....
.....
.....
- What will be the value for R2 operand if:
 - Mask operation clears register R1
 - Bits 1011 0001 is to be inserted in an 8 bit R1 register.
- What are the differences between circular and logical shift micro-operations?
.....
.....
.....
.....

2.6 INSTRUCTION EXECUTIONS AND MICRO - OPERATIONS

Let us now discuss instruction execution using the micro-operations. A simple instruction may require:

- Instruction fetch: fetching the instruction from the memory.
- Instruction decode: decode the instruction.
- Operand address calculation: find out the effective address of the operands.
- Execution: execute the instruction.
- Interrupt Acknowledge: perform an interrupt acknowledge cycle if an interrupt request is pending.

Let us explain how these steps of instruction execution can be broken down to micro-operations. For simplifying the discussion, let us assume that the machine has the structure as shown in Figure 1. In addition, let us also assume that the instruction set of the machine has only two addressing modes direct and indirect memory addresses and a memory access take same time as that of a register access that is one clock cycle.

Instruction fetch: In this phase the instruction is brought from the address pointed by PC to instruction register. The steps required are:

Transfer the address of PC to MAR. (Register Transfer)	$MAR \leftarrow PC$
MAR puts its contents on the address bus for main memory location selection, the control unit instructs the MAR to do so and also uses a memory read signal. The word so read is placed on the data bus where it is accepted by the Data register (Memory-read using bus.	$DR \leftarrow (MAR), PC \leftarrow PC + 1$

It may take more than one clock pulses depending on the t_{cpu} and t_{mem} . The PC is incremented by one memory word length to point to the next instruction in sequence. This micro-operation can be carried out in parallel to the micro-operation above.	
The instruction so obtained is transferred from data register to the Instruction register for further processing. (Register Transfer)	$IR \leftarrow DR$

Instruction Decode: This phase is performed under the control of the Control Unit of the computer. The Control Unit determines the operation that is to be performed and the addressing mode of the data. In our example, the addressing modes can be direct or indirect.

Operand Address Calculation: In actual machines the effective address may be a memory address, register or I/O port address. The register reference instructions such as complement R1, clear R2 etc. normally do not require any memory reference (assuming register indirect addressing is not being used) and can directly go to the execute cycle. However, the memory reference instruction can use several addressing modes. Depending on the type of addressing the effective address (EA) of operands in the memory is calculated. The calculation of effective address may require more memory fetches (for example in the case of indirect addressing), thus in this step we may calculate the effective address as:

For Direct Address: <ul style="list-style-type: none"> Transfer the address portion of instruction is the direct address so no further calculation is needed. 	IR (Address) and DR (Address) contain the Effective address.
For Indirect Address: <ul style="list-style-type: none"> Transfer the address bits of instruction to the MAR. This transfer can be achieved using DR, as DR and IR at this point of time contain the same value. (Register Transfer) Perform a memory read operation as done in fetch cycle and the desired address of the operand is obtained in the DR. (Memory Read) Transfer the address part so obtained in DR as the address part of instruction. (Register Transfer) Thus, the indirect address is now converted to direct address or effective address. 	$MAR \leftarrow DR \text{ (Address)}$ $DR \leftarrow (MAR)$ $IR \text{ (Address)} \leftarrow DR \text{ (Address)}$

Thus, the address portion of IR now contains the effective address, which is the direct address of the operand.

Execution: Now the instruction is ready for execution. A different opcode will require different sequence of steps for the execution. Therefore, let us discuss a few examples of execution of some simple instructions for the purpose of identifying some of the steps needed during instruction execution. Let us start the discussions with a simple case of addition instruction. Suppose, we have an instruction: Add R1, A which adds the content of memory location A to R1 register storing the result in R1. This instruction will be executed in the following steps:

Transfer the address portion of the instruction to the MAR. (Register transfer)	$MAR \leftarrow IR \text{ (Address)}$
Read the memory location A and bring the operand in the DR. (Memory read)	$DR \leftarrow (MAR)$
Add the DR with R1 using ALU and bring the results back to R1. (Add micro-operations)	$R1 \leftarrow R1 + DR$

Now, let us try a complex instruction - a conditional jump instruction. Suppose an instruction:

INCSKIP A

increments A and skips the next instruction if the content of A has become zero. This is a complex instruction and requires intermediate decision-making. The micro operations required for this instruction execution are:

Transfer the address portion of IR to the MAR. (Register transfer)	$MAR \leftarrow IR \text{ (Address)}$
Read memory. DR on reading will contain the operand A. (Memory read)	$DR \leftarrow (MAR)$
Transfer the contents of DR to R1. We are assuming that DR, although it can be used in computation, it cannot be used as destination of an ALU operation. Thus, we need to transfer its content to a general purpose register R1 where the operation can be performed. (Register transfer)	$R1 \leftarrow DR$
Increment the R1. (Increment micro-operation)	$R1 \leftarrow R1 + 1$
Transfer the content of R1 to DR. (Register transfer)	$DR \leftarrow R1$
Store the contents of DR- into the location A using MAR. This operation proceeds through as: Address bits are applied on address bus by MAR. The data is put into the data bus. The control unit providing control signal for memory write, thus resulting in a memory write at a location specified by MAR. (Memory write)	$(MAR) \leftarrow DR$
If the content of R1 is zero then increment PC by one, thus skipping the next instruction. This operation can be performed in parallel to the memory write. Please note in the last step a comparison and an action is taken as a single step. This is possible as it is a simple comparison based on status flags. (Increment on a condition)	If $R1 = 0$ then $PC \leftarrow PC + 1$

Let us now take an example of branching operation. Suppose we are using the first location of subroutine to store the return address, then the steps involved in this subroutine call (CALL A) can be:

Transfer the contents of address portion of IR to MAR. (Register Transfer) Transfer the return address , that is, the contents of PC to DR. This micro-operation can be performed in parallel to the previous micro-operation. (Register transfer)	$MAR \leftarrow IR \text{ (Address)},$ $DR \leftarrow PC$
Transfer the branch address that is stored in Address part of the instruction to program counter. (Register transfer)	$PC \leftarrow IR \text{ (Address)}$
Store the DR using MAR. Thus, the return address is stored at the first location of the subroutine. (This operation normally is done in stack, but in this example we are storing the return address in the first location of the subroutine). This micro-operation can be performed in parallel to previous micro-	$(MAR) \leftarrow DR$

operation. (Memory write)	
Increment the PC as it contains the first location of subroutine, which is used to store the return address. The first instruction of subroutine starts from the next location. (Increment)	$PC \leftarrow PC + 1$

Thus, the number of steps required in execution may differ from instruction to instruction.

Interrupt Processing: On completion of the execution of an instruction, the machine checks whether there is any pending interrupt request for the interrupts that are enabled. If an enabled interrupt has occurred then that Interrupt may be processed. The nature of interrupt varies from machine to machine. However, let us discuss one simple illustration of interrupt processing events. A simple sequence of steps followed in interrupt phase is:

Transfer the contents of PC to DR, as this is the return address after the interrupt service program has been executed. This address must be saved.	$DR \leftarrow PC$
Place the address of location, where the return address is to be saved, into MAR. Please note that this address is normally predetermined in computers.	$MAR \leftarrow$ Address of location for saving return address.
Store the contents of PC in the memory using DR and MAR. (Memory write) Transfer the address of the first instruction of interrupt servicing routine to the PC. This micro-operation can be performed in parallel to the above micro-operation.	$(MAR) \leftarrow DR$ $PC \leftarrow$ address of the first instruction interrupt service programs

After completing the above interrupt processing, CPU will fetch the next instruction that may be interrupt service program instruction. Thus, during this time CPU might be doing the interrupt processing or executing the user program. Please note each instruction of interrupt service program is executed as an instruction in an instruction cycle.

Please note for a complex machine the instruction cycle will not be as easy as this. You can refer to further readings for more complex instruction cycles.

2.7 INSTRUCTION PIPELINING

After discussing instruction execution, let us now define a concept that is very popular in any CPU implementation. This concept is instruction pipeline.

To extract better performance, as defined earlier, instruction execution can be done through instruction pipeline. The instruction pipelining involves decomposing of an instruction execution to a number of pipeline stages. Some of the common pipeline stages can be instruction fetch (IF), instruction decode (ID), operand fetch (OF), execute (EX), store results (SR). An instruction pipe may involve any combination of such stages. A major design decision here is that the instruction stages should be of equal execution time. Why?

A pipeline allows overlapped execution of instructions. Thus, during the course of execution of an instruction the following may be a scenario of execution.

Time Slot -	1	2	3	4	5	6	7	8	9	10	11
>											
Instruction	IF	ID	OF	EX	SR						

1											
Instruction 2		IF	ID	OF	EX	SR					
Instruction 3			IF	ID	OF	EX	SR				
Instruction 4				IF	ID	OF	EX	SR			
Instruction 5					IF	ID	OF	EX	SR		
Instruction 6						IF	ID	OF	EX	SR	
Instruction 7							IF	ID	OF	EX	SR

Figure 5: Instruction Pipeline

Please note the following observations about the above figure:

- The pipeline stages are like steps. Thus, a step of the pipeline is to be complete in a time slot. The size of the time slot will be governed by the stage taking maximum time. Thus, if the time taken in various stages is almost similar, we get the best results.
- The first instruction execution is completed on completion of 5th time slot, but afterwards, in each time slot the next instruction gets executed. So, in ideal conditions one instruction is executed in the pipeline in each time slot.
- Please note that after the 5th time slot and afterwards the pipe is full. In the 5th time slot the stages of execution of five instructions are:

SR (instruction 1) (Requires memory reference)
EX (instruction 2) (No memory reference)
OF (instruction 3) (Requires memory reference)
ID (instruction 4) (No memory reference)
IF (instruction 5) (Requires memory reference)

The Pipelining Problems:

- On the 5th time slot and later, there may be a register or memory conflict in the instructions that are performing memory and register references that is various stages may refer to same registers/memory location. This will result in slower execution instruction pipeline that is one of the higher number instruction has to wait till the lower number instructions completed, effectively pushing the whole pipelining by one time slot.
- Another important situation in Instruction Pipeline may be the branch instruction. Suppose that instruction 2 is a conditional branch instruction, then by the time the decision to take the branch is taken (at time interval 5) three more instructions have already been fetched. Thus, if the branch is to be taken then the whole pipeline is to be emptied first. Thus, in such cases, pipeline cannot run at full load.

How can we minimize the problems occurring due to the branch instructions?

We can use many mechanisms that may minimize the effect of branch penalty.

- To keep multiple streams in pipeline in case of branch
- Pre-fetching the next as well as instruction to which branch is to take place
- A loop buffer may be used to store the instructions of a loop instruction
- Predicting whether the branch will take place or not and acting accordingly
- Delaying the pipeline fill up till the branch decision is taken.

Check Your Progress 3

State True or False

- 1) An instruction cycle does not include indirect cycle if the operands are stored in the register.

T	F
---	---

☐
- 2) Register transfer micro-operations are not needed for instruction execution. ☐
- 3) Interrupt cycle results only in jumping to an interrupt service routine. The actual processing of the instructions of this routine is performed in instruction cycle. ☐

2.8 SUMMARY

In this unit, we have discussed in detail the register organisation and a simple structure of the CPU. After this we have discussed in details the micro-operations and their implementation in hardware using simple logical circuits. While discussing micro-operations our main emphasis was on simple arithmetic, logic and shift micro-operations, in addition to register transfer and memory transfer. The knowledge you have acquired about register sets and conditional codes, helps us in giving us an idea that conditional micro-operations can be implemented by simply checking flags and conditional codes. This idea will be clearer after we go through Unit 3 and Unit 4. We have completed the discussions on this unit, with providing a simple approach of instruction execution with micro-operations. We have also defined the concepts of Instruction Pipeline. We will be using this approach for discussing control unit details in Unit 3 and Unit 4. The following table gives the details of various terms used in this unit.

General purpose registers	These registers are used for any address or data computation / storage
Status and control register	Stores the various condition codes
Programmer visible registers	Used by programmers during programming
Micro-operations	Involves register transfer micro operations arithmetic micro-operations like add, subtract, logic micro-operations like AND, OR, NOT, XOR and shift micro-operations left or right shift
Micro-operations and instruction execution	An instruction is executed through a sequence of micro-operations. Thus, a program is executed as a sequence of instruction is executed when a sequence of microinstructions are executed.
Instruction pipeline	Allows overlapped execution of instructions. A good pipe can produce one instruction per clock cycle.

You will also get the details on 8086 microprocessor register sets, conditional codes, instructions etc. in Unit 1 of Block 4.

You can refer to further readings for more register organisation examples and for more details on micro-operations and instruction execution.

2.9 SOLUTIONS /ANSWERS

Check Your Progress 1

1. Registers, which are used only for the calculation of operand addresses, are called address registers.
2. 5 bits
3. It helps in implementing parallelism in the instruction execution unit.
4. Yes. Normally, the first few hundreds of words of memory are allocated for storing control information.

Check Your Progress 2

1. Read operation involves reading of location pointed to by MAR. The address bus is loaded with the contents of MAR

$$\text{address BUS} \leftarrow \text{MAR}$$
 In addition a read signal is issued by control unit, and data is stored to MBR register or data register.

$$\text{DR} \leftarrow \text{data BUS}$$
 The combined operation can be shown as

$$\text{DR} \leftarrow [\text{MAR}]$$
2. Yes, if implemented through circuits.
 No, if implemented through algorithms involving add/ subtract and shift micro-operations.
3. (i) 0000 0000
 (ii) Initially AND with 0000 0000 followed by OR with 1011 0001
4. The bits circulate and after a complete cycle the data is still intact in circular shift. Not so in logical shift.

Check Your Progress 3

1. True
2. False
3. True

UNIT 3 ALU ORGANISATION

Structure	Page No.
3.0 Introduction	53
3.1 Objectives	53
3.2 ALU Organisation	53
3.2.1 A Simple ALU Organization	
3.2.2 A Sample ALU Design	
3.3 Arithmetic Processors	62
3.4 Summary	63
3.5 Solutions/ Answers	64

3.0 INTRODUCTION

By now we have discussed the instruction sets and register organisation followed by a discussion on micro-operations and instruction execution. In this unit, we will first discuss the ALU organisation. Then we will discuss the floating point ALU and arithmetic co-processors, which are commonly used for floating point computations.

This unit provides a detailed view on implementation of simple micro-operations that include register-transfer, arithmetic, logic and shift micro-operation. Finally, the construction of a simple ALU is given. Thus, this unit provides you the basic insight into the computer system. The next unit covers details of the control unit. Together these units describe the two most important components of CPU: the ALU and the CU.

3.1 OBJECTIVES

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

- describe the basic organisation of ALU;
- discuss the requirements of a floating point ALU;
- define the term arithmetic coprocessor; and
- create simple arithmetic logic circuits.

3.2 ALU ORGANISATION

As discussed earlier, 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. The simple ALUs can be constructed for fixed-point numbers. On the other hand the floating-point arithmetic implementation requires more complex control logic and data processing capabilities, i.e., the hardware. 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 co-processor. Let us discuss all these issues in greater detail in this section.

3.2.1 A Simple ALU Organisation

An ALU consists of circuits that perform data processing micro-operations. But how are these ALU circuits used in conjunction of other registers and control unit? The simplest organisation in this respect for fixed point ALU was suggested by John von Neumann in his IAS computer design (Please refer to Figure 1).

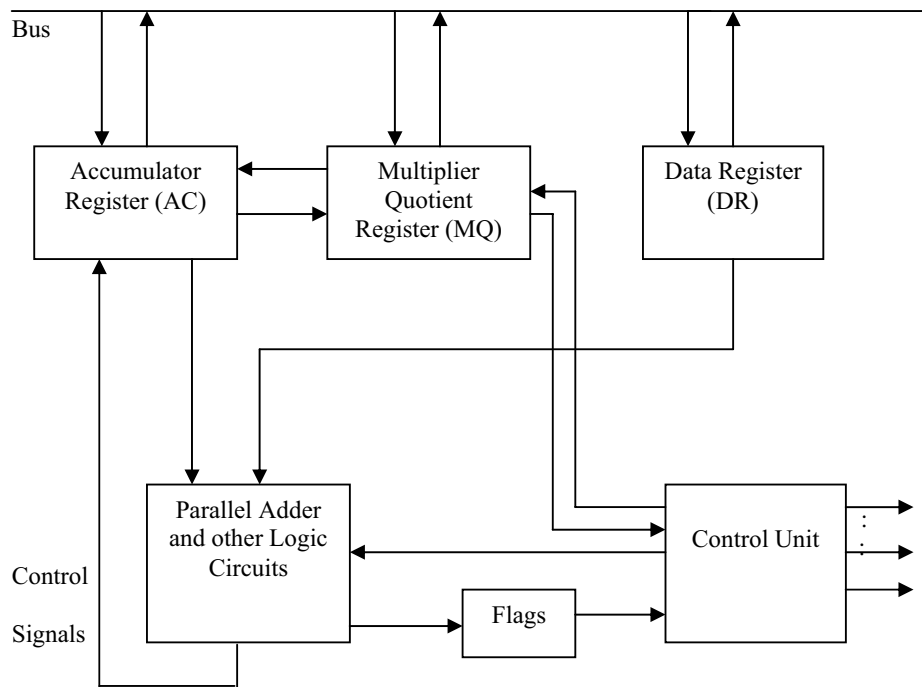


Figure 1: Structure of a Fixed point Arithmetic logic unit

The above structure has three registers AC, MQ and DR for data storage. Let us assume that they are equal to one word each. Please note that the Parallel adders and other logic circuits (these are the arithmetic, logic circuits) have two inputs and only one output in this diagram. 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 present case 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 : $AC \leftarrow AC (+) DR$
 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. We are not giving the details of how this register can be used for implementing multiplication and division algorithms. For more details on these algorithms please refer to further readings. 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. These operations can be represented as:

Multiplication : $AC.MQ \leftarrow DR \times MQ$
 Division : $AC.MQ \leftarrow MQ \div DR$

DR is another important register, which is used for storing second operand. In fact it acts as a buffer register, which stores the data brought from the memory for an instruction. In machines where we have general purpose registers any of the registers can be utilized as AC, MQ and DR.

Bit Slice ALUs

It was feasible to manufacture smaller such as 4 or 8 bits fixed point ALUs on a single IC chip. If these chips are designed as expendable types then using these 4 or 8 bit ALU chips we can make 16, 32, 64 bit array like circuits. These are called bit-slice ALUs.

The basic advantage of such ALUs is that these ALUs can be constructed for a desired word size. More details on bit-slice ALUs can be obtained from further readings.

Check Your Progress 1

T	F
---	---

State True or False

1. A multiplication operation can be implemented as a logical operation. ☐
2. The multiplier-quotient register stores the remainder for a division operation. ☐
3. A word is processed sequentially on a bit slice ALU. ☐

3.2.2 A Sample ALU Design

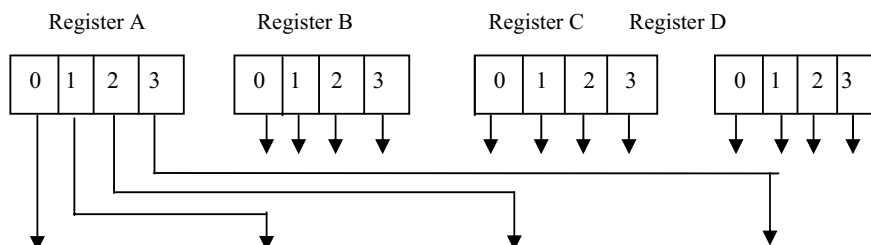
The basis of ALU design starts with the micro-operation implementation. So, let us first explain how the bus can be used for Data transfer micro-operations.

A digital computer has many registers, and rather than connecting wires between all registers to transfer information between them, a common bus is used. Bus is a path (consists of a group of wires) one for each bit of a register, over which information is transferred, from any of several sources to any of several destinations. In general the size of this data bus should be equal to the number of bits in a general purpose register.

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, is made using the multiplexers. The select lines are connected to the control inputs of the multiplexers and the bits of one register are chosen thus allowing multiplexers to select a specific source register for data transfer.

The construction of a bus system for four registers using 4×1 multiplexers is shown below. Each register has four bits, numbered 0 through 3. Each multiplexer has 4 data inputs, numbered 0 through 3, and two control or selection lines, C_0 and C_1 . The data inputs of 0^{th} MUX are connected to the corresponding 0^{th} input of every register to form four lines of the bus. The 0^{th} multiplexer multiplexes the four 0^{th} bits of the registers, and similarly for the three other multiplexers.

Since the same selection lines C_0 and C_1 are connected to all multiplexers, therefore they choose the four bits of one register and transfer them into the four-line common bus.



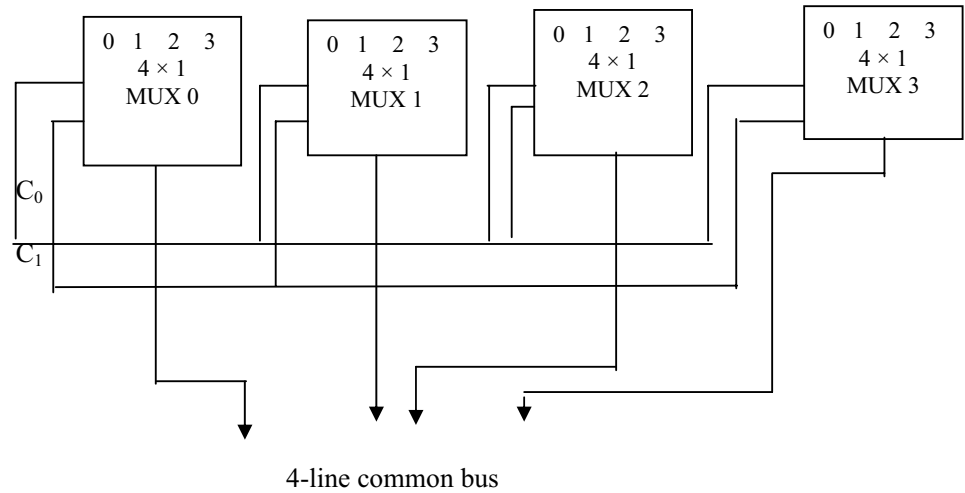


Figure 2: Implementation of BUS

When $C_1 C_0 = 00$, the 0th data input of all multiplexers are selected and this causes the bus lines to receive the content of register A since the outputs of register A are connected to the 0th data inputs of the multiplexers which is then applied to the output that forms the bus. Similarly, when $C_1 C_0 = 01$, register B is selected, and so on. The following table shows the register that is selected for each of the four possible values of the selection lines:

C_1	C_0	Register Selected
0	0	A
0	1	B
1	0	C
1	1	D

Figure 3: Bus Line Selection

To construct a bus for 8 registers of 16 bits each, you would require 16 multiplexers, one for each line in the bus. The number of multiplexers needed to construct the bus is equal to the number of bits in each register. Each multiplexer must have eight data input lines and three selection lines ($2^3 = 8$) to multiplex one bit in the eight registers.

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. Figure 4 shows a logical implementation of a 4-bit arithmetic circuit. The circuit is constructed by using 4 full adders and 4 multiplexers.

Figure 4: A Four-bit arithmetic circuit

The diagram of a 4-bit arithmetic circuit has four 4×1 multiplexers and four full adders (FA). Please note that the FULL ADDER is a circuit that can add two input bits and a carry-in bit to produce one sum-bit and a carry-out-bit.

So what does the adder do? It just adds three bits. What does the multiplexer do? It controls one of the input bits. Thus, such combination produces a series of micro-operations.

Let us find out how the multiplexer control lines will change one of the Inputs for Adder circuit. Please refer to the following table. (Please note the convention VALID ONLY FOR THE TABLE are that an uppercase alphabet indicates a Data Word, whereas the lowercase alphabet indicates a bit.)

Control Input	Output of 4×1 Multiplexers	Y input to	Comments
---------------	-------------------------------------	------------	----------

S ₁	S ₀	MUX(a)	MUX(b)	MUX(c)	MUX(d)	Adder	
0	0	b ₀	b ₁	b ₂	b ₃	B	The data word B is input to Full Adders
0	1	$\overline{b_0}$	$\overline{b_1}$	$\overline{b_2}$	$\overline{b_3}$	\overline{B}	1's complement of B is input to Full Adders
1	0	0	0	0	0	0	Data word 0 is input to Full Adders
1	1	1	1	1	1	F _H	Data word 1111 = F _H is input to Full Adders

Figure 5: Multiplexer Inputs and Output of the Arithmetic Circuit of Figure 4

Now let us discuss how by coupling carry bit (C_{in}) with these input bits we can obtain various micro-operations.

Input to Circuits

- Register A bits as a₀, a₁, a₂ and a₃ in the corresponding X bits of the Full Adder (FA).
- Register B bits as given in the Figure 5 above as in the corresponding Y bits of the FA.
- Please note each bit of register A and register B is fed to different full adder unit.
- Please also note that each of the four inputs from A are applied to the X inputs of the binary adder and each of the four inputs from B are connected to the data inputs of the multiplexers. It means that the A input directly goes to adder but B input can be manipulated through the Multiplexer to create a number of different input values as given in the figure above. The B inputs through multiplexers are controlled by two selection lines S₁ and S₀. Thus, using various combinations of S₁ and S₀ we can select data bits of B, complement of B, 0 word, or word having All 1's.
- The input carry C_{in}, which can be equal to 0 or 1, goes to the carry input of the full adder in the least significant position. The other carries are cascaded from one stage to the next. Logically it is the same as that of addition performed by us. We do pass the carry of lower digits addition to higher digits. The output of the binary adder is determined from the following arithmetic sum:

$$D = X + Y + C_{in}$$

OR

$$D = A + Y + C_{in}$$

By controlling the value of Y with the two selection lines S₁ and S₀ and making C_{in} equal to 0 or 1, it is possible to implement the eight arithmetic micro-operations listed in the truth table.

S ₁	S ₀	C _{in}	Y val	D = A+Y+C _{in}	Equivalent Micro-Operation	Micro-Operation Name
0	0	0	B	D = A + B	R ← R1 + R2	Add

0	0	1	B	$D = A + B + 1$	$R \leftarrow R1 + R2 + 1$	Add with carry
0	1	0	\bar{B}	$D = A + B$	$R \leftarrow R1 + \bar{R2}$	Subtract with borrow
0	1	1	\bar{B}	$D = A + \bar{B} + 1$	$R \leftarrow R1 + 2\text{'s complement of } R2$	Subtract
1	0	0	0	$D = A$	$R \leftarrow R1$	Transfer
1	0	1	0	$D = A + 1$	$R \leftarrow R1 + 1$	Increment
1	1	0	1	$D = A - 1$	$R \leftarrow R1 + (\text{All } 1\text{'s})$	Decrement
1	1	1	1	$D = A$	$R \leftarrow R1$	Transfer

Figure 6: Arithmetic Circuit Function Table

Let us refer to some of the cases in the table above.

When $S_1S_0 = 00$, input line B is enabled and its value is applied to the Y inputs of the full adder. Now,

If input carry $C_{in} = 0$, the output will be $D = A + B$
 If input carry $C_{in} = 1$, the output will be $D = A + B + 1$.

When $S_1S_0 = 01$, the complement of B is applied to the Y inputs of the full adder. So If $C_{in} = 1$, then output $D = A + \bar{B} + 1$. This is called subtract micro-operation. (Why?)

Reason: Please observe the following example, where $A = 0111$ and $B = 0110$, then $\bar{B} = 1001$. The sum will be calculated as:

$$\begin{array}{r}
 0111 \quad (\text{Value of } A) \\
 1001 \quad (\text{Complement of } B) \\
 1 \quad 0000 + (\text{Carry in } = 1) = 0001
 \end{array}$$

Ignore the carry out bit. Thus, we get simple subtract operation.

If $C_{in} = 0$, then $D = A + \bar{B}$. This is called subtract with borrow micro-operation. (Why?). Let us look into the same addition as above:

$$\begin{array}{r}
 0111 \quad (\text{Value of } A) \\
 1001 \quad (\text{Complement of } B) \\
 1 \quad 0000 + (\text{Carry in } = 0) = 0000
 \end{array}$$

This operation, thus, can be considered as equivalent to:

$$\begin{aligned}
 D &= A + \bar{B} \\
 \Rightarrow D &= (A - 1) + (\bar{B} + 1) \\
 \Rightarrow D &= (A - 1) + 2\text{'s complement of } B \\
 \Rightarrow D &= (A - 1) - B \quad \text{Thus, is the name complement with Borrow}
 \end{aligned}$$

When $S_1S_2 = 10$, input value 0 is applied to Y inputs of the full adder.

If $C_{in} = 0$, then output $D = A + 0 + C_{in} \Rightarrow D = A$
 If $C_{in} = 1$, then $D = A + 0 + 1 \Rightarrow D = A + 1$

The first is a simple data transfer micro-operation; while the second is an increment micro-operation.

When $S_1S_2 = 11$, input word all 1's is applied to Y inputs of the full adder.

If $C_{in} = 0$, then output $D = A + \text{All } (1\text{'s}) + C_{in} \Rightarrow D = A - 1$ (How? Let us explain with the help of the following example).

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.

$$\text{If } C_{in} = 1, \text{ then } D = A + \text{All}(1s) + 1 \Rightarrow D = A$$

The first is the decrement micro-operation; while the second is a data transfer micro-operation.

Please note that the micro-operation $D = A$ is generated twice, so there are only seven distinct micro-operations possible through the proposed arithmetic circuit.

Implementation of Logic Micro-operations

For implementation, let us first ask the questions how many logic operations can be performed with two binary variables. We can have four possible combinations of input of two variables. These are 00, 01, 10, and 11. Now, for all these 4 input combinations we can have $2^4 = 16$ output combinations of truth-values for a function. This implies that for two variables we can have 16 logical operations. The above stated fact will be clearer by going through the following figure.

I_3	I_2	I_1	I_0	Function	Operation	Comments
0	0	0	0	$F_0 = 0$	$R \leftarrow 0$	Clear
0	0	0	1	$F_1 = x \cdot y$	$R \leftarrow R_1 \wedge R_2$	AND
0	0	1	0	$F_2 = x \cdot \bar{y}$	$R \leftarrow R_1 \wedge \bar{R}_2$	R_1 AND with complement R_2
0	0	1	1	$F_3 = x$	$R \leftarrow R_1$	Transfer of R_1
0	1	0	0	$F_4 = \bar{x} \cdot y$	$R \leftarrow \bar{R}_1 \wedge R_2$	R_2 AND with complement R_1
0	1	0	1	$F_5 = y$	$R \leftarrow R_2$	Transfer of R_2
0	1	1	0	$F_6 = x \oplus y$	$R \leftarrow R_1 \oplus R_2$	Exclusive OR
0	1	1	1	$F_7 = x + y$	$R \leftarrow R_1 \vee R_2$	OR
1	0	0	0	$F_8 = \overline{(x + y)}$	$R \leftarrow \overline{(R_1 \vee R_2)}$	NOR
1	0	0	1	$F_9 = \overline{(x \oplus y)}$	$R \leftarrow \overline{(R_1 \oplus R_2)}$	Exclusive NOR
1	0	1	0	$F_{10} = \bar{y}$	$R \leftarrow \bar{R}_2$	Complement of R_2
1	0	1	1	$F_{11} = x + \bar{y}$	$R \leftarrow R_1 \vee \bar{R}_2$	R_1 OR with complement R_2
1	1	0	0	$F_{12} = \bar{x}$	$R \leftarrow \bar{R}_1$	Complement of R_1
1	1	0	1	$F_{13} = \bar{x} + y$	$R \leftarrow \bar{R}_1 \vee R_2$	R_2 OR with complement R_1
1	1	1	0	$F_{14} = \overline{(x \cdot y)}$	$R \leftarrow \overline{(R_1 \wedge R_2)}$	NAND
1	1	1	1	$F_{15} = 1$	$R \leftarrow \text{All } 1's$	Set all the Bits to 1

Figure 7: Logic micro-operations on two inputs

Please note that in the figure above the micro-operations are derived by replacing the x and y of Boolean function with registers R1 and R2 on each corresponding bit of the registers R1 and R2. Each of these bits will be treated as binary variables.

In many computers only four: AND, OR, XOR (exclusive OR) and complement micro-operations are implemented. The other 12 micro-operations can be derived

from these four micro-operations. Figure 8 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

(b) Functional representation

Figure 8: Logic diagram of one stage of logic circuit

Implementation of a Simple 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 unit but sometimes it can be made as a part of overall ALU. The following figure gives a simple structure of one stage of an ALU.

Figure 9: One stage of ALU with shift capability

Please note that in this figure we have given reference to two previous figures for arithmetic and logic circuits. This stage of ALU has two data inputs; the i^{th} bits of the registers to be manipulated. However, the $(i-1)^{\text{th}}$ or $(i+1)^{\text{th}}$ bit is also fed for the case of shift micro-operation of only one register. There are four selection lines, which determine what micro-operation (arithmetic, logic or shift) on the input. The F_i is the resultant bit after desired micro-operation. Let us see how the value of F_i changes on the basis of the four select inputs. This is shown in Figure 10:

Please note that in Figure 10 arithmetic micro-operations have both S_3 and S_2 bits as zero. Input C_i is important for only arithmetic micro-operations. For logic micro-operations S_3, S_2 values are 01. The values 10 and 11 cause shift micro-operations.

For this shift micro-operation S_1 and S_0 values and C_i values do not play any role.

S_3	S_2	S_1	S_0	C_i	F	Micro-operation	Name	
0	0	0	0	0	$F = x$	$R \leftarrow R_1$	Transfer	Arithmetic Micro-operation
0	0	0	0	1	$F = x+1$	$R \leftarrow R_1+1$	Increment	
0	0	0	1	0	$F = x+y$	$R \leftarrow R_1+R_2$	Addition	
0	0	0	1	1	$F = x+y+1$	$R \leftarrow R_1+R_2+1$	Addition with carry	
0	0	1	0	0	$F = x + \overline{y}$	$R \leftarrow R_1 + \overline{R_2}$	Subtract with borrow	
0	0	1	0	1	$F = x + (\overline{y} + 1)$	$R \leftarrow R_1 - R_2$	Subtract	
0	0	1	1	0	$F = x - 1$	$R \leftarrow R_1 - 1$	Decrement	
0	0	1	1	1	$F = x$	$R \leftarrow R_1$	Transfer	
0	1	0	0	-	$F = x.y$	$R \leftarrow R_1 \wedge R_2$	AND	Logic Micro-operation
0	1	0	1	-	$F = x+y$	$R \leftarrow R_1 \vee R_2$	OR	
0	1	1	0	-	$F = x \oplus y$	$R \leftarrow R_1 \oplus R_2$	Exclusive OR	
0	1	1	1	-	$F = \overline{x}$	$R \leftarrow \overline{R_1}$	Complement	
1	0	-	-	-	$F = \text{Shl}(x)$	$R \leftarrow \text{Shl}(R_1)$	Shift left	Shift Micro- operations
1	1	-	-	-	$F = \text{Shr}(y)$	$R \leftarrow \text{Shr}(R_1)$	Shift right	

Figure 10: Micro-operations performed by a Sample ALU

3.3 ARITHMETIC PROCESSORS

The questions in this regard are: “What is an arithmetic processor?” and, “What is the need for arithmetic processors?”

A typical CPU needs most of the control and data processing hardware for implementing non-arithmetic functions. As the hardware costs are directly related to chip area, a floating point circuit being complex in nature is costly to implement. They need not be included in the instruction set of a CPU. 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 CPU 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 CPU 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?

Two mechanisms are used for connecting the arithmetic processor 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. The data transfer instructions must be specific instructions in the CPU. This type of connection is called loosely coupled.

On the other hand if the arithmetic processor has a register and instruction set which can be considered an extension of the CPU registers and 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 are termed as co-processors.

The concept of co-processor existed in the 8086 machine till Intel 486 machines where co-processor was separate. However, Pentium at present does not have a separate co-processor. Similarly, peripheral processors are not found as arithmetic processors in general. However, many chips are used for specialized I/O architecture. These can be found in further readings.

Check Your Progress 2

1. Draw the logic circuit for a ALU unit.

2. What is an Arithmetic Processor?

.....

3.4 SUMMARY

In this unit, we have discussed in detail the hardware implementation of micro-operations. The unit starts with an implementation of bus, which is the backbone for any register transfer operation. This is followed by a discussion on arithmetic circuit and micro-operation thereon using full adder circuits. The logic micro-operation implementation has also been discussed. Thus, leading to a logical construction of a simple arithmetic – logic –shift unit. The unit revolves around the basic ALU with the help of the units that are constructed for the implementation of micro-operations.

In the later part of the unit, we discussed the arithmetic processors. Finally, we have presented a few chipsets that support the working of a processor for input/output functions from key board, printer etc.

3.5 SOLUTIONS/ ANSWERS

Check Your Progress 1

1. False
2. False
3. True

Check Your Progress 2

1. The diagram is the same as that of Figure 9.
2. Arithmetic processor performs arithmetic computation. These are support processors to a computer.

UNIT 4 THE CONTROL UNIT

Structure	Page No.
4.0 Introduction	65
4.1 Objectives	65
4.2 The Control Unit	65
4.3 The Hardwired Control	71
4.4 Wilkes Control	72
4.5 The Micro-Programmed Control	74
4.6 The Micro-Instructions	75
4.6.1 Types of Micro-Instructions	
4.6.2 Control Memory Organisation	
4.6.3 Micro-Instruction Formats	
4.7 The Execution of Micro-Program	78
4.8 Summary	81
4.9 Solutions/ Answers	81

4.0 INTRODUCTION

By now we have discussed instruction sets and register organisation followed by a discussion on micro-operations and a simple arithmetic logic unit circuit. We have also discussed the floating point ALU and arithmetic processors, which are commonly used for floating point computations.

In this unit we are going to discuss the functions of a control unit, its structure followed by the hardwired type of control unit. We will discuss the micro-programmed control unit, which are quite popular in modern computers because of flexibility in designing. We will start the discussion with several definitions about the unit followed by Wilkes control unit. Finally, we will discuss the concepts involved in micro-instruction execution.

4.1 OBJECTIVES

After going through this unit you will be able to:

- define what is a control unit and its function;
 - describe a simple control unit organization;
 - define a hardwired control unit;
 - define the micro-programmed control unit;
 - define the term micro-instruction; and
 - identify types and formats of micro-instruction.
-

4.2 THE CONTROL UNIT

The two basic components of a CPU are the control unit and the arithmetic and logic unit. The control unit of the CPU selects and interprets program instructions and then sees that they are executed. The basic responsibilities of the control unit are **to control**:

- a) Data exchange of CPU with the memory or I/O modules.
- b) Internal operations in the CPU such as:
 - moving data between registers (register transfer operations)

- making ALU to perform a particular operation on the data
- regulating other internal operations.

But how does a control unit control the above operations? What are the functional requirements of the control unit? What is its structure? Let us explore answers of these questions in the next sections.

Functional Requirements of a Control Unit

Let us first try to define the functions which a control unit must perform in order to get things to happen. But in order to define the functions of a control unit, one must know what resources and means it has at its disposal. A control unit must know about the:

- (a) Basic components of the CPU
- (b) Micro-operation this CPU performs.

The CPU 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 CPU.
- **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 CPU 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 CPU.

The micro-operations performed by the CPU 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 the various components of CPU to perform a specific sequence of micro-operations to achieve the execution of an instruction.

What are the functions, which a control unit performs to make an instruction execution feasible? The instruction execution is achieved by executing micro-operations in a specific sequence. For different instructions this sequence may be different. Thus the control unit must perform two basic functions:

- Cause the execution of a micro-operation.
- Enable the CPU to execute a proper sequence of micro-operations, which is determined by the instruction to be executed.

But how are these two tasks achieved? The control unit generates control signals, which in turn are responsible for achieving the above two tasks. 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.

A control unit has a set of input values on the basis of which it produces an output control signal, which in turn performs micro-operations. These output signals control the execution of a program. A general model of control unit is shown in Figure 1.

Figure 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 square. 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 the various cycles to be performed and hence determines the related micro-operations, which are needed to be performed.
- **Flags:** Flags are used by the control unit for determining the status of the CPU & the outcomes of a previous ALU operation. For example, a zero flag if set conveys to control unit that for instruction ISZ (skip the next instruction if zero flag is set) the next instruction is to be skipped. For such a case control unit cause increment of PC by program instruction length, thus skipping 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 CPU. Some of these signals are interrupt signals and acknowledgement signals.

On the basis of the input signals the control unit activates certain output control signals, which in turn are responsible for the execution of an instruction. These output control signals are:

- **Control signals, which are required within the CPU:** These control signals cause two types of micro-operations, viz., for data transfer from one register to another; and for performing an arithmetic, logic and shift operation using ALU.
- **Control signals to control bus:** These control signals transfer data from or to CPU 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.

Now, let us discuss the requirements from such a unit. A prime requirement for control unit is that it must know how all the instructions will 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, op-codes, clock and some control signals to itself.

A control unit contains a clock portion that provides clock-pulses. This clock signal is used for measuring the timing of the micro-operations. In general, the timing signals from control unit are kept sufficiently long to accommodate the proportional delays of signals within the CPU 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 achieve the particular operation? *The control signals are applied directly as the binary inputs to the logic gates of the logic circuits.* 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.

A program execution consists of a sequence of instruction cycles. Each instruction cycle is made up of a number of sub cycles. One such simple subdivision includes fetch, indirect, execute, and interrupt cycles, with only fetch and execute cycles always occurring. Each sub cycle involves one or more micro-operations.

Let us revisit the micro-operations described in Unit 2 to discuss how the events of any instruction cycle can be described as a sequence of such micro-operations.

The Fetch Cycle

The beginning of each instruction cycle is the fetch cycle, and causes an instruction to be fetched from memory.

The fetch cycle consists of four micro-operations that are executed in three timing steps. The fetch cycle can be written as:

$$\begin{array}{ll} T_1: & \text{MAR} \leftarrow \text{PC} \\ T_2: & \text{MBR} \leftarrow [\text{MAR}] \\ & \text{PC} \leftarrow \text{PC} + I \\ T_3: & \text{IR} \leftarrow \text{MBR} \end{array}$$

where I is the instruction length. We assume that a clock is available for timing purposes and that it emits regularly spaced clock pulses. Each clock pulse defines a time unit. Thus, all the units are of equal duration. Each micro-operation can be performed within the time of a single time unit. The notation (T_1 , T_2 , T_3) represents successive time units. What is done in these time units?

- In the first time unit the content of PC is moved to MAR.
- In the second time unit the contents of memory location specified by MAR is moved to MBR and the contents of the PC is incremented by I.
- In the third time unit the content of MBR is moved to IR.

The Indirect Cycle

Once an instruction is fetched, the next step is to fetch the operands. Considering the same example as of Unit 2, the instruction may have direct and indirect addressing modes. An indirect address is handled using indirect cycle. The following micro-operations are required in the indirect cycle:

$T_1: \text{MAR} \leftarrow \text{IR (address)}$
 $T_2: \text{MBR} \leftarrow [\text{MAR}]$
 $T_3: \text{IR (address)} \leftarrow \text{MBR (address)}$

The MAR is loaded with the address field of IR register. Then the memory is read to fetch the address of operand, which is transferred to the address field of IR through MBR as data is received in MBR during the read operation.

Thus, the IR now is in the same state as of direct address, viz., as if indirect addressing had not been used. IR is now ready for the execute cycle.

The Execute Cycle

The fetch and indirect cycles involve a small, fixed sequence of micro-operations. Each of these cycles has fixed sequence of micro-operations that are common to all instructions.

This is not true of the execute cycle. For a machine with N different opcodes, there are N different sequences of micro-operations that can occur. Let us consider some hypothetical instructions:

An add instruction that adds the contents of memory location X to Register R1 with R1 storing the result:

ADD R1, X

The sequence of micro-operations may be:

$T_1: \text{MAR} \leftarrow \text{IR (address)}$
 $T_2: \text{MBR} \leftarrow [\text{MAR}]$
 $T_3: \text{R1} \leftarrow \text{R1} + \text{MBR}$

At the beginning of the execute cycle IR contains the ADD instruction and its direct operand address (memory location X). At time T_1 , the address portion of the IR is transferred to the MAR. At T_2 the referenced memory location is read into MBR. Finally, at T_3 the contents of R1 and MBR are added by the ALU.

Let us discuss one more instruction:

ISZ X, it increments the content of memory location X by 1. If the result is 0, the next instruction in the sequence is skipped. A possible sequence of micro-operations for this instruction may be:

$T_1: \text{MAR} \leftarrow \text{IR (address)}$
 $T_2: \text{MBR} \leftarrow [\text{MAR}]$
 $T_3: \text{MBR} \leftarrow \text{MBR} + 1$
 $T_4: [\text{MAR}] \leftarrow \text{MBR}$
 If $(\text{MBR} = 0)$ then $(\text{PC} \leftarrow \text{PC} + 1)$

Please note that for this machine we have assumed that MBR can be incremented by ALU directly.

The PC is incremented if MBR contains 0. This test and action can be implemented as one micro-operation. Note also that this micro-operation can be performed during the same time unit during which the updated value in MBR is stored back to memory. Such instructions are useful in implementing looping.

The Interrupt Cycle

On completion of the execute cycle the current instruction execution gets completed. At this point a test is made to determine whether any enabled interrupts have occurred. If so, the interrupt cycle is performed. This cycle does not execute an interrupt but causes start of execution of Interrupt Service Program (ISR). Please note that ISR is executed as just another program instruction cycle. The nature of this cycle varies greatly from one machine to another. A typical sequence of micro-operations of the interrupt cycle are:

$T_1: \text{MBR} \leftarrow \text{PC}$
 $T_2: \text{MAR} \leftarrow \text{Save-Address}$
 $\text{PC} \leftarrow \text{ISR-Address}$
 $T_3: [\text{MAR}] \leftarrow \text{MBR}$

At time T_1 , the contents of the PC are transferred to the MBR, so that they can be saved for return from the interrupt. At time T_2 the MAR is loaded with the address at which the contents of the PC are to be saved, and PC is loaded with the address of the start of the interrupt-servicing routine. At time T_3 MBR, which contains the old value of the PC, is stored in the memory. The processor is now ready to begin the next instruction cycle.

The Instruction Cycle

The instruction cycle for this given machine consists of four cycles. Assume a 2-bit instruction cycle code (ICC). The ICC can represent the state of the processor in terms of cycle. For example, we can use:

00 : Fetch
01 : Indirect
10 : Execute
11 : Interrupt

At the end of each of the four cycles, the ICC is set appropriately. Please note that an indirect cycle is always followed by the execute cycle and the interrupt cycle is always followed by the fetch cycle. For both the execute and fetch cycles, the next cycle depends on the state of the system. Let us show an instruction execution using timing diagram and instruction cycles:

Figure 2: Timing Diagram for ISZ instruction

Please note that the address line determine the location of memory. Read/ write signal controls whether the data is being input or output. For example, at time T_2 in M_2 the read control signal becomes active, $A_9 - A_0$ input contains MAR that value is kept enabled on address bits and the data lines are enabled to accept data from RAM, thus enabling a typical RAM data output on the data bus.

For reading no data input is applied by CPU but it is put on data bus by memory after the read control signal to memory is activated. Write operation is activated along with data bus carrying the output value.

This diagram is used for illustration of timing and control. However, more information on these topics can be obtained from further readings.

4.3 THE HARDWIRED CONTROL

With 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. That is, the control unit is implemented by gates, flip-flops, decoder and other digital circuits. Hardwired control units can be optimised for fast operations.

The block diagram of control unit is shown in Figure 3. The major inputs 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 for different instructions. The control unit logic has a unique logic input for each opcode. This simplifies the control logic. This control line selection can be performed by a decoder.

A decoder will have n binary inputs and 2^n binary outputs. Each of these 2^n different input patterns will activate a single unique output line.

The clock portion of the control unit issues a repetitive sequence of pulses for the SS duration 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.

Figure 3: Block Diagram of Control Unit Operation

Check Your Progress 1

1. What are the inputs to control unit?

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

2. How does a control unit control the instruction cycle?

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

3. What is a hardwired control unit?

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

4.4 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 transformations and transfer of information from one register in the processor to another take place. These were also called the micro operations. **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:

- a) Control field which indicates the control lines which are to be activated and
- b) Address field, which provides the address of the next microinstruction to be executed.

The figure 4 below is an example of Wilkes control unit design.

Figure 4: Wilkes Control Unit

The control memory in Wilkes control is organized, as a PLA's like matrix made of diodes. This is partial matrix and consists of two components, the control signals and the address of the next micro-instruction. The register I contains the address of the next micro-instruction that is one step of instruction execution, for example T_1 in M_1 or T_2 in M_2 etc. as in Figure 2. On decoding the control signals are generated that cause execution of micro-operation(s) of that step. In addition, the control unit indicates the address of the next micro-operation which gets loaded through register II to register I. Register I can also be loaded by register II and "enable IR input" control signal. This will pass the address of first micro-instruction of execute cycle. During a machine cycle one row of the matrix is activated. The first part of the row generates the control signals that control the operations of the processor. The second part generates the address of the row to be selected in the next machine cycle.

At the beginning of the cycle, the address of the row to be selected is contained in register I. This address is the input to the decoder, which is activated by a clock pulse. This activates the row of the control matrix. The two-register arrangement is needed, as the decoder is a combinational circuit; with only one register, the output would become the input during a cycle. This may be an unstable condition due to repetitive loop.

4.5 THE MICRO-PROGRAMMED CONTROL

An alternative to a hardwired control unit is a micro-programmed control unit, in which the logic of the control unit is specified by a micro-program. 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 demonstrated in Figure 5 below:

Figure 5: Operation of Micro-Programmed Control Unit

The micro-instructions are stored in the control memory. The address register for the control memory contains the address of the next instruction that is to be read. The control memory Buffer Register receives the micro-instruction that has been read. 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 section loads the control memory address register. It also issues a read command to control memory. The following functions are performed by the micro-programmed control unit:

1. The sequence logic unit specifies the address of the control memory word that is to be read, in the Address Register of the Control Memory. It also issues the READ signal.
2. The desired control memory word is read into control memory Buffer Register.

3. The content of the control memory buffer register is decoded to create control signals and next-address information for the sequencing logic unit.
4. The sequencing logic unit finds the address of the next control word on the basis of the next-address information from the decoder and the ALU flags.

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 dependent on the opcode of the instruction Register (IR). Thus, IR input to Address Register for Control Memory is desirable. Thus, there exist a decoder from IR to Address Register for control memory. (Refer Figure 5). This decoder translates the opcode of the IR into a control memory address.

Check Your Progress 2

1. What is firmware? How is it different from software?

2. **State True or False**

T	F
---	---

 - (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 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 4, if the entry address for a machine instruction selects the last but one (branching control line) and the conditional bit value for branch is true?

4.6 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.

4.6.1 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.

4.6.2 Control Memory Organization

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 6 shows such an organisation.

Figure 6: 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 can be;

Test "zero flag" If SET branch to label ZERO

Unconditional branch to label NON-ZERO

ZERO: (Microcode which causes replacement of program counter with the address provided in the instruction)

Branch to interrupt or fetch cycle.

NON -ZERO: (Microcode which may set flags if desired indicating the branch has not taken place).

Branch to interrupt or fetch cycle. (For Next- Instruction Cycle)

4.6.3 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 7(a).

(a) Horizontal Micro-instruction

(b) Vertical Micro-instructions

(c) A Realistic Micro-instructions

Figure 7: 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 7(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 7(c).

4.7 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 CPU (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.

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 we follow the Wilkes scheme we require N bits, one for each control signal in the control unit.

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. We can provide only one pattern of control signals at a time to ALU, making some of the combinations redundant.
4. We can provide only one pattern of control signals at a time to the external control bus also.

Therefore, we do not need 2^N combinations. Suppose, we only need 2^K (which is less than 2^N) combinations, then we need only K encoded bits instead of N control signals. The K bit 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 small.
- It provided an aggregated view that is a higher view of the CPU 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 a longer time than that of an unencoded micro-instruction.
- The highly encoded micro-instructions are aimed at optimizing programming effort.

In most of the cases, the design is kept between the two extremes. The LSI 11 (highly encoded) and IBM 3033 (unencoded) control units are close examples of these two approaches.

Execution/decoding of slightly encoded micro-instructions

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. The basic technique for encoding is shown in Figure 8. The micro-instruction is organised as a set of fields. Each field contains a code, which, upon decoding, activates one or more control signals. The execution of a micro-instruction means that every field is decoded and generates control signals. Thus, with N fields, N simultaneous actions can be specified. Each action results in the activation of one or more control signals. Generally each control signal is activated by no more than one field. The design of an encoded micro-instruction format can be stated in simple terms:

- Organize the format into independent fields. That is, each field depicts a set of actions such that actions from different fields can occur simultaneously.
- Define each field such that the alternative actions that can be specified by the field are mutually exclusive. That is, only one of the actions specified for a given field could occur at a time.

Another aspect of encoding is whether it is direct or indirect (Figure 8). With indirect encoding, one field is used to determine the interpretation of another field.

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 (refer Figure 6):

- Next micro-instruction in sequence
- Calculated on the basis of opcode
- Branch address (conditional or unconditional).

A detailed discussion on these topics is beyond this unit. You must refer to further readings for more detailed information on Micro-programmed Control Unit Design.

Figure (a):

Figure (b):
Figure 8: Micro-instruction Encoding

Check Your Progress 3

T	F
---	---

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 5) is to cause reading of micro-instruction addressed by a micro-program counter into the micro-instruction buffer. ☐
- 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.

.....

 ...

4.8 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.

4.9 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. Firmwares are more difficult to write than software.
2. (a) False (b) False (C) False (d) False
3. In sequence from left to right as per figure.
 110.....00 (control signals indicate more values)
 110.....00 (address of next, micro-instruction is found after assuming that bottom line after condition code represent true in the Figure 4)

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 	<ul style="list-style-type: none"> Relatively less bits Easy to program Need decoding logic

**The Central
Processing Unit**

<ul style="list-style-type: none">• Optimizes machine performances• Detailed hardware view	Optimizes programming effort Aggregated view
---	---

UNIT 5 REDUCED INSTRUCTION SET COMPUTER ARCHITECTURE

Structure	Page No.
5.0 Introduction	83
5.1 Objectives	83
5.2 Introduction to RISC	83
5.2.1 Importance of RISC Processors	
5.2.2 Reasons for Increased Complexity	
5.2.3 High Level Language Program Characteristics	
5.3 RISC Architecture	88
5.4 The Use of Large Register File	90
5.5 Comments on RISC	93
5.6 RISC Pipelining	94
5.7 Summary	98
5.8 Solutions/ Answers	98

5.0 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. We will also discuss the arithmetic and logic units here.

5.1 OBJECTIVES

After going through this unit you should be able to:

- define why complexity of instruction increased?;
 - describe the reasons for developing RISC;
 - define the basic design principles of RISC;
 - describe the importance of having large register file;
 - discuss some of the common comments about RISC;
 - describe RISC pipelining; and
 - define the optimisation in RISC pipelining.
-

5.2 INTRODUCTION TO RISC

The aim of computer architects is to design computers which are cheaper and more powerful than their predecessors. A cheaper computer has:

- Low hardware manufacturing cost.
- Low Cost for programming scalable/ portable architecture that require low costs for debugging the initial hardware and subsequent programs.

If we review the history of computer families, we find that the most common architectural change is the trend towards even more complex machines.

5.2.1 Importance of RISC Processors

Reduced Instruction Set Computers recognize a relatively limited number of instructions. One advantage of a reduced instruction set is that RISC can execute the instructions very fast because these are so simple. Another advantage is that RISC chips require fewer gates and hence transistors, which makes them cheaper to design and produce.

An instruction of RISC machine can be executed in one cycle, as there exists an instruction pipeline. This may enhance the speed of instruction execution. In addition, the control unit of the RISC processor is simpler and smaller, so much so that it acquires only 6% space for a processor in comparison to Complex Instruction Set Computers (CISC) in which the control unit occupies about 50% of space. This saved space leaves a lot of room for developing a number of registers.

This further enhances the processing capabilities of the RISC processor. It also necessitates that the memory to register “LOAD” and “STORE” are independent instructions.

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 very popular. Some of the RISC processors are:

SPARC Processors

Sun 4/100 series, Sun 4/310 SPARCserver 310, Sun 4/330 SPARCserver 330, Sun 4/350 SPARCserver 350, Sun 4/360 SPARCserver 360, Sun 4/370 SPARCserver 370, Sun 4/20, SPARCstation SLC, Sun 4/40 SPARCstation IPC, Sun 4/75, SPARCstation 2.

PowerPC Processors

MPC603, MPC740, MPC750, MPC755, MPC7400/7410, MPC745x, MPC7450, MPC8240, MPC8245.

Titanium – IA64 Processor

5.2.2 Reasons for Increased Complexity

Let us see what the reasons for increased complexity are, and what exactly we mean by this.

Speed of Memory Versus Speed of CPU

In the past, there existed a large gap between the speed of a processor and memory. Thus, a subroutine execution for an instruction, 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 sequence. 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
- Store the above results in A
Checks overflow underflow if any.

If all these steps are coded as one machine instruction, then this simple instruction will require many instruction execution cycles. If this instruction is made as part of the machine instruction set as: ADDF A,B (Add floating point numbers A & 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.

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, it was opined that the instruction set should be more complex, 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. However, today memory is very inexpensive, this potential advantage today is not so compelling. More important, small programs should improve performance. How? Fewer instructions mean fewer instruction bytes to be fetched.

However, the problem with this reasoning is that it is not certain that a CISC program will be smaller than the corresponding RISC program. In many cases CISC program expressed in symbolic machine language may be smaller but the number of bits of machine code program may not be noticeably smaller. This may result from the reason that in RISC we use register addressing and less instruction, which require fewer bits in general. In addition, the compilers on CISCs often favour simpler instructions, so that the conciseness of complex instruction seldom comes into play.

Let us explain this with the help of the following example:

Assumptions:

- The Complex Instruction is: Add C, A, B having 16 bit addresses and 8 bit data operands
- All the operands are direct memory reference operands
- The machine has 16 registers. So the size of a register address is $= 2^4 = 16 = 4$ bits.
- The machine uses an 8-bit opcode.

				8		4		16	
				Load		rA		A	
				Load		rB		B	
				Add		rC		rA rB	
				Store		rC		C	

8		16		16		16	
Add		C		A		B	

Memory-to-Memory

Instruction size (I) = 56 bits

Data Size (D) = 24 bits

Total Memory Load (M) = 80 bits

Register-to-Register

I = 104 bits

D = 24bits

M = 128 bits

(a) Add A & B to store result in C

				8		4		16	
				Load		rA		A	
				Load		rB		B	
				Add		rC		rB rA	
				Load		rD		D	
				Add		rA		rC Rd	
				Sub		rD		rD rB	
				Store		rD		D	

8		16		16		16	
Add		C		A		B	
Add		A		C		D	
Sub		D		D		B	

Memory-to-Memory

Instruction size (I) = 168 bits

Data Size (D) = 72 bits

Total Memory Load (M) = 240 bits

Register-to-Register

I = 172 bits

D = 32bits

M = 204 bits

(b) Execution of the Instruction Sequence: C = A + B, A = C + D, D = D - B

Figure 1: Program size for different Instruction Set Approaches

So, the expectation that a CISC will produce smaller programs may not be realised.

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.

5.2.3 High Level Language Program Characteristics

Thus, it is clear that 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 are:

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. Most of these have fewer than 6 local variables
Array/ structure 20-30%	Procedure call 10-15%	
	IF 35-45%	
	GOTO FEW	
	Others 1-5%	

Figure 2: 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 lower 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. ☐

5.3 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, therefore, use LOADs and STOREs to access memory so that operands are not implicitly discarded after being fetched. (Refer to Figure 1(b)).

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.

5.4 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.

On the face of it the use of a large set of registers should 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 multiple small sets of registers and assigned to a different function. A function call automatically changes each of these sets. 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.

The diagram shows the use of registers: when there is call to function A (f_A) which calls function B (f_B) and function B calls function C (f_C).

Registers Nos.	Used for			
0 – 9	Global variables required by f_A , f_B , and f_C	Function A	Function B	Function C
10 – 83	Unused			
84 – 89 (6 Registers)	Used by parameters of f_C that may be passed to next call			Temporary variables of function C
90 – 99 (10 Registers)	Used for local variable of f_C			Local variables of function C
100 – 105 (6 Registers)	Used by parameters that were passed from $f_B \rightarrow f_C$		Temporary variables of function B	Parameters of function C
106 – 115 (10 Registers)	Local variables of f_B		Local variables of function B	
116 – 121 (6 Registers)	Parameters that were passed from f_A to f_B	Temporary variables of function A	Parameters of function B	
122 – 131 (10 Registers)	Local variable of f_A	Local variables of function A		
132 – 138 (6 Registers)	Parameter passed to f_A	Parameters of function A		

Figure 3: Use of three Overlapped Register Windows

Please note the functioning of the registers: at any point of time the global registers and only one window of registers is visible and is addressable as if it were the only set of registers. 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: They are physically the same as the parameter registers at the next level. This overlap permits parameter passing without the actual movement of data.

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 this pointer to determine the actual physical registers. On the other hand the saved window pointer identifies the window most recently saved in memory. 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.

Figure 4: Circular-.Buffer Organization of Overlapped Windows

If function E now makes a call to function F, the call cannot be made with the current status of the buffer, unless we 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.

All but one point above basically show comparative equality. The basic difference is due to addressing overhead of the two approaches.

The following figure shows the difference. Small register (R) address is added with current window Pointer W#. This generates the address in register file, which is decoded by decoder for register access. On the other hand Cache reference will be generated from a long memory address, which first goes through comparison logic to ascertain the presence of data, and if the data is present it goes through the select circuit. 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.

(a) Windows based Register file

(b) Cache Reference

Figure 5: Referencing a local Simple Variables

Check Your Progress 2

1. State True or False in the context of RISC architecture:

T	F
---	---

 - a. RISC has a large register file so that more variables can be stored in register or longer periods of time. ☐
 - b. Only global variables are stored in registers. ☐
 - c. Variables are passed as parameters in registers using temporary registers in a window. ☐
 - d. Cache is superior to a large register file as it stores most recently used local scalars. ☐
2. 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?

.....

.....

.....

5.5 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. In fact the support can be measured as the inverse of the costs of using typical HLL constructs on a particular machine. 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 we will 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 is possible due to reduction in control unit size from 50 to 6 per cent. In addition, the control is simple in RISC than CISC, thus further helping the simple instructions to execute faster.

5.6 RISC PIPELINING

Instruction pipelining is often used to enhance performance. Let us consider this in the context of RISC architecture. In RISC machines most of the operations are register-to-register. Therefore, the instructions can be executed in two phases:

- F: Instruction Fetch to get the instruction.
- E: Instruction Execute on register operands and store the results in register.

In general, the memory access in RISC is performed through LOAD and STORE operations. For such instructions the following steps may be needed:

- F: Instruction Fetch to get the instruction
- E: Effective address calculation for the desired memory operand
- D: Memory to register or register to memory data transfer through bus.

Let us explain pipelining in RISC with an example program execution sample. Take the following program (R indicates register).

LOAD R_A (Load from memory location A)
 LOAD R_B (Load from memory location B)
 ADD R_C, R_A, R_B ($R_C = R_A + R_B$)
 SUB R_D, R_A, R_B ($R_D = R_A - R_B$)
 MUL R_E, R_C, R_D ($R_E = R_C \times R_D$)
 STOR R_E (Store in memory location C)
 Return to main.

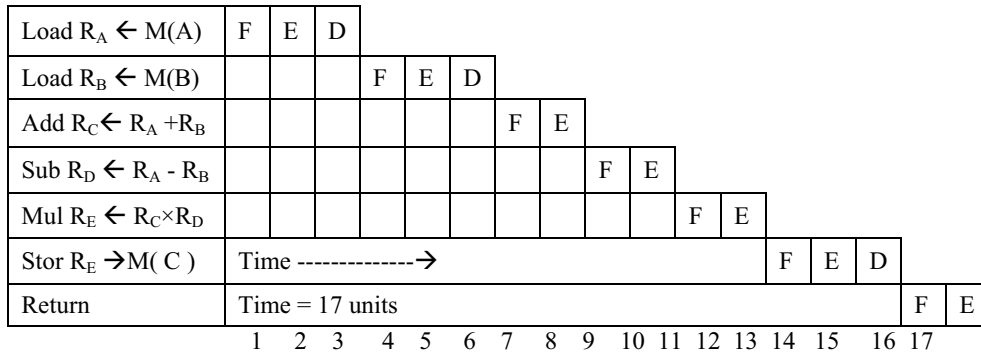


Figure 6: Sequential Execution of Instructions

Figure 7 shows a simple pipelining scheme, in which F and E phases of two different instructions are performed simultaneously. This scheme speeds up the execution rate of the sequential scheme.

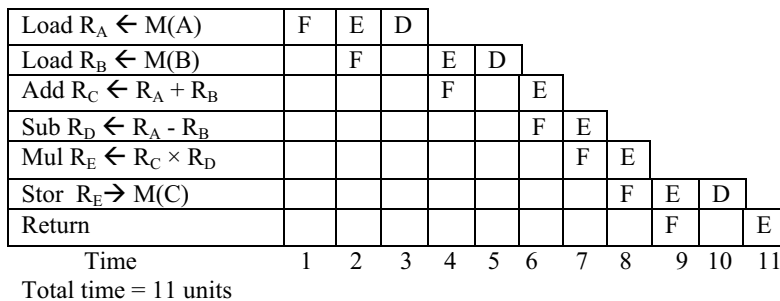


Figure 7: Two Way Pipelined Timing

Please note that the pipeline above is not running at its full capacity. This is because of the following problems:

- We are assuming a single port memory thus only one memory access is allowed at a time. Thus, Fetch and Data transfer operations cannot occur at the same time. Thus, you may notice blank in the time slot 3, 5 etc.
- The last instruction is an unconditional jump. Please note that after this instruction the next instruction of the calling program will be executed. Although not visible in this example a branch instruction interrupts the sequential flow of instruction execution. Thus, causing inefficiencies in the pipelined execution.

This pipeline can simply be improved by allowing two memory accesses at a time.

Thus, the modified pipeline would be:

The pipeline may suffer because of data dependencies and branch instructions penalties. A good pipeline has equal phases.

Load $R_A \leftarrow M(A)$	F	E	D						
Load $R_B \leftarrow M(B)$		F	E	D					
Add $R_C \leftarrow R_A + R_B$			F	E					
Sub $R_D \leftarrow R_A - R_B$				F	E				
Mul $R_E = R_C \times R_D$					F	E			
Stor $R_E \rightarrow M(C)$	Time ----->					F	E	D	
Return	Time = 8 units						F	E	

Figure 8: Three-way Pipelining Timing

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. RISC optimizes this problem by using a technique called delayed branching.

One of the common techniques used to avoid branch penalty is to pre-fetch the branch destination also. RISC follows a branch optimization technique called delayed jump as shown in the example given below:

Load $R_A \leftarrow M(A)$	F	E	D						
Load $R_B \leftarrow M(B)$		F	E	D					
Add $R_C \leftarrow R_A + R_B$			F	E					
Sub $R_D \leftarrow R_A - R_B$				F	E				
If $R_D < 0$ Return					F	E			
Stor $R_C \rightarrow M(C)$						F	E	D	
Return							F	E	

(a) The instruction “If $R_D < 0$ Return” may cause pipeline to empty

Load $R_A \leftarrow M(A)$	F	E	D						
Load $R_B \leftarrow M(B)$		F	E	D					
Add $R_C \leftarrow R_A + R_B$			F	E					
Sub $R_D \leftarrow R_A - R_B$				F	E				
If $R_D < 0$ Return					F	E			
NO Operation						F	E		
Stor $R_C \rightarrow M(C)$ Or Return as the case may be							F	E	D
Return								F	E

(b) The No operation instruction causes decision of the If instruction known, thus correct instruction can be fetched.

Load $R_A \leftarrow M(A)$	F	E	D						
Load $R_B \leftarrow M(B)$		F	E	D					
Sub $R_D \leftarrow R_A - R_B$			F	E					
If $R_D < 0$ Return				F	E				
Add $R_C \leftarrow R_A + R_B$					F	E			
Stor $R_C \rightarrow M(C)$						F	E	D	
Return							F	E	

(c) The branch is calculated before, thus the pipeline need not be emptied. This is delayed branch.

Figure 9: Delayed Branch

Finally, let us summarize the basic differences between CISC and RISC architecture. The following table lists these differences:

CISC	RISC
1. Large number of instructions – from 120 to 350.	1. Relatively fewer instructions - less than 100.
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 usually 32 bits, 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/STORE instructions.
5. Number of Cycles Per Instruction (CPI) varies from 1-20 depending upon the instruction.	5. Number of CPI is one as it uses pipelining. Pipeline in RISC is optimised because of simple instructions and instruction formats.
6. GPRs varies from 8-32. But no support is available for the parameter passing and function calls.	6. Large number of GPRs are available that are primarily used as Global registers and as a register based procedural call and parameter passing stack, thus, optimised for structured programming.
7. Microprogrammed Control Unit.	7. Hardwired Control Unit.

Check Your Progress 3

1. What are the problems, which prevent RISC pipelining to achieve maximum speed?

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

2. 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?

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

5.7 SUMMARY

RISC represents new styles of computers that take less time to build yet provide a higher performance. While traditional machines support HLLs with instructions 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. It also appears that we can build such a single chip computer much sooner and with much less effort than traditional architectures.

Thus, we see that because of all the features discussed above, the RISC architecture should prove to be far superior to even the most complex CISC architecture.

In this unit we have also covered the details of the pipelined features of the RISC architecture, which further strengthen our arguments for the support of this architecture.

5.8 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

- The following are the problems:
 - It has a single port memory reducing the access to one device at a time
 - Branch instruction
 - The data dependencies between the instructions
- It can be improved by:
 - allowing two memory accesses per phase
 - introducing three phases of approximately equal duration in pipelining
 - causing optimized delayed jumps/loads etc.
- 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.

UNIT 1 MICROPROCESSOR ARCHITECTURE

Structure	Page No.
1.0 Introduction	5
1.1 Objectives	5
1.2 Microcomputer Architecture	5
1.3 Structure of 8086 CPU	7
1.3.1 The Bus Interface Unit	
1.3.2 Execution Unit (EU)	
1.4 Register Set of 8086	11
1.5 Instruction Set of 8086	13
1.5.1 Data Transfer Instructions	
1.5.2 Arithmetic Instructions	
1.5.3 Bit Manipulation Instructions	
1.5.4 Program Execution Transfer Instructions	
1.5.5 String Instructions	
1.5.6 Processor Control Instructions	
1.6 Addressing Modes	29
1.6.1 Register Addressing Mode	
1.6.2 Immediate Addressing Mode	
1.6.3 Direct Addressing Mode	
1.6.4 Indirect Addressing Mode	
1.7 Summary	33
1.8 Solutions/Answers	33

1.0 INTRODUCTION

In the previous blocks of this course, we have discussed concepts relating to CPU organization, register set, instruction set, addressing modes with a few examples. Let us look at one microprocessor architecture in regard of all the above concepts. We have selected one of the simplest processors 8086, for this purpose. Although the processor technology is old, all the concepts are valid for higher end Intel processor. Therefore, in this unit, we will discuss the 8086 microprocessor in some detail.

We have started the discussion of the basic microcomputer architecture. This discussion is followed by the details on the components of CPU of the 8086 microprocessor. Then we have discussed the register organization for this processor. We have also discussed the instruction set and addressing modes for this processor. Thus, this unit presents exhaustive details of the 8086 microprocessor. These details will then be used in Assembly Programming.

1.1 OBJECTIVES

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

- describe the features of the 8086 microprocessor;
 - list various components of the 8086 microprocessor; and
 - identify the instruction set and the addressing modes of the 8086 microprocessor.
-

1.2 MICROCOMPUTER ARCHITECTURE

The word micro is used in microscopes, microphones, microwaves, microprocessors, microcomputers, microprogramming, microcodes etc. It means small. A

microprocessor is an example of VLSI bringing the whole processor to a single small chip. With the popularity of distributed processing, the emphasis has shifted from the single mainframe system to independently working workstations or functioning units with their own CPU, RAM, ROM and a magnetic or optical disk memory. Thus, the advent of the microprocessor has transformed the mainframe environment to a distributed platform.

Let us recapitulate the basic components of a microprocessor:

Figure 1: Components of a Microcomputer

Please note the following in the above figure:

- ROM stores the boot program.
- The path from CPU to devices is through Buses. But what would be the size of these Buses?

Bus Sizes

1. The Address bus: 8085 microprocessor has 16 bit lines. Thus, it can access up to $2^{16} = 64K$ Bytes. The address bus of 8086 microprocessor has a 20 bits address bus. Thus it can access upto $2^{20} = 1M$ Byte size of RAM directly.
2. Data bus is the number of bits that can be transferred simultaneously. It is 16 bits in 8086.

Microprocessors

The microprocessor is a complete CPU on a single chip. The main advantages of the microprocessor are:

- compact but powerful;
- can be microprogrammed for user's needs;
- easily programmable and maintainable due to small size; and
- useful in distributed applications.

A microprocessor must demonstrate:

- More throughput
- More addressing capability
- Powerful addressing modes
- Powerful instruction set
- Faster operation through pipelining
- Virtual memory management.

However, RISC machine do not agree with above principles.

Some of the most commercially available microprocessors are: Pentium, Xeon, G4 etc.

The assembly language for more advanced chips subsumes the simplest 8086/ 8088 assembly language. Therefore, we will confine our discussions to Intel 8086/8088 assembly language. You must refer to the further readings for more details on assembly language of Pentium, G4 and other processors.

All microprocessors execute a continuous loop of fetch and execute cycles.

```
while (1)
{
    fetch (instruction); ,
    execute (using date);
}
```

1.3 STRUCTURE OF 8086 CPU

The 8086 microprocessor consists of two independent units:

1. The Bus Interface unit, and
2. The Execution unit.

Please refer to Figure 2.

Figure 2: The CPU of INTEL 8086 Microprocessor

The word independent implies that these two units can function parallel to each other. In other words they may be considered as two stages of the instruction pipeline.

1.3.1 The Bus Interface Unit

The BIU (Bus Interface Unit) primarily interacts with the system bus. It performs almost all the activities relating to fetch cycle such as:

- Calculating the physical address of the next instruction
- Fetching the instruction
- Reading or writing data memory or I/O port from memory or Input/ Output.

The instruction/ data is then passed to the execution unit. This BIU consists of:

(a) The Instruction Queue

The instruction queue is used to store the instruction “bytes” fetched. Please note two points here: that it is (1) A Byte (2) Queue. This is used to store information in byte form, with the underlying queue data structure. The advantage of this queue would only be if the next expected instructions are fetched in advance, thus, allowing a pipeline of fetch and execute cycles.

(b) The Segment Registers

These are very important registers of the CPU. Why? We will answer this later. In 8086 microprocessor, the memory is a byte organized, that is a memory address is byte address. However, the number of bits fetched is 16 at a time. The segment registers are used to calculate the address of memory location along with other registers. A segment register is 16 bits long.

The BIU contains four sixteen-bit registers, viz., the CS: Code Segment, the DS: Data Segment, the SS: Stack Segment, and the ES: Extra Segment. But what is the need of the segments: Segments logically divide a program into logical entities of Code, Data and Stack each having a specific size of 64 K. The segment register holds the upper 16 bits of the starting address of a logical group of memory, called the segment. But what are the advantages of using segments? The main advantages of using segments are:

- Logical division of program, thus enhancing the overall possible memory use and minimise wastage.
- The addresses that need to be used in programs are relocatable as they are the offsets. Thus, the segmentation supports relocatability.
- Although the size of address, is 20 bits, yet only the maximum segment size, that is 16 bits, needs to be kept in instruction, thus, reducing instruction length.

The 8086 microprocessor uses overlapping segments configuration. The typical memory organization for the 8086 microprocessor may be as per the following figure.

Figure 3: Logical Organisation of Memory in INTEL 8086 Microprocessor

Although the size of each segment can be 64K, as they are overlapping segments we can create variable size of segments, with maximum as 64K. Each segment has a specific function. 8086 supports the following segments:

As per model of assembly program, it can have more than one of any type of segments. However, at a time only four segments one of each type, can be active.

The 8086 supports 20 address lines, thus supports 20 bit addresses. However, all the registers including segment registers are of only 16 bits. So how may this mapping of 20 bits to 16 bits be performed?

Let us take a simple mapping procedure:

The top four hex digits of initial physical address constitute segment address.

You can add offset of 16 bits (4 Hex digits) from 0000h to FFFFh to it . Thus, a typical segment which starts at a physical address 10000h will range from 10000h to 1FFFFh. The segment register for this segment will contain 1000H and offset will

range from 0000h to FFFFh. But, how will the segment address and offset be added to calculate physical address? Let us explain using the following examples:

Example 1 (In the Figure above)

The value of the stack segment register (SS) = 6000h

The value of the stack pointer (SP) which is Offset = 0010h

Thus, Physical address of the top of the stack is:

SS	6	0	0	0	0	——— Implied zero
SP	+	0	0	1	0	
Physical Address		6	0	0	1	0

This calculation can be expressed as:

Physical address = SS (hex) × 16 + SP (hex)

Example 2

The offset of the data byte = 0020h

The value of the data segment register (DS) = 3000h

Physical address of the data byte

DS	3	0	0	0	0	——— Implied Zero
Offset	+	0	0	2	0	
Physical Address		3	0	0	2	0

This calculation can be expressed as physical address = DS (Hex) × 16 + Data byte offset (hex).

Example 3

The value of the Instruction Pointer, holding address of the instruction = 1234h

The value of the code segment register (CS) = 448Ah

Physical address of the instruction

CS	4	4	8	A	0	——— ImpliedZero
	+	1	2	3	4	
IP						
Physical Address		4	5	A	0	4

Physical Address = CS (Hex) × 16 + IP

(c) Instruction Pointer

The instruction pointer points to the offset of the current instruction in the code segment. It is used for calculating the address of instruction as shown above.

1.3.2 Execution Unit (EU)

Execution unit performs all the ALU operations. The execution unit of 8086 is of 16 bits. It also contains the control unit, which instructs bus interface unit about which memory location to access, and what to do with the data. Control unit also performs decoding and execution of the instructions. The EU consists of the following:

(a) Control Circuitry, Instruction Decoder and ALU

The 8086 control unit is primarily micro-programmed control. In addition it has an instruction decoder, which translates an instruction into sequence of micro operations. The ALU performs the required operations under the control of CU which issues the necessary timing and control sequences.

(b) Registers

All CPUs have a defined number of operational registers. 8086 has several general purpose and special purpose registers. We will discuss these registers in the following sections.

1.4 REGISTER SET OF 8086

The 8086 registers have five groups of registers. These groupings are done on the basis of the main functions of the registers. These groups are:

General Purpose Register

8086 microprocessors have four general purpose registers namely, AX, BX, CX, DX. All these registers are 16 – bit registers. However, each register can be used as two general-purpose byte registers also. These byte registers are named AH and AL for AX, BH and BL for BX, CH and CL for CX, and DH and DL for DX. The H in register name represents higher byte while L represents lower byte of the 16 bits registers. These registers are primarily used for general computation purposes. However, in certain instruction executions they acquire a special meaning.

AX register is also known as accumulator. Some of the instructions like divide, rotate, shift etc. require one of the operands to be available in the accumulator. Thus, in such instructions, the value of AX should be suitably set prior to the instruction.

BX register is mainly used as a base register. It contains the starting base location of a memory region within a data segment.

CX register is a defined counter. It is used in loop instruction to store loop counter.

DX register is used to contain I/O port address for I/O instruction.

You will experience their usage in various assembly programs discussed later.

Segment Registers

Segment Registers are used for calculating the physical address of the instruction or memory. Segment registers cannot be used as byte registers.

Pointer and Index Registers

The 8086 microprocessor has three pointer and index registers. Each of these registers is of 16 bit and cannot be accessed byte wise. These are Base Pointer (BP), Source Index (SI) and Destination Index (DI). Although they can be used as general purpose registers, their main objective is to contain indexes. BP is used in stack segment, SI in Data segment and DI in Extra Data segment.

Special Registers

A Last in First Out (LIFO) stack is a data structure used for parameter passing, return address storage etc. 8086 stack is 64K bytes. Base of the stack is pointed to by the stack segment (SS) register while the offset or top of the stack is stored in Stack Pointer (SP) register. Please note that although the memory in 8086 has byte addresses, stack is a word stack, which is any push operation will occupy two bytes.

Flags Register

A flag represents a condition code that is 0 or 1. Thus, it can be represented using a flip-flop. 8086 employs a 16-bit flag register containing nine flags. The following table shows the flags of 8086.

Flags	Meaning	Comments
Conditional Flags represent result of last arithmetic or logical instruction executed. Conditional flags are set by some condition generated as a result of the last mathematical or logical instruction executed. The conditional flags are:		
CF	Carry Flag	1 if there is a carry bit
PF	Parity Flag	1 on even parity 0 on odd parity
AF	Auxiliary Flag	Set (1) if auxiliary carry for BCD occurs
ZF	Zero Flag	Set if result is equal to zero
SF	Sign Flag	Indicates the sign of the result (1 for minus, 0 for plus)
OF	Overflow Flag	set whenever there is an overflow of the result
Control flags, which are set or reset deliberately to control the operations of the execution unit. The control flags of 8086 are as follows:		
TF	Single step trap flag	Used for single stepping through the program
IF	Interrupt Enable flag	Used to allow/inhibit the interruption of the program
DF	String direction flag	Used with string instruction.

Check Your Progress 1

- What is the purpose of the queue in the bus interface unit of 8086 microprocessors?
.....
.....
.....
- Find out the physical addresses for the following segment register: offset
(a) SS:SP = 0100h:0020h
(b) DS:BX = 0200h:0100h
(c) CS:IP = 4200h:0123h

- State True or False.

T	F
---	---

- BX register is used as an index register in a data segment.

☐

- CX register is assumed to work like a counter.

☐

- (c) The Source Index (SI) and Destination Index(DI) registers in 8086 can also be used as general registers. ☐
- (d) Trag Flag (TR) is a conditional flag. ☐

1.5 INSTRUCTION SET OF 8086

After discussing the basic organization of the 8086 micro-processor, let us now provide an overview of various instructions available in the 8086 microprocessor. The instruction set is presented in the tabular form. An assembly language instruction in the 8086 includes the following:

Label: Op-code Operand(s); Comment

For example, to add the content of AL and BL registers to get the result in AL, we use the following assembly instruction.

NEXT: ADD AL,BL ; AL \leftarrow AL + BL

Please note that NEXT is the label field. It is giving an identity to the statement. It is an optional field, and is used when an instruction is to be executed again through a LOOP or GO TO. ADD is symbolic op-code, for addition operation. AL and BL are the two operands of the instructions. Please note that the number of operands is dependent upon the instructions. 8086 instructions can have zero, one or two operands. An operand in 8086 can be:

1. A register
2. A memory location
3. A constant called literal
4. A label.

We will discuss the addressing modes of these operands in section 1.6.

Comments in 8086 assembly start with a semicolon, and end with a new line. A long comment can be extended to more than one line by putting a semicolon at the beginning of each line. Comments are purely optional, however recommended as they provide program documentation. In the next few sections we look at the instruction set of the 8086 microprocessor. These instructions are grouped according to their functionality.

1.5.1 Data Transfer Instructions

These instructions are used to transfer data from a source operand to a destination operand. The source operand in most of the cases remains unchanged. The operand can be a literal, a memory location, a register, or even an I/O port address, as the case may be. Let us discuss these instructions with the following table:

MNEMONIC	DESCRIPTION	EXAMPLE
MOV des, src	des \leftarrow src; Both the operands should be byte or word. src operand can be register, memory location or an immediate operand des can be register or memory operand. Restriction: Both source and destination cannot be memory operands at the same time.	MOV CX,037AH ; CX register is initialized ; with immediate value ; 037AH. MOV AX,BX ; AX \leftarrow BX

PUSH operand	Pushes the operand into a stack. $SP \leftarrow SP - 2;$ value [TOS] \leftarrow operand. Initialise stack segment register, and the stack pointer properly before using this instruction. No flags are effected by this instruction. The operand can be a general purpose register, a segment register, or a memory location. Please note it is a word stack and memory address is a byte address, thus, you decrement by 2. Also you decrement as SP is initialised to maximum offset and condition of stackful is a zero offset (so it is a reversed stack)	PUSH BX ; decrement stack pointer ; by; two, and copy BX to ; stack. ; decrement stack pointer ; by two, and copy ; BX to stack
POP des	POP a word from stack. The des can be a general-purpose register, a segment register (except for CS register), or a memory location. Steps are: $des \leftarrow$ value [TOS] $SP \leftarrow SP + 2$	POP AX ; Copy content for top ; of stack to AX.
XCHG des, src	Used to exchange bytes or words of src and des. It requires at least one of the operands to be a register operand. The other can be a register or memory operand. Thus, the instruction cannot exchange two memory locations directly. Both the operands should be either byte type or word type. The segment registers cannot be used as operands for this instruction.	XCHG DX,AX ; Exchange word in DX ; with word in AX
XLAT	Translate a byte in AL using a table stored in the memory. The instruction replaces the AL register with a byte from the lookup table. This instruction is a complex instruction.	Example is available in Unit 3.
IN accumulator, port address	It transfers a byte or word from specified port to accumulator register. In case an 8-bit port is supplied as an operand then the data byte read from that part will be transferred to AL register. If a 16-bit port is read then the AX will get 16 bit word that was read. The port address can be an immediate operand, or contained in DX register. This instruction does not change any flags.	IN AL,028h ; read a byte from port ; 028h to AL register
OUT port address, Accumulator	It transfers a byte or word from accumulator register to specified port. This instruction is used to output on devices like the monitor or the printer.	
LEA register, source	Load “effective address” (refer to this term in block 2, Unit 1 in addressing modes) of operand into specified 16 – bit register. Since, an address is an offset in a segment and maximum can	LEA BX, PRICES ; Assume PRICES is ; an array in the data ; segment. The ; instruction loads the

	be of 16 bits, therefore, the register can only be a 16-bit register. LEA instruction does not change any flags. The instruction is very useful for array processing.	; offset of the first byte of ; PRICES directly into ; the BX register.
LDS des-reg	It loads data segment register and other specified register by using consecutive memory locations.	LDS SI, DATA ; DS ← content of memory ; location DATA & ; DATA + 1 ; SI ← content of ; memory locations ; DATA + 2 & DATA + ; 3
LES des-reg	It loads ES register and other specified register by using consecutive memory locations. This instruction is used exactly like the LDS except in this case ES & other specified registers are initialized.	
LAHF	Copies the lower byte of flag register to AH. The instruction does not change any flags and has no operands.	
SAHF	Copies the value of AH register to low byte of flag register. This instruction is just the opposite of LAHF instruction. This instruction has no operands.	
PUSHF	Pushes flag register to top of stack. SP ← SP – 2; stack [SP] ← Flag Register.	
POPF	Pops the stack top to Flag register. Flag register ← stack [SP] SP ← SP + 2	

1.5.2 Arithmetic Instructions

MNEMONIC	DESCRIPTION	EXAMPLE
ADD	Adds byte to byte, or word to word. The source may be an immediate operand, a register or a memory location. The rules for operands are the same as that of MOV instruction. To add a byte to a word, first copy the byte to a word location, then fill up the upper byte of the word with zeros. This instruction effects the following flags: AF, CF, OF, PF, SF, ZF.	ADD AL,74H ; Add the number 74H to ; AL register, and store the ; result back in AL ADD DX,BX ; Add the contents of DX to ; BX and store the result in ; DX, BX remains ; unaffected.
ADC des, src	Add byte + byte + carry flag, or word + word + carry flag. It adds the two operands with the carry flag. Rest all the details are the same as that of ADD instruction.	
INC des	It increments specified byte or word operand by one. The operand can be a register or a memory location. It can effect AF, SF, ZF, PF, and OF flags. It does not affect the carry flag, that is, if you increment a byte operand	INC BX ; Add 1 to the contents of ; BX register INC BL ; Add 1 to the contents of ; BL register

	having 0FFH, then it will result in 0 value in register and no carry flag.	
AAA	ASCII adjusts after addition. The data entered from the terminal is usually in ASCII format. In ASCII 0-9 are represented by codes 30-39. This instruction allows you to add the ASCII codes instead of first converting them to decimal digit using masking of upper nibble. AAA instruction is then used to ensure that the result is the correct unpacked BCD.	ADD AL,BL ; AL=00110101, ASCII 05 ; BL=00111001, ASCII 09 ; after addition ; AL = 01101110, that is, ; 6EH- incorrect ; temporary result AAA ; AL = 00000100. ; Unpacked BCD for 04 ; carry = 1, indicates ; the result is 14
DAA	Decimal (BCD) adjust after addition. This is used to make sure that the result of adding two packed BCD numbers is adjusted to be a correct BCD number. DAA only works on AL register.	; AL = 0101 1001 (59 ; BCD) ; BL = 0011 0101 (35 ; BCD) ADD AL, BL ; AL = 10001101 or ; 8Eh (incorrect BCD) DAA ; AL = 1001 0100 ; = 94 BCD : Correct.
SUB des, src	Subtract byte from byte, or word from word. (des ← des – src). For subtraction the carry flag functions as a borrow flag, that is, if the number in the source is greater than the number in the destination, the borrow flag is to set 1. Other details are equivalent to that of the ADD instruction.	SUB AX, 3427h ; Subtract 3427h from AX ; register, and store the ; result back in AX
SBB des, src	Subtract operands involving previous carry if any. The instruction is similar to SUB, except that it allows us to subtract two multibyte numbers, because any borrow produced by subtracting less-significant byte can be included in the result using this instruction.	SBB AL,CH ; subtract the contents ; of CH and CF from AL ; and store the result ; back in AL.
DEC src	Decrement specified byte or specified word by one. Rules regarding the operands and the flags that are affected are same as INC instruction. Please note that if the contents of the operand is equal to zero then after decrementing the contents it becomes 0FFH or 0FFFFH, as the case may be. The carry flag in this case is not affected.	DEC BP ; Decrement the contents ; of BP ; register by one.
NEG src	Negate - creates 2's complement of a given number, this changes the sign of a number. However, please note that if you apply this instruction on operand having value –128 (byte operand) or –32768 (word operand) it will result in overflow condition. The overflow (OF) flag will be set to	NEG AL ; Replace the number in ; AL with it's 2's ; complement

	indicate that operation could not be done.	
CMP des,src	It compares two specified byte operands or two specified word operands. The source and destination operands can be an immediate number, a register or a memory location. But, both the operands cannot be memory locations at the same time. <i>The comparison is done simply by internally subtracting the source operand from the destination operand.</i> The value of source and the destination, operands is not changed, but the flags are set to indicate the results of the comparison.	CMP CX,BX ; Compare the CX register ; with the BX register ; In the example above, the ; CF, ZF, and the SF flags ; will be set as follows. ; CX=BX 0 1 0; result of ; subtraction is zero ; CX>BX 0 0 0; no borrow ; required therefore, CF=0 ; CX<BX 1 0 1 ; subtraction require ; borrow, so CF=1
AAS	ASCII adjust after subtraction. This instruction is similar to AAA (ASCII adjust after addition) instruction. The AAS instruction works on the AL register only. It updates the AF and CF flags, but the OF, PF, SF and the ZF flags remain undefined.	; AL = 0011 0101 ASCII 5 ; BL = 0011 1001 ASCII 9 SUB AL,BL ; (5-9) result: ; AL= 1111 1100 = - 4 in ; 2's complement, CF = 1 AAS ;result: ; AL = 0000 0100 = ; BCD 04, ; CF = 1 borrow needed.
DAS	Decimal adjust after subtraction. This instruction is used after subtracting two packed BCD numbers to make sure the result is the packed BCD. DAS only works on the AL register. The DAS instruction updates the AF, CF, SF, PF and ZF flags. The overflow (OF) is undefined after DAS.	; AL=86 BCD ; BH=57 BCD SUB AL,BH ; AL=2Fh, CF =0 DAS ; Results in AL = 29 BCD
MUL src	This is an unsigned multiplication instruction that multiplies two bytes to produce a word operand or two words to produce a double word such as: $AX \leftarrow AL * src$ (byte multiplication src is also byte) $DX \text{ or } AX \leftarrow AX * src$ (word multiplication is two word). This instruction assumes one of the operand in AL (byte) or AX (word): the src operand can be register or memory operand. If the most significant word of the result is zero then, the CF and the OF flags are both made zero. The AF, SF, PF, ZF flags are not defined after the MUL instruction. If you want to multiply a byte with a word, then first convert byte to a word operand.	MOV AX,05; AX=05 MOV CX,02; CX=02 MUL CX ; results in DX=0 ; AX=0Ah
AAM	ASCII adjust after multiplication. Please note that two ASCII numbers cannot be multiplied directly. To multiply first convert the ASCII	; AL=0000 0101 unpacked ; BCD 05 ; BH=0000 1001 unpacked ; BCD 09

	number to numeric digits by masking off the upper nibble of each byte. This leaves unpacked BCD in the register. AAM instruction is used to adjust the product to two unpacked BCD digits in AX after the multiplication has been performed. AAM defined by the instruction while the CF, OF and the AF flags are left undefined.	MUL BH ; AX=AL * BH=002Dh AAM ; AX=00000100 00000101 ; BCD 45 : Correct result
DIV src	This instruction divides unsigned word by byte, or unsigned double word by word. For dividing a word by a byte, the word is stored in AX register, divisor the src operand and the result is obtained in AH : remainder AL: quotient. It can be represented as: AH: Remainder } ← AX/ src AL: Quotient Similarly for double word division by a word we have DX: Remainder } ← DX:AX/ src AX: Quotient A division by zero result in run time error. The divisor src can be either in a register or a memory operand.	; AX = 37D7h = 14295 ; decimal ; BH = 97h = 151 decimal DIV BH ; AX / BH quotient ; AL = 5Eh = 94 ; decimal RemainderAH = ; 65h = 101 ; decimal
IDIV	Divide signed word by byte or signed double word by word. For this division the operand requirement, the general format of the instruction etc. are all same as the DIV instruction. IDIV instruction leaves all flags undefined.	; AL = 11001010 = -26h = ; - 38 decimal ; CH = 00000011 = + 3h = ; 3 decimal ; According to the operand ; rules to divide by a byte ; the number should be ; present in a word register, ; i.e. AX. So, first convert ; the operand in AL to word ; operand. This can be done ; by sign extending the ; AL register, ; this makes AX ; 11111111 11001010. ; (Sign extension can also ; be done with the help of ; an instruction, discussed ; later) IDIV CH ; AX/CH ; AL = 11110100 = - 0CH ; = -12 Decimal ; AH = 11111110 = -02H = ; - 02 Decimal ; Although the quotient is ; actually closer to -13 ; (-12.66667) than -12, but ; 8086 truncates the result ; to give -12.
AAD	ASCII adjust after division. The BCD numbers are first unpacked, by	; AX= 0607 unpacked ; BCD for 6

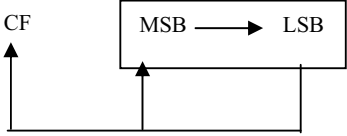
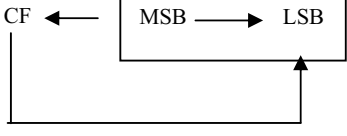
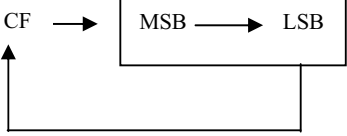
	masking off the upper nibble of each byte. Then ADD instruction is used to convert the unpacked BCD digits in AL and AH registers to adjust them to equivalent binary prior to division. Such division will result in unpacked BCD quotient and remainder. The PF, SF, ZF flags are updated, while the AF, CF, and the OF flags are left undefined.	; and 7 CH = 09h AAD ; adjust to binary before ; division AX= 0043 = ; 043h = 67 Decimal DIV CH ; Divide AX by unpacked ; BCD in CH ; AL = 07 unpacked BCD ; AH = 04 unpacked BCD ; PF = SF = ZF = 0
CBW	Fill upper-byte or word with copies of sign bit of lower bit. This is called sign extension of byte to word. This instruction does not change any flags. This operation is done with AL register in the result being stored in AX.	; AL = 10011011 = -155 ; decimal AH = 00000000 CBW ;convert signed ; byte in AL to signed ; word in AX = 11111111 ; 10011011 = -155 decimal
CWD	Fill upper word or double word with sign bit of lower word. This instruction is an extension of the previous instruction. This instruction results in sign extension of AX register to DX:AX double word.	; DX : 0000 0000 0000 0000 ; AX : 1111 0000 0101 0001 CWD ; DX:AX = 1111 1111 1111 1111; ; 1111 0000 0101 0001

1.5.3 Bit Manipulation Instructions

These instructions are used at the bit level. These instructions can be used for testing a zero bit, set or reset a bit and to shift bits across registers. Let us look into some such basic instructions.

MNEMONIC	DESCRIPTION	EXAMPLE
NOT des	Complements each bit to produce 1's complement of the specified byte or word operand. The operand can be a register or a memory operand.	; BX = 0011 1010 0001 0000 NOT BX ; BX = 1100 0101 1110 1111
AND des, src	Bitwise AND of two byte or word operands. The result is $des \leftarrow des \text{ AND } src$. The source can be an immediate operand a register, or a memory operand. The destination can be a register or a memory operand. Both operands cannot be memory operands at the same time. The CF and the OF flags are both zero after the AND operation. PF, SF and ZF area updated, Afis left undefined.	; BH = 0011 1010 before AND BH, 0Fh ; BH = 0000 1010 ; after the AND operation
OR des, src	OR each corresponding bits of the byte or word operands. The other operands rules are same as AND. $des \leftarrow des \text{ OR } src$; BH = 0011 1010 before OR BH, 0Fh ; BH = 0011 1111 after
XOR des,src	XOR each corresponding bit in a byte or word operands rules are two same as AND and OR. $des \leftarrow Des + src$; BX = 00111101 01101001 ; CX = 00000000 11111111 XOR BX,CX ; BX=0011110110010110 ; Please note, that the bits in ; the lower byte are inverted.

TEST des, src	AND the operands to update flags, but donot change operands value. It can be used to set and test conditions.CF and OF are both set to zero, PF, SF and ZF are all updated, AF is left undefined after the operation.	; AL = 0101 0001 TEST AL, 80h. ; This instruction would ; test if the MSB bit of the AL ; register is zero or one. After ; the TEST operation ZF will ; be set to 1 if the MSB of AL ; is zero.
SHL/SAL des, count	Shift bits of word or byte left, by count. It puts zero(s) in LSB(s). MSB is shifted into the carry flag. If more than one bits are shifted left, then the CF gets the most recently moved MSB. If the number of bits desired to be shifted is only 1, then the immediate number. 1 can be written as one of the operands. However, if the number of bits desired to be shifted is more than one, then the second operand is put in CL register.	SAL BX, 01 ; if CF = 0 ; BX = 1000 1001 ; result : CF = 1 ; BX = 0001 0010
SHR des, count	It shifts bits of a byte or word to register put zero in MSB. LSB is moved into CF.	SHR BX,01 ; if CF = 0 ; BX = 1000 1001 ; result: CF = 1 ; BX = 0100 0100 MOV CL, 02 SHR BX, CL ; with same BX, the ; result would be ; CF = 0 ; BX = 0010 0100
SAR des, count	Shift bits of word or byte right, but it retains the value of new MSB to that of old MSB. This is also called arithmetic shift operation, as it does not change the MSB, which is sign bit of a number.	; AL=0001 1101 = +29 ; decimal, CF = 0 SAR AL, 01 ; AL = 0000 1110 = +14 ; decimal, CF = 1 ; OF = PF = SF = ZF = 0 ; BH = 1111 0011 = -13 ; decimal SAR BH,01 ; BH = 1111 1001 = -7 ; decimal, CF = 1 ; OF = ZF = 0 ; PF = SF = 1
ROL des, count	Rotate bits of word or byte left, MSB is transferred to LSB and also to CF. Diagrammatically, it can be represented as: <div style="text-align: center;"> </div> The operation is called rotate as it circulates bits. The operands can be register or memory operand.	
ROR des, count	Rotate bits of word or byte right,	; CF = 0,

	<p>LSB is transferred to MSB and also to CF. The same can be represented diagrammatically as follows:</p> 	<pre>; BX = 0011 1011 0111 0101 ROR BX, 1 ; results ; CF = 1, ; BX = 1001 1101 1011 1010</pre>
RCL des, count	<p>Rotate bits of words or byte left, MSB to CF and CF to LSB. The operation is circular and involves carry flag in rotation.</p> 	
RCR des, count	<p>Rotate bits of word or byte right, LSB to CF and CF to MSB. This instruction rotates left.</p> 	

Check Your Progress 2

1. Point out the error/ errors in the following 8086 assembly instruction (if any)?

- PUSHF AX
- MOV AX, BX
- XCHG MEM_WORD1, MEM_WORD2
- AAA BL, CL
- IDIV AX, CH

2. State True or False in the context of 8086 assembly language.

T	F
---	---

- LEA and MOV instruction serve the same purpose. The only difference between the two is the type of operands they take. ☐
- NEG instruction produces 1's complement of a number. ☐
- MUL instruction assumes one of the operands to be present in the AL or AX register. ☐
- TEST instruction performs an OR operation, but does not change the value of operands. ☐
- Suppose AL contains 0110 0101 and CF is set, then instructions ROL AL and RCL AL will produce the same results. ☐

1.5.4 Program Execution Transfer Instructions

These instructions are the ones that causes change in the sequence of execution of instruction. This change can be through a condition or sometimes may be unconditional. The conditions are represented by flags. For example, an instruction may be jump to an address if zero flag is set, that is the last ALU operation has resulted in zero value. These instructions are often used after a compare instruction, or some arithmetic instructions that are used to set the flags, for example, ADD or SUB. LOOP is also a conditional branch instruction and is taken till loop variable is below a certain count.

Please note that a "/" is used to separate two mnemonics which represent the same instruction.

MNEMONIC	DESCRIPTION	EXAMPLE
CALL proc 1	<p>This function results in a procedure/ function call. The return address is saved on the stack. There are two basic types of CALLS. NEAR or Intra-Segment calls: if the call is made to a procedure in the same segment as the calling program. FAR or Inter segment call: if the call is made to a procedure in the segment, other than the calling program. The saved return address for NEAR procedure call is just the IP. For FAR Procedure call IP and CS are saved as return address.</p> <p>A procedure can also be called indirectly, by first initializing some 16-bit register, or some other memory location with the new addresses as follows.</p>	<p>CALL proc1 CALL proc2</p> <p>The new instruction address is determined by name declaration proc1 is a near procedure, thus, only IP is involved. proc2 involves new CS: IP pair.</p> <p>On call to proc1 stack \leftarrow IP IP \leftarrow address offset of proc1 on call to proc2 Stack [top] \leftarrow CS Stack [top] \leftarrow IP CS \leftarrow code segment of proc2 IP \leftarrow address offset of proc2</p> <p>Here we assume that proc1 is defined within the same segment as the calling procedure, while proc2 is defined in another segment. As far as the calling program is concerned, both the procedures have been called in the same manner. But while declaring these procedures, we declare proc1 as NEAR procedure and proc2 as FAR procedure, as follows:</p> <pre>proc1 PROC NEAR proc2 PROC FAR LEA BX, proc1 ; initialize BX with the ; offset of the procedure ; proc1 CALL BX ; CALL proc1 indirectly ; using BX register</pre>
RET number	It returns the control from	RET 6

	procedure to calling program. Every CALL should be a RET instruction. A RET instruction, causes return from NEAR or FAR procedure call. For return from near procedure the values of the instruction pointer is restored from stack. While for far procedure the CS:IP pair get is restored. RET instruction can also be followed by a number.	; In this case, 8086 ; increments the stack ; pointer by this number ; after popping off the IP ; (for new) or IP and CS ; registers (for far) from ; the stack. This cancels ; the local parameters, or ; temporary parameters ; created by the ; programmer. RET ; instruction does not ; affect any flags.
JMP Label	Unconditionally go to specified address and get next instruction from the label specified. The label assigns the instruction to which jump has to take place within the program, or it could be a register that has been initialised with the offset value. JMP can be a NEAR JMP or a FAR jump, just like CALL.	JMP CONTINUE ; CONTINUE is the label ; given to the instruction ; where the control needs ; to be transferred. JMP BX ; initialize BX with the ; offset of the instruction, ; where the control needs ; to be transferred.
Conditional Jump	All the conditional jumps follow some conditional statement, or any instruction that affects the flag.	MOV CX, 05 MOV BX, 04 CMP CX, BX ; this instruction will set ; various flags like the ZF, ; and the CF. JE LABEL1 ; conditional jump can ; now be applied, which ; checks for the ZF, and if ; it is set implying CX = ; BX, it makes ; a jump to LABEL1, ; otherwise the control ; simply falls ; through to next ; instruction ; in the above example as ; CX is not equal to BX ; the jump will not take ; place and the next ; instruction to conditional ; jump instruction will be ; executed. However, if ; JNE (Jump if not equal ; to) or JA (Jump if ; above), ; or JAE (Jump ; above or ; equal) jump instructions ; if applied instead of JE, ; will cause the conditional ; jump to occur.
	All the conditional jump instructions which are given below are self explanatory.	
JA/JNBE	Jump if above / Jump if not below nor equal	

JAE/JNB	Jump if above or equal/ Jump if not below	
JB/JNAE	Jump if below/ Jump if not above nor equal	
JBE/JNA	Jump if below or equal/ Jump if not above	
JC	Jump if carry flag set	
JE/JZ	Jump if equal / Jump if zero flag is set	
JNC	Jump if not carry	
JNE/JNZ	Jump if not equal / Jump if zero flag is not set	
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 even	
JNP/JPO	Jump if not parity / Jump if parity odd	
JG/JNLE	Jump if greater than / Jump if not less than nor equal	
JA/JNL	Jump if above / Jump if not less than	
JL/JNGE	Jump if less than / Jump if not greater than nor equal	
JLE/JNG	Jump if less than or equal to / Jump if not greater than	
JS	Jump if sign flag is set	
JNS	Jump if sign flag is not set	
LOOP label	This is a looping instruction of assembly. The number of times the looping is required is placed in CX register. Each iteration decrements CX register by one implicitly, and the Zero Flag is checked to check whether to loop again. If the zero flag is not set (CX is zero) greater than the control goes back to the specified label in the instruction, or else the control falls through to the next instruction. The LOOP instruction expects the label destination at offset of – 128 to +127 from the loop instruction offset.	; Let us assume we want to ; add 07 to AL register, ; three times. MOV CX,03 ; count of iterations L1: ADD AL,07 LOOP L1 ; loop back to L1, ; until CX ; becomes equal to zero ; Loop affects no flags.
LOOPE/ LOOPZ label	Loop through a sequence of instructions while zero flag = 1 and CX is not equal to zero. There are two ways to exit out of the loop, firstly, when the count in the CX register becomes equal to zero, or when the quantities that are being compared become unequal.	Let us assume we have an array of 20 bytes. We want to see if all the elements of that array are equal to 0FFh or not. To scan 20 elements of the array, we loop 20 times. And we come out of the loop, when either the count of iterations has become equal to 20, or in other words CX register has

		<p>decremented to zero, which means all the elements of the array are equal to 0FFh, or an element in the array is found which is not equal to 0FFh. In this case, the CX register may still be greater than zero, when the control comes out. This can be coded as follows: (Please note here that you might not understand everything at this place, that is because you are still not familiar with the various addressing modes. Just concentrate on the LOOPE instruction):</p> <pre> MOV BX, OFFSET ARRAY ; Point BX at the start ; of the ARRAY DEC BX ; put number of ; array elements in CX MOV CX,10 L1: INC BX ; point to ; next element in array CMP [BX],0FFh ; compare array element ; with 0FFh LOOPE L1 ; When the control comes ; out of the loop, it has ; either scanned all the ; elements and found them ; to be all equal to 0FFh, or ; it is pointing to the first ; non-0FFh, element in the ; array. </pre>
LOOPNE/LOOPNZ label	This instruction causes Loop through a sequence of instructions while zero flag = 0 and CX is not equal to zero. This instruction is just the opposite of the previous instruction in its functionality.	
JCXZ label	Jump to specified address if CX =0. This instruction will cause a jump, if the value of CX register is zero. Otherwise it will proceed with the next instruction in sequence.	This instruction is useful when you want to check whether CX is zero even prior to entering into a loop. Please note that LOOP instruction executes the loop at least once before decrementing and checking the value of CX register. Thus, CX=0 will execute the loop once and decrement the CX register,

		making it 0FFFFh, which is non zero. This will cause FFFFh times execution of loop. To avoid such type of conditions you can proceed as follows: JCXZ SKIP_LOOP ; if CX is already 0, skip ; loop L1: SUB [BX],07h INC BX LOOP L1 ; loop until CX=0 SKIP_LOOP:
--	--	--

In addition to these instructions, there are other interrupt handling instructions also, which too transfer the control of the program to some specified location. We will discuss these instructions in later units.

1.5.5 String Instructions

These are a very strong set of 8086 instructions as these instructions process strings, in a compact manner, thus, reducing the size of the program by a considerable amount. “String” in assembly is just a sequentially stored bytes or words. A string often consists of ASCII character codes. A subscript B following the instruction indicates that the string of bytes is to be acted upon, while “W” indicates that it is the string of words that is being acted upon.

MNEMONIC	DESCRIPTION	EXAMPLES
REP	This is an instruction prefix. It causes repetition of the following instruction till CX becomes zero. REP. It is not an instruction, but it is an instruction prefix that causes the CX register to be decremented. This prefix causes the string instruction to be repeated, until CX becomes equal to zero.	REP MOVSB STR1, STR2 The above example copies byte by byte contents. The CX register is initialized to contain the length of source string REP repeats the operation MOVSB that copies the source string byte to destination byte. This operation is repeated until the CX register becomes equal to zero.
REPE/REPZ	It repeats the instruction following until CX =0 or ZF is not equal to one. REPE/REPZ may be used with the compare string instruction or the scan string instruction. REPE causes the string instruction to be repeated, till compared bytes or words are equal, and CX is not yet decremented to zero.	
REPNE/REPNZ	It repeats instruction following it until CX =0 or ZF is equal to 1. This comparison here is just inverse of REPE except for CX, which is checked to be equal to zero.	
MOVS/MOVSb/ MOVSW	It causes moving of byte or word from one string to another. This	Assumes both data and extra segment start at address 1000

	<p>instruction assumes that:</p> <ul style="list-style-type: none"> • Source string is in Data segment. • Destination string is in extra data segment • SI stores offset of source string in extra segment • DI stores offset of destination string is in data segment • CX contains the count of operation <p>A single byte transfer requires;</p> <ul style="list-style-type: none"> • One byte transfer from source string to destination • Increment of SI and DI to next byte • Decrement count register that is CX register 	<p>in the memory. Source string starts at offset 20h and the destination string starts at offset 30h. Length of the source string is 10 bytes. To copy the source string to the destination string, proceed as follows:</p> <pre>MOV AX,1000h MOV DS,AX ; initialize data segment and MOV ES,AX ; extra segment MOV SI,20h MOV DI,30h ; load offset of start of ; source string to SI ; Load offset of start of ; destination string to DI MOV CX,10 ; load length of string to CX ; as counter REP MOVSB ; Decrement CX and ; MOVSB until ; CX =0 ; after move SI will be one ; greater than offset of last ; byte in source string, DI ; will be one greater than ; offset of last destination ; string. CX will be equal ; to zero.</pre>
CMPS/CMPSB/ CMPSW	<p>It compares two string bytes or words. The source string and the destination strings should be present in data segment and the extra segment respectively. SI and DI are used as in the previous instruction. CX is used if more than one bytes or words are to be compared, however for such a case appropriate repeating prefix like REP, PEPE etc. need to be used.</p>	<pre>MOV CX,10 MOV SI,OFFSET SRC_STR ; offset of source ; string in SI MOV DI, OFFSET DES_STR ; offset of destination ; string in DI REPE CMPSB ; Repeat the comparison of ; string bytes until ; end of string or until ; compared bytes are not ; equal.</pre>
SCAS/SCASB/ SCASW	<p>It scans a string. Compare a string byte with byte in AL or a string word with a word in AX. The instruction does not change the operands in AL (AX) or the operand in the string. The string to be scanned must be present in the extra segment, and the offset of the string must be contained in the DI register. You can use CX if operation is to be repeated using REP prefixes.</p>	<pre>MOV AL, 0Dh ; Byte to be scanned ; for in AL MOV DI,OFFSET DES_STR MOV CX,10 REPNE SCAS DES_STR ; Compare byte in DES_STR ; with byte in AL register ; Scanning is repeated while ; the bytes are not equal and ; it is not end of string. If a ; carriage return 0Dh is ; found, ZF = DI will point ;</pre>

		at the next byte after the ; carriage return. If a ; carriage return is not ; found then, ZF = 0 and ; CX = 0. SCASB or ; SCASW can be used to ; explicitly state whether ; the byte comparison or the word comparison is ; required.
LODS/LODSB/ LODSW	It loads string byte into AL or a string word into AX. The string byte is assumed to be pointed to by SI register. After the load, the SI pointer is automatically adjusted to point to the next byte or word as the case may be. This instruction does not affect any flag.	MOV SI,OFFSET SRC_STR LODS SRC_STR ; LODSB or LODSW can ; be used to indicate to the ; assembler, explicitly, ; whether it is the byte that ; is required to be loaded or ; the word.
STOS/STOSB/ STOSW	It stores byte from AL or word from AX into the string present in the extra segment with offset given by DI. After the copy, DI is automatically adjusted to point to the next byte or word as per the instruction. No flags are affected.	MOV DI,OFFSET DES_STR STOSB DES_STR

1.5.6 Processor Control Instructions

The objectives of these instructions are to control the processor. This raises two questions:

How can you control processor, as this is the job of control unit?
How much control of processor is actually allowed?

Well, 8086 only allows you to control certain control flags that causes the processing in a certain direction, processor synchronization if more than one processors are attached through LOCK instruction for buses etc.

Note: Please note that these instructions may not be very clear to you right now. Thus, some of these instructions have been discussed in more detail in later units. You must refer to further readings for more details on these instructions.

MNEMONIC	DESCRIPTION	EXAMPLE
STC	It sets carry flag to 1.	
CLC	It clears the carry flag to 0.	
CMC	It complements the state of the carry flag from 0 to 1 or 1 to 0 as the case may be.	CMC; Invert the carry flag
STD	It sets the direction flag to 1. The string instruction moves either forward (increment SI, DI) or backward (decrement SI, DI) based on this flag value. STD instruction does not affect any other flag. The set direction flag causes strings to move from right to left.	
CLD	This is opposite to STD, the string	CLD

	operation occurs in the reverse direction.	; Clear the direction flag ; so that the string pointers ; auto-increment. MOV AX,1000h MOV DS, AX ; Initialize data segment ; and extra segment MOV ES, AX MOV SI, 20h ; Load offset of start of ; source string to SI MOV DI,30h ; Load offset of start of ; destination string to DI MOV CX,10 ; Load length of string to ; CX as counter REP MOVSB ; Decrement CX and ; increment ; SI and DI to point to next ; byte, then MOVSB until ; CX = 0
--	--	--

There are many process control instructions other than these; you may please refer to further reading for such instructions. These instructions include instructions for setting and closing interrupt flag, halting the computer, LOCK (locking the bus), NOP etc.

1.6 ADDRESSING MODES

The basic set of operands in 8086 may reside in register, memory and immediate operand. How can these operands be accessed through various addressing modes? The answer to the question above is given in the following sub-section. Large number of addressing modes help in addressing complex data structures with ease. Some specific Terms and registers roles for addressing:

Base register (BX, BP): These registers are used for pointing to base of an array, stack etc.

Index register (SI, DI): These registers are used as index registers in data and/or extra segments.

Displacement: It represents offset from the segment address.

Addressing modes of 8086

Mode	Description	Example
Direct	Effective address is the displacement of memory variable.	
Register Indirect	Effective address is the contents of a register.	[BX]
		[SI]
		[DI]
		[BP]
Based	Effective address is the sum of a base register and a displacement.	LIST[BX] (OFFSET LIST + BX)
		[BP + 1]
Indexed	Effective address is the sum of an index register and a displacement.	LIST[SI]
		[LIST + DI]
		[DI + 2]
Based Indexed		[BX + SI]

	Effective address is the sum of a base and an index register.	[BX][DI]
		[BP + DI]
Based Indexed with displacement	Effective address is the sum of a base register, an index register, and a displacement.	[BX + SI + 2]

1.6.1 Register Addressing Mode

Operand can be a 16-bit register:

Addressing Mode	Description	Example
AX, BX, CX, DX, SI, DI, BP, IP, CS, DS, ES, SS Or it may be AH, AL, BH, BL, CH, CL, DH, DL	In general, the register addressing mode is the most efficient because registers are within the CPU and do not require memory access.	MOV AL,CH MOV AX,CX

1.6.2 Immediate Addressing Mode

An immediate operand can be a constant expression, such as a number, a character, or an arithmetic expression. The only constraint is that the assembler must be able to determine the value of an immediate operand at assembly time. The value is directly inserted into the machine instruction.

MOV AL,05

Mode	Description	Example
Immediate	Please note in the last examples the expression (2 + 3)/5, is evaluated at assembly time.	MOV AL,10 MOV AL,'A' MOV AX,'AB' MOV AX, 64000 MOV AL, (2 + 3)/5

1.6.3 Direct Addressing Mode

A direct operand refers to the contents of memory at an address implied by the name of the variable.

Mode	Description	Example
DIRECT	The direct operands are also called as relocatable operands as they represent the offset of a label from the beginning of a segment. On reloading a program even in a different segment will not cause change in the offset that is why we call them relocatable. Please note that a variable is considered in Data segment (DS) and code label in code segment (SS) by default. Thus, in the example, COUNT, by	MOV COUNT, CL ; move CL to COUNT (a ; byte variable) MOV AL,COUNT ; move COUNT to AL JMP LABEL1 ; jump to LABEL1 MOV AX,DS:5 ; segment register and ; offset MOV BX,CSEG:2Ch ; segment name and offset MOV AX,ES:COUNT ; segment register and ; variable.

	default will be assumed to be in data segment, while LABEL 1, will be assumed to be in code segment. If we specify, as a direct operand then the address is non-relocatable. Please note the value of segment register will be known only at the run time.	; The offsets of these ; variables are calculated ; with respect to the ; segment name (register) ; specified in the ; instruction.
--	--	---

1.6.4 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/ arrays etc. For this mode two types of registers are used. These are:

- Base register BX, BP
- Index register SI, DI

BX contain offset/ pointer in Data Segment
BP contains offset/ pointer in Stack segment.
SI contains offset/pointer in Data segment.
DI contains offset /pointer in extra data segment.

There are five different types of indirect addressing modes:

1. Register indirect
2. Based indirect
3. Indexed indirect
4. Based indexed
5. Based indexed with displacement.

Mode	Description	Example
Register indirect	Indirect operands are particularly powerful when processing list of arrays, because a base or an index register may be modified at runtime.	<p>MOV BX, OFFSET ARRAY ; point to start of array MOV AL,[BX] ; get first element INC BX ; point to next MOV DL,[BX] ; get second element The brackets around BX signify that we are referring to the contents of memory location, using the address stored in BX. In the following example, three bytes in an array are added together: MOV SI,OFFSET ARRAY ; address of first byte MOV AL,[SI] ; move the first byte to AL INC SI ; point to next byte ADD AL,[SI] ; add second byte INC SI ; point to the third byte ADD AL,[SI] ; add the third byte</p>

Based Indirect and Indexed Indirect	Based and indirect addressing modes are used in the same manner. The contents of a register are added to a displacement to generate an effective address. The register must be one of the following: SI, DI, BX or BP. If the registers used for displacement are base registers, BX or BP, it is said to be base addressing or else it is called indexed addressing. A displacement is either a number or a label whose offset is known at assembly time. The notation may take several equivalent forms. If BX, SI or DI is used, the effective address is usually an offset from the DS register; BP on the other hand, usually contains an offset from the SS register.	; Register added to an offset MOV DX, ARRAY[BX] MOV DX,[DI + ARRAY] MOV DX,[ARRAY + SI] ; Register added to a constant MOV AX,[BP + 2] MOV DL,[DI - 2] ; DI + (-2) MOV DX,2[SI]
-------------------------------------	---	--

Mode	Description	Example
Based Indexed	In this type of addressing the operand's effective address is formed by combining a base register with an index register.	MOV AL,[BP] [SI] MOV DX,[BX + SI] ADD CX,[DI] [BX] ; Two base registers or two ; index registers cannot be ; combined, so the ; following would be ; incorrect: MOV DL,[BP + BX] ; error : two base registers MOV AX,[SI + DI] ; error : two index registers
Based Indexed with Displacement	The operand's effective address is formed by combining a base register, an index register, and a displacement.	MOV DX,ARRAY[BX][SI] MOV AX, [BX + SI + ARRAY] ADD DL,[BX + SI + 3] SUB CX, ARRAY[BP + SI] Two base registers or two index registers cannot be combined, so the following would be incorrect: MOV AX,[BP + BX + 2] MOV DX,ARRAY[SI + DI]

Check Your Progress 3

State True or False.

T	F
---	---

1. CALL instruction should be followed by a RET instruction.

☐

2. Conditional jump instructions require one of the flags to be tested. ☐
3. REP is an instruction prefix that causes execution of an instruction until CX value become 0. ☐
4. In the instruction MOV BX, DX register addressing mode has been used. ☐
5. In the instruction MOV BX,ES:COUNTER the second operand is a direct operand. ☐
6. In the instruction ADD CX, [DI] [BX] the second operand is a based index operand, whose effective address is obtained by adding the contents of DI and BX registers. ☐
7. The instruction ADD AX,ARRAY [BP + SI] is incorrect. ☐

1.7 SUMMARY

In this unit, we have studied one of the most popular series of microprocessors, viz., Intel 8086. It serves as a base to all its successors, 8088, 80186, 80286, 80486, and Pentium. The successors of 8086 can be directly run on any successors. Therefore, though, 8086 has become obsolete from the market point of view, it is still needed to understand advanced microprocessors.

To summarize the features of 8086, we can say 8086 has:

- a 16-bit data bus
- a 20-bit address bus
- CPU is divided into Bus Interface Unit and Execution Unit
- 6-byte instruction prefetch queue
- segmented memory
- 4 general purpose registers (each of 16 bits)
- instruction pointer and a stack pointer
- set of index registers
- powerful instruction set
- powerful addressing modes
- designed for multiprocessor environment
- available in versions of 5Mhz and 8Mhz clock speed.

You can refer to further readings for obtaining more details on INTEL and Motorola series of microprocessors.

1.8 SOLUTIONS/ANSWERS

Check Your Progress 1

1. It improves execution efficiency by storing the next instruction in the register queue.
2.
 - a) $0100 \times 10h (-16 \text{ in decimal}) + 0020h$
 $= 01000h + 0020h$
 $= 01020h$
 - b) $0200h \times 10h + 0100h$
 $= 02000h + 0100h$
 $= 02100h$
 - c) $4200h \times 10h + 0123$

= 42000h + 0123h
= 42123h

3. a) False b) True c) True d) False

Check Your Progress 2

1.
 - (a) PUSHF instructions do not take any operand.
 - (b) No error.
 - (c) XCHG instruction cannot have two memory operands
 - (d) AAA instruction performs ASCII adjust after addition. It is used after an ASCII Add. It does not have any operands.
 - (e) IDIV assumes one operand in AX so only second operand is needed to be specified.
2.
 - (a) False
 - (b) False
 - (c) True
 - (d) False
 - (e) False

Check Your Progress 3

1. False
2. True
3. True
4. True
5. True
6. True
7. False

UNIT 2 INTRODUCTION TO ASSEMBLY LANGUAGE PROGRAMMING

Structure	Page No.
2.0 Introduction	35
2.1 Objectives	35
2.2 The Need and Use of the Assembly Language	35
2.3 Assembly Program Execution	36
2.4 An Assembly Program and its Components	41
2.4.1 The Program Annotation	
2.4.2 Directives	
2.5 Input Output in Assembly Program	45
2.5.1 Interrupts	
2.5.2 DOS Function Calls (Using INT 21H)	
2.6 The Types of Assembly Programs	51
2.6.1 COM Programs	
2.6.2 EXE Programs	
2.7 How to Write Good Assembly Programs	53
2.8 Summary	55
2.9 Solutions/Answers	56
2.10 Further Readings	56

2.0 INTRODUCTION

In the previous unit, we have discussed the 8086 microprocessor. We have discussed the register set, instruction set and addressing modes for this microprocessor. In this and two later units we will discuss the assembly language for 8086/8088 microprocessor. Unit 1 is the basic building block, which will help in better understanding of the assembly language. In this unit, we will discuss the importance of assembly language, basic components of an assembly program followed by discussions on the program developmental tools available. We will then discuss what are COM programs and EXE programs. Finally we will present a complete example. For all our discussions, we have used Microsoft Assembler (MASM). However, for different assemblers the assembly language directives may change. Therefore, before running an assembly program you must consult the reference manuals of the assembler you are using.

2.1 OBJECTIVES

After going through this unit you should be able to:

- define the need and importance of an assembly program;
 - define the various directives used in assembly program;
 - write a very simple assembly program with simple input – output services;
 - define COM and EXE programs; and
 - differentiate between COM and EXE programs.
-

2.2 THE NEED AND USE OF THE ASSEMBLY LANGUAGE

Machine language code consists of the 0-1 combinations that the computer decodes directly. However, the machine language has the following problems:

- It greatly depends on machine and is difficult for most people to write in 0-1 forms.
- DEBUGGING is difficult.
- Deciphering the machine code is very difficult. Thus program logic will be difficult to understand.

To overcome these difficulties computer manufacturers have devised English-like words to represent the binary instruction 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 more 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 modules: as the programs are closer to machine, you may be able to write highly optimised programs. 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 as of today the compilers produce a long list of code for every instruction as compared to assembly language, which produces single line of code for a single instruction. This will be true especially in case of string related programs.

On the other hand 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 in construction of their system.

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 HLL with some highly efficient but non-portable routines. It will be worth mentioning here that UNIX mostly is written in C but has about 5-10% machine dependent assembly code. Similarly in telecommunication application assembly routine exists for enhancing efficiency.

2.3 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 the assembler is used to translate the program into machine code.

There are 2 ways of converting an assembly language program into machine language:

- 1) Manual assembly
- 2) By using an assembler.

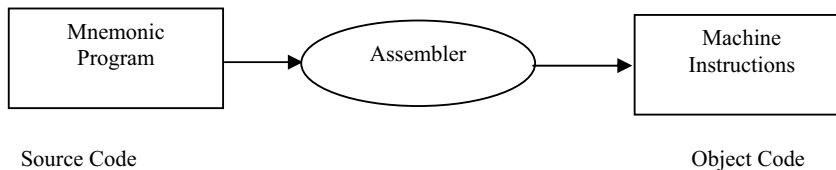
Manual Assembly

It was an old method that required the programmer to translate each opcode into its numerical machine language representation by looking up a table of the microprocessor instructions set, which contains both assembly and machine language instructions. Manual assembly is acceptable for short programs but becomes very inconvenient for large programs. The Intel SDK-85 and most of the earlier university kits were programmed using manual assembly.

Using an Assembler

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.



The steps required to assemble, link and execute a program are:

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 creates a header immediately in front of the generated .OBJ module; part of the header contains information about incomplete addresses. The .OBJ module is not quite in executable form.

Step 2: The link step involves converting the .OBJ module to an .EXE machine code module. The linker's tasks include completing any address left open by the assembler and combining separately assembled programs into one executable module.

The linker:

- combines assembled module into one executable program
- generates an .EXE module and initializes with special instructions to facilitate its subsequent loading for execution.

Step 3: The last step is to load the program for execution. Because the loader knows where the program is going to load in memory, it is now able to resolve any remaining address still left incomplete in the header. The loader drops the header and creates a program segment prefix (PSP) immediately before the program is loaded in memory.

Figure 2: Program Assembly

All this conversion and execution of Assembly language performed by Two-pass assembler.

Two-pass assembler: Assemblers typically make two or more passes through a source program in order to resolve forward references in a program. A forward reference is defined as a type of instruction in the code segment that is referencing the label of an instruction, but the assembler has not yet encountered the definition of that instruction.

Pass 1: Assembler reads the entire source program and constructs a symbol table of names and labels used in the program, that is, name of data fields and programs labels and their relative location (offset) within the segment.

Pass 1 determines the amount of code to be generated for each instruction.

Pass 2: The assembler uses the symbol table that it constructed in Pass 1. Now it knows the length and relative position of each data field and instruction, it can complete the object code for each instruction. It produces .OBJ (Object file), .LST (list file) and cross reference (.CRF) files.

Tools required for assembly language programming

The tools of the assembly process described below may vary in details.

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 programs can be classified in 2 groups.

- Line editors
- Full screen editors.

Line editors, such as EDIT in MS DOS, work with the manage one line at a time. Full screen editors, such as Notepad, Wordpad etc. manage the full screen or a paragraph at a time. To write text, the user must call the editor under the control of the operating system. As soon as the editor program is transferred from the disk to the system memory, the program control is transferred from the operating system to the editor program. The editor has its own command and the user can enter and modify text by using those commands. Some editor programs such as WordPerfect are very easy to use. At the completion of writing a program, the exit command of the editor program will save the program on the disk under the file name and will transfer the control to the operating system. If the source file is intended to be a program in the 8086 assembly language the user should follow the syntax of the assembly language and the rules of the assembler.

Assembler

An assembly program is used to transfer assembly language mnemonics to the binary code for each instruction, after the complete program has been written, with the help of an editor it is then assembled with the help of an assembler.

An assembler works in 2 phases, i.e., it reads your source code two times. In the first pass the assembler collects all the symbols defined in the program, along with their offsets in symbol table. On the second pass through the source program, it produces binary code for each instruction of the program, and give all the symbols an offset with respect to the segment from the symbol table.

The assembler generates three files. The object file, the list file and cross reference file. The object file contains the binary code for each instruction in the program. It is created only when your program has been successfully assembled with no errors. The errors that are detected by the assembler are called the symbol errors. For example,

`MOVE AX1, ZX1 ;`

In the statement, it reads the word MOVE, it tries to match with the mnemonic sets, as there is no mnemonic with this spelling, it assumes it to be an identifier and looks for its entry in the symbol table. It does not even find it there therefore gives an error as undeclared identifier.

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 assemblers available on PC are MASM, TURBO etc.

Linker

For modularity of your programs, it is better to break your program into several sub routines. It is even better to put the common routine, like reading a hexadecimal number, writing hexadecimal number, etc., which could be used by a lot of your other programs into a separate file. These files are 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 link file, which contains the binary code for all compound modules. The linker also produces link maps, which contains the address information about the linked files. The linker however does not assign absolute addresses to your program. It only assigns continuous relative addresses to all the modules linked starting from the zero. This form a program is said to be relocatable because it can be put anywhere in memory to be run.

Loader

Loader is a program which assigns absolute addresses to the program. These addresses are generated by adding the address from where the program is loaded into the memory to all the offsets. Loader comes into action when you want to execute your program. This program is brought from the secondary memory like disk. The file name extension 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 register contents after the execution.
- Trace the execution of the specified segment of the program and display the register and memory contents after the execution of each instruction.
- Disassemble a section of the program, i.e., convert 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 or Emulator
- LINK.EXE, it may be included in the assembler
- DEBUG.COM for debugging if the need so be.

Errors

Two possible kinds of errors can occur in assembly programs:

- a. Programming errors: They are the familiar errors you can encounter in the course of executing a program written in any language.
- b. 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.

2.4 AN ASSEMBLY PROGRAM AND ITS COMPONENTS

Sample Program

In this program we just display:

Line Numbers	Offset	Machine Code	Source Code
0001			DATA SEGMENT
0002	0000		MESSAGE DB "HAVE A NICE DAY!\$"
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	Machine Code	
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

The details of this program are:

2.4.1 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 numbers: There is a good reason for not leaving the determination of segment numbers up to the assembler. It allows programs written in 8086 assembly language to be almost entirely relocatable. They can be loaded practically anywhere in memory and run just as well. Program1 has to store the message "Have a nice day\$" somewhere in memory. It is located in the DATA SEGMENT. Since the

characters are stored in ASCII, therefore it will occupy 15 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 carriage return, a line feed.

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 a name that you apply to an item 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 MAIN is the label for a process as:

```
MAIN PROC FAR
A20: BL,45 ; defines a label A20.
```

Identifier can use alphabet, digit or special character but 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.

2.4.2 Directives

Assembly languages support a number of statements. This enables you to control the way in which a source program assembles and list. These statements, called directives, act only when the assembly is in progress and generate no machine-executable code. Let us discuss some common directives.

1. **List:** A list directive causes the assembler to produce an annotated listing on the printer, the video screen, a disk drive or some combination of the three. An annotated listing shows the text of the assembly language programs, numbers of each statement in the program and the offset associated with each instruction and each datum. The advantage of list directive is that it produces much more informative output.
2. **HEX:** The HEX directive facilitates the coding of hexadecimal values in the body of the program. That statement directs the assembler to treat tokens in the

source file that begins with a dollar sign as numeric constants in hexadecimal notation.

3. **PROC Directive:** The code segment contains the executable code for a program, which consists of one or more procedures defined initially with the PROC directive and ended with the ENDP directive.

Procedure-name PROC FAR ; Beginning of Procedure
Procedure-name ENDP FAR ; End Procedure

4. **END DIRECTIVE:** ENDS directive ends a segment, ENDP directive ends a procedure and END directive ends the entire program that appears as the last statement.
5. **ASSUME Directive:** An .EXE program uses the SS register to address the base of stack, DS to address the base of data segment, CS to address base of the code segment and ES register to address the base of Extra segment. This directive tells the assembler to correlate segment register with a segment name. For example,

ASSUME SS: stack_seg_name, DS: data_seg_name, CS: code_seg_name.

6. **SEGMENT Directive:** The segment directive defines the logical segment to which subsequent instructions or data allocations statement belong. It also gives a segment name to the base of that segment.

The address of every element in a 8086 assembly program must be represented in segment - relative format. That means that every address must be expressed in terms of a segment register and an offset from the base of the segmented addressed by that register. By defining the base of a logical segment, a segment directive makes it possible to set a segment register to address that base and also makes it possible to calculate the offset of each element in that segment from a common base.

An 8086 assembly language program consists of logical segments that can be a code segment, a stack segment, a data segment, and an extra segment.

A segment directive indicates to assemble all statements following it in a single source file until an ENDS directive.

CODE SEGMENT

The logical program segment is named code segment. When the linker links a program it makes a note in the header section of the program's executable file describing the location of the code segment when the DOS invokes the loader to load an executable file into memory, the loader reads that note. As it loads the program into memory, the loader also makes notes to itself of exactly where in memory it actually places each of the program's other logical segments. As the loader hands execution over to the program it has just loaded, it sets the CS register to address the base of the segment identified by the linker as the code segment. This renders every instruction in the code segment addressable in segment relative terms in the form CS: xxxx.

The linker also assumes by default that the first instruction in the code segment is intended to be the first instruction to be executed. That instruction will appear in memory at an offset of 0000H from the base of the code segment, so the linker passes that value on to the loader by leaving another note in the header of the program's executable file.

The loader sets the IP (Instruction Pointer) register to that value. This sets CS:IP to the segment relative address of the first instruction in the program.

STACK SEGMENT

8086 Microprocessor 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. Every 55 milliseconds the real time clock interrupts. 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 when the interruption occurred. All such information gets recorded in the stack. If your program has no stack and if the real time clock were to pulse while the CPU is running your program, there would be no way for the CPU to find the way back to your program when it was through updating the clock. 0400H byte is the default size of allocation of stack. Please note if you have not specified the stack segment it is automatically created.

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: Specifying the data type of assembly.

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; that is 10 zero values.

EQU directive is used to define a name to a constant

CONST EQU 20

Type of number used in data statements can be octal, binary, hexadecimal, decimal and ASCII. The above statement defines a name CONST to a value 20.

Some other examples of using these directives are:

```
TEMP    DB    0111001B    ; Binary value in byte operand
                        ; named temp
VALI    DW    7341Q        ; Octal value assigned to word
                        ; variable
Decimal DB    49          ; Decimal value 49 contained in
                        ; byte variable
HEX     DW    03B2AH       ; Hex decimal value in word
                        ; operand
ASCII   DB    'EXAMPLE'    ; ASCII array of values.
```

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. ☐

2.5 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 constructed using these interrupts.

2.5.1 Interrupts

An **interrupt** causes interruption of an ongoing program. Some of the common interrupts are: keyboard, printer, monitor, an error condition, trap etc.

8086 recognizes two kinds of interrupts: **Hardware** interrupts and **Software** interrupts.

Hardware interrupts are generated when a peripheral Interrupt servicing program requests for some service. A software interrupt causes a call to the operating system. It usually is the **input-output** routine.

Let us discuss the software interrupts in more detail. A software interrupt is initiated using the following statements:

INT number

In 8086, this interrupt instruction is processing using the **interrupt vector table (IVT)**. The IVT is located in the first 1K bytes of memory, and has a total of 256 entities, each of 4 bytes. An entry in the interrupt vector table is identified by the number given in the interrupt instruction. 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 1 shows the processing of an interrupt.

Figure 1: 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 will be found at IVT at an address 40h. Similarly the entry of INT 3h will be placed at 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 location at a segment address F000h and an offset F065h.
- Step 3:** The CPU loads the CS register and the IP register, with this new address in the IVT, and transfers the control to that address, just like a far CALL, (discussed in the unit 4).
- Step 4:** IRET (interrupt return) causes the program to resume execution at the next instruction in the calling program.

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 character is most tedious to program. However, these tasks were greatly simplified by the convenient architecture of the 8086/8088. That

architecture provides for a pack of software interrupt vectors beginning at address 0000:0000.

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 most commonly used Interrupts for Input /Output is called DOS function call. Let us discuss more about it in the next subsection:

2.5.2 DOS Function Calls (Using INT 21H)

INT 21H supports about 100 different functions. A function is identified by putting the function number in the AH register. For example, if we want to call function number 01, then we place this value in AH register first by using MOV instruction and then call INT 21H:

Some important DOS function calls are:

DOS Function Call	Purpose	Example
AH = 01H	For reading a single character from keyboard and echo it on monitor. The input value is put in AL register.	To get one character input in a variable in data segment you may include the following in the code segment: MOV AH,01 INT 21H MOV X, AL (Please note that interrupt call will return value in AL which is being transferred to variable of data segment X. X must be byte type).
AH = 02H	This function prints 8 bit data (normally ASCII) that is stored in DL register on the screen.	To print a character let say ‘?’ on the screen we may have to use following set of commands: MOV AH, 02H; MOV DL, ‘?’ INT 21H
AH = 08H	This is an input function for inputting one character. This is same as AH = 01H functions with the only difference that value does not get displayed on the screen.	Same example as 01 can be used only difference in this case would be that the input character wouldn’t get displayed MOV AH, 08H INT 21H MOV X, AL
AH = 09H	This program outputs a string whose offset is stored in DX register and that is terminated using a \$ character. One can print newline, tab character also.	To print a string “hello world” followed by a carriage return (control character) we may have to use the following assembly program segment.

Example of AH = 09H	CR EQU 0DH ; ASCII code of carriage return. DATA SEGMENT STRING DB 'HELLO WORLD', CR, '\$' DATA ENDS CODE SEGMENT : MOV AX, DATA MOV DS, AX MOV AH, 09H MOV DX, OFFSET STRING ; Store the offset of string in DX register. INT 21H	
AH = 0AH	For input of string up to 255 characters. The string is stored in a buffer.	Look in the examples given.
AH = 4CH	Return to DOS	

Some examples of Input

(i) Input a single ASCII character into BL register without echo on screen

CODE SEGMENT

```

MOV  AH, 08H ; Function 08H
INT  21H    ; The character input in AL is
MOV  BL, AL ; transfer to BL
:
```

CODE ENDS

(ii) Input a Single Digit for example (0,1, 2, 3, 4, 5, 6, 7, 8, 9)

CODE SEGMENT

```

...
; Read a single digit in BL register with echo. No error check in the Program
MOV  AH, 01H

INT  21H
; Assuming that the value entered is digit, then its ASCII will be stored in AL.
; Suppose the key pressed is 1 then ASCII '31' is stored in the AL. To get the
; digit 1 in AL subtract the ASCII value '0' from the AL register.
; Here it store 0 as ASCII 30,
; 1 as 31, 2 as 32.....9 as 39
; to store 1 in memory subtract 30 to get 31 - 30 = 1
MOV  BL, AL
SUB  BL, '0' ; '0' is digit 0 ASCII
; OR
SUB  BL, 30H
; Now BL contain the single digit 0 to 9
; The only code missing here is to check whether the input is in the specific
; range.
...
CODE ENDS.
```

(iii) Input numbers like (10, 11.....99)

```

; If we want to store 39, it is actually 30 + 9
; and it is 3 × 10 + 9
; to input this value through keyboard, first we input the tenth digit e.g., 3 and
```

```

; then type 9
    MOV  AH, 08H
    INT  21H
    MOV  BL, AL ; If we have input 39 then, BL will first have character
; 3, we can convert it to 3 using previous logic that is  $33 - 30 = 3$ .
    SUB  BL, '0'
    MUL  BL, AH      ; To get 30 Multiply it by 10.
                    ; Now BL Store 30
                    ; Input another digit from keyboard

    MOV  AH, 08H
    INT  21H;
    MOV  DL, AL      ; Store AL in DL
    SUB  DL, '0'      ; (39 - 30) = 9.
; Now BL contains the value: 30 and DL has the value 9 add them and get the
; required numbers.
    ADD  BL, DL
; Now BL store 39. We have 2 digit value in BL.

```

Let us try to summarize these segments as:

CODE SEGMENT

```

; Set DS register
    MOV  AX, DATA    ; } boiler plate code to set the DS register so that the
    MOV  DS, AX        ; } program can access the data segment.

; read first digit from keyboard
    MOV  AH, 08
    INT  21H
    MOV  BL, AL
    SUB  BL, '0'
    MUL  BL, 10H
; read second digit from keyboard
    MOV  AH, 08H
    INT  21H
    MOV  DL, AL
    SUB  DL, '0'
; DL = 9 AND BL = 30
    SUM  BL, DL
; now BL store 39
CODE ENDS.

```

Note: Boilerplate code is the code that is present more or less in the same form in every assembly language program.

Strings Input

CODE SEGMENT

```

...
    MOV  AH, 0AH ; Move 04 to AH register
    MOV  DX, BUFF ; BUFF must be defined in data segment.
    INT  21H
.....
CODE ENDS
DATA SEGMENT
    BUFF DB 50 ; max length of string,
                ; including CR, 50 characters
        DB ? ; actual length of string not known at present
        DB 50 DUP(0) ; buffer having 0 values
DATA ENDS.

```


Explanation

The above DATA segment creates an input buffer BUFF of maximum 50 characters. On input of data 'JAIN' followed by enter data would be stored as:

50	4	J	A	I	N	#
----	---	---	---	---	---	---

Examples of Display on Video Monitor

(1) Displaying a single character

```
; display contents of BL register (assume that it has a single character)
MOV AH, 02H
MOV DL, BL.
INT 21H.
```

Here data from BL is moved to DL and then data display on monitor function is called which displays the contents of DL register.

(2) Displaying a single digit (0 to 9)

Assume that a value 5 is stored in BL register, then to output BL as ASCII value add character '0' to it

```
ADD BL, '0'
MOV AH, 02H
MOV DL, BL
INT 21H
```

(3) Displaying a number (10 to 99)

Assuming that the two digit number 59 is stored as number 5 in BH and number 9 in BL, to convert them to equivalent ASCII we will add '0' to each of them.

```
ADD BH, '0'
ADD BL, '0'
MOV AH, 02H
MOV DL, BH
INT 21H
MOV DL, BL
INT 21H
```

(4) Displaying a string

```
MOV AH, 09H
MOV DX, OFFSET BUFF
INT 21H
```

Here data in input buffer stored in data segment is going to be displayed on the monitor.

A complete program:

Input a letter from keyboard and respond. "The letter you typed is ____".

```
CODE SEGMENT
;      set the DS register
        MOV AX, DATA
        MOV DS, AX
;      Read Keyboard
        MOV AH, 08H
        INT 21H
;      Save input
        MOV BL, AL
;      Display first part of Message
        MOV AH, 09H
        MOV DX, OFFSET MESSAGE
        INT 21H
;      Display character of BL register
        MOV AH, 02H
        MOV DL, BL
        INT 21H
;      Exit to DOS
        MOV AX, 4C00H
        INT 21H
CODE ENDS

DATA SEGMENT
        MESSAGE DB "The letter you typed is $"
DATA ENDS
END.
```

2.6 THE TYPES OF ASSEMBLY PROGRAMS

Assembly language programs can be written in two ways:

COM Program: Having all the segments as part of one segment

EXE Program: which have more than one segment.

Let us look into brief details of these programs.

2.6.1 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. The program code begins at an offset 100h, the first 1K locations being occupied by the IVT.

A COM program keeps its code, data, and stack segments within the same segment. Since the offsets in a physical segment can be of 16 bits, therefore the size of COM program is limited to $2^{16} = 64\text{K}$ which includes code, data and stack. The following program shows a COM program:

; Title add two numbers and store the result and carry in memory variables.
; name of the segment in this program is chosen to be CSEG

```
CSEG SEGMENT
        ASSUME CS:CSEG, DS:CSEG, SS:CSEG
        ORG    100h
START: MOV AX, CSEG      ; Initialise data segment
        MOV DS, AX      ; register using AX
        MOV AL, NUM1    ; Take the first number in AL
```

```
ADD AL, NUM2      ; Add the 2nd number to it
MOV RESULT, AL    ; Store the result in location RESULT
RCL AL, 01        ; Rotate carry into LSB
AND AL, 00000001B ; Mask out all but LSB
MOV CARRY, AL     ; Store the carry result
MOV AX, 4C00h
INT 21h
NUM1 DB 15h      ; First number stored here
NUM2 DB 20h      ; Second number stored here
RESULT DB ?      ; Put sum here
CARRY DB ?       ; Put any carry here
CSEG ENDS
END START
```

These programs are stored on a disk with an extension .com. A COM program requires less space on disk rather than 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.

2.6.2 EXE Programs

An EXE program is stored on disk with extension .exe. EXE programs are longer than the 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 calculate the addresses of segments and other components. We will not go into such details in this unit.

The load module of EXE program consists of up to 64K segments, although at the most only four segments may be active at any time. The segments may be of variable size, with maximum size being 64K.

We will write only EXE programs for the following reasons:

- EXE programs are better suited for debugging.
- EXE-format assembler programs are more easily converted into subroutines for high-level languages.
- EXE programs are more easily relocatable. Because, there is no ORG statement, forcing the program to be loaded from a specific 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 equivalent EXE program for the COM program is:

```
; ABSTRACT      this program adds 2 8-bit numbers in the memory locations
;              NUM1 and NUM2. The result is stored in the
;              memory location RESULT. If there was a carry
;              from the addition it will be stored as 0000 0001 in
;              the location CARRY
; REGISTERS     Uses CS, DS, AX
DATA SEGMENT
NUM1 DB 15h     ; First number
NUM2 DB 20h     ; Second number
```

```

        RESULT DB    ?        ; Put sum here
        CARRY  DB    ?        ; Put any carry here
DATA ENDS
CODE SEGMENT
        ASSUME CS:CODE, DS:DATA
START:MOV AX, DATA    ; Initialise data segment
        MOV DS, AX      ; register using AX
        MOV AL, NUM1     ; Bring the first number in AL
        ADD AL, NUM2     ; Add the 2nd number to AL
        MOV RESULT, AL   ; Store the result
        RCL AL, 01       ; Rotate carry into Least Significant Bit (LSB)
        AND AL, 00000001B ; Mask out all but LSB
        MOV CARRY, AL    ; Store the carry
        MOV AX, 4C00h    ; Terminate to DOS
        INT 21h
CODE ENDS
        END START

```

2.7 HOW TO WRITE GOOD ASSEMBLY PROGRAMS

Now that we have seen all the details of assembly language programming, let us discuss the art of writing assembly programs in brief.

Preparation of writing the program

1. Write an algorithm for your program closer to assembly language. For example, the algorithm for preceding program would be:


```

get NUM1
add NUM2
put sum into memory at RESULT
position carry bit in LSB of byte
                mask off upper seven bits
                store the result in the CARRY location.

```
2. Specify the input and output required.


```

input required   - two 8-bit numbers
output required  - an 8-bit result and a 8-bit carry in memory.

```
3. Study the instruction set carefully. This step helps in specifying the available instructions and their format and constraints. For example, the segment registers cannot be directly initialized by a memory variable. Instead we have to first move the offset for segment into a register, and then move the contents of register to the segment register.

You can exit to DOS, by using interrupt routine 21h, with function 4Ch, placed in AH register.

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 do not affect the size of the program.

After the program development, you may assemble it using an assembler and correct it for errors, finally creating exe file for execution.

Check Your Progress 2

State True or False

	T	F
1. For input/ output on Intel 8086/8088 machine running on DOS require special routines to be written by the assembly programmers.		<input type="checkbox"/>
2. Intel 8086 processor recognises only the software interrupts.		<input type="checkbox"/>
3. INT instruction in effect calls a subroutine, which is identified by a number.		<input type="checkbox"/>
4. Interrupt vector table IVT stores the interrupt handling programs.		<input type="checkbox"/>
5. INT 21H is a DOS function call.		<input type="checkbox"/>
6. INT 21H will output a character on the monitor if AH register contains 02.		<input type="checkbox"/>
7. String input and output can be achieved using INT 21H with function number 09h and 0Ah respectively.		<input type="checkbox"/>
8. To perform final exit to DOS we must use function 4CH with the INT 21H.		<input type="checkbox"/>
9. Notepad is an editor package.		<input type="checkbox"/>
10. Linking is required to link several segments of a single assembly program.		<input type="checkbox"/>
11. Debugger helps in removing the syntax errors of a program.		<input type="checkbox"/>
12. COM program is loaded at the 0 th location in the memory.		<input type="checkbox"/>
13. The size of COM program should not exceed 64K.		<input type="checkbox"/>
14. A COM program is longer than an EXE program.		<input type="checkbox"/>
15. STACK of a COM program is kept at the end of the occupied segment by the program.		<input type="checkbox"/>
16. EXE program contains a header module, which is used by DOS for calculating segment addresses.		<input type="checkbox"/>
17. EXE program cannot be easily debugged in comparison to COM programs.		<input type="checkbox"/>
18. EXE programs are more easily relocatable than COM programs.		<input type="checkbox"/>

2.8 SUMMARY

We summarize the complete discussion in the following flow chart.

2.9 SOLUTIONS/ ANSWERS

Check Your Progress 1

1.
 - (a) It helps in better understanding of computer architecture and work in 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 identifier 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

1. False
2. False
3. True
4. False
5. True
6. True
7. True
8. True
9. True
10. False
11. False
12. False
13. True
14. False
15. True
16. True
17. False
18. True

2.10 FURTHER READINGS

1. Yu-Cheng Lin, Genn. A. Gibson, “*Microcomputer System the 8086/8088 Family*” 2nd Edition, PHI.
2. Peter Abel, “*IBM PC Assembly Language and Programming*”, 5th Edition, PHI.
3. Douglas, V. Hall, “*Microprocessors and Interfacing*”, 2nd 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 3 ASSEMBLY LANGUAGE PROGRAMMING (PART – I)

Structure	Page No.
3.0 Introduction	57
3.1 Objectives	57
3.2 Simple Assembly Programs	57
3.2.1 Data Transfer	
3.2.2 Simple Arithmetic Application	
3.2.3 Application Using Shift Operations	
3.2.4 Larger of the Two Numbers	
3.3 Programming With Loops and Comparisons	63
3.3.1 Simple Program Loops	
3.3.2 Find the Largest and the Smallest Array Values	
3.3.3 Character Coded Data	
3.3.4 Code Conversion	
3.4 Programming for Arithmetic and String Operations	69
3.4.1 String Processing	
3.4.2 Some More Arithmetic Problems	
3.5 Summary	75
3.6 Solutions/ Answers	75

3.0 INTRODUCTION

After discussing a few essential directives, program developmental tools and simple programs, let us discuss more about assembly language programs. In this unit, we will start our discussions with simple assembly programs, which fulfil simple tasks such as data transfer, arithmetic operations, and shift operations. A key example here will be about finding the larger of two numbers. Thereafter, we will discuss more complex programs showing how loops and various comparisons are used to implement tasks like code conversion, coding characters, finding largest in array etc. Finally, we will discuss more complex arithmetic and string operations. You must refer to further readings for more discussions on these programming concepts.

3.1 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; and
- write simple assembly programs for implementing arrays.

3.2 SIMPLE ASSEMBLY PROGRAMS

As part of this unit, we will discuss writing assembly language programs. We shall start with very simple programs, and later graduate to more complex ones.

3.2.1 Data Transfer

Two most basic data transfer instructions in the 8086 microprocessor are MOV and XCHG. Let us give examples of the use of these instructions.

; Program 1: This program shows the difference of MOV and XCHG instructions:

```
DATA SEGMENT
    VAL DB 5678H ; initialize variable VAL
DATA ENDS

CODE SEGMENT
    ASSUME CS: CODE, DS: DATA
MAINP: MOV AX, 1234H ; AH=12 & AL=34
        XCHG AH, AL ; AH=34 & AL=12
        MOV AX, 1234H ; AH=12 & AL=34
        MOV BX, VAL ; BH=56 & BL=78
        XCHG AX, BX ; AX=5678 & BX=1234
        XCHG AH, BL ; AH=34, AL=78, BH=12, & BL=56
        MOV AX, 4C00H ; Halt using INT 21h
        INT 21H
CODE ENDS
END MAINP
```

Discussion:

Just keep on changing values as desired in the program.

; Program 2: Program for interchanging the values of two Memory locations

; input: Two memory variables of same size: 8-bit for this program

```
DATA SEGMENT
    VALUE1 DB 0Ah ; Variables
    VALUE2 DB 14h
DATA ENDS
CODE SEGMENT
    ASSUME CS:CODE, DS:DATA
    MOV AX, DATA ; Initialise data segments
    MOV DS, AX ; using AX
    MOV AL, VALUE1 ; Load Value1 into AL
    XCHG VALUE2, AL ; exchange AL with Value2.
    MOV VALUE1, AL ; Store A1 in Value1
    INT 21h ; Return to Operating system
CODE ENDS
END
```

Discussion:

The question is why cannot we simply use XCHG instruction with two memory variables as operand? To answer the question let us look into some of constraints for the MOV & XCHG instructions:

The MOV instruction has the following constraints and operands:

- 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, VALUE1, copies the VALUE1 that is 0Ah in the AL register:

AX: 00 14

0A (VALUE1)

0A (VALUE2)

AX : 00 14 \longrightarrow 14 (VALUE1)
 0A (VALUE2)

the carry bit into the AH register such that the AX(AH:AL) reflects the added value. 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 add, 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 average, we divide the sum given in AX by 2. A more general program would require positive and negative numbers. 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.

3.2.3 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: Convert the ASCII code to its BCD equivalent. This can be done by simply replacing the bits in the upper four bits of the byte by four zeros. For example, the ASCII '1' is 32h = 0011 0010B. By making the upper four bits as 0 we get 0000 0010 which is 2 in BCD. The number 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 59 is 00000101 00001001, that is, 05 09. The packed BCD will be 0101 1001, that is 59.

The algorithm to convert two ASCII digits to packed BCD can be stated as:

Convert first ASCII digit to unpacked BCD.

Convert the second ASCII digit to unpacked BCD.

Decimal	ASCII	BCD
5	00110101	00000101
9	00111001	00001001

Move first BCD to upper four positions in byte.

0101 0000	Using Rotate Instructions
-----------	---------------------------

Pack two BCD bits in one byte.

Pack	0101 0000	Using OR
	0000 1001	
	0101 1001	

;The assembly language program for the above can be written in the following manner.

; ABSTRACT

Program produces a packed BCD byte from 2 ASCII
; encoded digits. Assume the number as 59.

; The first ASCII digit (5) is loaded in BL.

; The second ASCII digit (9) is loaded in AL.

; The result (packed BCD) is left in AL.

```

; REGISTERS      ; Uses CS, AL, BL, CL
; PORTS          ; None used
CODE            SEGMENT
                ASSUME     CS:CODE
START:          MOV  BL,    '5'    ; Load first ASCII digit in BL
                MOV  AL,    '9'    ; Load second ASCII digit in AL
                AND  BL,    0Fh    ; Mask upper 4 bits of first digit
                AND  AL,    0Fh    ; Mask upper 4 bits of second digit
                MOV  CL,    04h    ; Load CL for 4 rotates
                ROL  BL,     CL    ; Rotate BL 4 bit positions
                OR   AL,     BL    ; Combine nibbles, result in AL contains 59
                                ; as packed BCD

CODE            ENDS
                END        START

```

Discussion:

8086 does not have any instruction to swap upper and lower four bits in a byte, therefore we need to use the rotate instructions that too by 4 times. Out of the two rotate instructions, ROL and RCL, 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.

Let us now look at a program that uses RCL instructions. This will make the difference between the instructions clear.

; Program 5: Add a byte number from one memory location to a byte from the next memory location and put the sum in the third memory location. Also, save the carry flag in the least significant bit of the fourth memory location.

```

; ABSTRACT      : This program adds 2-8-bit words in the memory locations
;              : NUM1 and NUM2. The result is stored in the memory
;              : location RESULT. The carry bit, if any will be stored as
;              : 0000 0001 in the location CARRY

```

```

; ALGORITHM:
;      get NUM1
;      add NUM2 in it
;      put sum into memory location RESULT
;      rotate carry in LSB of byte
;      mask off upper seven bits of byte
;      store the result in the CARRY location.
;

```

```

; PORTS         : None used
; PROCEDURES    : None used
; REGISTERS     : Uses CS, DS, AX
;

```

```

DATA            SEGMENT
                NUM1      DB    25h    ; First number
                NUM2      DB    80h    ; Second number
                RESULT     DB    ?      ; Put sum here
                CARRY      DB
DATA            ENDS
CODE            SEGMENT
                ASSUME CS:CODE, DS:DATA
START:MOV AX, DATA    ; Initialise data segment
                MOV DS, AX    ; register using AX
                MOV AL, NUM1   ; Load the first number in AL
                ADD AL, NUM2   ; Add 2nd number in AL

```

```
MOV RESULT, AL           ; Store the result
RCL AL, 01                ; Rotate carry into LSB
AND AL, 00000001B        ; Mask out all but LSB
MOV CARRY, AL             ; Store the carry result
MOV AH, 4CH
INT 21H
CODE    ENDS
END     START
```

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 the carry, now in AL.

In a similar manner we can also write applications using other shift instructions.

3.2.4 Larger of the Two Numbers

How are the comparisons done in 8086 assembly language? There exists a compare instruction CMP. However, this instruction only sets the flags on comparing two operands (both 8 bits or 16 bits). Compare instruction just subtracts the value of source from destination without storing the result, but setting the flag during the process. Generally only three comparisons are more important. These are:

Result of comparison	Flag(s) affected
Destination < source	Carry flag = 1
Destination = source	Zero flag = 1
Destination > source	Carry = 0, Zero = 0

Let's look at three examples that show how the flags are set when the numbers are compared. In example 1 BL is less than 10, so the carry flag is set. In example 2, the zero flag is set because both operands are equal. In example 3, the destination (BX) is greater than the source, so both the zero and the carry flags are clear.

Example 1:

```
MOV BL, 02h
CMP BL, 10h           ; Carry flag = 1
```

Example 2:

```
MOV AX, F0F0h
MOV DX, F0F0h
CMP AX, DX           ; Zero flag = 1
```

Example 3:

```
MOV BX, 200H
CMP BX, 0           ; Zero and Carry flags = 0
```

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 3.2.2 we can change instruction DIV BL with a shift. ☐
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. ☐

3.3 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.

3.3.1 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 above the control of the program will directly transfer to the label THERE if the value stores 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 above code is not efficient, but suggest that there are many ways through which a conditional jump can be implemented. Select the most optimum way.

Example 6:

```
CMP DX, 00      ; checks if DX is zero.
JE  Label1      ; if yes, jump to Label1 i.e. if ZF=1
```

Label1:---- ; 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  Lab1        ; carry flag = 1 then jump to Lab1
```

Lab1: ----- ; control comes here if condition is satisfied

LOOPING

Program 6: Assume a constant inflation factor that is added to a series of prices stored in the memory. The program copies the new price over the old price. It is assumed that price data is available in BCD form.

; The algorithm:

```
;Repeat
;   Read a price from the array
;   Add inflation factor
;   Adjust result to correct BCD
;   Put result back in array
;   Until all prices are inflated
```

; REGISTERS: Uses DS, CS, AX, BX, CX

; PORTS : Not used

ARRAYS SEGMENT

PRICE DB 36h, 55h, 27h, 42h, 38h, 41h, 29h, 39h

ARRAYS ENDS

CODE SEGMENT

ASSUME CS:CODE, DS:ARRAYS

START: MOV AX, ARRAYS ; Initialize data segment

MOV DS, AX ; register using AX

LEA BX, PRICES ; initialize pointer to base of array

MOV CX, 0008h ; Initialise counter to 8 as array have 8
; values.

DO_NEXT: MOV AL, [BX] ; Copy a price to AL. BX is addressed in
; indirect mode.

ADD AL, 0Ah ; Add inflation factor

DAA ; Make sure that result is BCD

MOV [BX], AL ; Copy result back to the memory

INC BX ; increment BX to make it point to next price

DEC CX ; Decrement counter register

JNZ DO_NEXT ; If not last, (last would be when CX will
; become 0) Loop back to DO_NEXT

MOV AH, 4CH ; Return to DOS

INT 21H

CODE ENDS

END START

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. [BX] is an indirection through BX and contains the value stored at that element of array. 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.

Let us demonstrate the use of LOOP instruction, with the help of following program:

; Program 7: This following program prints the alphabets (A-Z)

; Register used : AX, CX, DX

```
CODE SEGMENT
    ASSUME : CS:CODE.
MAINP:  MOV  CX, 1AH    ; 26 in decimal = 1A in hexadecimal Counter.
        MOV  DL, 41H    ; Loading DL with ASCII hexadecimal of A.
NEXTC:  MOV  AH, 02H    ; display result character in DL
        INT  21H        ; DOS interrupt
        INC  DL         ; Increment DL for next char
        LOOP NEXTC      ; Repeat until CX=0.(loop automatically decrements
                        ; CX and checks whether it is zero or not)
        MOV  AX, 4C00H ; Exit DOS
        INT  21H        ; DOS Call
CODE ENDS
END MAINP
```

Let us now discuss a slightly more complex looping program.

; Program 8: This program compares a pair of characters entered through keyboard.

; Registers used: AX, BX, CX, DX

```
DATA SEGMENT
    XX DB ?
    YY DB ?
DATA ENDS

CODE SEGMENT
    ASSUME  CS: CODE, DS: DATA
MAINP:     MOV  AX, DATA    ; initialize data
           MOV  DS, AX      ; segment using AX
           MOV  CX, 03H     ; set counter to 3.
NEXTP:     MOV  AH, 01H     ; Waiting for user to enter a char.
           INT  21H
           MOV  XX, AL      ; store the 1st input character in XX
           MOV  AH, 01H     ; waiting for user to enter second
           INT  21H         ; character.
           MOV  YY, AL      ; store the character to YY
           MOV  BH, XX      ; load first character in BH
           MOV  BL, YY      ; load second character in BL
           CMP  BH, BL      ; compare the characters
           JNE  NOT_EQUAL   ;
           ;
```



```

EQUAL:      MOV AH, 02H      ; if characters are equal then control
              MOV DL, 'Y'    ; will execute this block and
              INT 21H        ; display 'Y'
              JMP CONTINUE   ; Jump to continue loop.

NOT_EQUAL:  MOV AH, 02H      ; if characters are not equal then
                              control
              MOV DL, 'N'    ; will execute this block and
              INT 21H        ; display 'N'

CONTINUE :   LOOP NEXT P     ; Get the next character
              MOV AH, 4C H    ; Exit to DOS
              INT 21 H

CODE ENDS
END MAINP

```

Discussion:

This program will be executed, at least 3 times.

3.3.2 Find the Largest and the Smallest Array Values

Let us now put together whatever we have done in the preceding sections and write down a program to find the largest and the smallest numbers from a given array. This program uses the JGE (jump greater than or equal to) instruction, because we have assumed the array values as signed. We have not used the JAE instruction, which works correctly for unsigned numbers.

Program 9: Initialise the **smallest** and the **largest** 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 **smallest** and the **largest** variables are changed with the new values respectively. Let us use register DI to point the current array value and LOOP instruction for looping.

```

DATA        SEGMENT
              ARRAY      DW    -1, 2000, -4000, 32767, 500, 0
              LARGE      DW    ?
              SMALL      DW    ?
DATA        ENDS

CODE        SEGMENT
MOV  AX, DATA
MOV  DS, AX      ; Initialize DS
MOV  DI, OFFSET ARRAY ; DI points to the array
MOV  AX, [DI]    ; AX contains the first element
MOV  DX, AX      ; initialize large in DX register
MOV  BX, AX      ; initialize small in BX register
MOV  CX, 6       ; initialize loop counter
A1:  MOV  AX, [DI] ; get next array value
      CMP  AX, BX  ; Is the new value smaller?
      JGE  A2      ; If greater then (not smaller) jump to
                      ; A2, to check larger than large in DX
      MOV  BX, AX  ; Otherwise it is smaller so move it to
                      ; the smallest value (BX register)
      JMP  A3      ; as it is small, thus no need
                      ; to compare it with the large so jump
                      ; to A3 to continue or terminate loop.

```

```

A2:      CMP    AX, DX      ; [DI] = large
        JLE     A3          ; if less than it implies not large so
                               ; jump to A3
        MOV     DX, AX      ; to continue or terminate
                               ; otherwise it is larger value, so move
A3:      ADD     DI, 2       ; it to DX that store the large value
        LOOP    A1          ; DI now points to next number
        MOV     LARGE, DX   ; repeat the loop until CX = 0
        MOV     SMALL, BX   ; move the large and small in the
                               ; memory locations
        MOV     AX, 4C00h
        INT     21h         ; halt, return to DOS
CODE     ENDS

```

Discussion:

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.

3.3.3 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, we may enter the numbers as:

Enter first number	1234	
Enter second number	3210	
	4444	
The sum is		

As each digit is input, we 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 entered as:

31	32	33	34	hexadecimal storage
1	2	3	4	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. Let us use the conversion operation to perform these calculations here.

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 of 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. The instruction DAA and DAS has already been explained in unit 1.

3.3.4 Code Conversion

The conversion of data from one form to another is needed. Therefore, in this section we will discuss an example, for converting a hexadecimal digit obtained in ASCII form to binary form. Many ASCII to BCD and other conversion examples have been given earlier in unit 2.

Program 10:

```
; This program converts an ASCII input to equivalent hex digit that it represents.
; Thus, valid ASCII digits are 0 to 9, A to F and the program assumes that the
; ASCII digit is read from a location in memory called ASCII. The hex 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
; IF number <30h THEN error
; ELSE
; IF number <3Ah THEN Subtract 30h (it's a number 0-9)
; ELSE (number is >39h)
; IF number <41h THEN error (number in range 3Ah-40h which is not a valid
; A-F character range)
; ELSE
; IF number <47h THEN Subtract 37h for letter A-F 41-46 (Please note
; that 41h - 37h = Ah)
; ELSE ERROR
;
; PORTS : None used
; PROCEDURES : None
; REGISTERS : Uses CS, DS, AX,
;
DATA SEGMENT
ASCII DB 39h ; Any experimental data
DATA ENDS
CODE SEGMENT
ASSUME CS:CODE, DS:DATA
START: MOV AX, DATA ; initialise data segment
MOV DS, AX ; Register using AX
MOV AL, ASCII ; Get the ASCII digits of the number
; start the conversion
CMP AL, 30h ; If the ASCII digit is below 30h then it is not
JB ERROR ; a proper Hex digit
CMP AL, 3Ah ; compare it to 3Ah
JB NUMBER ; If greater then possibly a letter between A-F
CMP AL, 41h ; This step will be done if equal to or above
; 3Ah
JB ERROR ; Between 3Ah and 40h is error
CMP AL, 46h
JA ERROR ; The ASCII is out of 0-9 and A-F range
SUB AL, 37h ; It's a letter in the range A-F so convert
JMP CONVERTED
NUMBER: SUB AL, 30h ; it is a number in the range 0-9 so convert
JMP CONVERTED
```

```

ERROR:      MOV  AL, 0FFh      ; You can also display some message here
CONVERTED: MOV  AX, 4C00h
            INT   21h          ; the hex result is in AL
CODE        ENDS
            END   START

```

Discussions:

The above program demonstrates a single hex digit represented by an 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

- Write the code sequence in assembly for performing following operation:

$$Z = ((A - B) / 10 * C) * * 2$$

.....

- Write an assembly code sequence for adding an array of binary numbers.

.....

- 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 packed BCD addition for such numbers? Why?

.....

- How can we implement nested loops, for example,

```

For (i = 1 to 10, step 1)
    { for (j = 1 to, step 1)
      add 1 to AX}

```

in assembly language?

.....

3.4 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 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. Let us discuss this in more detail in the next subsection:

3.4.1 String Processing

Let us write a program for comparing two strings. Consider the following piece of code, which has been written in C to compare two strings. Let us assume that 'str1' and 'str2' are two strings, initialised by some values and 'ind' is the index for these character strings:

```
for (ind = 0; ( (ind < 9) and (str1[ind] == str2[ind]) ), ind ++)
```

The intermediate code in assembly language generated by a non-optimising compiler for the above piece may look like:

```

L3:      MOV     IND, 00          ; ind := 0
        CMP     IND, 08          ; ind < 9
        JG      L1              ; not so; skip
        LEA     AX, STR1         ; offset of str1 in AX register
        MOV     BX, IND          ; it uses a register for indexing into
                                ; the array
        LEA     CX, STR2         ; str2 in CX
        MOV     DL, BYTE PTR CX[BX]
        CMP     DL, BYTE PTR AX[BX] ; str1[ind] = str2[ind]
        JNE     L1              ; no, skip
        MOV     IND, BX
        ADD     IND, 01
L2:      JMP     L3              ; loop back
L1:

```

What we find in the above code: a large code that could have been improved further, if the 8086 string instructions would have been used.

; Program 11: Matching two strings of same length stored in memory locations.

; REGISTERS : Uses CS, DS, ES, AX, DX, CX, SI, DI

```

DATA     SEGMENT
PASSWORD DB 'FAILSAFE' ; source string
DESTSTR  DB 'FEELSAFE' ; destination string
MESSAGE  DB 'String are equal $'
DATA     ENDS
CODE     SEGMENT
        ASSUME CS:CODE, DS:DATA, ES:DATA
        MOV     AX, DATA
        MOV     DS, AX          ; Initialise data segment register
        MOV     ES, AX          ; Initialise extra segment register
; as destination string is considered to be in extra segment. Please note that ES is also
; initialised to the same segment as of DS.
        LEA     SI, PASSWORD    ; Load source pointer
        LEA     DI, DESTSTR     ; Load destination pointer
        MOV     CX, 08          ; Load counter with string length
        CLD                    ; Clear direction flag so that comparison is
                                ; done in forward direction.

        REPE    CMPSB           ; Compare the two string byte by byte
        JNE     NOTEQUAL        ; If not equal, jump to NOTEQUAL
        MOV     AH, 09          ; else display message
        MOV     DX, OFFSET MESSAGE ;
        INT     21h             ; display the message
NOTEQUAL:MOV     AX, 4C00h       ; interrupt function to halt
        INT     21h
CODE     ENDS
        END

```

Discussion:

In the above program 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 next word. 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.

Similarly, you can write efficient programs for moving one string to another, using MOVS, and scanning a string for a character using SCAS.

3.4.2 Some More Arithmetic Problems

Let us now take up some more practical arithmetic problems.

Use of delay loops

A very useful application of assembly is to produce delay loops. Such loops are used for waiting for some time prior to execution of next instruction.

But how to find the time for the delay? The rate at which the instructions are executed is determined by the clock frequency. Each instruction takes a certain number of clock cycles to execute. This, multiplied by the clock frequency of the microprocessor, gives the actual time of execution of a instruction. For example, MOV instruction takes four clock cycles. This instruction when run on a microprocessor with a 4Mhz clock takes 4/4, i.e. 1 microsecond. NOP is an instruction that is used to produce the delay, without affecting the actual running of the program.

Time delay of 1 ms on a microprocessor having a clock frequency of 5 MHz would require:

$$\begin{aligned} 1 \text{ clock cycle} &= \frac{1}{5\text{MHz}} \\ &= \frac{1}{5 \times 10^6} \text{ Seconds} \end{aligned}$$

Thus, a 1-millisecond delay will require:

$$\begin{aligned} &= \frac{1 \times 10^{-3}}{\left(\frac{1}{5 \times 10^6} \right)} \text{ clock cycles} \\ &= 5000 \text{ clock cycles.} \end{aligned}$$

The following program segment can be used to produce the delay, with the counter value correctly initialised.

```

MOV     CX, N           ; 4 clock cycles N will vary depending on
                        ; the amount of delay required

DELAY:  NOP             ; 3 cycles
        NOP             ; 3 cycles
        LOOP DELAY ; 17 or 5

```

LOOP instruction takes 17 clock cycles when the condition is true and 5 clock cycles otherwise. The condition will be true, 'N' number of times and false only once, when the control comes out of the loop.

To calculate 'N':

$$\begin{aligned} \text{Total clock cycles} &= \text{clock cycles for MOV} + N(2 \times \text{NOP clock} \\ &\quad \text{cycles} + 17) - 12 \text{ (when CX} = 0) \end{aligned}$$

$$5000 = 4 + N(6 + 17) - 12$$

$$N = 5000/23 = 218 = 0DAh$$

Therefore, the counter, CX, should be initialized by 0DAh, in order to get the delay of 1 millisecond.

Use of array in assembly

Let us write a program to add two 5-byte numbers stored in an array. For example, two numbers in hex can be:

	20	11	01	10	FF
	FF	40	30	20	10
1	1F	51	31	31	1F

Carry

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 12: Add two five-byte numbers using arrays

; ALGORITHM:

```

;      Make count = LEN
;      Clear the carry flag
;      Load address of NUM1
;      REPEAT
;          Put byte from NUM1 in accumulator
;          Add byte from NUM2 to accumulator + carry
;          Store result in SUM
;          Decrement count
;          Increment to next address
;      UNTIL count = 0
;      Rotate carry into LSB of accumulator
;      Mask all but LSB of accumulator
;      Store carry result, address pointer in correct position.
; PORTS      : None used
; PROCEDURES : None used
; REGISTERS  : Uses CS, DS, AX, CX, BX, DX

```

```

DATA      SEGMENT
NUM1      DB      0FFh, 10h, ,01h, ,11h, ,20h
NUM2      DB      10h, 20h, 30h, 40h, ,0FFh
SUM        DB      6DUP(0)
DATA      ENDS
LEN        EQU     05h      ; constant for length of the array

```

```

CODE      SEGMENT
ASSUME CS:CODE, DS:DATA
START:    MOV     AX, DATA    ; initialise data segment
          MOV     DS, AX      ; using AX register
          MOV     SI, 00       ; load displacement of 1st number.
                                   ; SI is being used as index register
          MOV     CX, 0000     ; clear counter
          MOV     CL, LEN      ; set up count to designed length
          CLC                ; clear carry. Ready for addition
AGAIN:    MOV     AL, NUM1[SI] ; get a byte from NUM1
          ADC     AL, NUM2[SI] ; add to byte from NUM2 with carry

```

```

MOV     SUM[SI], AL    ; store in SUM array
INC     SI
LOOP    AGAIN          ; continue until no more bytes
RCL     AL, 01h        ; move carry into bit 0 of AL
AND     AL, 01h        ; mask all but the 0th bit of AL
MOV     SUM[SI], AL    ; put carry into 6th byte
FINISH: MOV     AX, 4C00h
INT     21h
CODE    ENDS
END     START

```

Program 13: A good example of code conversion: Write a program to convert a 4-digit BCD number into its binary equivalent. The BCD number is stored as a word in memory location called BCD. The result is to be stored in location HEX.
; ALGORITHM:

```

;      Let us assume the BCD number as 4567
;      Put the BCD number into 4, 16bit registers
;      Extract the first digit (4 in this case)
;      by masking out the other three digits. Since, its place value is 1000.
;      So Multiply by 3E8h (that is 1000 in hexadecimal) to get 4000 = 0FA0h
;      Extract the second digit (5)
;      by masking out the other three digits.
;      Multiply by 64h (100)
;      Add to first digit and get 4500 = 1194h
;      Extract the third digit (6)
;      by masking out the other three digits (0060)
;      Multiply by 0Ah (10)
;      Add to first and second digit to get 4560 = 11D0h
;      Extract the last digit (7)
;      by masking out the other three digits (0007)
;      Add the first, second, and third digit to get 4567 = 11D7h
; PORTS      : None used
; REGISTERS: Uses CS, DS, AX, CX, BX, DX

```

```

THOU    EQU      3E8h          ; 1000 = 3E8h
DATA    SEGMENT
        BCD      DW      4567h
        HEX      DW      ?      ; storage reserved for result
DATA    ENDS

```

```

CODE    SEGMENT
        ASSUME CS:CODE, DS:DATA
START:  MOV     AX, DATA      ; initialise data segment
        MOV     DS, AX        ; using AX register
        MOV     AX, BCD        ; get the BCD number AX = 4567
        MOV     BX, AX        ; copy number into BX; BX = 4567
        MOV     AL, AH        ; place for upper 2 digits in AX = 4545
        MOV     BH, BL        ; place for lower 2 digits in BX = 6767
        ; split up numbers so that we have one digit
        ; in each register
        MOV     CL, 04        ; bit count for rotate
        ROR     AH, CL        ; digit 1 (MSB) in lower four bits of AH.
        ; AX = 54 45
        ROR     BH, CL        ; digit 3 in lower four bits of BH.
        ; BX = 76 67
        AND     AX, 0F0FH      ; mask upper four bits of each digit.
        ; AX = 04 05

```



```
AND    BX, 0F0FH    ; BX = 06 07
MOV    CX, AX        ; copy AX into CX so that can use AX for
                     ; multiplication CX = 04 05
```

```
; CH contains digit 4 having place value 1000, CL contains digit 5
; having place value 100, BH contains digit 6 having place value 10 and
; BL contains digit 7 having unit place value.
; so obtain the number as CH × 1000 + CL × 100 + BH × 10 + BL
```

```
MOV    AX, 0000H    ; zero AH and AL
                     ; now multiply each number by its place
                     ; value
MOV    AL, CH        ; digit 1 to AL for multiply
MOV    DI, THOU      ; no immediate multiplication is allowed so
                     ; move thousand to DI
MOV    DI            ; digit 1 (4)*1000
                     ; result in DX and AX. Because BCD digit
                     ; will not be greater than 9999, the result will
                     ; be in AX only. AX = 4000

MOV    DH, 00H       ; zero DH
MOV    DL, BL        ; move BL to DL, so DL = 7
ADD    DX, AX        ; add AX; so DX = 4007
MOV    AX, 0064h     ; load value for 100 into AL
MOV    CL            ; multiply by digit 2 from CL
MUL    CL            ; multiply by digit 2 from CL
ADD    DX, AX        ; add to total in DX. DX now contains
                     ; (7 + 4000 + 500)

MOV    AX, 000Ah     ; load value of 10 into AL
MUL    BH            ; multiply by digit 3 in BH
ADD    DX, AX        ; add to total in DX; DX contains
                     ; (7 + 4000 + 500 +60)

MOV    HEX, DX       ; put result in HEX for return
MOV    AX, 4C00h
INT    21h
CODE   ENDS
END    START
```

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 NOP statement?
.....
.....
.....

3.5 SUMMARY

In this unit, we have covered some basic aspects of assembly language programming. We started with some elementary arithmetic problems, code conversion problems, various types of loops and graduated on to do string processing and slightly complex arithmetic. As part of good programming practice, we also noted some points that should be kept in mind while coding. Some of them are:

- An algorithm should always 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. You can even use a debugger to get a clear understanding of the instructions and addressing modes.
- 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 expect some registers to be initialised by some values before being executed, example, LOOP expects the counter value to be contained in CX register, string instructions expect DS:SI to be initialised 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, and depending on them some of the additional registers also to be initialised. So study them 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 we have also seen some kind of applications can be written and efficiently executed in assembly language. We justified this with string processing instructions; you will appreciate it more when you actually start doing the assembly language programming. You can now perform some simple exercises from the further readings.

In the next block, we take up more advanced assembly language programming, which also includes accessing interrupts of the machine.

3.6 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, 10 | ; move dividend to BX |
| IDIV | BX | ; divide |
| 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.

```
        MOV     CX, COUNT    ; put the number of elements of the array in
                                ; CX register
        MOV     AX, 0000h    ; zero SI and AX
        MOV     SI, AX
; add the elements of array in AX again and again
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
                                ; terminate when CX becomes zero.
        MOV     TOTAL, AX    ; store the results in TOTAL.
```

3. Yes, because the conversion efforts are less.
4. We 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 we 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, we can perform string processing very efficiently in 8086 assembly language.
2. Direction flag if clear will cause REPE statement to perform in forward direction. That is, in the given example the strings will be compared from first element to last.
3. It produces a delay of a desired clock time in the execution. This instruction is useful while development of program. A collection of these instructions can be used to fill up some space in the code segment, which can be changed with new code lines without disturbing the position of existing code. This is particularly used when a label is specified.

UNIT 4 ASSEMBLY LANGUAGE PROGRAMMING (PART-II)

Structure	Page No.
4.0 Introduction	77
4.1 Objectives	77
4.2 Use of Arrays in Assembly	77
4.3 Modular Programming	80
4.3.1 The stack	
4.3.2 FAR and NEAR Procedures	
4.3.3 Parameter Passing in Procedures	
4.3.4 External Procedures	
4.4 Interfacing Assembly Language Routines to High Level Language Programs	93
4.4.1 Simple Interfacing	
4.4.2 Interfacing Subroutines With Parameter Passing	
4.5 Interrupts	97
4.6 Device Drivers in Assembly	99
4.7 Summary	101
4.8 Solutions/ Answers	102

4.0 INTRODUCTION

In the previous units, we have discussed the instruction set, addressing modes, and other tools, which are needed to develop assembly language programs. We shall now use this knowledge in developing more advanced tools. We have divided this unit broadly into four sections. In the first section, we discuss the design of some simple data structures using the basic data types. Once the programs become lengthier, it is advisable to divide them into small modules, which can be easily written, tested and debugged. This leads to the concept of modular programming, and that is the topic of our second section in this unit. In the third section, we will discuss some techniques to interface assembly language programs to high level languages. We have explained the concepts using C and C ++ as they are two of the most popular high-level languages. In the fourth section we have designed some tools necessary for interfacing the microprocessor with external hardware modules.

4.1 OBJECTIVES

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

- implement simple data structures in assembly language;
 - write modular programs in assembly language;
 - interface assembly program to high level language program; and
 - analyse simple interrupt routines.
-

4.2 USE OF ARRAYS IN ASSEMBLY

An array is referencing 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 string processing, viz. SI and DI. In addition to this you can use any general purpose register also for indexing.

An important application of array is the tables that are used to store related information. For example, the names of all the students in the class, their CGPA, the list of all the books in the library, or even the list of people residing in a particular area can be stored in different tables. An important application of tables would be character translation. It can be used for data encryption, or translation from one data type to another. A critical factor for such kind of applications is the speed, which just happens to be a strength of assembly language. The instruction that is used for such kind of applications is XLAT.

Let us explain this instruction with the help of an example:

Example 1:

Let us assume a table of hexadecimal characters representing all 16 hexadecimal digits in table:

```
HEXA    DB    '0123456789ABCDEF'
```

The table contains the ASCII code of each hexadecimal digit:

Offset	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
Contents	30	31	32	33	34	35	36	37	38	39	41	42	43	44	45	46

(all value in hexadecimal)

If we place 0Ah in AL with the thought of converting it to ASCII, we need to set BX to the offset of HEXA, and invoke XLAT. You need not specify the table name with XLAT because it is implicitly passed by setting BX to the HEXA table offset. This instruction will do the following operations:

It will first add BX to AL, generating an effective address that points to the eleventh entry in the HEXA table.

The content of this entry is now moved to the AL register, that is, 41h is moved to AL.

In other words, XLAT sets AL to 41h because this value is located at HEXA table offset 0Ah. Please note that the 41h is the ASCII code for hex digit A. The following sequence of instructions would accomplished this:

```
MOV  AL, 0Ah           ; index value
MOV  BX, OFFSET HEXA   ; offset of the table HEXA
XLAT
```

The above tasks can be done without XLAT instruction but it will require a long series of instructions such as:

```
MOV    AL, 0Ah           ; index value
MOV    BX, OFFSET HEXA   ; offset of the table HEXA
PUSH    BX               ; save the offset
ADD     BL, AL            ; add index value to table
                        ; HEXA offset
MOV     AL, [BX]          ; retrieve the entry
POP     BX               ; restore BX
```

Let us use the instruction XLAT for data encoding. When you want to transfer a message through a telephone line, then such encoding may be a good way of preventing other users from reading it. Let us show a sample program for encoding.

PROGRAM 1:

; A program for encoding ASCII Alpha numerics.

; ALGORITHM:

; create the code table
; read an input string character by character
; translate it using code table
; output the strings

```
DATA    SEGMENT
        CODETABLE    DB 48 DUP (0)    ; no translation of first
                                ; 48 ASCII
                                DB '4590821367'    ; ASCII codes 48 –
                                ; 57 ≡ (30h – 39h)
                                DB 7 DUP (0)    ; no translation of
                                                these 7 characters
                                DB 'GVHZUSOBMIKPJCADLFTYEQNWXR'
                                DB 6 DUP (0)    ; no translation
                                DB 'gvhzusobmikpjcadlfteqnxwr'
                                DB 133 DUP (0)    ; no translation of remaining
                                                ; character

DATA    ENDS

CODE    SEGMENT
        MOV    AX, DATA
        MOV    DS, AX    ; initialize DS
        MOV    BX, OFFSET CODETABLE    ; point to lookup table

GETCHAR:
        MOV    AH, 06    ; console input no wait
        MOV    DL, 0FFh    ; specify input request
        INT    21h    ; call DOS
        JZ     QUIT    ; quit if no input is waiting
        MOV    DL, AL    ; save character in DL
        XLAT    CODETABLE    ; translate the character
        CMP    AL, 0    ; translatable?
        JE     PUTCHAR    ; no : write it as is.
        MOV    DL, AL    ; yes : move new character
                        ; to DL

PUTCHAR:
        MOV    AH, 02    ; write DL to output
        INT    21h
        JMP    GETCHAR    ; get another character

QUIT:    MOV    AX, 4C00h
        INT    21h

CODE    ENDS
END
```

Discussion:

The program above will code the data. For example, a line from an input file will be encoded:

A SECRET Message	(Read from an input file)
G TUHFUY Juttgou	(Encoded output)

The program above can be run using the following command line. If the program file name is coding.asm

coding infile > outfile

The infile is the input data file, and outfile is the output data file.
You can write more such applications using 8086 assembly tables.

Check Your Progress 1

1. Write a program to convert all upper case letters to lower case.
.....
.....
.....
2. **State True or False**

T	F
---	---

 - a. Table handling cannot be done without using XLAT instruction. ☐
 - b. In XLAT instruction AX register contains the address of the first entry of the table. ☐
 - c. In XLAT instruction the desired element value is returned in AL register. ☐

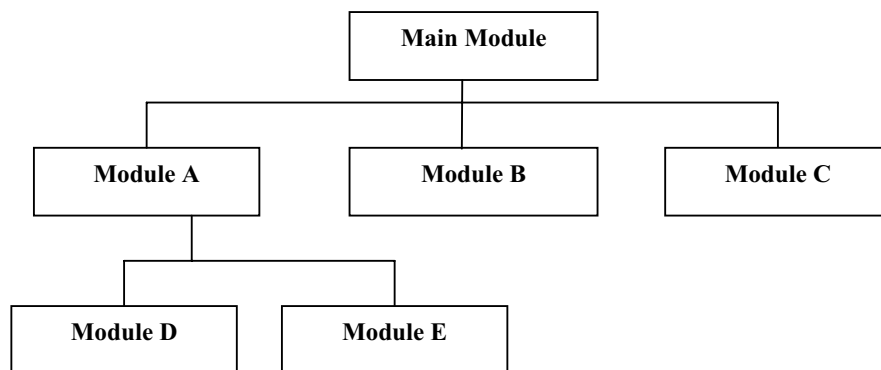
4.3 MODULAR PROGRAMMING

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 modularise 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 involved.
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 localised to a module.

A modular program can be represented using hierarchical diagram:



The advantages of modular programming are:

1. Smaller, easier modules to manage
2. Code repetition may be avoided by reusing modules.

You can divide a program into subroutines or procedures. You need to CALL the procedure whenever needed. A subroutine call transfers the control to subroutine instructions and brings the control back to calling program.

4.3.1 The Stack

A procedure call is supported by a stack. So let us discuss stack in assembly. Stacks are Last In First Out data structures, and are used for storing the return addresses of the procedures and for parameter passing and saving the return value.

In 8086 microprocessor a stack is created in the stack segment. The SS register stores the offset of stack segment and SP register stores the top of the stack. A value is pushed in to top of the stack or taken out (popped) from the top of the stack. The stack segment can be initialized as follows:

```
STACK_SEG 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          ; initialise stack segment
    LEA  SP, TOP         ; initialise stack pointer
CODE ENDS
    END
```

The directive STACK_SEG SEGMENT STACK declares the logical segment for the stack segment. DW 100 DUP(0) assigns actual size of the stack to 100 words. All locations of this stack are initialized to zero. The stacks are identified by the stack top and that is why the Label Top of Stack (TOS) has been selected. Please note that the stack in 8086 is a WORD stack. Stack facilities involve the use of indirect addressing through a special register, the stack pointer (SP). SP is automatically decremented as items are put on the stack and incremented as they are retrieved. Putting something on to stack is called a PUSH and taking it off is called a POP. The address of the last element pushed on to the stack is known as the top of the stack (TOS).

Name	Mnemonics	Description
Push onto the stack	PUSH SRC	$SP \leftarrow SP - 2$ SP+1 and SP location are assign the SRC
Pop from the stack	POP DST	DST is a assigned values stored at stack top $SP \leftarrow SP + 2$

4.3.2 Far and Near Procedures

Procedure provides the primary means of breaking the code in a program into modules. Procedures have one major disadvantage, that is, 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:

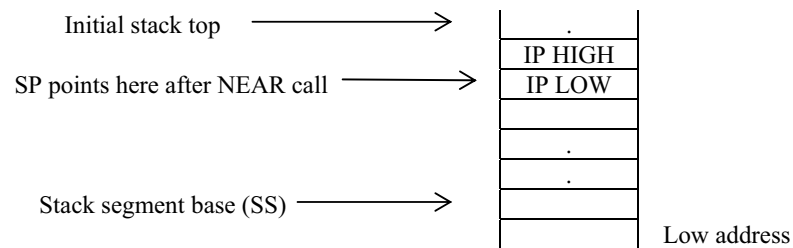
1. Unlike other branch instructions, a procedure call must save the address of the next instruction so that the return will be able to branch back to the proper place in the calling program.
2. The registers used by the procedures need to be stored before their contents are changed and then restored just before the procedure is finished.
3. A procedure must have a 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.

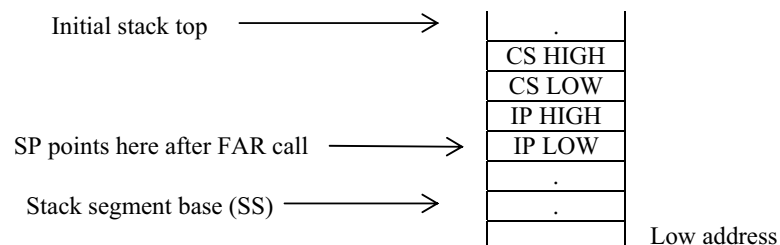
The CALL instruction not only branches to the indicated address, but also pushes the return address onto the stack. In addition, it also initialized IP with the address of the procedure. The RET instructions simply pops the return address from the stack. 8086 supports two kinds of procedure call. These are FAR and NEAR calls.

The NEAR procedure call is also known as Intrasegment 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. The IP can be stored on the stack as:



Please note the growth of stack is towards stack segment base. So stack becomes full on an offset 0000h. Also for push operation we decrement SP by 2 as stack is a word stack (word size in 8086 = 16 bits) while memory is byte organised 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 one 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.

Let us describe an example of procedure call using NEAR procedure, which contains a call to a procedure in the same segment.

PROGRAM 2:

Write a program that collects in data samples from a port at 1 ms interval. The upper 4 bits collected data same as mastered and stored in an array in successive locations.

```
; REGISTERS      :Uses CS, SS, DS, AX, BX, CX, DX, SI, SP
; PROCEDURES     : Uses WAIT

DATA_SEG  SEGMENT
    PRESSURE  DW      100      DUP(0)  ; Set up array of 100 words
    NBR_OF_SAMPLES EQU    100
    PRESSURE_PORT EQU  0FFF8h      ; hypothetical input port
DATA_SEG  ENDS

STACK_SEG SEGMENT STACK
    DW      40      DUP(0)      ; set stack of 40 words
    STACK_TOP LABEL WORD
STACK_SEG ENDS

CODE_SEG  SEGMENT
    ASSUME CS:CODE_SEG, DS:DATA_SEG, SS:STACK_SEG
START:    MOV     AX, DATA_SEG      ; Initialise data segment register
          MOV     DS, AX
          MOV     AX, STACK_SEG      ; Initialise stack segment register
          MOV     SS, AX
          MOV     SP, OFFSET STACK - TOP ; initialise stack pointer top of
          ; stack
          LEA     SI, PRESSURE        ; SI points to start of array
          ; PRESSURE
          MOV     BX, NBR_OF_SAMPLES ; Load BX with number
          ; of samples
          MOV     DX, PRESSURE_PORT   ; Point DX at input port
          ; it can be any A/D converter or
          ; data port.

READ_NEXT: IN     AX, DX              ; Read data from port
          ; please note use of IN instruction
          AND     AX, 0FFFFH          ; Mask upper 4 bits of AX
          MOV     [SI], AX            ; Store data word in array
          CALL    WAIT                ; call procedures wait for delay
```

```
                                INC     SI           ; Increment SI by two as dealing with
                                INC     SI           ; 16 bit words and not bytes
                                DEC     BX           ; Decrement sample counter
                                JNZ     READ_NEXT    ; Repeat till 100
                                                ; samples are collected
STOP:    NOP
WAIT     PROC     NEAR
          MOV     CX, 2000H           ; Load delay value
                                                ; into CX
HERE:    LOOP     HERE               ; Loop until CX = 0
          RET
WAIT     ENDP
CODE_SEG ENDS
          END
```

Discussion:

Please note that the CALL to the procedure as above does not indicate whether the call is to a NEAR procedure or a FAR procedure. This distinction is made at the time of defining the procedure.

The procedure above can also be made a FAR procedure by changing the definition of the procedure as:

```
WAIT     PROC FAR
          .
          .
WAIT     ENDS
```

The procedure can now be defined in another segment if the need so be, in the same assembly language file.

4.3.3 Parameter Passing in Procedures

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.

Let us discuss a program that uses a procedure for converting a BCD number to binary number.

PROGRAM 3:

Conversion of BCD number to binary using a procedure.

Algorithm for conversion procedure:

Take a packed BCD digit and separate the two digits of BCD.
Multiply the upper digit by 10 (0Ah)
Add the lower digit to the result of multiplication

The implementation of the procedure will be dependent on the parameter-passing scheme. Let us demonstrate this with the help of three programs.

Program 3 (a): Use of registers for parameter passing: This program uses AH register for passing the parameter.

We are assuming that data is available in memory location. BCD and the result is stored in BIN

```
;REGISTERS      : Uses CS, DS, SS, SP, AX
;PROCEDURES     : BCD-BINARY

DATA_SEG        SEGMENT
    BCD          DB 25h                ; storage for BCD value
    BIN          DB ?                  ; storage for binary value
DATA_SEG        ENDS
STACK_SEG       SEGMENT      STACK
    DW          200 DUP(0)           ; stack of 200 words
    TOP_STACK   LABEL        WORD
STACK_SEG       ENDS

CODE_SEG        SEGMENT
    ASSUME CS:CODE_SEG, DS:DATA_SEG, SS:STACK_SEG
START: MOV      AX, DATA_SEG          ; Initialise data segment
      MOV      DS, AX                 ; Using AX register
      MOV      AX, STACK_SEG          ; Initialise stack
      MOV      SS, AX                 ; Segment register. Why
                                      ; stack?
      MOV      SP, OFFSET TOP_STACK   ; Initialise stack pointer
      MOV      AH, BCD
      CALL     BCD_BINARY              ; Do the conversion
      MOV      BIN, AH                ; Store the result in the
                                      ; memory
      :
      :
; Remaining program can be put here
;PROCEDURE      : BCD_BINARY - Converts BCD numbers to binary.
;INPUT          : AH with BCD value
;OUTPUT         : AH with binary value
;DESTROYS       : AX

BCD_BINARY      PROC NEAR

    PUSHF                    ; Save flags
    PUSH  BX                 ; and registers used in procedure
    PUSH  CX                 ; before starting the conversion
                                ; Do the conversion
    MOV   BH, AH             ; Save copy of BCD in BH
    AND   BH, 0Fh            ; and mask the higher bits. The lower digit
                                ; is in BH
    AND   AH, 0F0h          ; mask the lower bits. The higher digit is in AH
                                ; but in upper 4 bits.
    MOV   CH, 04             ; so move upper BCD digit to lower
    ROR   AH, CH             ; four bits in AH
    MOV   AL, AH             ; move the digit in AL for multiplication
    MOV   BH, 0Ah            ; put 10 in BH
    MUL   BH                 ; Multiply upper BCD digit in AL
                                ; by 0Ah in BH, the result is in AL
    MOV   AH, AL             ; the maximum/ minimum number would not
                                ; exceed 8 bits so move AL to AH
    ADD   AH, BH             ; Add lower BCD digit to MUL result
; End of conversion, binary result in AH
    POP   CX                 ; Restore registers
    POP   BX
    POPF
```

```

                                RET                ; and return to calling program
BCD_BINARY    ENDP
CODE_SEG      ENDS
                                END      START

```

Discussion:

The above program is not an optimum program, as it does not use registers minimally. By now you should be able to understand this module. The program copies the BCD number from the memory to the AH register. The AH register is used as it is in the procedure. Thus, the contents of AH register are used in calling program as well as procedure; or in other words have been passed from main to procedure. The result of the subroutine is also passed back to AH register as returned value. Thus, the calling program can find the result in AH register.

The advantage of using the registers for passing the parameters is the ease with which they can be handled. The disadvantage, however, is the limit of parameters that can be passed. For example, one cannot pass an array of 100 elements to a procedure using registers.

Passing Parameters in General Memory

The parameters can also be passed in the memory. In such a scheme, the name of the memory location is used as a parameter. The results can also be returned in the same variables. This approach has a severe limitation. It is that you will be forced to use the same memory variable with that procedure. What are the implications of this bound? Well in the example above we will be bound that variable BCD must contain the input. This procedure cannot be used for a value stored in any other location. Thus, it is a very restrictive method of procedural call.

Passing Parameters Using Pointers

This method overcomes the disadvantage of using variable names directly in the procedure. It uses registers to pass the procedure pointers to the desired data. Let us explain it further with the help of a newer version of the last program.

Program 3 (c) version 2:

```

DATA_SEG      SEGMENT
                BCD          DB      25h      ; Storage for BCD test value
                BIN          DB      ?        ; Storage for binary value
DATA_SEG      ENDS

STACK_SEG SEGMENT  STACK
                DW      100 DUP(0)      ; Stack of 100 words
                TOP_STACK LABEL  WORD
STACK_SEG ENDS

CODE_SEG      SEGMENT
                ASSUME CS:CODE_SEG, DS:DATA_SEG, SS:STACK_SEG
START:        MOV     AX, DATA_SEG      ; Initialize data
                MOV     DS, AX            ; segment using AX register
                MOV     AX, STACK_SEG     ; initialize stack
                MOV     SS, AX            ; segment. Why stack?
                MOV     SP, OFFSET TOP_STACK ; initialize stack pointer
; Put pointer to BCD storage in SI and DI prior to procedure call.
                MOV     SI, OFFSET BCD    ; SI now points to BCD_IN
                MOV     DI, OFFSET BIN     ; DI points BIN_VAL
                ; (returned value)
                CALL    BCD_BINARY        ; Call the conversion

```

```

                                ; procedure
                                ; Continue with program
                                ; here

                                NOP

; PROCEDURE      : BCD_BINARY Converts BCD numbers to binary.
; INPUT          : SI points to location in memory of data
; OUTPUT         : DI points to location in memory for result
; DESTROYS       : Nothing

BCD_BINARY PROC NEAR
    PUSHF                ; Save flag register
    PUSH AX              ; and AX registers
    PUSH BX              ; BX
    PUSH CX              ; and CX
    MOV AL, [SI]         ; Get BCD value from memory
                        ; for conversion
    MOV BL, AL           ; copy it in BL also
    AND BL, 0Fh          ; and mask to get lower 4 digits
    AND AL, 0F0h         ; Separate upper 4 bits in AL
    MOV CL, 04           ; initialize counter CL so that upper digit
                        ; in AL can be brought to lower 4 bit
    ROR AL, CL           ; positions in AL
                        ; Load 10 in BH
    MOV BH, 0Ah
    MUL BH               ; Multiply upper digit in AL by 10
                        ; The result is stored in AL
    ADD AL, BL           ; Add lower BCD digit in BL to result of
                        ; multiplication

; End of conversion, now restore the original values prior to call. All calls will be in
; reverse order to save above. The result is in AL register.
    MOV [DI], AL        ; Store binary value to memory
    POP CX              ; Restore flags and
    POP BX              ; registers
    POP AX
    POPF
    RET
BCD_BINARY ENDP
CODE_SEG ENDS
END START

```

Discussion:

In the program above, SI points to the BCD and the DI points to the BIN. The instruction `MOV AL,[SI]` copies the byte pointed by SI to the AL register. Likewise, `MOV [DI], AL` transfers the result back to memory location pointed by DI.

This scheme allows you to pass the procedure pointers to data anywhere in memory. You can pass pointer to individual data element or a group of data elements like arrays and strings. This approach is used for parameters passing to BIOS procedures.

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. Let us revisit the same example, but using stack for parameter passing.

PROGRAM 3: Version 3

```

DATA_SEG      SEGMENT
                BCD          DB      25h      ; Storage for BCD test value
                BIN          DB      ?        ; Storage for binary value
DATA_SEG      ENDS

STACK_SEG     SEGMENT      STACK
                DW          100 DUP(0)      ; Stack of 100 words
                TOP_STACK LABEL WORD
STACK_SEG     ENDS

CODE_SEG      SEGMENT
                ASSUME CS:CODE_SEG, DS:DATA_SEG, SS:STACK_SEG
START:        MOV     AX, DATA              ; Initialise data segment
                MOV     DS, AX                ; using AX register
                MOV     AX, STACK-SEG .      ; initialise stack segment
                MOV     SS, AX                ; using AX register
                MOV     SP, OFFSET TOP_STACK ; initialise stack pointer
                MOV     AL, BCD                ; Move BCD value into AL
                PUSH    AX                    ; and push it onto word stack
                CALL    BCD_BINARY            ; Do the conversion
                POP     AX                    ; Get the binary value
                MOV     BIN, AL                ; and save it
                NOP                             ; Continue with program

; PROCEDURE      : BCD_BINARY Converts BCD numbers to binary.
; INPUT          : None - BCD value assumed to be on stack before call
; OUTPUT         : None - Binary value on top of stack after return
; DESTROYS       : Nothing

BCD_BINARY     PROC     NEAR
                PUSHF                          ; Save flags
                PUSH    AX                      ; and registers : AX
                PUSH    BX                      ; BX
                PUSH    CX                      ; CX
                PUSH    BP                      ; BP. Why BP?
                MOV     BP, SP                  ; Make a copy of the
                                                ; stack pointer in BP
                MOV     AX, [BP+ 12]            ; Get BCD number from
                                                ; stack. But why it is on
; BP+12 location? Please note 5 PUSH statements + 1 call which is intra-segment (so
; just IP is stored) so total 6 words are pushed after AX has been pushed and since it is
; a word stack so the BCD value is stored on  $6 \times 2 = 12$  locations under stack. Hence
; [BP + 12] (refer to the figure given on next page).
                MOV     BL, AL                  ; Save copy of BCD in BL
                AND     BL, 0Fh                 ; mask lower 4 bits
                AND     AL, F0H                 ; Separate upper 4 bits
                MOV     CL, 04                  ; Move upper BCD digit to low
                ROR     AL, CL                  ; position BCD digit for multiply location
                MOV     BH, 0Ah                 ; Load 10 in BH
                MUL     BH                      ; Multiply upper BCD digit in AL by 10
                                                ; the result is in AL
                ADD     AL, BL                  ; Add lower BCD digit to result.
                MOV     [BP + 12], AX           ; Put binary result on stack
                ; Restore flags and registers
                POP     BP
                POP     CX
                POP     BX
                POP     AX

```

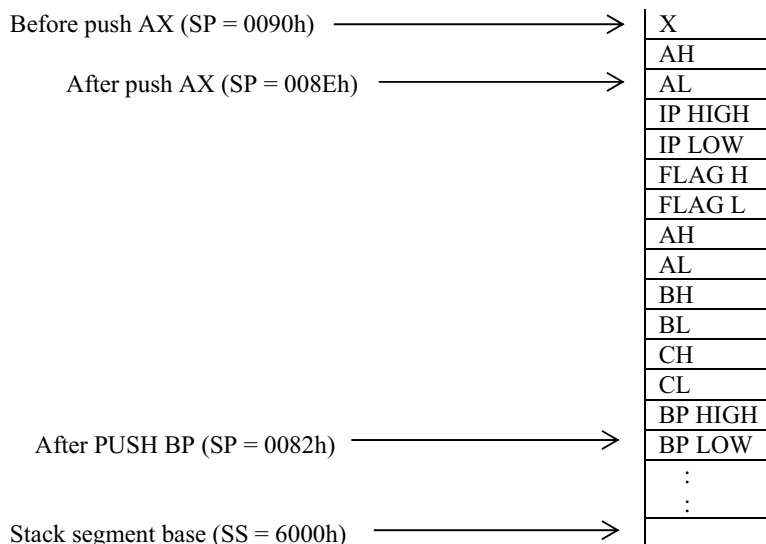
```

        POPF
        RET
BCD_BINARY    ENDP
CODE_SEG      ENDS
END           START

```

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 looks to be:



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. It copies the pushed value of AH and AL at offset 008Eh into the AX register. 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.

4.3.4 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, we need segment combination and global identifier directives. Let us discuss them briefly.

Segment Combinations

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 names and class (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, with the length the sum of the lengths of individual segments.

These details will be more clear after you go through program 4 and further readings.

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.

Let us explain all the above with the help of the following example:

PROGRAM 4:

Write a procedure that divides a 32-bit number by a 16-bit number. The procedure should be general, that is, it is defined in one module, and can be called from another assembly module.

```
; REGISTERS           :Uses CS, DS, SS, AX, SP, BX, CX
; PROCEDURES          : Far Procedure SMART_DIV
DATA_SEG SEGMENT WORD PUBLIC
    DIVIDEND DW 2345h, 89AB ; Dividend =
                                ; 89AB2345H
    DIVISOR DW 5678H ; 16-bit divisor
    MESSAGE DB 'INVALID DIVIDE', '$'
DATA_SEG ENDS

MORE_DATA SEGMENT WORD
    QUOTIENT DW 2 DUP(0)
    REMAINDER DW 0
MORE_DATA ENDS

STACK_SEG SEGMENT STACK
    DW 100 DUP(0) ; Stack of 100 words
    TOP - STACK LABEL WORD ; top of stack pointer
STACK_SEG ENDS

PUBLIC DIVISOR

PROCEDURES SEGMENT PUBLIC ; SMART_DIV is declared as an
    EXTRN SMART_DIV: FAR ; external label in procedure
                        ; segment of type FAR
PROCEDURES ENDS
; declare the code segment as PUBLIC so that it can be merged with other PUBLIC
; segments
CODE_SEG SEGMENT WORD PUBLIC
    ASSUME CS:CODE, DS:DATA_SEG, SS:STACK_SEG
START: MOV AX, DATA_SEG ; Initialize data segment
        MOV DS, AX ; using AX register
        MOV AX, STACK_SEG ; Initialize stack segment
```

```

MOV    SS, AX                ; using AX register
MOV    SP, OFFSET TOP_STACK ; Initialize stack pointer
MOV    AX, DIVIDEND          ; Load low word of
                             ; dividend
MOV    DX, DIVIDEND + 2      ; Load high word of
                             ; dividend
MOV    CX, DIVISOR           ; Load divisor
CALL SMART_DIV
; This procedure returns Quotient in the DX:AX pair and Remainder in CX register.
; Carry bit is set if result is invalid.
JNC    SAVE_ALL              ; IF carry = 0, result valid
JMP    STOP                  ; ELSE carry set, don't
                             ; save result
        ASSUME DS:MORE_DATA ; Change data segment
SAVE_ALL:  PUSH DS            ; Save old DS
MOV    BX, MORE_DATA         ; Load new data segment
MOV    DS, BX                ; register
MOV    QUOTIENT, AX          ; Store low word of
                             ; quotient
MOV    QUOTIENT + 2, DX      ; Store high word of
                             ; quotient
MOV    REMAINDER, CX         ; Store remainder
        ASSUME DS:DATA_SEG
POP     DS                   ; Restore initial DS
JMP     ENDING
STOP:     MOV    DL, OFFSET MESSAGE
MOV     AX, AH 09H
INT     21H
ENDING:   NOP
CODE_SEG  ENDS
END       START

```

Discussion:

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 next 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. On the other hand 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. Let us now define the PROCEDURE module:

```

; PROGRAM    MODULE    PROCEDURES

; INPUT      : Dividend - low word in AX, high word in DX, Divisor in CX
; OUTPUT     : Quotient - low word in AX, high word in DX. Remainder in CX
               ; Carry    - carry flag is set if try to divide by zero
; DESTROYS   : AX, BX, CX, DX, BP, FLAGS
DATA_SEG    SEGMENT    PUBLIC ; This block tells the assembler that
EXTRN DIVISOR:WORD          ; the divisor is a word variable and is
DATA_SEG    ENDS           ; external to this procedure. It would be
                             ; found in segment named DATA_SEG
PUBLIC      SMART_DIV      ; SMART_DIV is available to
                             ; other modules. It is now being defined

```

```

PROCEDURES      SEGMENT PUBLIC      ; in PROCEDURES SEGMENT.
SMART_DIV       PROC FAR
    ASSUME CS:PROCEDURES, DS:DATA_SEG
    CMP         DIVISOR, 0           ; This is just to demonstrate the use of
                                     ; external variable, otherwise we can
                                     ; check it through CX register which
                                     ; contains the divisor.

    JE          ERROR_EXIT          ; IF divisor = 0, exit procedure
    MOV         BX, AX              ; Save low order of dividend
    MOV         AX, DX              ; Position high word for 1st divide
    MOV         DX, 0000h           ; Zero DX
    DIV         CX                  ; DX:AX/CX, quotient in AX,
                                     ; remainder in DX

    MOV         BP, AX              ; transfer high order of final result to BP
    MOV         AX, BX              ; Get back low order of dividend. Note
                                     ; DX contains remainder so DX : AX is
                                     ; the actual number

    DIV         CX                  ; DX:AX/CX, quotient in AX,
                                     ; 2nd remainder that is final remainder
                                     ; in DX

    MOV         CX, DX              ; Pass final remainder in CX
    MOV         DX, BP              ; Pass high order of quotient in DX
                                     ; AX contains lower word of quotient

    CLC                           ; Clear carry to indicate valid result
    JMP         EXIT                ; Finished
ERROR_EXIT: STC                     ; Set carry to indicate divide by zero
EXIT: RET
SMART_DIV       ENDP
PROCEDURES      ENDS
                END

```

Discussion:

The procedure accesses the data item named DIVISOR, which is defined in the main, 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 main that is DATA_SEG and the procedure is in a PUBLIC segment.

Check Your Progress 2

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 register is not suitable when large numbers of parameters are to be passed. ☐
 - (e) Parameter passing in general memory is a flexible way of passing parameters. ☐
 - (f) Parameter passing through pointers can be used to pass a group of data elements. ☐

- (f) Parameter passing through stack is used whenever assembly language programs are interfaced with any high level language programs. ☐
- (h) In multiuser systems parameters should be passed using pointers. ☐
- (i) A variable say USAGE is declared in a PROCEDURE segment, however it is used in a separate module. In such a case the declaration of USAGE should contain EXTRN verb. ☐
- (i) 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.

4.4 INTERFACING ASSEMBLY LANGUAGE ROUTINES TO HIGH LEVEL LANGUAGE PROGRAMS

By now you can write procedures, both external and internal, and pass parameters, especially through stack, let us use these concepts, to see how assembly language can be interfaced to some high level language programs. It is very important to learn this concept, because then you can combine the advantages of both the types of languages, that is, the ease of programming of high level languages and the speed and the scope of assembly language. Assembly language can be interfaced with most of the high level languages like C, C++ and database management systems.

What are the main considerations for interfacing assembly to HLL? To answer that we need to answer the following questions:

- How is the subroutine invoked?
- How are parameters passed?
- How are the values returned?
- How do you declare various segments so that they are consistent across calling program?

The answer to the above questions are dependent on the high level language (HLL). Let us take C Language as the language for interfacing. The C Language is very useful for writing user interface programs, but the code produced by a C compiler does not execute fast enough for telecommunications or graphics applications. Therefore, system programs are often written with a combination of C and assembly language functions. The main user interface may be written in C and specialized high speed functions written in assembly language.

The guidelines for calling assembly subroutines from C are:

- (i) Memory model: The calling program and called assembly programs must be defined with the same memory model. One of the most common convention that makes NEAR calls is .MODEL SMALL, C
- (ii) The naming convention normally involve an underscore (_) character preceding the segment or function name. Please note, however, this underscore is not used while making a call from C function. Please be careful about case sensitivity.

You must give a specific segment name to the code segment of your assembly language subroutine. The name varies from compiler to compiler. Microsoft C, and Turbo C require the code segment name to be `_TEXT` or a segment name with suffix `_TEXT`. Also, it requires the segment name `_DATA` for the data segment.

- (iii) The arguments from C to the assembly language are passed through the stack. For example, a function call in C:

```
function_name (arg1, arg2, ..., argn) ;
```

would push the value of each argument on the stack in the reverse order. That is, the argument *argn* is pushed first and *arg1* is pushed last on the stack. A value or a pointer to a variable can also be passed on the stack. Since the stack in 8086 is a word stack, therefore, values and pointers are stored as words on stack or multiples of the word size in case the value exceeds 16 bits.

- (iv) You should remember to save any special purpose registers (such as CS, DS, SS, ES, BP, SI or DI) that may be modified by the assembly language routine. If you fail to save them, then you may have undesirable/ unexplainable consequences, when control is returned to the C program. However, there is no need to save AX, BX, CX or DX registers as they are considered volatile.

- (v) Please note the compatibility of data types:

char	Byte (DB)
int	Word (DW)
long	Double Word (DD)

- (vi) Returned value: The called assembly routine uses the followed registers for returned values:

char	AL
Near/ int	AX
Far/ long	DX : AX

Let us now look into some of the examples for interfacing.

4.4.1 Simple Interfacing

The following is a sample of the coding, used for procedure interfacing:

```
PUBLIC CUROFF
_TEXT SEGMENT WORD PUBLIC 'CODE'
    ASSUME CS:_TEXT
    _CUROFF PROC NEAR    ; for small model
:
:
```

The same thing can be written using the newer simplified directives in the following manner:

```
PUBLIC CUROFF
.MODEL small,C
.CODE
CUROFF PROC
:
:
```

This second source code is much cleaner and easier to read. The directives `.MODEL` and `.CODE` instruct the assembler to make the necessary assumptions and adjustments so that the routine will work with a small model of C program. (Please

refer to Assembler manuals on details on models of C program. The models primarily differ in number of segments).

PROGRAM 5:

Write an assembly function that hides the cursor. Call it from a C program.

```

                . PUBLIC CUROFF
                . MODEL small,C
                . CODE
CUROFF PROC
    MOV     AH,3                ; get the current cursor position
    XOR     BX,BX              ; empty BX register
    INT     10h                ; use int 10h to do above
    OR      CH,20h             ; force to OFF condition
    MOV     AH,01              ; set the new cursor values
    INT     10h
    RET
CUROFF ENDP
                END

```

For details on various interrupt functions used in this program refer to further readings.

The C program to test this routine is as follows:

```

#include <stdio.h>
void curoff(void);
void main()
{
    printf("%s\n", "The cursor is now turning off");
    curoff();
}

```

You can write another procedure in assembly language program to put the cursor on. This can be done by replacing OR CH,20h instruction by AND CH,1Fh. You can call this new function from C program to put the cursor on after the curoff.

4.4.2 Interfacing Subroutines With Parameter Passing

Let us now write a C program that calls the assembly program for parameter passing. Let us extend the previous two programs such that if on function call 0 is passed then cursor is turned off and if 1 is passed then cursor is turned on. The calling C program for such may look like:

```

#include <stdio.h>
void cursw(int);
void main()
{
    printf("%s\n", "the cursor is now turning off");
    cursw(0); /* call to assembly routine with 0 as parameter
    getchar();
    printf("%s\n", "the cursor is now turning on");
    cursw(1); /* call to assembly routine with parameter as 1.
}

```

The variables in C or C++ are passed on the stack.

Let us now write the assembly routine:

PROGRAM 6:

Write a subroutine in C for toggling the cursor using old directives.

```

;
; use small memory model for C – near code segment

_DATA      SEGMENT WORD    'DATA'
  CURVAL EQU      [BP+4]    ; parameters
_DATA      ENDS

_TEXT      SEGMENT        BYTE PUBLIC      'CODE'
DGROUP     GROUP         _DATA
            ASSUME        CS:_TEXT,      DS:DGROUP, SS:DGROUP
            PUBLIC        _CURSW

_CURSW     PROC          NEAR
            PUSH          BP              ; BP register of caller is saved
            MOV           BP, SP          ; BP is pointing to stack now
            MOV           AX, CURVAL
            CMP           AX, 0H
            JZ            CUROFF          ; Execute code for cursor off
            CMP           AX, 01H
            JZ            CURON           ; Execute code for cursor on
            JMP           OVER            ; Error in parameter, do nothing
; write code for curoff
CUROFF:
:
:
            JMP           OVER

CURON:
:
:
; write code for curon

OVER:      POP           BP
            RET
_CURSW     ENDP
_TEXT      ENDS
            END

```

Why the parameter is found in [BP+4]? Please look into the following stack for the answer.

Parameter (0 or 1)	BP + 4
Return Address	BP + 2
Old value	BP + 0

PROGRAM 7:

Write a subroutine in C that toggles the cursor. It takes one argument that toggles the value between on (1) and off (0) using simplified directives:

```

PUBLIC CURSW
.MODEL small, C
.CODE
CURSW PROC switch:word

    MOV     AX, SWITCH      ; get flag value
    XOR     AX, AX          ; test zero / nonzero
    :
    :
    // routine to test the switch and accordingly

```

```
switch off or switch on the cursor //
:
:
CURSW ENDP
END
```

In a similar manner the variables can be passed in C as pointers also. Values can be returned to C either by changing the variable values in the C data segment or by returning the value in the registers as given earlier.

4.5 INTERRUPTS

Interrupts are signals that cause the central processing unit to suspend the currently executing program and transfer to a special program called an interrupt handler. The interrupt handler determines the cause of the interrupt, services the interrupt, and finally returns the control to the point of interrupt. Interrupts are caused by events external or internal to the CPU that require immediate attention. Some external events that cause interrupts are:

- Completion of an I/O process
- Detection of a hardware failure

An 8086 interrupt can occur because of the following reasons:

1. Hardware interrupts, caused by some external hardware device.
2. Software interrupts, which can be invoked with the help of INT instruction.
3. Conditional interrupts, which are mainly caused due to some error condition generated in 8086 by the execution of an instruction.

When an interrupt can be serviced by a procedure, it is called as the Interrupt Service Routine (ISR). The *starting addresses* of the interrupt service routines are present in the first 1K addresses of the memory (Please refer to Unit 2 of this block). This table is called the interrupt vector table.

How can we write an Interrupt Servicing Routine? The following are the basic but rigid sequence of steps:

1. Save the system context (registers, flags etc. that will be modified by the ISR).
2. Disable the interrupts that may cause interference if allowed to occur during this ISR's processing
3. Enable those interrupts that may still be allowed to occur during this ISR processing.
4. Determine the cause of the interrupt.
5. Take the appropriate action for the interrupt such as – receive and store data from the serial port, set a flag to indicate the completion of the disk sector transfer, etc.
6. Restore the system context.
7. Re-enable any interrupt levels that were blocked during this ISR execution.
8. Resume the execution of the process that was interrupted on occurrence of the interrupt.

MS-DOS provides you facilities that enable you to install well-behaved interrupt handlers such that they will not interfere with the operating system function or other interrupt handlers. These functions are:

Function	Action
Int 21h function 25h	Set interrupt vector
Int 21h function 35h	Get interrupt vector
Int 21h function 31h	Terminate and stay residents

Here are a few rules that must be kept in mind while writing down your own Interrupt Service Routines:

1. Use Int 21h, function 35h to get the required IVT entry from the IVT. Save this entry, for later use.
2. Use Int 21h, function 25h to modify the IVT.
3. If your program is not going to stay resident, save the contents of the IVT, and later restore them when your program exits.
4. If your program is going to stay resident, use one of the terminate and stay resident functions, to reserve proper amount of memory for your handler.

Let us now write an interrupt routine to handle “division by zero”. This file can be loaded like a COM file, but makes itself permanently resident, till the system is running.

This ISR is divided into two major sections: the initialisation and the interrupt handler. The initialisation procedure (INIT) is executed only once, when the program is executed from the DOS level. INIT takes over the type zero interrupt vector, it also prints a sign-on message, and then performs a terminate and “stay resident exit” to MS-DOS. This special exit reserves the memory occupied by the program, so that it is not overwritten by subsequent application programs. The interrupt handler (ZDIV) receives control when a divide-by-zero interrupt occurs.

```
CR          EQU    0DH          ; ASCII carriage return
LF          EQU    0Ah         ; ASCII line feed
BEEP        EQU    07h         ; ASCII beep code
BACKSP      EQU    08h         ; ASCII backspace code
```

```
CSEG SEGMENT PARA PUBLIC 'CODE'
        ORG 100h
        ASSUME CS:CSEG,DS:CSEG,ES:CSEG,SS:CSEG
```

```
INIT PROC NEAR
        MOV     DX,OFFSET ZDIV    ; reset interrupt 0 vector
                                   ; to address of new
                                   ; handler using function 25h, interrupt
                                   ; 0 handles divide-by-zero
        MOV     AX, 2500h
        INT     21h
        MOV     AH,09             ; print identification message
        INT     21h
                                   ; DX assigns paragraphs of memory
                                   ; to reserve
        MOV     DX,((OFFSET PGM_LEN + 15)/16) + 10h
        MOV     AX,3100h          ; exit and stay resident
        INT     21h              ; with a return code = 0
INIT     ENDP
```

```
ZDIV PROC FAR
                                   ; this is the zero-divide
                                   ; hardware interrupt handler.
        STI                     ; enable interrupts.
        PUSH    AX               ; save general registers
        PUSH    BX
        PUSH    CX
        PUSH    DX
        PUSH    SI
        PUSH    DI
        PUSH    BP
        PUSH    DS
        PUSH    ES
```

```

MOV     AX,CS
MOV     DS,AX
MOV     DX,OFFSET WARN    ; Print warning "divide by
MOV     AH, 9              ; zero "and" continue or
INT     21h                ; quit?"

ZDIV1:  MOV     AH,1        ; read keyboard
        INT     21h
        CMP     AL,'C'     ; is it 'C' or 'Q'?
        JE      ZDIV3      ; jump if it is a 'C'.
        CMP     AL,'Q'

        JE      ZDIV2      ; jump if it's a 'Q'
        MOV     DX,OFFSET BAD ; illegal entry, send a
        MOV     AH,9       ; beep, erase the bad char
        INT     21h        ; and try again
        JMP     ZDIV1

ZDIV2:  MOV     AX, 4CFFh   ; user wants to abort the
        INT     21h        ; program, return with
                          ; return code = 255

ZDIV3:  MOV     DX,OFFSET CRLF ; user wishes to continue
        MOV     AH,9       ; send CRLF
        INT     21h
        POP     ES         ; restore general registers
        POP     DS         ; and resume execution
        POP     BP
        POP     DI
        POP     SI
        POP     DX
        POP     CX
        POP     BX
        POP     AX
        IRET

ZDIV    ENDP

SIGNON  DB      CR, LF, 'Divide by zero interrupt'
        DB      'Handler Installed'
        DB      CRLF,$'

WARN    DB      CR, LF, 'Divide by zero detected:'
        DB      CR, LF, 'Quit or Continue (C/Q) ?'
        DB      '$'

BAD     DB      BEEP, BACKSP, ",BACKSP,$'
CRLF    DB      CR,LF,$'

PGM_LEN EQU $-INIT
CSEG    ENDS

        END

```

4.6 DEVICE DRIVERS IN ASSEMBLY

Device drivers are special programs installed by the config.sys file to control installable devices. Thus, personal computers can be expanded at some future time by the installation of new devices.

The device driver is .com file organized in 3 parts.

- 1) The leader
- 2) The strategy procedure

3) The interrupt procedure

The driver has either .sys or .exe extension and is originated at offset address 0000h.

The Header

The header contains information that allows DOS to identify the driver. It also contains pointers that allow it to chain to other drivers loaded into the system.

The header section of a device driver is 18 bytes in length and contains pointers and the name of the driver.

Following structure of the header:

```
CHAIN DD -1      : link to next driver
ATTR DW 0        : driver attribute
STRT DW START    : address of strategy
INTER DW INT      : address if interrupt
DNAME DB 'MYDRIVER' : driver name.
```

The first double word contains a -1 that informs DOS this is the last driver in the chain. If additional drivers are added DOS inserts a chain address in this double word as the segment and offset address. The chain address points to the next driver in the chain. This allows additional drivers installed at any time.

The attribute word indicates the type of headers included for the driver and the type of device the driver installs. It also indicates whether the driver control a character driver or a block device.

The Strategy Procedure

The strategy procedure is called when loaded into memory by DOS or whenever the controlled device request service. The main purpose of the strategy is to save the request header and its address for use by the interrupt procedure.

The request header is used by DOS to communicate commands and other informations to the interrupt procedure in the device driver

The request header contains the length of the request header as its first byte. This is necessary because the length of the request header varies from command to command. The return status word communicate information back to DOS from the device driver.

The initialise driver command (00H) is always executed when DOS initialises the device driver. The initialisation commands pass message to the video display indicating that the driver is loaded into the system and returns to DOS the amount of memory needed by the driver. You may only use DOS INT 21H functions 00H. You can get more details on strategy from the further readings.

The Interrupt Procedure

The interrupt procedure uses the request header to determine the function requested by DOS. It also performs all functions for the device driver. The interrupt procedures must respond to at least the initialised driver command (00H) and any other commands required to control the device operated by the device driver. You must refer to the further readings for more details and examples of device drivers.

Check Your Progress 3

State True or False

T	F
---	---

- (a) Assembly language routines cannot be interfaced with BASIC programs. ☐
- (b) The key issue in interfacing is the selection of proper parameter passing method. ☐
- (c) The value of arguments to be passed are pushed in the stack in reverse order. ☐
- (d) AX, BX, CX or DX registers need not be saved in interfacing of assembly programs with high level language programs. ☐
- (e) Hardware interrupts can be invoked with the help of INT function. ☐

2. What are the sequences of steps in an interrupt service routine?

.....

.....

.....

4.7 SUMMARY

In the above module, we studied some programming techniques, starting from arrays, to interrupts.

Arrays can be of byte type or word type, but the addressing of the arrays is always done with respect to bytes. For a word array, the address will be incremented by two for the next access.

As the programs become larger and larger, it becomes necessary to divide them into smaller modules called procedures. The procedures can be NEAR or FAR depending upon where they are being defined and from where they are being called. The parameters to the procedures can be passed through registers, or through memory or stack. Passing parameters in registers is easier, but limits the total number of variables that can be passed. In memory locations it is straight forward, but limits the use of the procedure. Passing parameters through stack is most complex out of all, but is a standard way to do it. Even when the assembly language programs are interfaced to high level languages, the parameters are passed on stack.

Interrupt Service Routines are used to service the interrupts that could have arisen because of some exceptional condition. The interrupt service routines can be modified- by rewriting them, and overwriting their entry in the interrupt vector table.

This completes the discussion on microprocessors and assembly language programming. The above programming was done for 8086 microprocessor, but can be tried on 80286 or 80386 processors also, with some modification in the assembler directives. The assembler used here is MASM, Microsoft assembler. The assembly language instructions remain the same for all assemblers, though the directives vary from one assembler to another. For further details on the assembler, you can refer to their respective manuals. You must refer to further readings for topics such as Interrupts, device drivers, procedures etc.

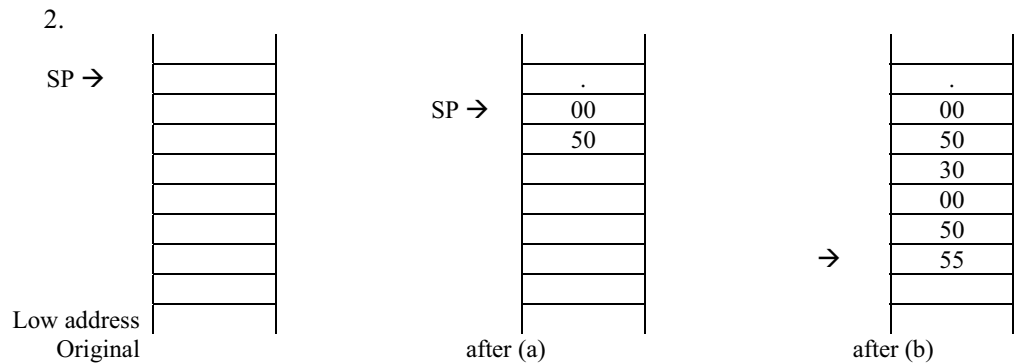
4.8 SOLUTIONS/ ANSWERS

Check Your Progress 1

1. We will give you an algorithm using XLAT instruction. Please code and run the program yourself.
 - Take a sentence in upper case for example 'TO BE CONVERTED TO LOWER CASE' create a table for lower case elements.
 - Check that the present ASCII character is an alphabet in upper case. It implies that ASCII character should be greater than 40h and less than 58h.
 - If it is upper case then subtract 41h from the ASCII value of the character. Put the resultant in AL register.
 - Set BX register to the offset of lower case table.
 - Use XLAT instruction to get the required lower case value.
 - Store the results in another string.
2. (a) False (b) False (c) True

Check Your Progress 2

1. (a) True (b) False (c) False (d) True (e) False (f) True (g) True (h) False (i) False (j) True.



- (c) The return for FIRST can occur only after return of SECOND. Therefore, the stack will be back in original state.

Check Your Progress 3

1. (a) False (b) False (c) True (d) True (e) False
2.
 - Save the system context
 - Block any interrupt, which may cause interference
 - Enable allowable interrupts
 - Determine the cause of interrupt
 - Take appropriate action
 - Restore system context
 - Enable interrupts which were blocked in Step 2