Lecture Notes

On

COMPUTER ORGANIZATION & MICROPROCESSOR (R20A1201)
Department of Information Technology

B.TECH II YEAR – I SEM
(2022-23)

Prepared by:
Mr. M. Ramanjaneyulu
Associate Professor, ECE Dept.
&
Mr.N.Suresh,
Assistant Professor, ECE Dept.

MALLA REDDY COLLEGE OF ENGINEERING & TECHNOLOGY
(Autonomous Institution – UGC, Govt. of India)

MALLA REDDY COLLEGE OF ENGINEERING AND TECHNOLOGY

B.TECH - II- YEAR I-SEM-I   **T L/T/P/C**

3/-/-/3

(R20A1201) COMPUTER ORGANISATION & MICROPROCESSORS

COURSE OBJECTIVES:

Students should be able:

1. To understand basic components of computers and architecture of 8086 microprocessor

2. To classify the instruction formats and various addressing modes of8086 microprocessor.

3. To represent the data and understand how computations are performed at machine level.

4. To outline the memory organization and I/Organization.

5. To understand the parallelism both in terms of single and multiple processors.

UNIT - I Digital Computers: Introduction, Block diagram of Digital Computer, Definition of Computer Organization, Computer Design and Computer Architecture.

Basic Computer Organization and Design: Instruction codes, Computer Registers, Computer instructions, Timing and Control, Instruction cycle, Memory Reference Instructions, Input – Output and Interrupt, Complete Computer Description.

Micro Programmed Control: Control memory, Address sequencing, micro program example, design of control unit.

UNIT - II Central Processing Unit: The 8086 Processor Architecture, register organization, Physical memory organization, General Bus Operation, I/O Addressing Capability, Special Processor Activities, Minimum and Maximum mode system and timings. 8086 Instruction Set and Assembler Directives-Machine language instruction formats, addressing modes, Instruction set of 8086, Assembler directives and operators.

UNIT - III Assembly Language Programming with 8086- Machine level programs, Machine coding the programs, Programming with an assembler, Assembly Language example programs. Stack structure of 8086, Interrupts and Interrupt service routines, Interrupt cycle of 8086, Interrupt programming, Passing parameters to procedures, Macros, Timings and Delays.

UNIT - IV Computer Arithmetic: Introduction, Addition and Subtraction, Multiplication Algorithms, Division Algorithms, Floating - point Arithmetic operations. Input-Output Organization: Peripheral Devices, Input-Output Interface, Asynchronous data transfer, Modes of Transfer, Priority Interrupt, Direct memory Access, Input –Output Processor (IOP), Intel 8089 IOP.

UNIT - V Memory Organization: Memory Hierarchy, Main Memory, Auxiliary memory, Associate Memory, Cache Memory.

Pipeline and Vector Processing: Parallel Processing, Pipelining, Arithmetic Pipeline, Instruction Pipeline, RISC Pipeline, Vector Processing, Array Processors.

REFERENCE BOOKS:

1. Microprocessors and Interfacing, D V Hall, SSSP Rao, 3rd edition, McGraw Hill India Education Private Ltd.

2. Carl Hamacher, Zvonko Vranesic, Safwat Zaky: Computer Organization, 5th Edition, Tata McGraw Hill,2002

3. Computer Organization and Architecture, William Stallings, 9th Edition, Pearson.

4. David A. Patterson, John L. Hennessy: Computer Organization and Design – The Hardware/ Software Interface ARM Edition, 4th Edition, Elsevier, 2009.

COURSE OUTCOMES:

Students will be able:

▪ To identify the basic components and the design of CPU, ALU and Control Unit.

 ▪ To interpret memory hierarchy and describe the impact on computer cost/performance.

▪ To express instruction level parallelism and pipelining for high performance Processor design.

▪ To represent the instruction set, instruction formats

INDEX

UNIT-1
INTRODUCTION TO DIGITAL COMPUTERS

CONTENTS:

Digital Computers:

- Introduction,
- Block diagram of Digital Computer,
- Definition of Computer Organization.

Digital Computers:

It is a digital system that performs various computational tasks.

First electronic digital computers introduced in the year 1940's were primarily used for the numerical computations.

Digital computer uses the binary number system, which has two digits, 0 & 1.

A Binary digit is called a bit.

In computers, information is represented in **'Group of bits'.**

Computer System:
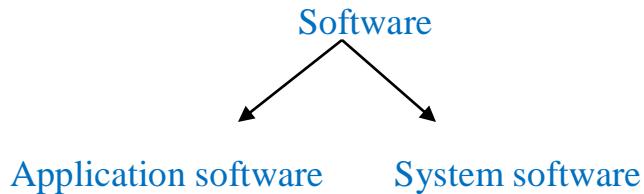
A computer system is subdivided into 2 functional units:

1. Hardware and

2. Software

1. Hardware: Consists of electronic components and electromechanical devices that comprise the physical entity of the system.

2. Software: Consists of instructions and data that the computer manipulates to perform various data processing tasks.

Program: It is a sequence of instructions for the computer

Software

Application software   System software

System Software:

Consists of collection of programs whose purpose is to make more effective use of computer.

The programs included in the system software are referred to as **operating system.** The system software is an indispensable part of a computer.

Application Software:

It is software **that performs specific tasks for an end-user**.

For example, A High level language program written by user to solve particular data processing needs is an **Application program.**

A compiler that is used to translate high level language to machine language is a **System program.**

WHY STUDY COMPUTER ORGANIZATION?

It gives an insight of how a computer executes programs internally and can help programmer to write more effective programs.

For system programmers, a good knowledge of Computer Organization is essential because they need to program the bare hardware without the support of an operating system.

# Relation between Computer Architecture, Organization, System program and Application program
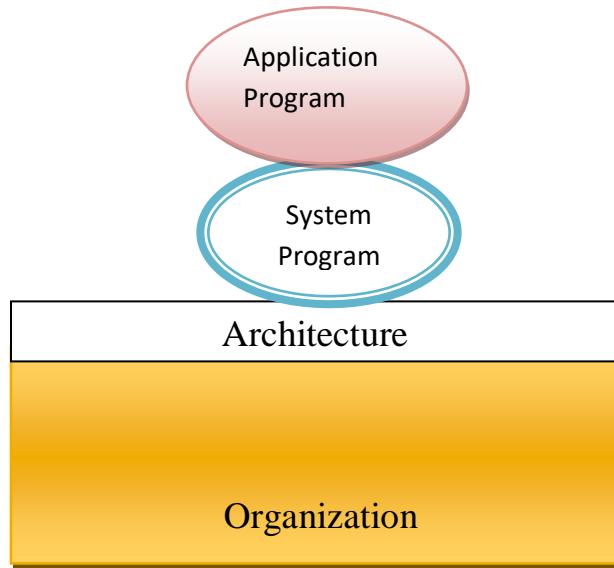
Fig1.1 Organization Implements Architecture

Computer Architecture:

It gives the external view of the computer. It is concerned with the **structure and behavior** of the computer.

An Assembly level programmer needs to be aware of *Specific Instruction supported by the processor, the instruction formats, the specific registers, and their roles, the way to perform input or output data.*

Computer Organization:

CO is concerned with the way the hardware components operate and the way they are connected together to form the computer system.

The various components are assumed to be in place and the task is to investigate the organizational structure to verify that computer parts operate as intended.

CO gives an internal view of a computer and the roles that internal components play during execution of a program.

In other words, CO deals with how different parts of the computer such as the processor, memory, and peripheral devices are interconnected and the roles that internal components play during program execution.

**Figure 1.1** shows that System program (operating system) directly interacts with the Computer Hardware.

The Application program invokes the services offered by the System programs.

Application programs are independent of the Architecture (High level Language) and are converted to machine dependent programs through a system program.

The internal organization of a basic computer is defined by its **internal registers, the timing and the control structure, & the set of instructions that it uses.**

The internal organization of a digital system is defined by sequence of micro operations it performs on data stored in registers.

Block Diagram of a Digital Computer:
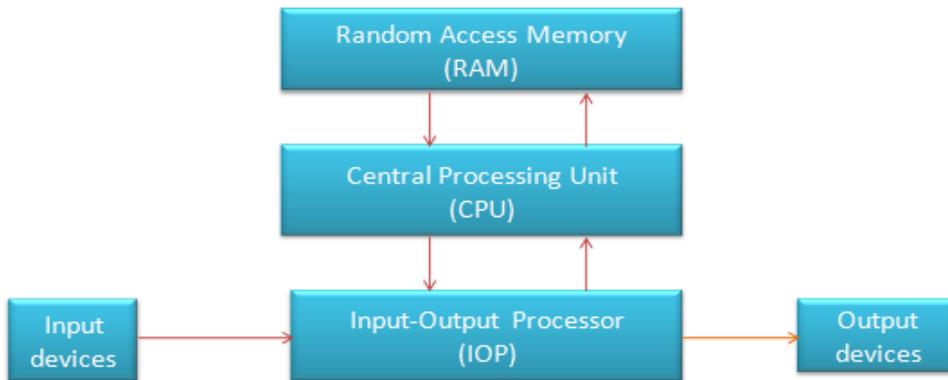
Fig1.2: Block Diagram of a Digital Computer

A digital computer consists of five functionally independent parts.
1. CPU: Contains an Arithmetic and logical unit for manipulating data, a number of registers for storing data, and control circuit for fetching and executing instructions.

2. RAM: Contains storage for instructions and data .Here, the CPU can access any location at random and retrieve the binary information within the fixed interval of time.

3. IOP: Contains electronic circuits for communicating and controlling the transfer of information between the computer and the outside the world.

4. Input Devices: Computers accept coded information through input units, which reads the data. Ex: Keyboard, Mouse, joy sticks.

5. Output Devices: Used to produce output through output devices. Ex: Printer, Plotter, Micro film Output, Voice Output, speakers.

Basic Computer Organization & Design

CONTENTS:

- Instruction Codes
- Computer Registers
- Computer Instructions
- Timing and Control
- Instruction Cycle
- Memory Reference Instructions
- Input-Output and Interrupt

Instruction Codes:

The user of a computer can **control the process** by means of a program

A program is a set of instructions that specifies the operations, operands, and the sequence by which processing has to occur.

The instruction code is a **group of bits** that instruct the computer to perform a **specific operation.**

An instruction consists of Opcode and operands.

Instruction Format

Opcode	Operands

Examples:
- ADD A, B
- ADD R1, R2
- MOV CX, 4929h
- MOV AX,BX
- SUB AX, BX
- INC AX

OPCODE:

The most basic part of instruction code is its operation part.

The **operation part** of an instruction code specifies the operation to be performed.

The operation code of an instruction is a group of bits that define operations such as ADD, Subtract, multiply, shift and complement.

Examples:

ADD A, B

SUB AX, BX

MUL AX, BX

The number of bits required for an operation code of an instruction depends on the total number of operations available on the computer.

The operation code must consists of at least n bits for a given 2^n distinct operations.

An **operation** is a part of instruction stored in computer memory.

The control unit receives the instruction from the memory and interprets the operation code bits.

It then (Opcode) issues a sequence of control signals to initiates micro operations in internal computer registers.

For every operation code, the control issues a sequence of micro operations needed for the hardware implementation of the specified operation.

OPERANDS:

An instruction code should not only specify the operation but also **the registers or the memory words where the operations are to be found** and also the registers or the memory words where the results are to be stored.

Memory words can be specified by instructions codes by their address.

Processor registers can be specified by assigning to the binary code of k bits that specifies one of 2^k registers.

DIFFERENT MODES OF INSTRUCTION:

Based on Second part of Instruction, We can specify the Different modes of an instruction. They are:

Immediate Mode:

When the second part of an instruction specifies an operand, the instruction is said to have an Immediate Operand

EX: ADD AX, 2387 H

Direct Address:

When the second part of an instruction specifies an address of an operand, The instruction is said to have a Direct Address

EX: MOV AX, [1592H]

Indirect Address:

When the second part of an instruction designates an address of a memory word in which the address of the operand is found, the instruction is said to have a Indirect Address.

EX: Load R1, @500

THE BASIC COMPUTER

The Basic Computer has two components, a Processor Register and Memory.

The Memory unit has a capacity of 4096 words. Each word contains 16 bits.

To specify the address of operand, 12 bits are needed; $4096 = 2^{12}$ .So 12 bits of an instruction word are needed to specify Address and 4 more bits are available for the Opcode. (Or 3 bits for opcode & 1 bit to specify Direct or Indirect Address).

Memory
4096 x 16

Instructions
(programs)

Operands
(data)

Processor Register
(accumulator or AC)
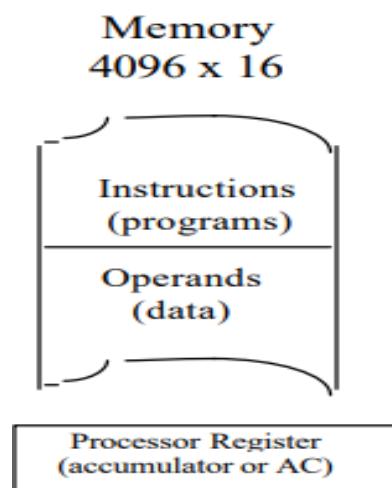
Fig1.3: Basic Computer

Stored program Organization:

The simplest way to organize a computer is to have **one processor register(AC) and a instruction code format** with two parts.

Stored Program Organization
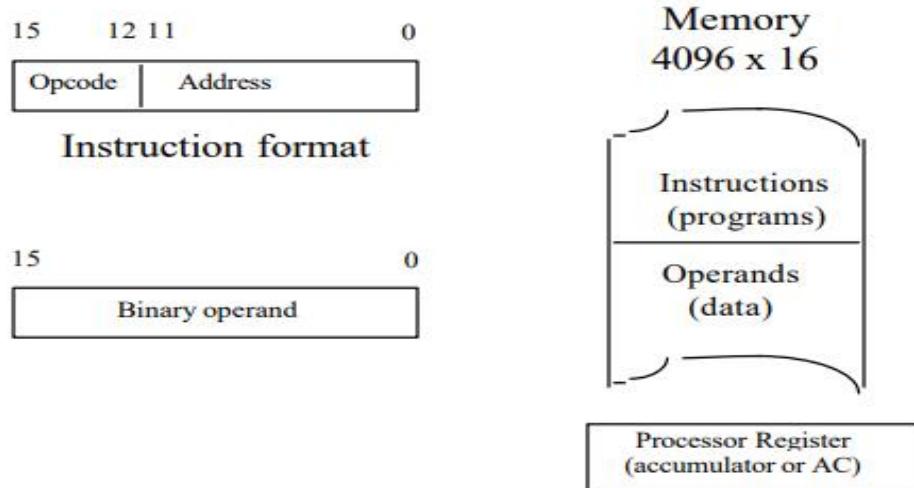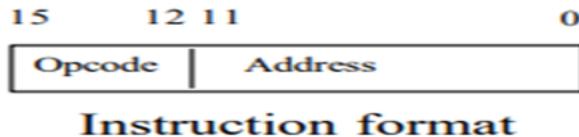
Fig1.4: Stored Program Organization

A computer instruction is often divided into two parts

The first part specifies the operation to be performed. The second part specifies an address.

The memory address tells the control where to find an operand in memory. This operand is read from memory and used as the data to be operated on together with the data stored in the processor register. Fig1.4 depicts the type of Organization.

MEMORY:

Instructions are stored in one section of memory and data in other.

For a memory unit with 4096 words we need 12 bits to specify an address since $2^{12}=4096$.

If we store each instruction code in one 16-bit memory word. There are 4 bits available for the operation code to specify one out of 16 possible operations and 12 bits to specify the address of an operand.

The control read a 16 bit operand form the data portion of the memory.

It then executes the operation specified by the operation code.
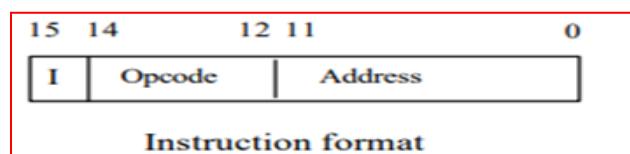
PROCESSOR REGISTER (ACCUMULATOR):

Computers that have a single processor register usually assign to it the name accumulator and label it AC. The operation is performed with the memory operand and content of AC.

If an operation in an instruction code does not need an operand from memory, the rest of the bits in the instruction can be used for other purposes. For example, operations such as clear AC complement AC, and increment AC operate on data stored in AC register. They do not need an operand from memory and they can be used to specify other operations for the computer. They do not need an operand from memory and they can be used to specify other operations for the computer.

Direct Addressing& Indirect Addressing:

Consider the instruction format shown in figure a.

Instruction format

It consists of 3 bit Opcode, a 12 bit address and an indirect address mode designated by I.

How to distinguish between a direct and indirect address?

A. One bit of the instruction code (I bit) can be used to distinguish between a direct and indirect address.

When I bit =0; It Specifies a Direct Address

When I bit =1; It Specifies an Indirect Address
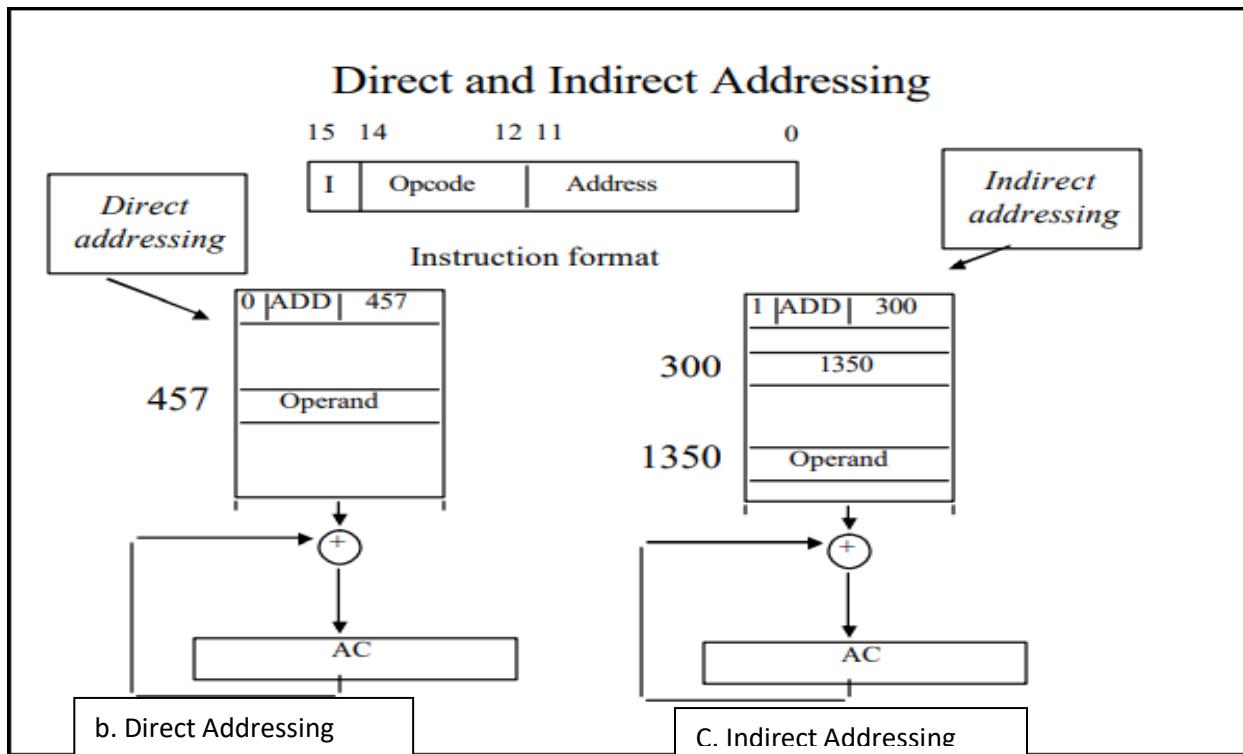
Fig1.5 Pictorial Representation of Direct & Indirect Addressing

Direct addressing

A direct address instruction is shown in figure b.

22 | 0 | Add | 457

It is placed in address 22 in memory and the I bit is 0, so the instruction is recognized as a direct address instruction. The opcode specifies an ADD instruction, and the address part is the binary equivalent of 457.The control finds the operand in memory at address 457 & adds it the content of AC.

Indirect addressing

The instruction in address 35 in memory is shown in figure c.

35 | 1 | ADD | 300

It has a mode bit I =1, therefore it is recognized as an Indirect address instruction. The address part is the binary equivalent of 300.The control goes to address 300 to find the address of the operand. The address of the operand in this case is 1350.The operand found in address 1350 is then added to the content of AC. The indirect address instruction needs two references to memory to fetch an operand.

- The first reference is to needed to read the address of the operand.
- The second reference is for the operand itself.

Effective Address:

It is defined as the address of an operand. Thus the Effective Address in the instruction of figure a is 457 and figure b is1350.

Computer Registers:
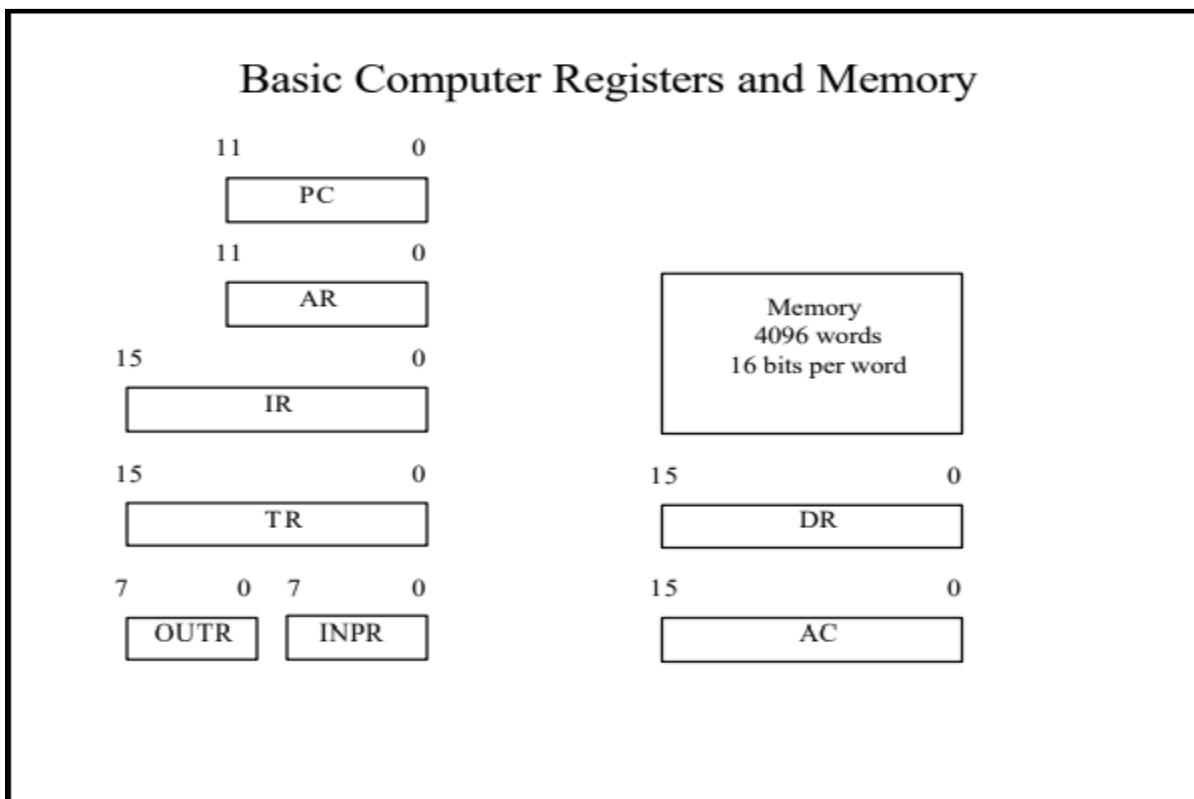
Fig1.6 Basic Computer Registers and Memory

A processor has many registers to hold instructions, addresses, data, etc

The processor has a register, the *Program Counter* (PC) **that holds the memory address of the next instruction .**

– Since the memory in the Basic Computer only has 4096 locations, the PC only needs 12 bits

The memory unit has a capacity of 4096 words and each word contains 16 bits. 12 bits of an instruction word are needed to specify the address of an operand. This leaves 3 bits for the operation part of the instruction and a bit (I) to specify a direct or indirect address. In a Direct or indirect addressing, **the processor needs to keep track of what locations in memory it is addressing:**

The *Address Register* (AR) is used for this

– The AR is a 12 bit register in the Basic Computer

When an operand is found, using either direct or indirect addressing, it is placed in the *Data Register* (DR). The **data register (DR) holds the operand read from memory.** The processor then uses this value as data for its operation

The accumulator (AC) register is a **general purpose processing register.** The significance of a general purpose register is that it can be referred to in instructions

– e.g. load AC with the contents of a specific memory location(LDA); store the contents of AC (STA)into a specified memory location

The instruction read form memory is placed in the instruction register (IR).

Often a processor will need a scratch register to store intermediate results or other temporary data; in the Basic Computer this is the *Temporary Register* (TR). **The temporary register (TR) is used for holding the temporary data during the processing.**

The Basic Computer uses a very simple model of input/output (I/O) operations. Input devices are considered to send 8 bits of character data to the processor. The processor can send 8 bits of character data to output devices

The *Input Register* (INPR) holds an **8 bit character received from an input device**

The *Output Register* (OUTR) holds **an 8 bit character to be send to an output device**

- The registers are also listed in table together with a brief description of their function and the number of bits that they contain..

List of Registers for the Basic Computer

Register Symbol	# of Bits	Register Name	Function
DR	16	Data Register	Holds memory operand
AR	12	Address Register	Holds mem. address
AC	16	Accumulator	Processor Reg.
IR	16	Instruction Register	Holds instruction code
PC	12	Program Counter	Holds instruction address
TR	16	Temporary Register	Holds temporary data
INPR	8	Input Register	Holds input character
OUTR	8	Output Register	Holds output character

Table1.1 List of registers for Basic computers

The memory address register has 12 bits since this is the width of a memory address.

$$\begin{array}{|c|} \hline 11 \qquad\qquad 0 \\ \hline AR \\ \hline \end{array}$$

The program counter also has 12 bits and it holds the next instruction to be read from memory after the current instruction is executed.

$$\begin{array}{|c|} \hline 11 \qquad\qquad 0 \\ \hline PC \\ \hline \end{array}$$

The PC goes through a counting sequence and causes the computer to read sequential instructions previously stored in memory. Instruction words are read and executed in sequence unless a branch instruction is encountered.

A branch instruction calls for a transfer to a nonconsecutive instruction in the program. The address part of a branch instruction is transferred to PC to become the address of the next instruction.

Common Bus System:

A basic computer has 8 registers, memory unit and a control unit. Paths must be provided to transfer information from one register to another and between memory and registers. . A more efficient scheme for transferring information in a system with many registers is to use a common bus. **To avoid excessive wiring,** memory and all the register are connected via **a common bus.** The connection of the registers and memory of the basic computer to a common bus system is shown in Fig.1.6. The **outputs of seven registers and memory are connected to the common bus.** The specific output that is selected for the bus lines at any given time is determined from the binary value of the selection variables $S_2 S_1 S_0$**.** The register who's **LD (Load)** is enabled receives the data from the bus. Registers can be incremented by setting the **INR** control input and can be cleared by setting the **CLR** control input.

The Accumulator's input must come via the Adder & Logic Circuit. This allows the Accumulator and Data Register to swap data simultaneously.

The address of any memory location being accessed must be loaded in the Address Register (AR).
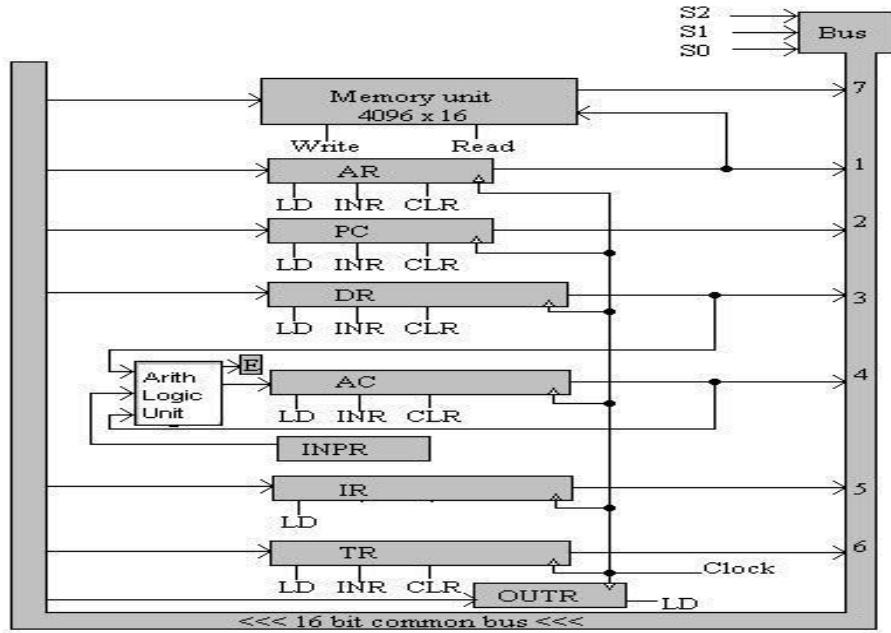
Fig1.6 Basic registers connected via a common Bus

SUMMARY

1. The organization of the computer is defined by its internal registers, the timing and control structure, and the set of instructions that it uses.

2. The user of a computer can control the process by means of a program. A program is a set of instructions that specify the operations, operations operands, and the sequence by which processing has to occur.

3. The general-purpose digital computer is capable of executing various micro operations and, in addition, can be instructed as to what specific sequence of operations it must perform.

4. An operation is part of an instruction stored in computer memory. It is a binary code tells the computer to perform a specific operation.

5. The operation part of an instruction code specifies the operation to be performed.

6. Instruction code formats are conceived computer designers who specify the architecture of the computer.

7. The simplest way to organize a computer is to have one processor register and instruction code format with two parts. The first part specifies the operation to be performed and the second specifies an address.

8. Computer instructions are normally stored in consecutive memory locations and are executed sequentially one at a time.

9. The direct and indirect addressing modes are used in the computer.

10. The memory word that holds the address of the operand in an indirect address instruction is used as a pointer to an array of data. The pointer could be placed in processor register instead of memory as done in commercial computers.

11. The basic computer has eight registers, a memory unit, and a control unit. Paths should be provided to transfer information from one register to another and between memory and registers.

12. The output of seven registers and memory are connected to the common bus.

13. The lines from the common bus are connected to the inputs of each register and the data input of each register and the data inputs of the memory.

14. The 16 lines of the common bus receive information from sex registers and the memory unit. The bus lines are connected to the inputs of six registers and the memory.

15. The content of any register can be applied onto the bus and an operation can be performed in the adder and logic circuit during the same clock cycle.

16. The input data and output data of the memory are connected to the common bus, but the memory address is connected to AR.

17. Content of any register can be applied onto the bus and an operation can be performed in the adder and logic circuit during the same clock cycle.

18. The clock transition at the end of the cycle transfers the content of the bus into the designated destination register and the output of the adder and logic circuit into AC.

Computer Instructions

The basic computer has 3 instruction code formats as shown in the figure below:
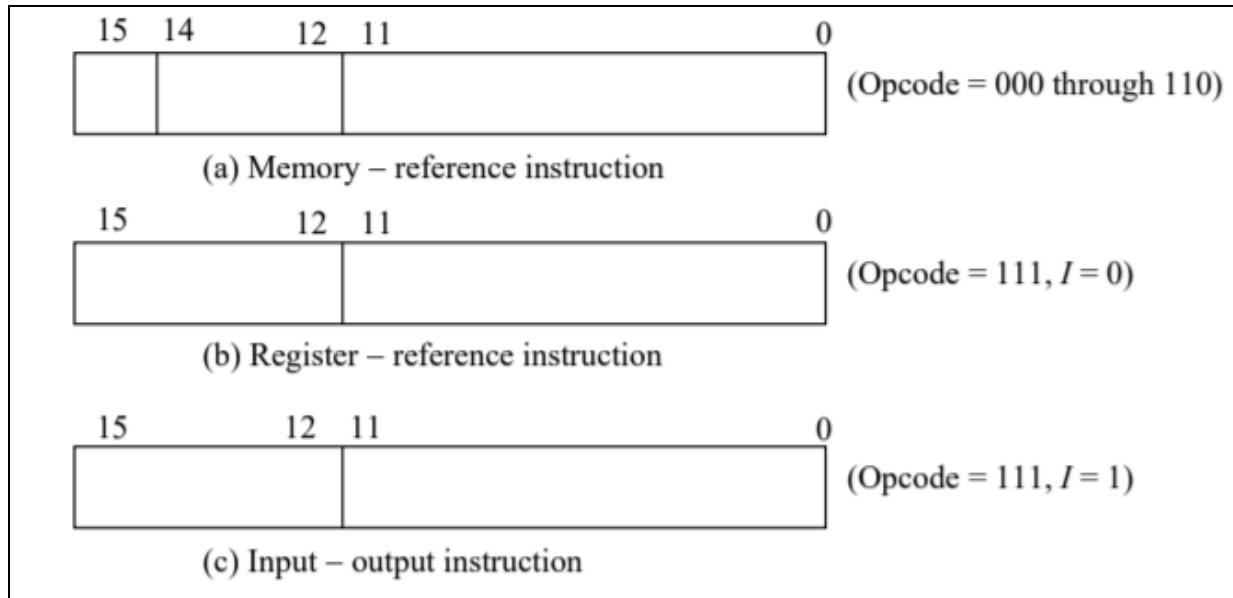
Fig1.7 Basic computer Instruction format

1. In Memory-reference instruction, 12 bits of memory is used to specify an address, 3bits for opcode and one bit to specify the addressing mode I.

 When I=0; represents direct Addressing Mode

I=1; represents Indirect Addressing Mode

2. The Register-reference instructions are represented by the **Opcode 111** with a **0 in the leftmost bit (bit 15) of the instruction.** A Register-reference instruction specifies an operation on or a test of the AC (Accumulator) register.Here the operand from memory is not needed, therefore the other 12 bits are used to specify the operation or test to be executed.

3. An Input-Output instruction does not need a reference to memory and is recognized by the **operation code 111** with a **1 in the leftmost bit of the instruction.** The remaining 12 bits are used to specify the type of the input-output operation or test performed.

The three operation code bits in positions 12 through 14 should be equal to 111. Otherwise, the instruction is a memory-reference type. When the three operation code bits are equal to 111, control unit inspects the bit in position 15. If

the bit (15) is 0, the instruction is a register-reference type. Otherwise, the instruction is an input-output type having bit 1 at position 15.

The instructions for the computer are listed in Table 1.2. The symbol designation is a three letter word and represents an abbreviation intended for programmers and users.

Symbol	Hex Code I = 0	Hex Code I = 1	Description
AND	0xxx	8xxx	AND memory word to AC
ADD	1xxx	9xxx	Add memory word to AC
LDA	2xxx	Axxx	Load AC from memory
STA	3xxx	Bxxx	Store content of AC into memory
BUN	4xxx	Cxxx	Branch unconditionally
BSA	5xxx	Dxxx	Branch and save return address
ISZ	6xxx	Exxx	Increment and skip if zero
CLA	7800		Clear AC
CLE	7400		Clear E
CMA	7200		Complement AC
CME	7100		Complement E
CIR	7080		Circulate right AC and E
CIL	7040		Circulate left AC and E
INC	7020		Increment AC
SPA	7010		Skip next instr. if AC is positive
SNA	7008		Skip next instr. if AC is negative
SZA	7004		Skip next instr. if AC is zero
SZE	7002		Skip next instr. if E is zero
HLT	7001		Halt computer
INP	F800		Input character to AC
OUT	F400		Output character from AC
SKI	F200		Skip on input flag
SKO	F100		Skip on output flag
ION	F080		Interrupt on
IOF	F040		Interrupt off

Table1.2 Basic computer Instructions

The hexadecimal code is equal to the equivalent hexadecimal number of the binary code used for the instruction.

By using the hexadecimal equivalent we reduced the **16 bits of an instruction code to four digits with each hexadecimal digit being equivalent to four bits.**

Instruction Set Completeness:

A computer should have a set of instructions so that the user can construct machine language programs to evaluate any function.

A set of instructions is said to be complete if the computer includes a sufficient number of instructions in each of the following categories:

- Arithmetic, logical and shift instructions
- A set of instructions for moving information to and from memory and processor registers.
- Program control Instructions together with instructions that check status conditions.
- Input and Output instructions

Arithmetic, logic and shift instructions provide computational capabilities for processing the type of data the user may wish to employ.

Transfer of Information: A huge amount of binary information is stored in the memory unit, but all computations are done in processor registers. Therefore, one must possess the **capability of moving information between these two units**.

Program control instructions such as branch instructions are used to change the sequence in which the program is executed.

Input and Output instructions act as an interface between the computer and the user. Programs and data must be transferred into memory, and the results of computations must be transferred back to the user.

Instruction Types

Functional Instructions
   - Arithmetic, logic, and shift instructions
   - ADD, CMA, INC, CIR, CIL, AND, CMA,CLA
Transfer Instructions: Data transfers between the main memory and the processoregisters
   - LDA, STA
Control Instructions
   - Program sequencing and control
   - BUN, BSA, ISZ
Input/Output Instructions
   - Input and output
   - INP, OUT

Timing and Control

The timings for all the registers in the basic computer is controlled by a master clock generator. Its clock pulses are applied to all flip-flops and register in the system & to flip-flops and registers in the control unit.

The clock pulses do not change the state of a register, unless the register is enabled by a control signal.

The control signals are generated in the control unit and provide control inputs for the bus's multiplexers and for the processor registers and provides micro operations for the accumulator.

CONTROL ORGANIZATION:

The Control Organization is classified into two major categories:

- Hardwired Control
- Micro programmed Control

Hardwired Control

The Hardwired Control organization involves the control logic to be implemented with gates, flip-flops, decoders, and other digital circuits.

The main advantage of Hardwired Control is its fast mode of operation.

If the design has to be modified or changed, it requires changes in the wiring among the various components.

Micro-programmed Control

The Micro programmed Control organization is implemented by using the programming approach.

The control information is stored in control memory.

The control memory is programmed to initiate requires sequence of micro operations.

Any required changes or modifications can be done by updating the micro program in control memory.

Control unit of a basic computer (Hardwired Control organization):

The following image shows the block diagram of a Hardwired Control organization.

Instruction Register
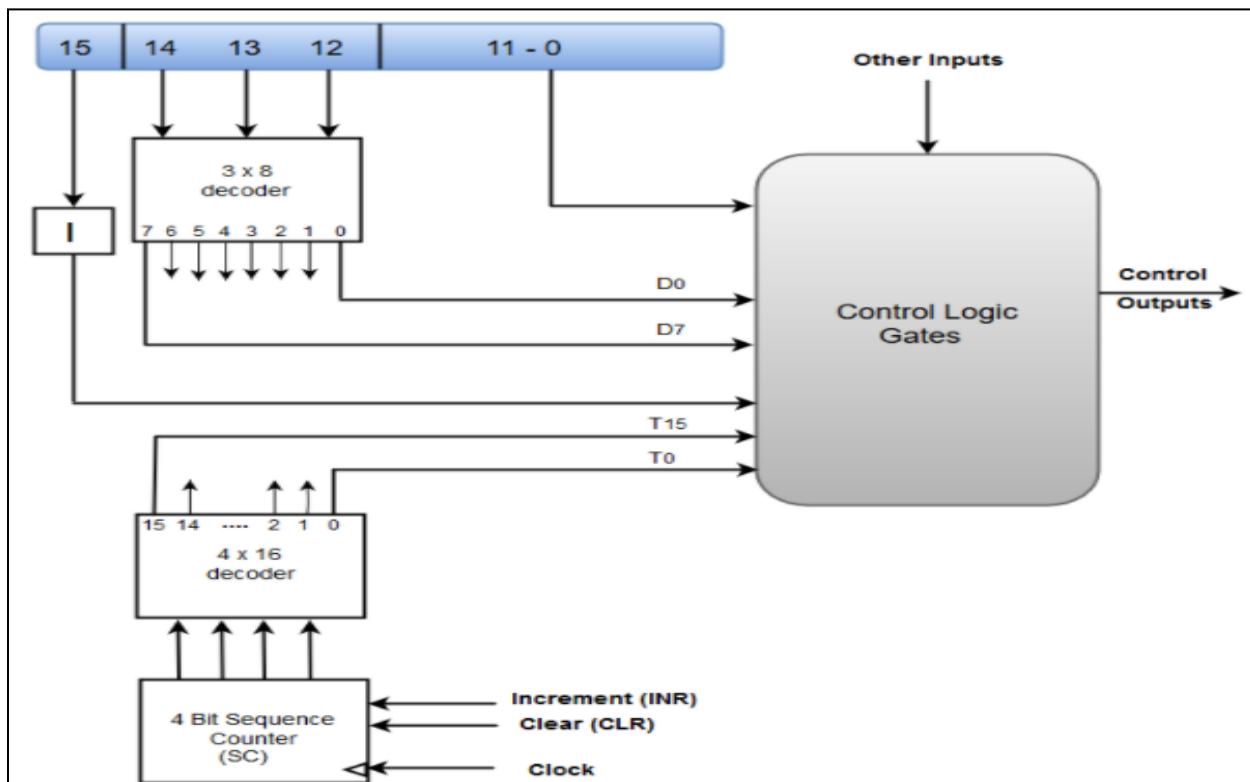
Fig1.8 Control Unit of Basic computer

A Hard-wired Control consists of **two decoders, a sequence counter, and a number of logic gates.**

An instruction fetched from the memory unit is placed in the instruction register (IR). The component of an instruction register includes: I bit, the operation code, and bits 0 through 11.

The operation code in bits 12 through 14 are decoded with a 3 x 8 decoder. The outputs of the decoder are designated by the symbols D0 through D7. The operation code at bit 15 is transferred to a flip-flop designated by the symbol I. The operation codes from Bits 0 through 11 are applied to the control logic gates.

The Sequence counter (SC) can count in binary numbers from 0 through 15.The outputs of the counter are decoded into 16 timing signals T_0 through T_{15}.The sequence counter SC can be incremented or cleared synchronously. Most of the time, the counter is incremented to provide the sequence of timing signals out of the 4 x 16 decoder. Once in a while, the counter is cleared to 0, causing the next active timing signal to be T_0.

As an example, consider the case where SC is incremented to provide timing signals T_0, T_1, T_2, T_3, and T_4 in sequence. At time T_4, SC is cleared to 0 if decoder output D_3 is active. This is expressed symbolically by the statement.

$$\mathbf{D_3T_4: SC \leftarrow 0}$$

Timing Signals

The timing diagram of Fig.1.9 shows the time relationship of the control signals.
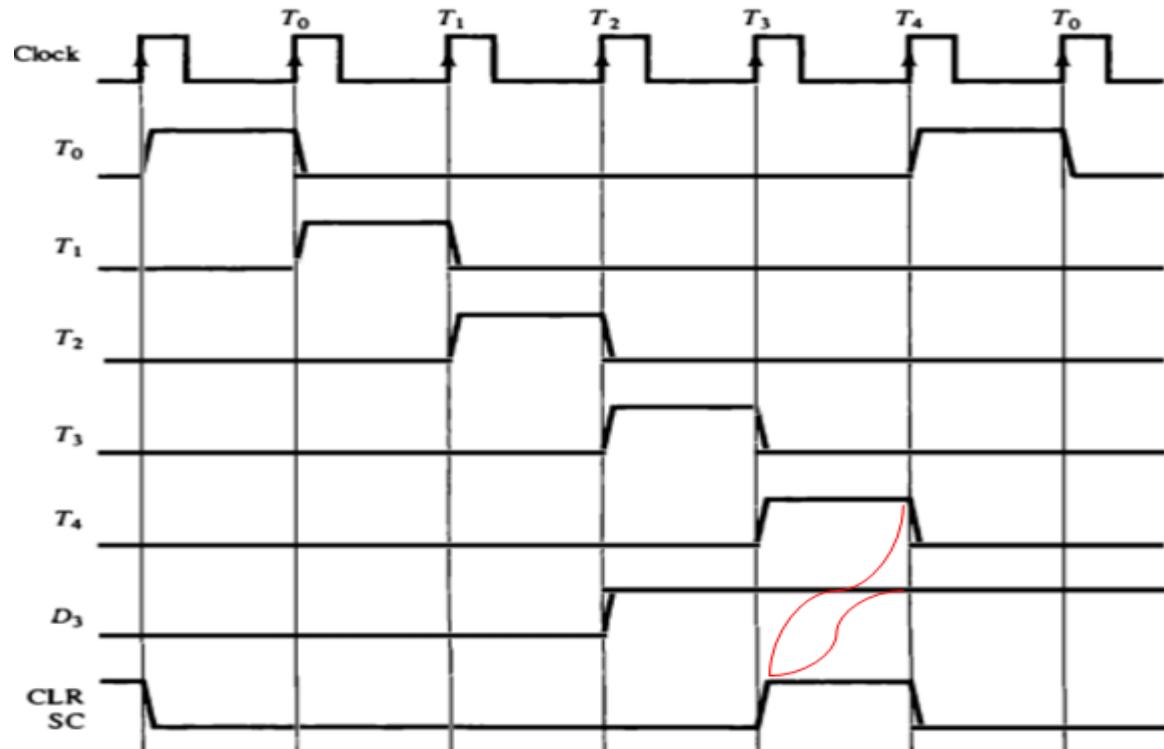
Fig1.9 Example of control Timing Signals

The sequence counter SC responds to the positive transition of the clock. Initially, the CLR input of SC is active. The first positive transition of the clock clears SC to 0, which in turn activates the timing signal T_0 out of the decoder.

T_0 is active during one clock cycle. The positive clock transition labeled T_0 in the diagram will trigger only those registers whose control inputs are connected to timing signal T_0.

SC is incremented with every positive clock transition unless its CLR input is active. This produces the sequence of timing signals T_0, T_1, T_2, T_3 ,T_4 and so on, as shown in the diagram.

If SC is not cleared, the timing signals will continue with T5, T_6 up to T_{15} and back to T_0.The last three waveforms in Fig. show how SC is cleared when $D_3T_4 = 1$. Output D_3 from the operation decoder becomes active at the end of timing signal T_2.

When timing signal T_4 becomes active, the output of the AND gate that implements the control function D_3T_4 becomes active. This signal is applied to the CLR input of SC. On the next positive clock transition (the one marked T_4 in the diagram) the counter is cleared to 0. This causes the timing signal T_0 to become active instead of T_5 that would have been active if SC were incremented instead of cleared.

  ➢ *Reference*

A memory read or writes cycle will be initiated with the rising edge of a timing signal. It will be assumed that a memory cycle time is less than the clock cycle time.

According to this assumption, a memory read or write cycle initiated by a timing signal will be completed by the time the next clock goes through its positive transition. The clock transition will then be used to load the memory word into a register. This timing relationship is not valid in many computers because the memory cycle time is usually longer than the processor clock cycle.

In such a case it is necessary to provide wait cycles in the processor until the memory word is available. To facilitate the presentation, we will assume that a wait period is not necessary in the basic computer.

To fully comprehend the operation of the computer, it is crucial that one understands the timing relationship between the clock transition and the timing signals. For example, the register transfer statement

$$T_0: AR \leftarrow PC$$

Specifies a transfer of the content of PC into AR if timing signal T_0 is active. T_0 is active during an entire clock cycle interval during this time the content of PC is placed onto the bus (with $S_2S_1S_0 = 010$) and the LD (load) input of AR is enabled.

The actual transfer does not occur until the end of the clock cycle when the clock goes through a positive transition. This same positive clock transition increments the sequence counter SC from 0000 to 0001. The next clock cycle has T_1 active and T_0 inactive.

INSTRUCTION CYCLE

A program residing in the memory unit of the computer consists of a sequence of Instructions. In the basic computer, each instruction cycle consists of the following phases:

> 1. Fetch an instruction from memory
>
> 2. Decode the instruction
>
> 3. Read the effective address from memory if the instruction has an indirect address
>
> 4. Execute the instruction

Upon the completion of step 4, the control goes back to step1 to fetch, decode and execute the next instruction. This process continues indefinitely unless a HALT instruction is encountered.

Fetch and Decode:

Initially, the PC is loaded with the address of the first instruction in the program.

The sequence counter SC is cleared to 0, provided a decoding timing signal T0

After each clock pulse, the SC is incremented by one, so that the timing signals go through the sequence **T0, T1, T2**, etc.

The micro operations for the **fetch and decode phases** can be specified by the following register transfer statements:

To: AR ← PC

T1: IR ← M [AR], PC ← PC+1

T2:D0,….,D7 ← Decode IR(12-14), AR ← IR(0-11), I ← IR(15)

FETCH PHASE:

Since only **AR is connected to the address inputs of memory**, it is necessary to transfer the address from PC to AR during the clock transition associated with timing signal T0.

The instruction read from memory is then placed in the instruction register IR with the clock transition associated with timing signal T1.

At the same time, PC is incremented by one to prepare it for the address of the next instruction in the program.

DECODE PHASE:

At time T2, the operation code in IR is decoded, the indirect bit is transferred to flip-flop I, and the address part of the instruction is transferred to AR.

Note that SC is incremented after each clock pulse to produce the sequence T0, T1, and T2

Determine the Type of Instruction:

The timing signal that is active after the decoding is T_3. During time T_3, the control unit determines the type of instruction that was just read from memory.

The flowchart of Fig. below presents an initial configuration for the instruction cycle and shows how the control determines the instruction type after the decoding.

The three possible instruction types available in the basic computer are specified in Fig. on basic computer formats.

1. **Memory Reference instructions**

2. **Register Reference instructions**

3. **I/O Reference instructions**

Decoder output D_7 is equal to 1 if the operation code is equal to binary 111. From Fig. on basic computer formats we determine that if $D_7 = 1$, the instruction must be **a register reference or I/O reference**.

If **$D_7 = 0$,** the operation code must be one of the other seven values 000 through 110, specifying a memory-reference instruction.

Control then inspects the value of the first bit of the instruction, which is now available in flip-flop I. If D7 = 0 and I = 1, we have a memory reference instruction with an indirect address.

It is then necessary to read the effective address from memory. The micro operation for the indirect address condition can be symbolized by the register transfer statement :

$$AR \leftarrow M[AR]$$

Initially, AR holds the address part of the instruction. This address is used during the memory read operation.

The word at the address given by AR is read from memory and placed on the common bus.

Flow chart for Instruction Cycle
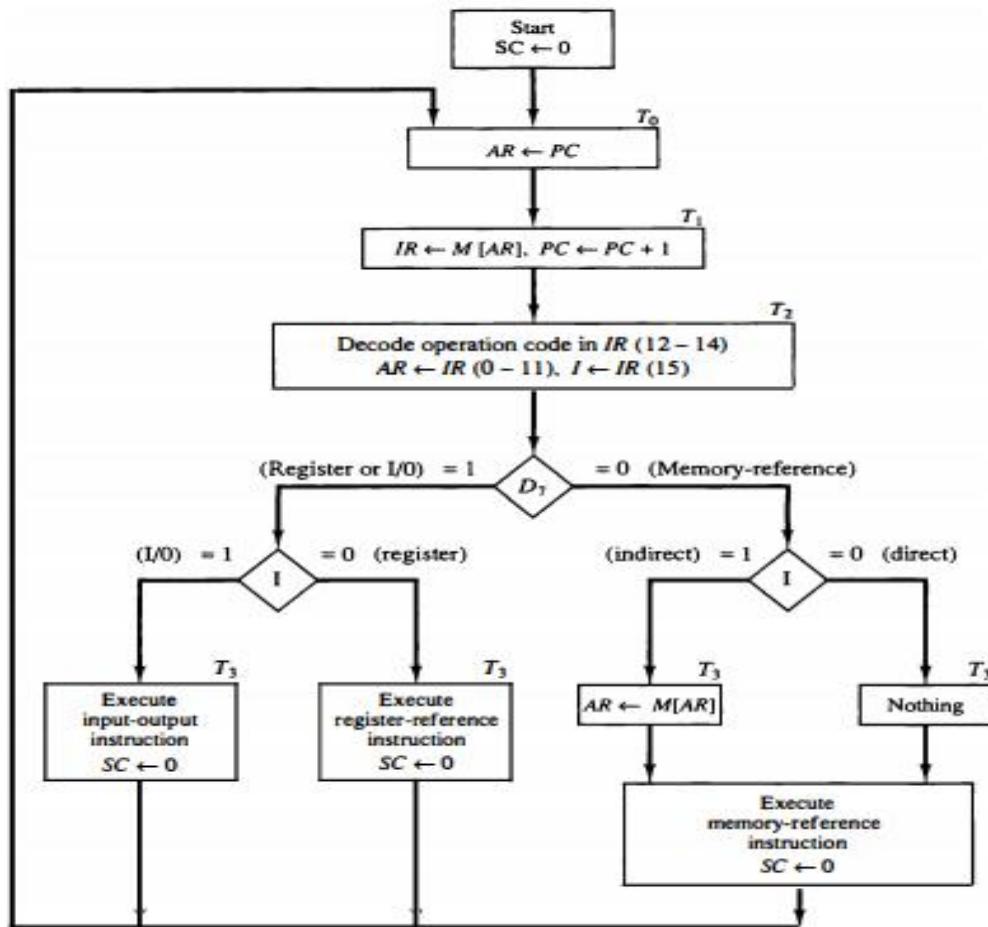

Fig1.10 Flow chart for Instruction cycle

The LD input of AR is then enabled to receive the indirect address that resided in the 12 least significant bits of the memory word.

The three instruction types are subdivided into four separate paths. The selected operation is activated with the clock transition associated with timing signal T_3. This can be symbolized as follows:

$D_7 I'T_3$: **Execute a register-reference instruction**

$D_7 IT_3$: **Execute an input-output instruction**

$D'_7 I'T_3$: **Nothing**

$D'_7 IT_3$: **AR ← M [AR]**

When a memory-reference instruction with $I = 0$ is encountered, it is not necessary to do anything since the effective address is already in AR.

However, the sequence counter SC must be incremented when $D'_7 T_3 = 1$, so that the execution of the memory-reference instruction can be continued with timing variable T_4

A register-reference or input-output instruction can be executed with the clock associated with timing signal T_3. After the instruction is executed, SC is cleared to 0 and control returns to the fetch phase with $T_0 = 1$.

Note that the sequence counter SC is either incremented or cleared to 0 with every positive clock transition.

We will adopt the convention that if SC is incremented, we will not write the statement SC ← SC + 1, but it will be implied that the control goes to the next timing signal in sequence.

When SC is to be cleared, we will include the statement SC ← 0.

Register-Reference Instructions($D_7 I'T_3$):

Register-reference instructions are recognized by the control when $D_7 = 1$ and $I = 0$. **These instructions** use bits 0 through 11 of the instruction code to specify one of 12 instructions.

These 12 bits are available in IR(0-11). They were also transferred to AR during time T_2. **The control functions** and micro operations for the register-reference instructions are listed in Table below.

These instructions are executed with the clock transition associated with timing variable T_3.

Each control function needs the Boolean relation $D_7I'T_3$, which we designate for convenience by the symbol **"r"**. **The control function** is distinguished by one of the bits in IR(0-11).

By assigning the symbol B_i to bit i of IR, all control functions can be simply denoted by rB_i.

TABLE Execution of Register-Reference Instructions

$D_7I'T_3 = r$ (common to all register-reference instructions)
$IR(i) = B_i$ [bit in $IR(0-11)$ that specifies the operation]

	r:	$SC \leftarrow 0$	Clear SC
CLA	rB_{11}:	$AC \leftarrow 0$	Clear AC
CLE	rB_{10}:	$E \leftarrow 0$	Clear E
CMA	rB_9:	$AC \leftarrow \overline{AC}$	Complement AC
CME	rB_8:	$E \leftarrow \overline{E}$	Complement E
CIR	rB_7:	$AC \leftarrow shr\ AC, AC(15) \leftarrow E, E \leftarrow AC(0)$	Circulate right
CIL	rB_6:	$AC \leftarrow shl\ AC, AC(0) \leftarrow E, E \leftarrow AC(15)$	Circulate left
INC	rB_5:	$AC \leftarrow AC + 1$	Increment AC
SPA	rB_4:	If $(AC(15) = 0)$ then $(PC \leftarrow PC + 1)$	Skip if positive
SNA	rB_3:	If $(AC(15) = 1)$ then $(PC \leftarrow PC + 1)$	Skip if negative
SZA	rB_2:	If $(AC = 0)$ then $PC \leftarrow PC + 1$	Skip if AC zero
SZE	rB_1:	If $(E = 0)$ then $(PC \leftarrow PC + 1)$	Skip if E zero
HLT	rB_0:	$S \leftarrow 0$ (S is a start–stop flip-flop)	Halt computer

Example of Register reference Instruction:

For example, the instruction CLA has the hexadecimal code 7800, which gives the binary equivalent 0111 1000 0000 0000.

The first bit is a zero and is equivalent to I'. **The next three bits** constitute the operation code and are recognized from decoder output D_7. **Bit 11 in IR is 1** and is recognized from B_{11}.

The control function that initiates the micro operation for this instruction is $D_7I'T_3B_{11} = rB_{11}$. **The execution** of a register-reference instruction is completed at time T_3. **The sequence counter SC** is cleared to 0 and the control goes back to fetch the next instruction with timing signal T_0.

The first seven register-reference instructions perform clear, complement, circular shift, and increment micro operations on the AC or E registers.

The next four instructions cause a skip of the next instruction in sequence when a stated condition is satisfied. The skipping of the instruction is achieved by incrementing PC once again (in addition, it is being incremented during the fetch phase at time T_1).

The condition control statements must be recognized as part of the control conditions.

- **The AC is positive when the sign bit** in AC $(15) = 0;$ **it is negative** when AC $(15) = 1$.

- The content of AC is zero (AC = 0) if all the flip-flops of the register are zero.

The **HLT instruction clears a start-stop flip-flop S** and stops the sequence counter from counting. To restore the operation of the computer, the start-stop flip-flop must be set manually.

Memory-Reference Instructions

In order to specify the micro operations needed for the execution of each instruction, it is necessary that the function that they are intended to perform be defined precisely.

We will now show that the function of the **memory-reference instructions can be defined precisely by means of register transfer notation.**

Table below lists the seven memory-reference instructions. The decoded output D_i for i = 0, 1, 2, 3, 4, 5, and 6 from the operation decoder that belongs to each instruction is included in the table.

The effective address of the instruction is in the address register AR and was placed there during timing signal T_2 when I = 0, or during timing signal T_3 when I = 1.

The execution of the memory-reference instructions starts with timing signal T_4. The symbolic description of each instruction is specified in the table in terms of register transfer notation.

TABLE Memory-Reference Instructions

Symbol	Operation decoder	Symbolic description
AND	D_0	$AC \leftarrow AC \wedge M[AR]$
ADD	D_1	$AC \leftarrow AC + M[AR], \quad E \leftarrow C_{out}$
LDA	D_2	$AC \leftarrow M[AR]$
STA	D_3	$M[AR] \leftarrow AC$
BUN	D_4	$PC \leftarrow AR$
BSA	D_5	$M[AR] \leftarrow PC, \quad PC \leftarrow AR + 1$
ISZ	D_6	$M[AR] \leftarrow M[AR] + 1,$ If $M[AR] + 1 = 0$ then $PC \leftarrow PC + 1$

AND to AC:

This is an instruction that performs the AND logic operation on pairs of bits in AC and the memory word specified by the effective address.

The result of the operation is transferred to AC. The micro operations that execute this instruction are:

$$D_0T_4: DR \leftarrow M[AR]$$
$$D_0T_5: AC \leftarrow AC \wedge DR, SC \leftarrow 0$$

The control function for this instruction uses the operation decoder D_0 since this output of the decoder is active when the instruction has an AND operation whose binary code value is 000.

Two timing signals are needed to execute the instruction. The clock transition associated with timing signal T_4 transfers the operand from memory into DR.

The clock transition associated with the next timing signal T_5 transfers to AC the result of the AND logic operation between the contents of DR and AC.

ADD to AC:

This instruction adds the content of the memory word specified by the effective address to the value of AC.

The sum is transferred into AC and the output carry C_{out} is transferred to the E (extended accumulator) flip-flop.

The micro operations needed to execute this instruction are

D_1T_4: DR ← M[AR]
D_1T_5: AC ← AC + DR, E ← C_{out}, SC ← 0

The same two timing signals, T_4 and T_5, are used again but with operation decoder D_1 instead of D_0, which was used for the AND instruction.

After the instruction is fetched from memory and decoded, only one output of the operation decoder will be active, and that output determines the sequence of rnicro operations that the control follows during the execution of a memory-reference instruction.

LDA: Load to AC

This instruction transfers the memory word specified by the effective address to AC.

The micro operations needed to execute this instruction are

D_2T_4: DR ← M [AR]
D_2T_5: AC ← DR, SC ← 0

Note that there is no direct path from the bus into AC (see figure under Common Bus System).

The adder and logic circuit receive information from DR which can be transferred into AC.

Therefore, it is necessary to read the memory word into DR first and then transfer the content of DR into AC.

The reason for not connecting the bus to the inputs of AC is the delay encountered in the adder and logic circuit.

It is assumed that the time it takes to read from memory and transfer the word through the bus as well as the adder and logic circuit is more than the time of one clock cycle.

By not connecting the bus to the inputs of AC we can maintain one clock cycle per micro operation.

STA: Store AC

This instruction stores the content of AC into the memory word specified by the effective address.

Since the output of AC is applied to the bus and the data input of memory is connected to the bus, we can execute this instruction with one micro operation:

$$D_3T_4: M[AR] \leftarrow AC, SC \leftarrow 0$$

BUN: Branch Unconditionally

This instruction transfers the program to the **instruction specified by the effective address.**

Remember that PC holds the address of the instruction to be read from memory in the next instruction cycle.

PC is incremented at time T_1 to prepare it for the address of the next instruction in the program sequence.

The BUN instruction allows the programmer to specify an instruction out of sequence and we say that the program branches (or jumps) unconditionally.

The instruction is executed with one micro operation:

$$D_4T_4: PC \leftarrow AR, SC \leftarrow 0$$

The effective address from AR is transferred through the common bus to PC.

Resetting SC to 0 transfers control to T_0. The next instruction is then fetched and executed from the memory address given by the new value in PC.

BSA: Branch and Save Return Address

This instruction is useful for branching to a portion of the program called a subroutine or procedure.
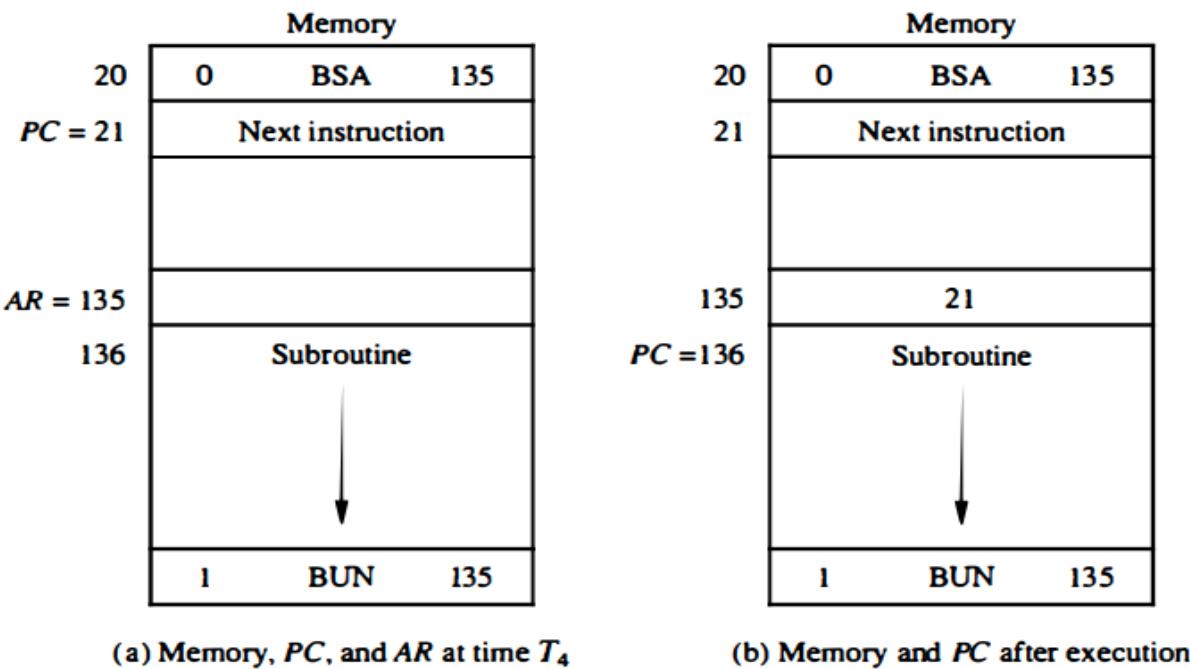
When executed, the BSA instruction stores the address of the next instruction in sequence (which is available in PC) into a memory location specified by the effective address.

The effective address plus one is then transferred to PC to serve as the address of the first instruction in the subroutine.

This operation was specified in Table above (see Memory-Reference Instructions) with the following register transfer:

$$M[AR] \leftarrow PC, PC \leftarrow AR + 1$$

A numerical example that demonstrates how this instruction is used with a subroutine is shown in Fig. below.

Figure Example of BSA instruction execution.

(a) Memory, *PC*, and *AR* at time T_4 (b) Memory and *PC* after execution

The BSA instruction is assumed to be in memory at address 20. Th I bit is 0 and the address part of the instruction has the binary equivalent of 135.

After the fetch and decode phases, PC contains 21, which is the address of the next instruction in the program (referred to as the return address). AR holds the effective address 135.

This is shown in part (a) of the figure. The BSA instruction performs the following numerical operation:

$$M [135] \leftarrow 21, PC \leftarrow 135 + 1 = 136$$

The result of this operation is shown in part (b) of the figure. The return address 21 is stored in memory location 135 and control continues with the subroutine program starting from address 136.

The return to the original program (at address 21) is accomplished by means of an indirect BUN instruction placed at the end of the subroutine.

When this instruction is executed, control goes to the indirect phase to read the effective address at location 135, where it finds the previously saved address 21.

When the BUN instruction is executed, the effective address 21 is transferred to PC.

The next instruction cycle finds PC with the value 21, so control continues to execute the instruction at the return address.

ISZ: Increment and Skip if Zero

This instruction increment the word specified by the effective address, and if the incremented value is equal to 0, PC is incremented by 1.

The programmer usually stores a negative number (in 2's complement) in the memory word.

As this negative number is repeatedly incremented by one, it eventually reaches the value of zero.

At that time PC is incremented by one in order to skip the next instruction in the program.

Since it is not possible to increment a word inside the memory, it is necessary to read the word into DR, increment DR, and store the word back into memory.

This is done with the following sequence of micro operations:

D_6T_4: DR ← M [AR]
D_6T_5: DR ← DR + 1
D_6T_6: M[AR] ← DR, if (DR = 0) then (PC ← PC + 1), SC ← 0

Control Flowchart
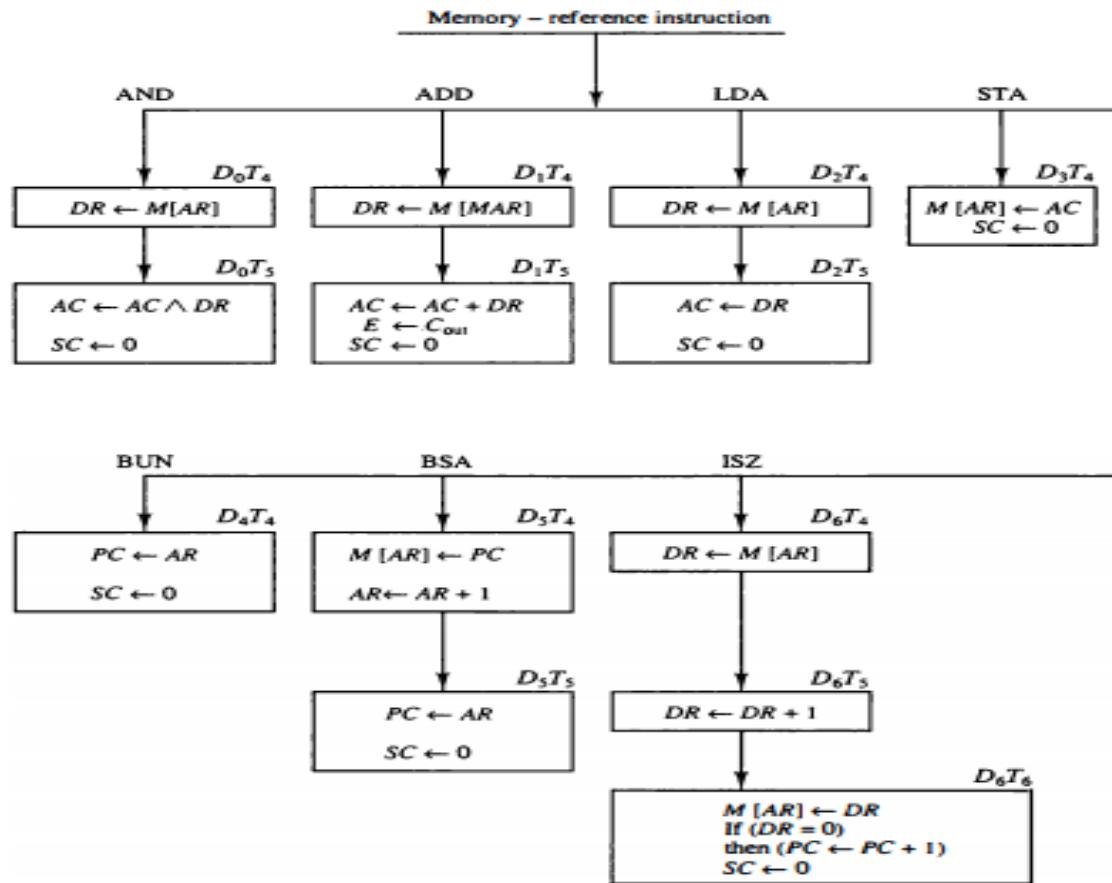
Figure Flowchart for memory-reference instructions.

The control functions are indicated on top of each box.

The micro operations that are performed during time T_4, T_5, or T_6 depend on the operation code value.

This is indicated in the flowchart by six different paths, one of which the control takes after the instruction is decoded.

The sequence counter SC is cleared to 0 with the last timing signal in each case.

This causes a transfer of control to timing signal T_0 to start the next instruction cycle.

Note that we need only seven timing signals to execute the longest instruction (ISZ).

Input-Output

A computer can serve no useful purpose unless it communicates with the external environment.

To demonstrate the most basic requirements for input and output communication, we will use as an illustration a terminal unit with a keyboard and printer.

Input-Output Configuration

The terminal sends and receives serial information. Each quantity of information has **eight bits of an alphanumeric code.**

The serial information from the keyboard is shifted into the input register INPR.

The serial information for the printer is stored in the output register OUTR.

These two registers communicate with a communication interface serially and with the AC in parallel.
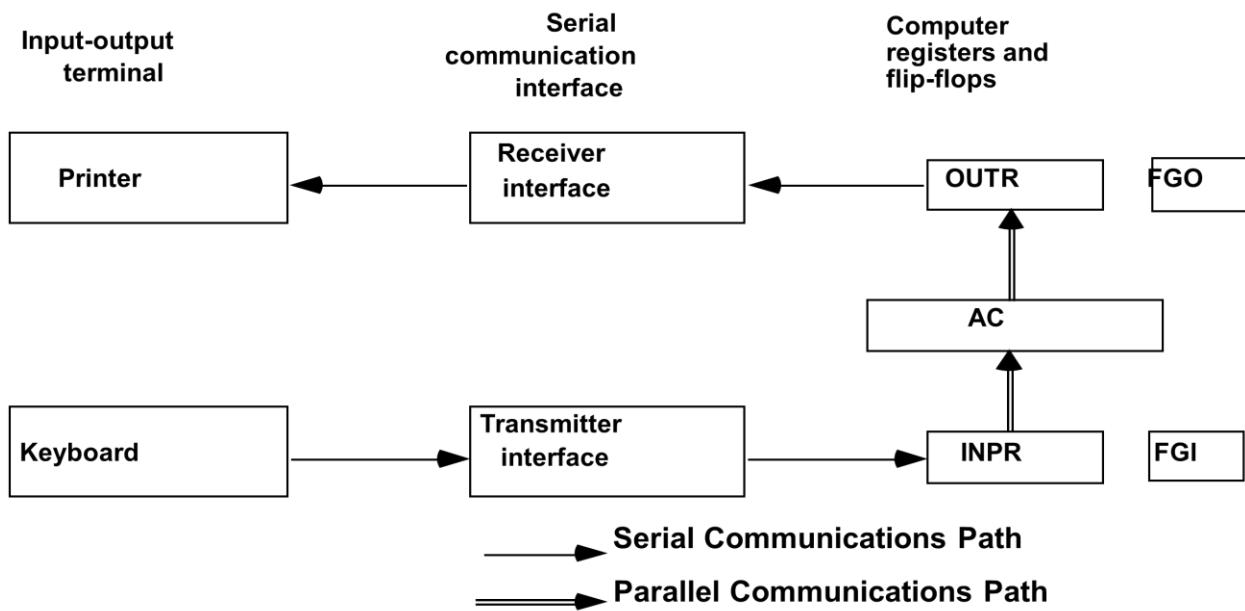
- The input-output configuration is shown in Fig. below.

- The transmitter interface receives serial information from the keyboard and transmits it to INPR.

- The receiver interface receives information from OUTR and sends it to the printer serially.

- The input register INPR consists of eight bits and holds alphanumeric input information.

The 1-bit input flag FGI is a control flip-flop.

The flag bit is set to 1 when new information is available in the input device and is cleared to 0 when the information is accepted by the computer.

The flag is needed to synchronize the timing rate difference between the input device and the computer.

Input –output Configuration

INPR Input register - 8 bits
OUTR Output register - 8 bits
FGI   Input flag - 1 bit
FGO Output flag - 1 bit
IEN   Interrupt enable - 1 bit

The process of information transfer is as follows. Initially, the input flag FGI is cleared to 0.

When a key is struck in the keyboard, an 8-bit alphanumeric code is shifted into INPR and the input flag FGI is set to 1.

As long as the flag is set, the information in INPR cannot be changed by striking another key.

The computer checks the flag bit FGI; if FGI= 1, the information from INPR is transferred in parallel into AC and FGI is cleared to 0.

Once the flag is cleared, new information can be shifted into INPR by striking another key.

The output register OUTR works similarly but the direction of information flow is reversed. Initially, the output flag FGO is set to 1.

The computer checks the flag bit FGO; if FGO=1, the information from AC is transferred in parallel to OUTR and FGO is cleared to 0.

The output device accepts the coded information, prints the corresponding character, and when the operation is completed, it sets FGO to 1.

The computer does not load a new character into OUTR when FGO is 0 because this condition indicates that the output device is in the process of printing the character.

Input-Output Instructions (D_7IT_3):

Input and output instructions are needed for **transferring information to and from AC register, for checking the flag bits, and for controlling the interrupt facility.**

Input-output instructions have an operation code 1111 and are recognized by the control when $D_7 = 1$ and $I = 1$. The remaining bits of the instruction specify the particular operation.

The control functions and micro operations for the input-output instructions are listed in Table below.

These instructions are executed with the clock transition associated with timing signal T_3.

Each control function needs a Boolean relation D_7IT_3, which we designate for convenience by the symbol **"p".**

The control function is distinguished by one of the bits **in IR (6-11).**

By assigning the symbol B_i to bit i of IR, all control functions can be denoted by **"pB_i "**for i = 6 though 11.

The sequence counter SC is cleared to 0 when p = D_7IT_3 = 1.

Input-output Instructions

$$D_7IT_3 = p$$
$$IR(i) = B_i, \; i = 6, \ldots, 11$$

	p:	SC ← 0	Clear SC
INP	pB_{11}:	AC(0-7) ← INPR, FGI ← 0	Input char. to AC
OUT	pB_{10}:	OUTR ← AC(0-7), FGO ← 0	Output char. from AC
SKI	pB_9:	if(FGI = 1) then (PC ← PC + 1)	Skip on input flag
SKO	pB_8:	if(FGO = 1) then (PC ← PC + 1)	Skip on output flag
ION	pB_7:	IEN ← 1	Interrupt enable on
IOF	pB_6:	IEN ← 0	Interrupt enable off

The INP instruction transfers the input information from lNPR into the eight low-order bits of AC and also clears the input flag to 0.

The OUT instruction transfers the eight least significant bits of AC into the output registers OUTR and clears the output flag to 0.

The next two instructions in Table above **check the status of the flags** and cause a skip of the next instruction if the flag is 1.

The instruction that is skipped will normally be a branch instruction to return and check the flag again.

The branch instruction is not skipped if the flag is 0. If the flag is 1, the branch instruction is skipped and an input or output instruction is executed.

The last two instructions set and clear an interrupt enable flip flop IEN. The purpose of IEN is explained in conjunction with the interrupt operation

Interrupt cycle

The interrupt cycle is an hardware implementation of a branch and save return address.

The return address is available in PC is stored in a specific location where it can be found later when the program returns to the instruction at which it was interrupted.
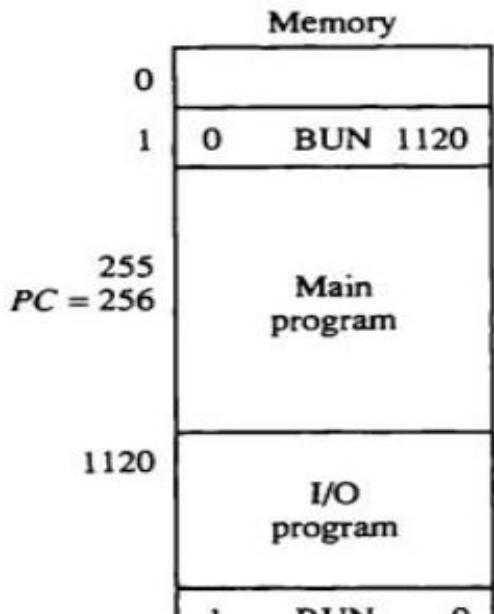
This location can be a processor register or a memory stack or a specific memory location.

Here we choose the **memory location to be 0 as the place for storing the return address;**
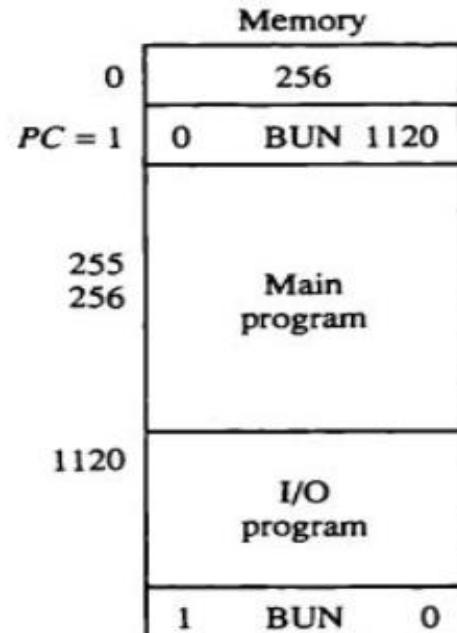
Control then inserts address 1 into PC and clears IEN and R so that no more interrupts can occur until the interrupt request from the flag has been received.

An example that shows what happens during the interrupt cycle is shown below

Demonstration of the interrupt cycle

(a) Before interrupt   (b) After interrupt cycle

Suppose if an interrupt has occurred then R is set to 1 while the control is executing the instruction at address 255.

At this time, the return address **256** is in PC.

The programme has previously placed an input-output service program in memory starting from address 1120 and a BUN 1120 instruction at address 1.

When control reaches timing signal T0 and finds that R=1, it proceeds with the interrupt cycle.

The content of PC (256) is stored in memory location 0, PC is set to 1, and R is cleared to 0.

At the beginning of the next instruction cycle, the instruction that is read from memory is in address 1, since this is the content of PC.

The **branch instruction at address1** causes the **program to transfer to the input –output service program at address 1120.**

This program checks the flags, determines which flag is set, and then transfers the required input or output information.

Once this is done, the instruction ION is executed to set IEN to 1 & the program returns to the location where it was interrupted.

The instruction that returns the computer to the original place in the main program is a **branch indirect instruction with an address part of 0**.

This instruction is placed at the I/O service program.

After this instruction is read from memory during the fetch phase, control goes to the indirect phase to read the effective address.

The effective address is in location 0 and is the return address that was stored there during the previous interrupt cycle.

The execution of the indirect BUN instruction results in placing into PC the return address from location 0.

Flow chart for Interrupt cycle
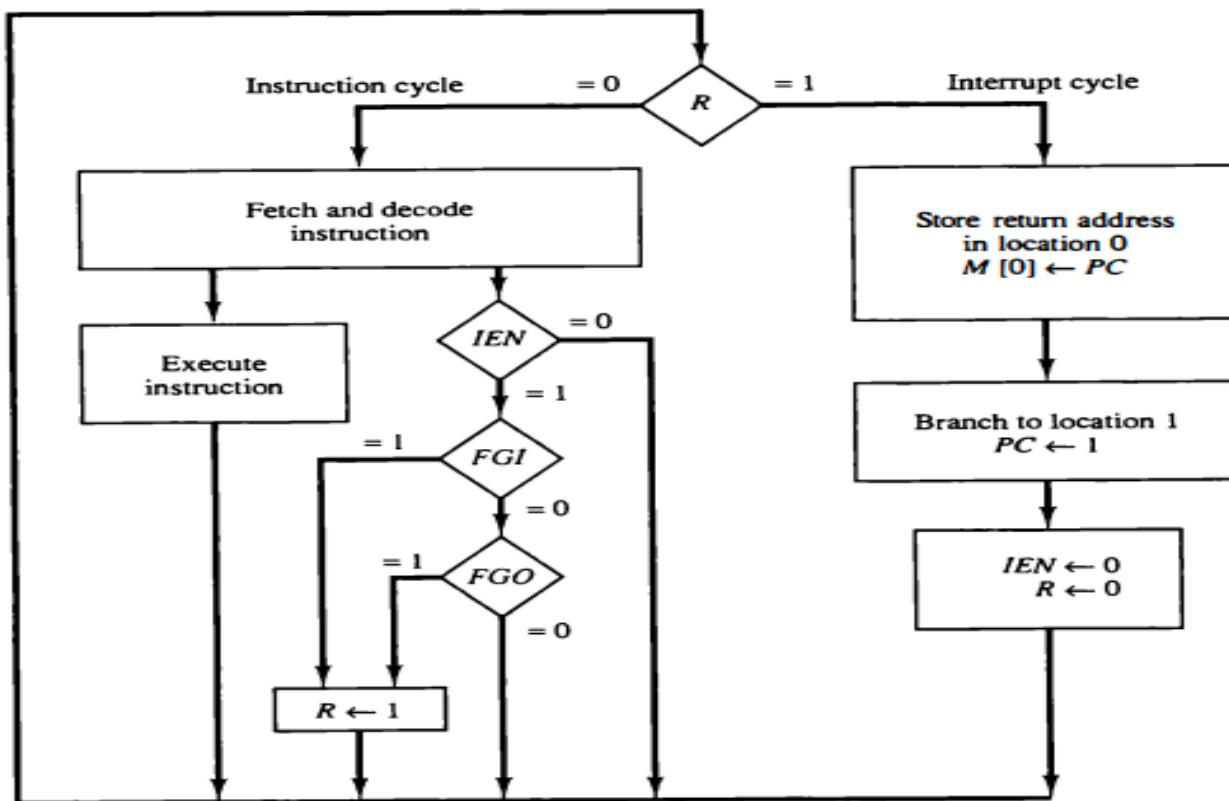
Figure Flowchart for interrupt cycle.

The way the interrupt is handled by the computer can be explained by means of the flow chart.

An **interrupt flip flop R** is included in the computer. When **R=0**; computer goes through an instruction cycle.

During the execution phase of the instruction cycle, **IEN** is checked by the control.

If it is 0 (IEN=0); it indicates the programmer does not want to use the interrupt. So control continues with the next instruction cycle.

If IEN=1; the control checks the flag bits (FGI &FGO) .

If both flags indicate 0 (**FGI=0 &FGO=0**); it indicates that a neither the input nor the output registers are ready for transfer of information

In this case, control continues with the next instruction cycle.

If either flag (FGI or FGO) is set to1 while IEN=1; flipflop R is set to 1.

At the end of the execution phase, control checks the value of R,& if is equal to 1 **(R=1)** it goes to an interrupt cycle.

Interrupt Cycle (Register Transfer Notation)

The list of Register transfer operations in interrupt cycle is given below

The interrupt cycle is initiated after the last execute phase if the interrupt flip-flop R is equal to 1.

This flip-flop (R) is set to 1 if IEN=1 and either the FGI or FGO are equal to 1. This can happen with any clock transition except when timing signals T0, T1, T2 are active.

The condition for setting flipflop R to 1 can be expressed with the following register transfer statement

$$T0'T1'T2' \ (IEN) \ (FGI + FGO): R \leftarrow 1$$

The symbol + between FGI and FGO in the control function designate a logic OR operation. This is ANDed with IEN and T0'T1'T2'.

Modified fetch phase

The fetch and decode phases of the instruction cycle must be modified: Replace T0, T1, T2 with **R'T0, R'T1, R'T2.**

The reason for this is that after the instruction is executed and SC is cleared to 0, the control will go through a fetch phase only if R=0.

If R=1, the control will go through interrupt cycle.

The interrupt cycle stores the return address (PC) into memory location 0, branches to memory location 1, clears IEN,R and SC to 0.

This can be done with following sequence of micro operations:

$$RT0: AR \leftarrow 0, TR \leftarrow PC$$

$$RT1: M[AR] \leftarrow TR, PC \leftarrow 0$$

$$RT2: PC \leftarrow PC + 1, IEN \leftarrow 0, R \leftarrow 0, SC \leftarrow 0$$

During the first timing signal AR is cleared to 0, the content of PC is transferred to the temporary register TR.

With the second timing signal, the return address is stored in memory at location 0 and PC is cleared to 0.

The third timing signal increments PC to1, clears IEN and R and control goes back to T0 by clearing SC to 0.

The beginning of the next instruction cycle has the condition RT0 and content of PC=1.

The control then goes through an instruction cycle that fetches and executes the BUN instruction in location 1.

Micro programmed Control

Control Unit:

The main function of control unit is to initiate sequences of micro operations. The number of micro operations in the systems is finite.

 Two major types of Control Unit:

1. Hardwired Control:

When the **control signals are generated by Hardware using conventional logic design techniques** then the control unit is said to be **hardwired.**

The control logic is implemented with gates, F/Fs, decoders, and other digital circuits

The key characteristics are

- High speed of operation
- Expensive
- Relatively complex
- No flexibility of adding new instructions (Wiring change-if the design has to be modified)

Examples of CPU with hardwired control unit are Intel 8085, Motorola 6802, Zilog 80, and any RISC CPUs.

2. Microprogrammed Control:

The control information is stored in a **control memory**, and

The control memory is programmed to initiate the required sequence of micro operations

Any required change can be done by updating the micro program in control memory, - Slow operation

The key characteristics are

- Speed of operation is low when compared with hardwired
- Less complex
- Less expensive
- Flexibility to add new instructions

Examples of CPU with micro programmed control unit are Intel 8080, Motorola 68000 and any CISC CPUs.

The control function that **specifies a micro operation is a binary variable.**

When it is in one state the corresponding micro operation is executed. The opposite state does not change the state of registers.

Control signal (that specify microoperations) in a bus-organized system are: **A groups of bits that select the paths in multiplexers, decoders, and arithmetic logic units**

Control unit initiates a series of micro operations. During any time certain micro operations are initiated while others are idle.

Control Word:

The control variables (specifying a micro operation) at any given time can be represented **by a string of 1's and 0's** is called **"control word".**

Microprogrammed Control Unit :

A control unit whose **binary control variables are stored in memory** (control memory)

Microinstruction

The microinstruction specifies one or more microoperations for the system.

Microprogram

A sequence of microinstruction

Control Memory:

A Memory that is a part of control unit. The control unit consists of **control memory used to store the micro program.** Control memory is a permanent i.e., read only memory (ROM).

A computer having a Micro programmed Control Unit has 2 separate Memories :

1. Main Memory: For storing user program (Machine instructions/data) . The contents of the main memory may alter.

2. Control Memory: For storing microprogram that can not be altered.

The **Microprogram c**onsists of microinstructions.

Microinstruction specifies various control signals for execution of register micro operations.
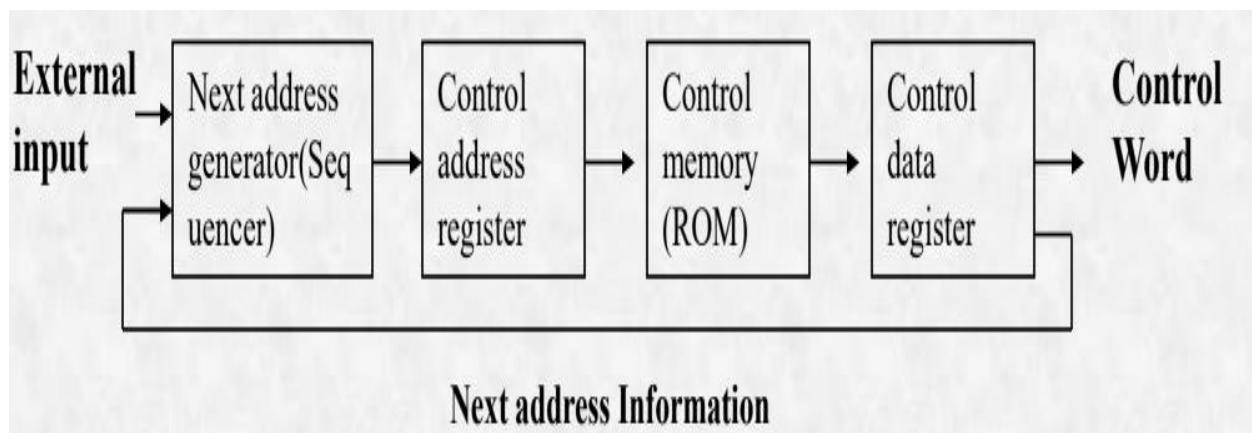
Each machine Instruction initiates a series of Microinstructions in control Memory.

These microinstructions generate the micro operations

- To fetch the instruction from main memory;
- To evaluate the effective address,
- To execute the operation specified by the instruction,
- To return control to the fetch phase in order to repeat the cycle for the next instruction.

Micro programmed Control Organization

The general configuration of a micro programmed control unit is shown below

1. Control Memory (ROM):

- A memory is part of a control unit.

- All the control Information is permanently stored.

2. Control Address Register

- Specify the address of the microinstruction

3. Control Data Register (Pipeline Register)

- Hold the present microinstruction(specifies one or moremicrooperations) read from control memory

- To generate the address of the next microinstruction, some bits of present micro instruction can be used.
- Thus a microinstruction contains bits for initiating the microoperations and bits that determine the address sequence for control memory.

4. Next Address Generator (Sequencer)

- Determine the address sequence that is read from control memory

- Next address of the next microinstruction can be specified several way depending on the sequencer input

Address Sequencing:

Microinstructions are stored in control memory in groups, with each group specifies a **Routine.**

The hardware that controls the address sequencing of the control memory must be **able of sequencing the microinstruction within a routine and be able to branch from one routine to another.**

Fetching:

An initial address is loaded into the control address register when power is turned on. This address is the address of the first microinstruction that activates the fetch routine.

After the end of fetch routine, the instruction is in the instruction register of the computer (Decoding).

The control memory next must go through the routine that **determines the effective address of the operand.** After computing the effective address, the address of the operand is available in the memory address register.

The next step is to generate the micro operations that **execute** the instruction fetched from memory.

The **microoperation** steps to be generated in processor registers depend upon the operation code part of instruction.

Each instruction has its own microprogram routine stored in a given location of the control memory.

The transformation from the instruction code bits to an address in the control memory where the routine is located is called as **Mapping.**

After the execution of the instruction control must return to the fetch routine.

Address Sequencing Capabilities:

1) Incrementing of the control address register

2) Unconditional branch or conditional branch, depending on status bit conditions

3) Mapping process (bits of the instruction address for control memory)

4) A facility for subroutine return

The below figure shows a block diagram of a control memory and the associated hardware needed for selecting the next Micro instruction address
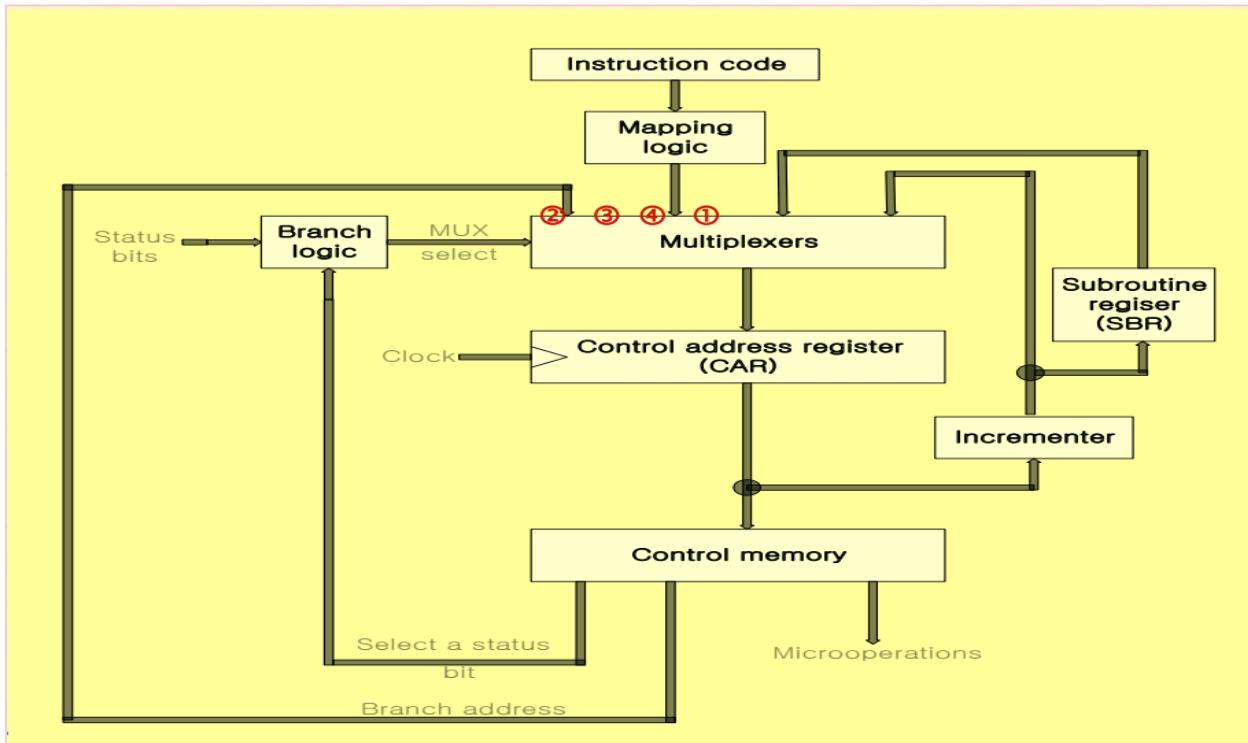
Fig: Selection of Address for Control Memory

Selection of Address for Control Memory

The Micro instruction in control memory contains **set of bits to initiate micro operations in computer registers** and **other bits to specify the method by which the next address is obtained.**

The diagram shows **four different paths** from which the **Control Address Register receives the information.**

1) Incrementer (Increment CAR by 1)

2) Branch address from control memory

3) Mapping Logic (External Address from main memoryto control memory)

4) SBR: Subroutine Register

- Return Address cannot be stored in ROM

- Return Address for a subroutine is stored in SBR

Conditional Branching

Status conditions are special bits in the system that provide parameter information such as carry-out of an adder, sign bit of number, mode bits of an instruction, input /output status condition.

Information in these bits can be tested and actions initiated based on their condition; whether their value is 1 or 0.

The status bits together with the field in the microinstruction that specifies a branch address, control the conditional branch decisions generated in the branch logic.

Branch Logic: It can be implemented in different ways.

The simple way is to **test the specified condition and branch to the indicated address** if the **condition is met**; otherwise the address register is incremented.

This can be implemented with the help of **Multiplexer.**

Example: Let there are eight status bit conditions in the system. Three bits in the microinstruction are used to specify one of eight status bits.

If the selected status bit is in the 1 state the output of the multiplexer is 1, otherwise 0.

The 1 output in the multiplexer generates a control signal **to transfer the branch address from the microinstruction into the control address register** otherwise **address register to be incremented.**

The unconditional branch microinstruction can be implemented by loading the branch address from control memory into control address register.

MAPPING OF INSTRUCTIONS TO MICROROUTINES:

Mapping from the OP-code of an instruction to the address of the microinstruction which is the starting microinstruction of its execution microprogram
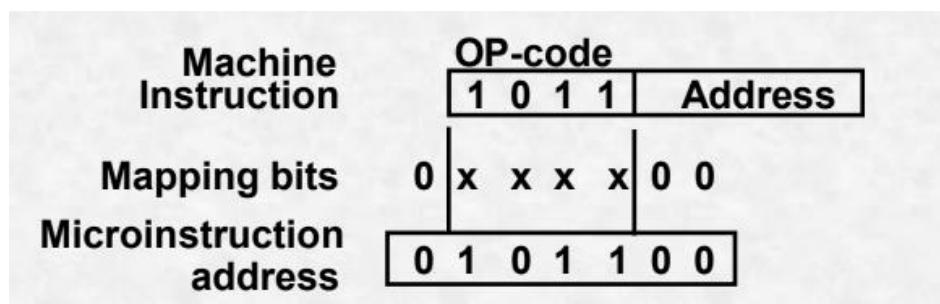
Fig Mapping from instruction code to Microinstruction address

Consider a 4 bit Opcode = specify up to 16 distinct instructions. And assume that control memory has 128 words.

Mapping Process : Converts the 4-bit Opcode to a 7-bit control memory address

       1) Place a "0" in the most significant bit of the address

       2) Transfer 4-bit Operation code bits

       3) Clear the two least significant bits of the CAR

Mapping Function:

Mapping is implemented by ROM or PLD. A PLD (an Integrated Circuit) is similar to ROM except that it uses AND and OR gates with internal electronic fuses.

The interconnection between AND, OR and outputs can be programmed as in ROM

Subroutine:

Subroutines are program that are used by other routines to accomplish a particular task.

A subroutine can be called from any point within the main body of the micro program.

Frequently many microprogram contain identical section of code. Microinstruction can be saved by employing subroutines that used common section of micro-code.

 Ex. The sequence of micro operations needed to generate the effective address of the operand for an instruction is common to all memory reference instructions.

This sequence could be a subroutine that is called from within many other routines to execute the effective address computation.

Microprogram that uses Subroutines must have for storing return address during a subroutine call and restoring the address during a subroutine return.

Microprogram Example:

The process of code generation for the control memory is called *microprogramming.*

The block diagram of the computer configuration is shown in below figure.

Two memory units:

> 1. Main memory – stores instructions and data
>
> 2. Control memory – stores microprogram

 Four processor registers :

1. Program counter – PC

2. Address register – AR

3. Data register – DR

4. Accumulator register - AC

Two control unit registers

1. Control address register – CAR

2. Subroutine register – SBR

Transfer of information among registers in the processor is through MUXs rather than a bus.

Computer Configuration
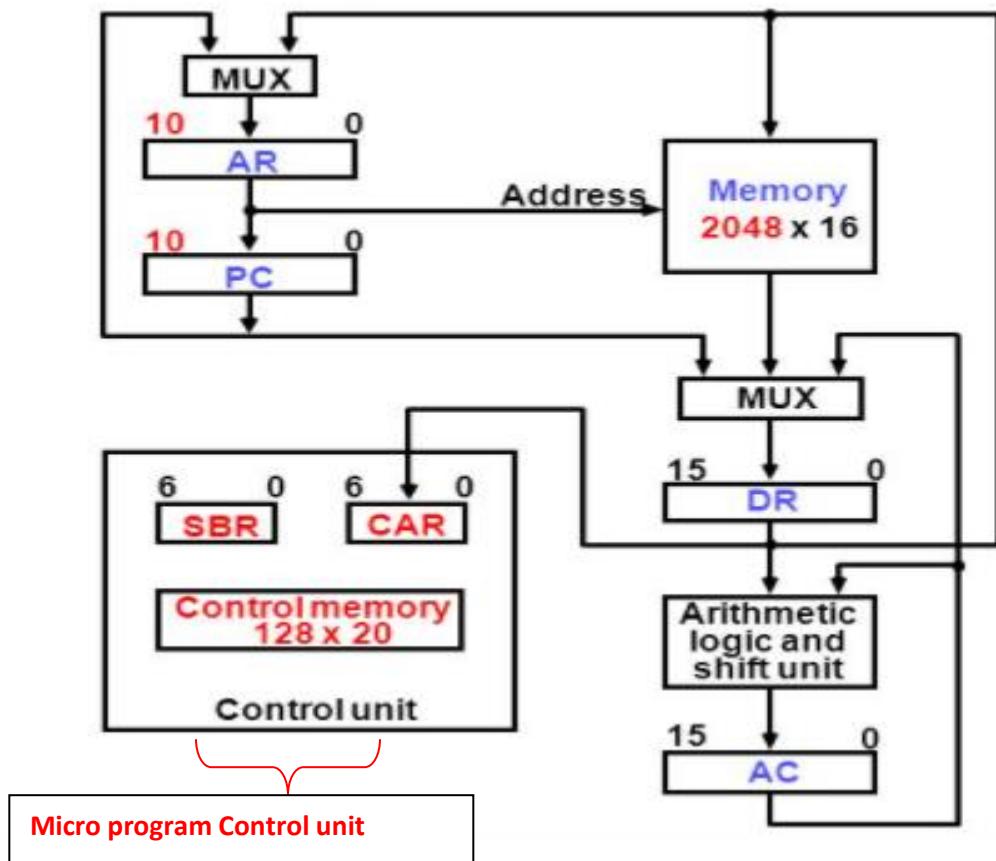
Fig: Computer Hardware configuration

MACHINE INSTRUCTION FORMAT:

Main Memory

2048*16
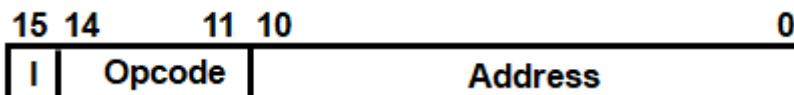
It consists of 3 fields:

1. I bit to denote Direct or Indirect Addressing
2. 4 bit opcode
3. 11 bit Address Field

Machine instruction format=16 BIT   **Address size=11bits**

15 14	11 10	0
I	Opcode	Address

Sample machine instructions

Symbol	OP-code	Description
ADD	0000	AC ← AC + M[EA]
BRANCH	0001	if (AC < 0) then (PC ← EA)
STORE	0010	M[EA] ← AC
EXCHANGE	0011	AC ← M[EA], M[EA] ← AC

EA is the effective address
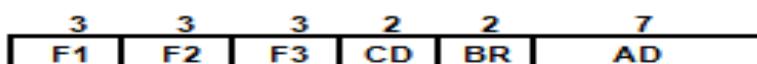
MICRO INSTRUCTION FORMAT:

The Microinstruction format for the control memory is shown

In figure. The 20 bits of the microinstruction are divided into

4 functional parts.

Control Memory

128*20

Microinstruction Format=20 BIT==6 FIELDS

3	3	3	2	2	7
F1	F2	F3	CD	BR	AD

F1, F2, F3: Microoperation fields
CD: Condition for branching
BR: Branch field
AD: Address field

The microinstruction format is composed of 20 bits with four parts to it

> **1. Three fields F1, F2, and F3 specify micro operations for the computer [3 bits each].**
>
> **2. The CD field selects status bit conditions [2 bits]**
>
> **3. The BR field specifies the type of branch to be used [2 bits]**
>
> **4. The AD field contains a branch address [7 bits]**

Each of the three micro operation fields can specify one of seven possibilities. This gives a total of 21 Instructions.

Not more than three micro operations can be chosen for a microinstruction.

 If fewer than three micro operations are used, the next 1 or more fields will use the binary code 000 = NOP.

Each Micro operation in Table is defined with a register transfer statement and is assigned a symbol for symbolic notation.

The three bits in each field are encoded to specify seven distinct microoperations listed in below table.

All transfer type micro operations symbols use five letters. The first 2 letters indicate source register and the third letter is always T, and last 2 letters designate destination register.

MICROINSTRUCTION FIELD DESCRIPTIONS - F1, F2, F3

F1	Microoperation	Symbol
000	None	NOP
001	$AC \leftarrow AC + DR$	ADD
010	$AC \leftarrow 0$	CLRAC
011	$AC \leftarrow AC + 1$	INCAC
100	$AC \leftarrow DR$	DRTAC
101	$AR \leftarrow DR(0\text{-}10)$	DRTAR
110	$AR \leftarrow PC$	PCTAR
111	$M[AR] \leftarrow DR$	WRITE

F2	Microoperation	Symbol
000	None	NOP
001	$AC \leftarrow AC - DR$	SUB
010	$AC \leftarrow AC \vee DR$	OR
011	$AC \leftarrow AC \wedge DR$	AND
100	$DR \leftarrow M[AR]$	READ
101	$DR \leftarrow AC$	ACTDR
110	$DR \leftarrow DR + 1$	INCDR
111	$DR(0\text{-}10) \leftarrow PC$	PCTDR

F3	Microoperation	Symbol
000	None	NOP
001	$AC \leftarrow AC \oplus DR$	XOR
010	$AC \leftarrow AC'$	COM
011	$AC \leftarrow shl\ AC$	SHL
100	$AC \leftarrow shr\ AC$	SHR
101	$PC \leftarrow PC + 1$	INCPC
110	$PC \leftarrow AR$	ARTPC
111	Reserved	

Table: Symbols and Binary code for MicroInstruction Fields

MICROINSTRUCTION FIELD DESCRIPTIONS - CD, BR

The condition field (CD) is two bits to specify four status bit conditions shown below

CD	Condition	Symbol	Comments
00	Always = 1	U	Unconditional branch
01	DR(15)	I	Indirect address bit
10	AC(15)	S	Sign bit of AC
11	AC = 0	Z	Zero value in AC

The branch field (BR) consists of two bits and is used with the address field to choose the. address of the next microinstruction.

BR	Symbol	Function
00	JMP	CAR ← AD if condition = 1
		CAR ← CAR + 1 if condition = 0
01	CALL	CAR ← AD, SBR ← CAR + 1 if condition = 1
		CAR ← CAR + 1 if condition = 0
10	RET	CAR ← SBR (Return from subroutine)
11	MAP	CAR(2-5) ← DR(11-14), CAR(0,1,6) ← 0

SYMBOLIC MICROINSTRUCTIONS

Symbols are used in microinstructions as in assembly language

A symbolic microprogram can be translated into its binary equivalent by a microprogram assembler.

Each symbolic microinstruction is divided into 5 fields:

Label, Micro operations, CD, BR, and AD.

Sample Format
Five fields: label; micro-ops; CD; BR; AD

Label: May be empty or may specify a symbolic address terminated with a colon

Micro-ops: consists of one, two, or three symbols separated by commas

CD: one of {U, I, S, Z}, where   U: Unconditional Branch
                                               I: Indirect address bit
                                               S: Sign of AC
                                               Z: Zero value in AC

BR: one of {JMP, CALL, RET, MAP}

AD: one of {Symbolic address, NEXT, empty}

SYMBOLIC MICROPROGRAM - FETCH ROUTINE

During FETCH, Read an instruction from memory and decode the instruction and update PC

Sequence of microoperations in the fetch cycle:

```
AR ← PC
DR ← M[AR], PC ← PC + 1
AR ← DR(0-10), CAR(2-5) ← DR(11-14), CAR(0,1,6) ← 0
```

Symbolic microprogram for the fetch cycle:

```
              ORG 64
FETCH:        PCTAR         U  JMP  NEXT
              READ, INCPC   U  JMP  NEXT
        →     DRTAR         U  MAP
```

Binary equivalents translated by an assembler

Binary address	F1	F2	F3	CD	BR	AD
1000000	110	000	000	00	00	1000001
1000001	000	100	101	00	00	1000010
1000010	101	000	000	00	11	0000000

• Control Storage: 128 20-bit words
• The first 64 words: Routines for the 16 machine instructions
• The last 64 words: Used for other purpose (e.g., fetch routine and other subroutines)
• Mapping: OP-code XXXX into 0XXXX00, the first address for the 16 routines are
                     0(0 0000 00), 4(0 0001 00), 8, 12, 16, 20, ..., 60

Partial Symbolic Microprogram

Label	Microops	CD	BR	AD
	ORG 0			
ADD:	NOP	I	CALL	INDRCT
	READ	U	JMP	NEXT
	ADD	U	JMP	FETCH
	ORG 4			
BRANCH:	NOP	S	JMP	OVER
	NOP	U	JMP	FETCH
OVER:	NOP	I	CALL	INDRCT
	ARTPC	U	JMP	FETCH
	ORG 8			
STORE:	NOP	I	CALL	INDRCT
	ACTDR	U	JMP	NEXT
	WRITE	U	JMP	FETCH
	ORG 12			
EXCHANGE:	NOP	I	CALL	INDRCT
	READ	U	JMP	NEXT
	ACTDR, DRTAC	U	JMP	NEXT
	WRITE	U	JMP	FETCH
	ORG 64			
FETCH:	PCTAR	U	JMP	NEXT
	READ, INCPC	U	JMP	NEXT
	DRTAR	U	MAP	
INDRCT:	READ	U	JMP	NEXT
	DRTAR	U	RET	

126

SBR= CAR+1=0000001
CAR= 0000001   DR(15)=1

BINARY MICROPROGRAM

Micro Routine	Address Decimal	Address Binary	F1	F2	F3	CD	BR	AD
ADD	0	0000000	000	000	000	01	01	1000011
	1	0000001	000	100	000	00	00	0000010
	2	0000010	001	000	000	00	00	1000000
	3	0000011	000	000	000	00	00	1000000
BRANCH	4	0000100	000	000	000	10	00	0000110
	5	0000101	000	000	000	00	00	1000000
	6	0000110	000	000	000	01	01	1000011
	7	0000111	000	000	110	00	00	1000000
STORE	8	0001000	000	000	000	01	01	1000011
	9	0001001	000	101	000	00	00	0001010
	10	0001010	111	000	000	00	00	1000000
	11	0001011	000	000	000	00	00	1000000
EXCHANGE	12	0001100	000	000	000	01	01	1000011
	13	0001101	001	000	000	00	00	0001110
	14	0001110	100	101	000	00	00	0001111
	15	0001111	111	000	000	00	00	1000000
FETCH	64	1000000	110	000	000	00	00	1000001
	65	1000001	000	100	101	00	00	1000010
	66	1000010	101	000	000	00	11	0000000
INDRCT	67	1000011	000	100	000	00	00	1000100
	68	1000100	101	000	000	00	10	0000000

This microprogram can be implemented using ROM

Design of Control Unit

The bits of microinstruction are usually divided into fields, with each field defining a distinct, separate function.

The various fields available in the instruction format provide control bits to initiate the microoperation.

Special bits(status bits) are used to specify the way that the next address is to be evaluated and an address field for branching.

Decoding of Microinstruction Fields :

– F1, F2, and F3 of Microinstruction are decoded with a 3 x 8 decoder

– Output of decoder must be connected to the proper circuit to initiate the corresponding microoperation .
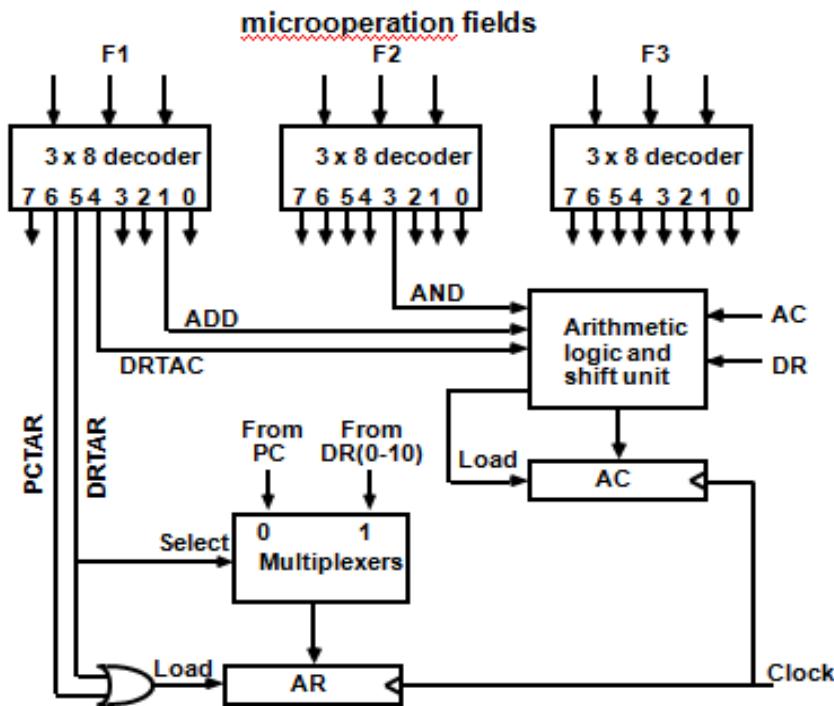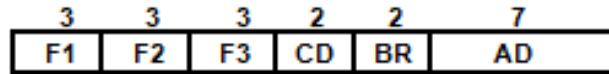
Fig Decoding of Micro operation fields

When F1 = 101 (binary 5), the next pulse transition transfers the content of DR (0-10) to AR.

Similarly, when F1= 110 (binary 6) there is a transfer from PC to AR (symbolized by PCTAR).

As shown in figure, outputs 5 and 6 of decoder *F1 are connected to the load input of AR so that when either one of these outputs is active, information from the multiplexers is transferred to AR.*

The multiplexers select the information from DR when output 5 is active and from PC when output 5 is inactive.

 The transfer into AR occurs with a clock transition only when output 5 or output 6 of the decoder is active.

For the arithmetic logic shift unit the control signals are instead of coming from the logical gates, now these inputs will now come from the outputs of AND, ADD and DRTAC respectively.

Microprogram Sequencer:

- The basic components of a microprogrammed control unit are the **control memory** and the **circuits that select the next address.**

- The address selection part is called a **microprogram sequencer.**

- The purpose of a microprogram sequencer is to present an address to the control memory so that a microinstruction may be read and executed.

- The next-address logic of the sequencer determines the specific address source to be loaded into the control address register.

- The block diagram of the microprogram sequencer is shown in below figure.
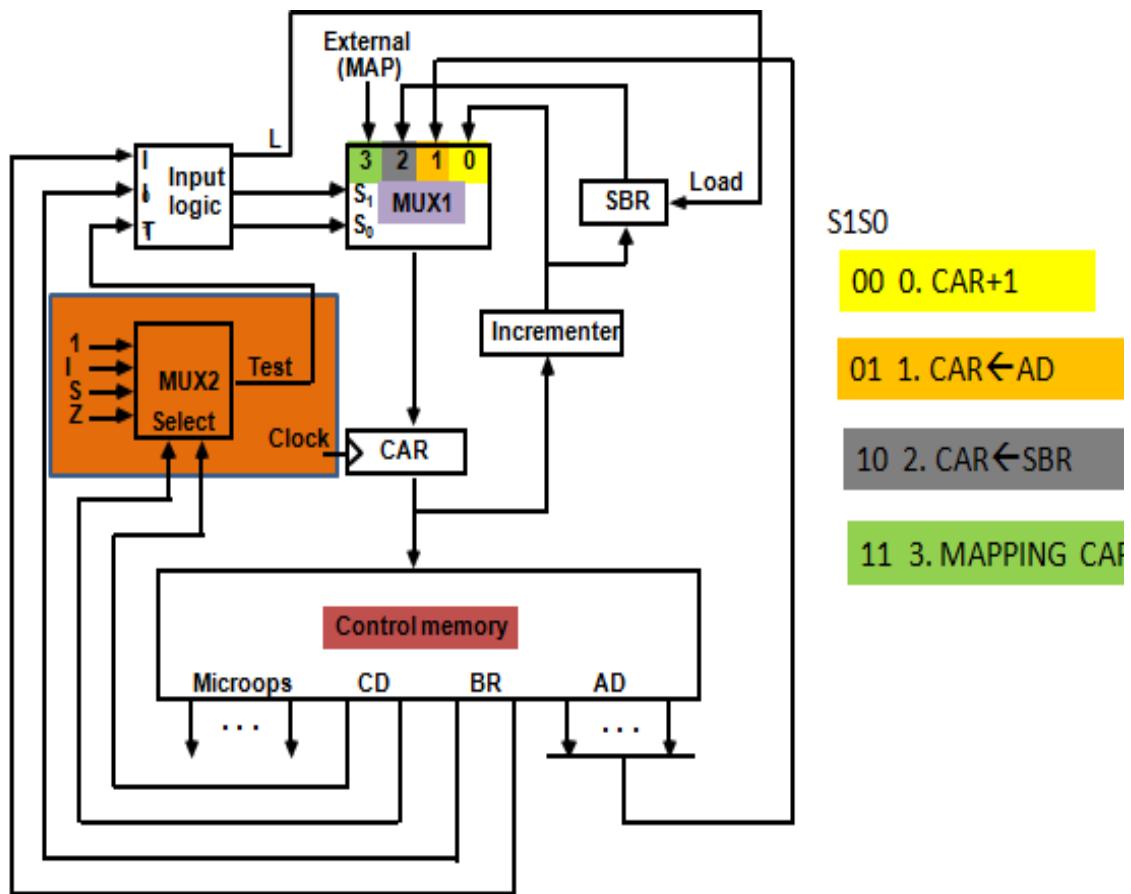
Fig Microprogram sequencer for Control Memory

There are two multiplexers in the circuit.

1The first multiplexer selects an address from one of four sources and routes it into control address register *CAR*.

2. The second multiplexer tests the value of a **selected status bit** and   the result of the test is applied to an input logic circuit.

The output from CAR provides the address for the control memory.

The content of CAR is incremented and applied to one of the multiplexer inputs and to the subroutine registers *SBR*.

The other three inputs to multiplexer come from

1. The address field of the present microinstruction

2. from the out of SBR

3. From an external source that maps the instruction

The CD (condition) field of the microinstruction selects one of the status bits in the second multiplexer.

If the bit selected is equal to 1, the T variable is equal to 1; otherwise, it is equal to 0.

The *T value together with two bits from the BR (branch) field goes to an input logic circuit.*
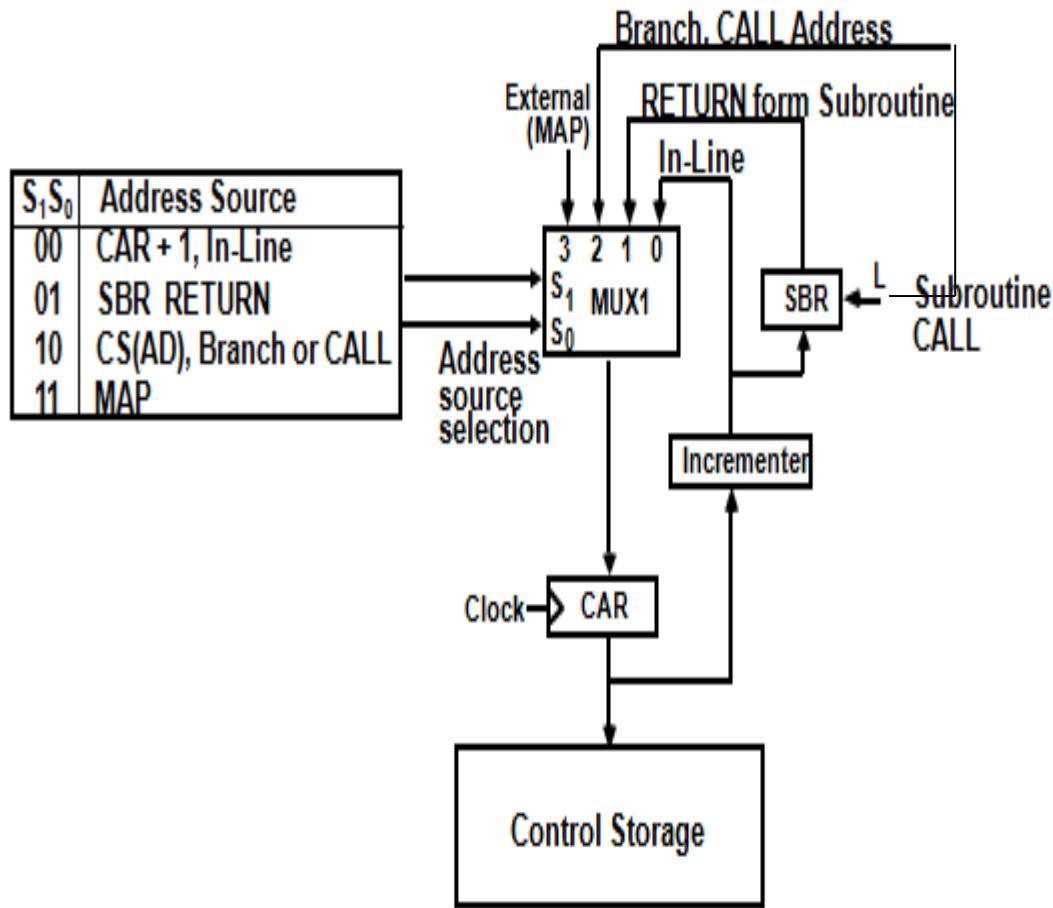
The input logic in a particular sequencer will determine the type of operations that are available in the unit.

MICROINSTRUCTION FIELD DESCRIPTIONS - CD, BR

CD	Condition	Symbol	Comments
00	Always = 1	U	Unconditional branch
01	DR(15)	I	Indirect address bit
10	AC(15)	S	Sign bit of AC
11	AC = 0	Z	Zero value in AC

| BR	Symbol	Function
00	JMP	CAR ← AD if condition = 1
		CAR ← CAR + 1 if condition = 0
01	CALL	CAR ← AD, SBR ← CAR + 1 if condition = 1
		CAR ← CAR + 1 if condition = 0
10	RET	CAR ← SBR (Return from subroutine)
11	MAP	CAR(2-5) ← DR(11-14), CAR(0,1,6) ← 0
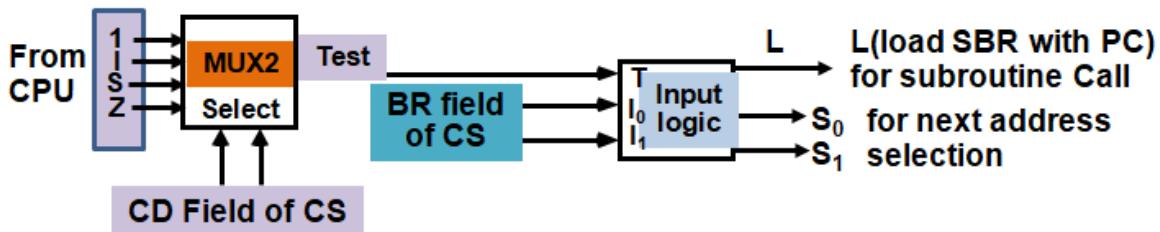
MICROPROGRAM SEQUENCER -NEXT MICROINSTRUCTION ADDRESS LOGIC

MUX-1 selects an address from one of four sources and routes it into a CAR

- In-Line Sequencing $\rightarrow$ CAR + 1

- Branch, Subroutine Call $\rightarrow$ CS(AD)

- Return from Subroutine $\rightarrow$ Output of SBR

- New Machine instruction $\rightarrow$ MAP

MICROPROGRAMSEQUENCER-CONDITION AND BRANCH CONTROL

Input Logic

I_0I_1T	Meaning	Source of Address	S_1S_0	L
000	In-Line	CAR+1	00	0
001	JMP	CAR←(AD)	10	0
010	In-Line	CAR+1	00	0
011	CALL	CS(AD) and SBR <- CAR+1	10	1
10x	RET	SBR	01	0
11x	MAP	DR(11-14)	11	0

$$S_0 = I_0$$
$$S_1 = I_0I_1 + I_0'T$$
$$L = I_0'I_1T$$

CD	Condition	Symbol	Comments
00	Always = 1	U	Unconditional branch
01	DR(15)	I	Indirect address bit
10	AC(15)	S	Sign bit of AC
11	AC = 0	Z	Zero value in AC

BR	Symbol	Function
00	JMP	CAR ← AD if condition = 1
		CAR ← CAR + 1 if condition = 0
01	CALL	CAR ← AD, SBR ← CAR + 1 if condition = 1
		CAR ← CAR + 1 if condition = 0
10	RET	CAR ← SBR (Return from subroutine)
11	MAP	CAR(2-5) ← DR(11-14), CAR(0,1,6) ← 0

UNIT-II

Introduction to basic concepts:

Important Terminology used in Microprocessor

The unit of data size can be represented as:

1. Bit: A binary digit that can have the value 0 or 1

2. Byte: Group of 8 bits

3. Nibble: Half of a byte, or group of 4 bits

4. Word: Two bytes or group of16 bits ‰

The terms used to describe the amounts of memory in IBM PCs and compatibles:

1. Kilobyte (Kb): 2^{10}bits

2. Megabyte (Mb): 2^{20}bits over 1 million

3. Gigabyte (Gb) : 2^{30}bits, over 1 billion

4. Terabyte (Tb) : 2^{40}bits, over 1 trillion

Number Representation Techniques:

- Binary system

- Octal system

- Decimal system

- Hexadecimal system

Hexadecimal system

- It is one of the type of Number Representation techniques, in which there value of base is 16.

- Hexadecimal Number System is commonly used in Computer programming and Microprocessors.

- It is used to describe locations in memory for every byte.

- The main advantage of using Hexadecimal numbers is that **it uses less memory to store more numbers.**

Examples:

1. To represent a binary number as its equivalent hexadecimal number

- Start from the right and group 4 bits at a time, replacing each 4-bit binary number with its hex equivalent

Ex. Represent binary 100111110101 in hex

|	1001	1111	0101
=	9	F	5
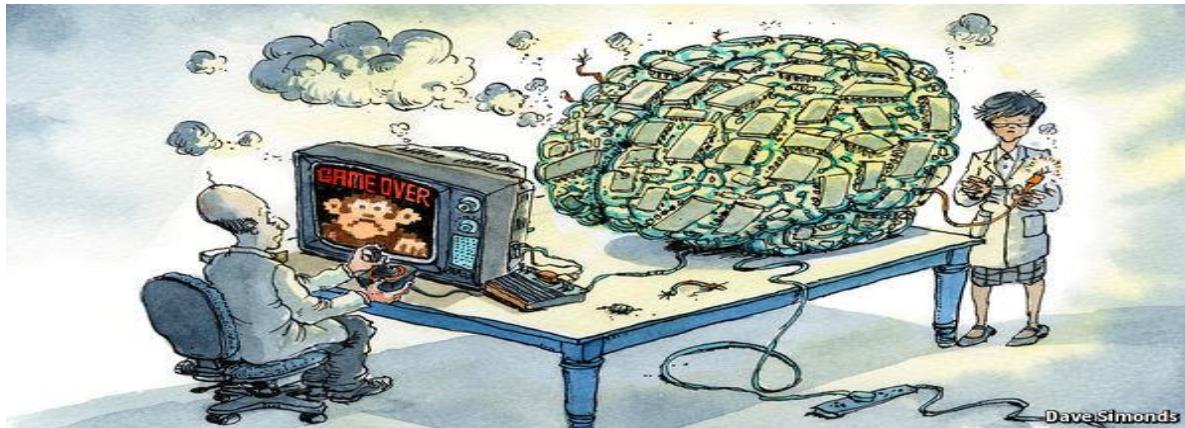
2. To represent a hexadecimal number as its binary equivalent number

- Start from the right and group 4 bits at a time, replacing each 4-bit binary number with its hex equivalent

Ex. Convert hex 29B to binary

	2	9	B
=	0010	1001	1011

➢ Microprocessor is Brain of the systems

What is a Microprocessor?

- The word comes from the combination micro and processor.

- Processor means a device that processes numbers, specifically binary numbers, 0's and 1's.

- In the late 1960's, processors performed the required operation, but were too large and too slow.

- In the early 1970's the microchip was invented. All of the components that made up the processor were now placed on a single piece of silicon.

- The size became several thousand times smaller and the speed became several hundred times faster.

- The "Micro" Processor was born.

Definition of Microprocessor

- The Microprocessor is a programmable IC that performs arithmetic or logical operations according to the program and produces the output results.

Or

- Microprocessor is a multipurpose, programmable device that accepts digital data as input, processes it according to instructions stored in its memory, and provides results as output.

Or

- A microprocessor is a computer processor which incorporates the functions of a computer's central processing unit (CPU) on a single integrated circuit (IC)

Evolution of Microprocessors

Processor	No. of bits	Clock speed (Hz)	Year of introduction
4004	4	740K	1971
8008	8	500K	1972
8080	8	2M	1974
8085	8	3M	1976
8086	16	5, 8 or 10M	1978
8088	16	5, 8 or 10M	1979
80186	16	6M	1982
80286	16	8M	1982
80386	32	16 to 33M	1986
80486	32	16 to 100M	1989
Pentium	32	66M	1993
Pentium II	32	233 to 500M	1997
Pentium III	32	500M to 1.4G	1999
Pentium IV	32	1.3 to 3.8G	2000
Dual core	32	1.2 to 3 G	2006
Core 2 Duo	64	1.2 to 3G	2006
i3, i5 and i7	64	2.4G to 3.6G	2010

➢ Latest one is the Intel i9 core processor

8086 Microprocessor
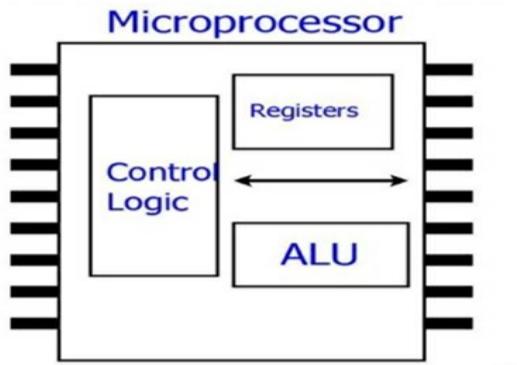
Intel 8086 microprocessor is the enhanced version of Intel 8085 microprocessor. **It was designed by Intel in 1976.**

The 8086 microprocessor is a16-bit, N-channel, HMOS microprocessor. Where the HMOS is used for "**High-speed Metal Oxide Semiconductor**".

It is a 40 pin, Dual Inline Packaged IC. It has 16 data lines and 20 address lines thus it can able to access 2^{20} i.e. 1 Mb address in the memory

It is able to perform the operation with 16 bit data in one cycle i.e it can carry 16 bit data at a time.8086 provides the programmer with 14 internal registers, each 16 bits or 2 Bytes wide.

Components of Microprocessor

A microprocessor consists of an **ALU, Control unit and Register array**

ALU performs arithmetic and logical operations on the data received from an input device or memory.

Control unit controls the instructions and flow of data within the computer.

Registers are used for temporary storage of data, instructions, addresses during execution of a program.

System Bus: The memory and i/o ports are interconnected to microprocessor through a **System bus.**

A **Bus** is a group of conducting wires which carries information; all the peripherals are connected to microprocessor through Bus.

➢ There are three types of buses:

1. Address bus

2. Data bus

3. Control bus
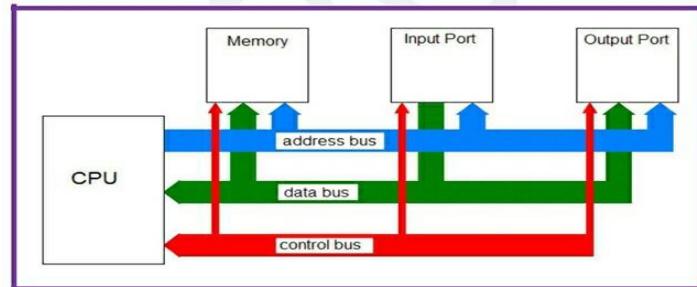
Figure 1 Block diagram of Computer

- The **control bus** carries the control, timing and coordination signals to manage the various functions across the system.
- The **address bus** is used to specify memory locations for the data being transferred.
- The **data bus**, which is a bidirectional path, carries the actual data between the processor, the memory and the peripherals.

Working Principle of Microprocessor:

The microprocessor follows a sequence to execute the instruction:

1. Fetch

2. Decode

3. Execute
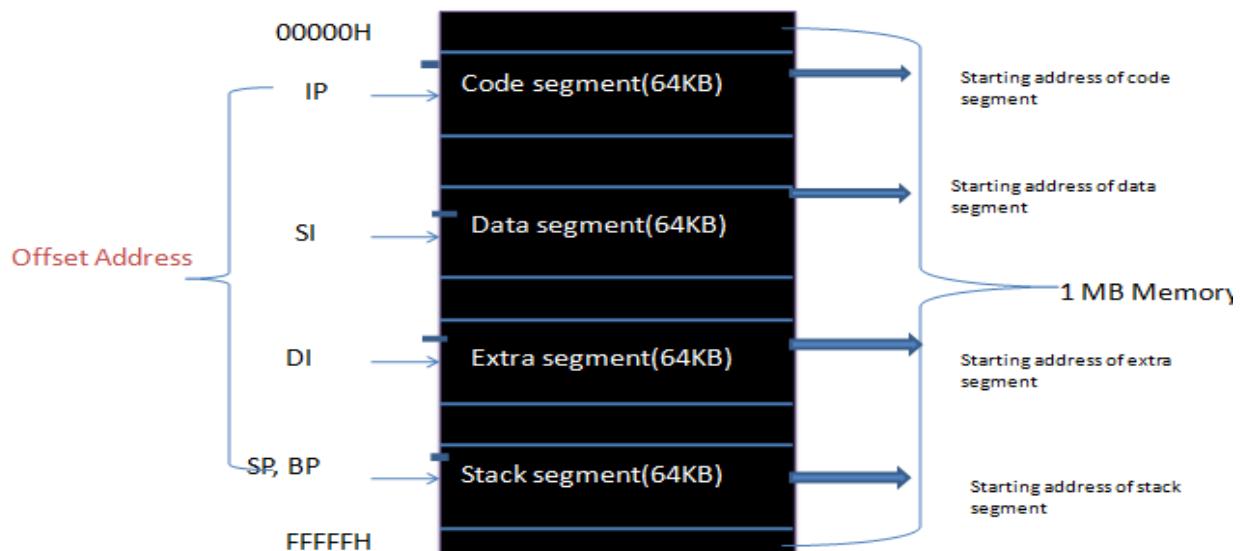
Memory Segmentation

- Segmentation is the **process of dividing.**
- The available memory space is divided into "chunks" called segments. Such a memory is known as segmented memory and each segment has its own base address.
- It is basically used to enhance the speed of execution, so that the processor is able to fetch and execute the data from the memory easily and fast.

Need for Segmentation:

- **To increase execution speed and fetching speed**, 8086 segments the memory.
- 1MB of memory is segmented into 4 64kB segments.
- Each segment has its own address or starting address and within the segment it has the offset address.
- 8086 works only with four 64KB segments within the whole 1MB memory.

Memory Segmentation in 8086:

Memory segments

These four memory segments are called:

1. Code segment: It holds the instruction codes of a program.
2. Data segment: It holds the data, variables and constants given in the program
3. Extra segment: It also holds the data of certain string instructions.
4. Stack segment: It is used as a stack and it is used to store the return address. It holds addresses and data of subroutines.

Offset Address:

To address a specific memory location within a segment we need an offset address.

The offset address is also 16-bit wide and it is provided by one of the associated pointer or index register

Pointers and index registers contain offset address:

Stack Pointer and Base Pointer:

SP (Stack Pointer) : This is the 16-bit register. It points to the program stack in stack segment.

BP (Base Pointer) : BP is also the 16-bit register. It points to data in stack segment.

Source index: It is of 16 bits, It is used to point the memory locations in the data segment for source data.

Destination index: It is of 16 bits It is used to point the memory locations in the data segment for destination data.

Calculating Physical Address:

How can a 20-bit address be obtained, if there are only 16-bit registers?

Address Adder: The BIU contains a dedicated adder which is used to generate the 20bit physical address.

This address is formed by adding an 16 bit segment address and a 16 bit offset address.

> **Physical address= Segment address*10H+offset address**

Example problems on calculating physical address
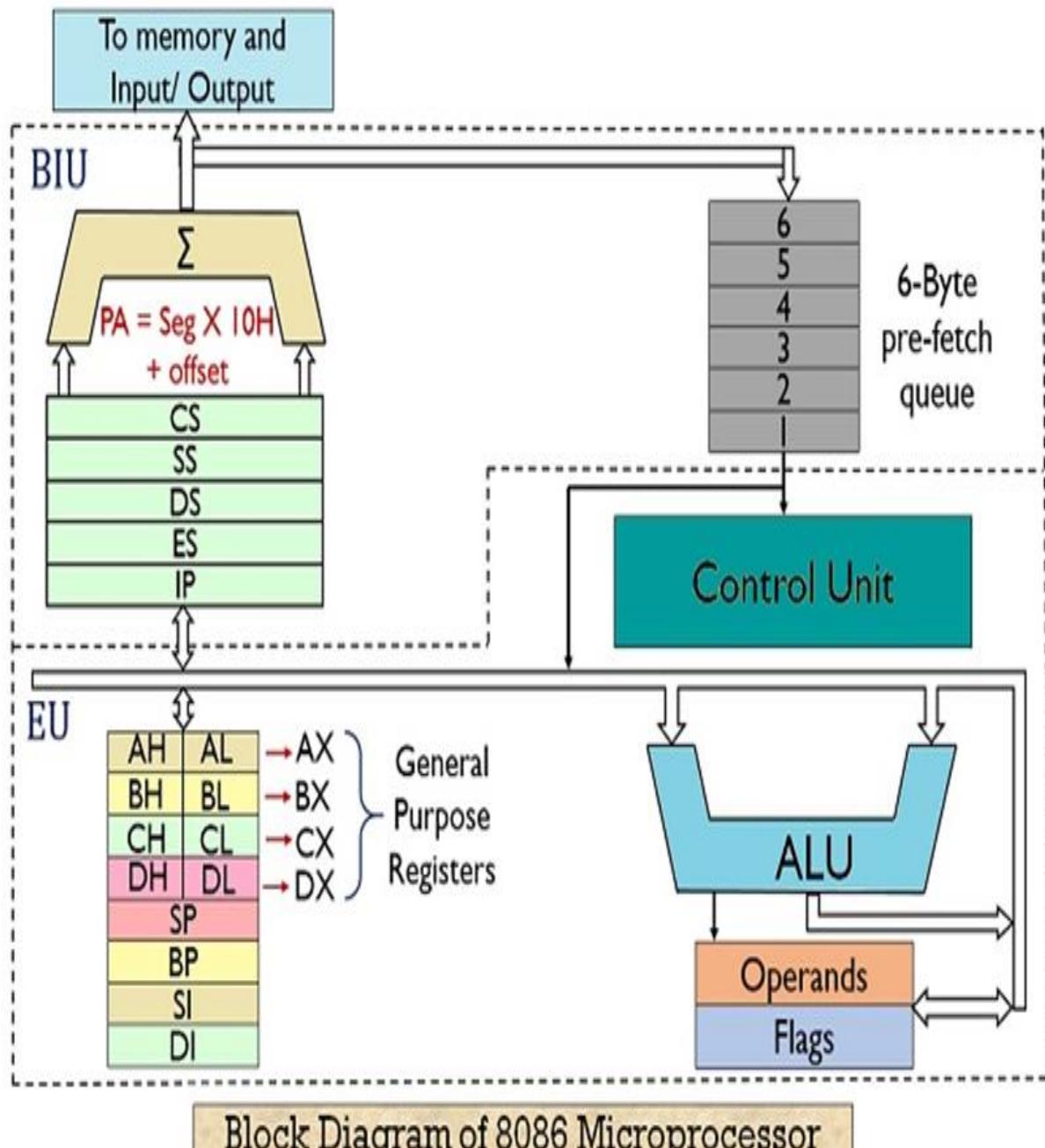
1Q.The value of Code Segment (CS) Register is 4042H and the value of different offsets is as follows: BX: 2025H , IP: 0580H , DI: 4247H Calculate the effective address/physical address of the memory location pointed by the CS register?

A: The offset of the CS Register is the IP register. Therefore, the effective address of the memory location pointed by the CS register is calculated as follows:

Effective address= Base address of CS register X 10_H + Address of IP

$$4042_H \text{ X } 10_H + 0580_H = (40420 + 0580)_H = 41000_H$$

Architecture of 8086 Microprocessor:

To memory and Input/ Output

BIU

Σ

PA = Seg X 10H + offset

CS
SS
DS
ES
IP

6
5
4
3
2
1

6-Byte pre-fetch queue

Control Unit

EU

AH	AL	→ AX
BH	BL	→ BX
CH	CL	→ CX
DH	DL	→ DX

General Purpose Registers

SP
BP
SI
DI

ALU

Operands

Flags

Block Diagram of 8086 Microprocessor

The architecture of 8086 can be internally divided into two separate functional units

   1. Bus Interface Unit (BIU)

   2. Execution Unit(EU)

The reason behind two separate sections for BIU and EU in the architecture of 8086 is to perform fetching and decoding-executing simultaneously, which is used to saves the processor time of operation i.ePipelined Architecture.

1. The Bus Interface Unit (BIU):

It provides the interface of 8086 to external memory and I/O devices via the System Bus. It performs various machine cycles such as memory read, I/O read etc. to transfer data between memory and I/O devices.

BIU performs the following functions-

- It generates the 20 bit physical address for memory access.
- It fetches instructions from the memory.
- It transfers data to and from the memory and I/O.
- Maintains the 6 byte prefetch instruction queue (**supports pipelining**).

- The BIU handles all transactions of data and addresses on the buses for EU.
- The instruction bytes are transferred to the instruction QUEUE.
- EU executes instructions from the instruction system byte queue.

➢ **BIU** mainly contains

- **4 segment registers**
- **6-byte pre-fetch queue**
- **Address Generation Unit**
- **Instruction Pointer**

Address Generation Unit:

- The physical address of the instruction is achieved by combining the segment address with that of the offset address.

6-byte pre-fetch queue

- This queue is used in 8086 in order to perform pipelining.

- As at the time of decoding and execution of the instruction in EU, the BIU fetches the sequential upcoming instructions and stores it in this queue.

- The size of this queue is 6-byte. This means at maximum a 6-byte instruction can be stored in this queue.

- The queue exhibits FIFO behavior, first in first out.

- BIU fills in the queue until the entire queue is full.

- BIU fetches 2 instruction bytes in a single memory cycle.

- BIU restart filling in the queue when at least two locations of queue are vacant.

Instruction Pointer: The Instruction Pointer is a register that holds the address of the next instruction to be fetched from memory.

4 Segment Registers

BIU contains 4 segment registers. Each segment register is of 16-bit.

The segments are present in the memory and these registers hold the base address of respective segments.

1. **Code Segment Register:**

It is a 16-bit register and holds the address of the instruction or program stored in the code segment of the memory.

2. **Stack segment register:**

The Stack segment register is usually used to store information about memory segment. It handles memory to store data and addresses during execution.
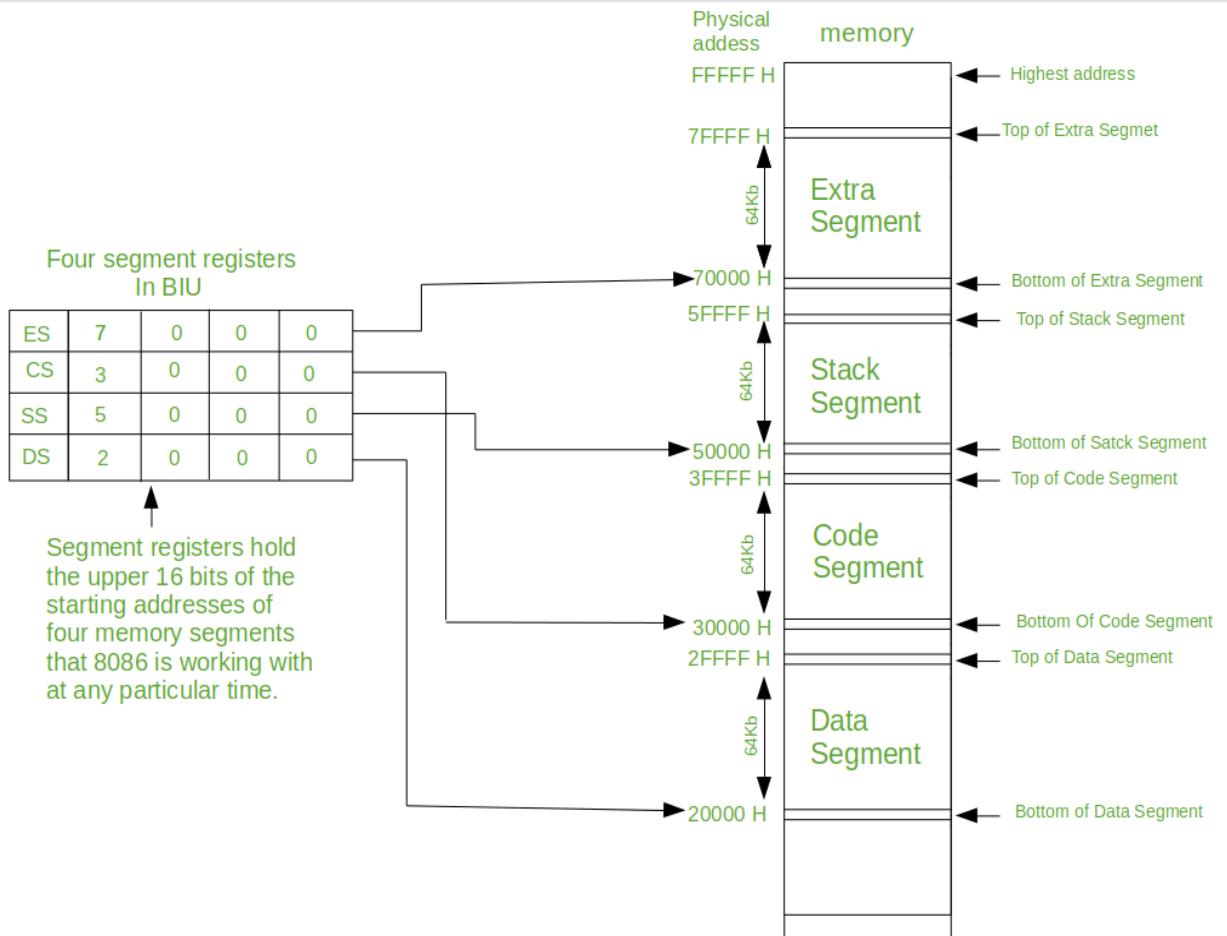
3. **Data segment register:**

It holds the address of the data segment. The data segment stores the data in the memory whose address is present in this 16-bit register.

4. **Extra segment register:**
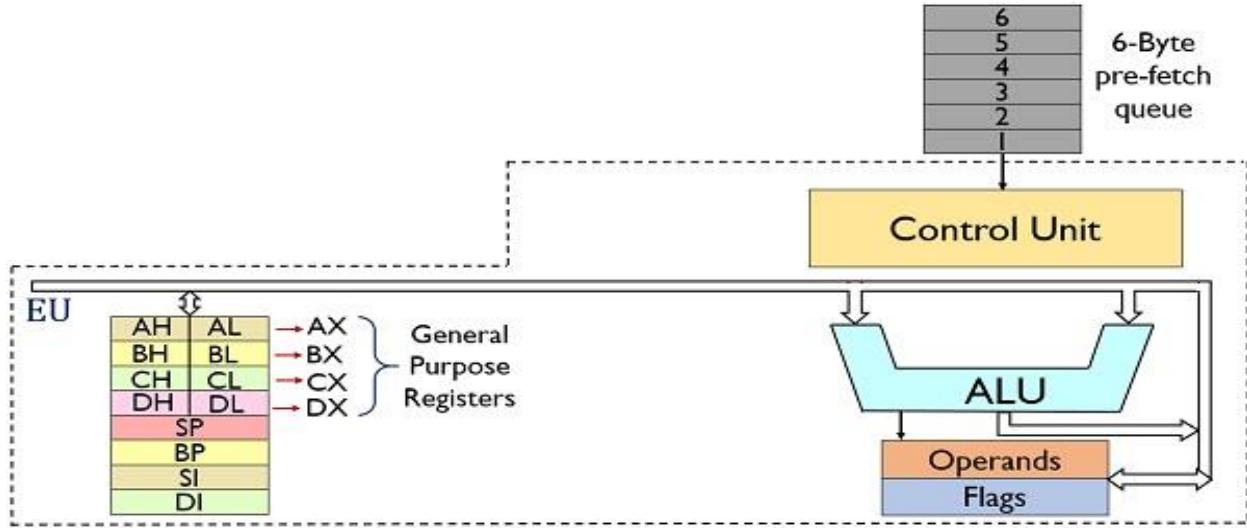
Here the starting address of the extra segment is present. This register basically contains the address of the string data.

String: mean a series of data words or bytes that reside in consecutive memory locations.

2. The Execution Unit (EU):

1. EU contains Control Unit, ALU, Pointer and Index register, Flag register, General Purpose Register, Operands
2. EU: Fetches instructions from the Queue in BIU, decodes and executes arithmetic and logic operations using the ALU.
3. Sends control signals for internal data transfer operations within the microprocessor.
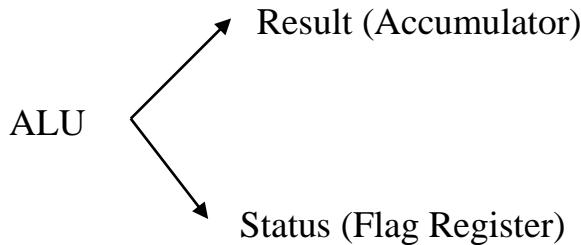
Control Unit:

It's directs the operation of the processor.

It also signals the ALU to perform the desired operation

ALU:

It handles all arithmetic and logical operations, like +, −, ×, /, OR, AND, NOT operations.

ALU → Result (Accumulator)

ALU → Status (Flag Register)

General purpose registers:

There are 8 general purpose registers, i.e., AH, AL, BH, BL, CH, CL, DH, and DL.

AX register

This is the accumulator. It is of 16 bits and is divided into two 8-bit registers AH and AL to also perform 8-bit instructions.

It is used to stores the 16-bit/8 bit result of certain ALU operations.

BX register

This is the base register. It is of 16 bits and is divided into two 8-bit registers BH and BL to also perform 8-bit instructions.

It is used to store the starting base address of the memory area within the data segment.

CX register

This is the counter register. It is of 16 bits and is divided into two 8-bit registers CH and CL to also perform 8-bit instructions.

It is referred to as counter. Used to hold the count value in SHIFT, ROTATE and LOOP instructions.

DX register

This is the data register. It is of 16 bits and is divided into two 8-bit registers DH and DL to also perform 8-bit instructions.

Pointer and Index Registers:

SP (Stack Pointer):

This is the 16-bit register. It points to the program stack in stack segment. SP is used during instructions like PUSH, POP, CALL, RET etc.

BP (Base Pointer) :

BP is also the 16-bit register. It points to data in stack segment. BP can hold offset address of any location in the stack segment. It is used to access random locations of the stack.

Source index:

It holds offset address in Data Segment during string operations

Destination index:

It is of 16 bits It holds offset address in Extra Segment during string operations
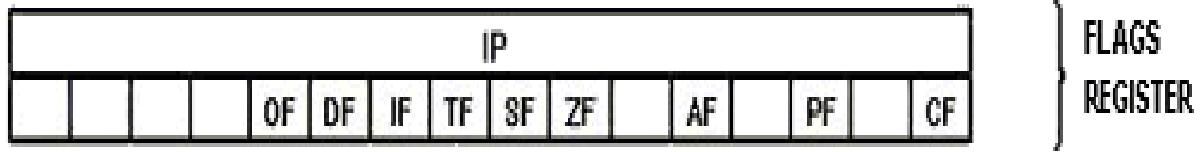
This register is used to hold I/O port address for I/O instruction

Flag Register:

Flag register holds the status of the result generated by the ALU.

The 8086 microprocessor has a 16 bit register for flag register. In this register 9 bits are active for flags.

In that 9 flags, they are divided into 2 groups − Conditional Flags and Control Flags.

Flag Registers Intel 8086-8088 Microprocessor

6 Status flags:

1. carry flag(CF)
2. parity flag(PF)
3. auxiliary carry flag(AF)
4. zero flag(Z)
5. sign flag(S)
6. overflow flag (O)

Status flags are updated after every arithmetic and logic operation.
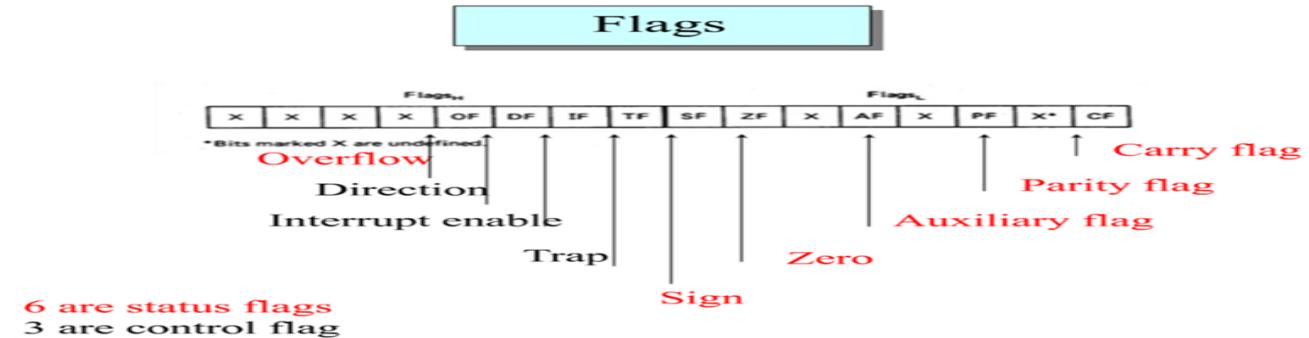
3 Control flags:
1. trap flag(TF)
2. interrupt flag(IF)
3. direction flag(DF)

Operand:

It is a temporary register and is used by the processor to hold the temporary values at the time of operation.

FLAG REGISTER OF 8086 MICROPROCESSOR:

Conditional Flags:

It represents the result of the last arithmetic or logical instruction executed. Following is the list of conditional flags

1. Carry flag − This flag indicates an overflow condition for arithmetic operations.(Carry is generated when performing n bit operations and the result is more than n bits) Addition-Carry , Subtraction-Barrow

Example: Add F0H and78H

2.Auxiliary flag − The AF is set/reset when a 1-byte arithmetic operation is performed at ALU, it results in a carry/barrow from lower nibble (i.e. D0 – D3) to upper nibble (i.e. D4 – D7), then this flag is set, i.e. carry given by D3 bit to D4 is AF flag.

EXAMPLE: Add 7CH, 0CH

$\overset{1}{D7}$	$\overset{1}{D6}$	$\overset{1}{D5}$	$\overset{1}{D4}$	D3	$\overset{1}{D2}$	D1	D0
0	1	1	1	1	1	0	0
0	0	0	0	1	1	0	0
1	0	0	0	1	0	0	0

AC=1

3. Parity flag: This flag is used to indicate the parity of the result, The result contains even number of 1's, then the Parity Flag is set. For odd number of 1's, the Parity Flag is reset.

➢ P=1

(AND A register with Accumulator)

D7	D6	D5	D4	D3	D2	D1	D0
0	1	1	1	0	0	0	1
0	1	1	1	0	0	0	1
0	1	1	1	0	0	0	1

4. Zero flag − this flag is set to 1 when the result of arithmetic or logical operation is zero else it is set to 0.

(XOR A register with Accumulator)

D7	D6	D5	D4	D3	D2	D1	D0
0	0	1	0	0	1	0	1
0	0	1	0	0	1	0	1
0	0	0	0	0	0	0	0

Z=1,The Zero flag is set because the ALU
operation result in 0

5. Sign flag − After any operation if the MSB (B(7)) of the result is 1, it indicates the number is negative and the sign flag becomes set, i.e. 1. If the MSB is 0, it indicates the number is positive

6. Overflow flag − Overflow flag became set as we added 2 positive numbers and we got a negative number.

```
MOV AL, 50 (50 is 01010000 which is positive)
MOV BL, 32 (32 is 00110010 which is positive)
ADD AL, BL (82 is 10000010 which is negative)
```

Control Flags:

Control flags controls the operations of the execution unit.

Trap flag − When a system is instructed to single-step, it will execute one instruction and then stop.

Interrupt flag − It is an interrupt enable/disable flag, i.e. used to allow/prohibit the interruption of a program. It is set to 1 for interrupt enabled condition and set to 0 for interrupt disabled condition.

Direction flag- It is used in string operation. As the name suggests when it is set then string bytes are accessed from the higher memory address to the lower memory address and vice-a-versa.

Features of 8086 Microprocessor:

It is a 16-bit Microprocessor introduced by INTEL in the year 1978. (The term "16-bit" means that its arithmetic logic unit(ALU), internal registers and most of its instructions are designed to work with 16-bit binary words.)

It requires +5V DC power supply.

8086 has a 20 bit address bus can access up to 2^{20} memory locations (1 MB).*Address ranges from 00000H to FFFFFH*

The 8086 can generate 16-bit I/O address; hence it can access 2^16 = 65536(64K) I/O ports.

It has 16-bit data bus, so it can read data from or write data to memory and ports either 16 bits or 8 bits at a time.

The 8086 has multiplexed address and data bus which reduced the number of pins needed (AD0-AD15)

It is available in 40 pin Dual In line Package(DIP).

It consists of 29,000 HMOS transistors.

It has 6 bytes Queue

It provides 14, 16-bit registers.

Clock frequency ranges from 5MHz to 10 MHz

It uses two stages of pipelining, i.e. Fetch Stage and Execute Stage, which improves performance.

8086 is designed to operate in two modes

Implementation of Pipelined Process in 8086

Implementation of pipelining is done by 2 units in the 8086 Microprocessor.

BIU(consists of 6 byte prefetch Queue):

- Fetches the sequenced instruction from the memory,

- Finds the physical address of that location in the memory where the instruction is stored and

- Manages the 6-byte pre-fetch queue.

6-byte pre-fetch queue:

- This queue is used in 8086 in order to perform pipelining.

- As at the time of decoding and execution of the instruction in EU, the BIU fetches the sequential upcoming instructions and stores it in this queue.

- The size of this queue is 6-byte.

- This means at maximum a 6-byte instruction can be stored in this queue.

- The queue exhibits FIFO behavior, first in first out.

- BIU fills in the queue until the entire queue is full.

- BIU fetches 2 instruction bytes in a single memory cycle.

- BIU restart filling in the queue when at least two locations of queue are vacant.

EU (Execution Unit)

- Decodes instructions fetched by the BIU

- Executes instructions.

- EU contains Control Unit, ALU, Pointer and Index register, Flag register, General Purpose Register, Operands

Register Organization:

A register is a very small amount of fast memory that is built in the CPU (or Processor) in order to speed up the operation.

Register is very fast and efficient than the other memories like RAM, ROM, external memory etc,.

That's why the registers occupied the top position in memory hierarchy model

The 8086 microprocessor has a total of fourteen registers that are accessible to the programmer.

All these registers are 16-bit in size. The registers of 8086 are categorized into 5 different groups.

- **General registers**
- **Index registers**
- **Pointer registers**
- **Segment registers**
- **Status registers**
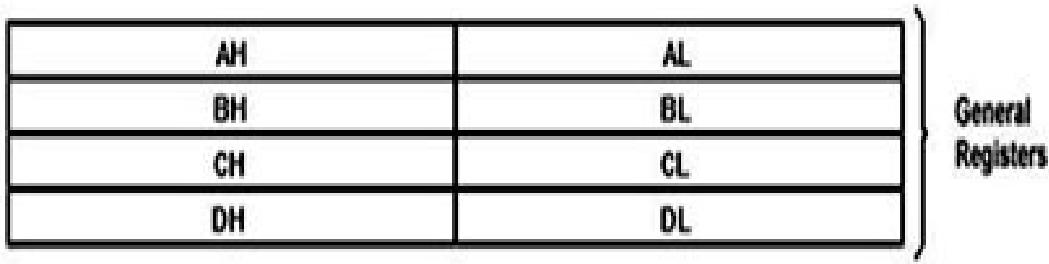
8086 REGISTER ORGANIZATION

ES	Extra Segment		
CS	Code Segment		
SS	Stack Segment		
DS	Data Segment		
IP	Instruction Pointer		

AX	AH	AL	Accumulator
BX	BH	BL	Base Register
CX	CH	CL	Count Register
DX	DH	DL	Data Register
	SP		Stack Pointer
	BP		Base Pointer
	SI		Source Index
	DI		Destination Index
	FLAGS		

Type	Register size	Name of the Register
General purpose registers	16 bit	AX, BX, CX, DX
	8 bit	AL, AH, BL, BH, CL, CH, DL, DH
Pointer registers	16 bit	SP, BP
Indexed registers	16 bit	SI, DI
Instruction Pointer	16 bit	IP
Segment registers	16 bit	CS, DS, SS, ES
Flags	16 bit	Flag register

General Purpose Registers:

General purpose registers are used to store temporary data within the microprocessor during arithmetic and logic operations. These all general registers can be used as either 8-bit or 16-bit registers.

The general registers are:

AH	AL
BH	BL
CH	CL
DH	DL

General Registers

AX (Accumulator):

This is the accumulator. It is of 16 bits and is divided into two 8-bit registers AH and AL to also perform 8-bit instructions.

It is used to stores the 16-bit/8 bit result of certain arithmetic and logical operations.

This Accumulator used in arithmetic, logic and data transfer operations. For manipulation and division operations, one of the numbers must be placed in AX or AL.

BX register:

This is the base register. It is of 16 bits and is divided into two 8-bit registers BH and BL to also perform 8-bit instructions.

It is used to store the starting base address of the memory area within the data segment.

CX register

This is the counter register. It is of 16 bits and is divided into two 8-bit registers CH and CL to also perform 8-bit instructions.

It is referred to as counter. Used to hold the count value in SHIFT, ROTATE and LOOP instructions.

DX register

This is the data register. It is of 16 bits and is divided into two 8-bit registers DH and DL to also perform 8-bit instructions.

This register is used to hold I/O port address for I/O instruction.

Index Register:

Source index:

It holds offset address in Data Segment during string operations

Destination index:

It is of 16 bits It holds offset address in Extra Segment during string operations

This register is used to hold I/O port address for I/O instruction

Pointer Registers:

SP (Stack Pointer) :

This is the 16-bit register. It points to the program stack in stack segment. SP is used during instructions like PUSH, POP, CALL, RET etc.

BP (Base Pointer):

BP is also the 16-bit register. It points to data in stack segment. BP can hold offset address of any location in the stack segment. It is used to access random locations of the stack.

Instruction Pointer

The Instruction Pointer is a register that holds the address of the next instruction to be fetched from memory.

Segment Registers:

The segments are present in the memory and these registers hold the address of respective segments.

These registers are as follows:

- **Code segment register**
- **Stack segment register**
- **Data segment register**
- **Extra segment register**

Code Segment Register:

It is a 16-bit register and holds the address of the instruction or program stored in the code segment of the memory.

Stack segment register:

The Stack segment register is usually used to store information about memory segment. It handles memory to store data and addresses during execution.

Data segment register:

It holds the address of the data segment. The data segment stores the data in the memory whose address is present in this 16-bit register.

Extra segment register:

Here the starting address of the extra segment is present. This register basically contains the address of the string data.
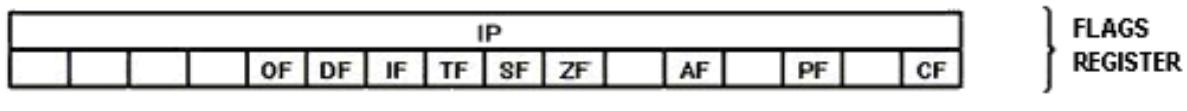
Status Register (Flag register):

The status register also called as flag register. The 8086 flag register contents indicate the results of computation in the ALU. It also contains some flag bits to control the CPU operations.

Flag register holds the status of the result generated by the ALU.

The 8086 microprocessor has a 16 bit register for flag register. In this register 9 bits are active for flags.

In that 9 flags, they are divided into 2 groups − Conditional Flags and Control Flags.

Flag Registers Intel 8086-8088 Microprocessor

Programming Model:
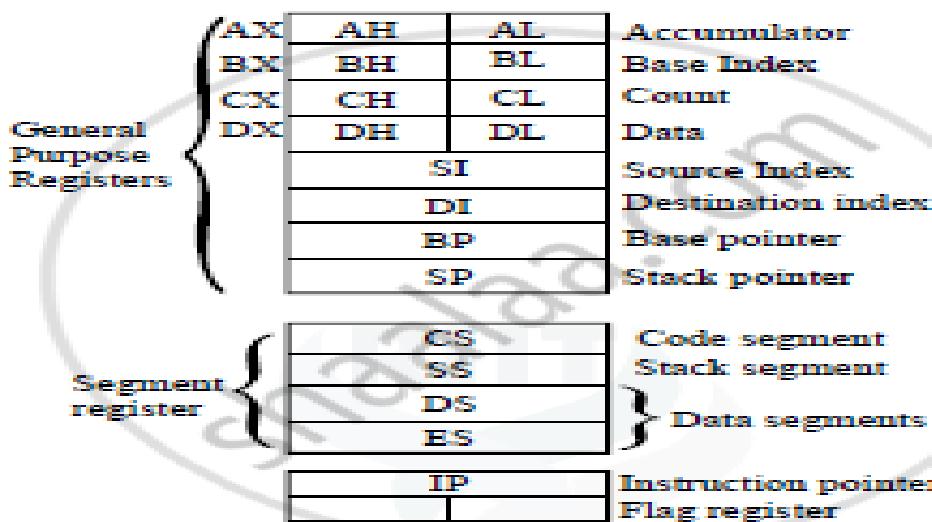
- The programming model of the 8086 considered to be program visible because its registers are used during application programming and are specified by the instructions.

- Other registers, are considered to be program invisible because they are not addressable directly during applications programming,

- But may be used indirectly during system programming.

Q. How can a 20-bit address be obtained, if there are only 16-bit registers?

- However, the largest register is only 16 bits (64k); so physical addresses have to be calculated. These calculations are done in hardware within the microprocessor.

- The 16-bit contents of segment register gives the starting/ base address of particular segment.

- To address a specific memory location within a segment we need an offset address.

- The offset address is also 16-bit wide and it is provided by one of the associated pointer or index register.

- **To be able to program a microprocessor**, one does not need to know all of its hardware architectural features. What is important to the programmer is being aware of the various registers within the device and to understand their purpose, functions, operating capabilities, and limitations.

The below figure illustrates the software architecture of the 8086 microprocessor. In the programming model there are

- 4 General Purpose registers(Data Registers)
- 4 Segment registers
- 2 Pointer registers
- 2 Index registers
- 1 Instruction Pointer register
- 1 Flag register

Programming model of 16-bit version of X-86 family

The point to note is that the beginning segment address must begin at an address divisible by 16.Also note that the four segments need not be defined separately. It is allowable for all four segments to completely overlap (CS = DS = ES = SS).

Logical and Physical Address

Addresses within a segment can range from address 00000h to address 0FFFFh. This corresponds to the 64K-bytelength of the segment. An address within a segment is called an offset or logical address.

A logical address gives the displacement from the base address of the segment to the desired location within it, as opposed to its "real" address, which maps directly anywhere into the 1 MByte memory space. This "real" address is called the physical address.

What is the difference between the physical and the logical address?

The physical address is 20 bits long and corresponds to the actual binary code output by the BIU on the address bus lines. The logical address is an offset from location 0 of a given segment.

Physical Address Generation in 8086

➤ The 20-bit physical address is generated by adding 16-bit contents of a **segment register** with an 16-bit **offset value** (also called **Effective Address**) which is stored in a corresponding **default register** (either in IP, BX, SI, DI, BP or SP. Different segments have different default register for offset, for example IP is default offset register for Code Segment)

➤ BIU always appends **4 zeros** automatically to the 16-bit address of a segment register (to make it 20-bit) because it knows the starting address of a segment always ends with 4 zeros
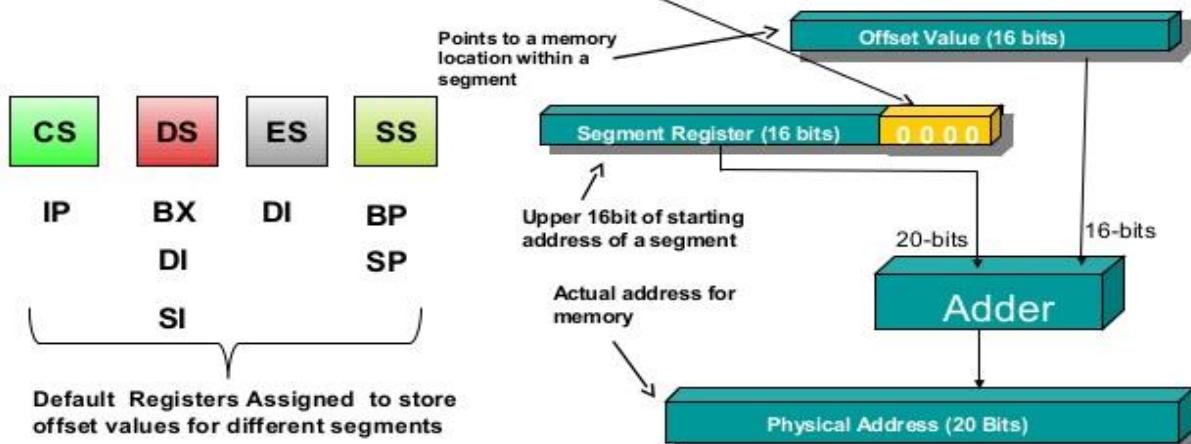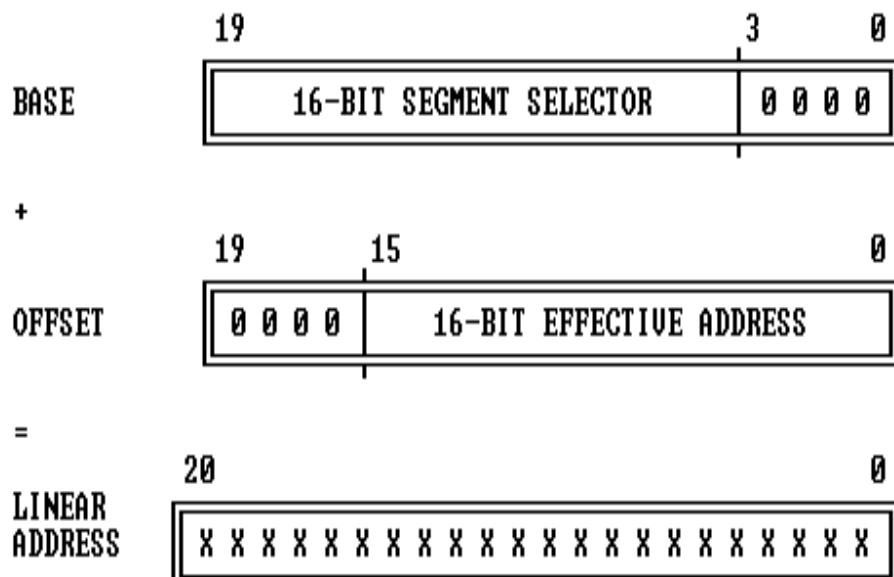
Figure 14-1. Real-Address Mode Address Formation

You should also be careful when writing addresses on paper to do so clearly. To specify the logical address XXXX in the stack segment, use the convention SS:XXXX, which is equal to [SS] * 16 + XXXX.

Logical address is in the form of: **Base Address: Offset**

Offset is the displacement of the memory location from the starting location of the segment. To calculate the physical address of the memory, BIU uses the following formula:

Physical address= Segment address*10H+offset address

Physical Memory Organization:

The total memory (1MB) of 8086 is physically organised as an odd bank and even bank each of 512K 8-bit bytes addressed in parallel by the processor.

 1. A high (odd) bank (D15-D8) and

 2. A low (low) bank (D7-D0)

- Byte data with even addresses is transferred on the D7-D0 bus lines ;
- While odd addressed byte data is transferred on the D15-D8 bus lines.
- The processor provides two enable signals, **BHE(Bus High Enable) and A0** for selection of either even or odd bank or both the banks.
- Even addresses are on the low half of the data bus(D0-D7)
- Odd addresses are on the high half of the data bus(D8-D15)
- **A0 =0;** when data is on the low half of the data bus(D0-D7)
- **$\overline{\text{BHE}}$ =0;** when data is on the high half of the data bus(D8-D15)
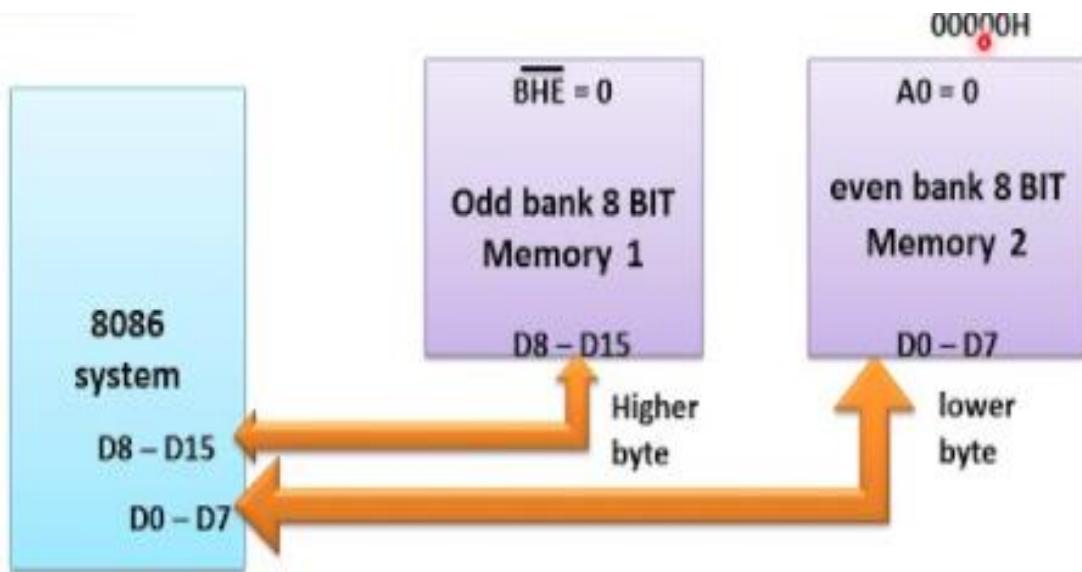
Fig: Physical Memory Organization

The two signals A0and $\overline{BHE}$ select the appropriate banks as shown in below table

$\overline{BHE}$	A0	Indication
0	0	Whole word(2 byte)
0	1	Upper byte from or to odd address
1	0	Lower byte from or to even address
1	1	None

Table: Selection of banks using BHE and A0

PIN DIAGRAM
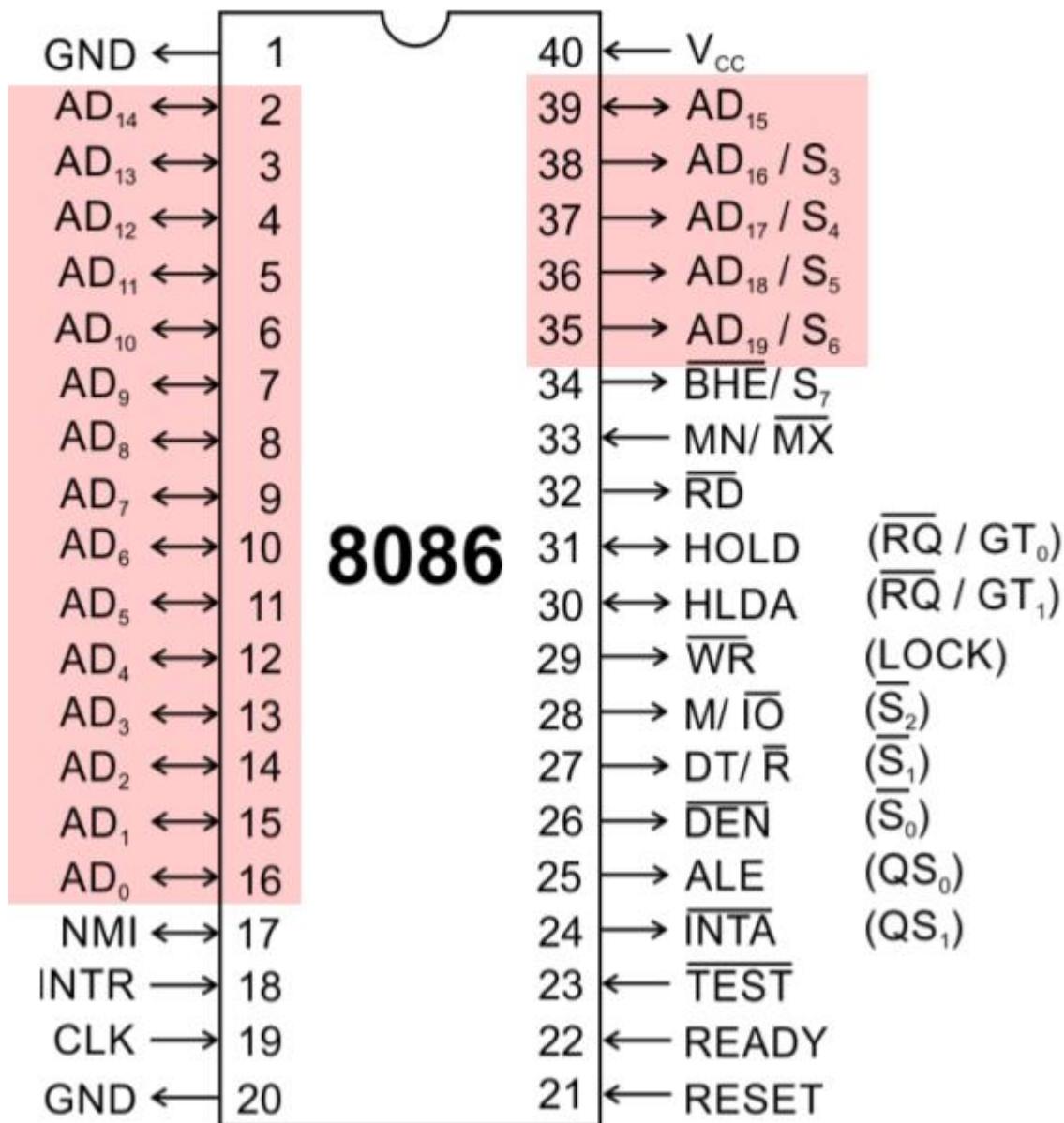
(Signal Descriptions of 8086)

```
                              MAX       ( MIN  )
                              MODE        MODE

         GND ▭ 1          40 ▭ U_cc
        AD14 ▭ 2          39 ▭ AD15
        AD13 ▭ 3          38 ▭ A16/S3
        AD12 ▭ 4          37 ▭ A17/S4
        AD11 ▭ 5          36 ▭ A18/S5
        AD10 ▭ 6          35 ▭ A19/S6
         AD9 ▭ 7          34 ▭ BHE/S7
         AD8 ▭ 8          33 ▭ MN/MX
         AD7 ▭ 9          32 ▭ RD
                 8086
         AD6 ▭ 10  CPU    31 ▭ RQ/GT0    (HOLD)
         AD5 ▭ 11         30 ▭ RQ/GT1    (HLDA)
         AD4 ▭ 12         29 ▭ LOCK      (WR)
         AD3 ▭ 13         28 ▭ S2        (M/IO)
         AD2 ▭ 14         27 ▭ S1        (DT/R)
         AD1 ▭ 15         26 ▭ S0        (DEN)
         AD0 ▭ 16         25 ▭ QS0       (ALE)
         NMI ▭ 17         24 ▭ QS1       (INTA)
        INTR ▭ 18         23 ▭ TEST
         CLK ▭ 19         22 ▭ READY
         GND ▭ 20         21 ▭ RESET
```

- Out of 40 pins, 32 pins are having same function in minimum or maximum mode,

- And remaining 8 pins are having different functions in minimum and maximum mode.

- Following are the pins which are having same functions

Common signals:

|		8086		
GND ←	1		40 ←	V_{CC}
AD_{14} ↔	2		39 ↔	AD_{15}
AD_{13} ↔	3		38 →	AD_{16} / S_3
AD_{12} ↔	4		37 →	AD_{17} / S_4
AD_{11} ↔	5		36 →	AD_{18} / S_5
AD_{10} ↔	6		35 →	AD_{19} / S_6
AD_9 ↔	7		34 →	$\overline{BHE} / S_7$
AD_8 ↔	8		33 ←	$MN / \overline{MX}$
AD_7 ↔	9		32 →	$\overline{RD}$
AD_6 ↔	10		31 ↔	HOLD $(\overline{RQ} / GT_0)$
AD_5 ↔	11		30 ↔	HLDA $(\overline{RQ} / GT_1)$
AD_4 ↔	12		29 →	$\overline{WR}$ (LOCK)
AD_3 ↔	13		28 →	$M / \overline{IO}$ $(\overline{S_2})$
AD_2 ↔	14		27 →	$DT / \overline{R}$ $(\overline{S_1})$
AD_1 ↔	15		26 →	$\overline{DEN}$ $(\overline{S_0})$
AD_0 ↔	16		25 →	ALE (QS_0)
NMI ↔	17		24 →	$\overline{INTA}$ (QS_1)
INTR →	18		23 ←	$\overline{TEST}$
CLK →	19		22 ←	READY
GND ←	20		21 ←	RESET

AD_0–AD_{15} (Bidirectional)

Address/Data bus

These are 16 address/data bus.

AD0-AD7 carries low order byte data and AD8-AD15 carries higher order byte data.

During the first clock cycle, it carries 16-bit address and after that it carries 16-bit data.

Address/status bus

A16/S3 - A17/S4 - A18/S5 - A19/S6.

These are the 4 address/status buses.

During the first clock cycle, it carries 4-bit address and later it carries status signals.

Address/Status bus

A17/S4	A16/S3	FUNCTION
0	0	Extra segment access
0	1	Stack segment access
1	0	Code segment access
1	1	Data segment access

BHE (Active Low)/S_7 (Output)

Bus High Enable/Status

It is used to indicate the transfer of most significant half of data using data bus D8-D15.

This signal is low during the first clock cycle, thereafter it is active.

It is multiplexed with status signal S_7.

MN/ MX
MINIMUM / MAXIMUM

It stands for Minimum/Maximum and is available at pin 33.

It indicates what mode the processor is to operate in; when it is high, it works in the minimum mode and vice-versa.

RD (Read) (Active Low)
The signal is used for read operation.

READY

- It is available at pin 22.
- It is an acknowledgement signal from Memory or I/O devices that data is transferred.
- It is an active high signal.
- When it is high, it indicates that the device is ready to transfer data.
- When it is low, it indicates wait state.

TEST

- TEST pin is examined by the "WAIT" instruction.
- If the TEST pin is Low, execution continues.
- Otherwise the processor waits in an **"idle"** state.

RESET (Input)

Reset causes the processor to immediately terminate its present activity.

CLK

Clock signal is provided through Pin-19. It provides timing to the processor for operations. Its frequency is different for different versions, i.e. 5MHz, 8MHz and 10MHz.

INTR Interrupt Request

Interrupt pin
This signal is active high and internally synchronized.

NMI (Non Maskable Interrupt) Signal

Min/ Max Pins

The 8086 microprocessor can work in two modes of operations : Minimum mode and Maximum mode.

In the <u>minimum mode</u> of operation the microprocessor <u>do not</u> associate with any co-processors and cannot be used for multiprocessor systems.

In the <u>maximum mode</u> the 8086 <u>can work</u> in multi-processor or co-processor configuration.

Minimum or maximum mode operations are decided by the pin MN/ MX(Active low).
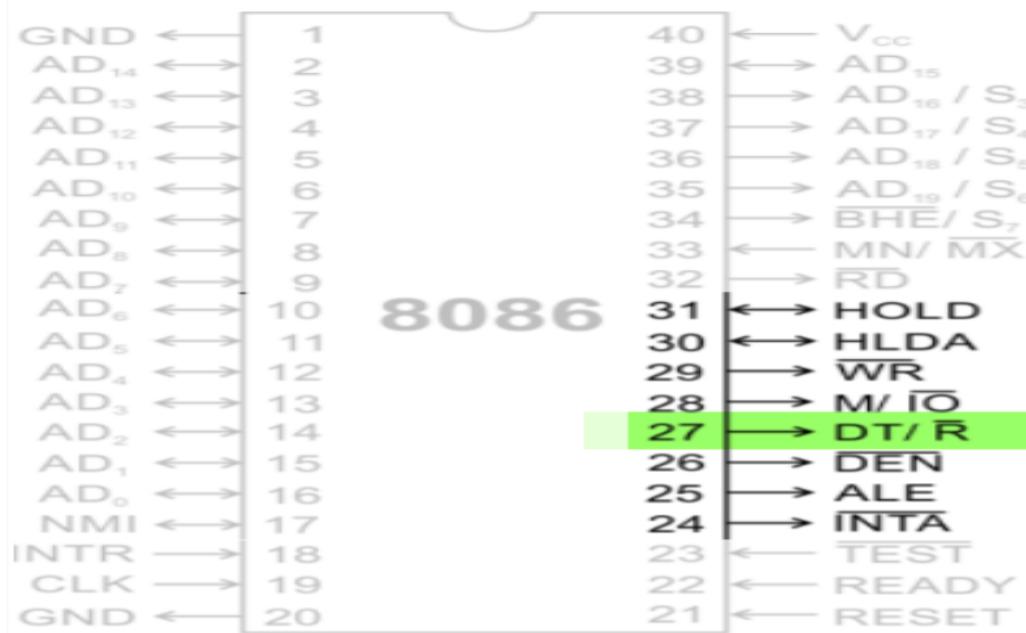
When this pin is <u>high</u> 8086 operates in <u>minimum mode</u> otherwise it operates in Maximum mode.

Minimum mode signals

Pins 24 -31

For minimum mode operation, the MN/ $\overline{\text{MX}}$ is tied to VCC (logic high)

8086 itself generates all the bus control signals

GND ← 1   40 ← V_{CC}
AD_{14} ↔ 2   39 ↔ AD_{15}
AD_{13} ↔ 3   38 → AD_{16}/S_3
AD_{12} ↔ 4   37 → AD_{17}/S_4
AD_{11} ↔ 5   36 → AD_{18}/S_5
AD_{10} ↔ 6   35 → AD_{19}/S_6
AD_9 ↔ 7   34 → $\overline{BHE}/S_7$
AD_8 ↔ 8   33 ← MN/$\overline{MX}$
AD_7 ↔ 9   32 → $\overline{RD}$
AD_6 ↔ 10   **8086**   31 ↔ HOLD
AD_5 ↔ 11   30 ↔ HLDA
AD_4 ↔ 12   29 → $\overline{WR}$
AD_3 ↔ 13   28 → M/$\overline{IO}$
AD_2 ↔ 14   27 → DT/$\overline{R}$
AD_1 ↔ 15   26 → $\overline{DEN}$
AD_0 ↔ 16   25 → ALE
NMI ↔ 17   24 → $\overline{INTA}$
INTR → 18   23 ← $\overline{TEST}$
CLK → 19   22 ← READY
GND ← 20   21 ← RESET
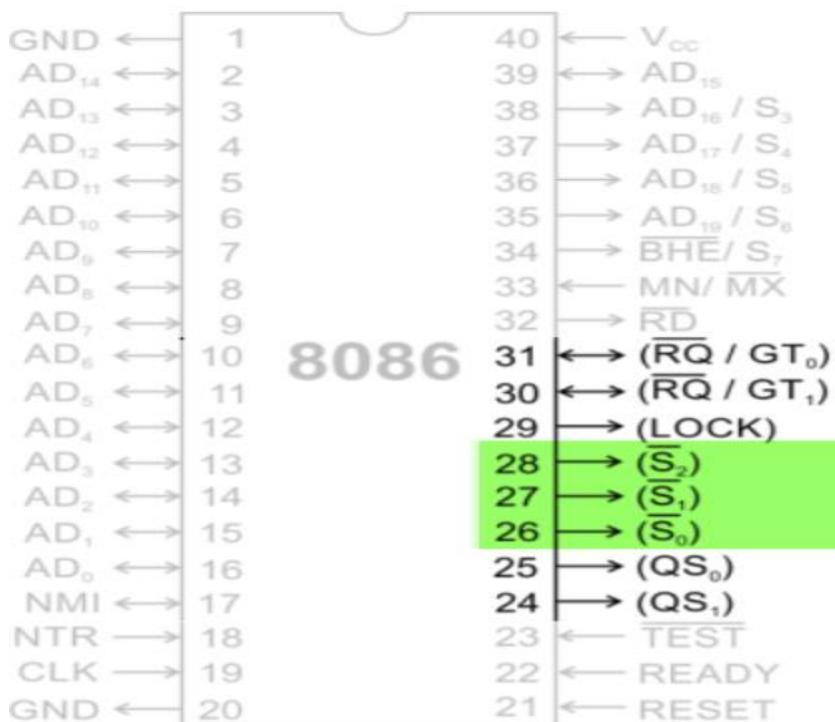
DT/$\overline{R}$ — **(Data Transmit/ Receive)** Output signal from the processor to control the direction of data flow through the data transceivers

$\overline{DEN}$ — **(Data Enable)** Output signal from the processor used as out put enable for the transceivers

ALE — **(Address Latch Enable)** Used to demultiplex the address and data lines using external latches

M/$\overline{IO}$ — Used to differentiate memory access and I/O access. For memory reference instructions, it is **high**. For IN and OUT instructions, it is **low**.

$\overline{WR}$ — Write control signal; asserted **low** Whenever processor writes data to memory or I/O port

$\overline{INTA}$ — **(Interrupt Acknowledge)** When the interrupt request is accepted by the processor, the output is **low** on this line.

HOLD   Input signal to the processor form the bus masters as a request to grant the control of the bus.

Usually used by the DMA controller to get the control of the bus.

HLDA   (**Hold Acknowledge**) Acknowledge signal by the processor to the bus master requesting the control of the bus through HOLD.

The acknowledge is asserted high, when the processor accepts HOLD.

Maximum mode signals:

During maximum mode operation, the MN/ $\overline{MX}$ is grounded (logic low)

Pins 24 –31 are reassigned

$\overline{S_0}, \overline{S_1}, \overline{S_2}$ **Status signals**; used by the 8086 bus controller to generate bus timing and control signals. These are decoded as shown.

Status Signal			Machine Cycle
$\overline{S_2}$	$\overline{S_1}$	$\overline{S_0}$	
0	0	0	Interrupt acknowledge
0	0	1	Read I/O port
0	1	0	Write I/O port
0	1	1	Halt
1	0	0	Code access
1	0	1	Read memory
1	1	0	Write memory
1	1	1	Passive/Inactive

$\overline{QS_0}, \overline{QS_1}$ **(Queue Status)** The processor provides the status of queue in these lines.

The queue status can be used by external device to track the internal status of the queue in 8086.

The output on QS_0 and QS_1 can be interpreted as shown in the table.

Queue status		Queue operation
QS_1	QS_0	
0	0	No operation
0	1	First byte of an opcode from queue
1	0	Empty the queue
1	1	Subsequent byte from queue

$\overline{RQ}/\overline{GT_0}$,
$\overline{RQ}/\overline{GT_1}$
(Bus Request/ Bus Grant) These requests are used by other local bus masters to force the processor to release the local bus at the end of the processor's current bus cycle.

These pins are bidirectional.

The request on $\overline{GT_0}$ will have higher priority than $\overline{GT_1}$

$\overline{LOCK}$
An output signal activated by the LOCK prefix instruction.

Remains active until the completion of the instruction prefixed by LOCK.

The 8086 output low on the $\overline{LOCK}$ pin while executing an instruction prefixed by LOCK to <u>prevent other bus masters from gaining control of the system bus.</u>

Minimum and Maximum mode Timing Signals of 8086:

Minimum Mode 8086:

The microprocessor 8086 is operated in minimum mode by strapping its at pin 33 MN/MX pin to logic 1.

In this mode, all the control signals are given out by the☐ microprocessor chip itself. There is a single microprocessor in the minimum mode system.

The remaining components in the system are latches, transreceivers, clock generator, memory and I/O devices.

Latches are generally buffered output D-type flip-flops like 74LS373 or 8282. They are used for **separating the valid address from the multiplexed**
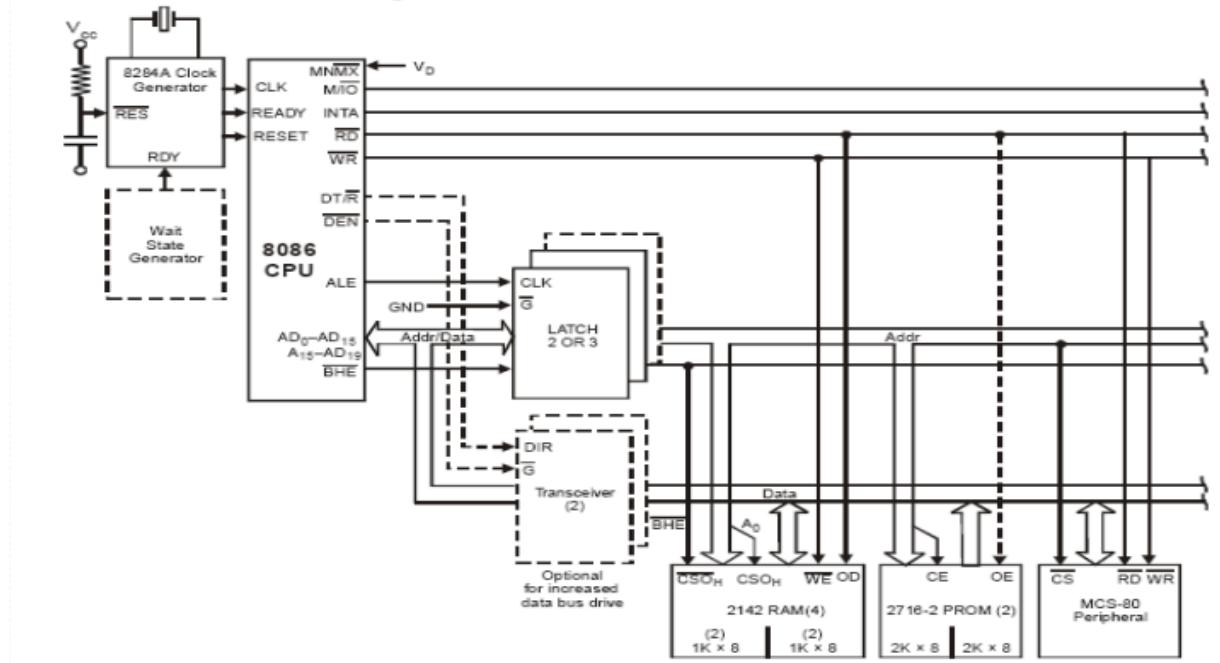
address/data signals and are controlled by the ALE signal generated by 8086.

Transreceivers are the bidirectional buffers and some times they are called as data amplifiers. They are required to **separate the valid data from the time multiplexed address/data signals.** They are controlled by two signals namely, **DEN and DT/R.** The DEN signal indicates the direction of data, i.e. from or to the processor.
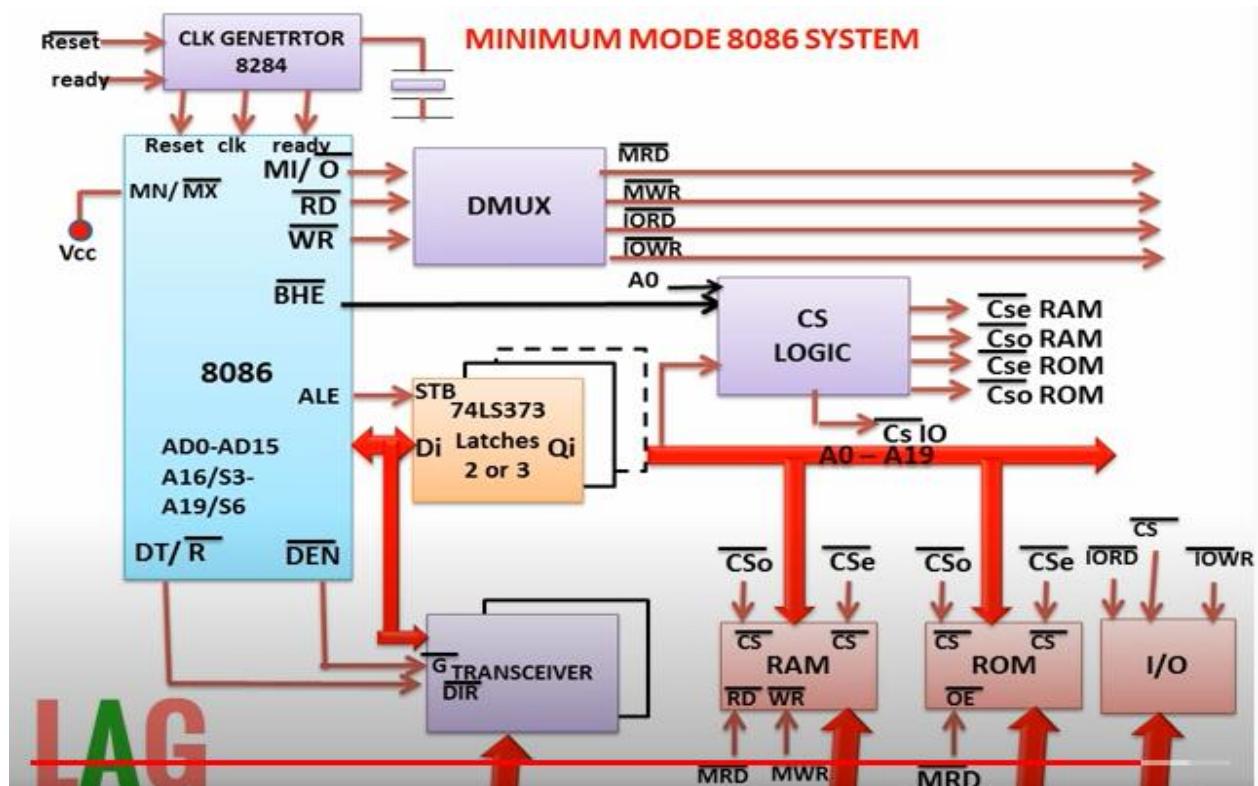
The system contains **memory** for the **monitor and users program storage**. Usually, EPROM are used for monitor storage, while RAM for users program storage.

A system may contain **I/O devices**. The opcode fetch and read cycles are similar.

Minimum Mode 8086 Configuration

OR

MINIMUM MODE 8086 SYSTEM

The timing diagram can be categorized in two parts, the first is the timing diagram **for read cycle and the second is the timing diagram for write cycle.**
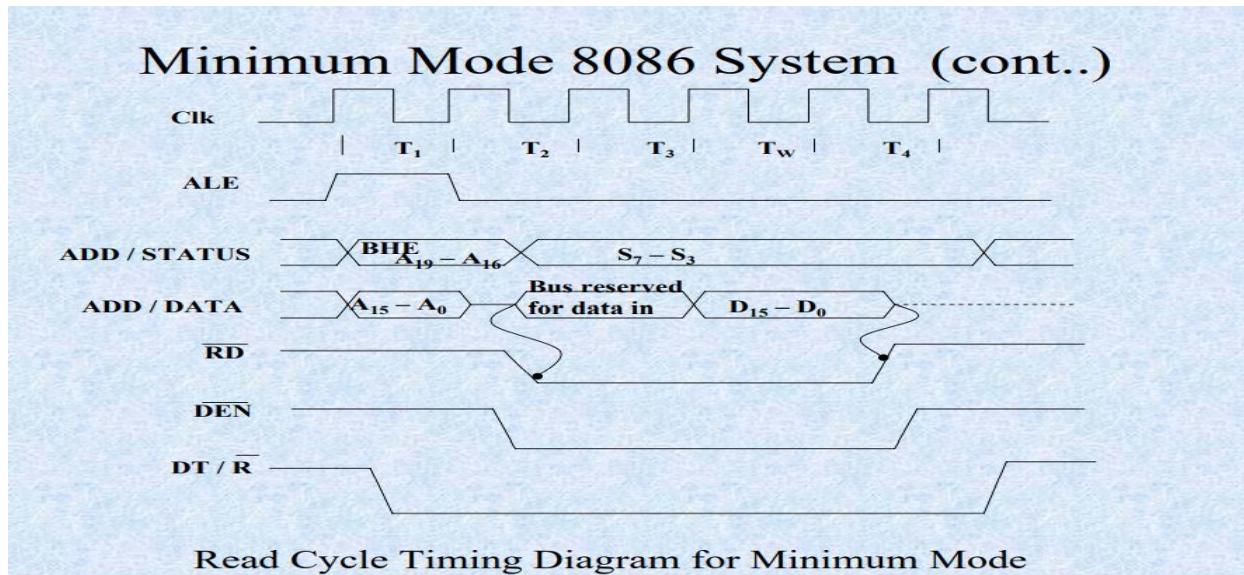
**Timing signals for Minimum Mode: Read Cycle:**

The read cycle begins in T1 with the assertion of **address latch enable (ALE)** signal and also M / IO signal. During the negative going edge of this signal, the **valid address is latched** on the local bus.

 The BHE and A0 signals address low, high or both bytes. From T1 to T4 , the M/IO signal indicates a memory or I/O operation.

At T2, **the address is removed from the local bus** and is sent to the output. The bus is then tristated. **The read (RD) control signal is also activated in T2.** The

read (RD) signal causes the address device to enable its data bus drivers. After RD goes low, the valid data is available on the data bus. The addressed device will drive the READY line high. When the processor returns the read signal to high level, the addressed device will again tristate its bus drivers.

Read Cycle Timing Diagram for Minimum Mode
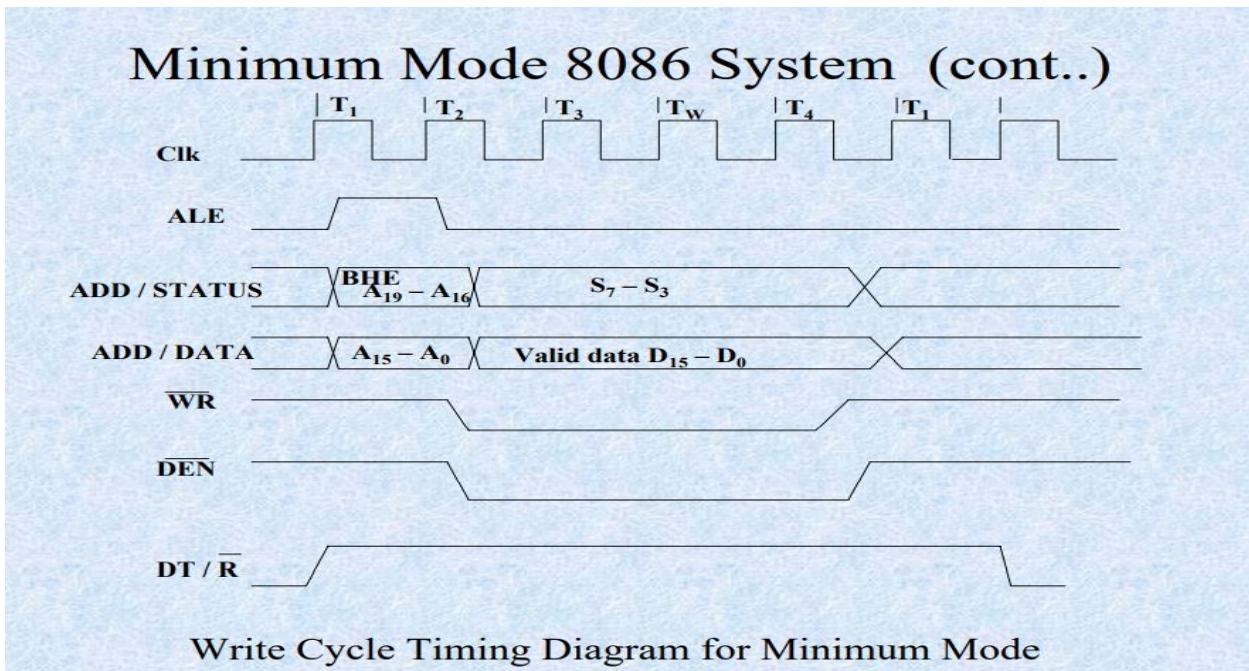
Timing signals for Minimum Mode: Write Cycle:

A write cycle also begins with the assertion of ALE and the emission of the address. The M/IO signal is again asserted to indicate a memory or I/Ooperation.

In T2, after sending the address in T1, **the processor sends the data to be written to the addressed location.** The data remains on the bus until middle of T4 state**. The WR becomes active at the beginning of T2** (unlike RD is somewhat delayed in T2 to provide time for floating). The BHE and A0 signals are used to select the proper byte or bytes of memory or I/O word to be read or write. The M/IO, RD and WR signals indicate the type of data transfer as shown in table below.

M / $\overline{IO}$	$\overline{RD}$	$\overline{WR}$	Transfer Type
0	0	1	I / O read
0	1	0	I/O write
1	0	1	Memory read
1	1	0	Memory write

Data Transfer table

Minimum Mode 8086 System (cont..)
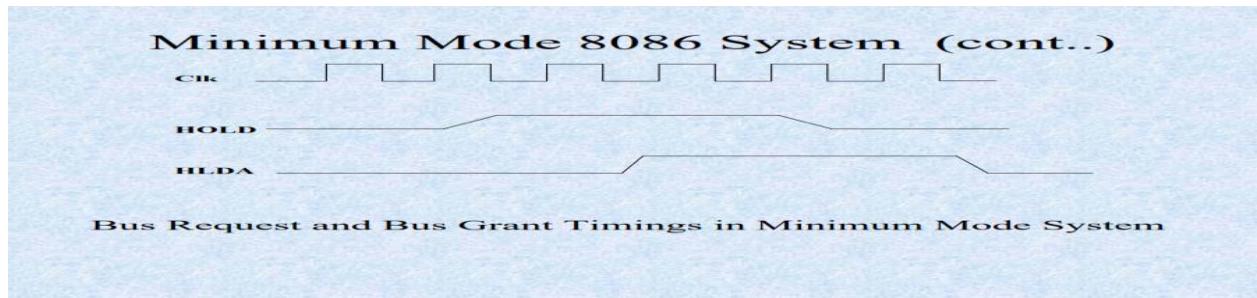
Write Cycle Timing Diagram for Minimum Mode

The HOLD pin is checked at leading edge of each clock pulse. **If it is received active by the processor before T4 of the previous cycle or during T1 state of the current cycle,** the CPU activates HLDA in the next clock cycle and for succeeding bus cycles, the bus will be given to another requesting master. The

control of the bus is not regained by the processor until the requesting master does not drop the HOLD pin low. When the request is dropped by the requesting master, the HLDA is dropped by the processor at the trailing edge of the next clock.

Minimum Mode 8086 System (cont..)

Bus Request and Bus Grant Timings in Minimum Mode System

Maximum Mode 8086:

In the maximum mode, the 8086 is operated by strapping the **MN/MX pin to ground.** In this mode, the processor derives the status signal S2, S1, S0.

Another chip called bus **controller** derives the control signal using this status information. **In the maximum mode, there may be more than one microprocessor in the system configuration**. The components in the system are same as in the minimum mode system. The basic function of the bus controller chip IC8288, is to derive control signals like RD and WR (for memory and I/O devices), DEN, DT/R, ALE etc. using the information by the processor on the status lines.
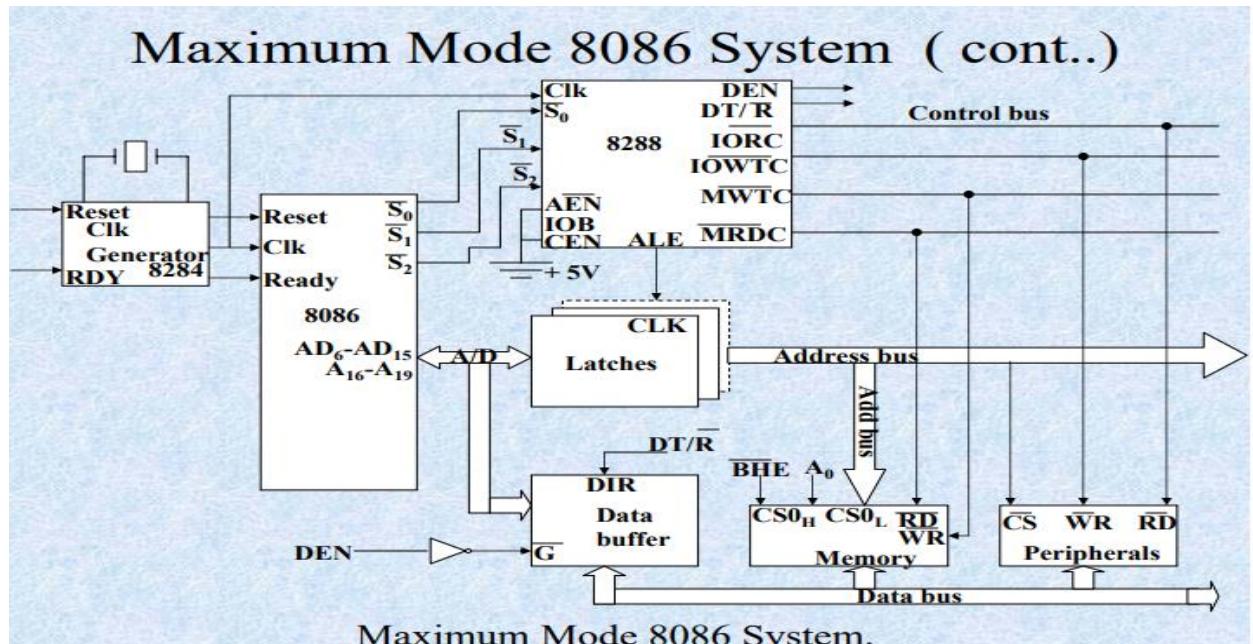
The bus controller chip has input lines S2, S1, S0 and CLK. These inputs to 8288 are driven by CPU. **It derives the outputs ALE, DEN, DT/R, MRDC, MWTC, AMWC, IORC, IOWC and AIOWC.**. INTA pin used to issue two

interrupt acknowledge pulses to the interrupt controller or to an interrupting device.

IORC, IOWC are I/O read command and I/O write command signals respectively . These signals enable an IO interface to read or write the data from or to the address port.

The MRDC, MWTC are memory read command and memory write command signals respectively and may be used as memory read or write signals. All these command signals instructs the memory to accept or send data from or to the bus.

For both of these write command signals, **the advanced signals namely AIOWC and AMWTC** are available. They also serve the same purpose, but are **activated one clock cycle earlier than the IOWC and MWTC** signals respectively.

Maximum Mode 8086 System.

Timing signals for Maximum Mode: Read Cycle:

The maximum mode system timing diagrams are divided in two portions as read (input) and write (output) timing diagrams. The address/data and address/status timings are similar to the minimum mode.
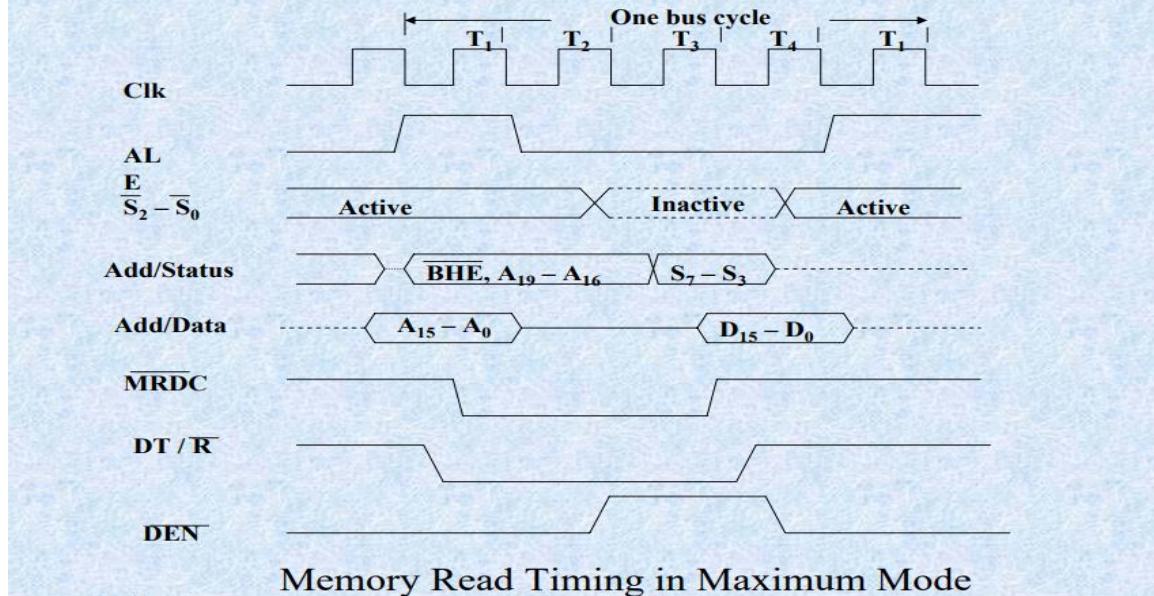
ALE is asserted in T1, just like minimum mode. The only difference lies in the status signal used and the available control and advanced command signals.

Here the only difference between in timing diagram between minimum mode and maximum mode **is the status signals used and the available control and advanced command signals.**
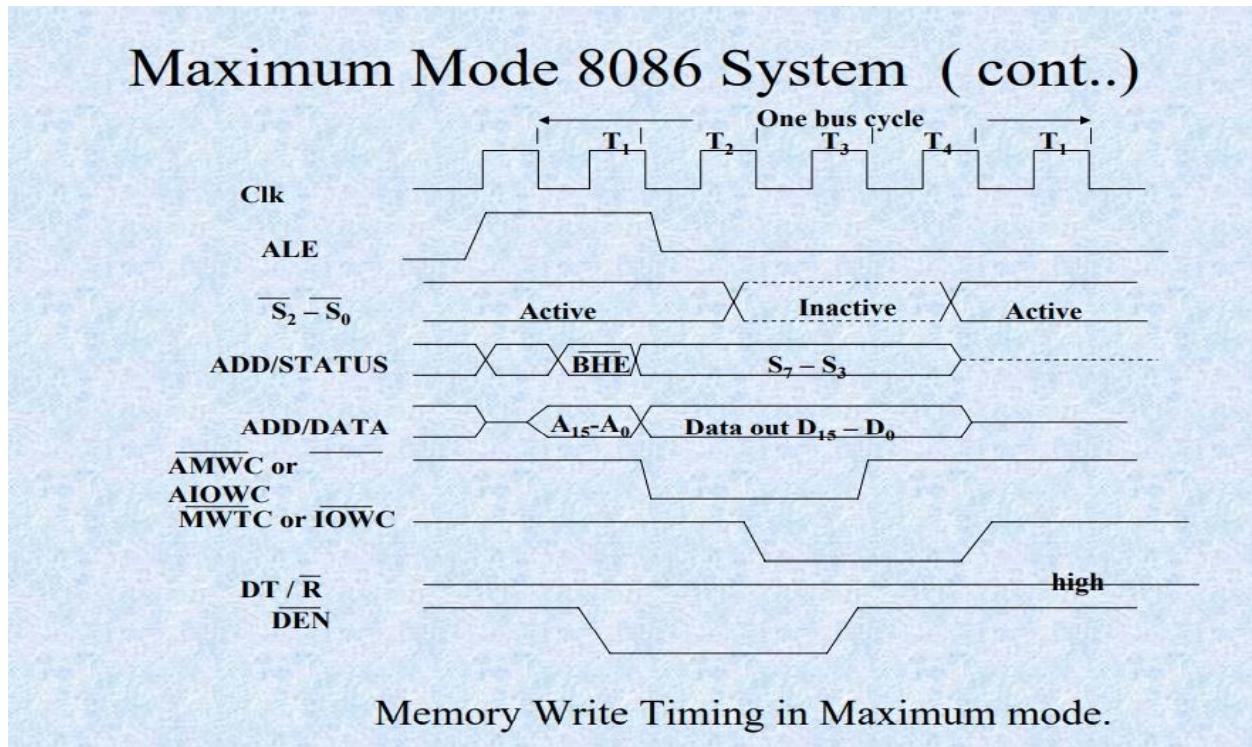
S0, S1, S2 are set at the beginning of bus cycle.8288 bus controller will output a pulse as on the ALE and apply a required signal to its DT / R pin during T1.

 In T2, 8288 will set DEN=1 thus enabling transceivers, and for an input it will activate MRDC or IORC. These signals are activated until T4. For an output, the AMWC or AIOWC is activated from T2 to T4 and☐ MWTC or IOWC is activated from T3 to T4. The status bit S0 to S2 remains active until T3 and become passive during T3 and T4

Memory Read Timing in Maximum Mode

Timing signals for Maximum Mode: Write Cycle:


Memory Write Timing in Maximum mode.
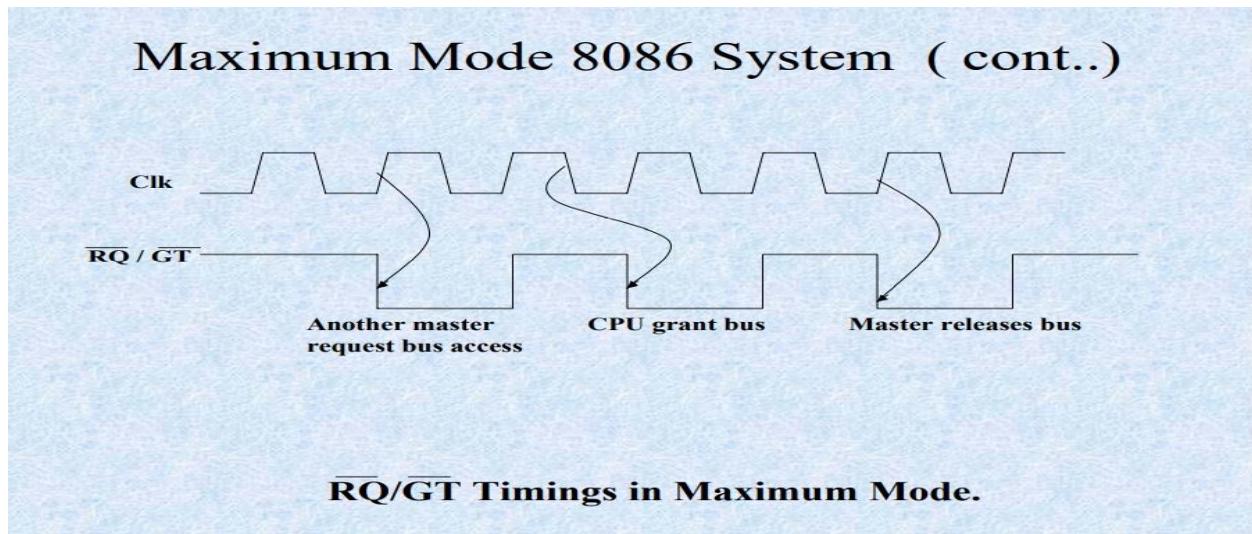
Timings for RQ/ GT Signals:

The request/grant response sequence contains a series of three pulses. The request/grant pins are checked at each rising pulse of clock input.

When a request is detected and if the condition for HOLD request are satisfied, the processor issues a grant pulse over the RQ/GT pin immediately during T4 (current) or T1 (next) state.

When the requesting master receives this pulse, it accepts the control of the bus, it sends a release pulse to the processor using RQ/GT pin.

Addressing modes of 8086

The way for which an operand is specified for an instruction in the accumulator, in a general purpose register or in memory location, is called **addressing mode**.

The 8086 microprocessors have 8 addressing modes. Two addressing modes have been provided for instructions which operate on register or immediate data.

These two addressing modes are:

Register Addressing: In register addressing, the operand is placed in one of the 16-bit or 8-bit general purpose registers.

Example

MOV AX, CX

ADD AL, BL

Immediate Addressing: In immediate addressing, the operand is specified in the instruction itself.

Example

MOV AL, 35H

MOV BX, 0301H

MOV [0401], 3598H

ADD AX, 4836H

The remaining 6 addressing modes specify the location of an operand which is placed in a memory.

These 6 addressing modes are:

Direct Addressing: In direct addressing mode, the operands offset is given in the instruction as an 8-bit or 16-bit displacement element.

Example

ADD AL, [0301]

The instruction adds the content of the offset address 0301 to AL. the operand is placed at the given offset (0301) within the data segment DS.

Register Indirect Addressing: The operand's offset is placed in any one of the registers BX, BP, SI or DI as specified in the instruction.

Example

MOV AX, [BX]

It moves the contents of memory locations addressed by the register BX to the register AX.

Based Addressing: The operand's offset is the sum of an 8-bit or 16-bit displacement and the contents of the base register BX or BP. BX is used as base register for data segment, and the BP is used as a base register for stack segment.

Effective address (Offset) = [BX + 8-bit or 16-bit displacement].

Example

MOV AL, [BX+05]; an example of 8-bit displacement.

MOV AL, [BX + 1346H]; example of 16-bit displacement.

Indexed Addressing: The offset of an operand is the sum of the content of an index register SI or DI and an 8-bit or 16-bit displacement.

Offset (Effective Address) = [SI or DI + 8-bit or 16-bit displacement]

Example

MOV AX, [SI + 05]; 8-bit displacement.

MOV AX, [SI + 1528H]; 16-bit displacement.

Based Indexed Addressing: The offset of operand is the sum of the content of a base register BX or BP and an index register SI or DI.

Effective Address (Offset) = [BX or BP] + [SI or DI]

Here, BX is used for a base register for data segment, and BP is used as a base register for stack segment.

Example

ADD AX, [BX + SI]

MOV CX, [BX + SI]

Based Indexed with Displacement: In this mode of addressing, the operand's offset is given by:

Effective Address (Offset) = [BX or BP] + [SI or DI] + 8-bit or 16-bit displacement

Example

MOV AX, [BX + SI + 05]; 8-bit displacement

MOV AX, [BX + SI + 1235H]; 16-bit displacement

Addressing Modes for control transfer instructions:

1. **Intersegment**

   a) Intersegment direct b) Intersegment indirect

2. **Intrasegment**

   a) Intrasegment direct b) Intrasegment indirect

1. (a) Intersegment direct: In this mode, the address to which the control is to be transferred is in a different segment. This addressing mode provides a means of branching from one code segment to another code segment. Here, the CS and IP of the destination address are specified directly in the instruction.

Example: JMP 5000H, 2000H;

Jump to effective address 2000H in segment 5000H.

1. (b) Intersegment indirect: In this mode, the address to which the control is to be transferred lies in a different segment and it is passed to the instruction indirectly, i.e. contents of a memory block containing four bytes, i.e. IP(LSB),

IP(MSB), CS(LSB) and CS(MSB) sequentially. The starting address of the memory block may be referred using any of the addressing modes, except immediate mode.

Example: JMP [2000H].

Jump to an address in the other segment specified at effective address 2000H in DS.

2.(a) Intrasegment direct mode: In this mode, the address to which the control is to be transferred lies in the same segment in which the control transfers instruction lies and appears directly in the instruction as an immediate displacement value. In this addressing mode, the displacement is computed relative to the content of the instruction pointer.

The effective address to which the control will be transferred is given by the sum of 8 or 16 bit displacement and current content of IP. In case of jump instruction, if the signed displacement (d) is of 8-bits (i.e. $-128 < d < +127$), it as short jump and if it is of 16 bits (i.e.

$-32768 < d < +32767$), it is termed as long jump.

Example: JMP SHORT LABEL.

2.(b) Intrasegment indirect mode: In this mode, the displacement to which the control is to be transferred is in the same segment in which the control transfer instruction lies, but it is passed to the instruction directly. Here, the branch address is found as the content of a register or a memory location.This addressing mode may be used in unconditional branch instructions.

Example: JMP [BX]; Jump to effective address stored in BX.

Instruction Set of 8086:

The sequence of commands used to tell a microcomputer what to do is called a program,

Each command in a program is called an instruction

The entire group of instructions that a microprocessor supports is called Instruction Set. 8086 has more than 20,000 instructions.

Classification of Instruction Set:

1. **Data Transfer Instructions**

2. **Arithmetic instructions**

3. **Bit Manipulation Instructions**

4. **Program Execution Transfer Instructions**

5. **String Instructions**

6. **Processor Control Instructions**

Data Transfer Instructions:

These instructions are used to transfer data from source to destination.The operand can be a constant, memory location, register or I/O port address.Instructions to transfer a word.

Data Transfer Instructions:

MOV

PUSH

POP

PUSHA

The MOV instruction copies a word or byte of data from a specified source to a specified destination

Data Transfer Instructions

- **MOV Des, Src:**
  - Src operand can be register, memory location or immediate operand.
  - Des can be register or memory operand.
  - Both Src and Des cannot be memory location at the same time.
  - E.g.:
    - MOV CX, 037A H
    - MOV AL, BL
    - MOV BX, [0301 H]

MOV:

- **MOV–**
Move byte or word to register or memory

MOV Destination, Source

- MOV CX, 045FH
- MOV BL, [43E4H]
- MOV AX, DX
- MOV DH, [BX]
- MOV DS, BX

PUSH:

- **PUSH Operand:**

  - It pushes the operand into top of stack.

  - E.g.: PUSH BX

- **POP Des:**

  - It pops the operand from top of stack to Des.

  - Des can be a general purpose register, segment register (except CS) or memory location.

  - E.g.: POP AX

XCHG

- **XCHG Des, Src:**

  - This instruction exchanges Src with Des.

  - It cannot exchange two memory locations directly.

  - E.g.: XCHG DX, AX

Instructions for input and output port transfer

- **IN Accumulator, Port Address:**

  - It transfers the operand from specified port to accumulator register.

  - E.g.: IN AX, 0028 H

- **OUT Port Address, Accumulator:**

  - It transfers the operand from accumulator to specified port.

  - E.g.: OUT 0028 H, AX

LOAD INSTRUCTION

● **LEA Register, Src:**

  ● It loads a 16-bit register with the offset address of the data specified by the Src.

  ● E.g.: LEA BX, [DI]

    ● This instruction loads the contents of DI (offset) into the BX register.

Arithmetic Instructions:

This instructions are use to perform the arithmetic operations like +,-,*,/,etc.

Instructions to perform addition

● **ADD Des, Src:**

  ● It adds a byte to byte or a word to word.

  ● It effects AF, CF, OF, PF, SF, ZF flags.

  ● E.g.:

    - ADD AL, 74H
    - ADD DX, AX
    - ADD AX, [BX]

● **ADC Des, Src:**

  ● It adds the two operands with CF.

  ● It effects AF, CF, OF, PF, SF, ZF flags.

  ● E.g.:

    - ADC AL, 74H
    - ADC DX, AX
    - ADC AX, [BX]

- **SUB Des, Src:**
  - It subtracts a byte from byte or a word from word.
  - It effects AF, CF, OF, PF, SF, ZF flags.
  - For subtraction, CF acts as borrow flag.
  - E.g.:
    - SUB AL, 74H
    - SUB DX, AX
    - SUB AX, [BX]

- **SBB Des, Src:**
  - It subtracts the two operands and also the borrow from the result.
  - It effects AF, CF, OF, PF, SF, ZF flags.
  - E.g.:
    - SBB AL, 74H
    - SBB DX, AX
    - SBB AX, [BX]

- **INC Src:**
  - It increments the byte or word by one.
  - The operand can be a register or memory location.
  - It effects AF, OF, PF, SF, ZF flags.
  - CF is not effected.
  - E.g.: INC AX

- **DEC Src:**
  - It decrements the byte or word by one.
  - The operand can be a register or memory location.
  - It effects AF, OF, PF, SF, ZF flags.
  - CF is not effected.
  - E.g.: DEC AX

- **AAA (ASCII Adjust after Addition):**
  - The data entered from the terminal is in ASCII format.
  - In ASCII, 0 – 9 are represented by 30H – 39H.
  - This instruction allows us to add the ASCII codes.
  - This instruction does not have any operand.
- **Other ASCII Instructions:**
  - **AAS** (ASCII Adjust after Subtraction)
  - **AAM** (ASCII Adjust after Multiplication)
  - **AAD** (ASCII Adjust Before Division)

- **DAA (Decimal Adjust after Addition)**

  - It is used to make sure that the result of adding two BCD numbers is adjusted to be a correct BCD number.

  - It only works on AL register.

- **DAS (Decimal Adjust after Subtraction)**

  - It is used to make sure that the result of subtracting two BCD numbers is adjusted to be a correct BCD number.

  - It only works on AL register.

- **NEG Src:**

  - It creates 2's complement of a given number.

  - That means, it changes the sign of a number.

- **CMP Des, Src:**
  - It compares two specified bytes or words.
  - The Src and Des can be a constant, register or memory location.
  - Both operands cannot be a memory location at the same time.
  - The comparison is done simply by internally subtracting the source from destination.
  - The value of source and destination does not change, but the flags are modified to indicate the result.

- **MUL Src:**
  - It is an unsigned multiplication instruction.
  - It multiplies two bytes to produce a word or two words to produce a double word.
  - AX = AL * Src
  - DX : AX = AX * Src
  - This instruction assumes one of the operand in AL or AX.
  - Src can be a register or memory location.

- **IMUL Src:**
  - It is a signed multiplication instruction.

- **DIV Src:**
  - It is an unsigned division instruction.
  - It divides word by byte or double word by word.
  - The operand is stored in AX, divisor is Src and the result is stored as:
    - AH = remainder   AL = quotient
- **IDIV Src:**
  - It is a signed division instruction.

- **CBW (Convert Byte to Word):**
  - This instruction converts byte in AL to word in AX.
  - The conversion is done by extending the sign bit of AL throughout AH.
- **CWD (Convert Word to Double Word):**
  - This instruction converts word in AX to double word in DX : AX.
  - The conversion is done by extending the sign bit of AX throughout DX.

Bit Manipulation Instructions:

These instructions are used at the bit level i.e. operations like logic &shift etc

These instructions can be used for:

Testing a zero bit

Set or reset a bit

Shift bits across registers

Logical	Shift	Rotate
• NOT	• SAL	• ROL
• AND	• SHL	• RCL
• OR	• SAR	• ROR
• XOR	• SHR	• RCR
• TEST		

- **NOT Src:**
  - It complements each bit of Src to produce 1's complement of the specified operand.
  - The operand can be a register or memory location.

    NOT Destination

    - NOT BX

• AND Des, Src:

- • It performs AND operation of Des and Src.

- • Src can be immediate number, register or memory location.

- • Des can be register or memory location.

- • Both operands cannot be memory locations at the same time.

- • CF and OF become zero after the operation.

- • PF, SF and ZF are updated.

AND Destination, Source

- • AND BH, CL
- • AND CX, [SI]
- • AND BX, 00FFH
- • AND DX, BX

• OR Des, Src:

- • It performs OR operation of Des and Src.

- • Src can be immediate number, register or memory location.

- • Des can be register or memory location.

- • Both operands cannot be memory locations at the same time.

- • CF and OF become zero after the operation.

- • PF, SF and ZF are updated.

• XOR Des, Src:

- It performs XOR operation of Des and Src.

- Src can be immediate number, register or memory location.

- Des can be register or memory location.

- Both operands cannot be memory locations at the same time.

- CF and OF become zero after the operation.

- PF, SF and ZF are updated.

XOR Destination, Source
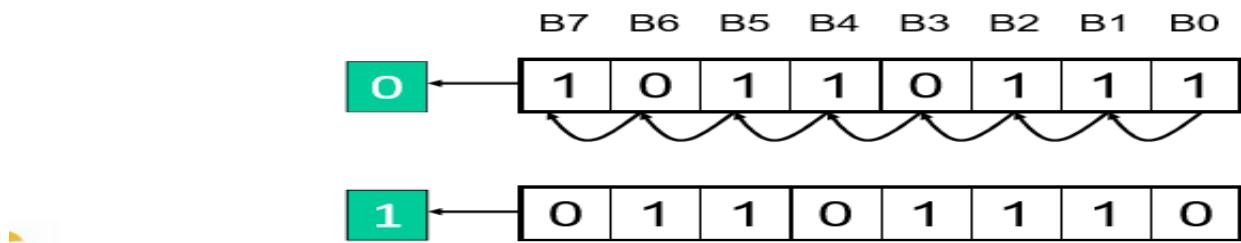
- XOR BH, CL
- XOR BP, DI
- XOR DX, BX

• SHL Des, Count:

- It shift bits of byte or word left, by count.

- It puts zero(s) in LSBs.

- MSB is shifted into carry flag.

- If the number of bits desired to be shifted is 1, then the immediate number 1 can be written in Count.

- However, if the number of bits to be shifted is more than 1, then the count is put in CL register.
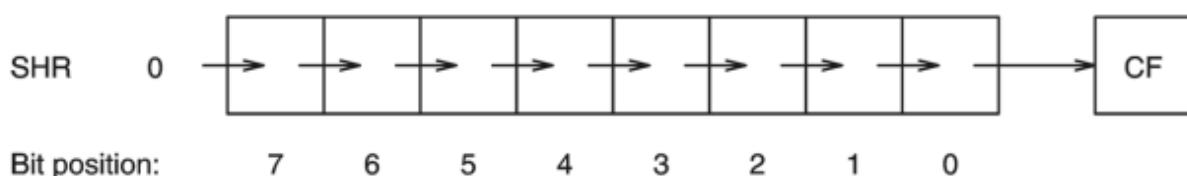
SAL / SHL Destination, Count

- SAL BX, 01
- SAL BP, CL
- MOV CL, 04H
- SAL AL, CL

	B7	B6	B5	B4	B3	B2	B1	B0
0	1	0	1	1	0	1	1	1

1	0	1	1	0	1	1	1	0
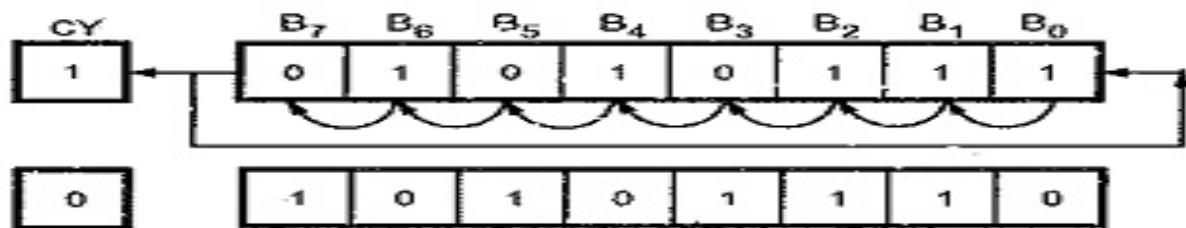
• SHR Des, Count:

- It shift bits of byte or word right, by count.

- It puts zero(s) in MSBs.

- LSB is shifted into carry flag.

- If the number of bits desired to be shifted is 1, then the immediate number 1 can be written in Count.

- However, if the number of bits to be shifted is more than 1, then the count is put in CL register.

SHR   0   →→→→→→→→→ CF

Bit position: 7   6   5   4   3   2   1   0
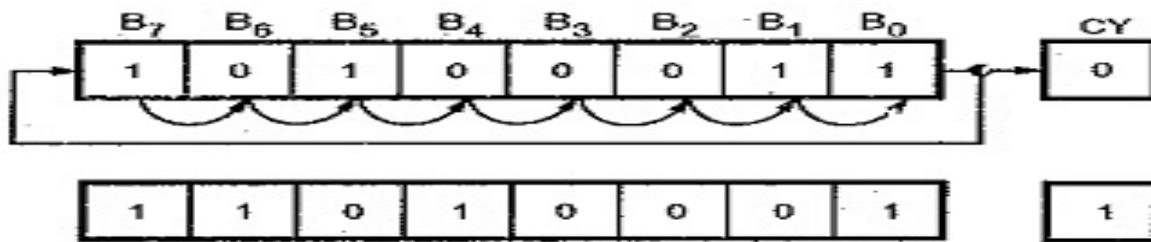
- ## ROL Des, Count:

  - It rotates bits of byte or word left, by count.

  - MSB is transferred to LSB and also to CF.

  - If the number of bits desired to be shifted is 1, then the immediate number 1 can be written in Count.

  - However, if the number of bits to be shifted is more than 1, then the count is put in CL register.
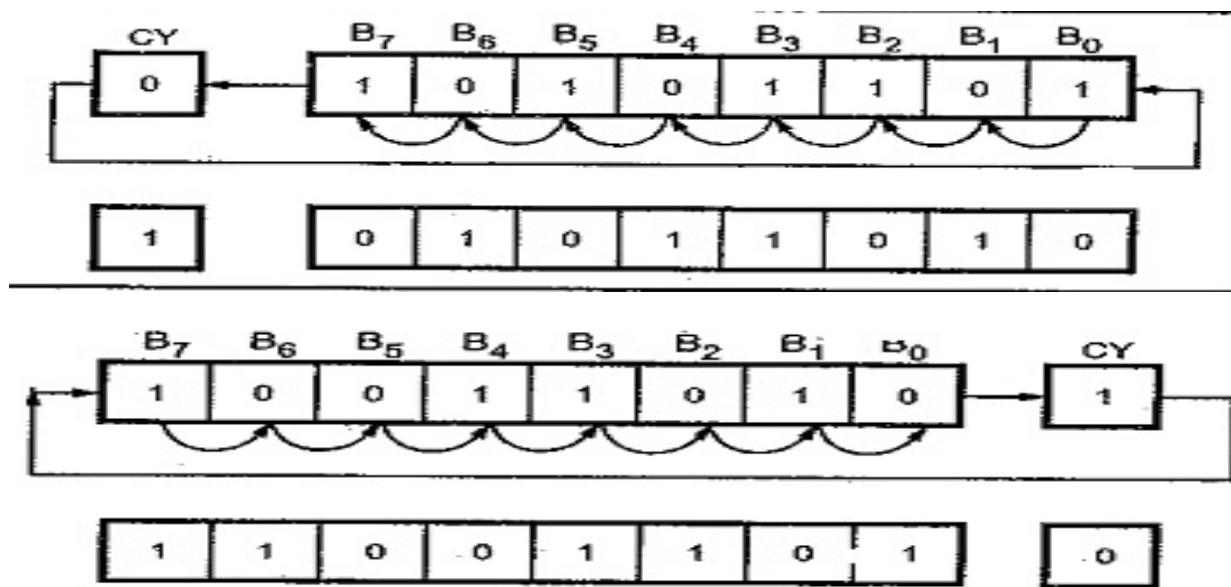

- ## ROR Des, Count:

  - It rotates bits of byte or word right, by count.

  - LSB is transferred to MSB and also to CF.

  - If the number of bits desired to be shifted is 1, then the immediate number 1 can be written in Count.

 - However, if the number of bits to be shifted is more than 1, then the count is put in CL register.

RCL Instruction : RCL destination, count.

RCR Instruction : RCR destination, count.

Program Execution Transfer Instructions:

These instructions cause change in the sequence of the execution of instruction. This change can be through a condition or sometimes unconditional. The conditions are represented by flags.

- **CALL Des:**
  - This instruction is used to call a subroutine or function or procedure.
  - The address of next instruction after CALL is saved onto stack.

- **RET:**
  - It returns the control from procedure to calling program.
  - Every CALL instruction should have a RET.

- **JMP Des:**
  - This instruction is used for unconditional jump from one place to another.

- **Jxx Des (Conditional Jump):**
  - All the conditional jumps follow some conditional statements or any instruction that affects the flag.

- **Loop Des:**
  - This is a looping instruction.
  - The number of times looping is required is placed in the CX register.
  - With each iteration, the contents of CX are decremented.
  - ZF is checked whether to loop again or not.

Conditional Jump Table

Mnemonic	Meaning	Jump Condition
JA	Jump if Above	CF = 0 and ZF = 0
JAE	Jump if Above or Equal	CF = 0
JB	Jump if Below	CF = 1
JBE	Jump if Below or Equal	CF = 1 or ZF = 1
JC	Jump if Carry	CF = 1
JE	Jump if Equal	ZF = 1
JNC	Jump if Not Carry	CF = 0
JNE	Jump if Not Equal	ZF = 0
JNZ	Jump if Not Zero	ZF = 0
JPE	Jump if Parity Even	PF = 1
JPO	Jump if Parity Odd	PF = 0
JZ	Jump if Zero	ZF = 1

String Instructions

String in assembly language is just a sequentially stored bytes or words.

There are very strong set of string instructions in 8086.

By using these string instructions, the size of the program is considerably reduced.

- **CMPS Des, Src:**
  - It compares the string bytes or words.

- **SCAS String:**
  - It scans a string.
  - It compares the String with byte in AL or with word in AX.

- **MOVS / MOVSB / MOVSW:**
  - It causes moving of byte or word from one string to another.
  - In this instruction, the source string is in Data Segment and destination string is in Extra Segment.
  - SI and DI store the offset values for source and destination index.

- **REP (Repeat):**
  - This is an instruction prefix.
  - It causes the repetition of the instruction until CX becomes zero.
  - E.g.: REP MOVSB STR1, STR2
    - It copies byte by byte contents.
    - REP repeats the operation MOVSB until CX becomes zero.

Processor Control Instructions:

These instructions control the processor itself.8086 allows to control certain control flags that:

Causes the processing in a certain direction processor synchronization if more than one microprocessor attached.

- **STC:**
  - It sets the carry flag to 1.

- **CLC:**
  - It clears the carry flag to 0.

- **CMC:**
  - It complements the carry flag.

- **STD:**
  - It sets the direction flag to 1.
  - If it is set, string bytes are accessed from higher memory address to lower memory address.

- **CLD:**
  - It clears the direction flag to 0.
  - If it is reset, the string bytes are accessed from lower memory address to higher memory address.

Assembler Directives:

There are some instructions in the assembly language program which are not a part of processor instruction set.

These instructions are instructions to the assembler, linker and loader.

These are referred to as pseudo-instructions or as assembler directives.

The assembler directives **enable us to control the way in which a program assembles and lists.**

They act during the assembly of a program and do not generate any executable machine code

ASSUME Directive

The ASSUME directive is used to tell the assembler that the name of the logical segment should be used for a specified segment.

The 8086 works directly with only 4 physical segments: a Code segment, a data segment, a stack segment, and an extra segment.

 Example:

1. ASUME CS:CODE ;This tells the assembler that the logical segment named CODE contains the instruction statements for the program and should be treated as a code segment.

2. ASUME DS:DATA ;This tells the assembler that for any instruction which refers to a data in the data segment, data will found in the logical segment DATA.

SEGMENT:

The SEGMENT directive is used to indicate the start of a logical segment.

Preceding the SEGMENT directive is the name you want to give the segment.

For example, the statement CODE SEGMENT indicates to the assembler the start of a logical segment called CODE.

The SEGMENT and ENDS directive are used to "bracket" a logical segment containing code of data.

ENDS (END SEGMENT):

This directive is used with the name of a segment to indicate the end of that logical segment.

CODE SEGMENT: Start of logical segment containing code instruction statements

CODE ENDS: End of segment named CODE

DB – Define Byte:

DB directive is used to declare a byte type variable or to store a byte in memory location.

Example:

1. PRICE DB 49h, 98h, 29h ;Declare an array of 3 bytes, named as PRICE and initialize.

2. NAME DB 'ABCDEF';Declare an array of 6 bytes and initialize with ASCII code for letters

3. TEMP DB 100 DUP (?) ;Set 100 bytes of storage in memory and give it the name as TEMP, but leave the 100 bytes uninitialized. Program instructions will load values into these locations.

DW – Define Word

The DW directive is used to define a variable of type word or to reserve storage location of type word in memory.

Example:

MULTIPLIER DW 437Ah ; this declares a variable of type word and named it as MULTIPLIER. This variable is initialized with the value 437Ah when it is loaded into memory to run.

EXP1 DW 1234h, 3456h, 5678h ; this declares an array of 3 words and initialized with specified values.

STOR1 DW 100 DUP (0); Reserve an array of 100 words of memory and initialize all words with 0000.Array is named as STOR1.

DD (DEFINE DOUBLE WORD):

The DD directive is used to declare a variable of type double word or to reserve memory locations, which can be accessed as type double word.

Example

1. ARRAY DD 25629261H, will define a double word named ARRAY and initialize the double word with the specified value when the program is loaded into memory to be run. The low word, 9261H, will be put in memory at a lower address than the high word.

DQ (DEFINE QUADWORD):

The DQ directive is used to tell the assembler to declare a variable 4 words in length or to reserve 4 words of storage in memory.

Example

1. BIG_NUMBER DQ 243598740192A92BH will declare a variable named BIG_NUMBER and initialize the 4 words set aside with the specified number when the program is loaded into memory to be run.

DT (DEFINE TEN BYTES):

The DT directive is used to tell the assembler to declare a variable, which is 10 bytes in length or to reserve 10 bytes of storage in memory.

Example

1.PACKED_BCD DT 11223344556677889900 will declare an array named PACKED_BCD, which is 10 bytes in length. It will initialize the 10 bytes with the

values 11, 22, 33, 44, 55, 66, 77, 88, 99, and 00 when the program is loaded into memory to be run.

2. RESULT DT 20H DUP (0) will declare an array of 20H blocks of 10 bytes each and initialize all 20 bytes to 00 when the program is loaded into memory to be run.

END – END Directive

END directive is placed after the last statement of a program to tell the assembler that this is the end of the program module.

The assembler will ignore any statement after an END directive.

ENDP – END PROCEDURE

ENDP directive is used along with the name of the procedure to indicate the end of a procedure to the assembler

Example:

1. SQUARE_NUM PROCE ; It start the procedure ;Some steps to find the square root of a number

2. SQUARE_NUM ENDP ;Hear it is the End for the procedure

EQU (EQUATE):

EQU is used to give a name to some value or symbol.

Each time the assembler finds the given name in the program, it replaces the name with the value or symbol you equated with that name.

Example:

1. FACTOR EQU 03H ;at the start of your program, and later in the program you write the instruction statement ADD AL, FACTOR. When the assembler codes this instruction statement, it will code it as if you had written the instruction ADD AL, 03H.

LENGTH:

 LENGTH is an operator, which tells the assembler to determine the number of elements in some named data item, such as a string or an array. Example:

1. MOV CX, LENGTH STRING1, for example, will determine the number of elements in STRING1 and load it into CX.

EVEN

This EVEN directive instructs the assembler to increment the location of the counter to the next even address if it is not already in the even address.

If the word is at even address 8086 can read a memory in 1 bus cycle.

If the word starts at an odd address, the 8086 will take 2 bus cycles to get the data.

A series of words can be read much more quickly if they are at even address.

When EVEN is used the location counter will simply incremented to next address and NOP instruction is inserted in that incremented location

Example:

DATA1 SEGMENT; Location counter will point to 0009 after assembler reads ;next statement

SALES DB 9 DUP(?) ;declare an array of 9 bytes

EVEN ; increment location counter to 000AH

RECORD DW 100 DUP(0) ;Array of 100 words will start ;from an even address for quicker read DATA1 ENDS

PROC

The PROC directive is used to identify the start of a procedure. The term near or far is used to specify the type of the procedure.

Example:

SMART PROC FAR ; This identifies that the start of a procedure named as SMART and instructs the assembler that the procedure is far .

SMART ENDP

This PROC is used with ENDP to indicate the break of the procedure.

PTR

This PTR operator is used to assign a specific type of a variable or to a label.

Example:

1. INC [BX] ; This instruction will not know whether to increment the byte pointed to by BX or a word pointed to by BX.

PUBLIC

The PUBLIC directive is used to instruct the assembler that a specified name or label will be accessed from other modules.

 Example:

PUBLIC DIVISOR, DIVIDEND; these two variables are public so these are available to all modules.

EXTRN

If an instruction in a module refers to a variable in another assembly module, we can access that module by declaring as EXTRN directive.
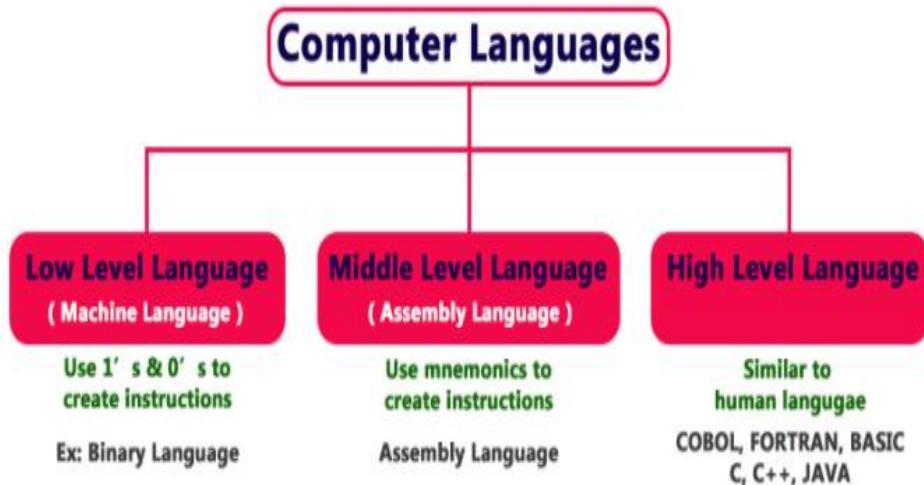
UNIT-III
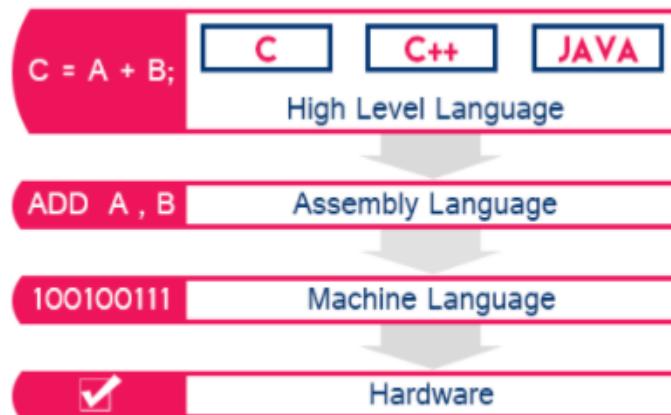
**Assembly Language Programming with 8086**-
Programming with an assembler, Assembly Language example programs. Stack structure of 8086, Interrupts and Interrupt service routines, Interrupt cycle of 8086, passing parameters to procedures, Macros.

Over the years, computer languages have been evolved from Low-Level to High-Level Languages.

In the earliest days of computers, only Binary Language was used to write programs. The computer languages are classified as follows:

The examples of instructions for different languages can be given as:


Machine Language(Low level language):
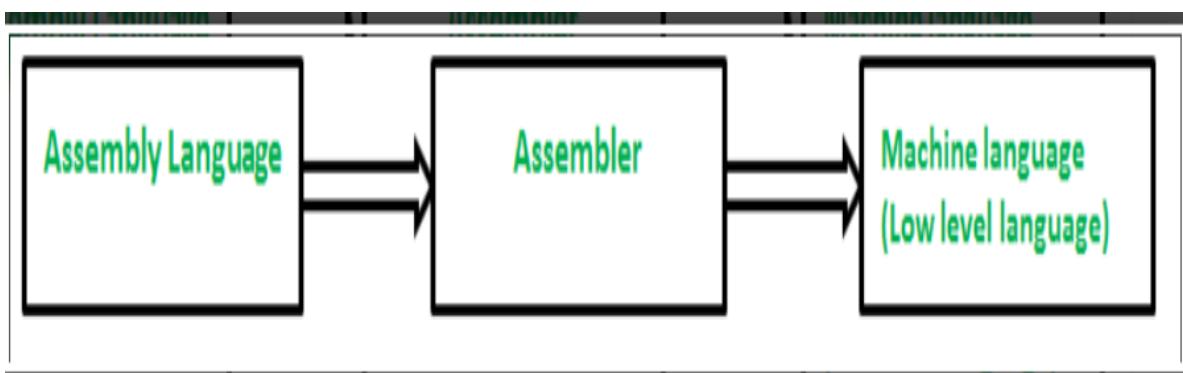
Low-Level language is the only language which can be understood by the computer. Low-level language is also known as **Machine Language**.

The machine language contains only two symbols **1 & 0**. All the instructions of machine language are written in the form of binary numbers 1's & 0's. A computer can directly understand the machine language.

Assembly Language

- *'Assembly Language'* is the human readable notation of *'machine language'*
- *'Machine language'* is a processor understandable language.
- Processors deal only with binaries (1s and 0s).
- Machine language is a binary representation and it consists of 1s and 0s.

- Machine language is made readable by using specific symbols called **"mnemonics"** in Assembly Language.
- Assembly language programming is the process of writing processor specific machine code in mnemonic form.
- **Assembler:** Converting the mnemonics into actual processor instructions (machine language) and associated data using an assembler.

Machine Language	Assembly Language
Machine language is only understand by the computers.	Assembly language is only understand by human beings not by the computers.
In machine language data only represented with the help of binary format(0s and 1s), hexadecimal and octadecimal.	In assembly language data can be represented with the help of mnemonics such as Mov, Add, Sub, End etc.
Machine language is very difficult to understand by the human beings.	Assembly language is easy to understand by the human being as compare to machine language.
Modifications and error fixing cannot be done in machine language.	Modifications and error fixing can be done in assembly language.
Machine language is very difficult to memorize so it is not possible to learn the machine language.	Easy to memorize the assembly language because some alphabets and mnemonics are used.
Execution is fast in machine language because all data is already present in binary format.	Execution is slow as compared to machine language.
There is no need of translator.The machine understandable form is the machine language.	Assembler is used as translator to convert mnemonics into machine understandable form.
Machine language is hardware dependent.	Assembly language is the machine dependent and it is not portable.

Instruction Format for ALP

- The general format of an assembly language instruction is an Opcode followed by Operands

Opcode   operand1, operand2

EX: MOV A, #30
- This instruction mnemonic moves decimal value 30 to the 8086 Accumulator register.
- MOV A ----- Opcode
- 30----------- Operand(Single Operand)
- The same instruction when written in machine language will look like
    - 01110100   00011110
    - First 8 bit binary value represent the opcode MOV A
    - The 2nd 8 bit binary value represent the operand 30.

➢ Each line of an assembly language program is split into four fields as:

LABEL   OPCODE   OPERAND COMMENTS

LABEL:
- A Label is an optional field
- Labels are symbolic names which are used to "identify"
- Label is commonly used for representing
    - A memory location, address of a program, sub-routine, code portion etc
    - Assembler insist strict format for labeling
    - Labels are always suffixed by colon(:)
    - Labels begin with a valid character ; labels can contain numbers from 0 to 9 and special character_(underscore)

OPCODE:
- The Opcode tells the processor/controller what operations it has to do

Operands:
- The Operands provide the data and information required to perform the action specified by the opcode. It is not necessary that all opcode should have Operands following them.
-

COMMENT:
- The **symbol;** represents the start of a comment. Assembler ignores the text in a line after the ; symbol while assembling the program

Example:

```
;########################################################
DELAY: MOV R0, #255 ;    Load Register R0 with 255
DJNZ R1, DELAY;          Decrement R1 and loop till R1= 0
RET ;                    Return to calling program
;########################################################
                     ; SUBROUTINE FOR GENERATING DELAY
                     ; DELAY PARAMETR PASSED THROUGH REGISTER R1
                     ; RETURN VALUE NONE,REGISTERS USED: R0, R1
```
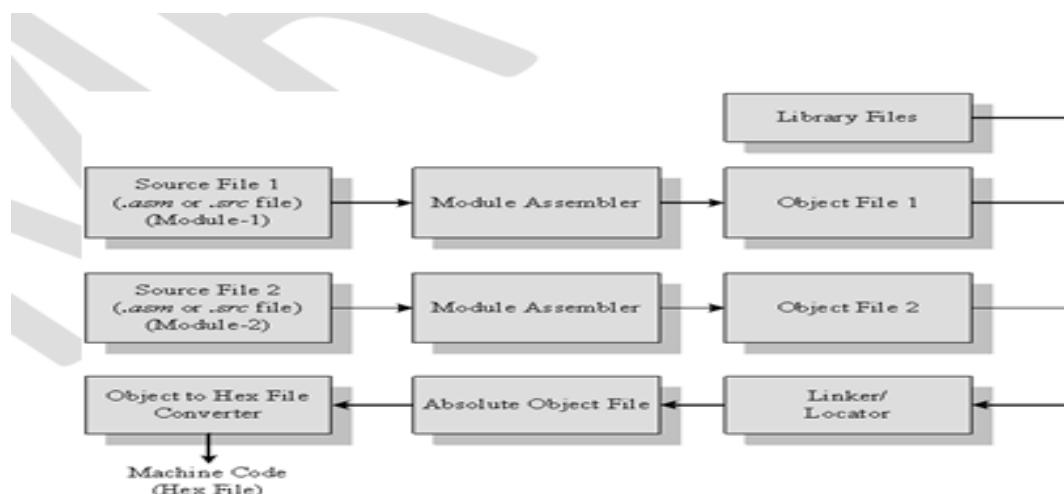
PROGRMMING WITH AN ASSEMBLER

- The Assembler performs the task of coding.
- An Assembler converts the mnemonics of instruction along with data into their equivalent object codes.
- Assembler is a program that converts an assembly input file called as source file to an object file.
- In Assembly language programming, the mnemonics are directly used in the user programs.
- **Linker and Loader:** Converts the object codes into an executable code.

Advantages of Assembly Language

- The programming is easy as compared to machine language because the **function of coding is performed by the assembler.**
- The chances of error being committed are less because mnemonics are used instead of numerical opcodes.
- As the mnemonics are purpose suggestive, the Debugging is easier.
- The constants and address locations can be labeled with suggestive labels, hence imparting a more friendly interface to the user.
- The memory control is in the hands of user and the results may be stored in a more user friendly form.

Converting Source file to Executable file

- The Assembly language programming is done by one of the popular assemblers called as **MASM(Microsoft Macro Assembler).**
- There are number of assemblers available like MASM , TASM & DOS assembler.
- MASM can be used along with a LINK program to structure the codes generated by an MASM in the form of an executable file.
- MASM reads **source program as an input and provides object files as output.**
- The LINK accepts the object file produced by the MASM and produces an EXE file.

Text Editor:

- While writing a program for assembler, the first step to be considered is the **Text editor**
- In the text editor, one can type the program and check the listing typed for any typing mistake and syntax error.
- Before quitting the program, one has to save the program.
- After saving the text file with any name, one is free to start the Assembly process.
- There are number of text editors are available in the market like Norton Editor[NE] , Turbo C & Edlin etc..
- Throughout this chapter, the NE is used.
- Thus for writing a program in assembly language one need **NE editor, MASM Assembler, Linker and Debug utility of DOS(Disk Operating System)**

Steps Involved in Assembly Program Development:

- In the following section, the procedure for opening a file for a program, assembling it, executing it and checking its results are described.
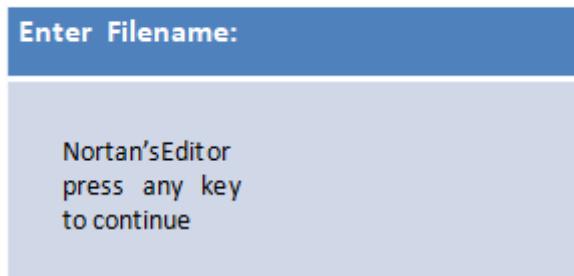
> **1. Entering a Program**
> **2. Assembling a program**
> **3. Linking a program**
> **4. Using DEBUG**

- Before starting the process, ensure that all files namely NE.COM(Nortan'sEditor),MASM.EXE(Assembler),LINK.EXE(Linker), DEBUG.EXE(Debugger) are available in the same directory in which you are working.

1. Entering a Program:
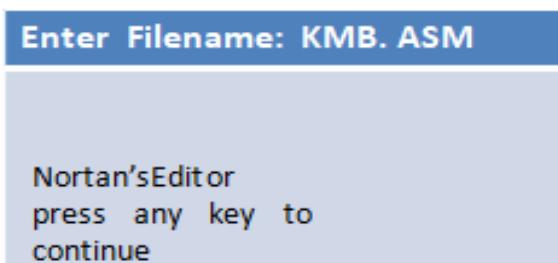
- Start the procedure with the following command after you boot the terminal and enter the directory containing all the files mentioned
- You will get a display as shown in figure

<p align="center">**C>NE**</p>

Nortan's Editor opening screen

- Suppose one types filename as **KMB.ASM as filename** the screen will display as shown in figure.

Nortan's Editor Alternative

- Press any of the keys you will get a display as shown in figure

Nortan's Editor opens a new file KMB.ASM

- Note that, every Assembly Language program, the extension **.ASM** must be there.
- The extension **.ASM** shows that it is an Assembly Language program file.
- Even if you type the file name without the .ASM extension, the assembler searches for the file and if it is not found issues the command 'File not found'.
- We can type the another type of command line, to get the same display

C> NE KMB.ASM

Nortan's Editor opens a new file KMB.ASM

- **You can modify or save the file KMB.ASM with the command F3-E.**
- Otherwise, simply quit **the file to abandon the changes and exit NE with the command F3-Q.**
- Once the above procedure is completed, you may now focus on assembling the program.
- Note that all the commands and the displays shown in the above section are for Nortan's Editor.
- Note that be for quitting the editor program, the modified file should be saved, otherwise it will be lost.

A program for KMB.ASM is shown in figure

```
ASSUME          CS:CODE, DS:DATA
DATA            SEGMENT
                OPR1 DW 1234 H
                OPR2 DW 0002 H
                RESULT DW 01 H DUP(?)

DATA            ENDS
CODE            SEGMENT
START           MOV AX, DATA
                MOV DS, AX
                MOV AX, OPR 1
                MOV BX, OPR 2
                CLC
                ADD AX, BX
                MOV DI, OFFSET RESULT
                MOV [DI], AX
                MOV AH, 4CH
                INT 21 H

CODE            ENDS
                END START
```

KBM.ASM
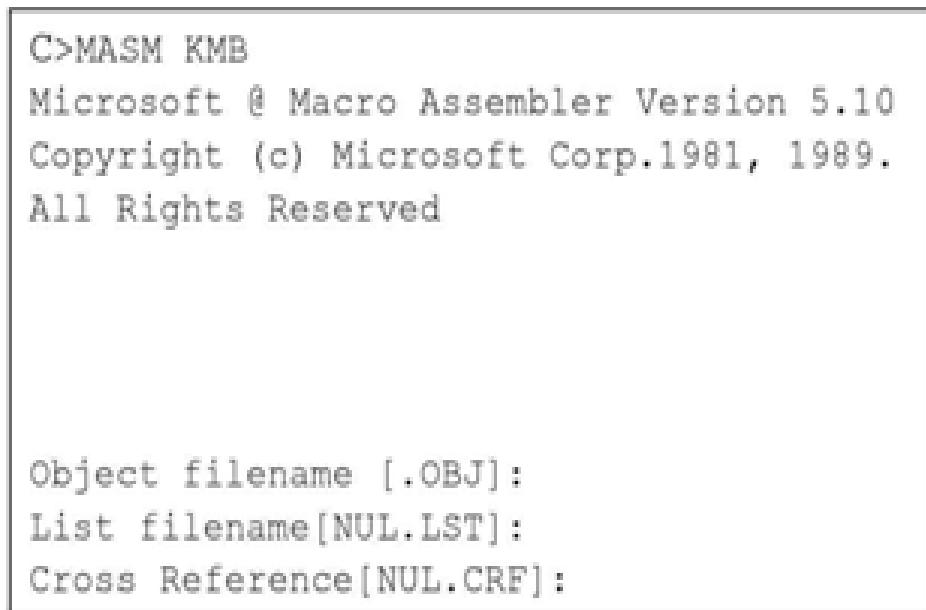
Fig. 3.9   A Program KMB.ASM in Norton's Editor

2. Assembling a program

- **Microsoft Assembler MASM** is easy to use and popular Assembler.
- The main task of **the Assembler program** is to accept the text-assembly language program file as an input and prepare an object file.
- The text-assembly language program file is prepared by using any of the editors program like NE.
- The **MASM** accepts **the file name only with the extension .ASM** Even if the filename without any extension is given as input, it provides .ASM extension to it.
- To assemble the program one may enter the following command

<div align="center">

C>MASM KMB
Or
C>MASM KMB.ASM

</div>

- If any of the above command is entered, the screen displays as shown in figure

```
C>MASM KMB
Microsoft @ Macro Assembler Version 5.10
Copyright (c) Microsoft Corp.1981, 1989.
All Rights Reserved




Object filename [.OBJ]:
List filename[NUL.LST]:
Cross Reference[NUL.CRF]:
```

Fig. 3.10   MASM Screen Display

- Another command line is available in MASM that does not need any file name in the command line, is given along with the corresponding display.
- If you do not enter the filename as shown in figure 3.10 ,then you may enter it as a source filename as shown in figure 3.11
- The source filename is to be typed in the source filename with or without the extension .ASM
- In the next line, the expected .OBJ filename is to be entered which creates the object file of the ALP.

```
C>MASM
Microsoft @ Macro Assembler Version 5.10
Copyright (c) Microsoft Corp.1981, 1989.
All Rights Reserved




Source filename [.ASM]:
Object filename [FILE.OBJ]:
List filename[NUL.LST]:
List filename[NUL.CRF]:
```

Fig. 3.11 MASM Alternative Screen Display

- The .OBJ file is created with the entered name and the .OBJ extension.
- If no filename is entered for it, before pressing any key, the new .OBJ file is created with the same name as the source file and extension .OBJ
- **The .OBJ file contains the coded object modules of the program to be assembled.**
- On the next line, a filename is entered for the expected listing file of the source file , in the same way as the object filename was entered.
- **The Listing file is automatically generated in the Assembly process.**
- The listing file is identified by the entered source filename and extension .LST.
- Listing file contains **the total offset map of the source files including labels, offset addresses,opcodes, memory allotment for different labels and directives and relocation information.**
- The **Cross reference file** name is also entered in the same way as for listing file.
- **Cross reference file** file is used for debugging the source program.
- It contains the **statistical information like size of the file in the bytes, number of labels, list of labels, routines to be called, etc.about the source program**.
- After the cross reference file name is entered the assembly process starts.
- If the program contains **syntax errors**, they are displayed using error code number and the corresponding line number at which they appear.
- Once these syntax errors and warnings are taken care of by the programmer, the assembly process is completed successfully.
- The successful assembly process **may generate the .OBJ, .LST, and CRF files** which may be further used by the linker programmer to link the object files and generate an executable file(.EXE) form a object file .OBJ
- The file generated by the MASM are further used by the program LINK.EXE to generate an executable file of the source program

3. Linking a program

- The DOS linking program LINK.EXE links the different object modules of a source program and function library routines to generate an executable code of the source program.
- The main input to the linker is the .OBJ file that contains the object modules of the source program
- The linker program is invoked by the following options

<p style="text-align:center">C>LINK
Or
C>LINK KMB.OBJ</p>

- The **.OBJ** extension is a must for a file to be accepted by the LINK as a valid object file.
- If no filenames are entered for these files, by default, the source filename is considered with different extensions.
- The LINK command display is shown in the FIG

```
C>LINK

Microsoft @ Overlay Linker Version. 3.64
Copyright(c)Microsoft Corp. 1983-88. All Rights Reserved.



Object Module[.OBJ]:
Run file[.EXE]:
List filename[NUL.MAP]:
Libraries[LIB]:
```

Fig. 3.12 *Link Command Screen Display*

- The option input libraries in the display expects any special library name of which the functions were used by the source program.
- **The output of the LINK program is an executable file with entered filename and .EXE extension**
- The executable filename can further be entered at the DOS prompt to execute the file

4. Using DEBUG

- DEBUG.COM is a DOS utility that facilitates debugging and trouble shooting of ALP.
- All the processor resource and memory resource management functions are carried out by the operating systems.
- The DEBUG utility enables you to have the control of these resources up to some extent.
- The Debug command at DOS prompt invokes the facility.
- **A _ (dash) signals the successful invoke operation of DEBUG , that is further used as DEBUG prompt for debugging commands.**
- The DEBUG command character display explain the DEBUG command entry procedure

```
C>DEBUG ←┘

-    ←┘

-R ←┘


         ←┘ └Symbol of <ENTER>key
```

Fig. 3.13   DEBUG Command Line and Prompt

- The list of generally used valid commands of DEBUG is given in table along with their respective syntax

Table 3.1   DEBUG Commands

COMMAND CHARACTER	Format/Formats	Functions
-R	<ENTER>	Display all Registers and flags
-R	reg<ENTER> old contents:New contents	Display specified register contents and modify with the entered new contents.
-D	<ENTER>	Display 128 memory locations of RAM starting from the current display pointer.
-D	SEG:OFFSET1 OFFSET2<ENTER>	Display memory contents in SEG from OFFSET1 to OFFSET2.
-E	<ENTER>	Enter Hex data at current display pointer SEG:OFFSET.
-E	SEG:OFFSET1 <ENTER>	Enter data at SEG:OFFSET1 byte by byte. The memory pointer is to be incremented by space key, data entry is to be completed by pressing the <ENTER> key.
-f	SEG:OFFSET1 OFFSET2 BYTE <ENTER>	Fill the memory area starting from SEG:OFFSET1 to OFFSET2 by the byte BYTE.
-f	SEG:OFFSET1 OFFSET2BYTE1,BYTE2,BYTE3<ENTER>	Fill the memory area as above with the sequence BYTE1, BYTE2, BYTE3, etc.
-a	<ENTER>	Assemble from the current CS:IP.
-a	SEG:OFFSET <ENTER>	Assemble the entered instruction from SEG:OFFSET address.
-u	<ENTER>	Unassemble from the current CS:IP.
-u	SEG:OFFSET <ENTER>	Unassemble from the address SEG:OFFSET.
-g	<ENTER>	Execute from current CS:IP. By modifying CS and IP using R command this can be used for any address.
-g	=OFFSET <ENTER>	Execute from OFFSET in the current CS.

INT 21H

- There are some DOS functions available under INT21H.
- **All the hardware resources (Memory, keyboard, CRT display, hard disk and floppy disk drive) of DOS** are handled by the instruction INT21H.
- The routines required to refer these resources are written as Interrupt Service Routines for INT21H.
- Under this Interrupt, specific resource is selected depending on the value of AH.
- For example, if AH contains 09H, then CRT display is to be used for displaying a message.
- If AH contains 0AH, then keyboard is accessed.
- The Interrupts are called function calls and the **value in AH** is called **Function value**

PROGRAMMING EXAMPLES:

- In this section, we sill study some programs which elucidate the use of instructions, directives and some other facilities.

1. ALP for addition of two 8-bit numbers

```
                ASSUME CS: CODE, DS: DATA
        DATA SEGMENT
                VAR1 DB 85H
                VAR2 DB 32H
                RES DB ?
        DATA ENDS
        CODE SEGMENT
                START: MOV AX, DATA
                        MOV DS, AX
                        MOV AL, VAR1
                        MOV BL, VAR2
                        ADD AL, BL
                        MOV RES, AL
                        MOV AH, 4CH
                         INT 21H
        CODE ENDS
        END START
         END
```

2. ALP for Subtraction of two 8-bit numbers

```
                ASSUME CS: CODE, DS: DATA
        DATA SEGMENT
                VAR1 DB 53H
                VAR2 DB 2AH
                RES DB ?
         DATA ENDS
        CODE SEGMENT
                START: MOV AX, DATA
                        MOV DS, AX
                        MOV AL, VAR1
                        MOV BL, VAR2
                        SUB AL, BL
                         MOV RES, AL
                        MOV AH, 4CH
                        INT 21H
        CODE ENDS
        END START
             END
```

3. ALP for Multiplication of two 8-bit numbers

```
        ASSUME CS: CODE, DS:DATA
DATA SEGMENT
        VAR1  DB 0EDH
        VAR2  DB 99H
        RES    DW  ?
DATA ENDS
CODE SEGMENT
        START: MOV AX, DATA
                MOV DS, AX
                MOV AL, VAR1
                MOV BL, VAR2
                MUL BL
                MOV RES, AX
                MOV AH, 4CH
                INT 21H
CODE ENDS
END START
        END
```

4. ALP for division of 16-bit number with 8-bit number

```
        ASSUME CS: CODE,DS:DATA
DATA SEGMENT
                VAR1  DW 6827H
                VAR2  DB 0FEH
                QUO   DB ?
                REM   DB  ?
        DATA ENDS
CODE SEGMENT
        START: MOV AX, DATA
                MOV DS, AX
                MOV AX, VAR1
                DIV VAR2
                MOV QUO, AL
                MOV REM, AH
                MOV AH, 4CH
                INT 21H
CODE ENDS
END START
END
```

5 ALP for addition of two 16-bit numbers

```
        ASSUME CS: CODE, DS: DATA
DATA SEGMENT
                VAR1 DW 8560H
                VAR2 DW 3297H
                RES  DW  ?
        DATA ENDS
```

```
CODE SEGMENT
        START: MOV AX, DATA
                MOV DS, AX
                MOV AX, VAR1
                CLC
                MOV BX, 0000H
                ADD AX, VAR2
                JNC K
                INC BX
        K:      MOV RES, AX
                MOV RES+2, BX
                MOV AH, 4CH
                INT 21H
CODE ENDS
END START
END
```

6. ALP for Subtraction of two 16-bit numbers

```
                ASSUME CS:CODE,DS:DATA
        DATA SEGMENT
                VAR1 DW 8560H
                VAR2 DW 3297H
                RES  DW  ?
DATA ENDS
CODE SEGMENT
                START:MOV AX,DATA
                        MOV DS,AX
                        MOV AX,VAR1
                        CLC
                        SUB AX,VAR2
                        MOV RES,AX
                        MOV AH,4CH
                        INT 21H
CODE ENDS
END START
END
```

7. ALP for Multiplication of two 16-bit numbers

```
                ASSUME CS: CODE, DS: DATA, ES: EXTRA
                DATA SEGMENT
                        OPR1 DW 5169H
                        OPR2 DW 1000H
                DATA ENDS
EXTRA SEGMENT
                        RES DW 2 DUP(0)
                        EXTRA ENDS
CODE SEGMENTSTART:
```

```
                        MOV AX, DATA
                        MOV DS, AX           ; REGISTER ADDRESIING MODE
                        MOV AX, EXTRA
                        MOV ES, AX           ; REGISTER ADDRESIING MODE
                        MOV SI, OFFSET OPR1
                        MOV AX, [SI]         ; INDEXED ADDRESSING MODE
                        MOV BX,OPR2          ; DIRECT ADDRESSING MODE
                        MUL BX               ; REGISTER ADDRESSING MODE
                        MOV RES, AX          ; DIRECT ADDRESSING MODE
                        MOV RES+2, DX        ; DIRECT ADDRESSING MODE
                        INT 03H
                CODE ENDS
                END START
                END
```

8. ALP to Sort a set of unsigned integer numbers in ascending/descending order.

ASCENDING ORDER

```
        ASSUME CS: CODE, DS: DATA
    DATA SEGMENT
            LIST DW 0125H,0144H,3001H,0003H,0002H
            COUNT EQU 05H
    DATA ENDS
    CODE SEGMENT
        START:MOV AX,DATA
            MOV DS,AX
            MOV DX,COUNT-1
    BACK: MOV CX,DX
        MOV SI, OFFSET LIST
    AGAIN: MOV AX,[SI]
        CMP AX,[SI+2]
        JC GO
        XCHG AX,[SI+2]
        XCHG AX,[SI]
    GO:INC SI
        INC SI
        LOOP AGAIN
        DEC DX
        JNZ BACK
        INT 03H
CODE ENDS
END START
END
```

DESCENDING ORDER

PROGRAM:

```
            ASSUME CS: CODE, DS:DATA
            DATA SEGMENT
                LIST  DW  0125H,0144H,3001H,0003H,0002H
```

```
                    COUNT  EQU  05H
          DATA ENDS
CODE SEGMENT
                    START: MOV AX, DATA
                    MOV DS, AX
                    MOV DX, COUNT-1
          BACK: MOV CX, DX
                    MOV SI, OFFSET LIST
          AGAIN: MOV AX, [SI]
                    CMP AX, [SI+2]
                    JAE GO
                    XCHG AX,[SI+2]
                    XCHG AX,[SI]
          GO:    INC SI
                    INC SI
                    LOOP AGAIN
                    DEC DX
                    JNZ BACK
                    INT 03H
     CODE ENDS
     END START
     END
```

MORE PROGRAMMING EXAMPLES:

1. Write an ALP to find factorial of number for 8086.

```
       MOV AX, 05H

       MOV CX, AX

Back:  DEC CX

       MUL CX

       LOOP back

       ; results stored in AX

       ; to store the result at D000H

       MOV [D000], AX

       HLT
```

Write a program for addition of two numbers.

Solution The following program adds two 16-bit operands. There are various methods of specifying operands depending upon the addressing modes that the programmer wants to use. Accordingly, there may be different program listings to achieve a single programming goal. A skilled programmer uses a simple logic and implements it by using a minimum number of instructions. Let us now try to explain the following program:

```
ASSUME CS:CODE, DS:DATA

DATA SEGMENT
OPR1 DW 1234H                    ; 1st operand
OPR2    DW 0002H                 ; 2nd operand
RESULT  DW 01 DUP(?)             ; A word of memory reserved for re-
                                   sult

DATA    ENDS
CODE    SEGMENT
START:  MOV AX, DATA             ; Initialize data segment
        MOV DS, AX               ;
        MOV AX, OPR1             ; Take 1st operand in AX
        MOV BX, OPR2             ; Take 2nd operand in BX
        CLC                      ; Clear previous carry if any
        ADD AX, BX               ; Add BX to AX
        MOV DI, OFFSET RESULT    ; Take offset of RESULT in DI
        MOV [DI], AX             ; Store the result at memory address in DI
        MOV AH, 4CH              ; Return to DOS prompt
        INT 21H
CODE    ENDS                     ; CODE segment ends
        END START                ; Program ends
```

Write a program for the addition of a series of 8-bit numbers. The series contains 100(numbers).

Solution   In the first program, we have implemented the addition of two numbers. In this program, we show the addition of 100 (D) numbers. Initially, the resulting sum of the first two numbers will be stored. To this sum, the third number will be added. This procedure will be repeated till all the numbers in the series are added. A conditional jump instruction will be used to implement the counter checking logic. The comments explain the purpose of each instruction.

```
ASSUME CS:CODE, DS:DATA
DATA SEGMENT                            ; Data segment starts
NUMLIST DB 52H, 23H,-                   ; List of byte numbers
COUNT EQU 100D                          ; Number of bytes to be added
RESULT DW 01H DUP(?)                    ; One word is reserved for result
DATA ENDS                               ; Data segment ends
CODE SEGMENT                            ; Code segment starts at relative
ORG 200H                                ; address 0200h in code segment
START:       MOV AX, DATA               ; Initialize data segment
             MOV DS, AX
             MOV CX, COUNT              ; Number of bytes to be added in CX
             XOR AX, AX                 ; Clear AX and CF
             XOR BX, BX                 ; Clear BH for converting the byte to
                                          word
             MOV SI,OFFSET NUMLIST      ; Point  to  the first  number  in the
                                          list
AGAIN:       MOV BL, [SI]               ; Take the first number in BL,BH is zero
             ADD AX, BX                 ; Add AX with BX
             INC SI                     ; Increment pointer to the byte list
             DEC CX                     ; Decrement counter
             JNZ AGAIN                  ; If all numbers are added,point to re-
                                          sult
             MOV DI, OFFSET RESULT      ; destination and store it
             MOV [DI], AX
             MOV AH, 4CH                ; Return to DOS
             INT 21H
             CODE ENDS
END          START
```

A program to find out the largest number from a given unordered array of 8-bit numbers, stored in the locations starting from a known address.

Solution Compare the *i*th number of the series with the (*i*+1)th number using CMP instruction. It will set the flags appropriately, depending upon whether the ith number or the (*i*+1)th number is greater. If the *i*th number is greater than (*i*+1)th, leave it in AX (any register may be used). Otherwise, load the (*i*+1)th number in AX, replacing the *i*th number in AX. The procedure is repeated till all the members in the array have been compared.

```
ASSUME CS:CODE, DS:DATA
DATA SEGMENT                          ; Data segment starts
LIST DB 52H, 23H, 56H, 45H, --        ; List of byte numbers
COUNT EQU OF                          ; Number of bytes in the list
LARGEST DB 01H DUP(?)                 ; One byte is reserved for the largest
                                        number.
DATA ENDS                             ; Data segment ends
CODE SEGMENT                          ; Code segment starts.
START:      MOV AX, DATA              ; Initialize data segment.
            MOV DS, AX
            MOV SI, OFFSET LIST
            MOV CL, COUNT             ; Number of bytes in CL.
            MOV AL, [SI]              ; Take the first number in AL
AGAIN:      CMP AL, [SI+1]            ; and compare it with the next number.
            JNL NEXT
            MOV AL, [SI+1]
NEXT:       INC SI                    ; Increment pointer to the byte list.
            DEC CL                    ; Decrement counter.
            JNZ AGAIN                 ; If all numbers are compared, point to
                                        result
            MOV SI, OFFSET LARGEST ; destination and store it.
            MOV [SI], AL
            MOV AH, 4CH               ; Return to DOS.
            INT 21H
            CODE ENDS
END         START
```

Stack Structure of 8086:

Stack

- The stack is a block of memory that may be used for temporarily storing the contents of the registers inside the CPU.

- It is a top-down data structure whose elements are accessed using the stack pointer (SP)

- SP is **decremented by two** as we **store a data** word into the stack

- SP gets incremented by two as we retrieve a data word from the stack back to the CPU register.

- The process of storing the data in the stack is called 'pushing into' the stack.

- The reverse process of transferring the data back from the stack to the CPU

register is known as 'popping off' the stack.

- The stack is essentially *Last-In-First-Out* (**LIFO**) data segment.

- This means that the data which is **pushed into the stack last** will be **on top of stack** and will be popped off the stack first.

Stack pointer:

- The stack pointer is a 16-bit register that contains the **offset address of the memory location** in the **stack segment.**

Stack segment

- The stack segment have a memory block of a maximum of 64 Kbytes locations,

Stack Segment register (SS)

- Stack Segment register (SS) contains the base address of the stack segment in the memory.

- The Stack Segment register (SS) and Stack pointer register (SP) together address the stack-top as explained below:

- Let the content of SS be 5000H and the content of stack pointer be 2050H.

- To find the current stack-top address, the stack segment register content is shifted left by four bit positions.

- The resulting 20 bit content is added with the 16 bit offset value, stored in the **stack point register.**

- In the above case, the stack-top address can be calculated as shown:

HOW WE CAN CALCULATE STACK TOP ADDRESS =

SS stack segment contain base address of stack segment in memory

SS stack segment and SP make a address together stack top
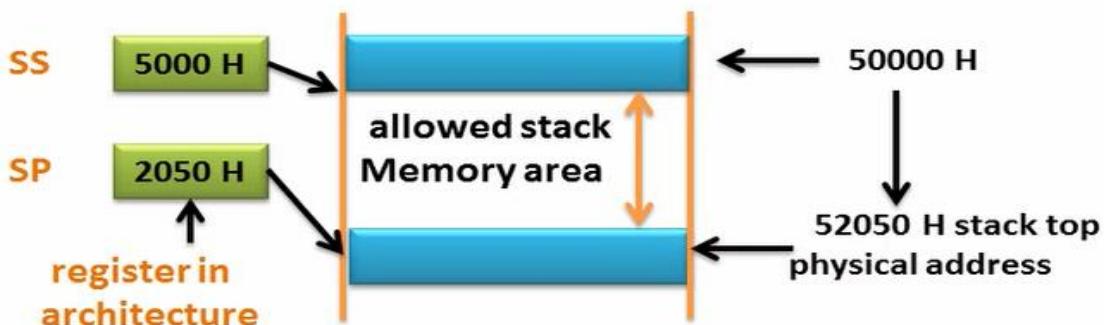
SS stack segment contain 5000H
SP stack pointer contain 2050H

SS = 5000H
SP = 2050H

SS =	0101 0000 0000 0000	
10H*SS =	0101 0000 0000 0000 0000	
SP =	➕ 0010 0000 0101 0000	
Stack top address =	0101 0010 0000 0101 0000	
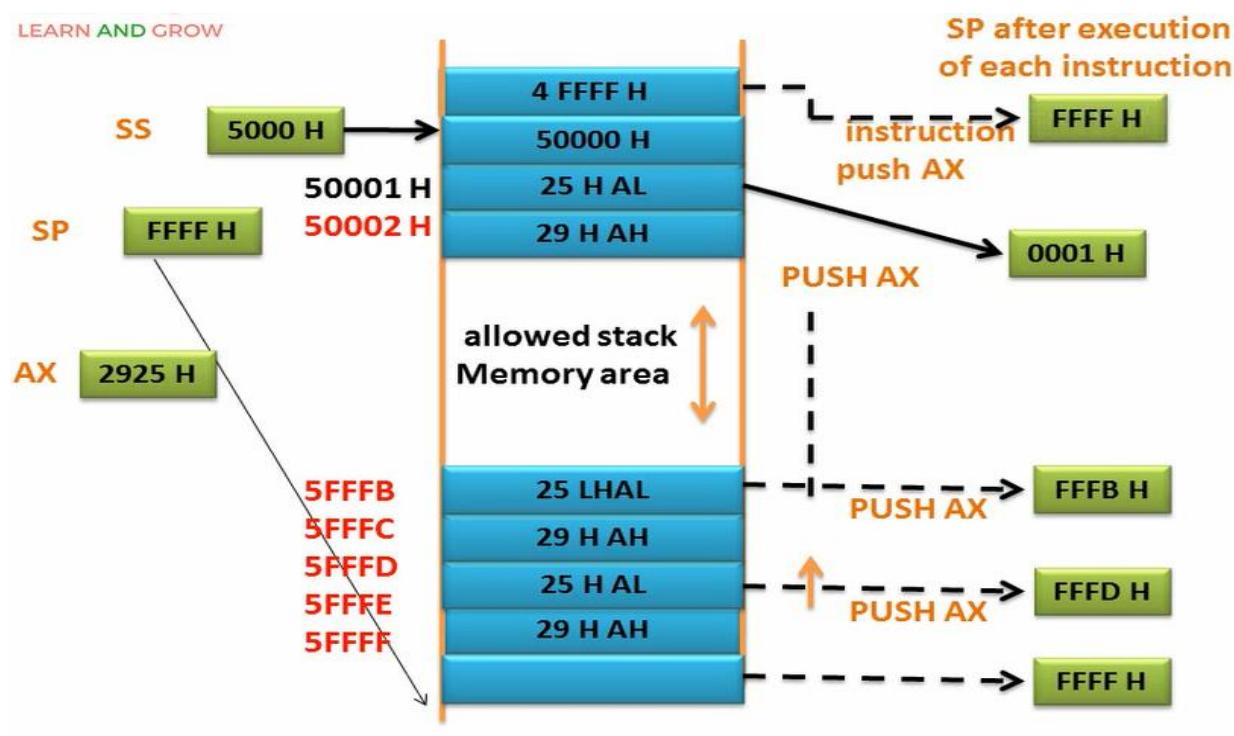
- If the stack top points to a memory location 52050H, it means that the location 52050H is already occupied with the previously pushed data.
- The next 16 bit push operation will decrement the stack pointer by two, so that it will point to the new stack-top **5204EH**
- The decremented contents of SP will be **204EH.** This location will now be occupied by the recently pushed data.
- Thus for a selected value of SS, the maximum value of **SP=FFFFH**
- The segment can have maximum of 64K locations.
- If the SP starts with an initial value of FFFFH, it will be decremented by two

whenever a 16-bit data is pushed onto the stack.

- After **successive push operation**s, when the stack pointer contains **0000H**, any attempt to further push the data to the stack will result in **stack overflow.**

- Each PUSH operation decrements the SP by 2. While Each POP operation increments the SP by 2.
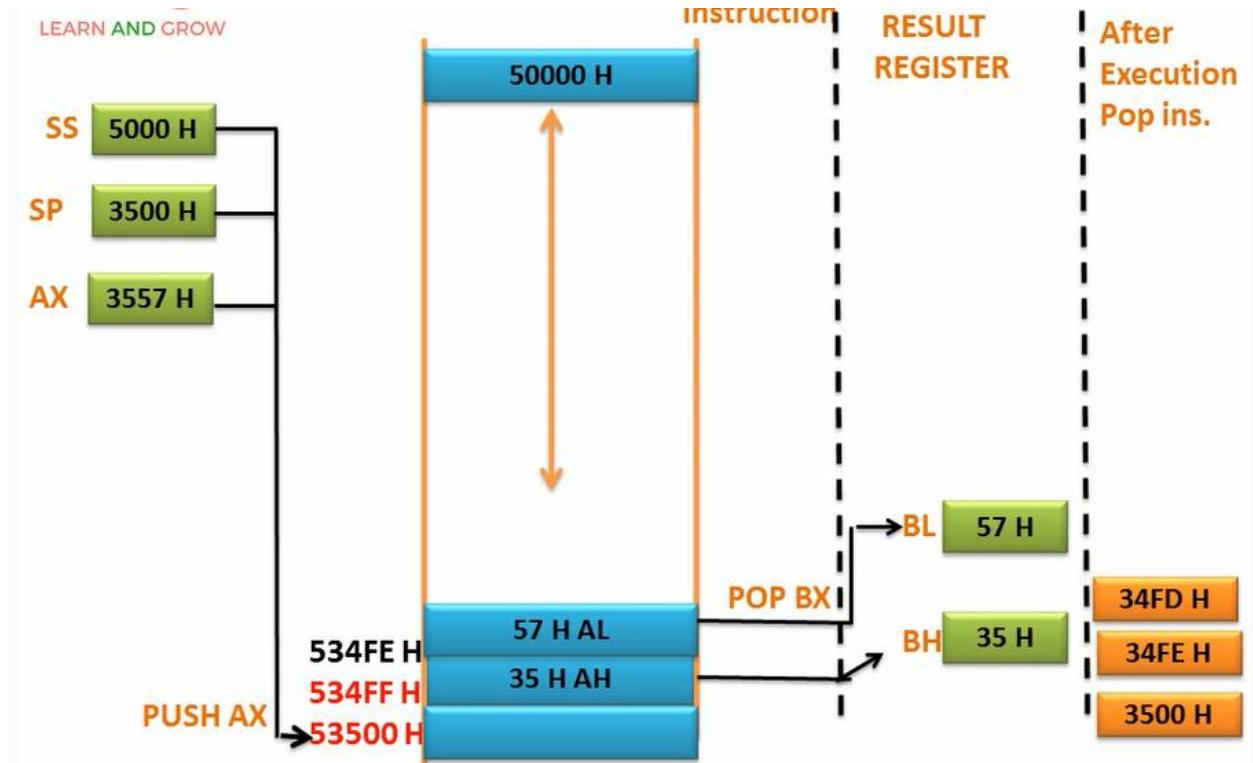
Stack Overflow Condition

- Suppose a main program is being executed by the processor.

- At some point during the execution of the program, all the registers in the CPU may contain the useful data.

- If there is a subroutine CALL Instruction at this stage, there is a possibility the all or some of the registers of the main program can be modified.

- This may result in loss of data.

- It can be avoided by using the stack.

- After a procedure is called using the CALL instruction, the IP is incremented to the next instruction.

- Then the contents of IP, CS and flag register are pushed automatically to the stack.

- The control is then transferred to the specified address in the CALL instruction i.e. starting address of the procedure. Then the procedure is executed.

- After each PUSH operation SP will be modified . Thus all the registers can be copied on to the stack.

- Now these registers may be used by the subroutine, since their original contents are saved onto the stack.

- At the end of execution of subroutine, all the registers can get back their original contents by popping the data from the stack.

Effect of PUSH and POP on SP
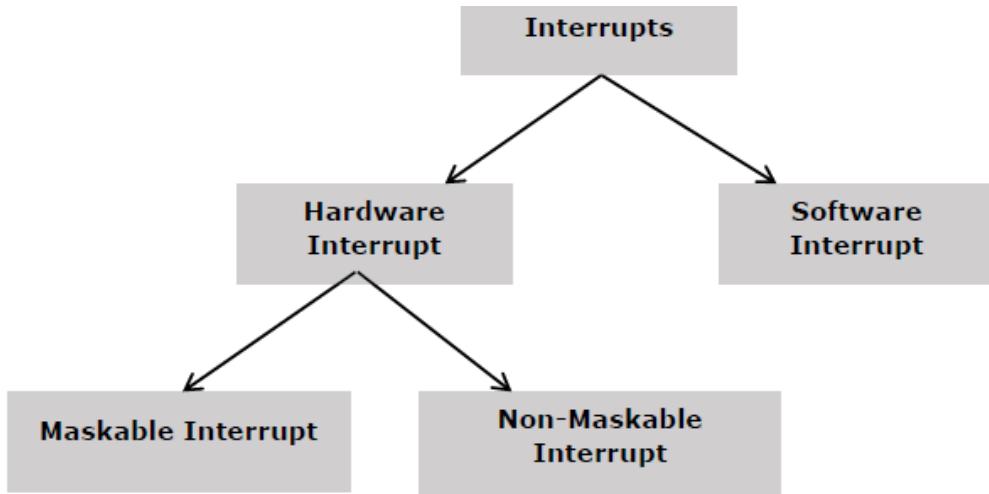
Interrupts and Interrupt Service Routines

Interrupt:

Interrupt means "Break the sequence of operation".

- An Interrupt is an indicating event that needs immediate attention.
- The interrupts can be either hardware interrupts or software interrupts.
- The following image shows the types of interrupts we have in a 8086 microprocessor

Types of Interrupt

Hardware Interrupt:

- It is an electronic alerting signal sent to the processor from an external device, like a disk controller or an external peripheral.

- The 8086 has two hardware interrupt pins, i.e. NMI and INTR.

- NMI is a non-maskable interrupt and INTR is a maskable interrupt having lower priority.

- One more interrupt pin associated is INTA called interrupt acknowledge.

NMI

It is a single non-maskable interrupt pin (NMI) having higher priority than the maskable interrupt request pin (INTR) and it is of type 2 interrupt.

When this interrupt is activated, these actions take place −

- Completes the current instruction that is in progress.

- Pushes the Flag register values on to the stack.

- Pushes the CS (code segment) value and IP (instruction pointer) value of the return address on to the stack.

- IP is loaded from the contents of the word location 00008H.

- CS is loaded from the contents of the next word location 0000AH.

- Interrupt flag and trap flag are reset to 0.

INTR

The INTR is a maskable interrupt because the microprocessor will be interrupted only if interrupts are enabled using set interrupt flag instruction.

It should not be enabled using clear interrupt Flag instruction.

The INTR interrupt is activated by an I/O port. If the interrupt is enabled and NMI is disabled, then the microprocessor first completes the current execution and sends '0' on INTA pin twice.

The first '0' means INTA informs the external device to get ready and during the second '0' the microprocessor receives the 8 bit, say X, from the programmable interrupt controller.

These actions are taken by the microprocessor −

- First completes the current instruction.

- Activates INTA output and receives the interrupt type, say X.

- Flag register value, CS value of the return address and IP value of the return address are pushed on to the stack.

- IP value is loaded from the contents of word location $X \times 4$

- CS is loaded from the contents of the next word location.

- Interrupt flag and trap flag is reset to 0

Software Interrupts

Some instructions are inserted at the desired position into the program to create interrupts. These interrupt instructions can be used to test the working of various interrupt handlers. It includes −

INT- Interrupt instruction with type number

It is 2-byte instruction. First byte provides the op-code and the second byte provides the interrupt type number. There are 256 interrupt types under this group.

Its execution includes the following steps −

- Flag register value is pushed on to the stack.

- CS value of the return address and IP value of the return address are pushed on to the stack.

- IP is loaded from the contents of the word location 'type number' × 4

- CS is loaded from the contents of the next word location.

- Interrupt Flag and Trap Flag are reset to 0

The starting address for type0 interrupt is 000000H, for type1 interrupt is 00004H similarly for type2 is 00008H and ……so on. The first five pointers are dedicated interrupt pointers. i.e. −

- **TYPE 0** interrupt represents division by zero situation.

- **TYPE 1** interrupt represents single-step execution during the debugging of a program.

- **TYPE 2** interrupt represents non-maskable NMI interrupt.

- **TYPE 3** interrupt represents break-point interrupt.

- **TYPE 4** interrupt represents overflow interrupt.

The interrupts from Type 5 to Type 31 are reserved for other advanced microprocessors, and interrupts from 32 to Type 255 are available for hardware and software interrupts.

INT 3-Break Point Interrupt Instruction

It is a 1-byte instruction having op-code is CCH. These instructions are inserted into the program so that when the processor reaches there, then it stops the normal execution of program and follows the break-point procedure.

Its execution includes the following steps −

- Flag register value is pushed on to the stack.

- CS value of the return address and IP value of the return address are pushed on to the stack.

- IP is loaded from the contents of the word location $3 \times 4 = 0000CH$

- CS is loaded from the contents of the next word location.

- Interrupt Flag and Trap Flag are reset to 0

INTO - Interrupt on overflow instruction

It is a 1-byte instruction and their mnemonic **INTO**. The op-code for this instruction is CEH. As the name suggests it is a conditional interrupt instruction, i.e. it is active only when the overflow flag is set to 1 and branches to the interrupt handler whose interrupt type number is 4. If the overflow flag is reset then, the execution continues to the next instruction.
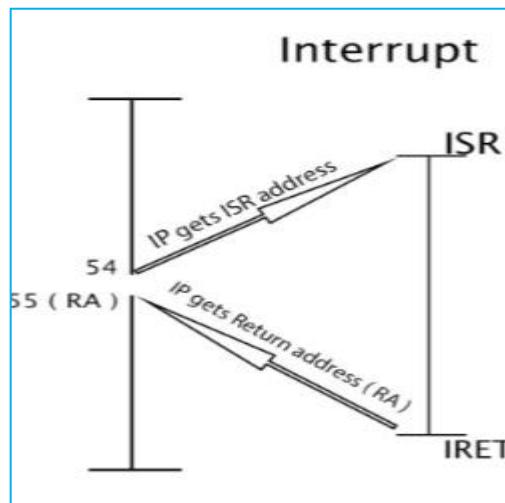
Its execution includes the following steps −

- Flag register values are pushed on to the stack.

- CS value of the return address and IP value of the return address are pushed on to the stack.

- IP is loaded from the contents of word location $4 \times 4 = 00010H$

- CS is loaded from the contents of the next word location.

- Interrupt flag and Trap flag are reset to 0

Software Interrupt:

- It is caused either by an exceptional condition or a special instruction in the instruction set

- When the CPU is executing a program, an Interrupt breaks the normal sequence of execution of instructions, Diverts it execution to some other program called Interrupt service routine(ISR)

- ISR is a program that tells the processor what to do when the interrupt occurs.

- Whenever an interrupt occurs the processor completes the execution of the current instruction and starts the execution of an Interrupt Service Routine (ISR) or Interrupt Handler.

- After executing ISR, the control is transferred back again to the main program which was being executed at the time of interruption.

- Whenever a number of devices interrupt a CPU at a time, and if the processor is able to handle them properly, it is said to have *multiple interrupt processing capability*.

Interrupt Service Routines

- For every interrupt, there must be an interrupt service routine (ISR), or interrupt handler.
- When an interrupt occurs, the microprocessor runs the interrupt service routine.
- For every interrupt, there is a fixed location in memory that holds the address of its interrupt service routine, ISR.
- *The table of memory locations set aside to hold the addresses of ISRs is called as the Interrupt Vector Table.*
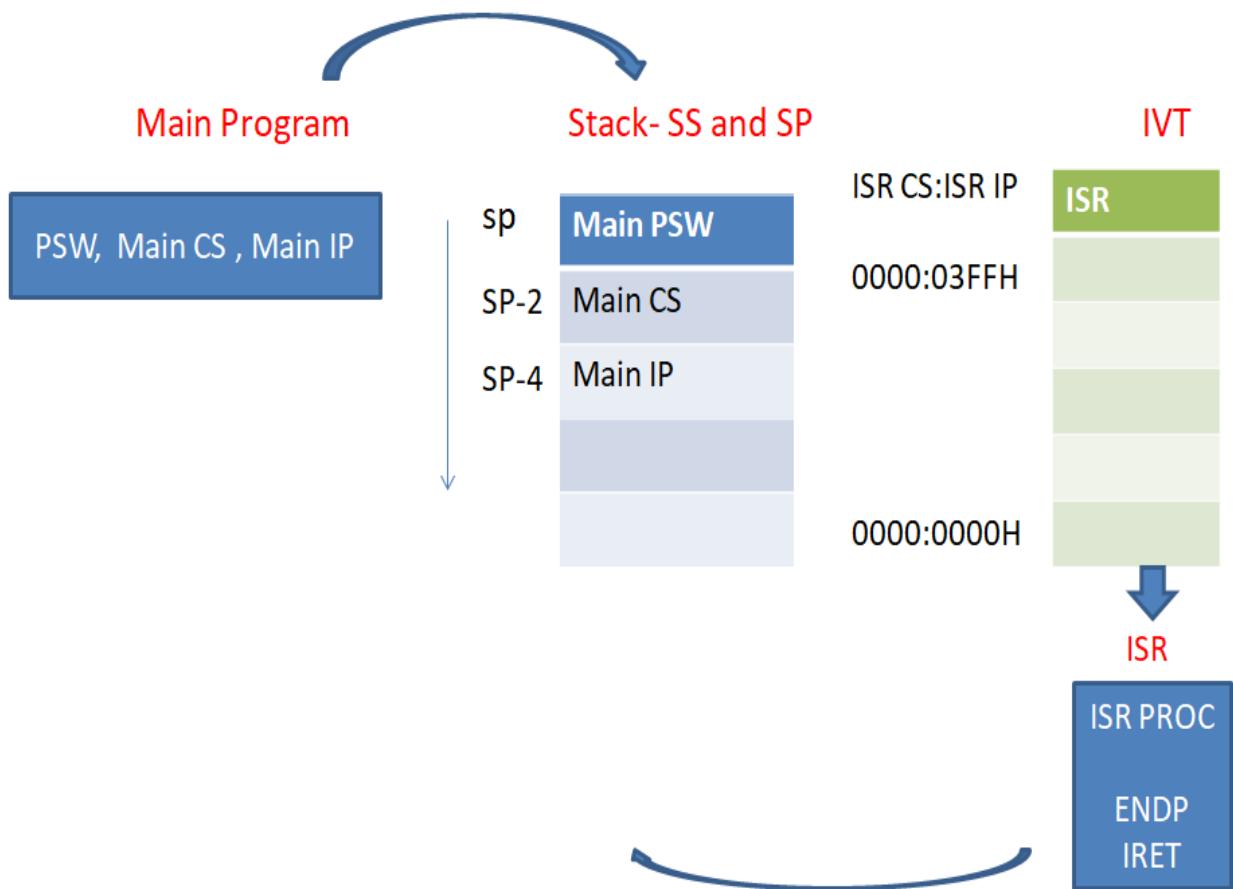
Interrupt Cycle of 8086:

- Suppose an external device interrupts the CPU at the interrupt pin , either NMI or INTR of the 8086,
- While the CPU is executing an instruction in the program, the CPU first completes the execution of the current Instruction.
- The IP is then incremented to point to the next instruction.
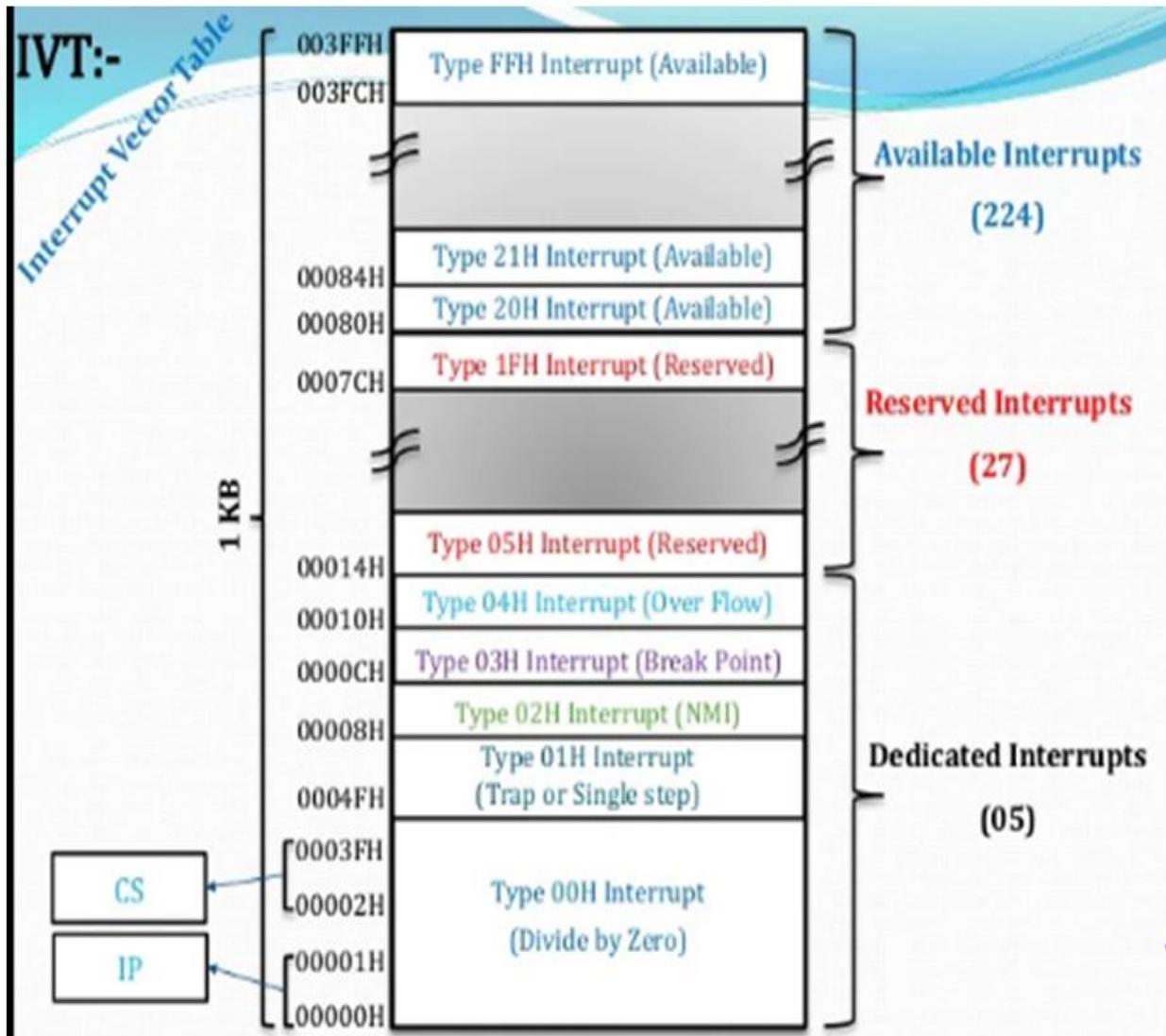
- The CPU acknowledges the requesting device on its INTA pin immediately if it is a NMI, TRAP, or Divide by Zero.

- If it is a INT request, the CPU checks the IF flag.

  -If IF is set, then it is acknowledged by using INTA pin.

  -If IF is not set(Reset), the interrupt requests are ignored.

- Note that the responses to NMI, TRAP, and divide by zero Interrupt request are independent of the IF flag.

- After an Interrupt is occurred , the CPU computes the vector address from the type of Interrupt.

- When a microprocessor receives an interrupt signal it stops executing current normnal program, save the status (or content) of various registers (IP, CS and flag registers in case of 8086) in stack and

- Then the processor executes a subroutine/procedure in order to perform the specific task/work requested by the interrupt.

- The subroutine/procedure that is executed in response to an interrupt is also called Interrupt Service Subroutine (ISR).

- At the end of ISR, the last instruction should be IRET

- At the end of ISR, the stored status of registers in stack is restored to respective registers, and the processor resumes the normal program execution

from the point {instruction) where it was interrupted

Interrupt Response Sequence

Interrupt Vector Table:
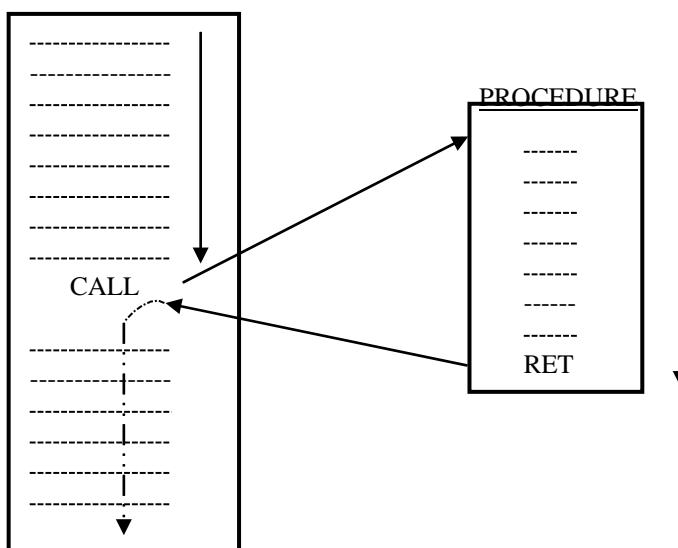
- Intel has reserved 1024 locations for Interrupt vector table.

- Each Interrupt requires 4 bytes, i.e 2 bytes for IP and CS of its ISR.

- Thus a total of 1024 bytes are required for 256 Interrupt types, hence the Interrupt vector table starts at 0000: 0000 and ends at 0000:03FFH

- The IVT contains the IP and CS of all the interrupt types stored sequentially.

PROCEDURES

Whenever we have a series of instructions that we want to execute several times in a program, we write the series of instructions as a separate subprogram. We can then call this subprogram each time we want to execute that series of instructions. This saves us from having to write the series of instructions over and over each time we want it to execute in the program. This subprogram is usually called a subroutine or a procedure.

The CALL instruction in the main program loads the instruction pointer with the starting address of the procedure. The next instruction fetched then will be the first instruction of the procedure. At the end of the procedure a return instruction, RET, sends execution back to the next instruction after the CALL in the main line program.

MAIN LINE PROGRAM

The CALL instruction performs two operations when it executes. First it stores the address of the instruction after the CALL instruction on the stack. The second operation is load the instruction pointer with the starting address of the procedure. The RET copies a word from the top of the stack to the instruction pointer register.

A near call is a call to a procedure which is in the same code segment as the CALL instruction. A far call is a call to a procedure which is in a different segment from which contains the call instruction.

A stack is a section of memory set a side for storing return addresses. The stack is also used to save the contents of the registers for the calling program while a procedure executes. An other use of the stack is to hold data or addresses that will be acted upon by a procedure.

An important point about the operation of the stack is that the SP register is automatically decremented by 2 before a word is written to the stack. This means that at the start of your program you must initialise the SP register to point to the top of the memory you set aside as a stack, rather than initialising it to point to the bottom location.

If the 8086 executes a near CALL instruction, the SP register will automatically be decremented by 2 and contents of IP register will be written to the stack.

When a near RET instruction executes, the IP value stored in the stack will be copied back to the IP register and the SP register will be automatically incremented by 2.

PROC:

The PROC directive is used to identify the start of a procedure. The PROC directive follows a name you give the procedure. After the PROC directive the term NEAR or FAR is used to specify the type of the procedure.

```
Factorial PROC near
        -----
        -----


        RET
Factorial  ENDP
```

ENDP :-   [end procedure]

This directive is used along with the name of the procedure

to indicate the end of aprocedure to the assembler.

```
SQUARE_ROOT  PROC  NEAR
            ----
            ----
            ----



SQUARE_ROOT
ENDP
```

Passing Parameters to Procedures

Procedures

- Procedure is a part of code that can be called from your program in order to make some specific task.

- Procedures make program more structural and easier to understand.

- Generally procedure returns to the same point from where it was called.

- The syntax for procedure declaration:

    name PROC

     ; here goes the code
       ; of the procedure ...

    RET
    ENDP

- name - is the procedure name, the same name should be in the top and the bottom, this is used to check correct closing of procedure.

- Procedures or subroutines may require input data or constants for their execution.

- Their data or constants may be passed to the subroutine by the main program or some subroutine may access readily available data of constants available in memory.

- The following are the techniques used to pass input data/parameter to procedures in ALP

1. **Using Global declared Variable**
2. **Using registers of CPU architecture.**
3. **Using Memory locations.**
4. **Using stack.**
5. **Using PUBLIC & EXTRN**

➢ If a procedure is interactive it may directly accepts inputs from input devices.

Using Global declared Variable

- A variable or parameter label may be declared global in the main program and the same variable can be used by all the routines or procedures of the application

  Example:

  ASSUME CS: CODE1,DS: DATA

  DATA SEGMENT

  NUMBER EQU 77H GLOBAL

  DATA ENDS

  CODE1 SEGEMENT

  START: MOV AX,DATA

  MOV DS, AX

  MOV AX, NUMBER

  CODE1 ENDS

  ASSUME CS: CODE2

  CODE2 SEGEMENT

  MOV AX,DATA

  MOV DS, AX

  MOV BX, NUMBER

  CODE2 ENDS

END START

Using registers of CPU Architecture.

- The CPU general purpose registers may be used to pass parameters to the procedures.

- The main program may store the parameters to be passed to the procedure

in the available CPU registers and the procedure may use the same register contents for execution.

- **Example:**
  ASSUME CS: CODE
  CODE SEGEMENT
  START: MOV AX,5555H
        MOV BX, 7272H

  CALL PROCEDURE1

  PROCEDURE PROCEDURE1 NEAR

        ADD AX, BX

  RET
  PROCEDURE1 ENDP
  CODE ENDS
  END START

Using Memory locations.

- Memory locations may also be used to pass parameters to procedures in the same way as registers.
- A main program may store the parameter to a procedure at known memory address location and the procedure may use the same location for accessing the parameter.

Example:
    ASSUME CS: CODE1,DS: DATA
    DATA SEGMENT
    NUMBER DB [55H]
    COUNT EQU 10H
    DATA ENDS
    CODE1 SEGEMENT
    START: MOV AX,DATA

```
                MOV DS, AX
                ---
                ---
CALL  ROUTINE
                ---
PROCEDURE ROUTINE NEAR
                MOV BX,NUMBER
                MOV CX, COUNT
                ADD  AX, BX
                ----
                ----
ROUTINE  ENDP
CODE ENDS
END START
```

Using Stack Memory

- Stack can be used to pass parameters to a procedure

- A main program may store the parameters to be passed to a procedure in its CPU registers.

- The registers will further be pushed on to the stack. The procedure during its execution pops back the appropriate parameters as and when required.

- **Example:**

```
ASSUME CS: CODE,SS: STACK
CODE SEGEMENT
START:  MOV AX,5555H
            MOV  BX, 7272H
            ---
            ---
PUSH AX
PUSH BX
CALL ROUTINE
            ---
PROCEDURE ROUTINE NEAR
            ----
```

MOV DX,SP

ADD SP, 02

POP AX

POP BX

MOV SP, DX

STACK SEGMENT

STACK DATA DB 200H DUP(?)

STACK ENDS

END START

Using PUBLIC & EXTRN

- For passing parameters to procedures using the PUBLIC & EXTRN directives, may be declared PUBLIC in the main routine and the same should be declared EXTRN in the procedure.

- Thus the main program can pass the PUBLIC parameters to a procedure in which it is declared EXTRN

- Example:

  ASSUME CS: CODE,DS: DATA

  DATA SEGMENT

  PUBLIC NUMBER EQU 200H

  DATA ENDS

  CODE SEGEMENT

  START: MOV AX,DATA

          MOV DS, AX

  CALL ROUTINE

  PROCEDURE ROUTINE NEAR

          EXTRN NUMBER

          MOV AX,NUMBER

```
                    ----
                    ----
      ROUTINE  ENDP
      CODE ENDS
      END START
```

MACROS

- It is a label assigned with the repeatedly appearing string of instructions.

- The process of assigning s label or macro name to the string is called macro.

- A macro within a macro is called nested macro

- The macro name or macro definition is then used throughout the main program to refer to that string of instructions.

Difference between Macro and Procedure

- In the macro, the complete code of string instruction is inserted at each place where the macro name appears.

- Hence for Macro, the EXE file becomes lengthy.

- Procedure/ subroutine is called whenever necessary. i.e. the control of execution is transferred to the subroutine every time it is called.

- So the EXE file becomes smaller as the subroutine appears only once in the code.

- Macro does not utilize the service of stack.

- Procedure/Subroutine utilize the service of stack.

- There is no question of transfer of control as the program using macro inserts the complete code of the macro at every reference of the macro name.

- The control is transferred to the subroutine/Procedure whenever it is called

- Macro requires large memory space compared to the procedure(less memory space) as it inserts the entire code in the program.

Macro requires less time for execution, as it does not contain CALL and RETURN instructions as the subroutines do.

Defining a Macro

- A Macro can be defined anywhere in the program using the directives MACRO and ENDM.

- The label prior to MACRO is the macro name which should be used in the actual program.

- The ENDM directive marks the end of the instructions or statements sequence assigned with the MACRO name.

- The following MACRO DISPLAY displays the message MSG on the CRT.

- The syntax is given as

      DISPLAY MACRO
      MOV AX,SEG MSG
      MOV DS, AX
      MOV DX,OFFSET MSG
      MOV AH,09H
      INT 21H
      ENDM

- The above definition of the macro assigns the name DISPLAY to the instruction sequence between the directives MACRO and ENDM.

- While assembling, the above sequence of instructions will replace label DISPLAY, whenever it appears in the program.

- A MACRO may also be used to represent statements and directives.

- The concept of macro remains the same independent of its contents.

The following example shows the MACRO containing the statements. The macro defines the strings to be displayed

STRINGS MACRO
    MSG1 DB 0AH,0DH, "Program terminated normally", 0AH,0DH, "$"
    MSG2 DB 0AH,0DH, "Retry, Abort, Fail", 0AH,0DH, "$"

- A macro may be called by its name, along with any values to be passed to the macro.
- Calling a macro means inserting the statements and instructions represented by the macro directly at the place of the macro name in the program.

Passing parameter to a MACRO

- Using parameters in a definition, the programmer specifies the parameters of the macro those are likely to be changed each time the macro is called.

- For example, DISPLAY macro written in program can be made to display to different messages MSG1 and MSG2, as shown

DISPLAY MACRO
                MOV AX,SEG MSG
                MOV DS, AX
                MOV DX,OFFSET MSG
                MOV AH,09H
                INT 21H
                ENDM

- Parameters MSG can be replaced by MSG1 and MSG2 while calling the macro as shown

    -----

    DISPLAY MSG1

    DISPLAY MSG2

MSG1 DB 0AH,0DH, "Program terminated normally", 0AH,0DH, "$"
MSG2 DB 0AH,0DH, "Retry, Abort, Fail", 0AH,0DH, "$"

- All the parameters are specified in the definition execute sequentially and also in the call with the same sequence.

- All the directives available in MASM can also be used in a macro and carry the same significanc

Unit-IV

Computer Arithmetic: Introduction, Addition and Subtraction, Multiplication Algorithms, Division Algorithms

Input-Output Organization: Peripheral Devices, Input-Output Interface, Asynchronous data transfer, Modes of Transfer, Priority Interrupt, Direct memory Access, Input –Output Processor (IOP).

COMPUTER ARITHMETIC

Introduction:

Data is manipulated by using the arithmetic instructions in digital computers.

Data is manipulated to produce results necessary to give solution for the computation problems.

The Addition, subtraction, multiplication and division are the four basic arithmetic operations.

Using these operations other arithmetic functions can be formulated and scientific problems can be solved by numerical analysis methods.

Arithmetic Processor:

- It is the part of a processor unit that executes arithmetic operations.

- The arithmetic instructions definitions specify the data type that should be present in the registers used .

- The arithmetic instruction may specify binary or decimal data and in each case the data may be in fixed-point or floating point form.

- Fixed point numbers may represent integers or fractions.

- The negative numbers may be in signed magnitude or signed-2's complement representation.

- The arithmetic processor is very simple if only a binary fixed point add instruction is included.

- It would be more complicated if it includes all four arithmetic operations for binary and decimal data in fixed and floating point representations.

Algorithm

- Algorithm can be defined as a finite number of well defined procedural steps to solve a problem.

- Usually, an algorithm will contain a number of procedural steps which are dependent on results of previous steps.
- A convenient method for presenting an algorithm is a flowchart which consists of rectangular and diamond –shaped boxes.
- The computational steps are specified in the rectangular boxes and the decision steps are indicated inside diamond-shaped boxes from which 2 or more alternate path emerge

Addition and Subtraction:

There are three ways of representing negative fixed point binary numbers:

 1.Signed-magnitude representation ---- used for the representation of mantissa for floating point operations by most computers.

2. Signed-1's complement

3. Signed -2's complement—Most computers use this form for performing arithmetic operation with integers.

Addition and subtraction algorithm for signed-magnitude data:

The representation of numbers in signed-magnitude is familiar because it is used in arithmetic calculations.

- Let the magnitude of two numbers be A & B.
- When signed numbers are added or subtracted, there are 4 different conditions to be considered for each addition and subtraction depending on the sign of the numbers.
- The conditions are listed in the table below. The table shows the operation to be performed with magnitude(addition or subtraction) are indicated for different conditions

Sl.No	Operation	Add Magnitudes	Subtract magnitudes		
			When A> B	When A< B	When A=B
1	(+A) + (+B)	+ (A + B)			
2	(+A) + (-B)		+(A-B)	-(B-A)	+(A-B)
3	(-A) + (+B)		-(A-B)	+(B-A)	+(A-B)
4	(-A) + (-B)	- (A + B)			
5	(+A) - (+B)		+(A-B)	-(B-A)	+(A-B)
6	(+A) - (-B)	+ (A + B)			
7	(-A) - (+B)	- (A + B)			
8	(-A) - (-B)		-(A-B)	+(B-A)	+(A-B)

- The last column is needed to prevent a negative zero. In other words, when two equal numbers are subtracted, the result should be +0 not -0.
- The algorithm for addition and subtraction (from the table above):

Addition Algorithm:
- When the signs of A and B are identical, add two magnitudes and attach the sign of A to the result.
- When the sign of A and B are different, compare the magnitudes and subtract the smaller number from the larger.
- Choose the sign of the result to be the same as A if A>B or the complement of sign of A if A < B.
- If the two magnitudes are equal, subtract B from A and make the sign of the result positive.

Subtraction Algorithm:
- When the signs of A and B are different, add two magnitudes and attach the sign of A to the result.

- When the sign of A and B are identical, compare the magnitudes and subtract the smaller number from the larger.
- Choose the sign of the result to be the same as A if A>B or the complement of sign of A if A < B.
- If the two magnitudes are equal, subtract B from A and make the sign of the result positive.

Hardware Implementation:

- Let A and B are two registers that hold the numbers. AS and BS are 2, flip-flops that hold sign of corresponding numbers.
- The result is stored in A and AS and thus they form Accumulator register.
- We need to perform micro operation, A+ B and hence a parallel adder is required.
- A comparator is needed to establish if A> B, A=B, or A=B, or A<B
- We need to perform micro operations A-B and B-A and hence two parallel subtractor are required.
- An exclusive OR gate can be used to determine the sign relationship, that is, equal or not.
- Thus the hardware components required are a magnitude comparator, an adder, and two subtractors

Reduction of hardware by using different procedure:

- We know subtraction can be done by complement and add.
- The result of comparison can be determined from the end carry after the subtraction.
- We find an adder and a complementer can do subtraction and comparison if 2's complement is used for subtraction.
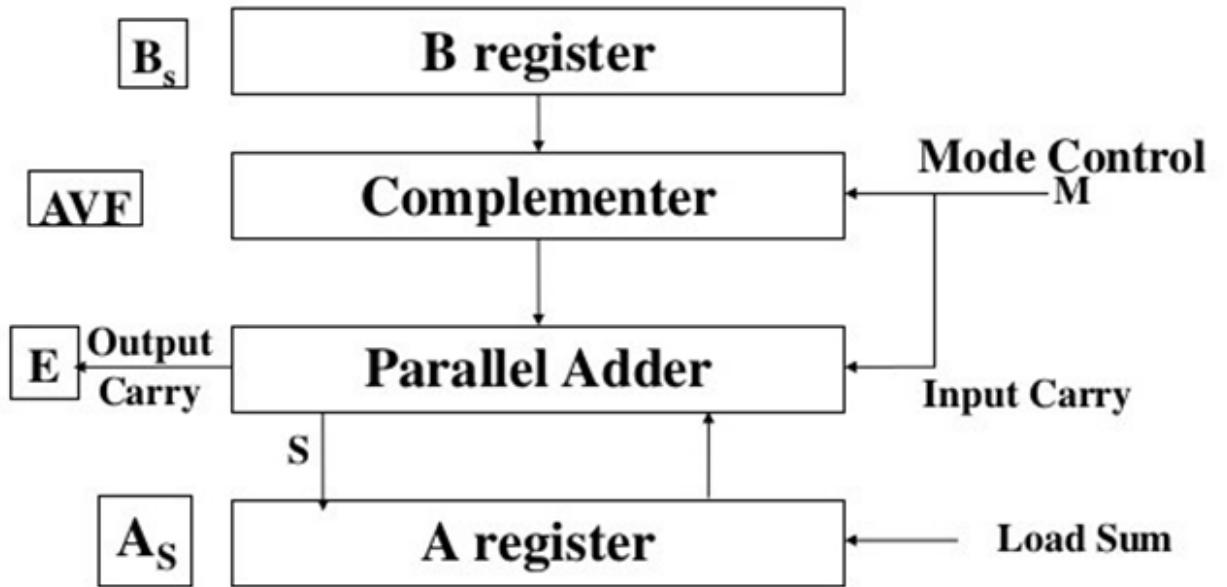
Figure 1: Hardware for signed-magnitude addition and subtraction

- **AVF** Add overflow flip flop. It hold the overflow bit when A & B are added.
- **Flip flop E** —Output carry is transferred to E. It can be checked to see the relative magnitudes of the two numbers.
- A-B = A +(-B)= Adding A and 2's complement of B.
- The A register provides other micro operations that may be needed when the sequence of steps in the algorithm is specified.
- The complementer passes the contents of B or the complement of B to the Parallel Adder depending on the state of the mode control M.
- It consists of EX-OR gates and the parallel adder consists of full adder circuits.
- The M(Mode Control) signal is also applied to the input carry of the adder. When input carry **M=0, the sum of full adder is A +B.**
- When M=1, $S = A + B' + 1 = A - B$

Hardware algorithm: Flow Chart for Add and Subtract operations:
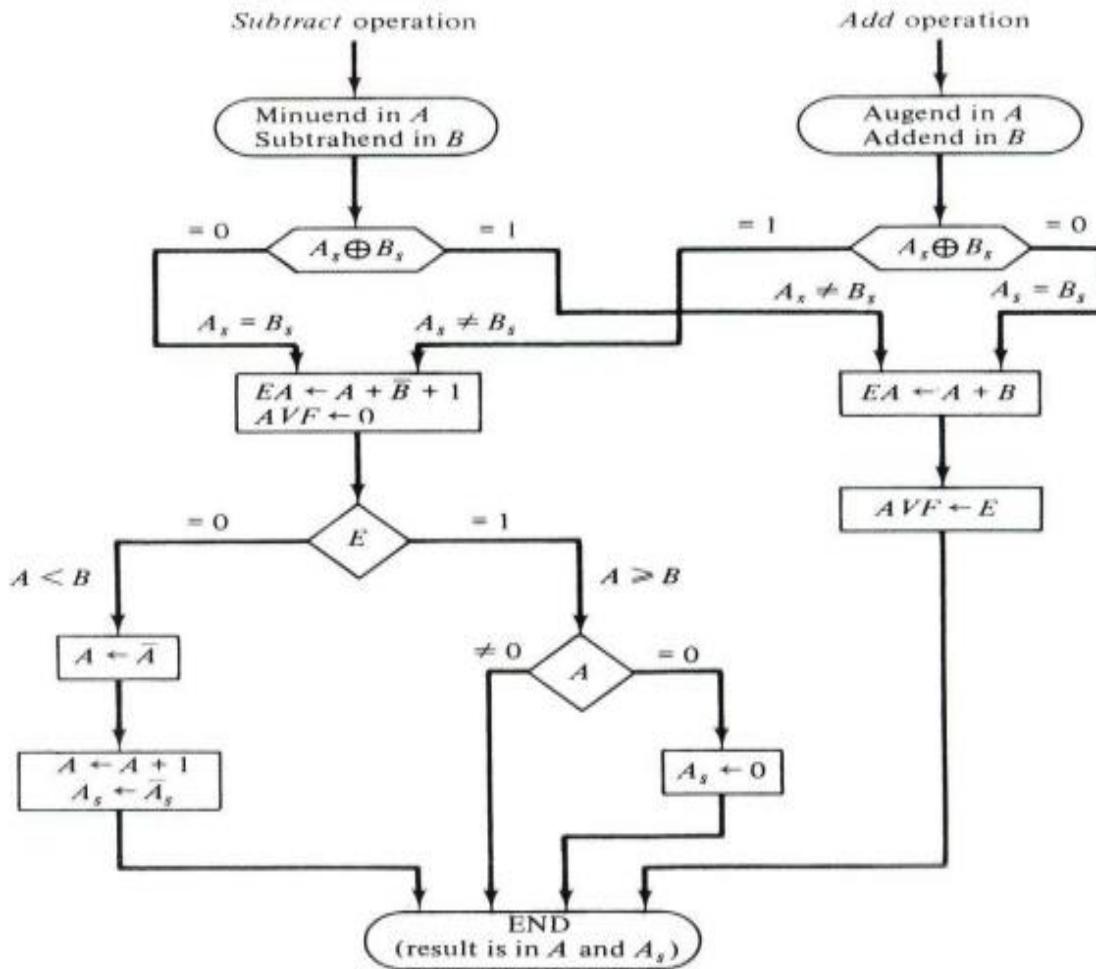
Figure 10-2 Flowchart for add and subtract operations.

- The EX-OR gate provides 0 as output when the signs are identical. It is 1 when the signs are different.
- **A + B is computed for the following and the sum is stored in EA:**
  1. When the signs are same and addition operation is required.
  2. When the signs are different and subtract operation is required.
- The carry in E after addition indicates an overflow if it is 1 and it is transferred to AVF, the add overflow flag
- **A-B = A+ B'+1 computed for the following:**
- 1. When the signs are different and addition operation is required.

- 2. When the signs are same and subtract operation is required. No overflow can occur if the numbers are subtracted and hence AVF is cleared to Zero.
- **A 1 in E indicates that A ≥ B** and the number in A is the correct result. If this number in A is zero, the sign AS must be made positive to avoid a negative zero.
- **A 0 in E indicates that A< B**. For this case it is necessary to take the 2's complement of the value in A.
- In the algorithm shown in flow chart, it is assumed that A register has circuits for micro operations complement and increment.
- Hence two complement of value in A is obtained in 2, micro operations..
- In other paths of the flow chart, the sign of the result is the same as the sign of A, so no change in AS is required.
- **However, when A < B, the sign of the result is the complement of original sign of A.**
- Hence the **complement of AS stored in AS.**
- Final Result: As and A

Addition and Subtraction with signed-2's complement Data

- The addition of two numbers in signed-2's complement form consists of adding the numbers with the sign bits treated the same as the other bits of the number.
- A carry-out of the sign-bit position is discarded.
- The subtraction consists of first taking the 2's complement of the subtrahend and then adding it to the minuend.
- The register configuration for the hardware implementation is shown in Figure below.
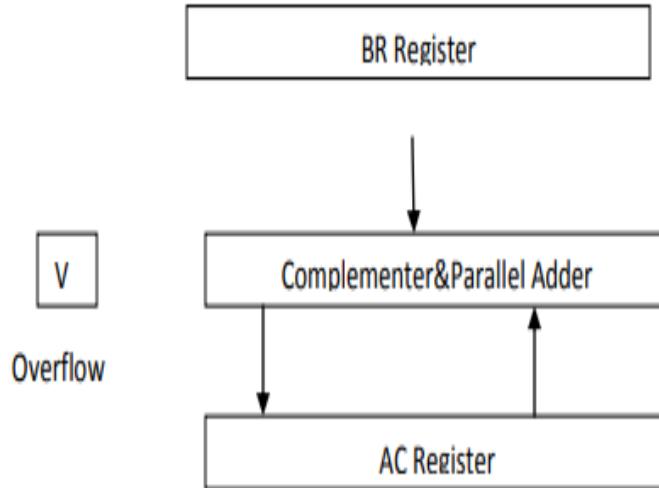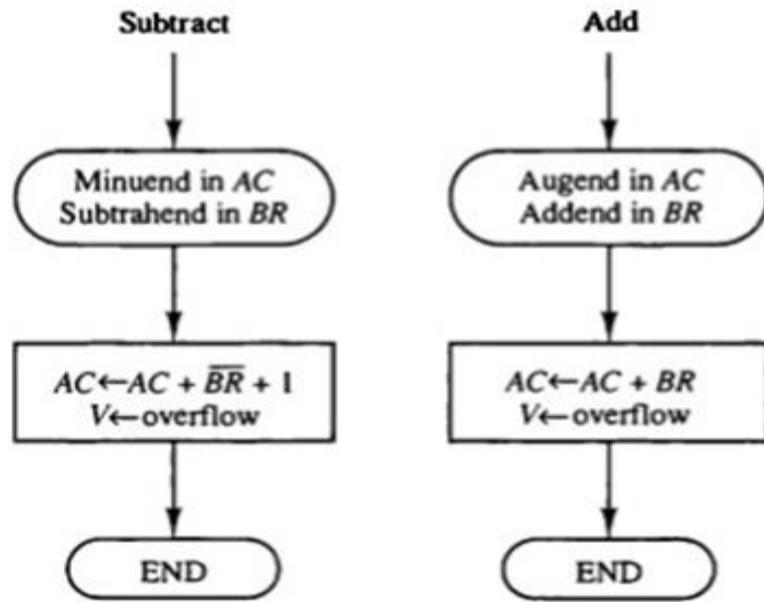
Fig: Hardware for Signed 2/s complement for addition/ subtractioin.

Hardware implementation of signed 2's complement for addition/subtraction

- Here the sign bits are not separated from the registers and named it as AC(Accumulator) and the B register(BR)
- The leftmost bit in AC and BR represents the sign bits of the numbers.
- The two sign bits are added or subtracted together with the other bits in the complementer and parallel adder.
- The overflow flip-flop V is set to1 if there is an overflow.
- The output of the carry in this case is discarded.
- The algorithm for adding and subtracting two binary numbers in signed2's complement representation is shown in the flow chart below

Algorithm for adding and subtracting numbers in 2's complement form:

Algorithm for adding and subtracting numbers in 2's complement form

- The sum is obtained by adding the contents of AC and BR (including their sign bits).

- The overflow bit V is set to 1 if the exclusive-OR of the last two carries is 1, and it is cleared to 0 otherwise.

- The subtraction operation is accomplished by adding the content of AC to the 2's complement of BR.

- Taking the 2's complement of BR has the effect of changing a positive number to negative, and vice versa.

- An overflow must be checked during this operation because the two numbers added could have the same sign.

- The programmer must realize that if an overflow occurs, there will be an erroneous result in the AC register.

Multiplication Algorithms:

- Multiplication of two fixed-point binary numbers in signed-magnitude representation is done with process of successive shift and adds operations.

- This process is best illustrated with a numerical example as follows:

```
23          10111      Multiplicand
19       ×  10011      Multiplier
            10111
           10111
          00000      +
         00000
        10111
437   110110101      Product
```

Numerical example of Multiplication

- If the multiplier bit is equal to 1, the multiplicand is copied down; otherwise zeros are copied down.
- The numbers copied down are shifted one position to the left from the previous number.
- Finally, the numbers are added and their sum forms the product.

Hardware Implementation for Signed-Magnitude Data Multiplication:
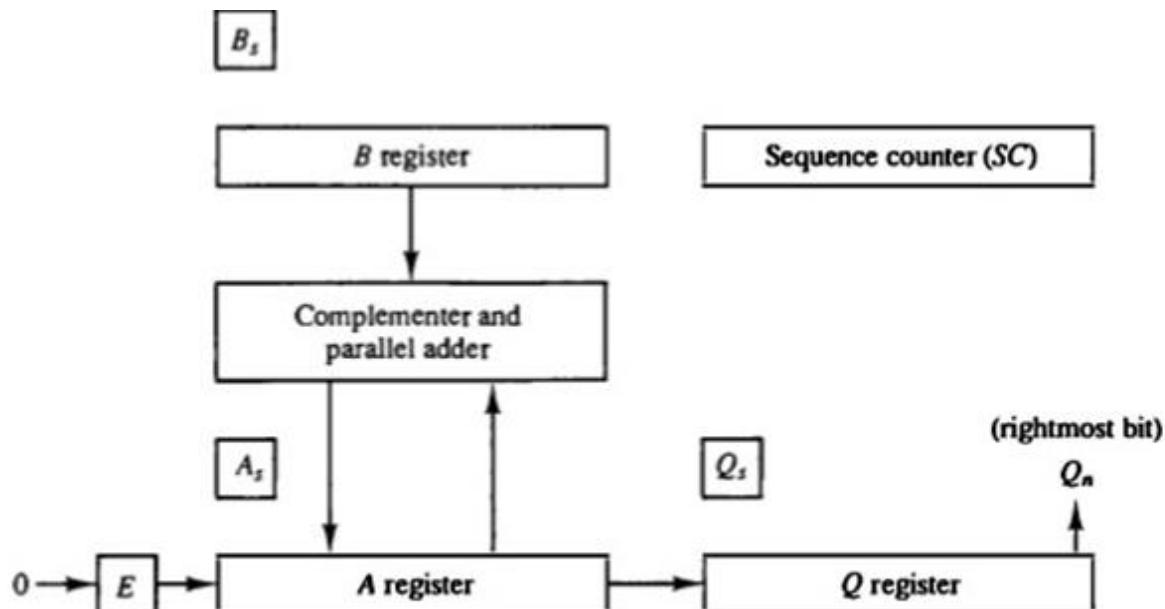
Figure 5: Hardware for multiply operation

- The hardware for multiplication consists of the equipment shown in Figure above.
- Initially, the multiplicand is in register B and the multiplier in Q.
- Their corresponding signs are stored in the flip-flops Qs and Bs
- Initially A is set to 0 as number of bits in the multiplicand.
- The **sequence counter SC** is initially set to a number equal to the number of bits in the multiplier.
- The sum of A and B forms a partial product which is transferred to the EA register.
- Both partial product and multiplier are **shifted to the right.**
- This shift will be denoted by the statement **shr EAQ** to designate the right shift depicted in Figure above.
- The least significant bit of A is shifted into the most significant position of Q, the bit from E is shifted into the most significant position of A, and 0 is shifted into E.
- After the shift, one bit of the partial product is shifted into Q, pushing the multiplier bits one position to the right.
- In this manner, the rightmost flip-flop in register Q, designated by Q_n, will hold the bit of the multiplier, which must be inspected next.
- The **counter is decremented by 1 after forming each partial product**. When the content of the counter reaches zero, the product is formed and the process stops.

Fig : Flowchart multiply operation on sign magnitude representation numbers

- Initially, the multiplicand is in B and the multiplier in Q. Their corresponding signs are in B_s and Q_s, respectively.
- **The signs are compared**, and both signs of A and Q are set to correspond to the sign of the product since a **double-length product** will be stored in registers A and Q.
- Registers A and E are cleared and the sequence counter SC is set to a

number equal to the number of bits of the multiplier.

- After the initialization, the low-order bit of the multiplier in Qn, is tested.
- If **Qn is a 1**, the multiplicand in B is added to the present partial product in A. If Q_n is a 0, nothing is done.
- Register **EAQ is then shifted once to the right** to form the new partial product.
- The sequence counter is decremented by 1 and its new value checked. If it is not equal to zero, the process is repeated and a new partial product is formed. The process stops when SC = 0.
- Note that the partial product formed in A is shifted into Q one bit at a time and eventually replaces the multiplier.
- The **final product** is available in **both A and Q,** with A holding the most significant bits and Q holding the least significant bits.
- The following table describes multiplication of binary numbers 10111(+23) and 10011(+19) which are represented using Sign Magnitude Representation.
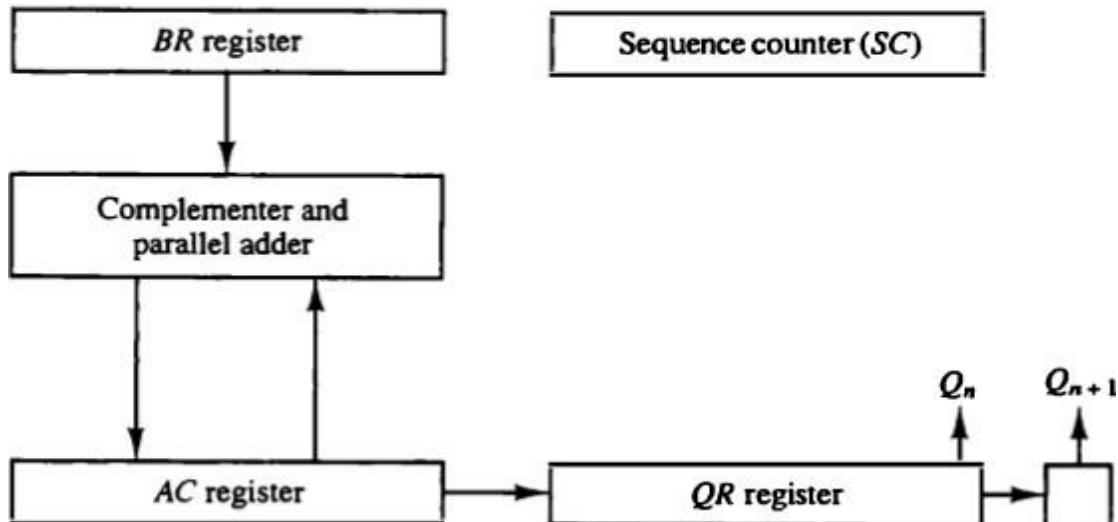
Table : Numerical Example for Binary Multiplier

Multiplicand B = 10111	E	A	Q	SC
Multiplier in Q	0	00000	10011	101
Q_n = 1; add B		10111		
First partial product	0	10111		
Shift right EAQ	0	01011	11001	100
Q_n = 1; add B		10111		
Second partial product	1	00010		
Shift right EAQ	0	10001	01100	011
Q_n = 0; shift right EAQ	0	01000	10110	010
Q_n = 0; shift right EAQ	0	00100	01011	001
Q_n = 1; add B		10111		
Fifth partial product	0	11011		
Shift right EAQ	0	01101	10101	000
Final product in AQ = 0110110101				

- Now Result is available in Registers A and Q. i.e. 0110110101 => 437 and sign bit of A is 0. So result is +437.
- The following table 3 describes multiplication of binary numbers **10011(+19)** and **00110(+6)**which are represented using Sign Magnitude Representation.
- Here Multiplicand is positive value, so $B_s = 0$. Here Multiplier is positive value, so $Q_s = 0$.
- Now $A_s = B_s + Q_s$, i.e A_s is positive; when both Bs and Qs are equal

Booth Multiplication Algorithm (for signed-2's complement numbers)

- Booth algorithm gives a procedure for multiplying binary integers in signed-2's complement representation.
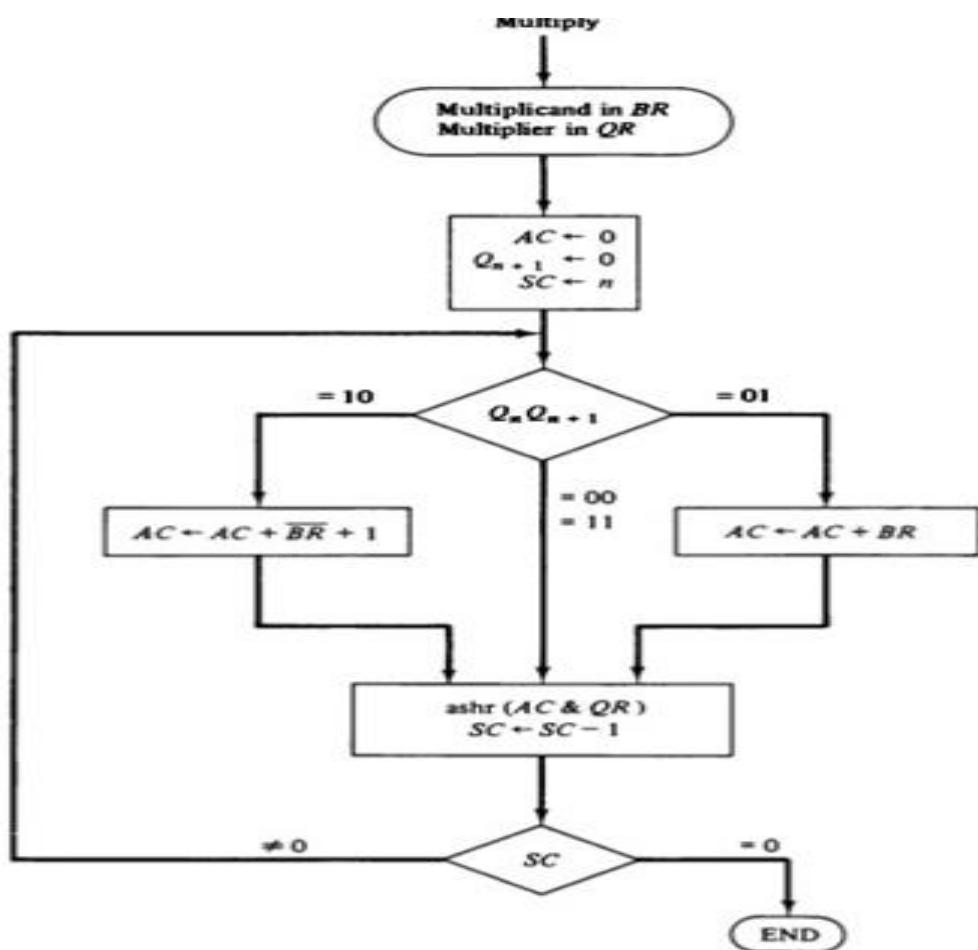
- As in all multiplication schemes, Booth algorithm requires examination of the multiplier bits and shifting of the partial product.
- Prior to the shifting, the multiplicand may be added to the partial product, subtracted from the partial product, or left unchanged according to the following rules:

1. The multiplicand is subtracted from the partial product upon encountering the first least significant 1 in a string of 1's in the multiplier.

2. The multiplicand is added to the partial product upon encountering the first 0 (provided that there was a previous 1) in a string of 0's in the multiplier.

3. The partial product does not change when the multiplier bit is identical to the previous multiplier bit.

The hardware implementation of Booth algorithm requires the register configuration shown in Figure.

Q_n designates the least significant bit of the multiplier in register QR. An extra flip-flop Q_{n+1} is appended to QR to facilitate a double bit inspection of the multiplier.

The flowchart for Booth algorithm is shown in Figure .

- AC and the appended bit Q_{n+1} are initially cleared to 0 and the sequence counter SC is set to a number n equal to the number of bits in the multiplier.
- **The two bits of the multiplier in Q_n and Q_{n+1} are inspected.**
- If the two bits are equal to 10, it means that the first 1 in a string of 1's has been encountered. This requires a subtraction of the multiplicand from the partial product in AC.
- If the two bits are equal to 01, it means that the first 0 in a string of 0's has been encountered. This requires the addition of the multiplicand to the partial product in AC.
- When the two bits are equal, the partial product does not change. An *overflow cannot* occur because the addition and subtraction of the multiplicand follow each other.
- The next step is to **shift right** the partial product and the multiplier (including **bit Q_{n+1}).**
- This is **an arithmetic shift right (ashr)** operation which shifts AC and QR to the right and leaves the sign bit in AC unchanged.
- The sequence counter is decremented and the computational loop is repeated n times.
- A numerical example of Booth algorithm is shown in Table 5. It shows the step-by-step multiplication of (-9) x (-13) = + 117.
- Here the multiplier in QR is negative and that the multiplicand in BR is also negative. The 10-bit product appears in AC and QR and is positive.

$Q_n\, Q_{n+1}$	$\begin{array}{l} BR = 10111 \\ \overline{BR} + 1 = 01001 \end{array}$	AC	QR	Q_{n+1}	SC
	Initial	00000	10011	0	101
1 0	Subtract BR	$\dfrac{01001}{01001}$			
	ashr	00100	11001	1	100
1 1	ashr	00010	01100	1	011
0 1	Add BR	$\dfrac{10111}{11001}$			
	ashr	11100	10110	0	010
0 0	ashr	11110	01011	0	001
1 0	Subtract BR	$\dfrac{01001}{00111}$			
	ashr	00011	10101	1	000

Table : Example of Multiplication with Booth Algorithm
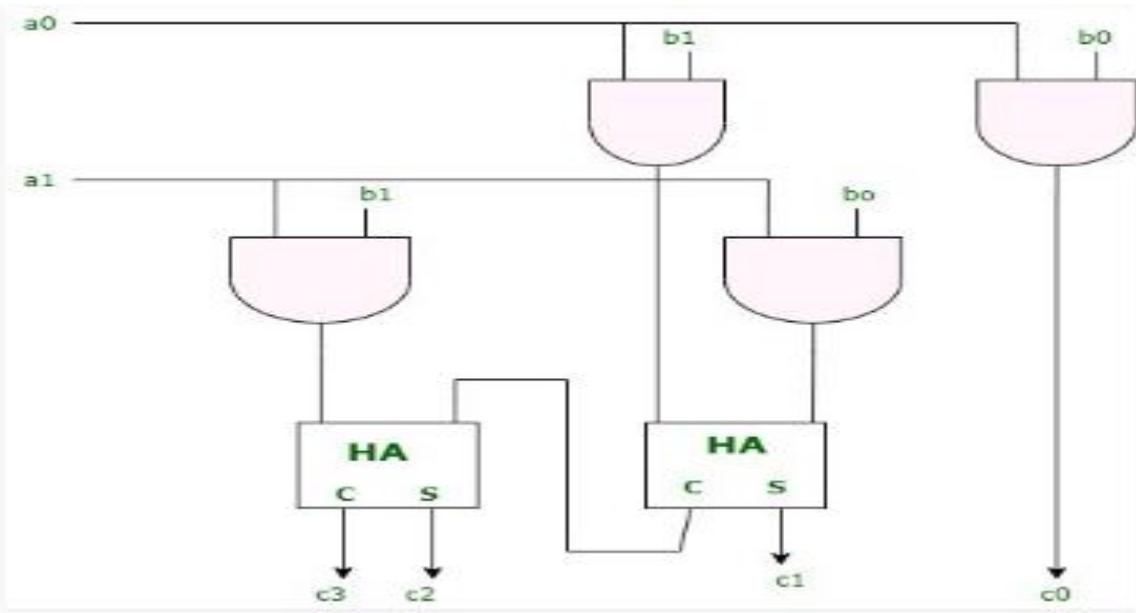
- Now Result is available in Registers AR and QR. i.e. 0001110101 =>+117.

ARRAY MULTIPLIER::

- An Array multiplier is implemented with **combinational circuit.**
- Consider the multiplication of two 2-bit numbers as shown in figure.
- The multiplicand bits are **b1 and bo**; the multiplier bits are **a1 and a0** and the product is c3c2c1c0.
- The partial product is formed by multiplying a0 by b1b0.
- The multiplication of two bits such as ao and b0 produces a result 1 if both bits are 1; otherwise , it produces a 0.
- This is identical to an AND operation and can be implemented with an AND gate.
- As shown in the figure, the first partial product is formed by means of two AND gates.
- The second partial product is formed by multiplying a1 by b1b0 and is shifted to one position to the left.
- The two partial products are added with **two half adders circuits.**

2 bit by 2 bit Array multiplier

Division Algorithm:

- Division of two fixed-point binary numbers in signed magnitude representation is performed with paper and pencil by a process of successive compare, shift and subtract operations.
- Binary division is much simpler than decimal division because here the quotient digits are either 0 or 1.
- The division process is described in Figure

Example of Division Operation:
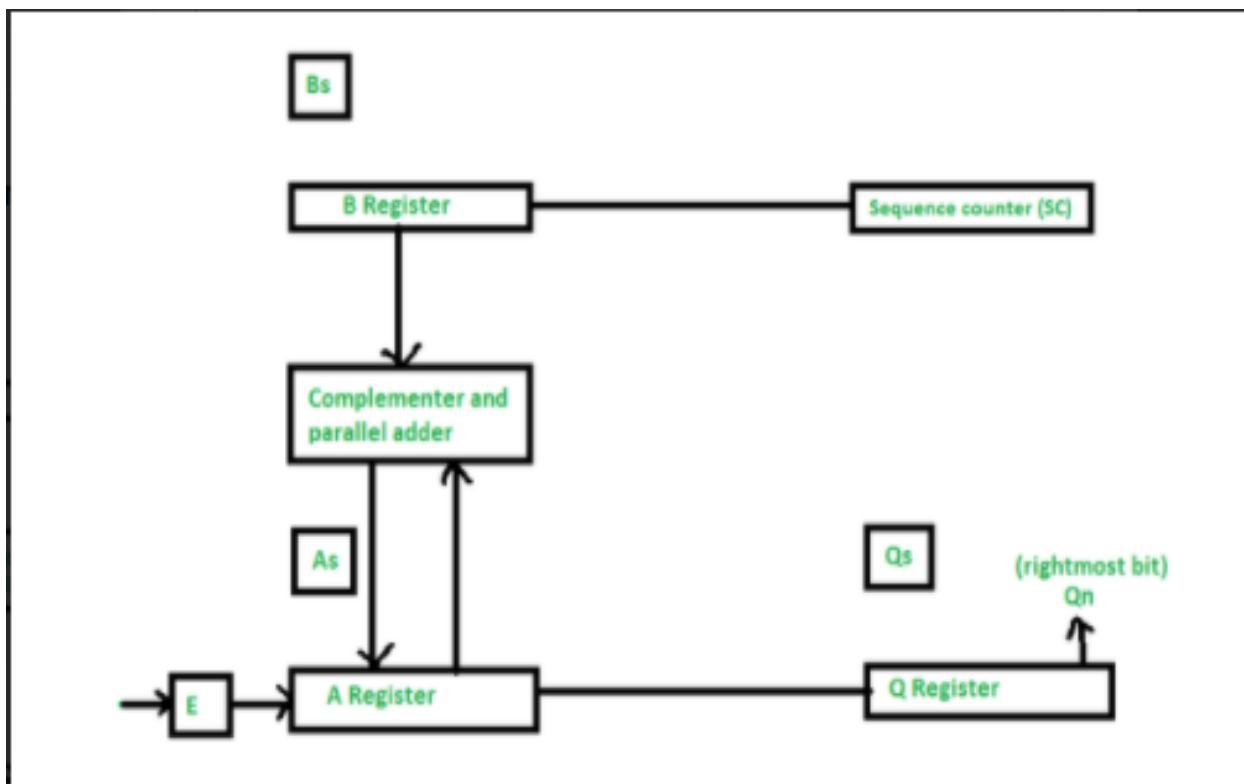
```
                    11010                 Quotient = Q
Divisor B =    ) 0111000000               Dividend = A
   10001         01110
                 011100
                 -10001

                 -010110
                 --10001

                 --001010
                 ---010100
                 ----10001

                 ----000110
                 -----00110         Remainder
```

Hardware Implementation
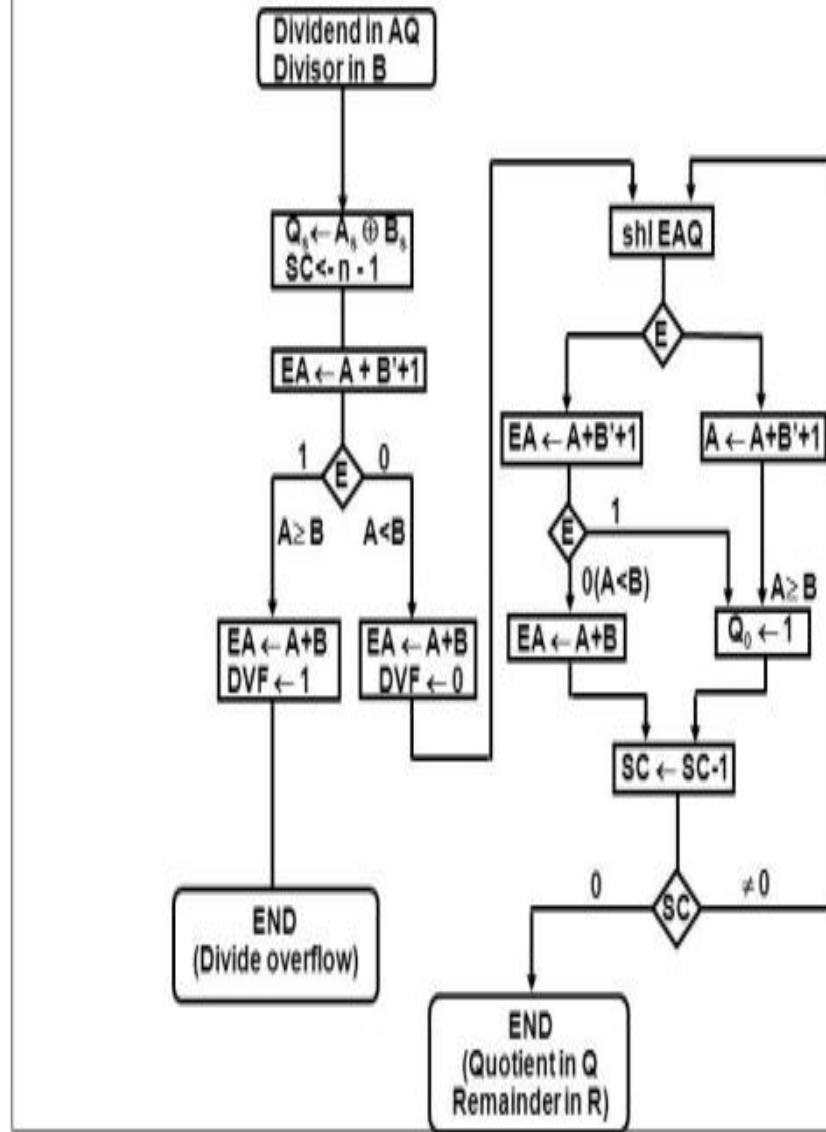
Division Operation using Pen and Paper:

- The divisor is compared with the five most significant bits of the dividend.

- Since the 5-bit number is smaller than B, we again repeat the same process.

- Now the 6-bit number is greater than B, so we place a 1 for the quotient bit in the sixth position above the dividend.

- Now we **shift the divisor once to the right** and subtract it from the dividend.

- The difference is known as a partial remainder because the division could have stopped here to obtain a quotient of 1 and a remainder equal to the partial remainder.

Hardware Implementation for Signed-Magnitude Data

- In hardware implementation for signed-magnitude data in a digital computer, it is convenient to change the process slightly.

- Instead of shifting the divisor to the right, two dividends, or **partial remainders, are shifted to the left,** thus leaving the two numbers in the required relative position.

- Subtraction is achieved by adding A to the 2's complement of B.

- End carry gives the information about the relative magnitudes.

- The hardware required is identical to that of multiplication.

FLOWCHART OF DIVIDE OPERATION

Dividend in AQ
Divisor in B

$Q_s \leftarrow A_s \oplus B_s$
$SC \leftarrow n-1$

$EA \leftarrow A + B'+1$

1 | E | 0
$A \geq B$ | $A < B$

$EA \leftarrow A+B$
$DVF \leftarrow 1$

$EA \leftarrow A+B$
$DVF \leftarrow 0$

END
(Divide overflow)

shl EAQ

E

$EA \leftarrow A+B'+1$ | $A \leftarrow A+B'+1$

E | 1

$0(A<B)$ | $A \geq B$

$EA \leftarrow A+B$ | $Q_0 \leftarrow 1$

$SC \leftarrow SC-1$

0 | SC | $\neq 0$

END
(Quotient in Q
Remainder in R)

- Comparing a partial remainder with the divisor continues the process.
- If the **partial remainder is greater than or equal to the divisor**, the quotient bit is equal to 1.
- The divisor is then shifted right and subtracted from the partial remainder. If the partial remainder is smaller than the divisor, the quotient bit is 0 and no subtraction is needed.
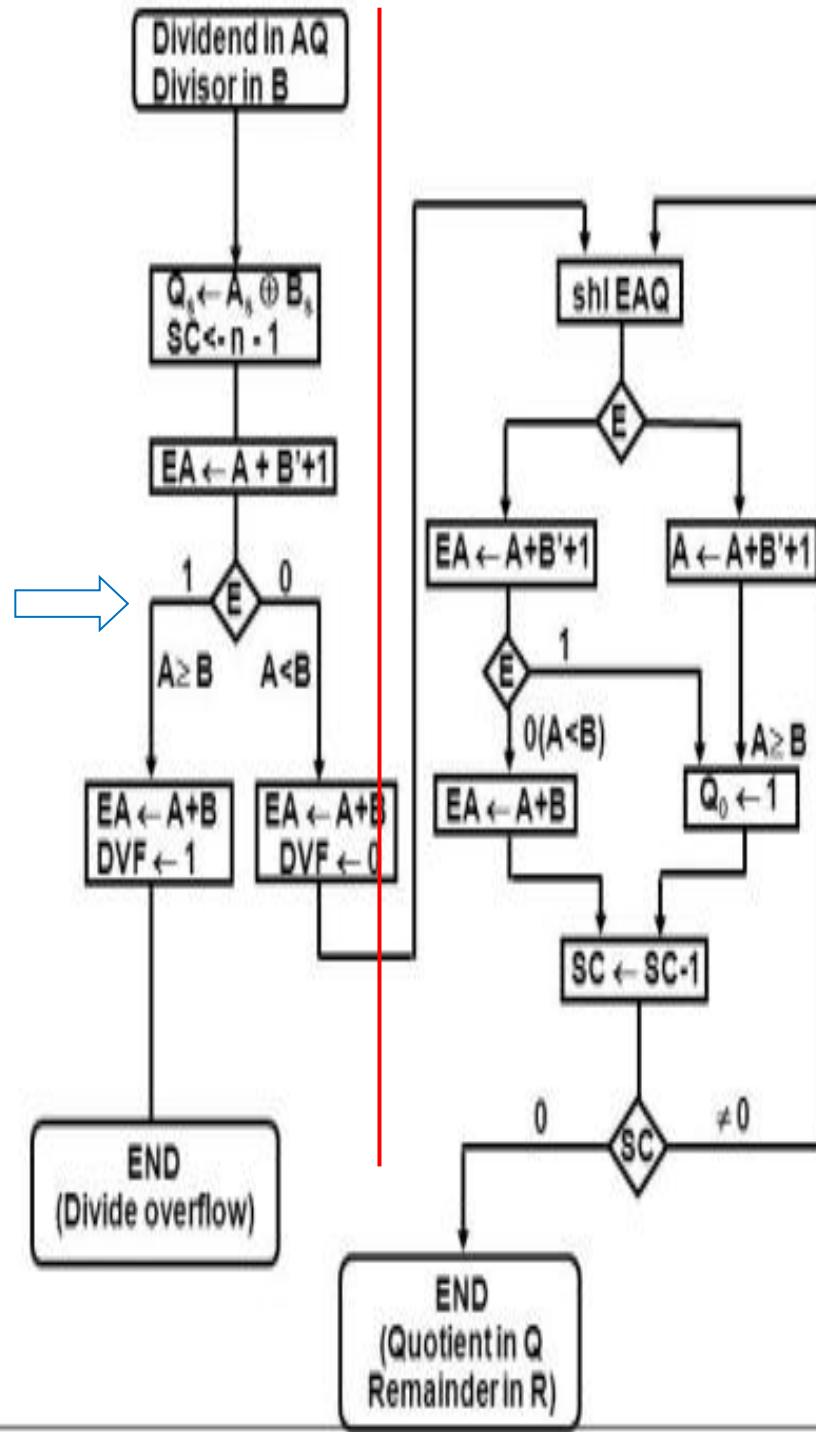
- The divisor is shifted once to the right in any case. Obviously the result gives both a quotient and a remainder.
- **Register EAQ is now shifted to the left** with 0 inserted into Qn and the previous value of E is lost.
- The example is given in Figure to clear the proposed division process.
- The divisor is stored in the B register and the double-length dividend is stored in registers A and Q.
- The dividend is shifted to the left and the divisor is subtracted by adding its 2's complement value.
- End carry(E) gives the information about the relative magnitudes.
- If E = 1, it signifies that A ≥ B. The quotient bit 1 is inserted into Qn and the partial remainder is shifted to left to the process.
- If E = 0, it signifies that A < B. So the quotient in Qn remains a 0.
- The value of B is added to restore the partial remainder in A to restore to its previous value.
- The partial remainder is shifted to the left and the process is repeated again until all quotient bits are formed.
- The remainder is then found in register A and the quotient is in register Q.
- Before showing the algorithm in flowchart form, we have to **consider the sign of the result and a overflow condition.**

Considering the sign of the result and a Overflow condition.

FLOWCHART OF DIVIDE OPERATION

- Initially, the dividend is in A & Q and the divisor is in B.
- Sign of result is transferred into Q, to be the part of quotient. Then a constant is set into the SC to specify the number of bits in the quotient.
- Since an operand must be saved with its sign, one bit of the word will be inhabited by the sign, and the magnitude will be composed of n -1 bits.
- **The condition of divide-overflow** is checked by subtracting the divisor in B from the half of bits of the dividend stored in A.
- **If A ≥ B, DVF is set and the operation is terminated before time.**
- **If A < B, no overflow condition occurs and so the value of the dividend is reinstated by adding B to A**.

Normal Division Process using Flowchart:

- The division of the magnitudes starts by shl dividend in AQ to left in the high-order bit shifted into E.
- Note – If shifted a bit into E is equal to 1, and we know that EA > B as EA comprises a 1 followed by n -1 bits whereas B comprises only n -1 bits). In this case, B must be subtracted from EA, and 1 should insert into Q, for the quotient bit.
- If the shift-left operation (shl) inserts a 0 into E, the divisor is subtracted by adding its 2's complement value and the carry is moved into E.
- If E = 1, it means that A ≥ B; thus, Q, is set to 1. If E = 0, it means that A < B and the original number is restored or reimposed by adding B into A.
- Now, this process is repeated with register A containing the partial remainder. After n-1 times, the final result is available in A and Q registers.

Example of Binary Division with Digital Hardware:

Divisor B = 10001	E	A	Q	SC
Dividend: shl EAQ add $\bar{B}+1$	 0 	01110 11100 <u>01111</u>	00000 00000 	5
$E=1$ Set $Q_n=1$ shl EAQ Add $\bar{B}+1$	1 1 0 	01011 01011 10110 <u>01111</u>	 00001 00010 	4
$E=1$ Set $Q_n=1$ shl EAQ Add $\bar{B}+1$	1 1 0 	00101 00101 01010 <u>01111</u>	 00011 00110 	3
$E=0$; leave $Q_n=0$ Add B	0 	11001 <u>10001</u>	00110 	2
Restore remainder shl EAQ Add $\bar{B}+1$	1 0 	01010 10100 <u>01111</u>	 01100 	
$E=1$ Set $Q_n=1$ shl EAQ Add $\bar{B}+1$	1 1 0 	00011 00011 00110 <u>01111</u>	 01101 11010 	1
$E=0$; leave $Q_n=0$ Add B	0 	10101 <u>10001</u>	11010 	
Restore remainder Neglect E Remainder in A: Quotient in Q:	1 	00110 00110 	11010 11010	0

Input-Output Organization:

Contents:

Peripheral Devices,

Input-Output Interface,

Asynchronous data transfer,

Modes of Transfer,

Priority Interrupt,

Direct memory Access,

Input –Output Processor (IOP)

Peripheral Devices:

The Input / output organization of computer depends upon the size of computer and the peripherals connected to it.

The I/O Subsystem of the computer, provides an efficient mode of communication between the central system and the outside environment

Input/output devices attached to the computer are called Peripheral devices.

The most common input output devices are:

i) Monitor

 ii) Keyboard

iii) Mouse

iv) Printer

v) Magnetic tapes

Input Devices

- Keyboard
- Optical input devices
- - Card Reader
- - Paper Tape Reader
- - Bar code reader
- - Digitizer
- - Optical Mark Reader
- Magnetic Input Devices
- - Magnetic Stripe Reader
- Screen Input Devices
- - Touch Screen
- - Light Pen
- - Mouse
- Analog Input Devices

Output Devices

- Card Puncher, Paper Tape Puncher
- CRT
- Printer (Impact, Ink Jet,
        Laser, Dot Matrix)
- Plotter
- Analog
- Voice

ASCII(American Standard Code for Information Interchange)- Alphanumeric Characters:

- Input/output devices that communicate with people and the computer are usually involved in the transfer of Alphanumeric Information to and from the device and the computer.
- The standard binary code for the alphanumeric characters is ASCII
- It uses **7 bits to code 128 characters**.
- ASCII code contains 94 characters that are printable and 34 characters that are nonprinting characters used for various control functions.
- Among 94, 26 used for uppercase letters, 26 used for lowercase letters,10 are used for numerical and 32 are used for special characters.
- 34 control characters are used for routing and arranging the printed text in a prescribed format
- **3 types of control characters:**

  1. Format Effectors (control the layout of printing includes BS-Back space,HT-Horizontaltab,CR-Carriage Return)

  2. Information Separators(used to separate data into paragraphs & pages includes RS-record seperator and FS-file seperator)

  3. Communication control characters (useful for transmission of text between remote terminals includes STX-Start of text, ETX-End of text)

Input - Output Interface

- 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 central processing unit.

- The purpose of communication link is to resolve the differences that exist between the central computer and each peripheral.

The Major Differences are:-
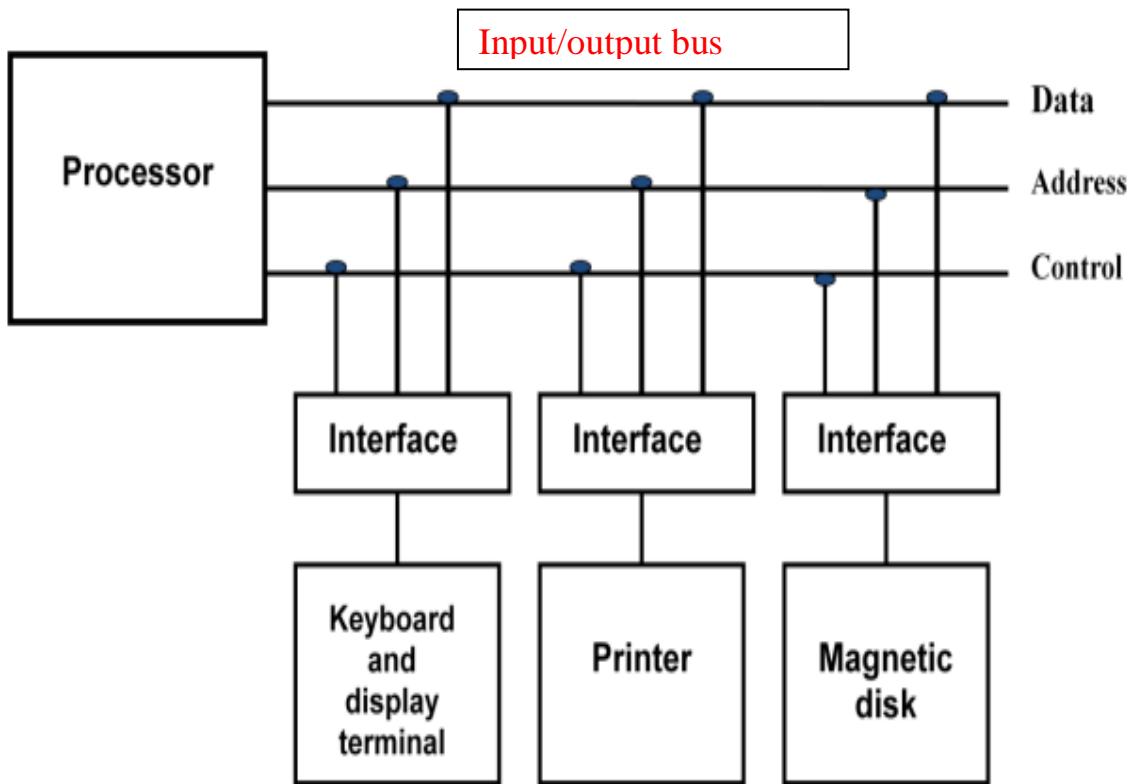
1. Peripherals are electromechnical and electromagnetic devices and CPU and memory are electronic devices. Therefore, a conversion of signal values may be needed.

2. The data transfer rate of peripherals is usually slower than the transfer rate of CPU and consequently, a synchronization mechanism may be needed.

3. Data codes and formats in the peripherals differ from the word format in the CPU and memory.

4. The operating modes of peripherals are different from each other and 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 components between the CPU and Peripherals to supervises and synchronizes all input and out transfers

These components are called Interface Units because they interface between the processor bus and the peripheral devices.

I/O BUS and Interface Module:

- It defines the typical link between the processor and several peripherals as shown in figure.

Connection of I/O bus to input-output devices

- The I/O Bus consists of data lines, address lines and control lines.
- The I/O bus from the processor is attached to all peripherals interface.
- To communicate with a particular device, the processor places a device address on address lines.
- **Each Interface decodes the address and control received from the I/O bus, interprets them for peripherals and provides signals for the peripheral controller.**
- It is also synchronizes the data flow and supervises the transfer between peripheral and processor.
- Each peripheral has its own controller. For example, the printer controller controls the paper motion, the print timing

- The processor provides a function code in the control lines.
- The control lines are referred **as I/O command.**
- The commands are as following:
- Control command- A control command is issued to activate the peripheral and to inform it what to do.
- Status command- A status command is used to test various status conditions in the interface and the peripheral.
- Data Output command- A data output command causes the interface to respond by transferring data from the bus into one of its registers.
- Data Input command- The data input command is the opposite of the data output.

I/O Bus Versus Memory Bus:

- To addition to communicate with I/O, the processor must communicate with the memory unit.
- Like the I/O bus, the memory bus contains data, address and read/write control lines.
- There are 3 ways that computer buses can be used to communicate with memory and I/O:

  i. Use two Separate buses , one for memory and other for I/O

  ii. Use one common bus for both memory and I/O but separate control lines for each.

  iii. Use one common bus for memory and I/O with common control lines.

- In the first method, the computer has independent sets of data, address and control buses one for accessing memory and other for I/O.
- This is done in computers that provide a separate I/O processor (IOP).

- The purpose of IOP is to provide an independent pathway for the transfer of information between external device and internal memory.

ISOLATED I/O Bus Versus MEMORY MAPPED I/O Bus

Isolated I/O Bus :

The distinction between **Memory transfer and I/O transfer** is made through separate read and Write lines.

During an I/O transfer, the **I/O read** and **I/O write** contro lsignals are enabled.

During an Memory transfer, the **Memory read** and **Memory write** contro lsignals are enabled.

This configuration isolates all I/O interface addresses from Memory Addresses.

- Separate (isolated) memory and I/O address spaces
- Distinct input and output instructions
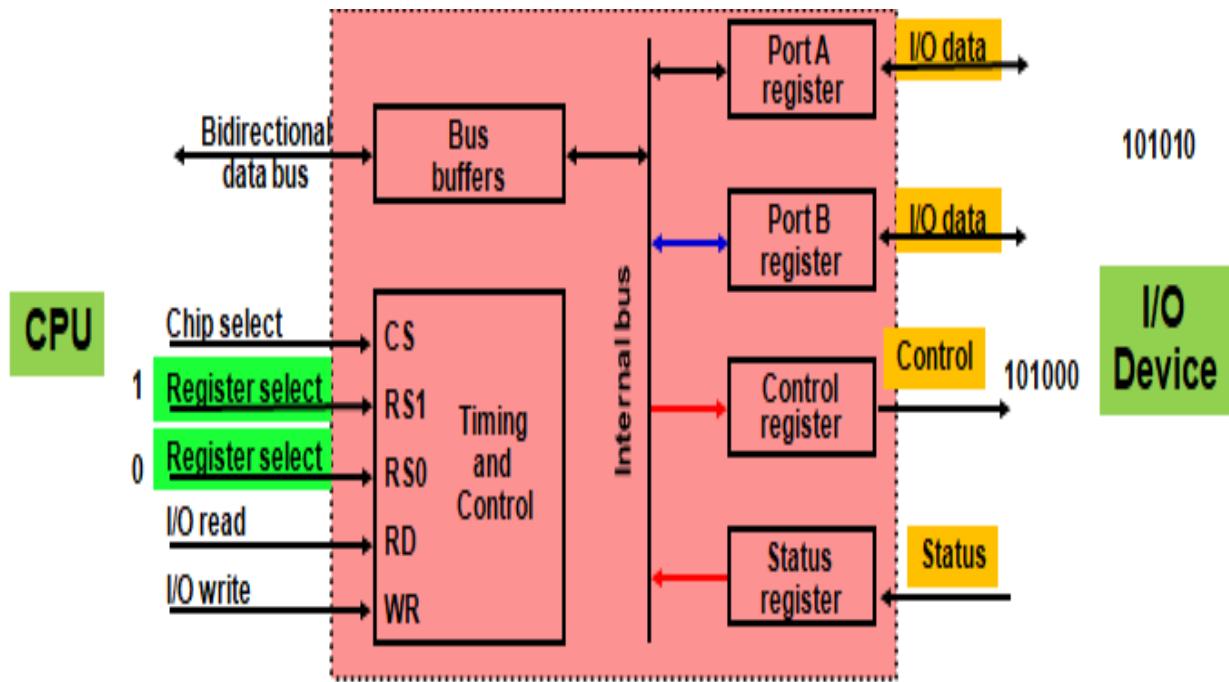
Memory Mapped I/O Bus:

- A single set of read/write control lines(no distinction between memory and I/O transfer)
- Memory and I/O addresses share the common address space (reduces memory address range available).
- No specific input or output instruction
- The same memory reference instructions can be used for I/O transfers
- Considerable flexibility in handling I/O operations

Example of I/O INTERFACE:

It consists of two data registers called ports, a control register, a status register, Bus buffers and Timing and control circuits.

The chip select and register select determines the address assigned to interface.

CS	RS1	RS0	Register selected
0	x	x	None - data bus in high-impedence
1	0	0	Port A register
1	0	1	Port B register
1	1	0	Control register
1	1	1	Status register

Asynchronous Data Transfer:

Two units such as CPU and I/O interface are designed independently of each other and the internal timing of each unit is independent of each other. In this case, the two units are said to be asynchronous.

- This Scheme is used when speed of I/O devices does not match with microprocessor, and timing characteristics of I/O devices is not predictable.
- In this method, process initiates the device and checks its status. As a result, CPU has to wait till I/O device is ready to transfer data.
- When device is ready CPU issues instruction for I/O transfer. In this method

two types of techniques are used based on signals before data transfer.

i. Strobe Control

ii. Handshaking

Strobe pulse

A strobe pulse is supplied by one unit to indicate the other unit when the transfer has to occur(or time at which data is being transferred)
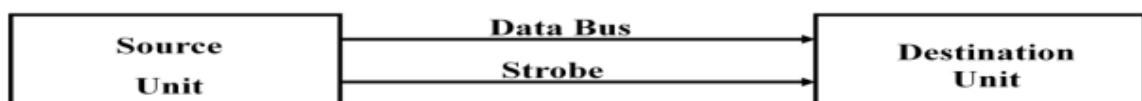
Handshaking

A control signal is accompanied with each data being transmitted to indicate the presence of data The receiving unit responds with another control signal to acknowledge receipt of the data.
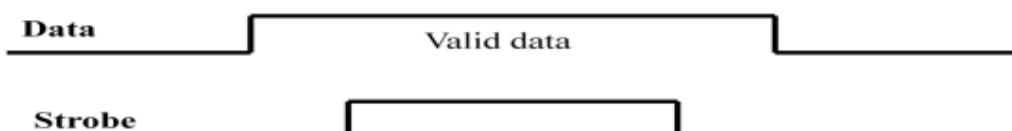
Strobe Signal :

The strobe control method of Asynchronous data transfer employs a single control line to time each transfer. The strobe may be activated by either the source or the destination unit.

Data Transfer Initiated by Source Unit(source initiated strobe signal for data transfer)
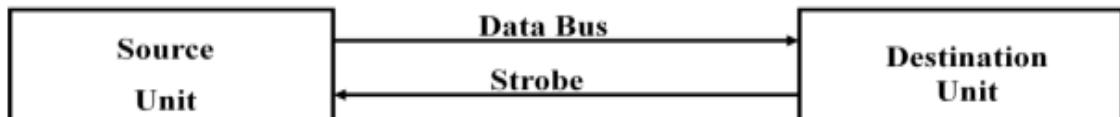
(a)   **Block Diagram**

(b)   **Timing Diagram**

Source-Initiated strobe for Data Transfer
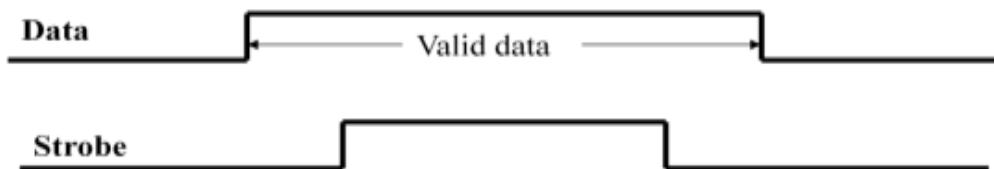
- In the block diagram fig. (a), the data bus carries the binary information from source to destination unit.
- Typically, the bus has multiple lines to transfer an entire byte or word.

- The strobe is a single line that informs the destination unit when a valid data word is available.
- The timing diagram fig. (b) the source unit first places the data on the data bus.
- The information on the data bus and strobe signal remain in the active state to allow the destination unit to receive the data.

Data Transfer Initiated by Destination Unit(Destinaation initiated strobe signal for data transfer)

(a) Block Diagram

(b) Timing Diagram

Destination-Initiated strobe for Data Transfer

- In this method, the destination unit activates the strobe pulse, to informing the source to provide the data.
- The source will respond by placing the requested binary information on the data bus.
- The data must be valid and remain in the bus long enough for the destination unit to accept it.
- When accepted the destination unit then disables the strobe and the source unit removes the data from the bus

The disadvantage of the strobe method is that, the source unit initiates the transfer has no way of knowing whether the destination unit has actually received the data item that was places in the bus.

Similarly, a destination unit that initiates the transfer has no way of knowing whether the source unit has actually placed the data on bus. The Handshaking method solves this problem
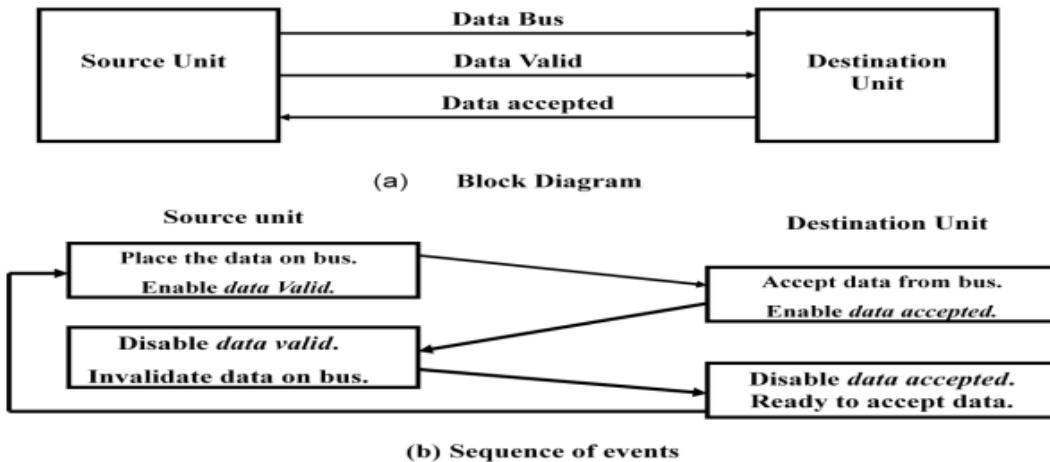
Handshaking

- The handshaking method solves the problem of strobe method by introducing a second control signal that provides a reply to the unit that initiates the transfer.

Principle of Handshaking:

- The basic principle of the two-wire handshaking method of data transfer is as follow:
- One control line is in the same direction as the data flows in the bus from the source to destination.
- It is used by source unit to inform the destination unit whether there a valid data in the bus.
- The other control line is in the other direction from the destination to the source.
- It is used by the destination unit to inform the source whether it can accept the data. The sequence of control during the transfer depends on the unit that initiates the transfer.
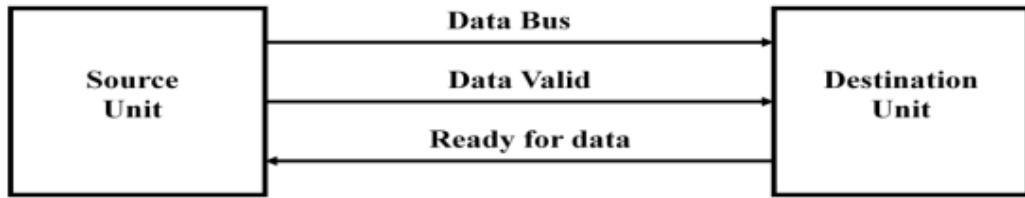
Source Initiated Transfer using Handshaking:

(a)   Block Diagram

(b) Sequence of events
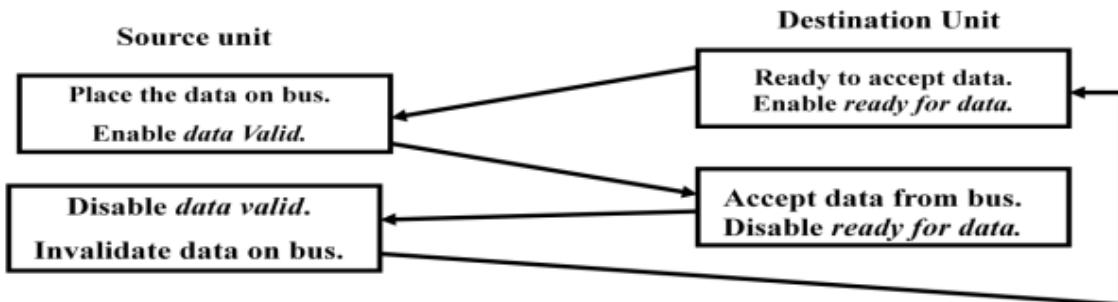
- The sequence of events shows four possible states that the system can be at any given time.
- The source unit initiates the transfer by placing the data on the bus and enabling its data valid signal.
- The data accepted signal is activated by the destination unit after it accepts the data from the bus.
- The source unit then disables its data accepted signal and the system goes into its initial state .

Destination Initiated Transfer Using Handshaking:

- The name of the signal generated by the destination unit has been changed to ready for data to reflects its new meaning.
- The source unit in this case does not place data on the bus until after it receives the ready for data signal from the destination unit.
- From there on, the handshaking procedure follows the same pattern as in the source initiated case.
- The only difference between the Source Initiated and the Destination Initiated transfer is in their choice of Initial sate

(a) Block Diagram

(b) Sequence of events

Destination-Initiated transfer using Handshaking

- ## Advantage of the Handshaking method:

- The Handshaking scheme provides degree of flexibility and reliability because the successful completion of data transfer relies on active participation by both units.

- If any of one unit is faulty, the data transfer will not be completed. Such an error can be detected by means of a Timeout mechanism which provides an alarm if the data is not completed within time.

Asynchronous Serial Transmission:
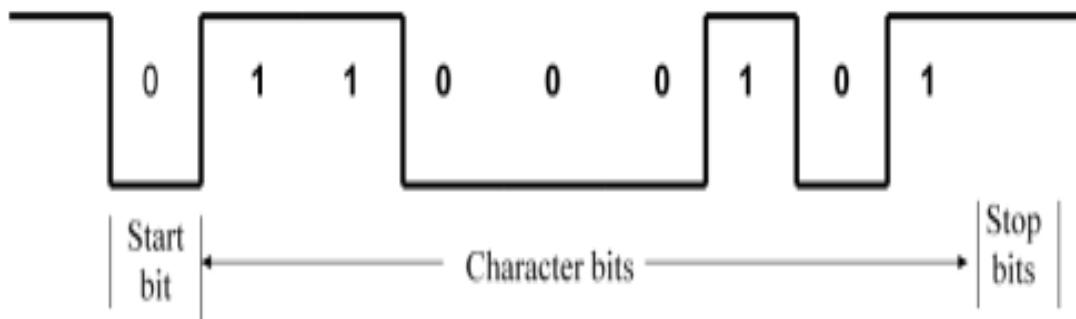
- The transfer of data between two units is serial or parallel.

- In parallel data transmission, n bit in the message must be transmitted through n separate conductor path.

- In serial transmission, each bit in the message is sent in sequence one at a time.

- Parallel transmission is faster but it requires many wires. It is used for short

distances and where speed is important.

- Serial transmission is slower but is less expensive.

- In Asynchronous serial transfer, each bit of message is sent a sequence at a time, and binary information is transferred only when it is available. When there is no information to be transferred, line remains idle.

- In this technique each character consists of three points :

  i. Start bit

  ii. Character bit

  iii. Stop bit

- **Start Bit-** First bit, called start bit is always zero and used to indicate the beginning character.

- **Stop Bit-** Last bit, called stop bit is always one and used to indicate end of characters. Stop bit is always in the 1- state and frame the end of the characters to signify the idle or wait state.

- **Character Bit**- Bits in between the start bit and the stop bit are known as character bits. The character bits always follow the start bit.

Asynchronous Serial Transmission
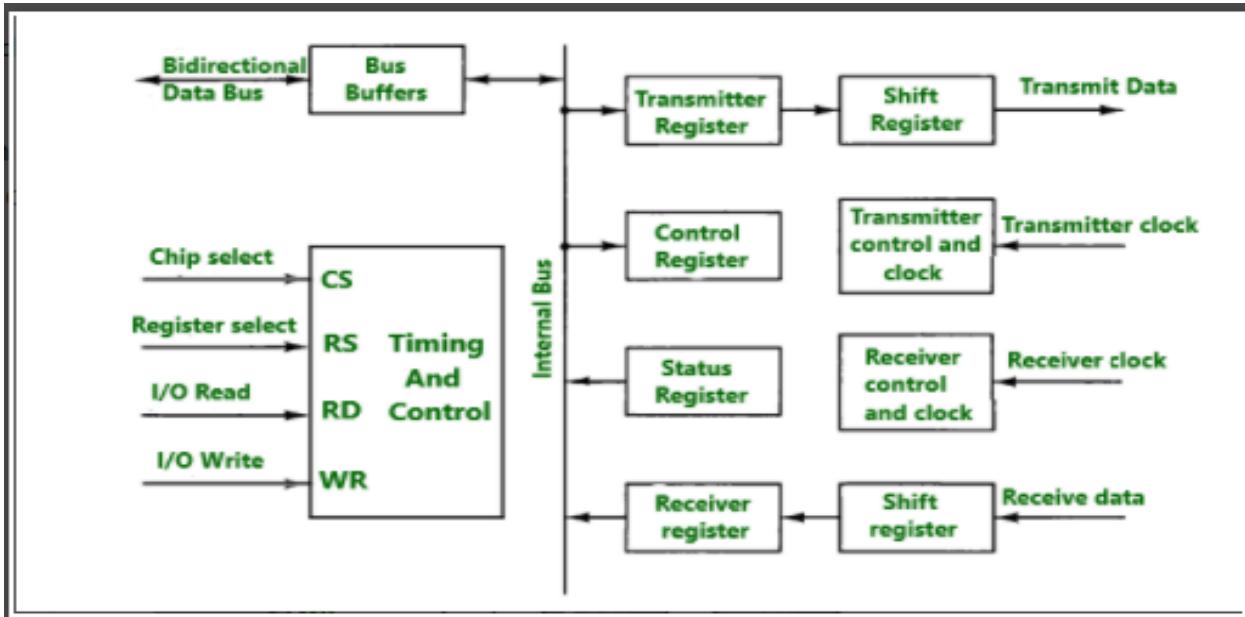

Asynchronous Serial Transmission

Serial Transmission of Asynchronous is done by two ways:

a) **Asynchronous Communication Interface**

 b) **First In First out Buffer**

Asynchronous Communication Interface:

- It works as both a receiver and a transmitter.
- Its operation is initialized by CPU by sending a byte to the control register.
- The transmitter register accepts a data byte from CPU through the data bus and transferred to a shift register for serial transmission.
- The receive portion receives information into another shift register, and when a complete data byte is received it is transferred to receiver register.
- CPU can select the receiver register to read the byte through the data bus. Data in the status register is used for input and output flags.

First In First Out Buffer (FIFO):

- A First In First Out (FIFO) Buffer is a memory unit that stores information in such a manner that the first item is in the item first out.
- A FIFO buffer comes with separate input and output terminals.

- The important feature of this buffer is that it can input data and output data at two different rates.
- When placed between two units, the FIFO can accept data from the source unit at one rate, rate of transfer and deliver the data to the destination unit at another rate.
- If the source is faster than the destination, the FIFO is useful for source data arrive in bursts that fills out the buffer.
- FIFO is useful in some applications when data are transferred asynchronously.

Modes of Data Transfer:

- The data transfer can be handled by various modes.
- Some of the modes use CPU as an intermediate path, others transfer the data directly to and from the memory unit and

- This can be handled by 3 following ways:

  i. Programmed I/O

  ii. Interrupt-Initiated I/O

 **iii. Direct Memory Access (DMA)**

Programmed I/O Mode
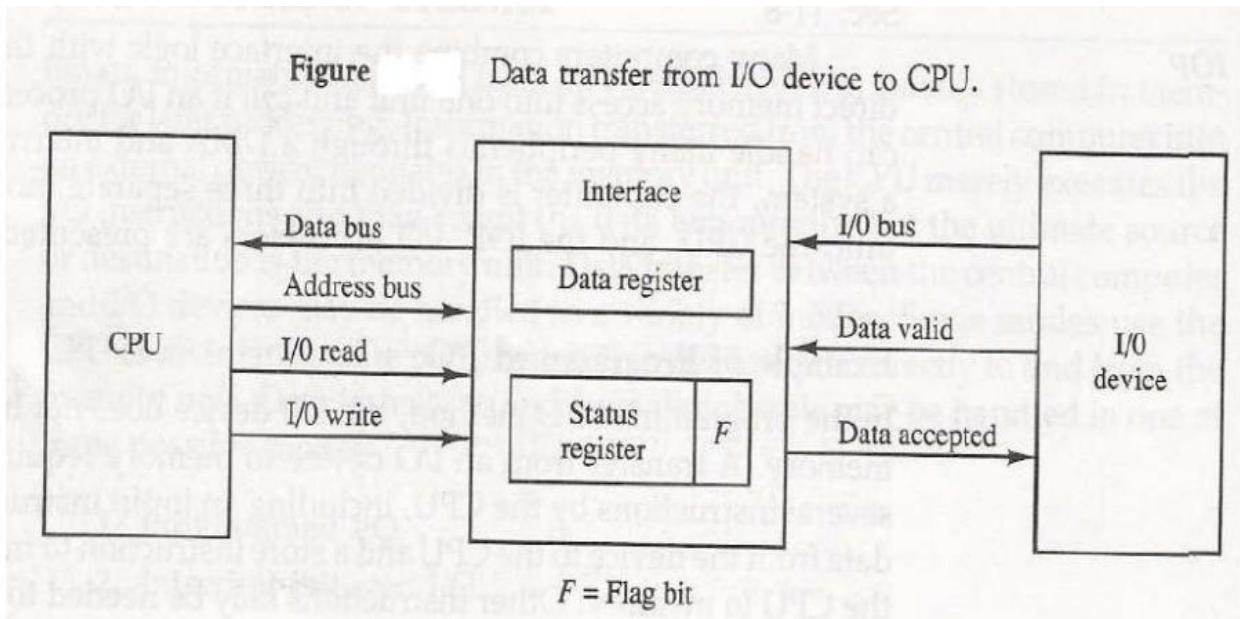
- In this mode, Programmed I/O operations are the results of I/O instructions which is a part of computer program.
- Each data transfer is initiated by a instruction in the program.
- Normally the transfer is from a CPU register to peripheral device or vice-versa.
- Once the data transfer is initiated the CPU starts monitoring the interface to see when next transfer can made.
- The instructions of the program keep close tabs on everything that takes

place in the interface unit and the I/O devices.
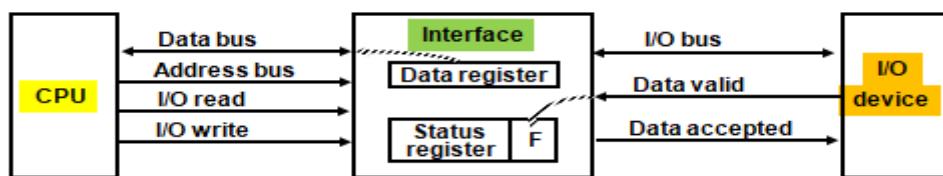
- In the Programmed I/O Mode, the CPU stays in the program loop until the I/O indicates that it is ready for data transfer.

Example of Programmed I/O:

- An example of data transfer from an I/O device through an interface into the CPU is shown in figure:


Figure   Data transfer from I/O device to CPU.
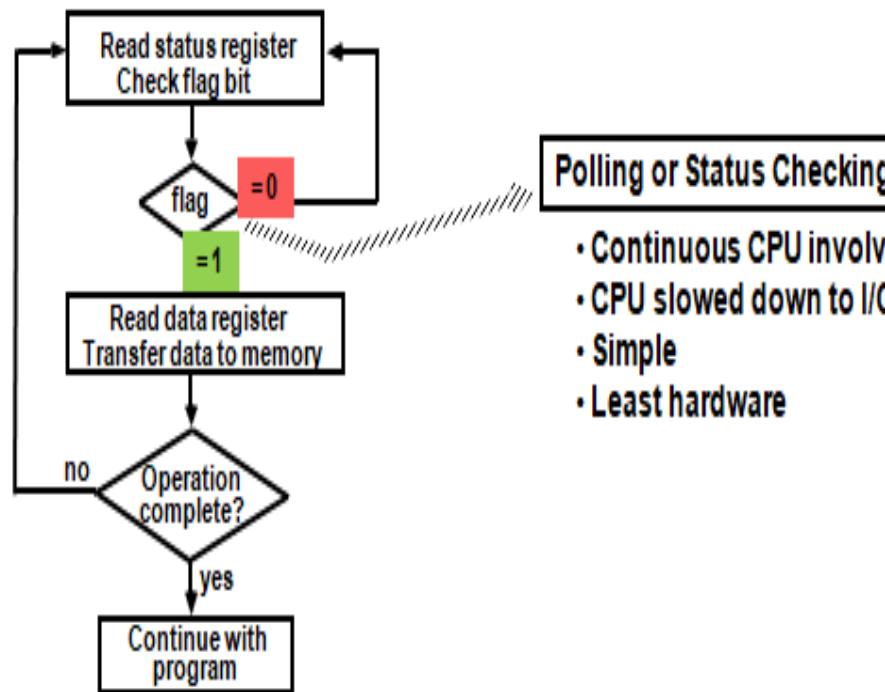
- The peripheral device transfers bytes of bytes of data one at a time when they are available.

- When **a byte of data is available, the device places it in the I/O bus** and enables data valid line.

- The interface accepts the data into its data register and enables data accepted line.

- The interface sets a bit in the status register that is referred as Flag bit(F).

- A program is written for the computer too check for flag in status register to determine if a byte has placed in the data register by the I/O device.
- This is done by reading the status register to a CPU register and checking the value of flag bit.
- **If F=1, CPU reads the data from data register.**
- **If F=0, CPU/interface disables the data accepted line.**
- A flowchart of the program is written for CPU is shown below
- Here the device is sending a sequence of bytes that must be stored in memory.
- The transfer of data requires three instructions:

1. Read the status register.
2. Check the status of the flag bit and branch to step 1 if not set or to step 3 if set.
3. Read the data register.

Flowchart:

Drawback of the Programmed I/O :

- The main drawback of the Program Initiated I/O was that the CPU has to monitor the units all the times when the program is executing.
- Thus the **CPU stays in a program loop** until the I/O unit indicates that it is ready for data transfer.
- This is a time consuming process and the CPU time is wasted a lot in keeping an eye to the executing of program.
- To remove this problem an Interrupt facility and special commands are used.
-

Interrupt-Initiated I/O :

- In this method an interrupt facility called **an interrupt command** is used to inform the device about the start and end of transfer.
- In the meantime the CPU executes other program. When the interface determines that the device is ready for data transfer it generates an Interrupt Request and sends it to the computer.
- When the CPU receives such an signal, it temporarily stops the execution of the program and branches to a service program to process the I/O transfer and after completing it returns back to task, what it was originally performing.
- In this type of IO, computer does not check the flag. It continue to perform its task.
- **Whenever any device wants the attention, it sends the interrupt signal to the CPU.**
- CPU then deviates from what it was doing, store the return address from PC and branch to the address of the subroutine.
- There are two ways of choosing the branch address:

- **Vectored Interrupt**
- **Non-vectored Interrupt**
- In **vectored interrupt the source that interrupts the CPU provides the branch information**. This information is called interrupt vectored.
- In non-vectored interrupt, **the branch address is assigned to the fixed address in the memory.**

Priority Interrupt:

- There are number of IO devices attached to the computer.
- They are all capable of generating the interrupt.
- When the interrupt is generated from more than one device, priority interrupt system is used to determine which device is to be serviced first.
- Devices with high speed transfer are given higher priority and slow devices are given lower priority.
- Establishing the priority can be done in two ways:

  Using Software
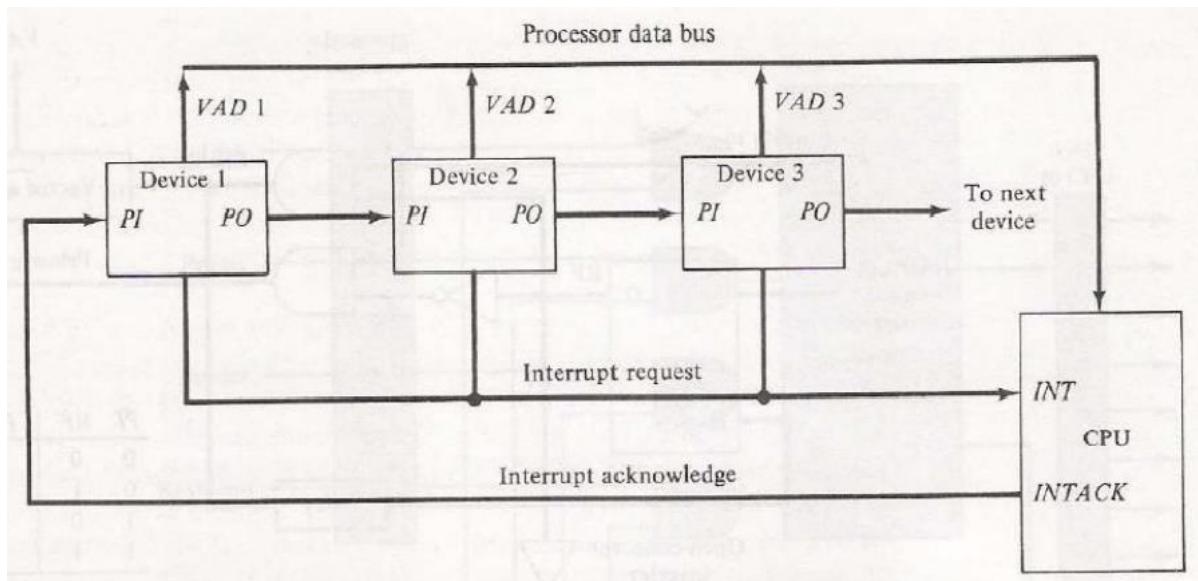
  Using Hardware
- A polling procedure is used to identify highest priority in software means.
- **Polling Procedure :**
- There is one common branch address for all interrupts.
- Branch address contain the code that polls the interrupt sources in sequence.
- The highest priority is tested first.
- The particular service routine of the highest priority device is served.
- The disadvantage is that time required to poll them can exceed the time to serve them in large number of IO devices.
- **Using Hardware:**
- Hardware priority system function as an overall manager

- It accepts interrupt request and determine the priorities.
- To speed up the operation each interrupting devices has its own interrupt vector.
- No polling is required, all decision are established by hardware priority interrupt unit.
- It can be established by serial or parallel connection of interrupt lines.
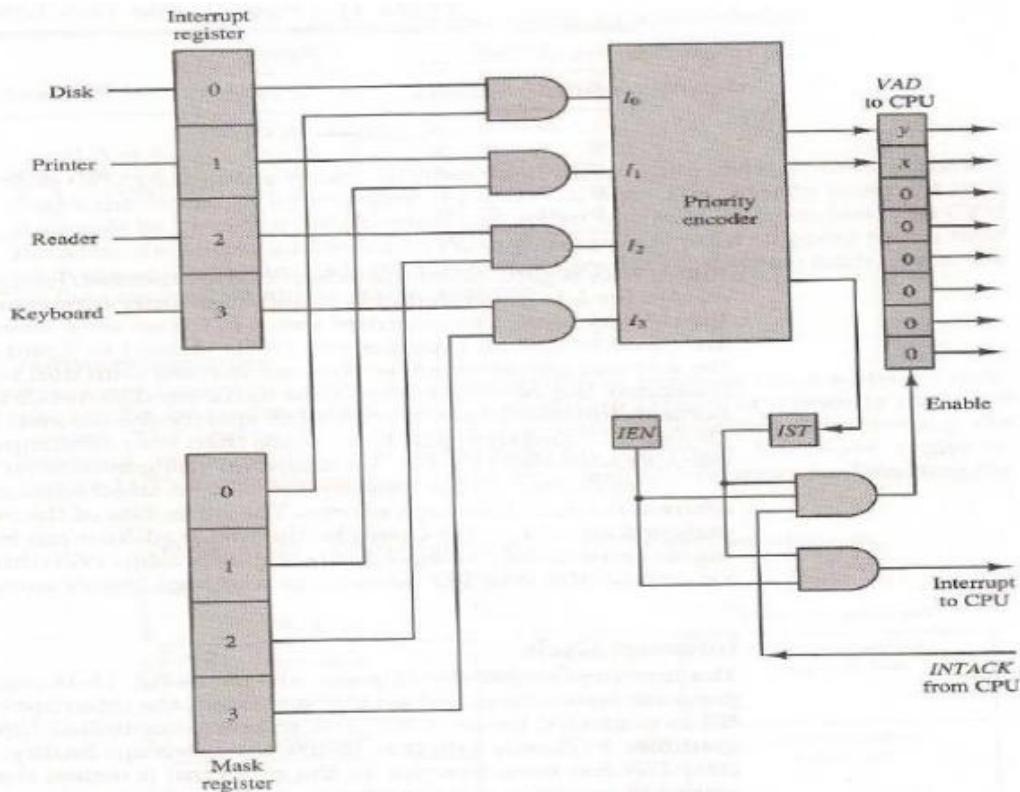
Serial or Daisy Chaining Priority:

- Device with highest priority is placed first.
- Device that wants the attention send the interrupt request to the CPU.
- CPU then sends the INTACK signal which is applied to PI(priority in) of the first device.
- If it had requested the attention, it place its VAD(vector address) on the bus. And it block the signal by placing 0 in PO(priority out)
- If not it pass the signal to next device through PO(priority out) by placing 1.
- This process is continued until appropriate device is found.
- The device whose PI is 1 and PO is 0 is the device that send the interrupt request

Parallel Priority Interrupt:

- It consists of interrupt register whose bits are set separately by the interrupting devices.
- **Priority is established according to the position of the bits in the register.**
- Mask register is used to provide facility for the higher priority devices to interrupt when lower priority device is being serviced or disable all lower priority devices when higher is being serviced.
- Corresponding interrupt bit and mask bit are ANDed and applied to priority encoder.
- **Priority encoder** generates two bits of vector address.
- Another output from it sets IST(interrupt status flip flop).

Priority Interrupt Hardware

Priority Encoder:

Determines the highest priority interrupt when more than one interrupts take place

If two or more inputs arrive at the same time, the input having the highest priority will take precedence.

Input I0 has the highest priority and I3 has the lowest Priority.

Priority Encoder Truth Table

Inputs				Outputs			Boolean functions
I_0	I_1	I_2	I_3	x	y	IST	
1	×	×	×	0	0	1	
0	1	×	×	0	1	1	$x = I_0' I_1'$
0	0	1	×	1	0	1	$y = I_0' I_1 + I_0' I_2'$
0	0	0	1	1	1	1	$(IST) = I_0 + I_1 + I_2 + I_3$
0	0	0	0	×	×	0	

Priority Encoder Truth Table

INTERRUPT CYCLE:

- At the end of each Instruction cycle
- CPU checks IEN and IST
- If IEN • IST = 1, CPU -> Interrupt Cycle

|	
SP ← SP - 1	Decrement stack pointer
M[SP] ← PC	Push PC into stack
INTACK ← 1	Enable interrupt acknowledge
PC ← VAD	Transfer vector address to PC
IEN ← 0	Disable further interrupts
Go To Fetch	to execute the first instruction in the interrupt service routine

Direct Memory Access (DMA):
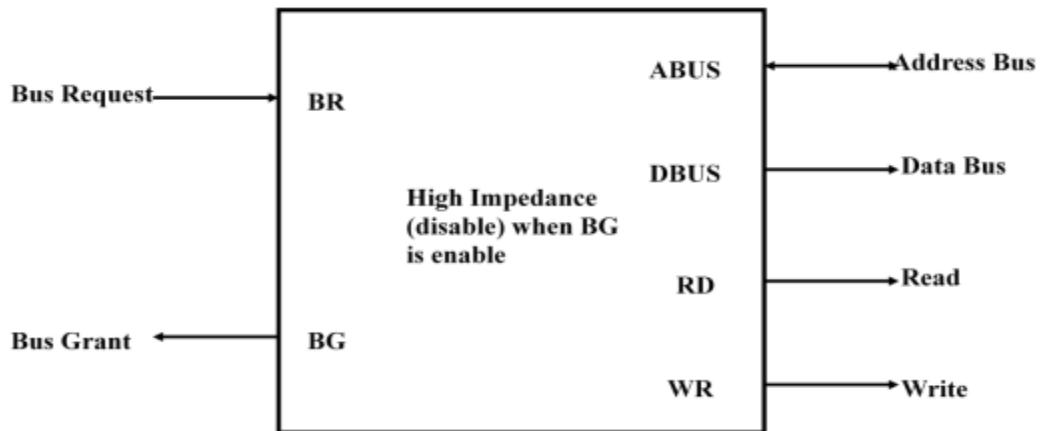
- In the Direct Memory Access (DMA) the interface transfer the **data into and out of the memory unit through the memory bus.**

- The transfer of data between a fast storage device such as magnetic disk and memory is often limited by the speed of the CPU.

- Removing the CPU from the path and letting the peripheral device manage the memory buses directly would improve the speed of transfer.

- This transfer technique is called Direct Memory Access (DMA).

- During the DMA transfer, the CPU is idle and has no control of the memory buses.

- A DMA Controller takes over the buses to manage the transfer directly between the I/O device and memory.

- The CPU may be placed in an idle state in a variety of ways. One common method extensively used in microprocessor is to disable the buses through special control signals such as:

  Bus Request (BR)

  Bus Grant (BG)

- These two control signals in the CPU that facilitates the DMA transfer.

- The Bus Request (BR) input is used by the DMA controller to request the CPU.

- When this input is active, the CPU terminates the execution of the current instruction and places the address bus, data bus and read write lines into a high Impedance state. High Impedance state means that the output is disconnected.

CPU Bus signals for DMA Transfer:

CPU bus Signals for DMA Transfer

- The CPU activates the Bus Grant (BG) output to inform the external DMA that the Bus Request (BR) can now take control of the buses to conduct memory transfer without processor.
- When the DMA terminates the transfer, it disables the Bus Request (BR) line.
- The CPU disables the Bus Grant (BG), takes control of the buses and return to its normal operation.
- The transfer can be made in several ways that are:
    - i. DMA Burst
    - ii. Cycle Stealing

DMA Burst :-

In DMA Burst transfer, a block sequence consisting of a number of memory words is transferred in continuous burst while the DMA controller is master of the memory buses.

Cycle Stealing:

- Cycle stealing allows the DMA controller to transfer one data word at a time, after which it must returns control of the buses to the CPU

DMA Controller:
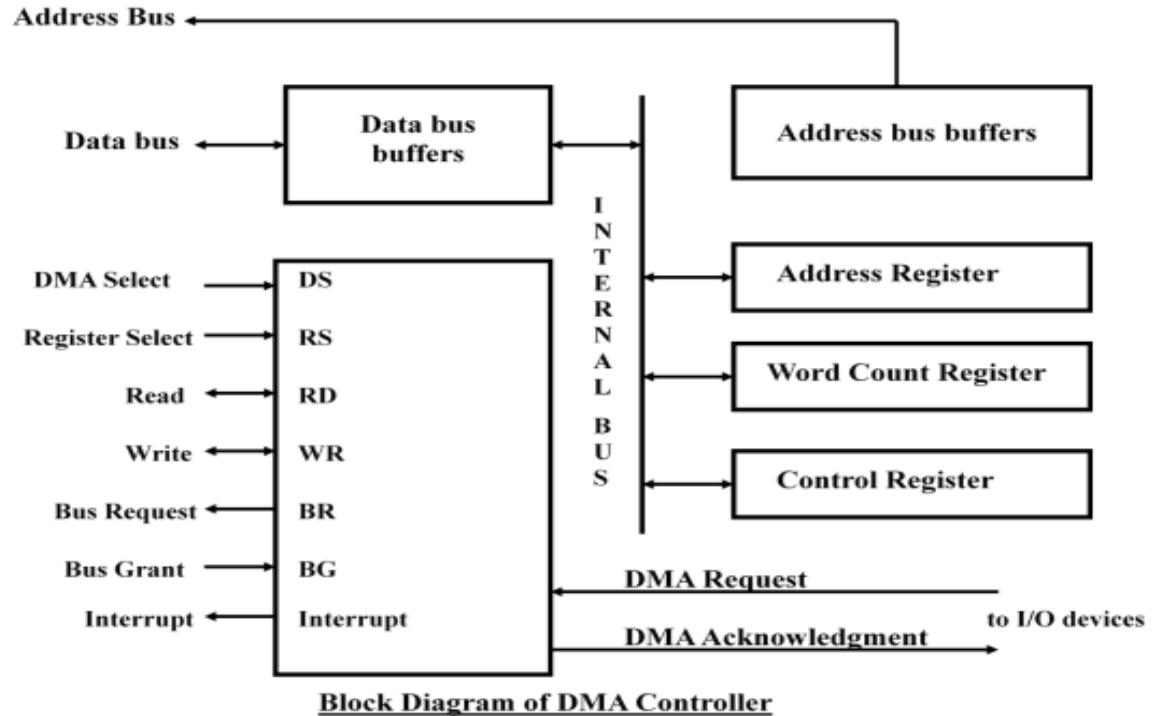
- The DMA controller needs the usual circuits of an interface to communicate with the CPU and I/O device. The DMA controller has three registers:

    i. Address Register

    ii. Word Count Register

    iii. Control Register

i. **Address Register** :- Address Register contains an address to specify the desired location in memory.

ii. **Word Count Register** :- WC holds the number of words to be transferred and internally tested for zero.

iii. **Control Register** :-

- Control Register specifies the mode of transfer

The unit communicates with the CPU via the data bus and control lines.
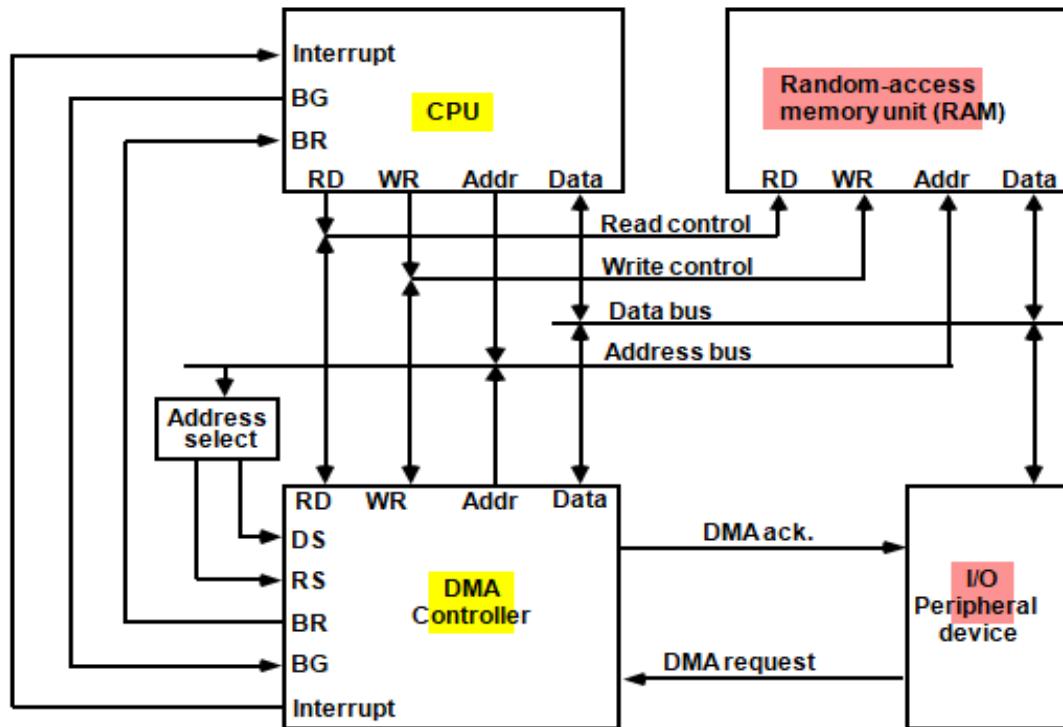
The registers in the DMA are selected by the CPU through the address bus by enabling the **DS (DMA select) and RS (Register select) inputs**.

- The RD (read) and WR (write) inputs are bidirectional.

- When the BG (Bus Grant) input is 0, the CPU can communicate with the DMA registers through the data bus to read from or write to the DMA registers.

- When BG =1, the DMA can communicate directly with the memory by specifying an address in the address bus and activating the RD or WR control.

Block Diagram of DMA Controller

**Block Diagram of DMA Controller**

DMA Transfer:

- The CPU communicates with the DMA through the address and data buses as with any interface unit.
- The DMA has its own address, which activates the DS and RS lines.
- The CPU initializes the DMA through the data bus. Once the DMA receives the start control command, it can transfer between the peripheral and the memory.
- When BG = 0 the RD and WR are input lines allowing the CPU to communicate with the internal DMA registers.
- When BG=1, the RD and WR are output lines from the DMA controller to the random access memory to specify the read or write operation of data.

Input-Output Processor:

- It is a processor with direct memory access capability that communicates with IO devices.
- IOP is similar to CPU except that it is designed to handle the details of IO operation.
- Unlike DMA which is initialized by CPU, IOP can fetch and execute its own instructions.
- IOP instruction are specially designed to handle IO operation.

Block Diagram of a computer with a I/O Processor:
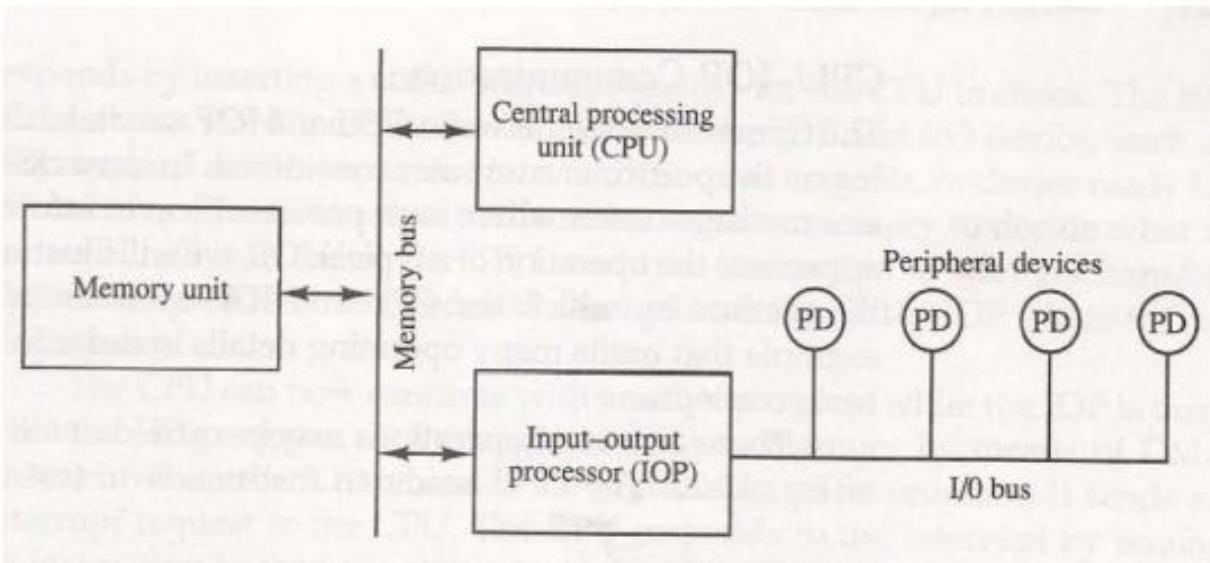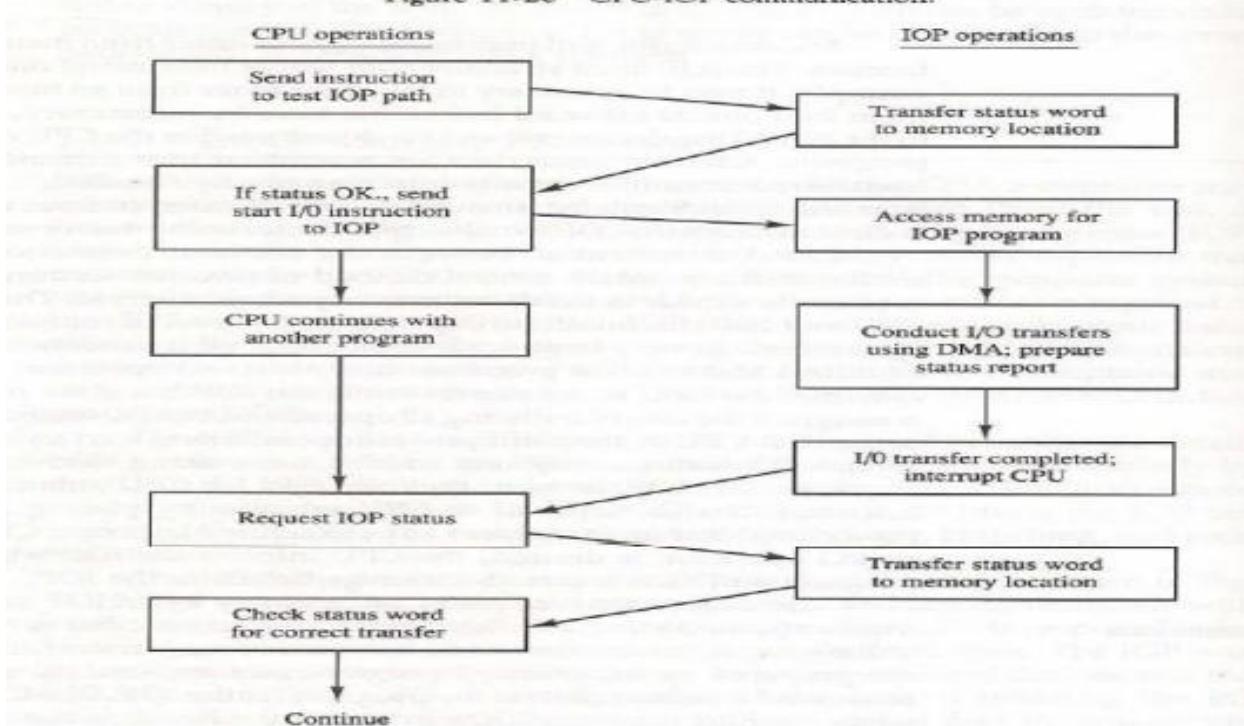
Figure   Block diagram of a computer with I/O processor.

- Memory occupies the central position and can communicate with each processor by DMA.
- CPU is responsible for processing data.
- IOP provides the path for transfer of data between various peripheral devices and memory.
- Data formats of peripherals differ from CPU and memory. IOP maintain such problems.
- Data is transferred from IOP to memory by stealing one memory cycle.
- Instructions that are read from memory by IOP are called commands to distinguish them from instructions that are read by the CPU.

CPU- IOP Communication:

Figure 11-20   CPU-IOP communication.

- Instruction that are read from memory by an IOP
- Distinguish from instructions that are read by the CPU.
- Commands are prepared by experienced programmers and are stored in memory
- Command word = IOP program

UNIT-V

Memory Organization:

Memory Hierarchy, Main Memory, Auxiliary memory, Associate Memory, Cache Memory.

Pipeline and Vector Processing:

Parallel Processing, Pipelining, Arithmetic Pipeline, Instruction Pipeline, RISC Pipeline, Vector Processing, Array Processors.

Memory Hierarchy:

# Introduction
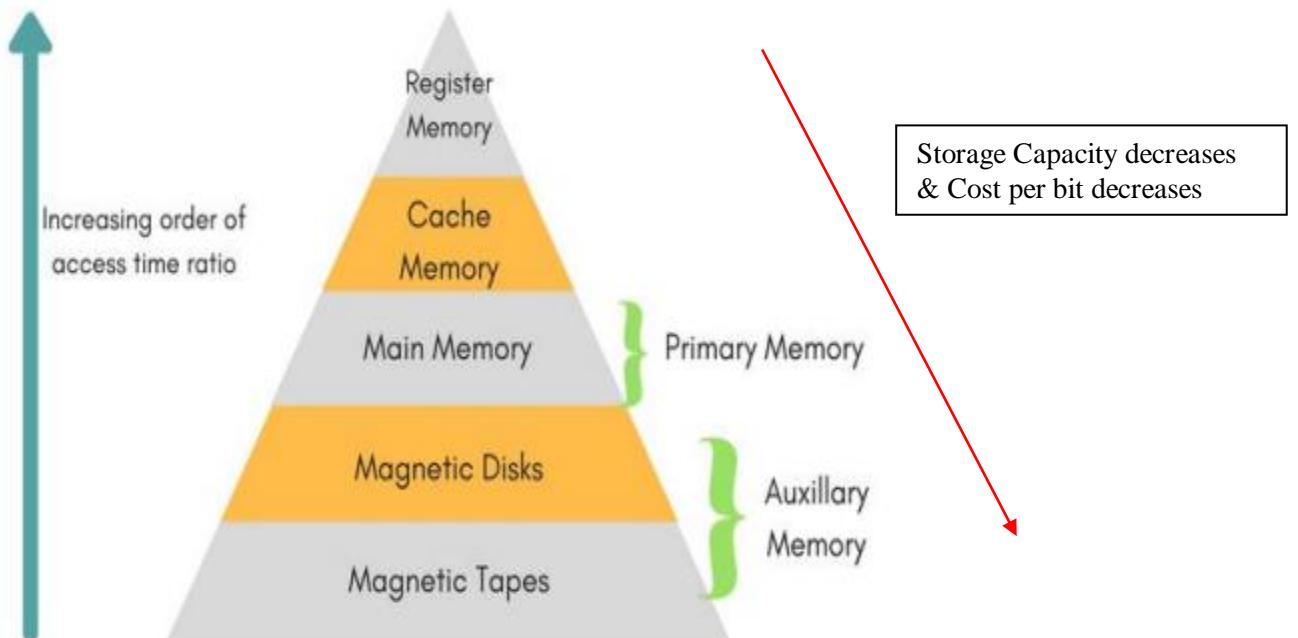
- The memory unit is an essential component needed for storing programs and data.

- Most general purpose computers run more efficiently if they are equipped with additional storage beyond the capacity of main memory.

- It is more economical to use low cost storage devices to serve as a backup for storing the information that is not currently used by the CPU.

- **Main Memory:** Memory unit that communicates directly with the CPU (RAM)

- **Auxiliary Memory**: Device that provide backup storage (Disk Drives)

Key characteristics of computer Memory system:
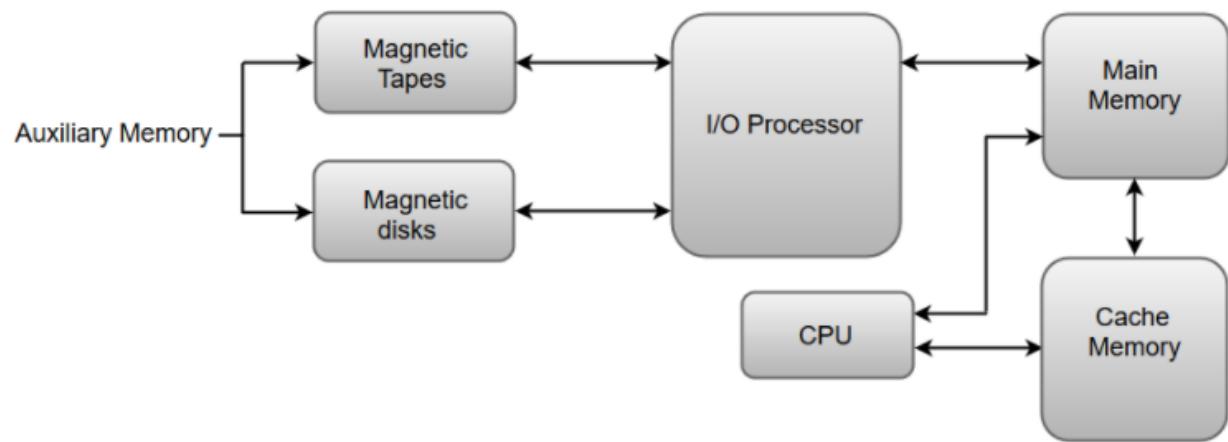
Location
    Internal (e.g. processor registers, main memory, cache)
    External (e.g. optical disks, magnetic disks, tapes)

Capacity
    Number of words
    Number of bytes

Unit of Transfer
    Word
    Block

Access Method
    Sequential
    Direct
    Random
    Associative

Performance
    Access time
    Cycle time
    Transfer rate

Physical Type
    Semiconductor
    Magnetic
    Optical
    Magneto-optical

Physical Characteristics
    Volatile/nonvolatile
    Erasable/nonerasable

Organization
    Memory modules

Memory Hierarchy in a computer system:

- The total memory capacity of a computer can be visualized as being a hierarchy of components.
- Only programs and data currently needed by the processor reside in main memory.
- All other information is stored in Auxiliary memory and transferred to main memory when needed.
- Memory hierarchy system consist of all storage devices from auxiliary memory to main memory to cache memory
- As one goes down the hierarchy :
- **Cost per bit decreases.**
- **Capacity increases.**
- **Access time increases.**
- **Frequency of access by the processor decreases**

Increasing order of access time ratio

Register Memory

Cache Memory

Main Memory } Primary Memory

Magnetic Disks

Magnetic Tapes } Auxillary Memory

Storage Capacity decreases & Cost per bit decreases

Memory Hierarchy in a Computer System:

- Figure illustrates the components in a typical memory hierarchy.
- At the bottom of the hierarchy are the relatively slow **magnetic tapes** used to store removable files.
- Next are the **Magnetic disks** used as **backup storage**.
- **The Main memory occupies a central position by being able to communicate directly with CPU and with auxiliary memory devices through an I/O process**
- The I/O processor manages data transfer between auxiliary memory and main memory.
- Program not currently needed in main memory are transferred into auxiliary memory to provide space for currently used programs and data.
- The **cache memory** is used for **storing segments of programs currently being executed in the CPU.**
- The auxiliary memory has a large storage capacity is relatively inexpensive, but has low access speed compared to main memory.
- The cache memory is very small, relatively expensive, and has very high access speed.

- The CPU has direct access to both cache and main memory but not to auxiliary memory.
- **Multiprogramming**
- Many operating systems are designed to enable the CPU to process a number of independent programs concurrently.
- Multiprogramming refers to the existence of 2 or more programs in different parts of the memory hierarchy at the same time.
- **Memory management System:**
- The part of the computer system that supervises the flow of information between auxiliary memory and main memory.

MAIN MEMORY:
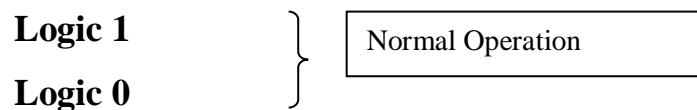
- Main memory is the central storage unit in a computer system.
- It is a relatively large and fast memory used to store programs and data during the computer operation.
- The principal technology used for the main memory is based on **semi conductor integrated circuits.**
- Integrated circuits RAM chips are available in two possible operating modes, static and dynamic.
- **Static RAM** – Consists of internal flip flops that store the binary information.
- **Dynamic RAM** – Stores the binary information in the form of electric charges that are applied to capacitors.
- Most of the main memory in a general purpose computer is made up of RAM integrated circuit chips, but a portion of the memory may be constructed with ROM chips.
- **Read Only Memory** –Store programs that are permanently resident in the

computer and for tables of constants that do not change in value once the production of the computer is completed.

- The ROM portion of main memory is needed for storing an initial program called a Bootstrap loader.
- **Boot strap loader** –function is start the computer software operating when power is turned on.
- Boot strap program loads a portion of operating system from disc to main memory and control is then transferred to operating system.
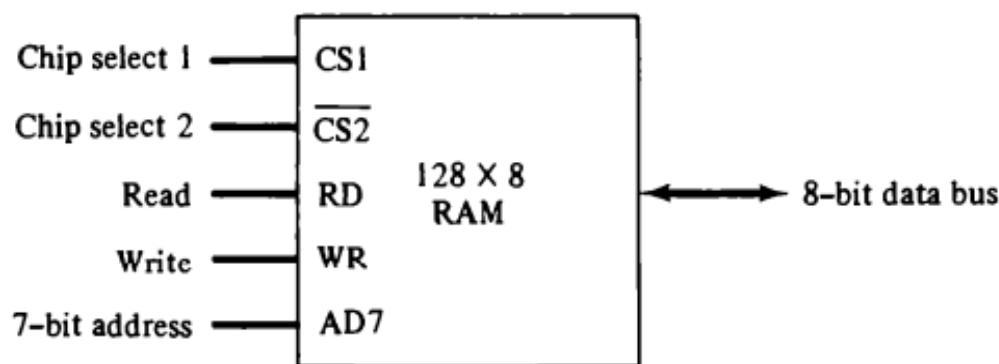
RAM and ROM CHIP:

- A RAM chip is better suited for communication with CPU if it has one or more control inputs that select the chip only when needed.
- RAM chip –utilizes bidirectional data bus with three state buffers to perform communication with CPU
- Three state buffers consists of

Logic 1
Logic 0   } Normal Operation

High impedance state -open circuit & has no logic significance

Figure 12-2 Typical RAM chip.

Chip select 1 —— CS1
Chip select 2 —— $\overline{CS2}$
Read —— RD   128 X 8
                        RAM   ←——→ 8–bit data bus
Write —— WR
7–bit address —— AD7

(a) Block diagram

- The block diagram of a RAM Chip is shown in Fig.
- The capacity of memory is 128 words of eight bits (one byte) per word.
- This requires a 7-bit address and an 8-bit bidirectional data bus.
- The read and write inputs specify the memory operation and the two chips select (CS) control inputs are enabling the chip only when it is selected by the microprocessor.
- The read and write inputs are sometimes combined into one line labeled R/W.
- The function table listed in Fig. specifies the operation of the RAM chip.
- **The unit is in operation only when CS1=1 and CS2=0.**
- The bar on top of the second select variable indicates that this input is enabled when it is equal to 0.
- If the chip select inputs are not enabled, or if they are enabled but the read or write inputs are not enabled, the memory is inhibited and its data bus is in a high-impedance state.

CS1	$\overline{CS2}$	RD	WR	Memory function	State of data bus
0	0	×	×	Inhibit	High-impedance
0	1	×	×	Inhibit	High-impedance
1	0	0	0	Inhibit	High-impedance
1	0	0	1	Write	Input data to RAM
1	0	1	×	Read	Output data from RAM
1	1	×	×	Inhibit	High-impedance

Chip is enabled when CS1=1; CS2=0

(b) Function table

- When CS1=1 and CS2=0, the memory can be placed in a write or read mode.
- When the WR input is enabled, the memory stores a byte from the data bus into a location specified by the address input lines.
- When the RD input is enabled, the content of the selected byte is placed into the data bus.
- The RD and WR signals control the memory operation as well as the bus buffers associated with the bidirectional data bus.

ROM Chip:

- A ROM chip is organized externally in a similar manner. However, since a **ROM can only read**, the data bus can only be in an output mode.
- The block diagram of a ROM chip is shown in fig.
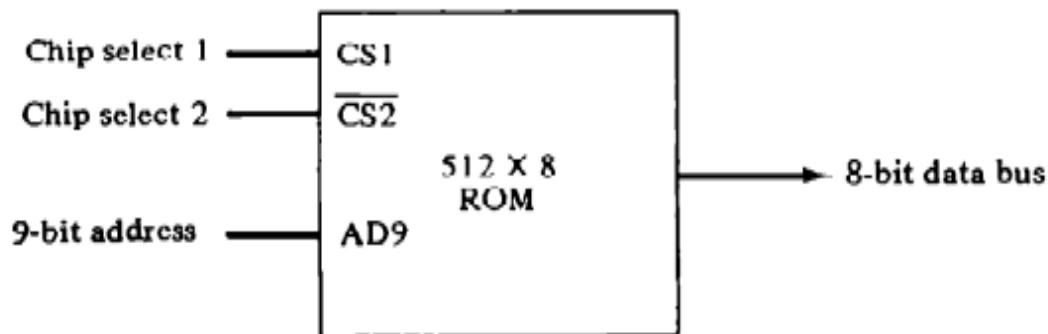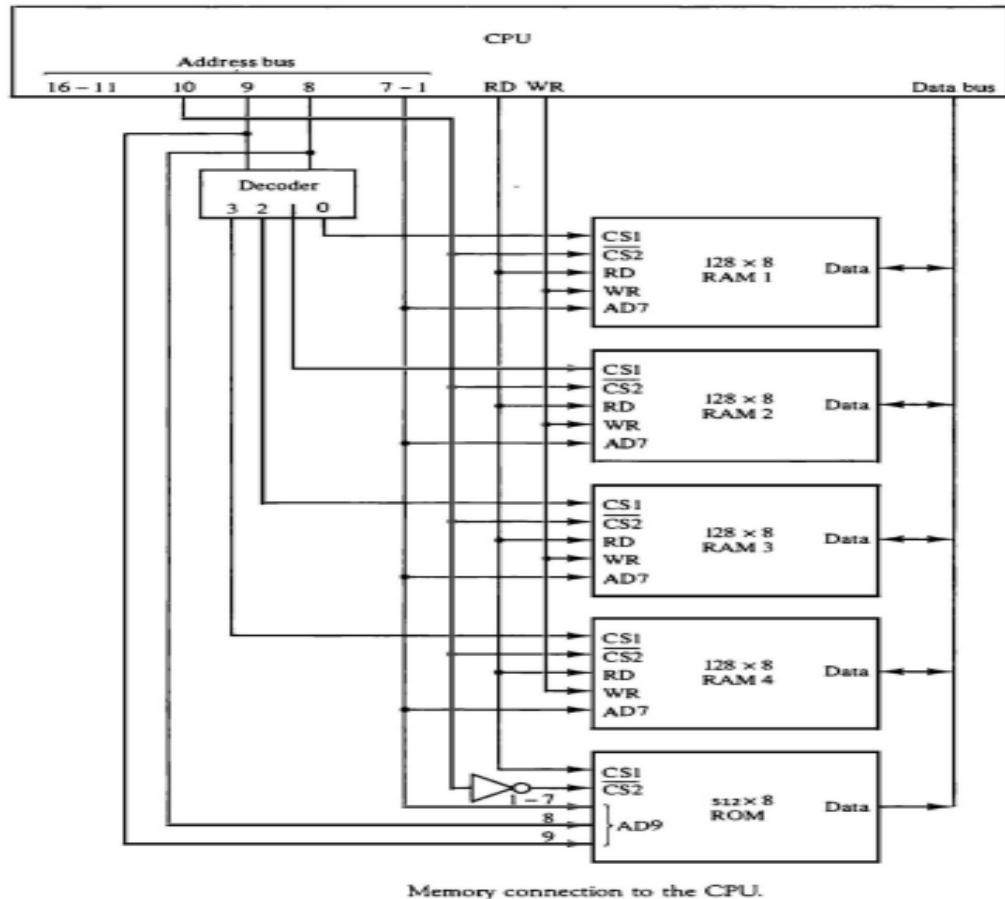
Figure 12-3 Typical ROM chip.

- The **nine address lines in the ROM chip** specify any one of the **512 bytes** stored in it.
- The two chip select inputs must be CS1=1 and CS2=0 for the unit to operate. Otherwise, the data bus is in a high-impedance state.

Memory Address Map:

- The interconnection between **memory and processor** is then established from knowledge of the size of memory needed and the type of RAM and ROM chips available.
- The addressing of memory can be established by means of a table that specifies the memory address assigned to each chip.
- The table called Memory address map, is a **pictorial representation of assigned address space for each chip in the system.**
- The memory address map for this configuration is shown in table.
- The component column specifies whether a RAM or a ROM chip is used.
- The hexadecimal address column assigns a range of hexadecimal equivalent addresses for each chip.
- The address bus lines are listed in the third column.
- The RAM chips have 128 bytes and need seven address lines.
- The ROM chip has 512 bytes and needs 9 address lines.

TABLE 12-1 Memory Address Map for Microprocomputer

Component	Hexadecimal address	Address bus										
		10	9	8	7	6	5	4	3	2	1
RAM 1	0000–007F	0	0	0	x	x	x	x	x	x	x
RAM 2	0080–00FF	0	0	1	x	x	x	x	x	x	x
RAM 3	0100–017F	0	1	0	x	x	x	x	x	x	x
RAM 4	0180–01FF	0	1	1	x	x	x	x	x	x	x
ROM	0200–03FF	1	x	x	x	x	x	x	x	x	x

Memory connection to the CPU.

- RAM and ROM chips are connected to a CPU through the data and address buses.

- The low order lines in the address bus select the byte within the chips and other lines in the address bus select a particular chip through its chip select inputs.

- The connection of memory chips to the CPU is shown in Fig.

- This configuration gives a memory capacity of 512 bytes of RAM and 512 bytes of ROM.

- Each RAM receives the seven low-order bits of the address bus to select one of 128 possible bytes.

- **The particular RAM chip selected is determined** from lines 8 and 9 in the address bus.

- This is done through a **2 X 4 decoder** whose outputs go to the CS1 inputs in each RAM chip.

- Thus, when address lines 8 and 9 are equal to 00, the first RAM chip is selected.

- When 01, the second RAM chip is select, and so on.

- The RD and WR outputs from the microprocessor are applied to the inputs of each RAM chip.

- The selection between RAM and ROM is achieved through **bus line 10**.

- The RAMs are selected when the bit in this line is 0 and the ROM when the bit is 1.

- Address bus lines 1 to 9 are applied to the input address of ROM without going through the decoder.

   The data bus of the ROM has only an output capability, whereas the data bus connected to the RAMs can transfer information in both directions.

AUXILIARY MEMORY:
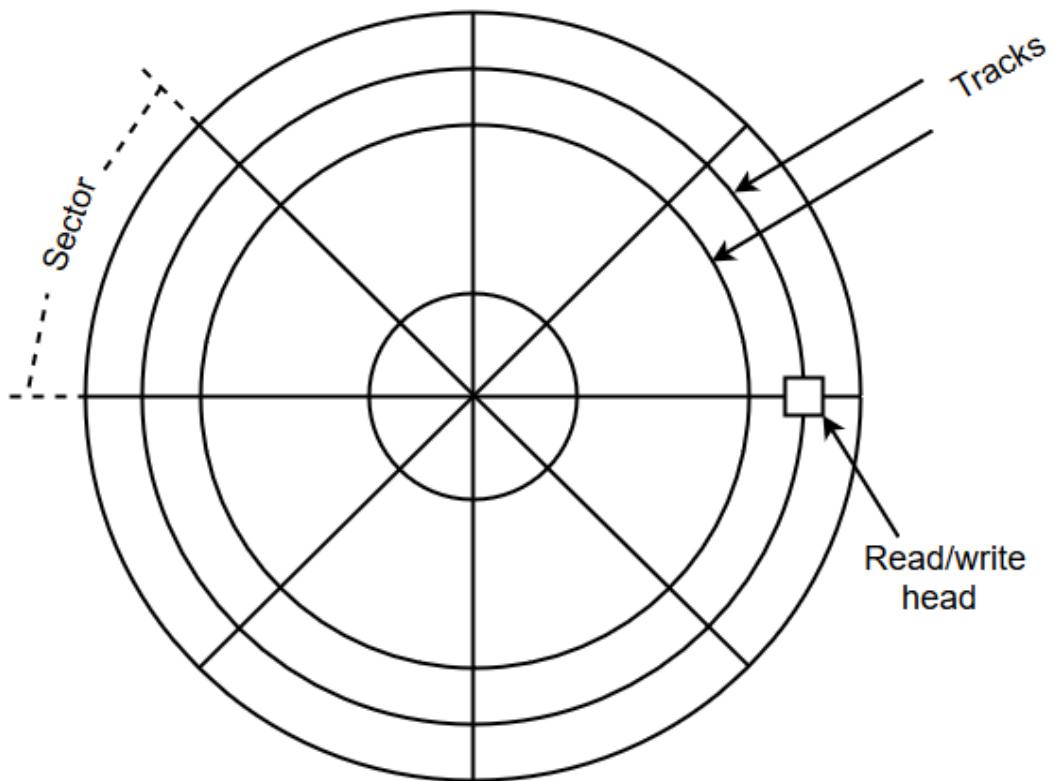
- Devices that provide **backup storage** are called auxiliary memory.

- For example: Magnetic disks and tapes are commonly used auxiliary devices.

- Other devices used as auxiliary memory are magnetic drums, magnetic bubble memory and optical disks.

- It is not directly accessible to the CPU, and is accessed using the Input/Output channels.

- An Auxiliary memory is known as the lowest-cost, highest-capacity and slowest-access storage in a computer system.It is where programs and data are kept for long-term storage or when not in immediate use.

Magnetic Disks
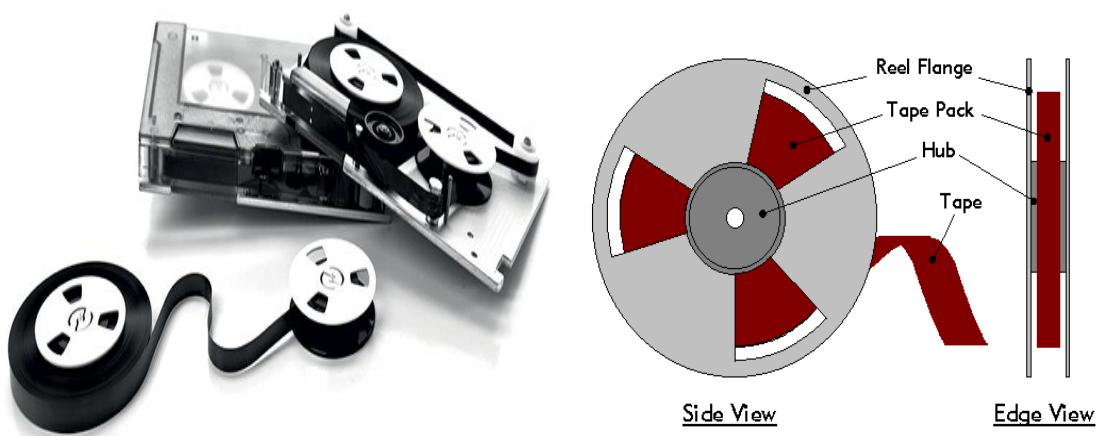
- A magnetic disk is a type of memory constructed using a **circular plate of metal or plastic coated with magnetized materials.**

- Usually, both sides of the disks are used to carry out read/write operations.

- However, several disks may be stacked on one spindle with read/write head available on each surface.

- The following image shows the structural representation for a magnetic disk.

- The **memory bits are stored in the magnetized surface in spots** along the concentric circles called **tracks.**

- The concentric circles (tracks) are commonly divided into sections called **sectors.**

Magnetic disks

Magnetic Tape

Side View   Edge View

- **Magnetic tape** is a storage medium that allows **data archiving, collection, and backup for different kinds of data.**
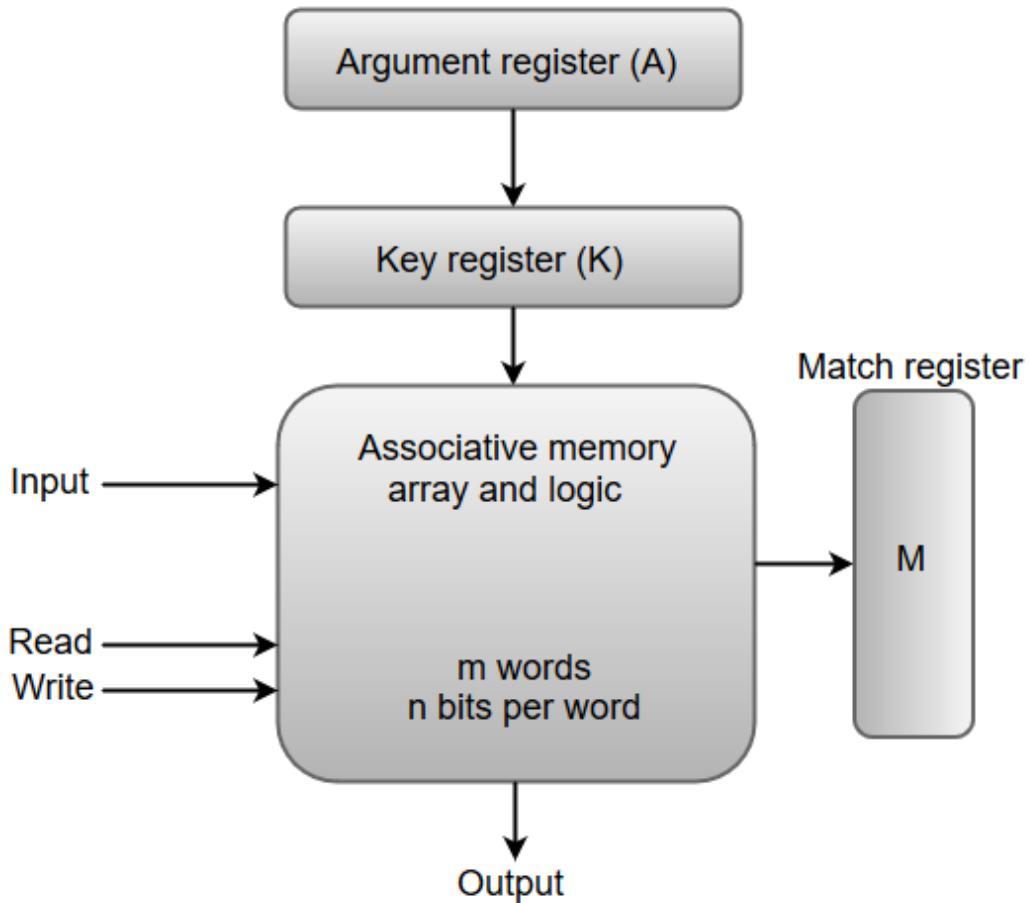
- The magnetic tape is constructed using a plastic strip coated with a magnetic recording medium.

- The bits are recorded as magnetic spots on the tape along several tracks.

- Usually, seven or nine bits are recorded simultaneously to form a character together with a parity bit.

- Magnetic tape units can be halted, started to move forward or in reverse, or can be rewound.

- However, they cannot be started or stopped fast enough between individual characters.

For this reason, information is recorded in blocks referred to as records.

Associative Memory:

- The time required to find an item stored in memory can be reduced considerably if stored data can be identified by the **content of the data** itself rather than by an address.

- A memory unit accessed by content is called an associative memory or **content addressable memory (CAM).**

- CAM is accessed simultaneously and in parallel on the basis of data content rather than by specific address or location

- Associative memory is more expensive than a RAM because each cell must have storage capability as well as logic circuits for matching its contents with external argument.

- **Argument register** –holds an external argument for content matching

- **Key register** –mask for choosing a particular field or key in the argument word
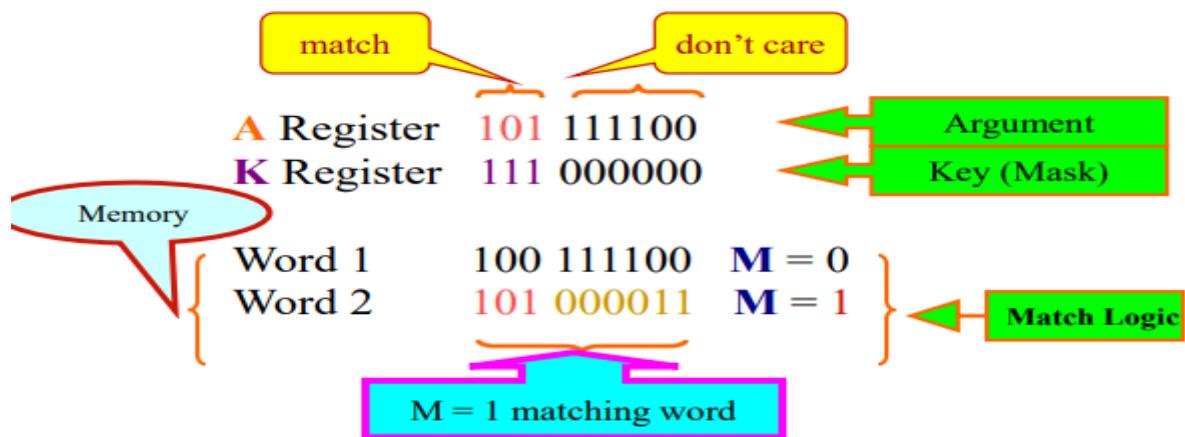
Hardware Organization
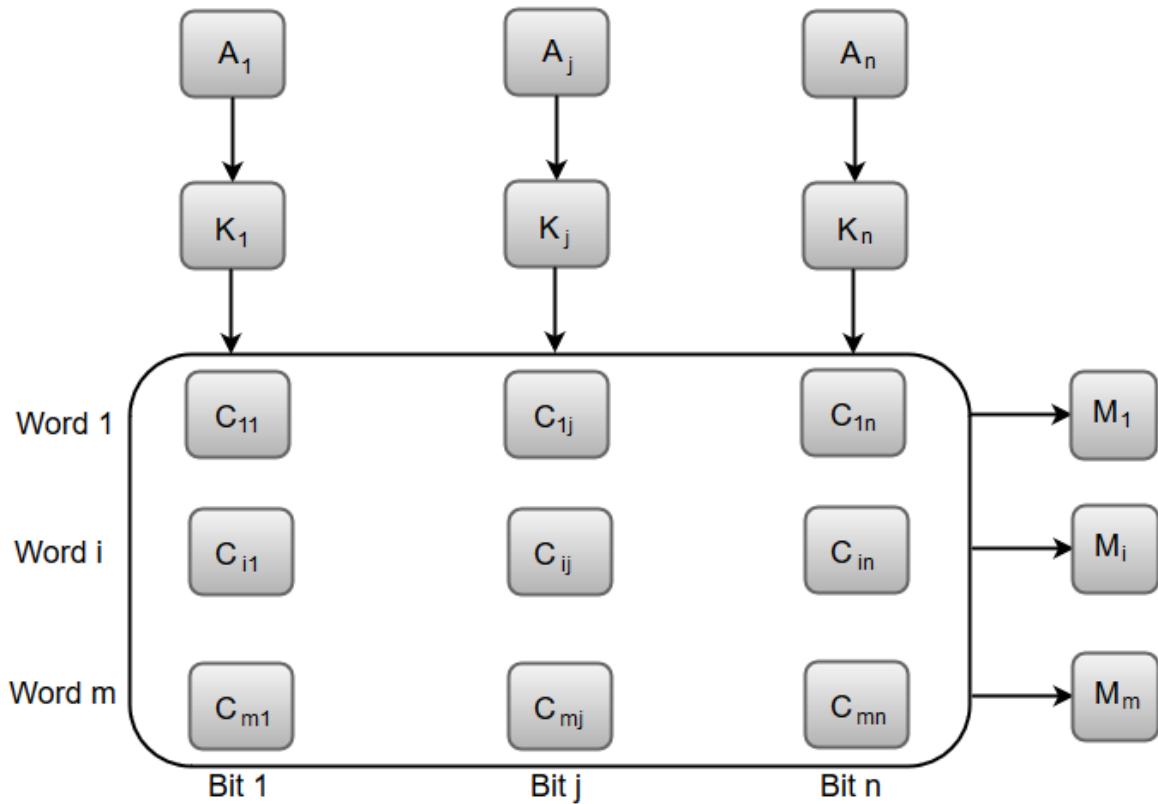
- It consists of a memory array and logic for m words with n bits per word.
- The argument register **A and key register K each have n bits**, one for each bit of a word.
- The match register M has m bits, one for each memory word.
- Each word in memory is compared in parallel with the content of the 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.

Example of Match logic:
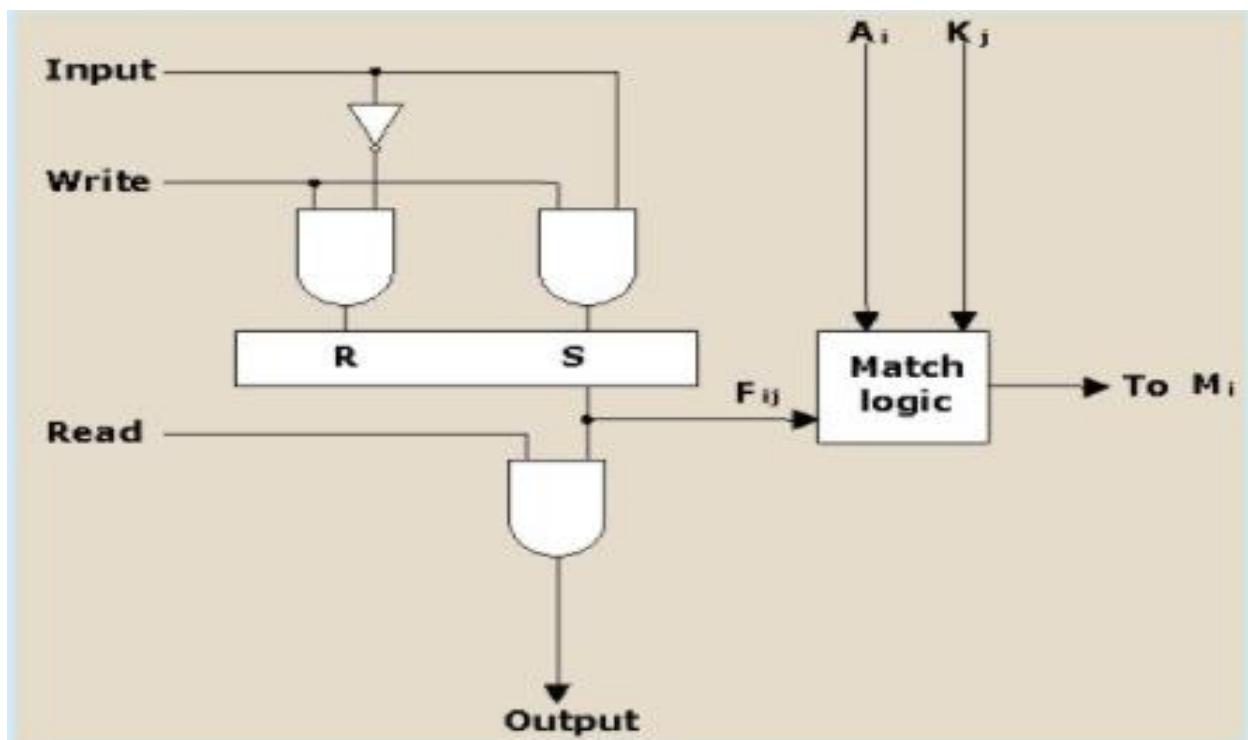
Associative Memory of m words with n cells per word.

- The relation between the memory array and external registers in an associative memory is shown in Fig. Above.
- The cells in the array are marked by the letter C with two subscripts.
- The **first subscript** gives the **word number** and **second** specifies the **bit position** in the word.
- Thus cell C_{ij} is the cell for bit j in word i.
- A bit A_j in the argument register is compared with all the bits in column j of the array provided that $k_j = 1$.
- This is done for all columns j=1,2,….n.
- If a match occurs between all the unmasked bits of the argument and the bits in word I, the corresponding bit M_i in the match register is set to 1.
- If one or more unmasked bits of the argument and the word do not match, M_i is cleared to 0.

One cell of Associative memory:

- It consists of **flip-flop storage element Fij** and the circuits for reading, writing, and matching the cell.
- The input bit is transferred into the storage cell during a write operation.
- The bit stored is read out during a read operation.
- The match logic compares the content of the storage cell with corresponding unmasked bit of the argument and provides an output for the decision logic that sets the bit in Mi.

Match Logic:

- The match logic for each word can be derived from the comparison algorithm for two binary numbers.
- First, neglect the key bits and compare the argument in A with the bits stored in the cells of the words.
- Word i is equal to the argument in A if

    Aj=F ij   for j=1,2,.....,n.

- Two bits are equal if they are both 1 or both 0.
- The equality of two bits can be expressed logically by the Boolean function

    xj=Aj Fij + Aj 'Fij '

    where xj = 1 if the pair of bits in position j are equal;

    otherwise , xj =0.

- For a word i is equal to the argument in A we must have all xj variables equal to 1.
- This is the condition for setting the corresponding match bit Mi to 1.
- The Boolean function for this condition is

    Mi = x1 x2 x3…… xn; j=1 to n

Include the key bit Kj in the comparison logic. The requirement is that if Kj = 0,the corresponding bits of Aj and Fij need no comparison. Only when Kj = 1 must they be compared. This requirement is achieved by ORing each term with K'j , thus:

$$x_j + K'_j = \begin{cases} x_j & \text{if } K_j = 1 \\ 1 & \text{if } K_j = 0 \end{cases}$$

When Kj = 1, we have Kj' = 0 and xj + 0 = xj. When Kj = 0, then Kj' = 1 xj + 1 = 1. A term (xj +Kj') will be in the 1 state if its pair of bitsis not compared. This is necessary because each term is ANDed with all other terms so that an output of 1 will have no effect. The comparison of the bits has an effect only when Kj = 1.

The match logic for word i in an associative memory can now be expressed by the following Boolean function:

$$M_i = (x_1 + K'_1)(x_2 + K'_2)(x_3 + K'_3) \dots (x_n + K'_n)$$

Each term in the expression will be equal to 1 if its corresponding Kj = 0. If Kj = 1, the term will be either 0 or 1 depending on the value of xj. A match will occur and Mi will be equal to 1 if all terms are equal to 1.

If we substitute the original definition of xj. the Boolean function above can be expressed as follows:

$$M_i = \prod_{j=1}^{n}(A_j F_{ij} + A'_j F'_{ij} + K'_j)$$

Match logic for one word of Associative Memory:
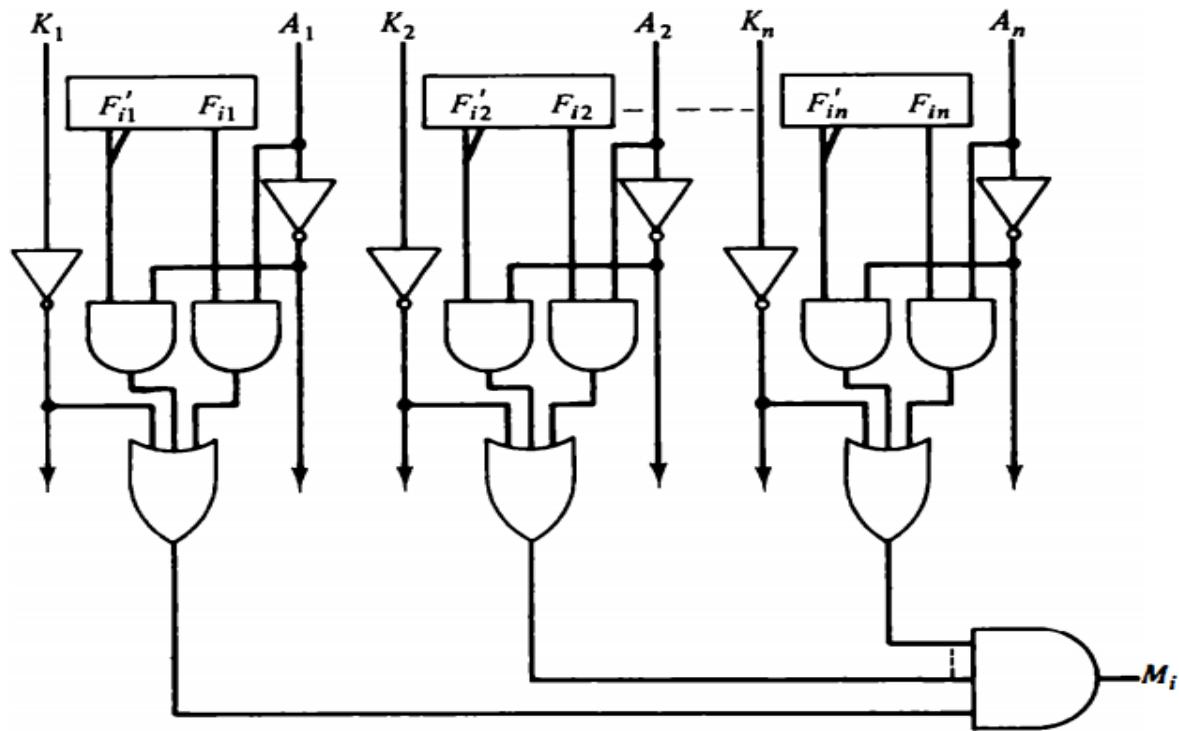
Figure ·9 Match logic for one word of associative memory.

- Each cell requires two AND gate and one OR gate. The inverters for A and K are needed once for each column and are used for all bits in the column. The output of all OR gates in the cells of the same word go to the input of a common AND gate to generate the match signal for Mi . Mi will be logic 1 if a match occurs and 0 if no match occurs.

Read Operation:

- If more than one word in memory matches the unmasked argument field , all the matched words will have 1's in the corresponding bit position of the match register

- **In read operation all matched words are read in sequence by applying a read signal to each word line whose corresponding Mi bit is a logic 1**

- In applications where no two identical items are stored in the memory , only one word may match , in which case we can use Mi output directly as a read
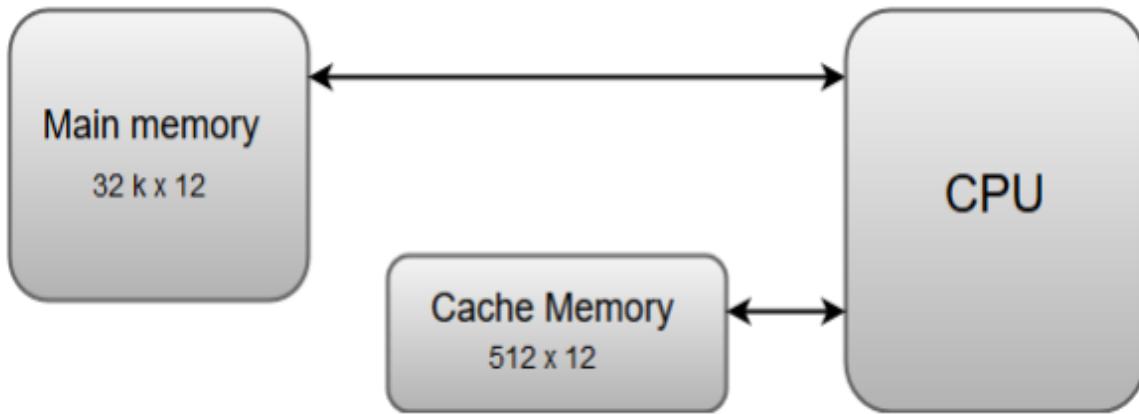
signal for the corresponding word

Write Operation

- It has a storing capability for the information to be searched
- Can take two different forms

1. Entire memory may be loaded with new information once prior to search operation then the writing can be done by addressing each location in sequence

This makes it random access memory for writing and content addressable memory for reading

2. Unwanted words to be deleted and new words to be inserted by using a tag register.

Cache Memory:

- The data or contents of the main memory that are used frequently by CPU are stored in the cache memory so that the processor can easily access that data in a shorter time.
- Whenever the CPU needs to access memory, it first checks the cache memory.
- If the data is not found in cache memory, then the CPU moves into the main memory.
- Cache memory is placed between the CPU and the main memory.
- The block diagram for a cache memory can be represented as:

- The basic operation of a cache memory is as follows:
- When the CPU needs to access memory, the cache is examined.
- If the word is found in the cache, it is read from the fast memory.
- 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 one just accessed is then transferred from main memory to cache memory.
- The block size may vary from one word (the one just accessed) to about 16 words adjacent to the one just accessed.
- The **performance of the cache memory** is frequently measured in terms of a quantity called **hit ratio**
- When the CPU refers to memory and finds the word in cache, it is said to produce a hit.
- If the word is not found in the cache, it is in main memory and it counts as a miss.
- The ratio of the number of hits divided by the total CPU references to memory (hits plus misses) is the hit ratio.

    Hit ratio= hits/ (hits + miss)

- Effectiveness of cache mechanism is based on a property of computer programs called "locality of reference"

Locality of Reference:

- Many instructions in localized areas of program are executed repeatedly during some time period. This property is called "Locality of Reference".

Principles of cache

- The **main memory** can store **32k words of 12 bits each**.
- The **cache** is capable of storing **512** of these words at any given time.
- The CPU communicates with both memories.
- It first sends a 15 bit address to cache. If there is a hit, the CPU accepts the 12 bit data from cache.
- If there is a miss, the CPU reads the word from main memory and the word is then transferred to cache.
- Assume cache is full and memory word not in cache is referenced
- Control hardware decides which block from cache is to be removed to create space for new block containing referenced word from memory
- Collection of rules for making this decision is called "**Replacement algorithm**"
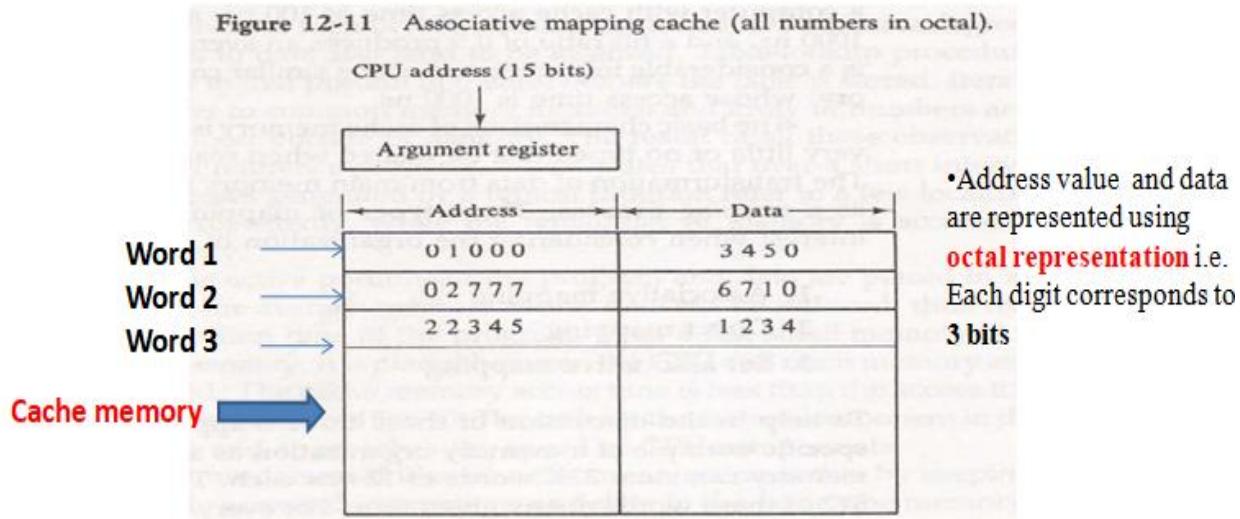
Mapping Functions

- The transformation of data from main memory to cache memory is referred as Mapping process.
- Correspondence between main memory blocks and those in the cache is specified by a memory mapping function
- There are three techniques in memory mapping

  1. Associative Mapping

  2. Direct Mapping

Associative Mapping

In this mapping function, any block of Main memory can potentially reside in any cache block position. This is much more flexible mapping method

Figure 12-11   Associative mapping cache (all numbers in octal).

- In fig 12-11, The associative memory stores both address and content(data) of the memory word.
- This permits any location in cache to store any word from main memory.
- The diagram shows three words presently stored in the cache.
- The address value of 15 bits is shown as a five-digit octal number and its corresponding 12-bit word is shown as a four-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 address is found, the corresponding 12-bit data is read and sent to the CPU. If no match occurs, the main memory is accessed for the word.
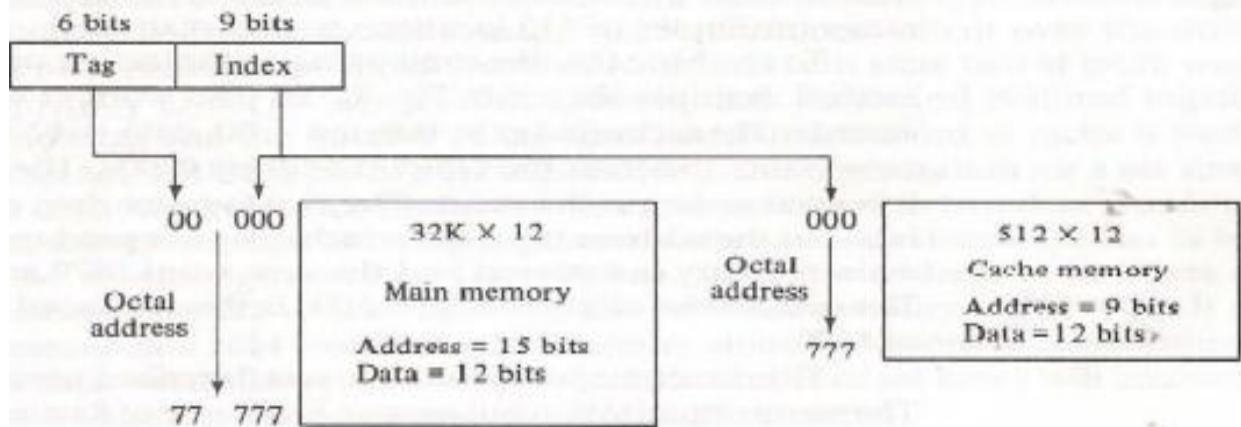
Direct Mapping:

- Associative memories are expensive compared to random access memories
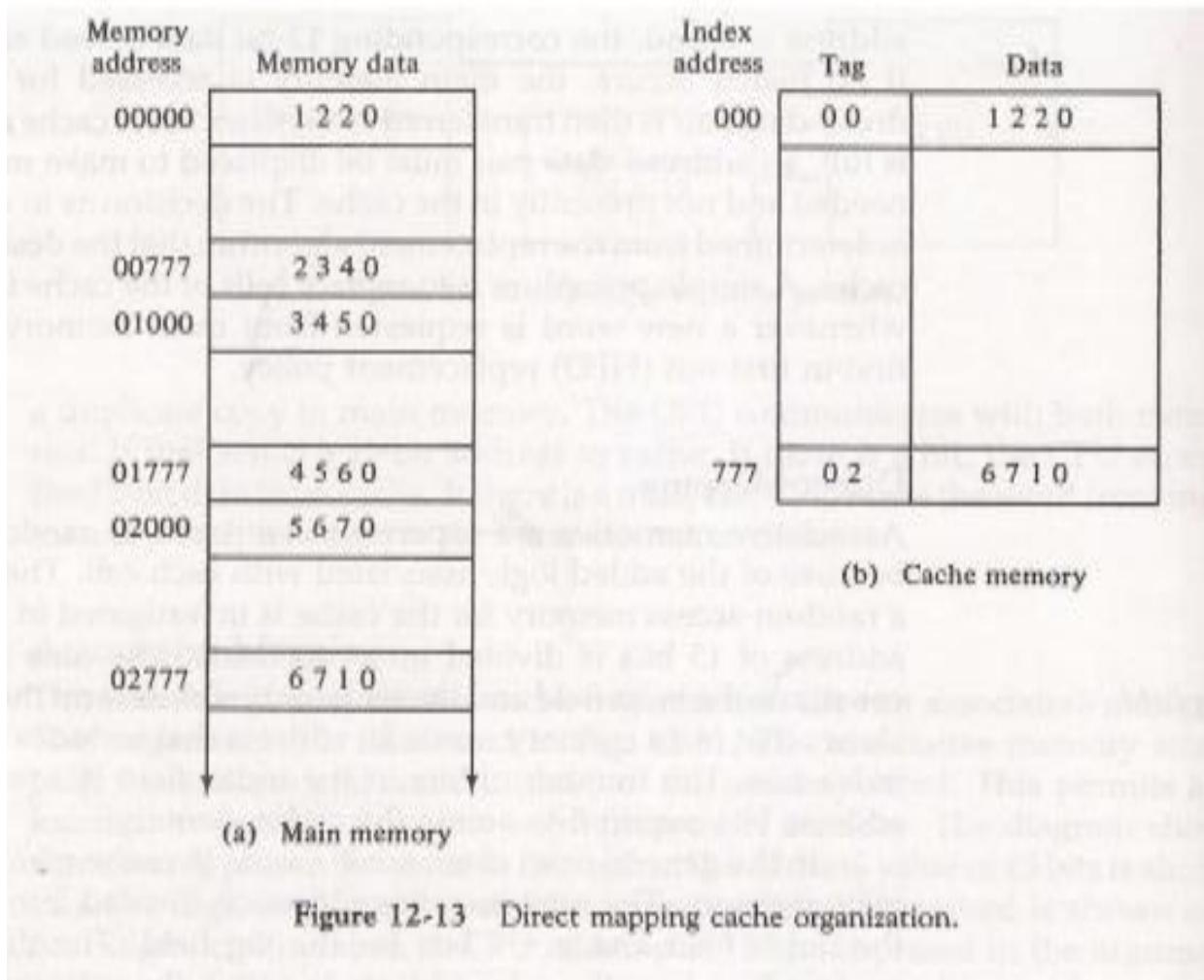
because of the added logic associated with each cell.

- A particular block of main memory can be brought to a particular block of cache memory. So, it is not flexible.

Figure 12-12 Addressing relationships between main and cache memories.

- In fig 12-12. The CPU address of 15 bits is divided into two fields.
- The **nine least significant bits constitute the index field** and remaining **six bits from the tag field.**
- The main memory needs an address that includes both the tag and the index bits.
- The **number of bits in the index field** is equal to the **number of address bits required to access the cache memory.**
- The direct mapping cache organization uses the n- bit address to access the main memory and the k-bit index to access the cache.

Figure 12-13   Direct mapping cache organization.

- Each word in **cache consists of the data word and 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 the index field is used for the address to access the 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 main memory.

- ## **Set Associative Mappin**

- In this method, blocks of cache are grouped into sets, and the mapping allows a block of main memory to reside in any block of a specific set.

- From the flexibility point of view, it is in between to the other two methods.

Index	Tag	Data	Tag	Data
000	0 1	3 4 5 0	0 2	5 6 7 0
777	0 2	6 7 1 0	0 0	2 3 4 0

Figure 12-15 Two-way set-associative mapping cache.

- The octal numbers listed in Fig.12-15 are with reference to the main memory contents.

- When the CPU generates a memory request, the index values of the address are used to access the cache.

- **The tag field of the CPU address is then compared with both tags in the cache** to determine if a match occurs.

- The comparison logic done by an associative search of the tags in the set similar to an associative memory search thus the name "Set Associative".

Pipelining and Vector Processing

Parallel Processing:

- Parallel processing is a term used for a large class of techniques that are used to provide **simultaneous data-processing tasks** for the purpose of increasing the computational speed of a computer system.
- It refers to techniques that are used to provide simultaneous data processing.
- The system may have two or more ALUs to be able to execute two or more instruction at the same time.
- The system may have two or more processors operating concurrently.
- It can be achieved **by having multiple functional units that perform same or different operation simultaneously.**
- **Parallel processing is done by distributing the data among multiple functional Units.**

Processor with Multiple function units:

The following figure shows one possible way of separating the execution unit into 8 functional units operating in parallel
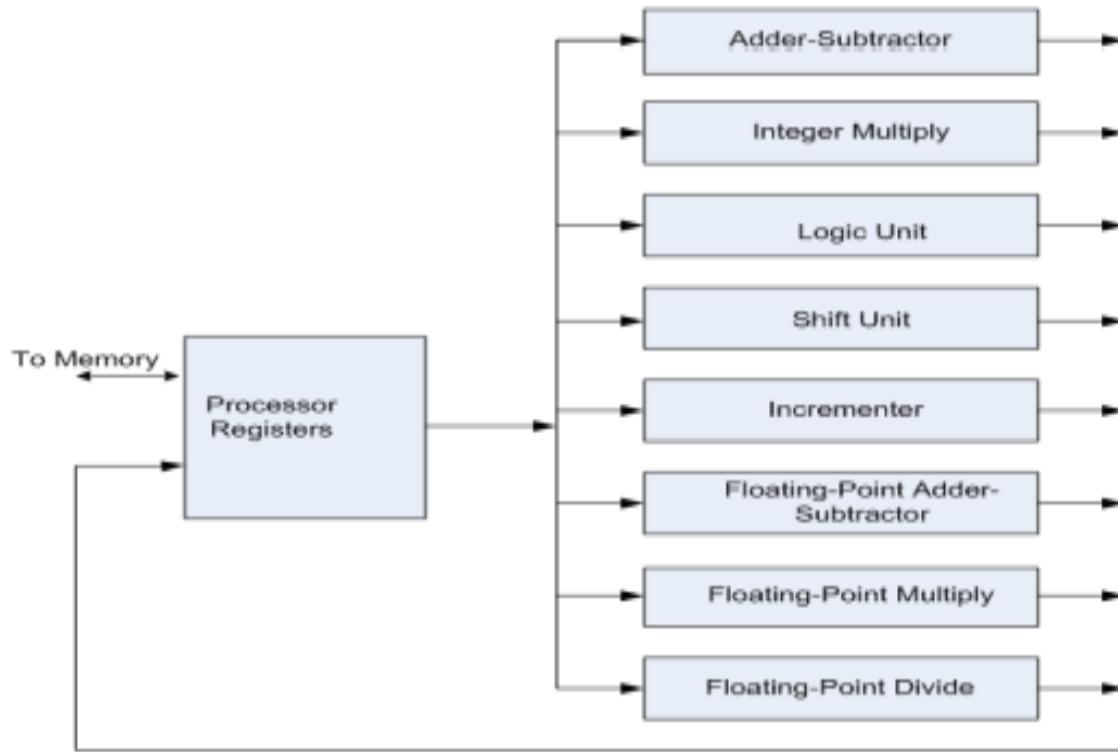
Fig: Processor with Multiple functional units

- The operation performed in each functional unit is indicated in each block of the diagram.

- The Adder and integer multiplier perform arithmetic operation with Integer numbers.

- The floating point operations are separated into 3 circuits operating in parallel.

- The logic, shift, and increment operation can be performed concurrently on different data.

- All units are independent, so one number can be shifted while another number is being activated.

- ## Architectural Classification: –

- ## Flynn's classification

- Considers the organization of a computer system by number of instructions and data items that are manipulated simultaneously.
- Based on the multiplicity of Instruction Streams and Data Streams
- **Instruction Stream**-Sequence of Instructions read from memory
- **Data Stream** - Operations performed on the data in the processor
- Parallel processing may occur in the instruction stream, in the data stream or in both.
- Flynn's classification divides computer into 4 major groups:

  1. SISD (Single Instruction stream, Single Data stream)

  2. SIMD (Single Instruction stream, Multiple Data stream)

  3. MISD (Multiple Instruction stream, Single Data stream)

  4. MIMD (Multiple Instruction stream, Multiple Data stream

		Number of *Data Streams*	
		Single	Multiple
Number of *Instruction Streams*	Single	SISD	SIMD
	Multiple	MISD	MIMD

- SISD represents the organization containing single control unit, a processor unit and a memory unit.
- Instruction are executed sequentially and system may or may not have

internal parallel processing capabilities.

- SIMD represents an organization that includes many processing units under the supervision of a common control unit.

- MISD structure is of only theoretical interest since no practical system has been constructed using this organization.

- MIMD organization refers to a computer system capable of processing several programs at the same time.

  The main difference between **multicomputer system and multiprocessor system** is that the multiprocessor system is controlled by one operating system that provides interaction between processors and all the component of the system cooperate in the solution of a problem

- Parallel Processing can be discussed under following topics:

- **Pipeline Processing**

- **Vector Processing**

- **Array Processors**

PIPELINING

- A technique of decomposing a sequential process into sub operations, with each sub process being executed in a special dedicated segment that operates concurrently with all other segments.

- A pipelinig is a **collection of processing segments.**

- Each segment performs partial processing dictated by the way task is partitioned.

- The result obtained from each segment is transferred to next segment.

- The final result is obtained when data have passed through all segments.

- Suppose we have to perform the following task:

- Each sub operation is to be performed in a segment within a pipeline.

- Each segment has one or two registers and a combinational circuit.

- The **register holds the data**. The **combinational circuit performs the suboperation** in the particular segment.

- A clock is applied to all registers after enough time has elapsed to perform all segment activity.

- A clock is applied to all registers after enough time has elapsed to perform all segment activity.

- The pipeline organization will be demonstrated by means of a simple example.

- To perform the combined multiply and add operations with a stream of numbers

$$\text{Ai * Bi + Ci} \qquad \text{for i = 1, 2, 3, …, 7}$$

- Each suboperation is to be implemented in a segment within a pipeline.

    R1 ← Ai , R2 ← Bi   Input Ai and Bi

    R3 ← R1 * R2, R4 ← Ci   Multiply and input Ci

    R5 ← R3 + R4   Add Ci to product

- Each segment has one or two registers and a combinational circuit as shown in Fig.
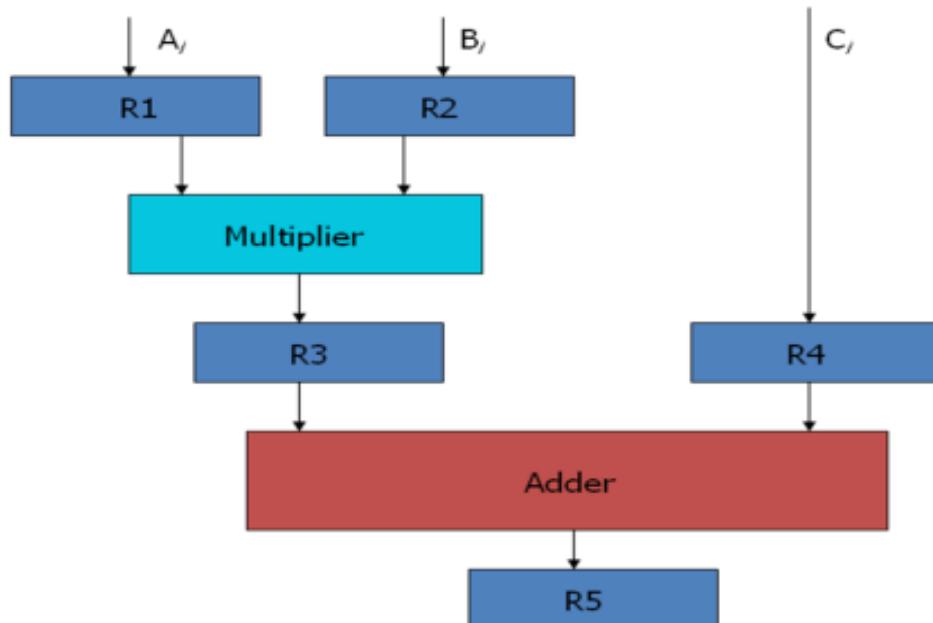
Fig 4-1: Example of pipeline processing

- The five registers are loaded with new data every clock pulse. The effect of each clock is shown in Table.

| Clock Pulse Number	Segment 1		Segment 2		Segment 3
	R1	R2	R3	R4	R5
1	A_1	B_1	--	--	--
2	A_2	B_2	$A_1{}^*B_1$	C_1	--
3	A_3	B_3	$A_2{}^*B_2$	C_2	$A_1{}^*B_1+C_1$
4	A_4	B_4	$A_3{}^*B_3$	C_3	$A_2{}^*B_2+C_2$
5	A_5	B_5	$A_4{}^*B_4$	C_4	$A_3{}^*B_3+C_3$
6	A_6	B_6	$A_5{}^*B_5$	C_5	$A_4{}^*B_4+C_4$
7	A_7	B_7	$A_6{}^*B_6$	C_6	$A_5{}^*B_5+C_5$
8	--	--	$A_7{}^*B_7$	C_7	$A_6{}^*B_6+C_6$
9	--	--	--	--	$A_7{}^*B_7+C_7$
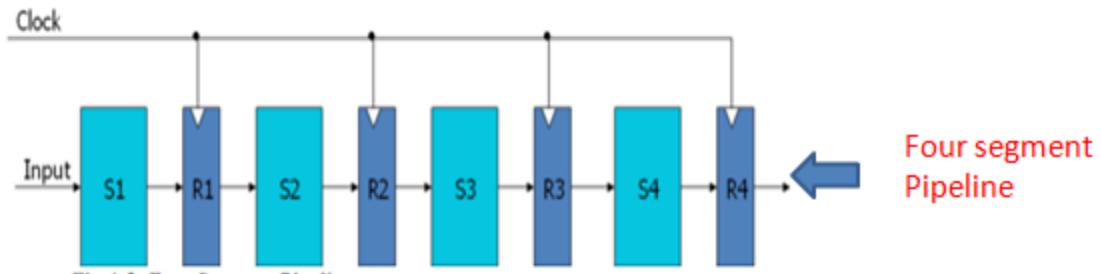
Table 4-1: Content of Registers in Pipeline Example

General Considerations:

- Any operation that can be decomposed into a sequence of suboperations of

about the same complexity can be implemented by a pipeline processor.

- The general structure of a **four-segment pipeline** is illustrated in Fig. 4-2. We define a task as the total operation performed going through all the segments in the pipeline.

- The behavior of a pipeline can be illustrated with a space-time diagram. o It shows the segment utilization as a function of time

- The space-time diagram of a four-segment pipeline is demonstrated in Fig
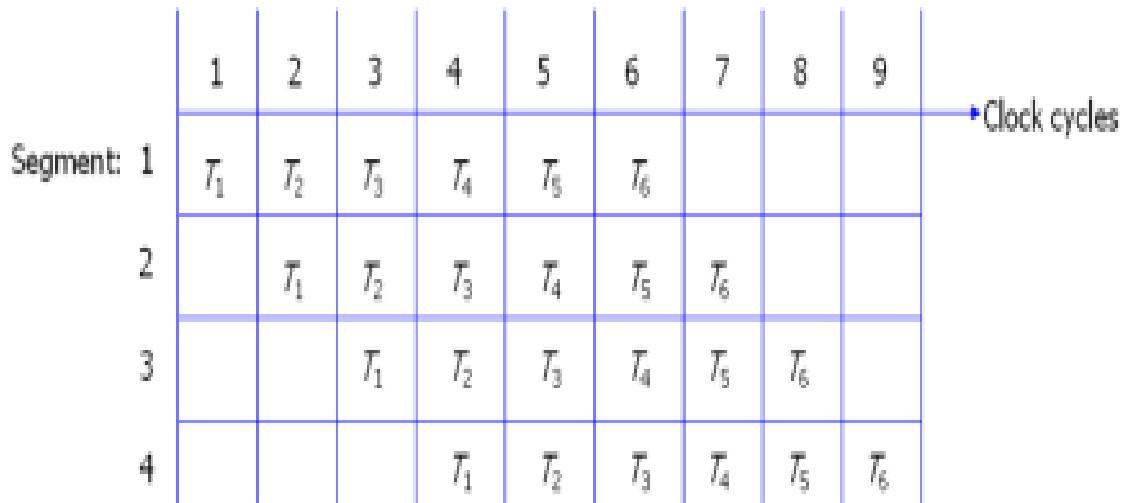
Fig 4-3: Space-time diagram for pipeline

- Where a k-segment pipeline with a clock cycle time tp is used to execute n tasks.

- The first task T1 requires a time equal to ktp to complete its operation.

- The remaining n-1 tasks will be completed after a time equal to (n-1)tp
- Therefore, to complete n tasks using a k-segment pipeline requires k+(n-1) clock cycles.
- Consider a nonpipeline unit that performs the same operation and takes a time equal to tn to complete each task.
- The total time required for n tasks is ntn.
- The speedup of a pipeline processing over an equivalent nonpipeline processing is defined by the ratio

  S = ntn/(k+n-1)tp .
- If n becomes much larger than k-1, the speedup becomes

  S = tn/tp.
- If we assume that the time it takes to process a task is the same in the pipeline and nonpipeline circuits, i.e.,tn = ktp, the speedup reduces to **S=ktp/tp=k.**
- This shows that the theoretical maximum speedup that a pipeline can provide is k, where k is the number of segments in the pipeline.
- To duplicate the theoretical speed advantage of a pipeline process by means of multiple functional units, it is necessary to construct k identical units that will be operating in parallel.
- This is illustrated in Fig. below, where four identical circuits are connected in parallel.
- Instead of operating with the input data in sequence as in a pipeline, the parallel circuits accept four input data items simultaneously and perform four tasks at the same time
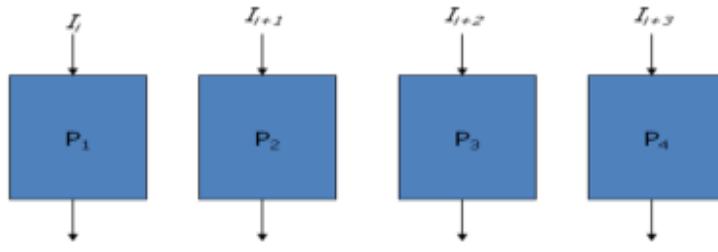
Fig 4-4: Multiple functional units in parallel

- There are various reasons why the pipeline cannot operate at its maximum theoretical rate.
- Different segments may take different times to complete their sub operation.
- It is not always correct to assume that a nonpipe circuit has the same time delay as that of an equivalent pipeline circuit.
- There are three areas of computer design where the pipeline organization is applicable.

Arithmetic pipeline

Instruction pipeline

RISC pipeline

Arithmetic pipeline:

- Pipeline arithmetic units are usually found in very high speed computers
- Floating–point operations, multiplication of fixed-point numbers, and similar computations in scientific problem
- Floating–point operations are easily decomposed into suboperations as demonstrated in Sec. 10-5.
- An example of a pipeline unit for floating-point addition and subtraction is showed in the following:
- The inputs to the floating-point adder pipeline are two normalized floating point binary number

$$X = A \times 2^a$$
$$Y = B \times 2^b$$

- A and B are two fractions that represent the mantissas, a and b are the exponents.
- The floating-point addition and subtraction can be performed in four segments, as shown in Fig. 9-6.
- The suboperations that are performed in the **four segments** are:

1.Compare the exponents

2. Align the mantissa

3. Add or subtract the mantissas

4. Normalize the result

Example: Consider two floating point numbers binary addition

$$X = 0.9504 * 10^3$$
$$Y = = 0.8200 * 10^2$$

1. Compare exponents by subtraction:

- The exponents are compared by subtracting them to determine their difference. The larger exponent is chosen as the exponent of the result.
- The difference of the exponents, i.e., 3 - 2 = 1 determines how many times the mantissa associated with the smaller exponent must be shifted to the right.

2. Align the mantissas:

- The next segment shifts the mantissa of Y to the right

$$X = 0.9504 * 10^3$$

$$Y = 0.08200 * 10^3$$

3. Add mantissas:

- The two mantissas are added in segment three.
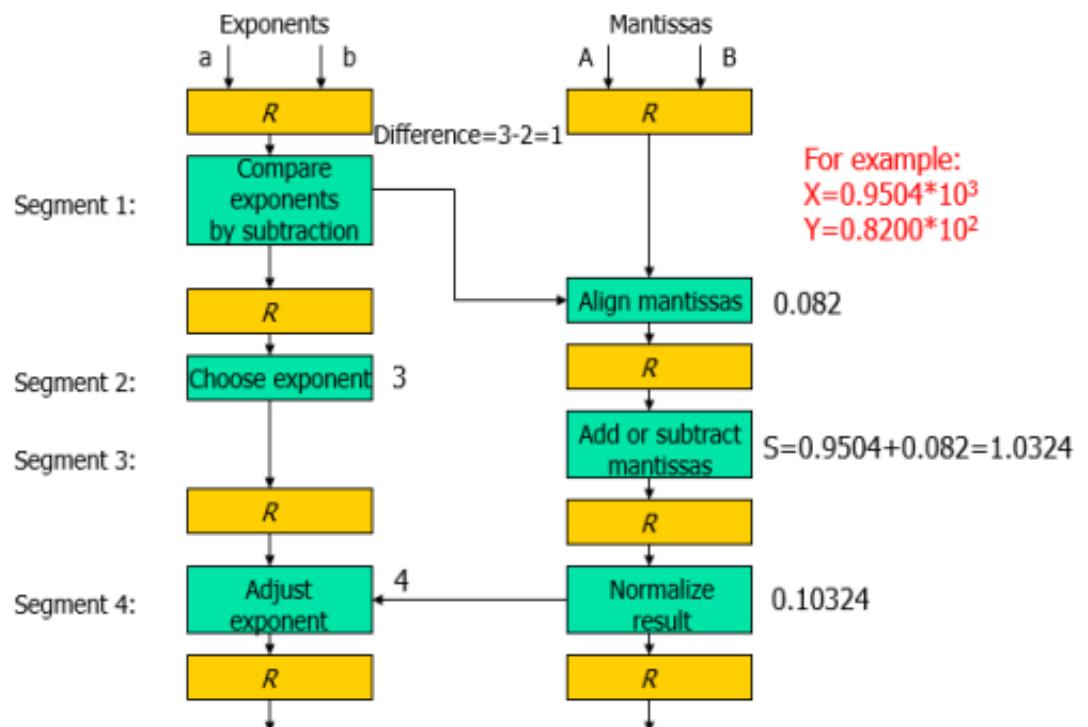
$$Z = X + Y = 1.0324 * 10^3$$

4. Normalize the result:

- After normalization, the result is written as:

$$Z = 0.1324 * 10^4$$

Flow chart for floating point addition and subtraction using Pipelining

Pipelining for Floating point Addition and Subtraction

- The larger exponent is chosen as the exponent of the result
- The exponent difference determines how many times the mantissa associated

with the smaller exponent must be shifted to the right.

- When an **overflow occurs**, the mantissa of the sum or difference is **shifted right** and the exponent incremented by one.

- If an **underflow occurs**, the number of leading zeros in the mantissa determines the number of **left shifts** in the mantissa and the the exponent decremented by one.

Instruction Pipeline:

- Pipeline processing can occur not only in the data stream but in the instruction as well.

- Consider a computer with an instruction fetch unit and an instruction execution unit designed to provide a two-segment pipeline.

- Computers with complex instructions require other phases in addition to above phases to process an instruction completely.

- In the most general case, the computer needs to process each instruction with the following sequence of steps.

> **1. Fetch the instruction from memory.**
>
> **2. Decode the instruction.**
>
> **3. Calculate the effective address.**
>
> **4. Fetch the operands from memory.**
>
> **5. Execute the instruction.**
>
> **6. Store the result in the proper place.**
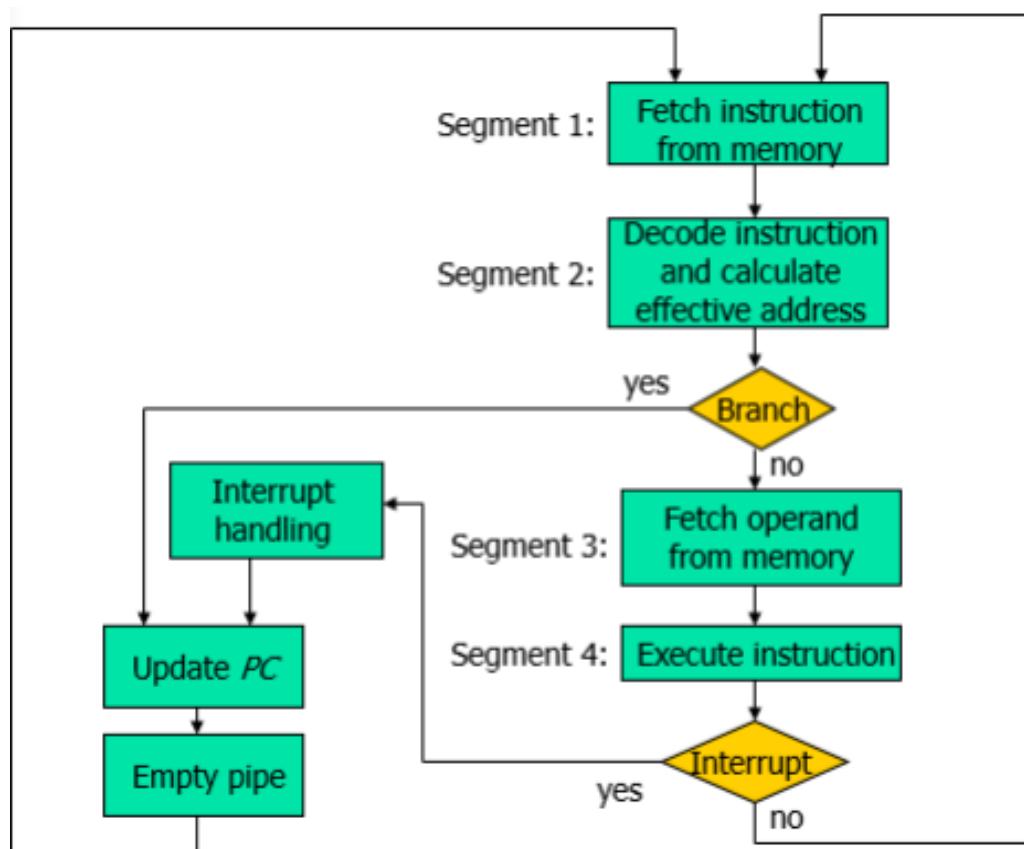
- There are certain difficulties that will prevent the instruction pipeline from operating at its maximum rate.

- Different segments may take different times to operate on the incoming information.

- Some segments are skipped for certain operations.
- Two or more segments may require memory access at the same time, causing one segment to wait until another is finished with the memory.

Example: four-segment instruction pipeline:

- Assume that:
- The decoding of the instruction can be combined with the calculation of the effective address into one segment (**DA in segment 2 and FI in segment 1**).
- The instruction execution and storing of the result can be combined into one **segment(FO in segment 3 and IE in segment 4)**
- Fig 9-7 shows how the instruction cycle in the CPU can be processed with a four segment pipeline.

- **Thus up to four suboperations in the instruction cycle can overlap** and

up to four different instructions can be in progress of being processed at the same time.

- An instruction in the sequence may be causes **a branch out of normal sequence.**

- In that case the pending operations in the last two segments are completed and all information stored in the instruction buffer is deleted.

- Similarly, an interrupt request will cause the pipeline to empty and start again from a new address value.

- Fig. above shows the operation of the instruction pipeline.

- **The four segments are represented in the diagram with an abbreviated symbol.**

 **1. FI is the segment that fetches an instruction.**
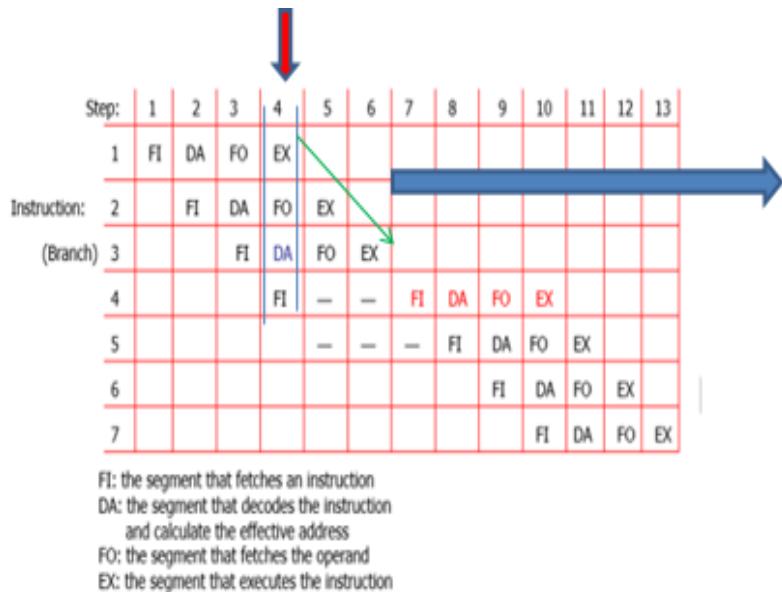
  2.DA is the segment that decodes the instruction and calculates the effective address.

  3. FO is the segment that fetches the operand.

  4. EX is the segment that executes the instruction

Timing of Instruction Pipeline

- The time in the horizontal axis is divided into steps of equal duration.

Step:	1	2	3	4	5	6	7	8	9	10	11	12	13
Instruction: 1	FI	DA	FO	EX									
2		FI	DA	FO	EX								
(Branch) 3			FI	DA	FO	EX							
4				FI	—	—	FI	DA	FO	EX			
5					—	—	—	FI	DA	FO	EX		
6									FI	DA	FO	EX	
7										FI	DA	FO	EX

FI: the segment that fetches an instruction
DA: the segment that decodes the instruction
 and calculate the effective address
FO: the segment that fetches the operand
EX: the segment that executes the instruction

Instruction 1,2 are executed sequentially, and at instruction 3, there is a branching address

At step4, Instruction 1 is executed
Instruction 2 is fetching operands from memory
Instruction 3(Brach Address) is decoded and calculating effective Address
Instruction 4 is fetching Instruction from memory
After Decoding the Branch address in Instruction 3, the transfer of other instruction from FI to DA is halted until the **current Instruction is executed in step6**
If the Branch address condition is satisfied, **a new Instruction** is fetched in **step7.**

- **Pipeline Hazards**
- It is a conflict that prevents an instruction from executing during its designated clock cycles.
- In general, there are three major difficulties that cause the instruction pipeline to deviate from its normal operation.

  1. Structural Hazards

  2. Data Hazard

  3. Control hazard

1. Structural Hazards:

- These are the Resource conflicts caused by access to memory by two segments at the same time.

- Can be resolved by using separate instruction and data memories

2. **Data Hazard:**

   These conflicts arise when an instruction depends on the result of a previous instruction, but this result is not yet available.

3. Control Hazard:

   These conflicts arise when an Branch instruction arise and this branch instruction causes the change the value of PC.

RISC (Reduced Instruction Set Computer)Pipeline:

- The data transfer instructions in RISC are LOAD and STORE.
- To prevent conflicts between a memory access to fetch an instruction and to load or store an operand, most RISC machines use two separate buses with two memories
- One for storing information and other for storing data.
- Example: Three-Segment Instruction Pipeline
- There are three types of instructions:
- The data manipulation instructions: operate on data in processor registers
- The data transfer instructions(load and store)
- The program control instructions(branch instructions)
- The instruction cycle can be divided into three suboperations and implemented in three segments:

I: Instruction fetch

- Fetches the instruction from program memory

A: ALU operation

- The instruction is decoded and an ALU operation is performed. It performs an operation for a data manipulation instruction, It evaluates the effective address for a load or store instruction. It calculates the branch address for a

program control instruction.

E: Execute instruction

- Directs the output of the ALU to one of three destinations, depending on the decoded instruction. It transfers the result of the ALU operation into a destination register in the register file.
- It transfers the effective address to a data memory for loading or storing.
- It transfers the branch address to the program counter.

Delayed Load:

- Consider the operation of the following four instructions:

  1. LOAD: R1 ← M[address 1]

  2. LOAD: R2 ← M[address 2]

  3. ADD: R3 ← R1 +R2

  4. STORE: M[address 3] ← R3

- There will be a data conflict in instruction 3 because the operand in R2 is not yet available in the A segment.
- This can be seen from the timing of the pipeline shown in Fig. 9-9(a).

Pipelining Timing with Delayed load:

Clock cycles:	1	2	3	4	5	6
1. Load R1	I	A	E			
2. Load R2		I	A	E		
3. Add R1+R2			I	A	E	
4. Store R3				I	A	E

9 (a) Pipeline timing with data conflict

At 4th clock cycle, the data is not placed in R2 but the A segment in clock cycle 4 wants the data from R2 to perform addition operation.

	1	2	3	4	5	6	7
1. Load R1	I	A	E				
2. Load R2		I	A	E			
3. No-operation			I	A	E		
4. Add R1+R2				I	A	E	
5. Store R3					I	A	E

9 (b) Pipeline timing with delayed load

This conflict can be overcomed by inserting a no operation instruction in the insturction3 and thus delaying the addition operation by one clock cycle.

This concept of delaying the data loaded into the memory is referred as "Delayed Load"

Delayed Branch

- The method used in most RISC processors is to rely on the compiler to redefine the branches so that they take effect at the proper time in the pipeline.

- This method is referred to as delayed branch.

- The compiler is designed to analyze the instructions before and after the branch and rearrange the program sequence by inserting useful instructions in the delay steps.

- It is up to the compiler to find useful instructions to put after the branch instruction. Failing that, the compiler can insert no-op instructions.

- **An Example of Delayed Branch**:

- The program for this example consists of five instructions.

    1. Load from memory to R1

2. Increment R2

3. Add R3 to R4

4. Subtract R5 from R6

5. Branch to address X

- In Fig. 9-10(a) the compiler inserts two no-op instructions after the branch.
- The branch address X is transferred to PC in clock cycle 7.
- The program in Fig. 9-10(b) is rearranged by placing the add and subtract instructions after the branch instruction.
- PC is updated to the value of X in clock cycle 5.

Pipelining Timing with Delayed Branch:

Clock cycles:	1	2	3	4	5	6	7	8	9	10
1. Load	I	A	E							
2. Increment		I	A	E						
3. Add			I	A	E					
4. Subtract				I	A	E				
5. Branch to X					I	A	E			
6. No-operation						I	A	E		
7. No-operation							I	A	E	
8. Instruction in X								I	A	E

10 (a) Using no-operation instructions

The compiler inserts two no-op instructions at 6 & 7 after the branch instruction (5).The branch address X is transferred to PC in clock cycle 7.so the fetching of instruction is delayed by 2 clock cycles and branch address x is fetched at instruction 8.

Clock cycles:	1	2	3	4	5	6	7	8
1. Load	I	A	E					
2. Increment		I	A	E				
3. Branch to X			I	A	E			
4. Add				I	A	E		
5. Subtract					I	A	E	
6. Instruction in X						I	A	E

10 (b) Rearranging the instructions

The program is rearranged by placing the add and subtract instructions after the branch instruction Inspection of the pipeline timing shows that PC is updated to the value of X in clock cycle 5

Vector processing:

- Normal computational systems are not enough in some special processing requirements
- In many science and engineering applications, the problems can be formulated in terms of vectors and matrices that lend themselves to vector processing.
- Computers with vector processing capabilities are in demand in specialized applications.

Examples:

- **Long-range weather forecasting**
- **Petroleum explorations**
- **Seismic data analysis**
- **Medical diagnosis**
- **Artificial intelligence and expert systems**
- **Image processing**
- **Mapping the human genome**

The term vector processing involves the data processing on the vectors of involving high amount of data.

- The large data can be classified as very big arrays.
- The vectors are considered as the large one dimensional array of data.
- The vector processing system can be understood by the example below.
- **EX: Consider a program which is adding two arrays A and B of length 100 to produce a vector C**
- Machine level program

      Initialize I=0

      Read A(I)
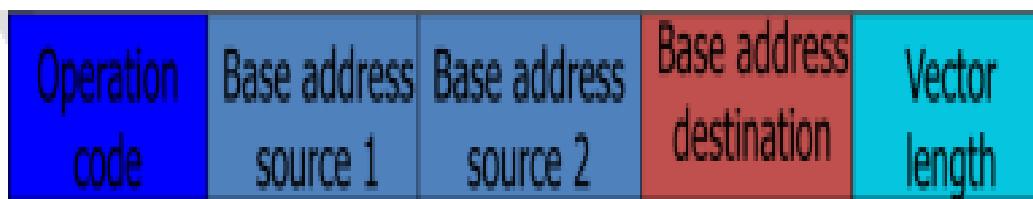
Read B(I)

20   Store C(I)=A(I)+B(I)

Increment I=I+1

If I<=100 go to 20 continue

- so in this above program we can see that the two arrays are being added in a loop format.
- First we are starting from the value of 0 and then we are continuing the loop with the addition operation until the I value has reached to 100.
- In the above program there are **5 loop statements** which will be executing 100 times.
- Therefore the total cycles of the CPU taken are **500 cycles**.
- But if we use the concept of vector processing then we can reduce the unnecessary fetch cycles.
- The same program written in the vector processing statement is given below:

C(1:100)=A(1:100)+B(1:100)

- In the above statement, when the system is creating a vector like this the original source values are fetched from the memory into the vector.
- Therefore the data is readily available in the vector.
- So when a operation is initiated on the data, naturally the operation will be performed directly on the data and will not wait for the fetch cycle.
- So the **total no of CPU Cycles** taken by the above instruction is only **100**
- **Instruction format of vector Instruction:**

Operation code	Base address source 1	Base address source 2	Base address destination	Vector length

Matrix Multiplication

- The multiplication of two n x n matrices consists of n2 inner products or n3 multiply-add operations.

- Consider, for example, the multiplication of two 3 x 3 matrices A and B.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{bmatrix}$$

The product matrix C is a 3×3 matrix whose elements are related to the elements of A and B by the inner product:

$$c_{ij} = \sum_{k=1}^{3} a_{ik} \times b_{kj}$$

For example, the number in the first row and first column of matrix C is calculated by letting $i = 1, j = 1$, to obtain

c11= a11b11+ a12b21+ a13b31

- This requires three multiplication and (after initializing c11 to 0) three additions.

- In general, the inner product consists of the sum of k product terms of the form
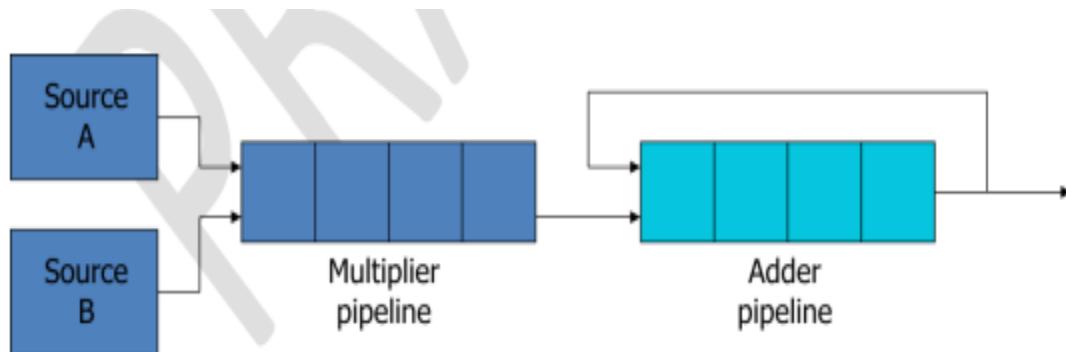
C = A1B1+A2B2+A3B3+…+AkBk.

- In a typical application k may be equal to 100 or even 1000.

- The inner product calculation on a pipeline vector processor is shown in Fig. 9-12.

$$C = A_1B_1 + A_5B_5 + A_9B_9 + A_{13}B_{13} + \cdots$$
$$+ A_2B_2 + A_6B_6 + A_{10}B_{10} + A_{14}B_{14} + \cdots$$
$$+ A_3B_3 + A_7B_7 + A_{11}B_{11} + A_{15}B_{15} + \cdots$$
$$+ A_4B_4 + A_8B_8 + A_{12}B_{12} + A_{16}B_{16} + \cdots$$

Implementation of the Vector Processing

- Below we can see the implementation of the vector processing concept on the following matrix multiplication.
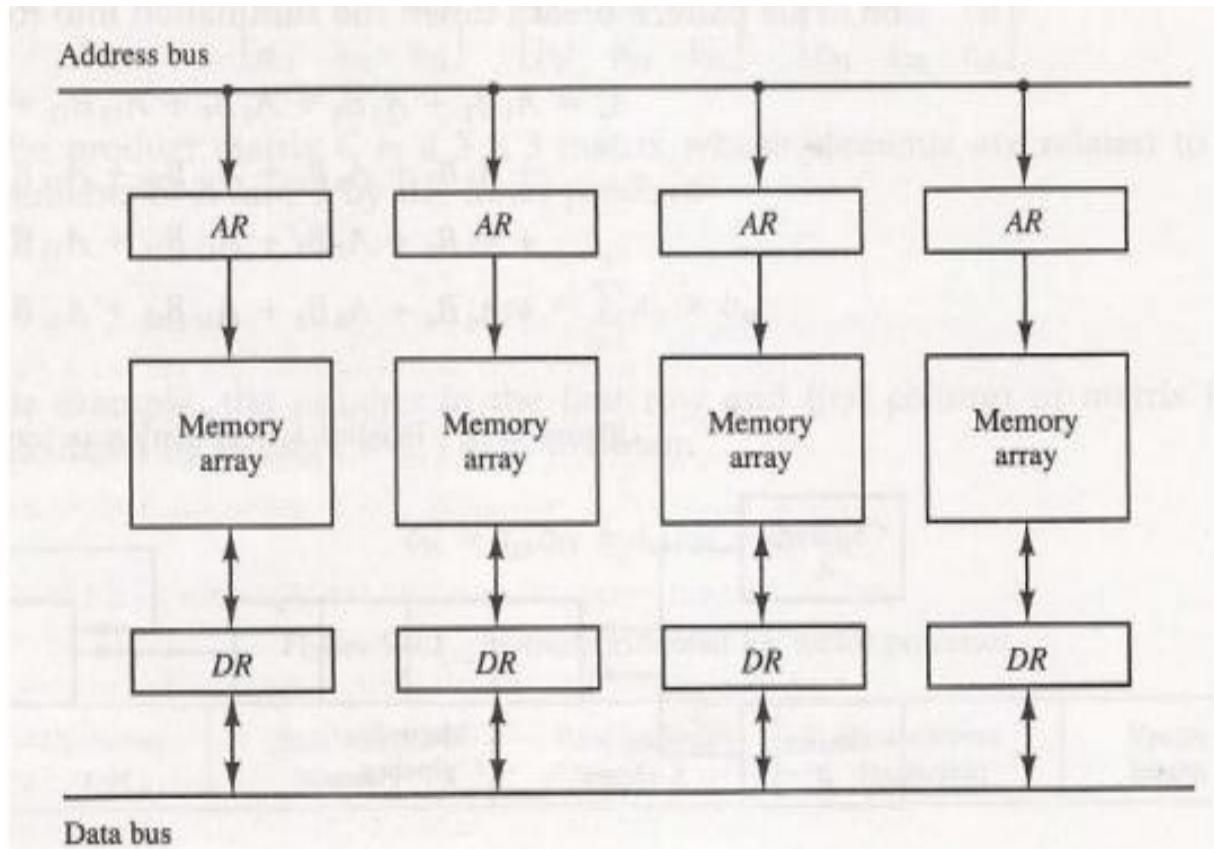

- In the above diagram we can see that how the values of A vector and B Vector which represents the matrix are being multiplied. Here we will be considering a 4x4 matrix A and B.
- When addition operation is taking place in the adder pipeline the next set of values will be brought into the multiplier pipeline, so that all the operations can be performed simultaneously using the parallel processing concepts by the implementation of pipeline.

Memory Interleaving:

- Pipeline and vector processors often require simultaneous access to memory from two or more sources.
- An instruction pipeline may require the fetching of an instruction and an operand at the same time from two different segments.

- An arithmetic pipeline usually requires two or more operands to enter the pipeline at the same time.
- **Instead of using two memory buses for simultaneous access, the memory can be partitioned into a number of modules** connected to a common memory address and data buses.
- A memory module is a memory array together with its own address and data registers.
- Fig. 9-13 shows a memory unit with four modules.

Multiple module Memory Organization

- The advantage of a modular memory is that it allows the use of a technique called interleaving.
- In an interleaved memory, different sets of addresses are assigned to different memory modules.

- By staggering the memory access, the effective memory cycle time can be reduced by a factor close to the number of modules.

Array Processors:

- An array processor is a processor that performs computations on large arrays of data.
- The term is used to refer to two different types of processors.

Attached array processor:

It is an auxiliary processor. It is intended to improve the performance of the host computer in specific numerical computation tasks.
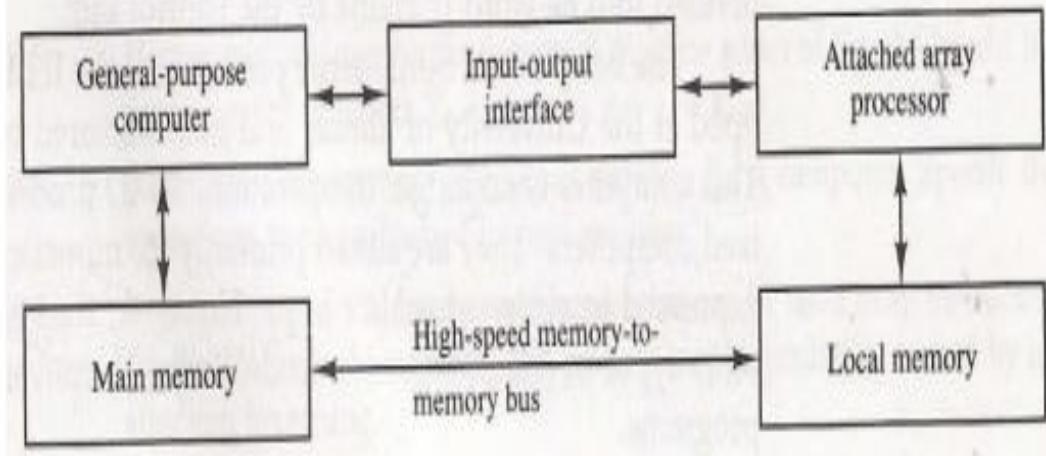
SIMD array processor:

Has a single-instruction multiple-data organization. It manipulates vector instructions by means of multiple functional units responding to a common instruction

Attached Array Procesor

- Its purpose is to enhance the performance of the computer by providing vector processing for complex scientific applications.
- Parallel processing with multiple functional units
- Fig. 9-14 shows the interconnection of an attached array processor to a host computer.

Figure 9-14 Attached array processor with host computer.

Attached Array Processor with host computer

- The host computer is a general-purpose commercial computer and the attached processor is a back-end machine driven by the host computer.
- The array processor is connected through an input-output controller to the computer and the computer treats it like an external interface.
- The data for the attached processor are transferred from main memory to a local memory through a high-speed bus.
- The general-purpose computer without the attached processor serves the users that need conventional data processing.
- The system with the attached processor satisfies the needs for complex arithmetic applications.
- For example, when attached to a VAX 11 computer, the FSP-164/MAX from Floating Point Systems increases the computing power of the VAX to 100megaflops.
- The objective of the attached array processor is to provide vector manipulation capabilities to a conventional computer at a fraction of the cost of supercomputer.

SIMD Array Processor:

- An SIMD array processor is a computer with multiple processing units operating in parallel.
- A general block diagram of an array processor is shown in Fig. 9-15.
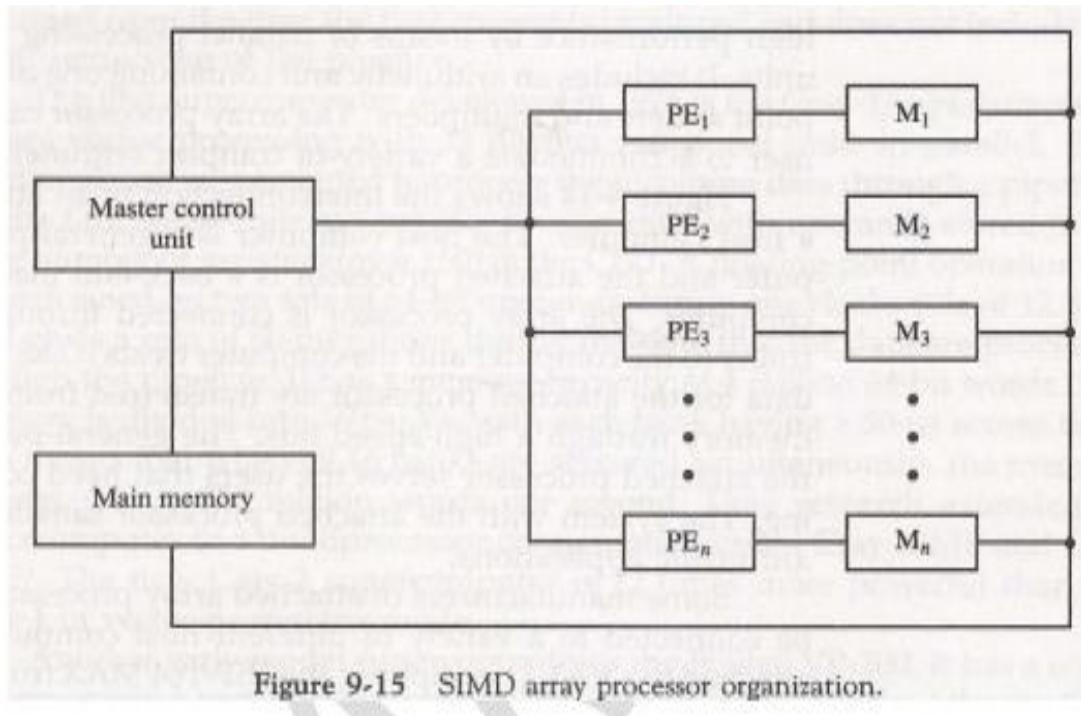
Figure 9-15   SIMD array processor organization.

- It contains a set of identical processing elements (PEs), each having a local memory M.
- Each PE includes an ALU, a floating-point arithmetic unit, and working registers.
- Vector instructions are broadcast to all PEs simultaneously.
- Masking schemes are used to control the status of each PE during the execution of vector instructions.
- Each PE has a flag that is set when the PE is active and reset when the PE is inactive.
- For example, the ILLIAC IV computer developed at the University of

Illinois and manufactured by the Burroughs Corp.

- Are highly specialized computers.
- They are suited primarily for numerical problems that can be expressed in vector or matrix form