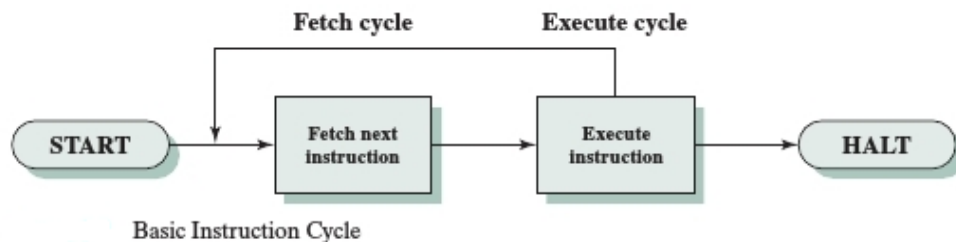


Processing cycle of a stored program computer

- The processing cycle is the series of steps involved with the execution of a single instruction
- The processing required for a single instruction is called an **instruction cycle**
- In its simplest form , instruction processing consists of two steps
 - The processor reads (fetches) instructions from memory one at a time and executes each instruction
- Program execution consists of repeating the process of instruction fetch and instruction execution
- The instruction execution may involve several operations and depends on the nature of the instruction
- Program execution halts only if the machine is turned off, some sort of unrecoverable error occurs, or a program instruction that halts the computer is encountered.



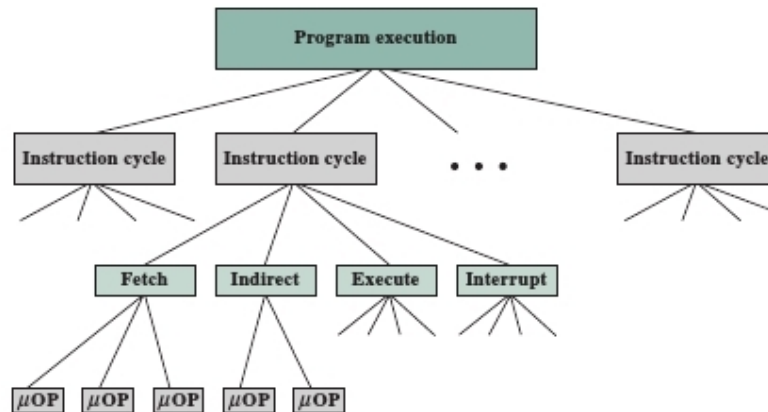
- In general processor has to do following thing to execute an instruction
 - **Instruction address calculation :**
 - Determine the address of the next instruction to be executed. Usually, this involves adding a number to the address of the previous instruction
 - In a typical processor, a register called the program counter (PC) holds the address of the instruction to be fetched next.
 - Unless told otherwise, the processor always increments the PC after each **instruction fetch** so that it will fetch the next instruction in sequence
 - **Instruction fetch :** Read instruction from its memory location into the processor
 - At the beginning of each instruction cycle, the processor fetches an instruction from memory.
 - **Instruction operation decoding:** Decode instruction to determine type of operation to be performed and operand(s) to be used.
 - The fetched instruction is loaded into a register in the processor known as the instruction register (IR). The instruction contains bits that specify

the action the processor is to take. The processor interprets the instruction and performs the required action.

- **Operand address calculation** : If the operation involves reference to an operand in memory or available via I/O, then determine the address of the operand
 - **Operand fetch**: Fetch the operand from memory or read it in from I/O.
 - **Data operation**: Perform the operation indicated in the instruction.
 - **Operand store** : Write the result into memory or out to I/O
- *Note that not all instructions have all above phases

Micro-operation

- Each instruction that is executed during an instruction cycle made up of shorter sub cycles (e.g., fetch, indirect, execute, and interrupt).
- The execution of each sub cycle involves one or more shorter operations, each of which involves the processor registers. These operations are known as micro-operation.
- A micro-operation is an elementary operation performed with the data stored in one or more processor registers.
- Hence events of any instruction cycle can be described as a sequence of micro-operations



Control unit

The control unit receives the instruction from memory and interprets the operation code bits. It then issues a sequence of control signals to initiate micro-operations in internal computer registers. For every operation code, the control unit issues a sequence of micro-operations needed for the hardware implementation of the specified operation.

- The control unit performs two basic tasks:
 - Sequencing: The control unit causes the processor to step through a series of micro-operations in the proper sequence, based on the program being executed.
 - Execution: The control unit causes each micro-operation to be performed
- Types of control unit :
 - Hardwired control unit
 - Microprogramed control unit

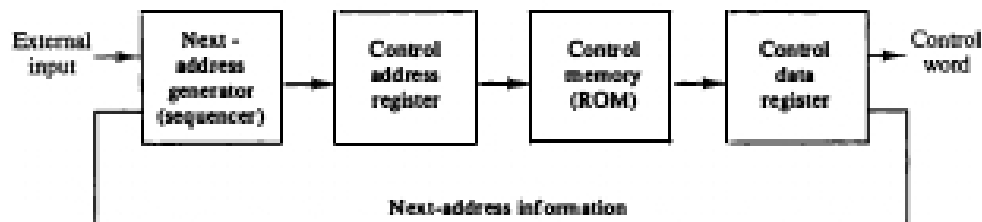
Hardwired Control unit

- When the control signals are generated **by hardware using conventional logic design technique**, the control unit is said to be hardwired.
- This control unit is essentially a combinatorial circuit
- Its input logic signals are transformed into a set of output logic signals, which are the control signals.
- To design this control unit we have to derive Boolean expression for each control signal as a function of its input.
- In a modern complex processor, the number of Boolean equations needed to define the control unit is very large. The task of implementing a combinatorial circuit that satisfies all of these equations becomes extremely difficult.
- To implement a hardwired control unit as an interconnection of basic logic elements is not easy task.
- The design must include logic for
 - sequencing through micro-operations
 - executing micro-operations
 - Interpreting opcodes, and for making decisions based on ALU flags.
- It is difficult to design and test such a piece of hardware. Furthermore, the design is relatively inflexible. For example, it is difficult to change the design if one wishes to add a new machine instruction.
- The solution is to implement a microprogrammed control unit.
- Features of hardwired control unit
 - It has fixed instructions
 - High speed
 - Expensive
 - Complex
 - No flexibility of adding new instructions
 - RISC processor uses hardwired control unit

Microprogramed control unit

- The control unit generates control signals that specifies one or more micro-operations.
- The control unit initiates a series of sequential steps of micro-operations.
- During any given time, certain micro-operations are to be initiated, while other remains idle.
- The control variables at any given time that represents one or more micro-operations is a string of 1's and 0's called control word.
- A control unit whose binary control variables are stored in memory is called a microprogrammed control unit.
- Each word in control memory contains within it a microinstruction.
- The microinstruction specifies one or more micro-operations for the system.
- The microinstruction specify various internal control signals for the execution of micro operations.
- To execute microinstruction, turn on all the control lines indicated by a 1 bit; leave off all control lines indicated by a 0 bit. The resulting control signals will cause one or more micro-operations to be performed.
- A sequence of microinstructions constitutes a microprogram.

Figure 7-1 Microprogrammed control organization.



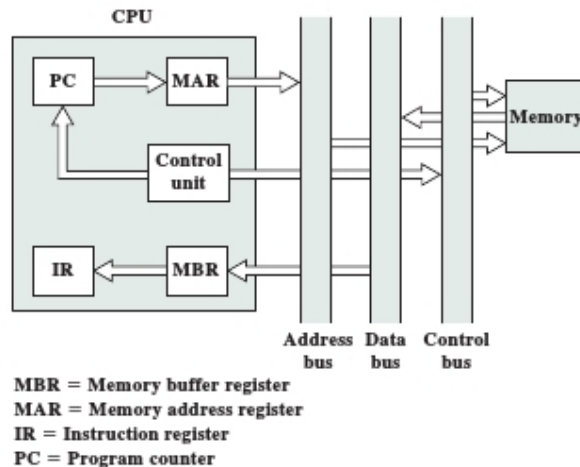
- Control memory address register specifies the address of the microinstruction and the control data register holds the microinstruction read from the control memory.
- The microinstruction contains a control word that specifies one or more micro-operations. Once these operations are executed, the control must determine the next address of microinstruction.
- The location of the next microinstruction may be the one next in sequence, or it may be located somewhere else in the control memory.
- For this reason it is necessary to use some bits of present microinstruction to control the generation of the address of next microinstruction.
- The next address may also be the function of external input conditions.
- While the micro-operations are being executed, the next address is computed in the next address generator circuit and then transferred into the control address register to read the next microinstruction.

- **Features of microprogrammed control unit**
 - Simplifies design of control unit
 - Cheaper
 - Less prone to error
 - Slower than hardwired control unit
 - CISC processor uses microprogrammed control unit

HARDWIRED CONTROL UNIT	MICROPROGRAMMED CONTROL UNIT
A unit that uses combinational logic units, featuring a finite number of gates that can generate specific results based on the instructions that were used to invoke those responses	A unit that contains microinstructions in the control memory to produce control signals
Speed of operations is fast	Speed of operations is slow because it requires frequent memory accesses
To do modifications, the entire unit should be redesigned	Modifications can be implemented by changing the microinstructions in the control memory
More costly to implement	Less costly to implement
It is difficult to handle complex instructions	It is easier to handle complex instructions
It is difficult to perform instruction decoding	Less difficult to perform instruction decoding
Uses a small instruction set	Uses a large instruction set
There is no control memory usage	Uses control memory
Used in processors that use a simple instruction set known as the Reduced Instruction Set Computers (RISC)	Used in processors based on a complex instruction set known as Complex Instruction Set Computer (CISC)

Register transfer language (RTL)

- A register transfer language is a system for expressing in symbolic form the micro-operation sequences among the registers of a digital module.



Memory address register (MAR): Is connected to the address lines of the system bus. It specifies the address in memory for a read or write operation.

Memory buffer register (MBR): Is connected to the data lines of the system bus. It contains the value to be stored in memory or the last value read from memory.

Program counter (PC): Holds the address of the next instruction to be fetched.

Instruction register (IR): Holds the last instruction fetched.

Op code fetch sequence

- Address of next instruction is in PC
- Move that address to the memory address register (MAR) because this is the only register connected to the address lines of the system bus.
- The desired address (in the MAR) is placed on the address bus
- Control unit issues READ command
- Result (data from memory) appears on data bus
- Data from data bus copied into MBR
- PC incremented by 1
- Data (instruction) moved from MBR to IR

Op code Fetch Cycle	T ₁ :	MAR ← PC
	T ₂ :	MBR ← [MAR]
	T ₃ :	IR ← MBR, PC ← PC+1
	T ₄ :	Unspecified

Example: Fetch and execute cycle of MOV R_d,R_s in terms of RTL

RTL of MOV R _d , R _s	Opcode Fetch Cycle	T ₁ :	MAR ← PC
		T ₂ :	MBR ← [MAR]
		T ₃ :	IR ← MBR, PC←PC+1
		T ₄ :	Unspecified

Example : MVI R, 8 bit data

RTL of MVI R, 8-bit Data	Opcode Fetch Cycle	T ₁ :	MAR ← PC
		T ₂ :	MBR ← [MAR]
		T ₃ :	IR ← MBR, PC←PC+1
		T ₄ :	Unspecified
	Memory Read Cycle	T ₅ :	MAR ← PC
		T ₆ :	MBR ← [MAR]
		T ₇ :	R ← MBR, PC ← PC + 1

Example : MOV R,M

RTL of MOV R, M	Opcode Fetch Cycle	T ₁ :	MAR ← PC
		T ₂ :	MBR ← [MAR]
		T ₃ :	IR ← MBR, PC←PC + 1,
		T ₄ :	Unspecified
	Memory Read Cycle	T ₅ :	MAR ← HL
		T ₆ :	MBR ← [MAR]
		T ₇ :	R ← MBR

Example : MVI M, 8 bit data

RTL of MVI M, 8-bit Data	Opcode Fetch cycle	T ₁ :	MAR ← PC
		T ₂ :	MBR ← [MAR]
		T ₃ :	IR ← MBR, PC ← PC+1
		T ₄ :	Unspecified
	Memory Read cycle	T ₅ :	MAR ← PC
		T ₆ :	MBR ← [MAR]
		T ₇ :	Z ← MBR, PC ← PC + 1
	Memory Write cycle	T ₈ :	MAR ← HL
		T ₉ :	MBR ← Z
		T ₁₀ :	[MAR] ← MBR
		.	

Example: LXI Rp, 16 bit data

RTL of LXI D, 16-bit Data	Opcode Fetch cycle	T ₁ :	MAR ← PC
		T ₂ :	MBR ← [MAR]
		T ₃ :	IR ← MBR, PC ← PC + 1
		T ₄ :	Unspecified
	Memory Read cycle	T ₅ :	MAR ← PC
		T ₆ :	MBR ← [MAR]
		T ₇ :	RP _L ← MBR, PC ← PC + 1
	Memory Read cycle	T ₈ :	MAR ← PC
		T ₉ :	MBR ← [MAR]
		T ₁₀ :	RP _H ← MBR, PC ← PC + 1

Example: LDA 16 bit data

RTL of LDA 16-bit Address	Opcode Fetch cycle	$T_1 :$ $MAR \leftarrow PC$ $T_2 :$ $MBR \leftarrow [MAR]$ $T_3 :$ $IR \leftarrow MBR, PC \leftarrow PC + 1$ $T_4 :$ Unspecified
	Memory read cycle	$T_5 :$ $MAR \leftarrow PC$ $T_6 :$ $MBR \leftarrow [MAR]$ $T_7 :$ $Z \leftarrow MBR, PC \leftarrow PC + 1$
	Memory Read cycle	$T_8 :$ $MAR \leftarrow PC$ $T_9 :$ $MBR \leftarrow [MAR]$ $T_{10} :$ $W \leftarrow MBR, PC \leftarrow PC + 1$
	Memory Read cycle	$T_{11} :$ $MAR \leftarrow WZ$ $T_{12} :$ $MBR \leftarrow [MAR]$ $T_{13} :$ $A \leftarrow MBR$

LDAX R_P

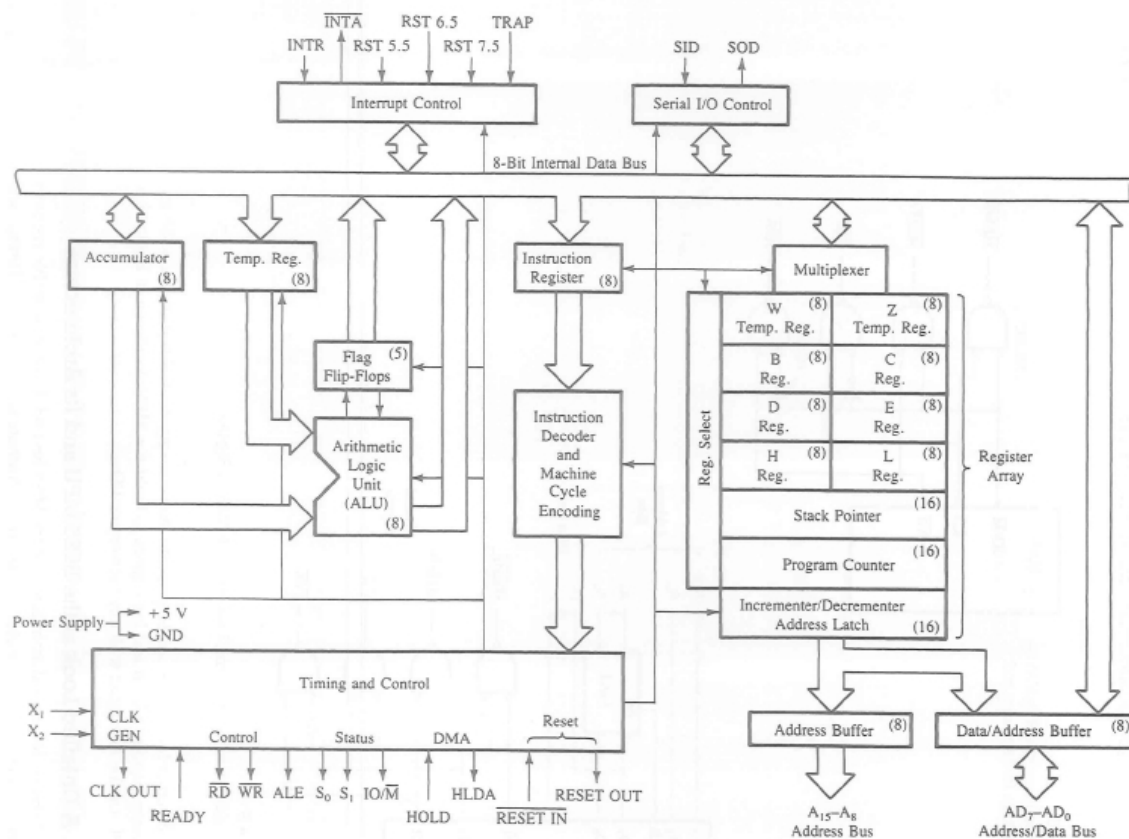
RTL of LDAX R _P	Opcode Fetch Cycle	T ₁ : T ₂ : T ₃ : T ₄ :	MAR ← PC MBR ← [MAR] IR ← MBR, PC ← PC + 1 Unspecified
	Memory Read Cycle	T ₅ : T ₆ : T ₇ :	MAR ← R _P MBR ← [MAR] A ← MBR

STA 16-bit Address

RTL of STA 16-bit Address	Opcode Fetch Cycle	T ₁ : T ₂ : T ₃ : T ₄ :	MAR ← PC MBR ← [MAR] IR ← MBR, PC ← PC + 1 Unspecified
	Memory Read Cycle	T ₅ : T ₆ : T ₇ :	MAR ← PC MBR ← [MAR] Z ← MBR, PC ← PC + 1
	Memory Read Cycle	T ₈ : T ₉ : T ₁₀ :	MAR ← PC MBR ← [MAR] W ← MBR, PC ← PC + 1
	Memory Write Cycle	T ₁₁ : T ₁₂ : T ₁₃ :	MAR ← WZ MBR ← A [MAR] ← MBR

Internal Architecture of 8085 Microprocessor

The Intel 8085 is a complete **8 bit parallel central processing unit**. The main components of 8085 are array of registers, the arithmetic logic unit, the encoder/decoder, and timing and control circuits linked by an internal data bus. The block diagram is shown below



1. ALU (Arithmetic Logic unit)

- It performs arithmetic and logical operations
- It includes Accumulator, temporary register, arithmetic and logic circuits and five flags.
- Temporary register is used to hold data during an arithmetic/logic operation.

2. Accumulator (Register A)

- 8 bit register
- Used to store 8 bit data and to perform arithmetic and logical operation.
- The result of an operation is stored in the accumulator.
- It is also known as register A.
- When data is read from input port, it first moved to accumulator and when data is sent to output port, it must be first placed in accumulator.

3. Temporary Register (W and Z)
 - They are 8 bit register
 - These registers are used to hold 8 bit data during execution of some instructions
 - They are not available to the programmer
4. Instruction Register (IR)
 - When an instruction is fetched from memory, it is loaded in the IR.
 - It is a 8 bit register
 - **It is not programmable and can't be accessed through any instruction.**
5. Instruction decoder
 - **It takes the information from instruction register and decodes the instruction**
6. General Registers
 - It has 6 general purpose register (B,C,D,E,H,L)
 - 8 bit registers
 - Can be used individually as 8 bit registers or in pair as HL,BC,DE as 16 bit registers
 - Accessible to user
7. Stack Pointer (SP)
 - **16 bit register used as a memory pointer**
 - It points to a memory location in R/W memory, called stack.
 - **The beginning of the stack is defined by loading a 16 bit address in the stack pointer.**
8. Program Counter(PC)
 - **16 bit register**
 - Microprocessor use the PC to sequence the execution of the instructions.
 - PC points to the memory address from which the next byte is to be fetched.
 - When a machine code is being fetched, the PC is incremented by 1 to point to the next memory location.
9. Flag

B ₇	B ₆	B ₅	B ₄	B ₃	B ₂	B ₁	B ₀
S	Z	—	AC	—	P	—	CY

- It is an 8 bit register containing five 1 bit flags : Zero(Z), Carry(CY) , Sign (S) , Parity (P) , Auxiliary Carry (AC)
- The flags are set or reset after an operation according to the data condition of the result in the Accumulator or other registers.
- In most of the cases the result of the arithmetic/logic operation is stored in the Accumulator, and the flags are set or reset according to the result of the operation.

- Microprocessor uses the conditions (set or reset) of flags to test data conditions and the flag conditions are tested through software instruction.

Z-zero flag:

- This flag is set if **ALU** operation results in 0 otherwise reset

AC-Auxiliary Carry flag:

- **In an arithmetic operation**, when a carry is generated by digit D_3 and passed to digit D_4 , the AC flag is set.

P-Parity flag:

- **After an arithmetic or logical operation, if the result has an even number of 1s**, the flag is set otherwise it is reset.

CY-Carry flag:

- If an arithmetic operation results in a carry, the CY flag is set otherwise it is reset. The carry flag also serves as a borrow flag for subtraction

S-sign flag:

- **After the execution of an arithmetic and logical operation, if bit D7 of the result is 1, the sign flag is set.**
- This flag is used with signed number.
- In signed number bit D7 is reserved for sign and remaining bits are used to represent magnitude of the number.
- **If the bit D7 is 1 the number is negative and if it is 0 the number is positive**
- This flag is irrelevant for unsigned number.

10. Timing and control unit

- This unit synchronizes all the microprocessor operation with clock
- generates the control signal necessary for communication between the microprocessor and peripheral
- \overline{RD} : This is active low used for reading operation
- \overline{WR} : This is active low used for writing operation

11. Interrupt controls

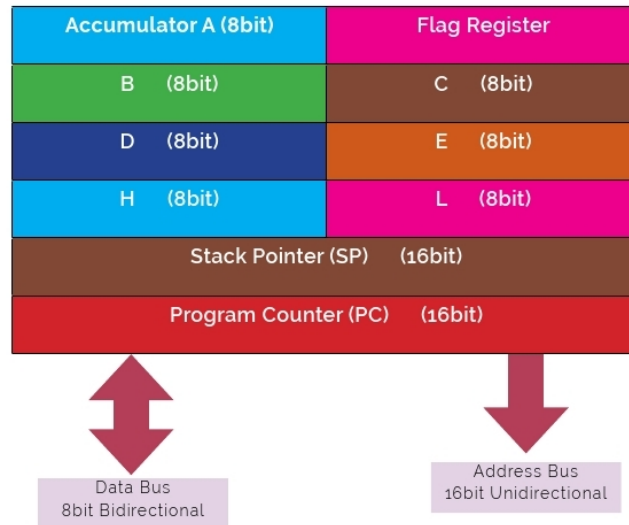
- Interrupt signals are used to interrupt a program execution
- 8085 has 5 interrupt signals (INTR, RST 5.5, RST 6.5, RST 7.5 and TRAP)
- Interrupt priority: TRAP, RST 7.5, RST 6.5, RST 5.5, and INTR.

12. Serial I/O control

- Two serial I/O control signals (SID and SOD) are used to implement the serial data transmission
- SID(serial input data) , SOD(Serial output data)

Programmer's model of an 8085 microprocessor

Programmers Model refers to an abstract model that describes how a programmer should use the core components of the processor to program it. It describes only about fewer hardware which are accessible to programmer



Programmer model of 8085 consists of

- Accumulator
- Flags
- Stack pointer
- Program counter
- General purpose registers (B,C,D,E,H,L)

Instruction

Instruction is a binary pattern entered through an input device in memory to **command the microprocessor** to perform that specific function

OR

An instruction is a command to the microprocessor to perform a given task on specified data.

Mnemonic

- Mnemonics are shortened form of English word for the operation performed by the instruction
- SUB is mnemonics for subtract , ADD is mnemonic for addition
- Program written in these mnemonic is called an assembly language program

Instruction format

Opcode Field	Operand Field
--------------	---------------

- Instruction has two parts
 - Opcode field :
 - It contains mnemonic that specify the task to be performed
 - This mnemonic is also known as operation code (op-code)
 - Operand field:
 - **Operand field contains the operand.**
 - Operand is just another name for the data item(s) acted on by an instruction
 - Operand can be specified in various ways
 - Operand may include 8 bit or 16 bit **data**, the memory address, the port address or the name of the register on which the instruction is to be performed.
 - **In some instruction operand is implicit.**

Instruction	Opcode	Operand
MOV C,A	MOV	C,A
MVI A,32H	MVI	A,32H
JMP 2085	JMP	2085H
CMA	CMA	

Instruction word size

Based upon the word size or byte size 8085 has 3 types of instruction

- 1 byte instruction
- 2 byte instruction
- 3 byte instruction

One byte instruction

- A mnemonic followed by a letter (or two letter) representing the register (such as A,B,C,D,E,H,L,M,SP) is one byte instruction
- Instruction in which registers are **implicit** are also 1 byte instruction

Instruction	Binary code	Hex code
MOV C,A	0100 1111	4FH
CMA	0010 1111	2FH
ADD B	1000 0000	80H

CMA: Accumulator is assumed to be implicit operand

Two byte instruction

- A mnemonic followed by 8 bit is two byte instruction
- First byte is opcode and second byte is operand
- Requires two memory location

Instruction	Binary code	Hex code
ADI 54H	1100 0110	C6H
	0101 0100	54H
MVI A,32H	0011 1110	3EH
	0011 0010	32H

Three byte instruction

- A mnemonic followed by 16 bit is three byte instruction
- First byte is opcode and following two bytes specifies the 16 bit address or immediate data
- Requires three memory location

Instruction	Binary code	Hex code	Remarks
LDA 2050H	0011 1010	3A	opcode
	0101 0000	50H	Second byte is lower order address
	0010 0000	20H	Third byte is higher order address
LXI H,2050H	0010 0001	21	opcode
	0101 0000	50	Second byte is lower order address
	0010 0000	20	Third byte is higher order address

How instructions are stored in memory location

Memory	Instruction	Hex value	Remarks
2030H	LDA 2050H	3AH	Hex value of LDA
2031H		50H	
2032H		20H	
2033H	MOV B,A	47H	Hex value of MOV B,A
2034H	MVI C,20H	0EH	Hex value of MVI C
2035H		20H	

The 8085 instruction set

The 8085 instructions can be classified into the following 5 functional categories

1. **Data transfer operation:** This group of instructions copies data from a location called source to another location called, a destination, without modifying the contents of source.
2. **Arithmetic operation:** The instructions which perform arithmetic operations such as addition, subtraction, increment and decrement.
3. **Logical operation:** These instructions perform various logical operations with the contents of accumulator.
 - AND , OR ,Exclusive OR
 - Rotate
 - Compare
 - Complement
4. **Branching operation :** This group of instructions alters the sequence of program execution either conditionally or unconditionally
 - Jump
 - Call , Return and Restart
5. **Machine control operation:** These instructions control machine functions such as Halt, interrupt or do nothing.

Data transfer (copy) group instructions

- This group of instruction copy data from a source location to destination location without modifying the contents of the source.
- The transfer of data may be between the registers or between register and memory or between an I/O device and accumulator.
- Data transfer instruction do not affect flag.

1. MOV R_d, R_s : $R_d \leftarrow R_s$

- Copies data from source register to destination register
- Source Register (R_s) & Destination Register (R_d) may be A, B, C, D, E, H & L
- MOV A,B

2. MVI R, 8 bit data (Move immediate) : $R \leftarrow 8 \text{ bit data}$

- Loads the second byte (8 bit immediate data) into the register specified.
- R may be A, B, C, D, E, H & L
- MVI A,08H

3. MOV M, R (Move from register to memory) : $[HL] \leftarrow R$

- Copy the contents of the specified register to memory
- Memory (M) is the location specified by contents of the HL register pair.
- MOV M, B

4. MOV R, M (Move from memory to register): $R \leftarrow [HL]$

- Copy the contents of memory location specified by HL pair to register.
- MOV B, M

5. LXI R_p, 16 bit (Load register pair): $R_p \leftarrow 16 \text{ bit data}$

- Load immediate data to register pair
- Register pair may be BC, DE, HL & SP(Stack pointer)
- The second byte is loaded into the low order register of register pair
- The Third byte is loaded into the high order register of register pair
- LXI H, 2050H : $H \leftarrow 20$, $L \leftarrow 50$

6. MVI M, 8 bit (load 8 bit data in memory) : $[HL] \leftarrow 8 \text{ bit}$

- Loads the 8-bit data to the memory location whose address is specified by the contents of HL pair.

- MVI M, 35H : [HL]← 35H
- 7. LDA 16-bit address (Load accumulator direct): $A \leftarrow [16 \text{ bit address}]$
 - Loads the accumulator with the contents of memory location whose address is specified by 16 bit address.
 - LDA 2050H ; $A \leftarrow [2050]$
- 8. LDAX Rp (Load accumulator indirect): $A \leftarrow [Rp]$
 - Copies data byte from memory location into the Accumulator
 - The memory location is specified by the contents of registers BC or DE
 - LDAX B ; $A \leftarrow [BC]$
 - LDAX D ; $A \leftarrow [DE]$
- 9. STA 16-bit address (store accumulator direct) $[16 \text{ bit address}] \leftarrow A$
 - Stores the contents of accumulator to specified address
 - STA 2050 H ; $[2050] \leftarrow A$
- 10. STAX Rp(store accumulator indirect): $[Rp] \leftarrow A$
 - Copies data from accumulator into memory location specified by contents of either BC or DE register.
 - STAX B ; $[BC] \leftarrow A$
- 11. IN 8-bit address (load accumulator from specified port): $A \leftarrow [8 \text{ bit address}]$
 - Read data from the input port address specified in the second byte and loads data into the accumulator i. e. input port content to accumulator.
 - IN 40H ; $A \leftarrow [40H]$
- 12. OUT 8-bit address (accumulator to output port): $[8 \text{ bit address}] \leftarrow A$
 - Copies the contents of the accumulator to the output port address specified in the second byte.
 - OUT 40H ; $[40H] \leftarrow A$
- 13. LHLD 16 bit address(Load H & L register direct):
 - $L \leftarrow [16 \text{ bit address}]$, $H \leftarrow [16 \text{ bit address} + 1]$
 - Loads the contents of specified memory location to L-register and contents of next higher location to H-register.
 - LHLD 2050H ; $L \leftarrow [2050]$, $H \leftarrow [2051]$
- 14. SHLD 16-bit address (store H and L register direct):
 - $[16 \text{ bit address}] \leftarrow L$, $[16 \text{ bit address} + 1] \leftarrow H$

- Stores the contents of L register to specified memory location and contents of H register to next higher memory location.
- SHLD 5000 ; [5000] \leftarrow L , [5001] \leftarrow H

15. XCHG(Exchange H and L with D and E) :

- Content of register H are exchanged with the content of register D
- Content of register L are exchanged with the content of register E
- $H \leftrightarrow D$, $L \leftrightarrow E$

ARITHMETIC GROUP INSTRUCTIONS

- The 8085 microprocessor performs various arithmetic operations such as addition, subtraction, increment and decrement.
- The arithmetic operations are performed in relation to the content of Accumulator.
- Increment and decrement operation can be performed in any register.

1. ADD R/M

$$A \leftarrow A + R/M$$

- Adds the contents of register/memory to the contents of the accumulator and stores the result in accumulator.

○ All flags are modified

○ ADD B : $A \leftarrow A + B$

○ ADD M : $A \leftarrow A + [HL]$

2. ADI 8 bit data (add immediate to accumulator): $A \leftarrow A + 8 \text{ bit data}$

- Adds the 8 bit data with the contents of accumulator and stores result in accumulator

○ All flags are modified

○ ADI 9BH : $A \leftarrow A + 9BH$

3. SUB R/M (subtract register or memory from accumulator): $A \leftarrow A - R/M$

- Subtracts the contents of specified register/M with the contents of accumulator and stores the result in accumulator.

○ All flags are modified

○ SUB D : $A \leftarrow A - D$

4. SUI 8 bit data (subtract immediate from accumulator): $A \leftarrow A - 8 \text{ bit data}$

- Subtracts the 8 bit data from the contents of accumulator and stores result in accumulator.

○ All flags are modified

○ SUI D3H : $A \leftarrow A - D3H$

5. INR R/M(increment contents of register/Memory by 1) , DCR R/M(decrement contents of register/Memory by 1)

- Increase or decrease the contents of R (register) / M (memory) by 1 and the results are stored in same place.

○ INR B , INR M , DCR H , DCR M

○ All flags are affected except carry i.e carry flag remains unchanged.

6. INX Rp (increment register pair by 1), DCX RP (decrement register pair by 1)
 - These instructions views the contents of two registers as a 16 bit number
 - Increase and decrease the content of register pair and SP by 1.
 - Acts as 16 bit counter made from the contents of 2 registers
 - **No flags are affected**
 - INX B : $BC \leftarrow BC + 1$
 - DCX D : $DE \leftarrow DE - 1$
 - INX SP : $SP \leftarrow SP + 1$
7. ADC R/M (Add with carry) $A \leftarrow A + R/M + CY$
 - Contents of R/M and carry flag are added to the contents of accumulator and the result is placed in accumulator.
 - **All flags are modified**
 - ADC B : $A \leftarrow A + B + CY$
8. ACI 8bit data (add immediate to accumulator with carry): $A \leftarrow A + 8 \text{ bit data} + CY$
 - The 8 bit data and carry flag are added to the content of accumulator and the result is stored in accumulator.
 - **All flags are modified**
 - ACI 05H
9. SBB R/M (subtract source and borrow from accumulator): $A \leftarrow A - R/M - \text{Borrow}$
 - The contents of R/M and borrow flag are subtracted from the contents of accumulator and the result is stored in accumulator.
 - **All flags are modified**
 - SBB B : $A \leftarrow A - B - \text{Borrow}$
10. SBI 8-bit (subtract immediate with borrow): $A \leftarrow A - 8 \text{ bit} - \text{Borrow}$
 - The 8 bit data and borrow are subtracted from the contents of accumulator and the result is stored in accumulator.
 - **All flags are modified**
 - SBI 06H
11. DAD Rp (Add register pair to H and L register):
 - The 16 bit contents of the specified register pair are added to the contents of HL register and the sum is saved in HL register.

- If the result is larger than 16 bits the CY flag is set. No other flags are affected.
- DAD B : $HL \leftarrow HL + BC$

12. DAA(Decimal adjust accumulator):

- It adjusts an 8 bit number in the accumulator to form two BCD numbers
- Works only with addition when BCD numbers are used; does not work with subtraction.

LOGICAL GROUP INSTRUCTIONS

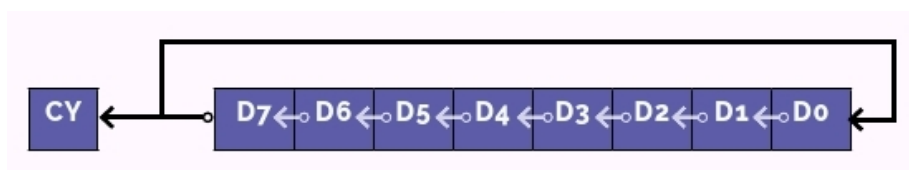
1. ANA R/M , ANI 8-bit data: $A \leftarrow A \& R/M$, $A \leftarrow A \& 8 \text{ bit data}$
 - Logically AND the contents of register/memory or 8bit data with the contents of accumulator and the result is placed in accumulator.
 - CY flag is reset and AC is set and others as per result.
 - ANA B : $A \leftarrow A \& B$, ANI 0FH: $A \leftarrow A \& 0FH$
2. ORA R/M and ORI 8-bit data: $A \leftarrow A | R/M$, $A \leftarrow A | 8 \text{ bit data}$
 - Logically OR the contents of register/memory or 8bit data with the contents of accumulator and the result is placed in accumulator.
 - CY and AC is reset and others as per result.
 - ORA C : $A \leftarrow A | C$, ORI FFH : $A \leftarrow A | FFH$
3. XRA R/M and XRI 8-bit data: $A \leftarrow A \wedge R/M$, $A \leftarrow A \wedge 8 \text{ bit data}$
 - Logically XOR the contents of register/memory or 8bit data with the contents of accumulator and the result is placed in accumulator.
 - CY and AC is reset and others as per result.
 - XRA B : $A \leftarrow A \wedge B$
4. CMA (complement accumulator): $A \leftarrow \sim A$
 - Complements the contents of the accumulator.
 - No flags are affected.
5. STC (set carry flag)
 - Sets carry flag.

FLAG STATUS		
Instructions	CY	AC
ANA/ANI	0	1
ORA/ORI	0	0
XRA/XRI	0	0

6. CMP R/M(compare with accumulator) and CPI 8 bit data (compare immediate with accumulator) : $A - R/M$, $A - 8 \text{ bit data}$
- Compare the contents of register/ memory and 8 bit data with the contents of accumulator.
 - **After comparison contents of A,R/M are not modified**
 - Status is shown by CY and Z flags
 - S,P and AC are also modified with result.

FLAG STATUS		
CASE:	CY	Z
$A < [R/M] \text{ or } 8 \text{ bit}$	1	0
$A = [R/M] \text{ or } 8 \text{ bit}$	0	1
$A > [R/M] \text{ or } 8 \text{ bit}$	0	0

7. RLC (Rotate accumulator left)
- Each binary bit of accumulator is rotated left by one position.
 - Bit D_7 is placed in the position of D_0 as well as in carry flag.
 - The carry flag is modified according to D_7 and remaining flags are not affected.



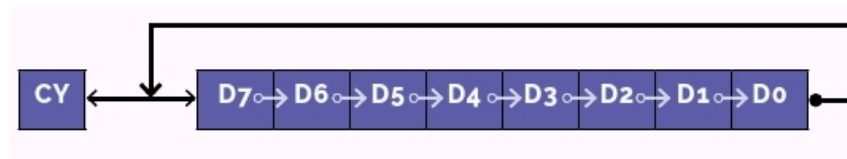
8. RAL (Rotate accumulator left through carry)

- Each bit of accumulator is rotated left by one position through the carry flag.
- Bit D_7 is placed in the bit in carry flag and the carry flag is placed in the bit D_0 .
- The carry flag is modified according to D_7 and remaining flags are not affected.



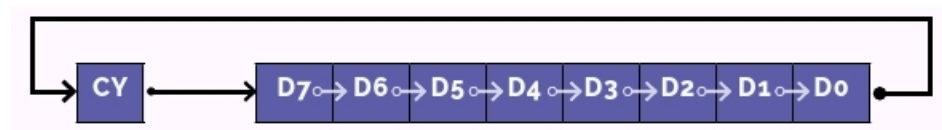
9. RRC (rotate accumulator right):

- Each binary bit of accumulator is rotated right by one position.
- Bit D_0 is placed in the position of D_7 as well as in the carry flag.
- The carry flag is modified according to D_0 and other flags are not affected.



10. RAR (Rotate accumulator right through carry):

- Each bit of accumulator is rotated right by 1 position through the carry flag.
- Bit D_0 is placed in carry flag and the bit in carry flag is placed in D_7 .
- The carry flag is modified according to D_0 and other flags are not affected



BRANCHING GROUP INSTRUCTIONS

The microprocessor is a sequential machine; it executes machine codes from one memory location to the next. The branching instructions instruct the microprocessor to go to a different memory location and the microprocessor continues executing machine codes from that new location.

The branching instructions are the most powerful instructions because they allow the microprocessor to change the sequence of a program, either unconditionally or under certain test conditions. The branching instruction code categorized in following three groups:

- Jump instructions
- Call and return instruction
- Restart instruction

Jump instructions

- The jump instructions specify the memory location explicitly.
- They are 3 byte instructions, one byte for the operation code followed by a 16 bit (2 byte) memory address.
- Jump instructions can be categorized into unconditional and conditional jump.

Unconditional Jump: **JMP 16 bit address**

- **Used to set up continuous loops.**
- Can also be specified either 16 bit address or using a label or name
 - JMP 2000H
 - JMP L1

Using address:

0800H : MVI A,20H

0802H: MVI B,01H

0804H: ADD B

0805H: JMP 0800H

0808H: STA 7070H

080BH: HLT

Using label:

START: MVI A,20H

MVI B,01H

ADD B

JMP **START**

STA 7070H

HLT

Conditional Jumps:

- The conditional jump instructions allow the microprocessor to make decisions based on certain conditions indicated by the flags.
- After logic and arithmetic operations, flags are set or reset to reflect the condition of data.
- These instructions check the flag conditions and make decisions to change or not to change the sequence of program
- The four flags namely carry, zero, sign and parity used by the jump instruction.

JC 16 bit address	Jump on carry (if CY=1)
JNC 16 bit address	Jump on no carry (if CY=0)
JZ 16 bit address	Jump on zero (if Z=1)
JNZ 16 bit address	Jump on no zero (if Z=0)
JP 16 bit address	Jump on plus(S=0)
JM 16 bit address	Jump on minus (S=1)
JPE 16 bit address	Jump on Even parity (P=1)
JPO 16 bit address	Jump on odd parity(P=0)

- WAP to subtract two 8 bit number. Store FFH in location 5000H if the result is 0 otherwise store AAH in location 5001H.

MVI A,32H

MVI B,03H

SUB B

JZ L1

LXI H,5001H

MVI M,AAH

HLT

L1: LXI H,5000H

MVI M,FFH

HLT

Stack

- The stack is defined as a set of memory location in R/W memory, specified by a programmer in a main memory. These memory locations are used to store binary information temporarily during the execution of a program.
- The beginning of the stack is defined in the program by using the instruction LXI SP, 16 bit address.
- Storing of data bytes begins at memory address that is one less than the address in SP

PUSH Rp: (store register pair on stack):

- Copies the contents of specified register pair or program status word (accumulator and flag) on the stack.
- PUSH B , PUSH D, PUSH H , PUSH PSW
- **How PUSH works:**
 - $SP \leftarrow SP - 1$
 - $[SP] \leftarrow \text{Register higher}$;(B,D,H,A)
 - $SP \leftarrow SP - 1$
 - $[SP] \leftarrow \text{Register lower}$;(C,E,L,flags)
- PUSH B : $[SP-1] \leftarrow B$, $[SP-2] \leftarrow C$
- After PUSH instruction SP register is decremented by 2.
- An address in SP register indicates that the next two memory location (in descending numerical order) can be used for storage.

POP Rp: Retrieve register pair from stack

- Copies the content of the top two memory locations of the stack into specified register pair.
- POP B , POP D, POP H, POP PSW
- How POP works
 - $\text{Register Lower} \leftarrow [SP]$
 - $SP \leftarrow SP + 1$
 - $\text{Register higher} \leftarrow [SP]$
 - $SP \leftarrow SP + 1$
- POP D : $E \leftarrow [SP]$, $D \leftarrow [SP + 1]$
- After POP instruction SP is incremented by 2.

```
LXI SP, 2099H
LXI H, 42F2H
PUSH H
POP B
HLT
```

```
LXI SP, 2404H
LXI H, 2150H
LXI B, 2280H
PUSH H
PUSH B
POP B
POPH
HLT
```

CALL and RET

- A subroutine is a group of instruction written separately from main program to perform a function that occurs repeatedly in the main program.

Unconditional CALL and RET

- CALL(call a subroutine) and RET(return to main program from a subroutine) are used to implement subroutine
- CALL is used in main program to call a subroutine
- RET is used at the end of subroutine to return to the main program.

CALL (Call subroutine unconditionally): CALL 16 bit address/Label

1. When a subroutine is called the, the contents of the PC (address of instruction following CALL instruction) is stored in stack.
2. Decrements the SP by two
3. Program execution is transferred to the subroutine address unconditionally.

RET (Return from subroutine unconditionally)

- When RET instruction is executed at the end of the subroutine
 - i. the memory address stored in the stack is placed in PC
 - ii. SP is incremented by 2
 - iii. Unconditionally returns from a subroutine

2000H	LXI SP, 2400H

2040H	CALL 2070H
2043H	Next instruction

205FH	HLT
2070H

207FH	RET

1. After fetching all bytes of CALL 2070H, PC register contains 2043H and SP register contains 23FFH
2. In the memory location 23FFH of stack 20H (PCH) is stored and SP register is decremented by 1 and it contains 23FEH.
3. In the memory location 23FEH of stack 43H(PCL) is stored

Write an ALP to add two numbers using subroutines and store result in location 2050H.

0800H	LXI SP, 000AH
0803H	MVI B, 22H
0805H	MVI C, 01H
0807H	CALL L
080AH	STA 2050H
080DH	HLT
080EH	L: MOV A, B
080FH	ADD C
0810H	RET

Conditional CALL and Return

Conditional Call: Calls subroutine only if the condition is satisfied otherwise the main program is continued.

CC 16 bit address	Call on carry (CY=1)
CNC 16 bit address	Call with no carry (CY=0)
CP 16 bit address	Call on positive(S=0)
CM 16 bit address	Call on minus (S=1)
CPE 16 bit address	Call on parity even (P=1)
CPO 16 bit address	Call on parity odd (P=0)
CZ 16 bit address	Call on zero (Z=1)
CNZ 16 bit address	Call on no zero (Z=0)

Conditional Return:

- Returns to calling program only if the condition is satisfied
- RC, RNC, RZ, RNZ, RP, RM, RPE, RPO

RC	Return on carry (CY=1)
RNC	Return with no carry (CY=0)
RP	Return on positive(S=0)
RM	Return on minus (S=1)
RPE	Return on parity even (P=1)
RPO	Return on parity odd (P=0)
RZ	Return on zero (Z=1)
RNZ	Return on no zero (Z=0)

Restart Instruction

- RST instructions are 1 byte Call instructions that transfer the program execution to a specific location.
- They are executed the same way as Call instructions.
- When RST instruction is executed, the 8085 stores the contents of PC (address of next instruction) on the top of the stack and transfers the program to the Restart location.

Restart instruction	Call location in HEX
RST 0	0000H
RST 1	0008H
RST 2	0010H
RST 3	0018H
RST 4	0020H
RST 5	0028H
RST 6	0030H
RST 7	0038H

Machine Control instructions

- These instructions affect the operation of processor
1. HLT : Halt and enter wait state
 - CPU finishes executing current instruction and stops further execution.
 2. NOP: No operation
 - No operation is performed
 - The instruction is fetched and decoded
 3. DI
 4. EI
 5. SIM
 6. RIM

MISCELLANEOUS GROUP INSTRUCTIONS

1. XTHL
2. PCHL : Load PC with HL contents
 - Contents of H and L are copied into PC
 - Content of H are placed as a higher order byte
 - Content of L are placed as a Low order byte
3. SPHL:

Looping

- The programming technique used to instruct the microprocessor to repeat task is called looping.
- Loops can be
 - Continuous loop
 - Conditional loop

Continuous Loop:

- Repeats a task continuously
- It is set up by using unconditional jump instruction
- A continuous loop does not stop until the system is reset.

START: MVI A,20H

MVI B,01H

ADD B

JMP START

HLT

Conditional Loop:

- Repeats a task until some data conditions are met
- It is set up by using conditional jump instruction
- To set up a counter using conditional loop
 - A register is used as counter and an appropriate value is loaded into that register
 - Counting is performed by either incrementing or decrementing the counter
 - Loop is set up by conditional jump instruction
 - End of counting is indicated by a flag

Example :WAP to repeat the addition of 02H and 01H five times

MVI C,05H

LOOP: MVI A,02H

MVI B,01H

ADD B

DCR C

JNZ LOOP

HLT

- **Advanced Microprocessors and peripherals :**
K M Bhurchandi , A K Ray
- **IBM PC Assembly language and programming :**
Peter Abel

The diagram illustrates the internal architecture of the 8086 microprocessor, divided into two main functional units: the **BIU (Bus Interface Unit)** and the **EU (Execution Unit)**.

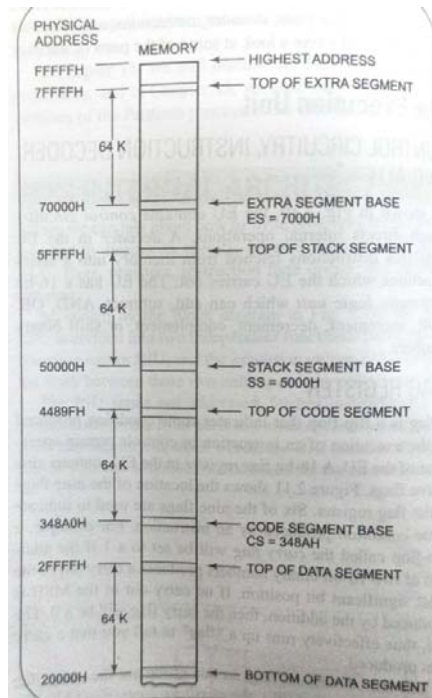
- BIU (Bus Interface Unit):**
 - Contains the **Instruction Stream Byte Queue** (a 6-byte queue numbered 1 to 6).
 - Contains the **Segment Registers** (ES, CS, SS, DS, IP).
 - Contains the **Instruction Pointer (IP)**.
 - Interfaces with the **Memory Interface** via the **C-BUS**.
 - Communicates with the **EU** via the **B-BUS**.
- EU (Execution Unit):**
 - Contains the **General Purpose Registers** (AH, BH, CH, DH, AL, BL, CL, DL, SP, BP, SI, DI).
 - Contains the **Arithmetic Logic Unit (ALU)**.
 - Contains the **Operands Flags**.
 - Communicates with the **BIU** via the **B-BUS**.
 - Communicates with the **Control System** via the **A-BUS**.
 - Communicates with the **Memory Interface** via the **C-BUS**.
- Control System:**
 - Manages the flow of data and instructions between the BIU and EU.
 - Controls the **Instruction Stream Byte Queue**.
 - Controls the **Arithmetic Logic Unit**.
 - Controls the **Operands Flags**.
- Buses:**
 - B-BUS:** Connects the BIU and EU.
 - C-BUS:** Connects the BIU and EU to the Memory Interface.
 - A-BUS:** Connects the EU to the Control System.

- 8086 is a 16 bit microprocessor
- 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
- 20 bit address bus
- It can address up to 1 MB memory location [$2^{20}=1048576=1\text{M}$]
- Divided internally into two separate unit:
 - Bus Interface Unit (BIU):
 - Delivers instructions and data to EU.
 - Prefetch instructions so that there is always a queue of instruction ready to execute.
 - Execution Unit (EU):
 - Tells BIU where to fetch instructions and data from
 - Decodes instructions
 - Executes instructions that have already been fetched.

Bus interface unit (BIU)

- The BIU sends out addresses, fetches instructions from memory, reads data from memory or ports and writes data to memory or ports. So it handles all transfers of data and address on the buses for EU.
- It delivers data and instruction to EU
- BIU contains
 - Instruction pointer
 - 4 segment registers
 - Instruction queue
 - Address generation circuit
- **Instruction Queue :**
 - The BIU can store up to 6 bytes of instructions with FIFO (First in First Out) manner in a register set called a queue.
 - While EU is decoding an instruction or executing an instruction which does not require use of busses, the BIU fetches up to six instruction bytes.
 - When EU is ready for next instruction, it simply reads the instruction byte(s) from the queue in the BIU.
 - This is done in order to speed up program execution by overlapping instruction fetch with execution. This mechanism is known as pipelining.
- **Segments**
 - A segment may be located almost anywhere in the memory and may be up to 64 K bytes , it requires only as much space as the program requires for its use.
 - At any given time 8086 works with only four 64 K bytes segments within its 1 M byte range.
 - Segments
 - **Code segment:** contains machine instructions that are to be executed.
 - **Data segment:** contains a program's defined data, constants and work areas.
 - **Stack segment:** Contains any data and address that the program needs to save temporarily.
 - **Extra segment:** It also refers to a segment which essentially is another data segment.
 - A segment may be up to 64 KB.

- A segment will always start at an address with zeros in the lowest 4 bits.



- **Segment Register**

- Four segment registers in the BIU are used to hold the upper 16 bits of the starting address of four memory segments that the 8086 is working at a particular time.
- **Code segment register (CS):**
 - The CS contains upper 16 bits of starting address of program's code segment i.e it points to the base or start of code segment.
 - **Instruction pointer register (IP):**
 - The 16 bit IP register contains the offset address of the next instruction that is to execute.
 - IP is associated with CS register as CS:IP
 - Address of next instruction is = $CS \times 10H + IP$
- **Data segment register (DS):**
 - The 16 bit DS register contains upper 16 bits of the starting address of data segment
 - Instruction use this address to locate data in data segment.
 - Actual location of data = $DS \times 10H + \text{Offset}$

- **Stack segment register (SS):**

- The 16 bit SS register contains upper 16 bits of the starting address of stack segment
- **Stack pointer (SP) register:**
 - The 16 bit SP register holds the 16 bit offset from the start of the stack segment to current top of the stack.
 - PUSH instruction decrements SP register by 2
 - POP instruction increments SP register by 2.
 - Physical address of current top of the stack = $SS \times 10H + SP$.

- **Extra segment register(ES):**

- The 16 bit ES register contains upper 16 bits of the starting address of Extra segment.
- Used by some string (character data) operation to handle memory addressing.

Execution unit (EU)

- Execution Unit (EU):
 - Tells BIU where to fetch instructions and data from
 - Decodes instructions
 - Executes instructions that have already been fetched.
- The EU contains arithmetic and logic (ALU), a control unit, and a number of registers.
- It has nine 16 bit registers which are AX, BX, CX, DX, SP, BP, SI, DI and flag.
- First four can be used as 8 bit register (AH, AL, BH, BL, CH, CL, DH, DL)

AX Register:

- It is used for operations involving input/output and most arithmetic.
- The lower 8 bit of AX is designated as AL and higher 8 bits as AH.
- AX register is called 16 bit accumulator and AL is called 8 bit accumulator.

BX Register

- BX is known as the **base register** since it is the only general purpose register that can be used to store offset address in some addressing mode.
- The lower 8 bit of BX is designated as BL and higher 8 bits as BH
- **BX can also be combined with DI or SI as base register** for special addressing
- It can be used for computation

CX Register

- **CX register is known as the counter register.**
- The lower 8 bit of CX is designated as CL and higher 8 bits as CH
- It may contain
 - A value to control the no. of times a loop is repeated
 - A value to shift bits left of right
- Some instructions such as SHIFT, ROTATE, LOOP use contents of CX as a counter.

DX Register

- The DX register is known as **data register**
- The lower 8 bit of DX is designated as DL and higher 8 bits as DH
- **Some input/output operation requires its use**
- **Multiply and divide operations that involve large values assume the use of DX and AX together as a pair.**

Stack pointer (SP) register:

- The 16 bit SP register provides an offset value, which, when associated with the SS register, refers to the current word being processed in the stack
- The SP's contents are automatically updated (increment/decrement) during execution of a POP and PUSH instructions

Base pointer (BP) Register

- 16 bit register
- The BP facilitates referencing data and addresses that a program passes via stack.
- The processor combines the address is SS with the offset in BP.
- BP can also be combined with DI and with SI as a base register for special addressing

Index register

- There are two 16 bit index register **SI (Source index) register** and **DI (Destination Index) register**.
- They are used as general purpose register as well as for offset storage in case of some addressing mode.
- The instructions that process data strings use the SI and DI index register together with DS and ES respectively, in order to distinguish between the source and destination address.

Flag Register

D ₁₅	D ₁₄	D ₁₃	D ₁₂	D ₁₁	D ₁₀	D ₉	D ₈	D ₇	D ₆	D ₅	D ₄	D ₃	D ₂	D ₁	D ₀
				O	D	I	T	S	Z		AC		P		CY

- It is 16 bit register
 - Status flags : 6 flags
 - Machine control flags : 3 flag
- **Status Flag**
 - **O-overflow flag :**
 - This flag is set if an arithmetic overflow occurs, i.e. if the result of a signed operation is large enough to be accommodated in a destination register.
 - For example, if the result of addition of two 8 bit signed number is of more than 7 bit in size in case of 8 bit signed operation or more than 15 bits in size in case of 16 bit signed operation, then the overflow flag will be set.
 - **S-Sign flag:**
 - This flag is set when the result of any computation is negative.
 - For signed computations, the sign flag equals the MSB of the result
 - **Z-zero flag:**
 - This flag is set when the result of an arithmetic or logical operation is zero.
 - **AC-Auxiliary carry flag:**
 - This is set if there is a carry from the lowest nibble, i.e. bit three during the addition or borrow for the lowest nibble i.e. bit three, during subtraction.
 - **P-Parity flag:**
 - This flag is set to 1 if the lower byte of the result contains even number of 1s otherwise reset.
 - **C-Carry flag:** This flag is set when there is a carry out of MSB in case of addition or a borrow in case of subtraction.

- **Control flag**
 - **D-Direction Flag:**
 - This is used by string manipulation instructions.
 - If this flag bit is '0', the string is processed beginning from the lowest address to the higher address, i.e. auto incrementing mode otherwise the string is processed from the highest address towards the lowest address, i.e. auto-decrementing mode
 - **I-Interrupt flag:** If this flag is set the maskable interrupts are recognized by the CPU, otherwise they are ignored.
 - **T- Trap flag:** If this flag is set the processor enters the single step execution mode i.e one instruction at a time under user control.

Control Circuitry, Instruction Decoder and ALU

- Control circuitry directs internal operation.
- Instruction decoder translates the instructions fetched from memory into a series of actions which the EU carries out.
- The 16 bit ALU performs Arithmetic and Logical operations.

Physical address

- Physical address is actual 20 bit address of memory location
- To reference any memory location in a segment, the processor combines the **segment address in segment register** with offset value of that location ,that is , its distance in bytes from start of the segment
- The 2 byte offset can range from 0000H to FFFFH.
- The physical address of any memory location is calculated using segment address and offset.

$$\text{Segment address: offset} = \text{Segment address} \times 10\text{H} + \text{offset}$$

- To implement above formula the content of the segment register is shifted left bit wise 4 times and to this result, the offset is added.

Segment address	1005H	0001 0000 0000 0101
shifting of segment address		0001 0000 0000 0101 0000
		+
offset	5555H	0101 0101 0101 0101
physical address	155A5	0001 0101 0101 1010 0101

- For code segment , **CS:IP**
- For data segment **DS:BX , DS:SI , DS:DI**
- For stack segment **SS:SP , SS:BP**
- For extra segment **ES:DI**

Addressing Modes in 8086

Register addressing:

- In this addressing the register may appear in the first operand, the second operand or both.

MOV BX, WORD_MEM ; Register in first operand

MOV WORD_MEM, AX ; Register in second operand

ADC AL, BL ; Register in both operands.

Immediate addressing:

- In this type of addressing, immediate data is a part of instruction, and appears in the form of successive byte or bytes.

MOV AX,0005H

MOV BL,06H

Direct memory addressing:

- In this format, one of the operand references a memory location and other operand references a register.
 - BYTE_VAL DB 32H
ADD BYTE_VAL, DL
 - MOV BH, BYTE_VAL
- MOV AX, [5000H] ;[5000] is offset of data in data segment.
- Here Physical address for [5000H]= 10HxDS+5000H

Direct offset addressing:

- This addressing mode, a variation of direct addressing, uses **arithmetic operators to modify an address.**

ARR DB 15, 17, 18, 21

MOV AL, ARR [2] ; it is same as MOV AL, 18; here ARR [0] =15, ARR [1] =17

MOV AL, ARR+2 ; same as above instruction

Indirect memory addressing:

- Indirect addressing takes advantage of computer's capability for segment: offset addressing.
- The registers used for this purpose are base register (BX and BP) and index register (DI and SI) **coded with square brackets**, which indicate a reference to memory.
- [SI], [DI], [BX] and [BP] are indirect address as it contains the offset address.
- BX, DI and SI are associated with DS as **DS: BX, DS: DI, DS: SI** for processing data in data segment.
- BP is associated with SS as **SS: BP** for handling data stack.
- **When first operand is indirect address , the second operand references a register or immediate value**
- **When second operand contains indirect address, the first operand references a register.**

Example 1:

MOV [BX], CL ; move CL to address DS:BX (Data)

Example 2:

ADD CL, [SI] ; ADD CL and DS:SI

Example 3

ADD [BP], CL ; add CL and SS:BP

Base displacement addressing:

- This addressing mode also uses base registers (BX and BP) and index register (SI and DI), but combined with a displacement (a number or offset value) to form an effective address.

MOV AL,[BX+2] ; BX offset plus 2

MOV CL,[SI+0100H];

Base index addressing:

- This addressing mode combines a base registers (BX or BP) with an index register (SI or DI) to form an effective address.

MOV AX,[BX+SI] , MOV [BX+DI],CL

Base index with displacement addressing:

- **This** addressing mode, a variation on base- index, combines a base register, an index register, and a displacement to form an effective address.

MOV AX,[BX+DI+10] ; or 10[BX+DI]

Assembly language programming

There are two types of programming language

- High level language :
 - They are machine independent language
 - Examples are C , Basic
 - They use powerful commands, each of which may generate many machine language instructions.
 - Uses compiler program to translate source code into machine code(technically, object code)
- Low level language:
 - Assembly language is considered as low level language.
 - They are microprocessor dependent
 - Each symbolic instruction generate one machine language instruction.
 - Uses assembler program to translate source code in to object code.

Advantage of assembly language programming

- They generate small and compact execution module.
- They have more control over hardware.
- Results in faster execution

Disadvantages of Assembly language programming

- Machine dependent.
- Lengthy code
- Error prone (likely to generate errors).

Assembly language features

1. Program Comments

- The use of comments throughout a program can improve its clarity.
- It starts with **semicolon (;)** and the assembler assumes that all characters on the line to its right are comments.
- Comment may contain any printable character , including blank
- E.g. ADD AX, BX ; Adds AX & B

2. Reserved word

- Certain names in assembly language are reserved for their own purpose
- Categories of reserved word
 - **Instructions:** MOV, ADD (operations that the computer can execute).
 - **Directives :** Such as END, SEGMENT (information to assembler)
 - **Operators:** Such as AND, OR, XOR, NOT, FAR, SIZE (operates on operand)
 - **Predefined symbols:** such as @DATA, @Model, which return information to your program during assembly

3. Identifiers

- An identifier (or symbol) is a name that applies to an item in the program that you expect to reference.
- Types
 - **Name :** Name refers to the address of a data item such as COUNTER in COUNTER DB 0
 - **Label:** Label refers to the address of an instruction , procedure or segment such as MAIN and B30 in following example
MAIN PROC FAR
B30: ADD BL, 25
- An identifier can use following character
 - Alphabet : A to Z and a to z
 - Digits : 0 to 9 (not first character)
 - Special character : @,\$,_ ,

- dot (.) cannot be first character
- The first character of identifier must be an alphabetic letter or a special character, except for dot(.)

4. Statements

- ALP consists of a set of statements with two types
- Types
 - **Instructions:** assembler translates instructions to object code.
 - **Directives:** it tells assembler to perform a specific action, such as define a data item.

	<i>[identifier]</i>	<i>operation</i>	<i>[operands]</i>
Directives	Count	DB	1
instruction	L30:	MOV	AX,0

[] : indicates optional entry

- The *operation* is most commonly used for defining data areas and coding instruction
- The *operand* (if any) provides information for the operation to act on.
 - For data item *operand* defines its initial value
 - For instruction *operand* indicates where to perform the action.

5. Directives

- An assembler directive is a statement to give direction to the assembler to perform task of the assembly process.
- Directives act only during the assembly of a program and generate no machine-executable code.

PAGE and TITLE listing directives:

- The page and title directives help to control the format of a listing of an assembled program

PAGE directive:

- Defines the maximum number of lines to list on a page and the maximum number of characters as a line.
- Its format is: `PAGE [length] [, width]`
- `PAGE 60,132`
- **Length 60:** when assembler is printing the assembled program and has listed 60 lines, it automatically advances to the top of next page and increments the page count
- **Width 132:** 132 character per line

TITLE directive: cause a title for a program to print on line 2 of each page of program

Listing.

```
TITLE text [comment] ; comment is optional  
TITLE A04ASM1
```

SEGMENT directive

- The directives for defining a segment ,SEGMENT and ENDS have following format

```
Segment-name SEGMENT [align] [combine] ['class']  
.....  
Segment-name ENDS
```

- **SEGMENT** statement defines the start of a segment.
- **Segment-name** must be present, must be unique and must follow assembly language naming conventions.
- **An ENDS** statement indicates the end of the segment and contains the same name as the SEGMENT statement

- **Align option** indicates the boundary on which the segment is to begin; **PARA** is used to align the segment on paragraph boundary so that the starting address is evenly divisible by 16 or 10H.
- **Combine option** indicates whether to combine the segment with other segments when they are linked after assembly. STACK, COMMON, PUBLIC, etc are combine types.
- The stack segment is commonly defined as

Segment-name SEGMENT PARA STACK
- **Class option**, enclosed in apostrophes, is used to group related segments when linking.
- The class 'code' for code segment, 'stack' for stack segment and 'data' for data segment is used.

```

TITLE      page    60,132
           A04ASM1  Segments for an .EXE Program
; -----
STACK      SEGMENT PARA STACK 'Stack'
           ...
STACK      ENDS
; -----
DATASEG    SEGMENT PARA 'Data'
           ...
DATASEG    ENDS
; -----
CODESEG    SEGMENT PARA 'Code'
           MAIN    PROC FAR
           ...
MAIN       ENDP
CODESEG    ENDS
           END      MAIN
;End of procedure
;End of segment
;End of program

```

PROC Directive

- The code segment contains the executable code for a program, which consists of one or more procedures, defined initially with the PROC directives and ended with the ENDP directive.

Procedure-name PROC FAR/NEAR

.....

Procedure-name ENDP

- Procedure-name** must be present, must be unique and must follow assembly language naming conventions
- FAR** is related to program execution. The procedure with FAR is entry point for first instruction to execute.
- The **ENDP** directive indicates the end of a procedure and contains the same name as the PROC statement.
- The code segment may contain any number of procedure used as subroutines. Each additional PROC is coded with NEAR operand.

END Directive

- An END directive ends the entire program and appears as the last statement.

END [procedure-name]

- In most program procedure-name is the name of the PROC designated as FAR

ASSUME Directive

- An .EXE program uses the SS register to address the stack segment, DS to address the data segment, CS to address the code segment and ES to address extra segment.
- ASSUME** directive is used to tell the assembler the purpose of each segment in the program. It is used in code segment.

ASSUME SS:stackname, DS:datasegname , CS:codesegname

- DS:datasegname** tells the assembler that segment named **datasegname** should be treated as data segment.
- ASSUME may also contain ES:datasegname for extra segment and if program doesnot use extra segment then we omit ES:datasegname


```

1      page      60,132
2  TITLE  A04ASM1 Skeleton of an .EXE Program
3  ; -----
4  STACK  SEGMENT PARA STACK 'Stack'
5      ...
6  STACK  ENDS
7  ; -----
8  DATASEG SEGMENT PARA 'Data'
9      ...
10 DATASEG ENDS
11 ; -----
12 CODESEG SEGMENT PARA 'Code'
13 MAIN    PROC    FAR
14          ASSUME  SS:STACK,DS:DATASEG,CS:CODESEG
15          MOV     AX,DATASEG      ;Set address of data
16          MOV     DS,AX          ; segment in DS
17          ...
18          MOV     AX,4C00H        ;End processing
19          INT     21H
20 MAIN    ENDP          ;End of procedure
21 CODESEG ENDS          ;End of segment
22          END      MAIN          ;End of program

```

CS Scanned with CamScanner

Processor Directive

- Most assemblers assume that the source program is to run on a basic 8086 level computer.
- As a result, when you use instructions or features introduced by later processor, you have to notify the assembler by means of a processor directives such as .286 , .386, .486 etc.

Dn Directives for defining data

- Dn directive is used to define data types.

[name] Dn expression

- A program that references a data item uses *name*. The *name* is otherwise optional.

Dn Directives		
DB	Define byte	1 byte
DW	Define word	2 byte
DD	Define doubleword	4 bytes
DF	defined farword	6 bytes
DQ	Define quadword	8 bytes
DT	Define tenbytes	10 bytes

- expression* may specify an uninitialized value or a constant value

DATA DB ? ; Uninitialized item

VAL1 DB 25 ; initialized item

ARR DB 21, 23, 27, 53; ARR references 21, ARR+1 references 23

EQU Directive

- It can be used to assign a name to constants
- It is short form of equivalent.

E.g. **FACTOR EQU 12**

MOV BX, FACTOR ; MOV 12 to BX

DUP DIRECTIVE

- It can be used to initialize several locations to an initial value.

INPUT DB 4 DUP(0) - It reserves 4 bytes starting at the offset INPUT in DS and initializes them to 0.

- Also used to reserve several locations that need not be initialized. In this case (?) is used with DUP directives.

E. g. **PRICE DB 100 DUP(?)** - Reserves 100 bytes of uninitialized data space to an offset PRICE

Program written in Conventional full segment directive

PAGE 60,132

TITLE SUM program to add two number

;.....

STACK SEGMENT PARA STACK 'STACK'

DW 32 DUP(0)

STACK ENDS

;.....

DATASEG SEGMENT PARA 'DATA'

FLDD DW 215

FLDE DW 125

FLDF DW ?

DATASEG ENDS

;.....

CODESEG SEGMENT PARA 'CODE'

MAIN PROC FAR

ASSUME SS:STACK,DS:DATASEG,CS:CODESEG

MOV AX,DATASEG ;get address of data segment

MOV DS,AX ; store address in DS

MOV AX,FLDD

ADD AX,FLDE

MOV FLDF,AX

MOV AX,4C00H ; End processing

INT 21H

MAIN ENDP ; end of procedure

CODESEG ENDS ; end of segment

END MAIN ; end of program

Simplified Segment directive

- The assembler provides some short cuts in defining segments.
- To use shortcuts we have to initialize memory model before defining any segment.
- The different models tell assembler how to use segments, to provide enough space for the object code and to ensure optimum execution speed.

Model	Number of code segment	Number of data segment
Tiny	Data , code and stack in one 64K segment	
Small	1 <=64K	1 <=64K
Medium	Any number , any size	1 <=64K
Compact	1 <=64K	Any number , any size
Large	Any number , any size	Any number , any size
Huge	Any number , any size	Any number , any size

- The Huge model is same as large , but may contain variables such as arrays greater than 64K.
- Format to define memory model ; `.MODEL memory-model`
Eg.; `.MODEL SMALL`
- `.MODEL` directives automatically generates the required `ASSUME` statement for all model.
- Format for directive that define stack, data and code segment are
`.STACK [SIZE in bytes]`
`.DATA`
`.CODE`

- We initialize the address of data segment in DS using following code

```
MOV AX,@data
```

```
MOV DS,AX
```

Program Structure using simplified directive

```
PAGE 60,132
```

```
TITLE SUM program to add two number
```

```
;.....
```

```
.MODEL SMALL
```

```
.STACK 64 ;define stack
```

```
.DATA ; define data
```

```
; .....
```

```
.CODE ;define code segment
```

```
MAIN PROC FAR
```

```
MOV AX,@data ;set address of data segment
```

```
MOV DS,AX
```

```
.....
```

```
.....
```

```
.....
```

```
MOV AX,4C00H ;end processing
```

```
INT 21H
```

```
MAIN ENDP ;end of procedure
```

```
END MAIN ;end of program
```

PAGE 60,132

TITLE SUM program to add two number

;.....

.MODEL SMALL

.STACK 64 ;define stack

.DATA ; define data

FLDD DB 10

FLDE DB 5

FLDF DB ?

;

.CODE ;define code segment

MAIN PROC FAR

MOV AX,@data ;set address of

MOV DS,AX ;data segment

MOV AH,FLDD

ADD AH, FLDE

MOV FLDF ,AH

MOV AX,4C00H ;end processing

INT 21H

MAIN ENDP ;end of procedure

END MAIN ;end of program

Programmer's model of 8086

AH	AL	Accumulator	IP	instruction pointer
BH	BL	base index	flags	flags
CH	CL	Count		
DH	DL	Data	CS	Code
SP		stack pointer	DS	data
BP		base pointer	ES	extra
DI		destination index	SS	stack
SI		source index		

Data Formats

Data are presented in 8086 microprocessor as

- ASCII
- BCD
- Byte
- Word
- Double word
- Real numbers.

Instructions in 8086

1. MOV register/memory, register/memory/immediate

- BYTEFIELD DB ?

MOV BYTEFIELD,0DH

- MOV AX,BX

MOV AH, BH

MOV [DI], BX

MOV AH, 40H

2. XCHG register/memory, register/memory

- It swaps two data items

- WORDQ DW 1122H

XCHG CX, WORDQ ; exchange content of register and memory

XCHG CL,BH ;exchange contents of registers

3. LEA register, memory

- Used to initialize a register with an offset address

- MYNAME DB 11, 12

LEA BX, MYNAME ; load offset address of MYNAME to BX

4. INC/DEC register/memory

5. ADD/SUB register/memory , register/memory/immediate

ADD AX, BX

ADD AH, [SI]

6. JMP address/label

7. LOOP address/label [store initial value in CX]

MOV CX,8

A20: INC AX

ADD AX,BX

LOOP A20

8. CMP register/memory, register/memory/immediate

operand 1 < operand 2 CF=1 SF=1 ZF=1

operand 1 < operand 2 CF=0 SF=0 ZF=1

operand 1 < operand 2 CF=0 SF=0 ZF=0

9. Jump for CMP

- JE/JZ jump equal or jump zero
- JNE/JNZ jump not equal or jump not zero
- JA/JNBE jump above or jump not below /equal
- JAE/JNB jump above/equal or jump not below
- JB/JNAE jump below or jump not above/equal
- JBE/JNA jump below/equal or jump not above

10. Conditional jump :JC ,JNC,JO,JNO,JPE,JPO,JS,JNS

11. AND/OR/XOR register/memory, register/memory/immediate

12. NOT register/memory

13. SHR,SAR,SHL,SAL,SHLR

14. ROR,RCR,ROL,RCL

15. MUL register₈/memory₈

- Multiplicand is in AL
- 16 bit Result is in AX

16. MUL register₁₆/memory₁₆

- Multiplicand is in AX
- 32 bit Result is stored in DX(higher order) and AX(lower order)

17. DIV register/memory

Dividing word by byte

- AX: 2 byte dividend
- Divisor :1 byte in register/memory
- AH: remainder
- AL: quotient

Video and Keyboard Processing

- Most of the program require input from an external source like keyboard ,mouse ,modem and must provide output in a useful format on a screen, printer etc.
- The INT (interrupt) instruction handles input and output for most purpose.
 - INT 10H functions : for screen handling
 - INT 21H functions: for displaying screen output accepting keyboard input.
- These functions request a particular action; you insert a function value in AH register to identify the type of service the interrupt is to perform.

INT 10H function

It is called video display control. It controls the screen format, color, text style, making windows, scrolling etc.

A typical video monitor has 25 rows (numbered 0 to 24) and 80 columns (numbered 0 to 79)

Screen Location	Decimal Format		Hex Format	
	Row	Column	Row	Column
Upper left corner	00	00	00H	00H
Upper right corner	00	79	00H	4FH
Center of screen	12	39/40	0CH	27H/28H
Lower left corner	24	00	18H	00H
Lower right corner	24	79	18H	4FH

Scanned with CamScanner

00H – set video mode

Load 00H in AH and required mode in AL

MOV AH, 00H ; set mode

MOV AL, 03H ; standard color text

INT 10H ; call interrupt service

01H- set cursor size

02H – Set cursor position

BH= page number, DH=row, DL=Column

Following example sets row 12, column 30 for page 0

```
MOV AH, 02H
MOV BH, 00H ; page no
MOV DH, 12 ; row 12
MOV DL, 30 ; column 30
INT 10H
```

#03H – return cursor status

04H- light pen function

05H- select active page

06H- scroll up screen

AL=no. of rows or lines to scroll (00 for full screen)

BH=attribute or pixel value

CX= starting row:column

DX=Ending Row : column

Scroll up by 1 row

```
MOV AX, 0601H ; request scroll up one line (text)
MOV BH, 61H ; brown background, blue foreground
MOV CX, 0C19H ; from row 12, column 25
MOV DX, 1236H; to row 18 , column 54
INT 10H
```

Clearing screen

```
MOV AX, 0600H ; request scroll up one line (text)
MOV BH, 61H ; brown background, blue foreground
MOV CX, 0000H ; from 00:00 through
MOV DX, 184F H ; to 24:79 (full screen)
INT 10H
```

07H-Scroll down screen

08H (Read character and Attribute at cursor)

09H -display character and attribute at cursor

0AH-display character at cursor

#0BH- Set color palette

#0CH- write pixel Dot

#0DH- Read pixel dot

#0EH- Display in teletype mode

#0F H- Get current video mode

INT 21function

01H (Keyboard input with echo)

MOV AH, 01H ; Request keyboard input

INT 21H

Character read is returned in AL in ASCII value

02H (Display single character)

- Load in DL the character that is to be display at current cursor position and request 21H.
- The tab, carriage return and line feed characters act normally and the operation automatically advances the cursor

MOV AH, 02H ; request display character

MOV DL, CHAR ; ch-aracter to display (use ASCII value or 'character')

INT 21H

#03H and 04H (Auxiliary input/output)

05H (Printer service)

06H (Direct keyboard and display)

07H(Direct keyboard input without echo)

- It works like 01H , except that the entered character does not echo on the screen and operation does not respond to a Ctrl+Break request. It could be used to key in a password that is to be invisible.

#08H(keyboard input without echo) : Same as function 01H but not echoed

09H(string display)

- It requires definition of a display string in data area , immediately followed by a dollar sign (\$) or 24H)
- \$ represents end of data

Write display string in data segment

NAMEE DB 'my name is ?' , '\$' ; or you can **NAME DB 'my name is ?\$'**

.....

Write following in code segment

MOV AH,09H

LEA DX,NAMEE ; load address of string in DX

INT 21H

0AH (Buffered keyboard input)

Data Segment

MAXLEN DB 255

ACTLEN DB ?

INPUT DB 255 DUP ('\$')

Code Segment

MOV AH, 0AH

LEA DX, MAXLEN

INT21H

Format of string stored in memory from keyboard

MAXLEN	ACTLEN	INPUT(0)	INPUT(1)	INPUT(N)
--------	--------	----------	----------	-----	------	----------

MAXLEN :

- defines maximum number of keyboard character
- MAXLEN DB 255 : you can enter 254 character with enter at the end of entry

ACTLEN: actual number of input character

***Note:** Initialize DX with effective address of INPUT or MAXLEN+2 for display using INT 21H by loading AH=09H

#0BH (check keyboard status)

#0CH(clear keyboard buffer and invoke functions)

WAP to read a character from keyboard and display it in next line

PAGE 60,132

TITLE SUM program to add two number

.....

.MODEL SMALL

.STACK 64 ; define stack

.DATA ; define data

.....

.CODE ; define code segment

MAIN PROC FAR

MOV AX,@data ; set address of data segment

MOV DS,AX

.....

; entering character from keyboard

MOV AH, 01H ; Request keyboard input

INT 21H

MOV CL, AL ; storing ASCII value of entered character in CL

.....

; moving to next line

MOV AH, 02H ; request display character

MOV DL, 0AH ; 0AH is ASCII of newline

INT 21H

.....

;cret

MOV AH, 02H ; request display character

MOV DL, 0DH ; 0DH is ASCII of cret

INT 21H

.....

; displaying entered character

MOV AH, 02H ; request display character

MOV DL,CL

INT 21H

MOV AX,4C00H ; end processing

INT 21H

MAIN ENDP ; end of procedure

END MAIN ; end of program

WAP to read string from keyboard and displaying it in next line

```
PAGE 60,132
TITLE  SUM program to add two number
;.....
.MODEL SMALL
.STACK 64                ; define stack
.DATA                    ; define data
MAXLEN DB 255
ACTLEN DB ?
INPUT DB 255 DUP ('$')
;.....
.CODE                    ; define code segment
MAIN PROC FAR
MOV AX,@data              ; set address of  data segment
MOV DS,AX
;.....
; entering string from keyboard
MOV AH, 0AH
LEA DX, MAXLEN
INT 21H
;.....
; moving to next line for display
MOV AH, 02H              ; request display character
MOV DL, 0AH              ; 0AH is ASCII of newline
INT 21H
;.....
;cret
MOV AH, 02H              ; request display character
MOV DL, 0DH              ; 0DH is ASCII of cret
INT 21H
;.....
; displaying entered string
MOV AH,09H
LEA DX,INPUT              ; load address of string in DX
INT 21H
MOV AX,4C00H              ; end processing
INT 21H
MAIN ENDP                ; end of procedure
END MAIN                  ; end of program
```

Program to display a character at the center of the screen

```
.model small
.stack 64
.data
;.....
.code
main proc far
mov ax, @data
mov ds, ax
; setting cursor position
MOV AH, 02H
MOV BH, 00H      ; page no
MOV DH, 12       ; row 12
MOV DL, 39       ; column 39
INT 10H
; displaying character on cursor position
MOV AH, 02H; request display character
MOV DL,'A'; character to display
INT 21H
;.....
mov ax, 4c00h
int 21h
main endp
end main
```


MACRO

- A macro is an instruction sequence that appears repeatedly in a program assigned with a specific name.
- A unique name is assigned to macro along with the set of instructions that the macro is to generate. Then, whenever you need to code the set of instructions, simply code the name of the macro.
- Macros are useful for following purposes
 - To simplify and reduce the amount of repetitive coding
 - To reduce errors caused by repetitive coding
 - To make an assembly language program more readable.

- **Format of macro**

Macroname	MACRO [parameter list]	; definition of macro
	[instructions]	; body of macro
	ENDM	; End of macro

- MACRO directive tells the assembler that the instructions that follows up to the ENDM are part of the macro definition.
- The macro definition appears before the coding of any segment.

WAP Program to add two number using macro

; Defining Macro for end processing

FINISH MACRO

MOV AX,4C00H

INT 21H

ENDM

; Defining macro for addition

ADDITION MACRO

MOV CH,DATA1

MOV CL,DATA2

ADD CH,CL

MOV DATA3,CH

ENDM

;.....

PAGE 60,132

TITLE SUM program to add two number

;.....

.MODEL SMALL

.STACK 64 ; define stack

.DATA

DATA1 DB 5

DATA2 DB 10

DATA3 DB ? ; define data

;.....

.CODE ; define code segment

MAIN PROC FAR

MOV AX,@DATA ; set address of

MOV DS,AX ;data segment

ADDITION ; Macro to add DATA1 and DATA 2

FINISH ; Macro to end processing

MAIN ENDP ; end of procedure

END MAIN ; end of program

WAP to add and subtract two number using macro

; Defining Macro for subtraction

SUBTRACTION MACRO VA,VB

MOV DH,VA

SUB DH,VB

MOV DATA4,DH

ENDM

; Defining Macro for addition

ADDITION MACRO VAL1,VAL2

MOV CH,VAL1

MOV CL,VAL2

ADD CH,CL

MOV DATA3,CH

ENDM

;.....

PAGE 60,132

TITLE SUM program to add two number

;.....

.MODEL SMALL

.STACK 64 ;define stack

.DATA

DATA1 DB 15H

DATA2 DB 10H

DATA3 DB ? ; To store result of addition

DATA4 DB ? ; To store result of subtraction

;.....

.CODE ; define code segment

MAIN PROC FAR

MOV AX,@DATA ; set address of

MOV DS,AX ; data segment

ADDITION DATA1,DATA2 ; Macro to add DATA1 and DATA 2

SUBTRACTION DATA1,DATA2 ; Macro to subtract DATA1 and DATA2

MOV AX,4C00H

INT 21H

MAIN ENDP ; end of procedure

END MAIN ; end of program

Calling Procedure (Subroutine)

- A code segment may contain any number of procedures, each distinguished by its own PROC and ENDP directives.
- A called procedure (subroutine) is a section of code that performs a clearly defined task.
- We use CALL and RET instruction for subroutine.
 - CALL procedure-name
 - RET

WAP to add and subtract two number using subroutine

PAGE 60,132

TITLE program to add and subtract two number

```
;.....  
.MODEL SMALL  
.STACK 64                ; define stack  
.DATA  
DATA1 DB 15H  
DATA2 DB 10H  
DATA3 DB ?                ; define data  
DATA4 DB ?  
;.....  
.CODE                    ; define code segment  
MAIN PROC FAR  
  
MOV AX,@DATA              ; set address of  
MOV DS,AX                 ; data segment  
  
CALL ADDITION              ; call subroutine  
CALL SUBTRACTION          ; call subroutine  
  
MOV AX,4C00H  
INT 21H  
MAIN ENDP                ; end of procedure  
;.....  
; Subroutine for addition  
ADDITION PROC NEAR  
MOV AH,DATA1  
ADD AH,DATA2  
MOV DATA3,AH  
RET  
ADDITION ENDP  
;.....  
; Subroutine for subtraction  
SUBTRACTION PROC NEAR  
MOV AH,DATA1  
SUB AH,DATA2  
MOV DATA4,AH  
RET  
SUBTRACTION ENDP  
;.....  
END MAIN                ;end of program
```

WAP to read a string from the user, convert it to uppercase, count the number of words and display each word in each line.

```
newline macro
```

```
mov ah, 02h
```

```
mov dl, 0ah
```

```
int 21h
```

```
mov dl, 0dh
```

```
int 21h
```

```
endm
```

```
;.....
```

```
.model small
```

```
.stack 64
```

```
;.....
```

```
.data db
```

```
maxlen db 255
```

```
actlen db ?
```

```
input db 255 dup('$')
```

```
msg db 'Total number of word is:','$'
```

```
;.....
```

```
.code
```

```
main proc far
```

```
mov ax, @data
```

```
mov ds, ax
```

; Reading input string from key board

```
MOV AH, 0AH
```

```
LEA DX, MAXLEN
```

```
INT 21H
```

```
;.....
```

```
newline
```

```

mov cl,actlen           ;cl is used as counter
mov ch,00h
mov bl,01h              ;bl is used to store word count
lea si,input
mainloop:cmp [si],32     ;compare if it is sapce
je label1               ;if space go to lebel 1,increase bi then si and loop
cmp [si],91             ;compare if it is upper case
jb display              ;if yes display it otherwise convert it to uppercase
sub [si],32              ;converting lower case to uppercase
display: mov ah,02h      ;displaying the uppercase letter
        mov dl,[si]
        int 21h
        jmp label2
label1: newline
        inc bl
label2: inc si
loop mainloop
        newline
        newline
;displaying the message total no. of word is
        mov ah,09h
        lea dx,msg
        int 21h
;displaying total count

mov dl, bl
or dl, 30h              ; converting hex to ascii
mov ah,02h
int 21h

```

```
mov ax, 4c00h
```

```
int 21h
```

```
main endp
```

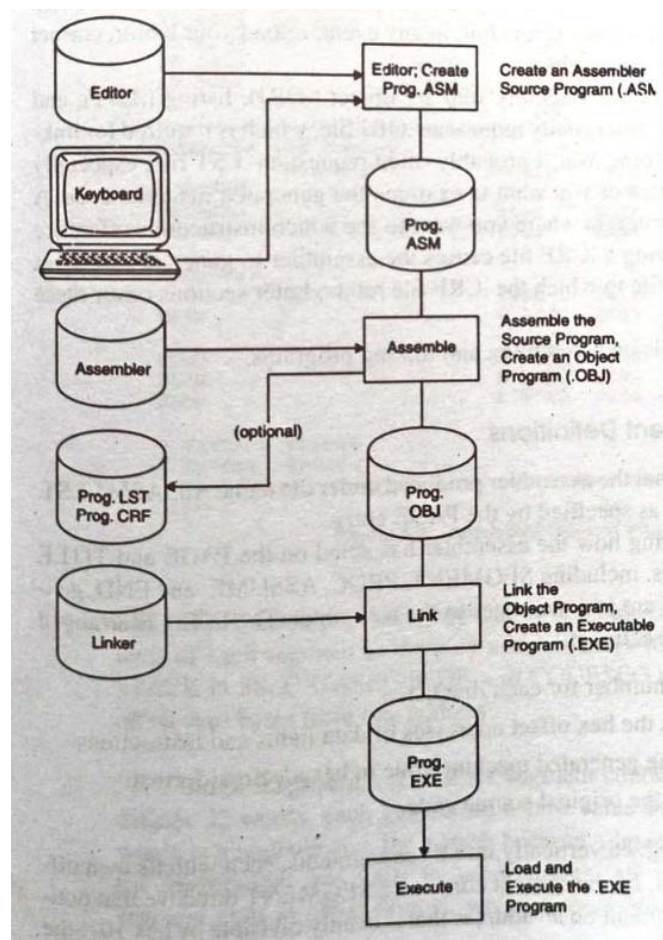
```
end main
```


Assembling, Linking and Executing

- **Source program:**

- The symbolic instructions that we code in assembly language are known as **source program**.
- We can use any editor or word processing program to key in the statements for the source program.
- This editor program is then converted to .ASM.
- This .ASM file is just a text file that cannot execute so it must be assembled and linked.

Steps required to assemble, link and execute a program



Assembling a source program:

- Assembler is used to translate the assembly language instructions to the corresponding binary codes.
- The assembler generates two files.
 - Object file (.OBJ)
 - It contains the binary codes for the instructions and information about the offset address of the instructions.
 - After further processing the, the contents of the Object file will be loaded into memory and run.
 - Assembler list file (.LST) :
 - It contains
 - Assembly language statements
 - Binary code for each instruction
 - Offset of each instruction and data
 - This .LST file is usually sent to a printer so that you will have a printout of the entire program to work with when you are testing and troubleshooting the program.
 - The assembler listing will also indicate any typing or syntax errors made in the source program.
- Assembler calculates the offset address for every data item in data segment and every instruction in code segment.
- The assembler also displays error message and it also attempts to correct some errors.
- Assembler also creates a **header** immediately in front of the generated .OBJ module; part of the header contains information about incomplete address.

Linking an object program

- When the program is free of error messages, the next step is to link the .OBJ module produced by the assembler and that contains machine code.
- The Link step involves converting the **.OBJ** module into **.EXE** (executable) machine code module.
- The linker performs following functions
 - Completes the address left open by the assembler

- Combines, if requested, more than one separately assembled module into one executable program.
 - Generates an .EXE module and initializes it with special instructions to facilitate its subsequent loading for execution
- Once you have linked one or more .OBJ modules into an .EXE module, you may execute the .EXE module any number of times.
- The output files from link step are executable (.EXE), map (.MAP) and library (.LIB).

Loading and Executing

- Having assembled and linked a program, you can now execute it.
- The last step is to load the program for execution.
- Because the loader knows where the program is going to load in memory, it resolves remaining addresses still left incomplete in the header.
- The loader drops the header and creates a **program segment prefix (PSP)** immediately before the program loaded in memory.

Assembler Types

One pass Assembler

- It is possible to implement an assembler that makes only a single pass through the source code.
- **Forward Reference:** The main difficulty in trying to assemble a program in one pass involves forward references to labels. Instruction operands may be symbols that have not yet been defined in the source program. Therefore, the assembler does not know what relative address to insert in the translated instruction.

JMP LATER

...

...

LATER:

Two pass Assembler

- This type of assembler scans the source program twice.
- First Pass:
 - the assembler reads the entire source program
 - Constructs a **symbol table** of names and labels used in program, that is, names of data field and program labels and their relative locations (offset) within the segment.
 - It examines each instruction sufficiently to determine the length of the corresponding machine instruction
- Second pass :
 - The second pass reads the program again from the beginning.
 - Each instruction is translated into the appropriate binary machine code.
 - The translation to binary code includes:
 - Translate the mnemonic into a binary opcode
 - Translate each operand name into the appropriate register or memory code.
 - Translate each immediate value into a binary string.
 - Translate any references to labels into the appropriate value using the symbol table.

Operators in 8086

An operator provides a facility for changing and analyzing operands during assembly.

Operators are active during assembling but no machine language code will be generated.

Types of operators

a. Calculation operator

- **Arithmetic operator:** These operators include the familiar arithmetic signs and performs arithmetic during assembly.

+: addition	Array+25	adds 25 to address of array
+: positive	+Arr	Treats Arr as positive
-: subtraction	arr-brr	calculates difference between arr-brr
*: multiplication	value*3	multiply value by 3

- **Index operator :**

- For indirect addressing of memory, an operand references a base or index register, constants, offset variables and variables.
- Index operator uses square brackets

MOV AX,[SI]

MOV CL,TABLE[4]

MOV AL,[5000H]

- **Logical operator :** it processes the bits in an expression

Operator	Used as
AND	Expression1 AND expression 2
OR	Expression1 OR expression 2
XOR	Expression1 XOR expression 2
NOT	NOT expression

MOV CL,10101011B AND 11111110B

- **Shift operator :**

- The operator SHL and SHR shift an expression during an assembly.
- Expression **SHL/SHR count**

MOV BL, 10101010B SHR 3 ; SHR operator shifts the bit constant three bit to right

b. Macro operators

- A macro is an instruction sequence that appears repeatedly in program assigned with a specific name.

c. Relational operator

- EQ,GE,GT,LE,LT and NE

d. Segment operator

- **OFFSET operator:** it returns the offset address of a variable or label.

OFFSET VARIABLE/LABEL

TABLE DB 11,22,33

MOV DX, OFFSET TABLE; moves offset address of table

- **SEG operator:** it returns address of the segment in which a specified variable or label is placed.

SEG VARIABLE/LABEL

MOV AX,SEG WORDA ; get address of data segment

MOV AX, SEG A10BEGIN ; get address of code segment

e. Type or attribute operator

- **HIGH and HIGHWORD operator:**

- HIGH operator returns the high(leftmost) byte of an expression
- HIGHWORD returns high word of an expression

NAME EQU 1234H

MOV CL, HIGH NAME ; load 12H in CL

- **LOW and LOWWORD operator:**

- LOW operator returns the low(rightmost) byte of an expression
- LOWWORD operator returns low word of an expression

- **LENGTH operator:** it returns the number of entries defined by a DUP operator.

NAME DW 10 DUP(?)

MOV DX, LENGTH NAME ; returns length 10 to DX

- **TYPE operator :** it returns the no. of bytes , according to the definition of referenced variable

BYTEA DB ?

PART DW 10 DUP(?)

MOV AX, TYPE BYTEA ; AX=0001H

MOV CX, TYPE PART ; CX=0002H

f. Record operator

- **MASK and WIDTH** are record operator

.COM and .EXE program

Segments

- **.COM** program combines PSP , stack , data segment and code segment into one code segment
- In **.COM** program data is defined within the code segment
- **.EXE** program consists of separate code, data and stack segment. The maximum size of each segment is 64 KB.
- **Stack segment**
 - In **.EXE** program you have to define stack segment
 - In **.Com** Program assembler automatically generates a stack.

Program size

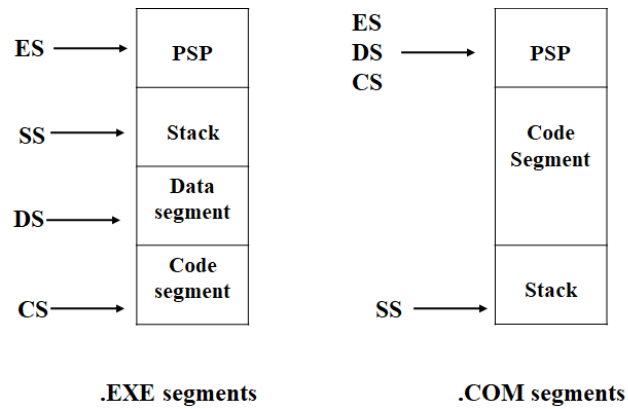
- Maximum size of **.COM** program is 64K.
- **.EXE** program is larger in size.

Initialization in .EXE

- When **.EXE** program is loaded the address of PSP is loaded in DS and ES register so we have to initialize DS with the address of Data segment.
- Loader loads the address of Code segment is loaded in CS register
- Loader loads the address of Stack segment is loaded in SS register.

Initialization in .COM

- When loader loads **.COM** program, it automatically initializes CS, DS, SS and ES with the address of PSP.
- Since DS and CS now contain the correct initial segment address at execution time, a **.COM** program does not have to initialize them.
- A **.COM** program uses TINY model.
- **PSP (Program segment Prefix)** is a 256 byte (100H) block that the program loader inserts immediately preceding **.COM** and **.EXE** program when it loads them from disk to memory.
- A program written as **.COM** requires ORG 100H immediately following the code segment's SEGMENT statement or .code statement.
- The ORG directive causes assembler to begin generating the object code at an offset of 100H bytes past the start of the PSP, where your coding for **.COM** program begins. The program loader stores the 100H in the IP register when it loads **.COM** program.



WAP to add two number using .COM program

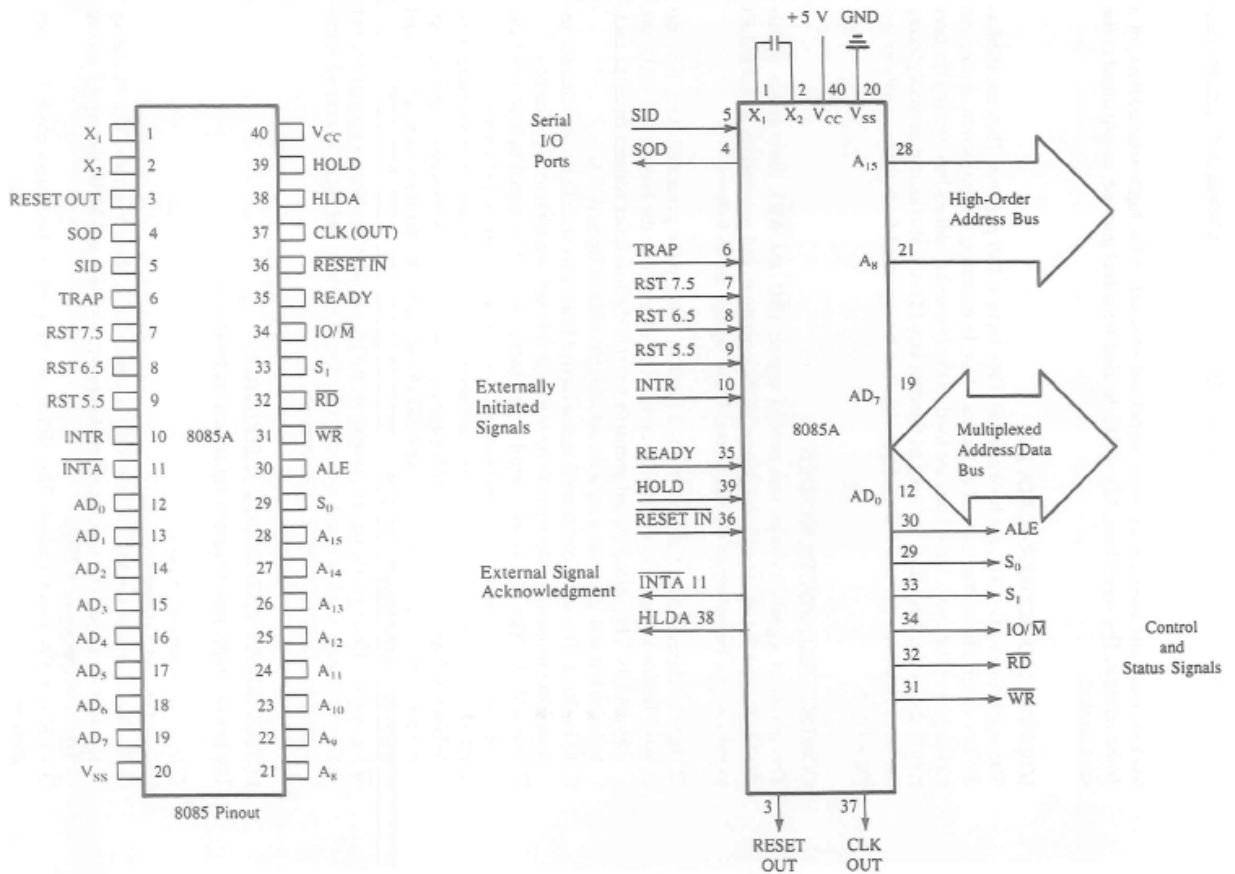
```

.MODEL TINY
.CODE
ORG 100H           ; START AT THE END OF PHP
BEGIN: JMP MAIN    ; JUMP PAST DATA
;.....
DATA1 DB 10
DATA2 DB 5
DATA3 DB ?
;.....
MAIN PROC NEAR
MOV AH,DATA1
ADD AH,DATA2
MOV DATA3,AH
MOV AX,4C00H
INT 21H
MAIN ENDP
END BEGIN

```


Pin Configuration of 8085 microprocessor

The **8085A** (commonly known as **8085**) is a 8-bit general purpose microprocessor capable of **addressing 64K of memory**. The device has **40 pins**, require a **+5V** single power supply and can operate with a **3-MHZ**, single phase clock.

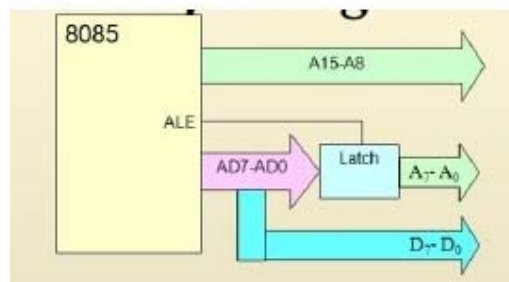


Address bus:

- 16 signal lines that are used as the address bus.
- These lines are split into two segments
 - $A_{15}-A_8$: Unidirectional and called **high order address bus**
 - AD_7-AD_0 - Bidirectional Multiplexed **Low order Address bus /8-bit Data Bus**

Data Bus:

- The signal lines AD_7-AD_0 are bidirectional, they serve a dual purpose.
- They are used as the low order address bus as well as data bus (this is known as multiplexing)



Control and Status signal:

- This group of signals includes
 - 2 control signals : \overline{RD} and \overline{WR}
 - 3 status signals : IO/\overline{M} , S_1 and S_0 to identify the nature of the operation
 - 1 special signals ALE to indicate the beginning of the operation

ALE (address Latch enable):

- This is +ve going pulse generated every time 8085 begins an operation.
- When ALE is high bits AD_7-AD_0 are address bits
- This signal is used primarily to latch the low-order address from the multiplexed bus and generate a separate set of eight address lines A_7-A_0 .

\overline{RD} (Read):

- Active low read control signal
- It indicates that the selected I/O or memory device is to be read and data are available on the data bus

\overline{WR} (Write):

- Active low write control signal
- It indicates that the data on the data bus are to be written into the selected memory or I/O.

IO/\overline{M} :

- Used to differentiate between I/O and memory operation
- When IO/\overline{M} = high : it indicates an I/O operation
- When IO/\overline{M} = low : it indicates memory operation

This signal is combined with \overline{WR} and \overline{RD} to generate I/O and memory control signal

S₁ and S₀ :

- These status signals can identify various operations

Machine Cycle	Status			Control Signals
	IO/\overline{M}	S ₁	S ₀	
Opcode Fetch	0	1	1	$\overline{RD} = 0$
Memory Read	0	1	0	$\overline{RD} = 0$
Memory Write	0	0	1	$\overline{WR} = 0$
I/O Read	1	1	0	$\overline{RD} = 0$
I/O Write	1	0	1	$\overline{WR} = 0$
Interrupt Acknowledge	1	1	1	$\overline{INTA} = 0$
Halt	Z	0	0	$\overline{RD}, \overline{WR} = Z$ and $\overline{INTA} = 1$
Hold	Z	X	X	
Reset	Z	X	X	

NOTE: Z = Tri-state (high impedance)
X = Unspecified

Power supply and clock frequency

- V_{CC}: +5V power supply
- V_{SS}: Ground reference
- X1 and X2:
 - A crystal is connected at these two pins for frequency.
 - The frequency is internally divided by two therefore to operate a system at 3 MHz, the crystal should have frequency of 6 MHz`
- CLK (OUT) –clock output: It can be used as the system clock for other devices.

Externally initiated signals

- INTR (input): interrupt request,-used as a general purpose interrupt.
- \overline{INTA} (Output): interrupt acknowledge -This is used to acknowledge an Interrupt.
- RST 7.5, 6.5, 5.5 (inputs): Restart interrupt-These are vectored interrupts that transfer the program control to specific memory locations. They have higher

priorities than INTR interrupt. Among these three, the priority order is 7.5, 6.5, and 5.5.

- **TRAP (input):** This is a non-maskable interrupt with highest priority
- **HOLD (input):** This is an **active high input** signal to indicate that a peripheral such as a DMA (Direct Memory Access) controller is requesting use of Address and data bus.
- **HLDA (output):** Hold Acknowledge-
 - This signal acknowledges the HOLD request
 - It is an **active high** output signal indicating that the Microprocessor is relinquishing the control of the buses.
- **READY (Input) :**
 - **This signal is used to synchronize slower peripherals with microprocessor.**
 - This signal is used to delay the microprocessor Read or Write cycles until a slow- responding peripheral is ready to send or accept data.
 - **When this signal goes low, the microprocessor waits for an integral number of clock cycles until it goes high.**
- **$\overline{RESET\ IN}$:** When the signal on this pin goes low
 - the program counter is set to zero
 - the buses are tri-stated
 - and MPU is reset
- ***RESET OUT*:** This signal indicates that the MPU is being reset. The signal can be used to reset other devices.

Serial I/O ports:

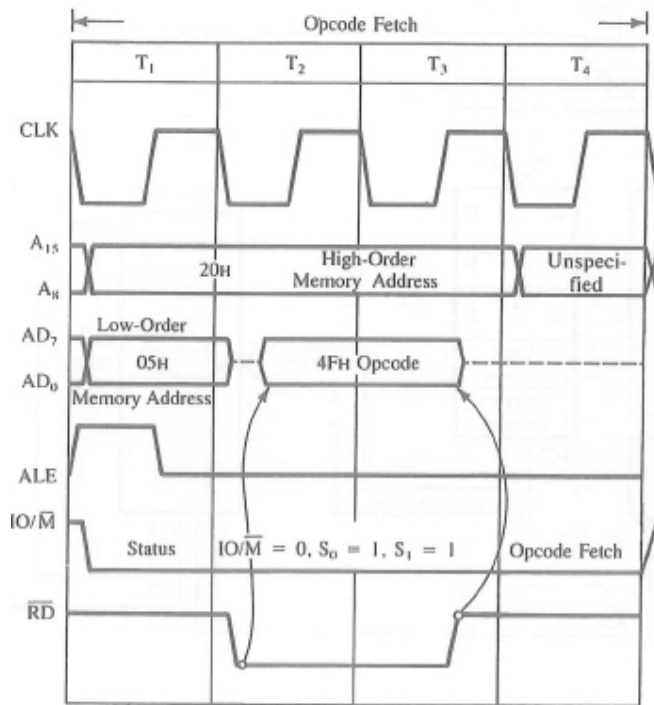
- SID and SOD are used for single bit serial in and serial out through RIM and SIM instructions.

Machine cycles and bus timing diagrams of 8085 microprocessor

- Microprocessor external communication functions can be divided into following category
 - Op- Code fetch
 - Memory Read /Write
 - I/O Read/ Write
 - Request acknowledgement
- **Machine cycle:**
 - It is defined as the time required to complete **one operation of accessing memory, I/O, or acknowledging an external request.**
 - This cycle may consist of three to six T-states.
- **Instruction Cycle:**
 - It is defined as the **time required to complete the execution of an instruction.**
 - The 8085 instruction cycle consists of one to six machine cycles or one to six operations.
- **T state:**
 - It is defined as one subdivision of the operation performed in one clock period.
 - Each T state is precisely equal to one clock period.
 - The term T state and clock period are used synonymously.

Op code fetch Machine cycle

Address	Instruction
2005H	MOV C,A (4FH)



-The dashed straight line indicates high impedance state.

-After T3:

- The contents of bus A₁₅-A₈ are unknown

Step 1: (Microprocessor places the 16 bit memory address from Program Counter on the address bus.)

- PC → Address Bus
- At T₁
 - Higher order memory address 20H → A₁₅-A₈
 - Lower order memory address 05H → AD₇-AD₀ and ALE goes high
 - For opcode fetch : IO/ \overline{M} = 0 , S₀ = 1, S₁ = 1

Step 2 : (The control unit sends the control signal \overline{RD} to enable the memory chip)

- At T₂: Control unit sends \overline{RD} to enable the memory chip.
- \overline{RD} signal is active during two clock pulses

Step 3: (The byte from memory location is placed on the data bus)

- The byte 4FH from the memory location is placed on the data bus and transferred to microprocessor

Instruction byte 4FH → AD₇-AD₀

- The \overline{RD} signal causes 4FH to be placed on bus AD₇-AD₀

- When \overline{RD} goes high, it causes the bus AD₇-AD₀ to go into high impedance.

Step 4 : (The byte is placed in instruction decoder and the task is carried out according to instruction)

- At T₄: The instruction byte 4FH is decoded by instruction decoder and the content of A are copied into register C.

NOTE:

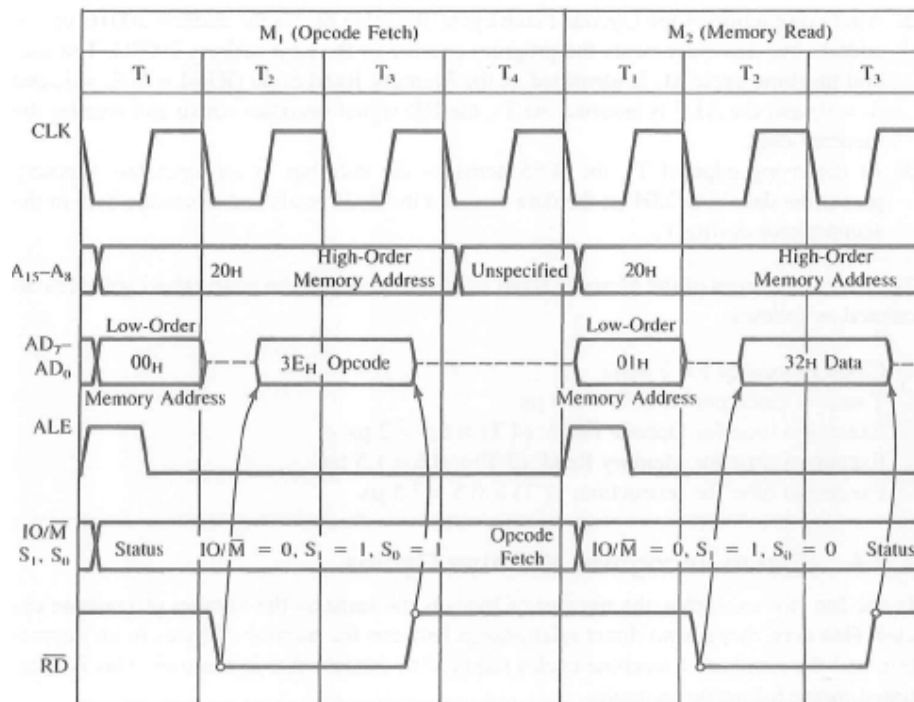
- Fetch cycle is completed in T₃
- During T₄, 8085 decodes the opcode
- After T₃, the contents of bus A₁₅-A₈ are unknown and data bus AD₇-AD₀ goes into high impedance state.
- Some instructions have opcodes with six T states.

Memory Read Machine cycle

Address	Instruction	HEX
2000H	MVI A,32H	3E
2001H		32

Two Machine Cycle are required:

- First machine cycle is Opcode Fetch (4T-State).
- Second machine cycle is Memory Read (3T-State).



Steps in Memory read machine cycle

Step1 (T₁ state):

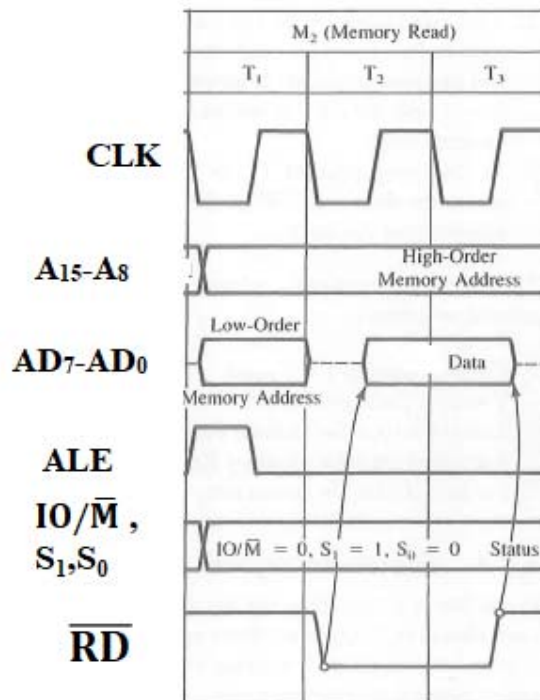
- places the address on the address lines
 - 20H → A₈-A₁₅
 - 01H → AD₀-AD₇
- Activates ALE in order to latch low-order of address.
- Sends the status signals with logical status $IO/\overline{M} = 0, S_1=1, S_0=0$ for memory read machine cycle.

Step2 (T₂ state):

- 8085 processor activates the \overline{RD} signals to enable the addressed memory location
- At the rising edge of T₂, 8085 activates the data bus as an input bus, memory places the data byte 32H on the data bus.

Step 3: (T₃ state)

- The processor loads the contents of data bus on specified register A and deactivates the \overline{RD} signal to disables the memory devices.



Memory read machine cycle

Example: Find the execution time of memory read machine cycle and instruction cycle of above timing diagram if the clock frequency is 2MHz.

Clock frequency $f = 2\text{MHz}$

T state = clock period $= 1/f = 0.5 \mu\text{s}$

Execution time of opcode fetch: $(4T) \times 0.5 \mu\text{s} = 2 \mu\text{s}$

Execution time of memory read: $(3T) \times 0.5 \mu\text{s} = 1.5 \mu\text{s}$

Execution time for instruction: $(7T) \times 0.5 \mu\text{s} = 3.5 \mu\text{s}$

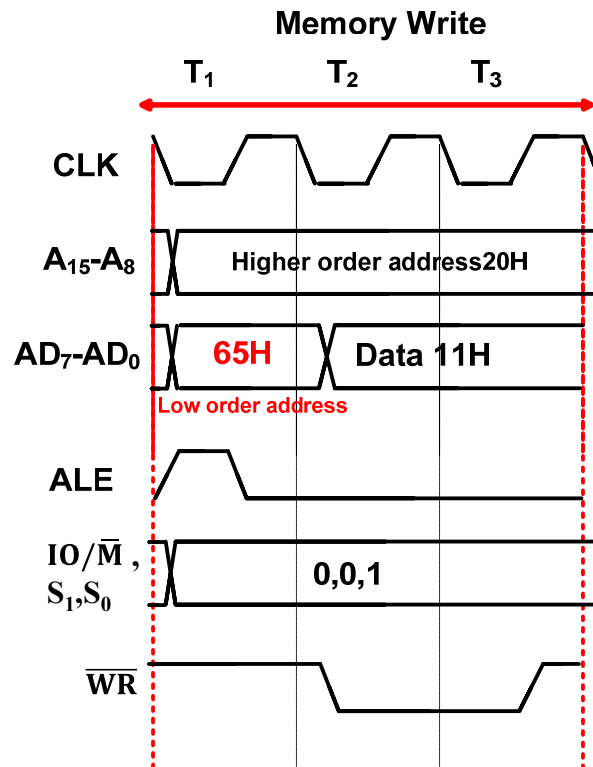
Memory write machine cycle

Address	Instruction	HEX
2010H	STA 2065H	32
2011H		65
2012H		20

Let A contains 11H

Four Machine cycles are required to execute this instruction:

- First Machine cycle is Opcode Fetch (4T-State).
 - Places address 2010H on address bus and fetches opcode 32H
- Second Machine cycle is Memory Read (3T-State).
 - Places address 2011H and get low order byte 65H
- Third Machine cycle is Memory Read (3T-State).
 - Places address 2012H and get high order byte 20H
- Fourth Machine cycle is Memory Write (3T-State)



Step1 (T1 state):

- Processor places the address on the address lines
 - 20H → A8-A15
 - 65H → AD0-AD7
- Activates ALE in order to latch low-order of address.
- Sends the status signals with logical status $IO/\overline{M} = 0, S_1=0, S_0=1$ for memory write machine cycle.

Step2 (T2 state):

- 8085 processor places the data (content of A) on data bus AD₇-AD₀ and then activates the \overline{WR} signal to writing data into addressed memory location.

Step 3: (T3 state)

- Contents of data bus are placed in memory location 2065H
- The processor deactivates the \overline{WR} signal which disables the memory device and terminates the write operation

STA 526AH

Address	Instruction	HEX
41FFH	STA 526AH	32
4200H		6A
4201H		52

A contains data C7H

It requires 4 machine cycle

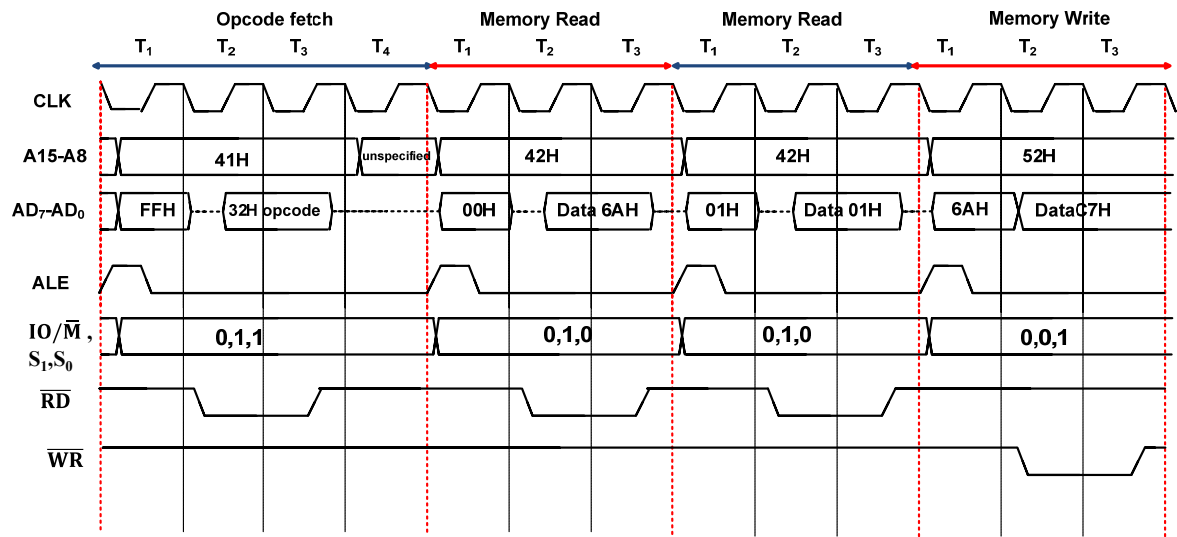
- First Machine cycle is Opcode Fetch (4T-State).
- Second Machine cycle is Memory Read (3T-State).
- Third Machine cycle is Memory Read (3T-State).
- Fourth Machine cycle is Memory Write (3T-State).

STEP1: Reads opcode (32H) by Opcode Fetch Machine Cycle.

STEP2: Reads lower byte (6AH) of address by memory read cycle.

STEP3: Reads Higher byte of address (52H) by memory read cycle.

STEP4: Stores the content of accumulator into the memory location by memory write cycle.

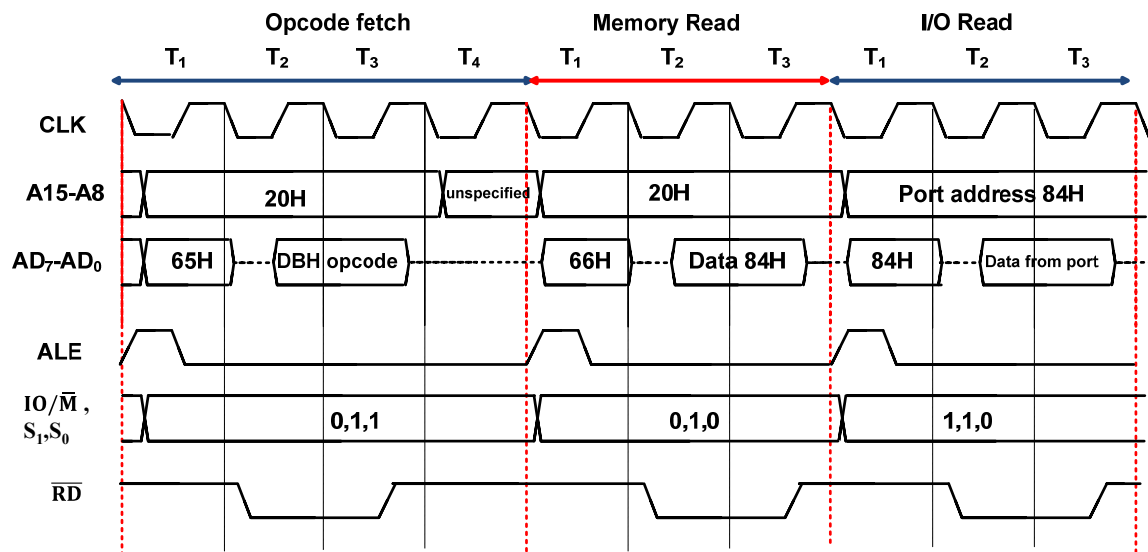


I/O Read Machine cycle

Address	Instruction	HEX
2065	IN 84H	DBH
2066		84H

This instruction requires three machine cycle

- First Machine cycle is Opcode Fetch (4T-State).
- Second Machine cycle is Memory Read (3T-State).
- Third Machine cycle is I/O Read (3T-State).



Steps in I/O Read cycle

Step1 (T1 state):

- Processor places the address of input port 84H on lower address bus as well as on higher address bus.
- Activates ALE in order to latch low-order of address.
- Sends the status signals with logical status (1 1 0) for IO read machine cycle.

Step2 (T2 state):

- Activates the \overline{RD} signals to enable the addressed input device which places its contents on the data bus (AD0-AD7).

Step 3: (T3 state)

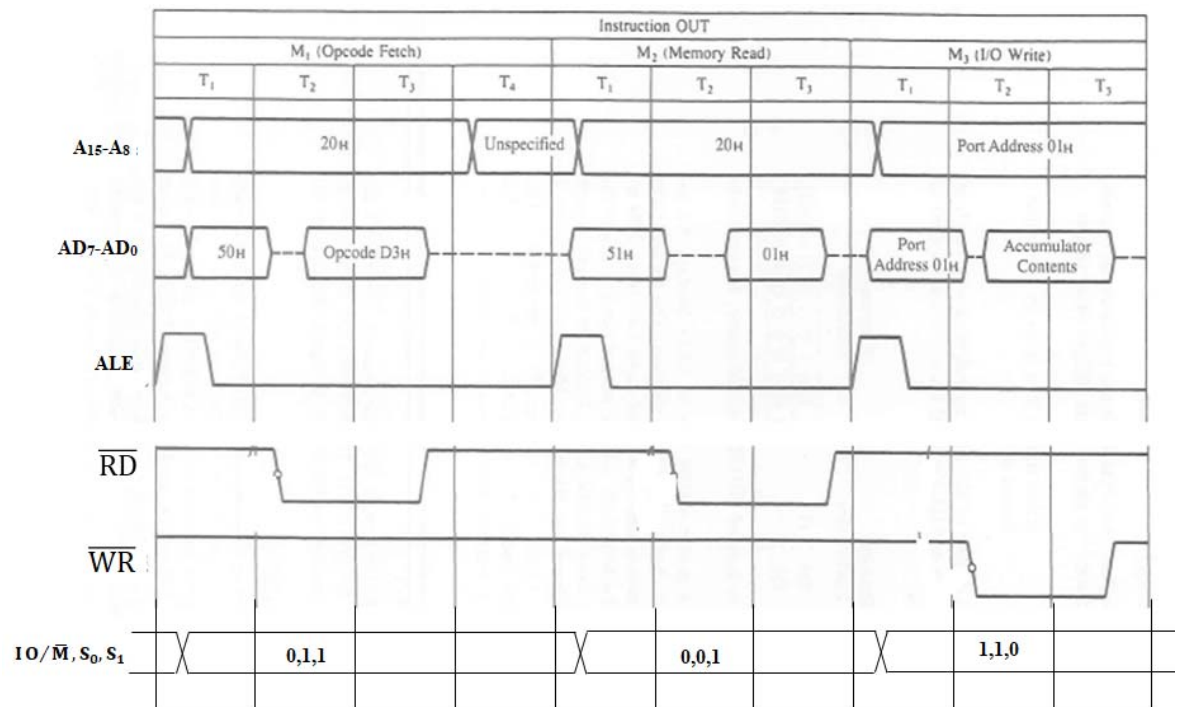
- The processor loads the contents of data bus on A and deactivates the RD signal to disable the input device.

I/O Write machine cycle

Address	Instruction	HEX
2050	OUT 01H	D3H
2051		01H

This instruction requires three machine cycle

- First Machine cycle is Opcode Fetch (4T-State).
- Second Machine cycle is Memory Read (3T-State).
- Third Machine cycle is I/O write (3T-State).



Steps in I/O Write machine cycle

Step1 (T1 state):

- Processor places the address of output port 01H on lower address bus as well as on higher address bus.
- Activates ALE in order to latch low-order of address.
- Sends the status signals $\text{IO}/\overline{\text{M}} = 1, S_1=0, S_0=1$ for I/O write machine cycle.

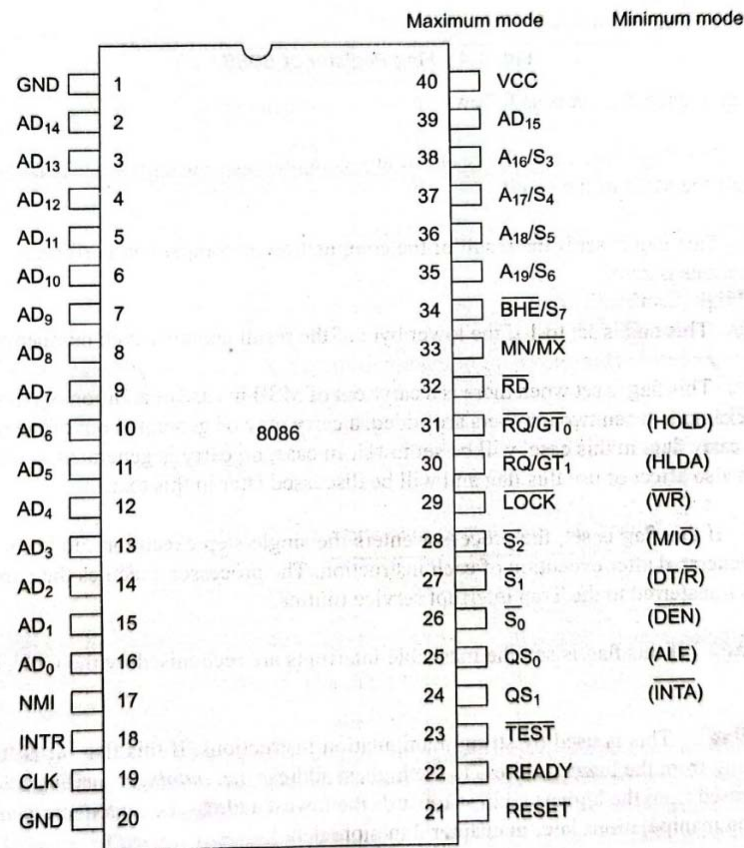
Step2 (T2 state):

- 8085 processor places the Accumulator content on data bus and activates the $\overline{\text{WR}}$ signal to writing data in to addressed output device.

Step 3: (T3 state):

- The processor deactivates the $\overline{\text{WR}}$ signal which disables the output device and terminates the writing operation

Pin Configuration of 8086



- The Microprocessor 8086 is a 16-bit CPU available in different clock rates and packaged in a 40 pin Cerdip or plastic package.
- The 8086 operates in
 - single processor mode (minimum mode)
 - multiprocessor mode (maximum mode)
- The pins serve a particular function in minimum mode (single processor mode) and other function in maximum mode configuration (multiprocessor mode).
- The 8086 signals can be categorized in three groups.
 - The first are the signal having common functions in minimum as well as maximum mode.
 - The second are the signals which have special functions for minimum mode
 - The third are the signals having special functions for maximum mode.

The following signal descriptions are common for both modes

- **AD₁₅-AD₀:**
 - These are the time multiplexed memory I/O address and data lines.
 - The address can be latched using ALE signal.
- **A₁₉/S₆, A₁₈/S₅, A₁₇/S₄, A₁₆/S₃ :**
 - These are the time multiplexed address and status lines.
 - The address bits are separated from the status bit using latches controlled by the ALE signal.
 - The status of interrupt enable flag is displayed on S₅.
 - S₄ and S₃ together indicate which segment register is presently being used for memory access.
 - S₆ shows the status of bus master. When S₆ is low it indicates that 8086 is bus master.

S3	S4	Operation
0	0	Extra Segment (ES)
0	1	Stack Segment (SS)
1	0	Code Segment (CS) or Idle
1	1	Data Segment (DS)

- **\overline{BHE}/S_7 (bus high enable/status):**
 - The bus high enable is used to indicate the transfer of data over the higher order (D₁₅-D₈) data bus.
 - It goes low for the data transfer over D₁₅-D₈ and is used to drive chip select of odd address memory bank or peripherals.
- **\overline{RD} (Read):**
 - This signal when low, indicates the peripheral that the processor is performing memory or I/O read operation.
- **READY :**
 - This is active high acknowledgement from the slow device or memory that they have completed the data transfer and ready for next data transfer.

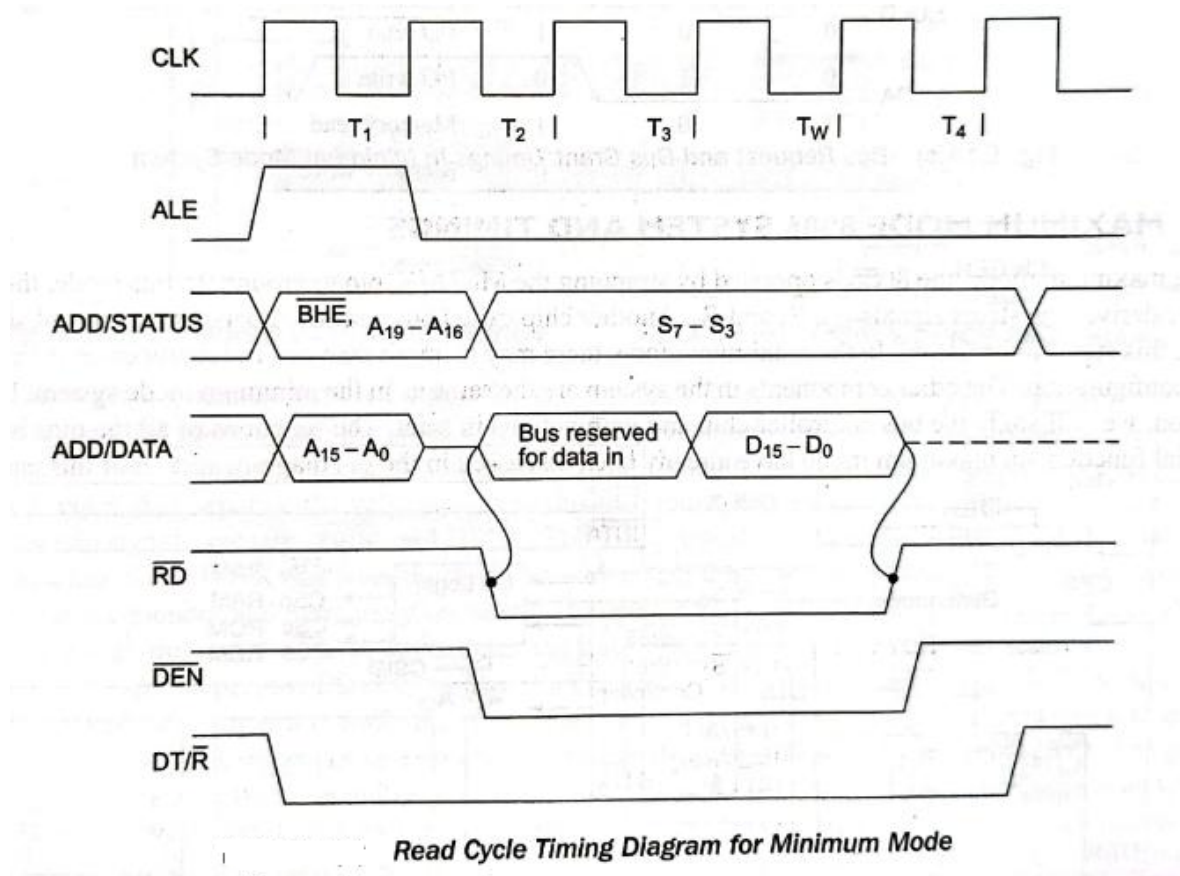
- **INTR-Interrupt Request :**
 - This is level triggered input
 - This is sampled during last clock cycle of each instruction to determine the availability of the request
 - If any interrupt request is pending, the processor enters the interrupt acknowledge cycle.
- **NMI (NON maskable interrupt) :** This is an edge triggered input which causes a Type 2 interrupt.
- **RESET :**
 - This input causes the processor to terminate the current activity and start execution from FFFF0 H
- **\overline{TEST} :**
 - When the 'WAIT' instruction is executed, 8086 enters into an idle condition in which it is doing no processing.
 - This input is examined by a 'WAIT' instruction.
 - If the TEST pin goes low, execution will continue, else the processor remains in an idle state.
- **CLK- Clock Input :**
 - The clock input provides the basic timing for processor operation and bus control activity
- **MN/\overline{MX} :**
 - The logic level at this pin decides whether the processor is to operate in either minimum or maximum mode.
 - $MN/\overline{MX}=1$ refers to minimum mode
 - MN/\overline{MX} is connected to ground for maximum mode

The following pin functions are for the minimum mode operation of 8086.

- **$\overline{M/I/O}$ (Memory/IO) :**
 - When it is low, it indicates the CPU is having an I/O operation
 - when it is high, it indicates that the CPU is having a memory operation.
- **\overline{INTA} (Interrupt Acknowledge):**
 - This signal is used for interrupt acknowledge
 - When it goes low; the processor has accepted the interrupt.
- **ALE – Address Latch Enable:**
 - This output signal indicates the availability of the valid address on the address/data lines.
 - It is connected to the latch enable input of latches.
 - It is active high signal.
- **DT/\overline{R} – Data Transmit/Receive:**
 - This output is used to decide the direction of data flow through the trans-receivers (bidirectional buffers).
 - When the processor sends out data, this signal is high and when the processor is receiving data, this signal is low.
- **\overline{DEN} – Data Enable:**
 - This signal indicates the availability of valid data over the address/data lines.
 - It is used to enable the trans-receivers (bidirectional buffers) to separate the data from the multiplexed address/data signal.
- **HOLD, HLDA- Hold/Hold Acknowledge:**
 - When the HOLD line goes high, it indicates to the processor that another master is requesting the bus access.
 - The processor, after receiving the HOLD request, issues the hold acknowledge signal on HLDA pin and processor floats the bus lines.
 - When processor detects the HOLD line low, it lowers HLDA signal.

Read cycle timing diagram for minimum mode

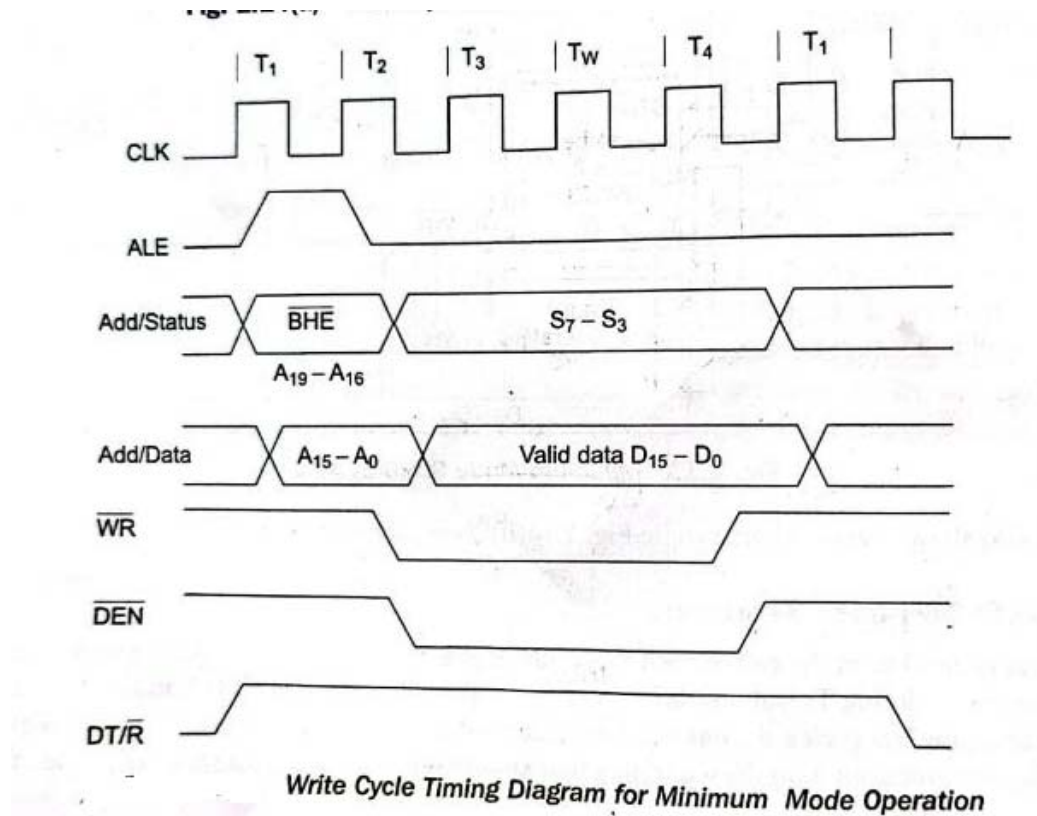
The working of 8086 in minimum mode can be better described by timing diagram. The opcode fetch and read cycle are similar. Hence the timing diagram can be categorized in two part : Timing diagram for read cycle and timing diagram for write cycle.



- The read cycle begins in T₁ with the assertion of
 - address latch enable (ALE) signal : to latch address
 - **M / \overline{IO} signal : to indicate memory operation (it is asserted from T₁ to T₄)**
- During the negative going edge of ALE signal, the valid address is latched on the local bus.
- The \overline{BHE} and A₀ signals address low, high or both bytes.
- At T₂
 - the address is removed from the local bus and is sent to the output. The bus is then tri-stated.

- The read (\overline{RD}) control signal is activated : The read (\overline{RD}) signal causes the address device to enable its data bus drivers
- After \overline{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

Write cycle timing diagram for minimum mode



- The write cycle begins in T1 with the assertion of
 - address latch enable (ALE) signal : to latch address
 - M/\overline{IO} signal : to indicate memory operation (it is asserted from T1 to T4)
- In T2 :
 - the processor sends the data to be written to the addressed location.
 - \overline{WR} becomes active
- The BHE and A0 signals are used to select the proper memory location or I/O.

The following pin functions are applicable for maximum mode operation of 8086.

- $\overline{S_2}, \overline{S_1}, \overline{S_0}$ – Status Lines:

- These are the status lines which reflect the type of operation, being carried out by the processor.

$\overline{S_2}$	$\overline{S_1}$	$\overline{S_0}$	indication
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

- \overline{LOCK} :

- This output pin indicates that other system bus master will be prevented from gaining the system bus, while the \overline{LOCK} signal is low.

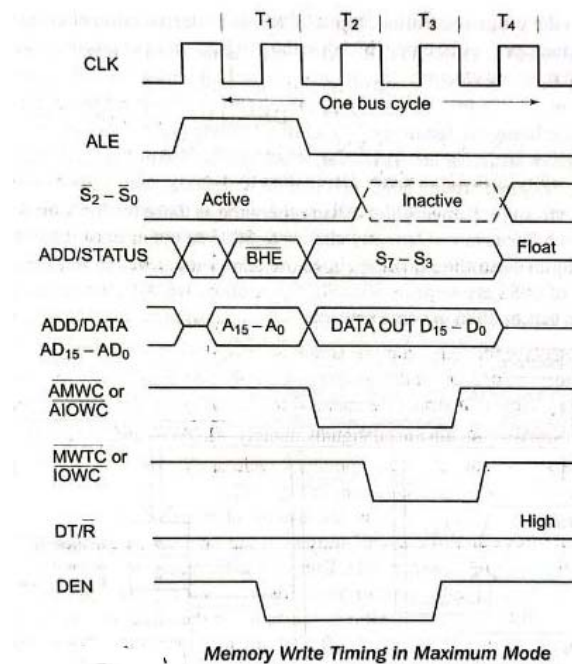
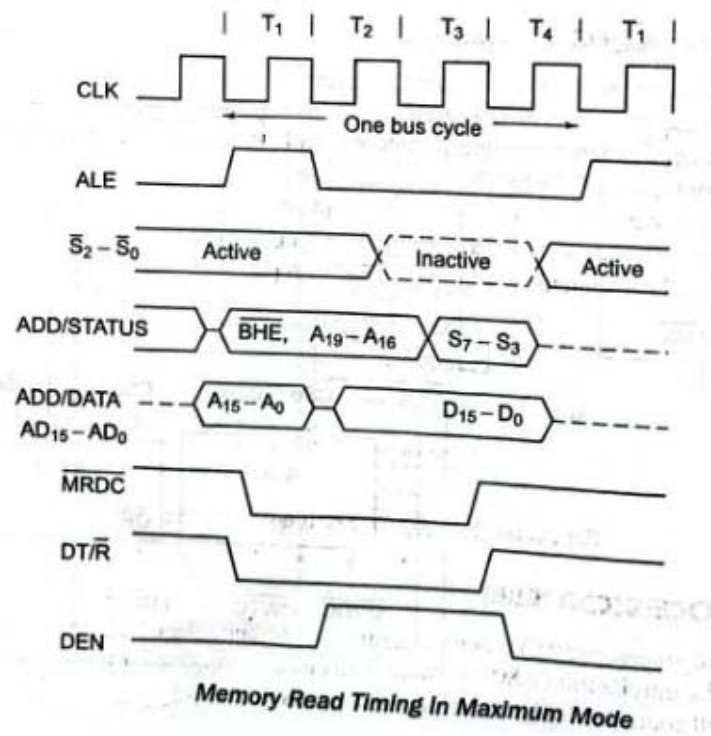
- $\overline{RQ}/\overline{GT_0}, \overline{RQ}/\overline{GT_1}$ – Request/Grant :

- These pins are used by the other local bus master in maximum mode, to force the processor to release the local bus at the end of the processor current bus cycle.
- $\overline{RQ}/\overline{GT_0}$ have high priority than $\overline{RQ}/\overline{GT_1}$

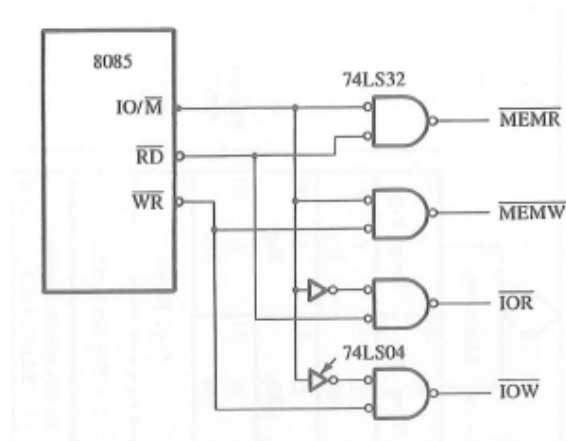
- QS_1, QS_0 (Queue status): These lines give information about the status of the code perfected queue.

QS_1	QS_0	Indication
0	0	No operation
0	1	First byte of opcode from queue
1	0	Empty queue
1	1	Subsequent byte from queue

Read and write cycle timing diagram for maximum mode



Generating Control signals



Address Decoding

- **Address decoding** refers to the way a computer system decodes the addresses on the address bus to select memory locations in one or more memory or peripheral devices.
- Microprocessor is connected with memory and I/O devices via common address and data bus.
- Only one device can send data at a time and other devices can only receive that data. If more than one device sends data at the same time, the data gets garbled. In order to avoid this situation, ensuring that the proper device gets addressed at proper time, the technique called address decoding is used.
- In address decoding method, all devices like memory blocks, I/O units etc. are assigned with a specific address. The address of the device is determined from the way in which the address lines are used to derive a special device selection signal known as chip select (\overline{CS}). If the microprocessor has to write or to read from a device, the chip select (\overline{CS}) signal to that block should be enabled and the address decoding circuit must ensure that chip select (\overline{CS}) signal to other devices are not activated.

Interfacing I/O

- The approach to designing an interface circuit for an I/O device is determined primarily by the instruction to be used for data transfer.
- An I/O device can be addressed using
 - **I/O mapped I/O or peripheral I/O**
 - **Memory mapped I/O**

I/O mapped I/O

- Each microprocessor's instruction set has instructions to communicate with I/O devices.
- In I/O mapped I/O, the I/O devices are addressed according to the I/O instructions defined in instruction set.

For 8085 microprocessor

- A device is identified with an 8 bit address
- Operated by I/O related instruction IN and OUT.
- Status Signal $IO/\overline{M}=1$.
- Either low order address bits A_0-A_7 or high order address bits A_8-A_{15} can be decoded to generate address decoding circuit.
- Can define 256 input and output ports using 8 bits.
- Control signal \overline{IOR} and \overline{IOW} are used.

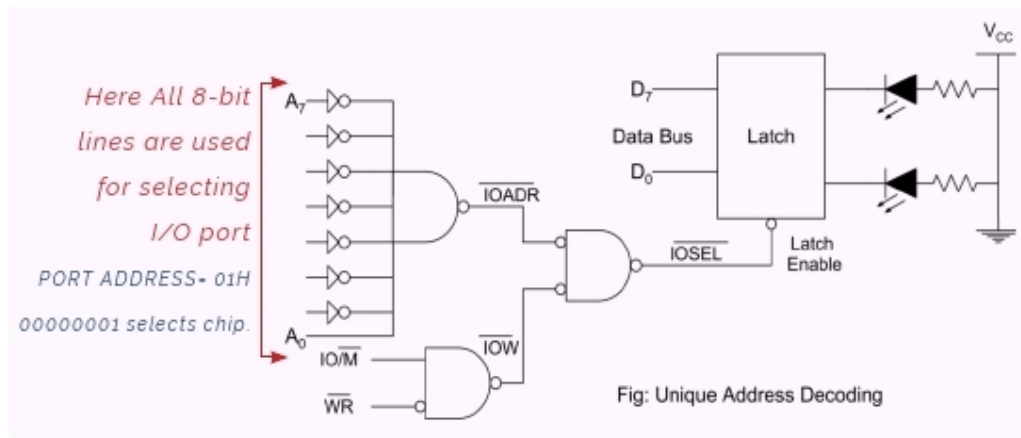
Memory mapped I/O

- The microprocessor communicates with an I/O device as if it were one of the memory location.

For 8085 microprocessor

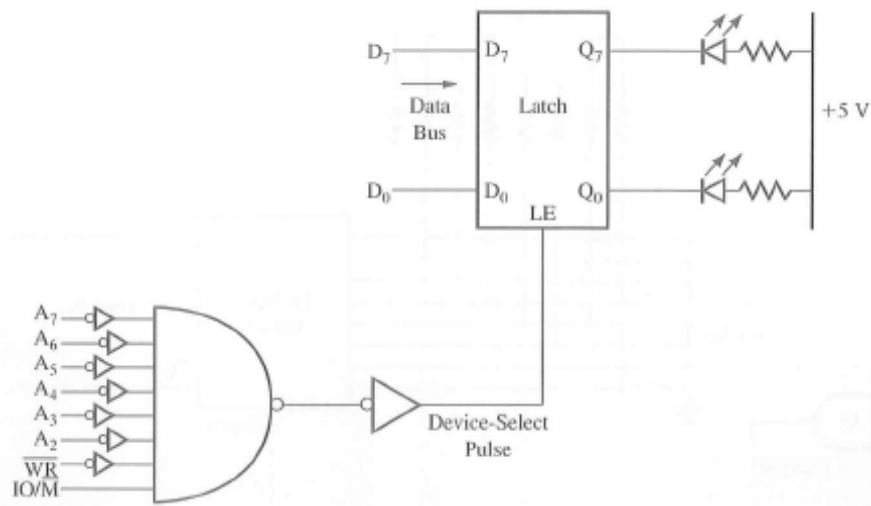
- The input and output devices are assigned and identified by 16 bit address.
- Memory related instructions (such as LDA, STA, MOV M,R etc.) are used to transfer data between microprocessor and I/O devices.
- Status Signal $IO/\overline{M}=0$.
- The control signal \overline{MEMR} and \overline{MEMW} should be connected to I/O devices instead of \overline{IOR} and \overline{IOW} .
- The 16 bit address bus should be decoded.

Unique/Absolute Address Decoding



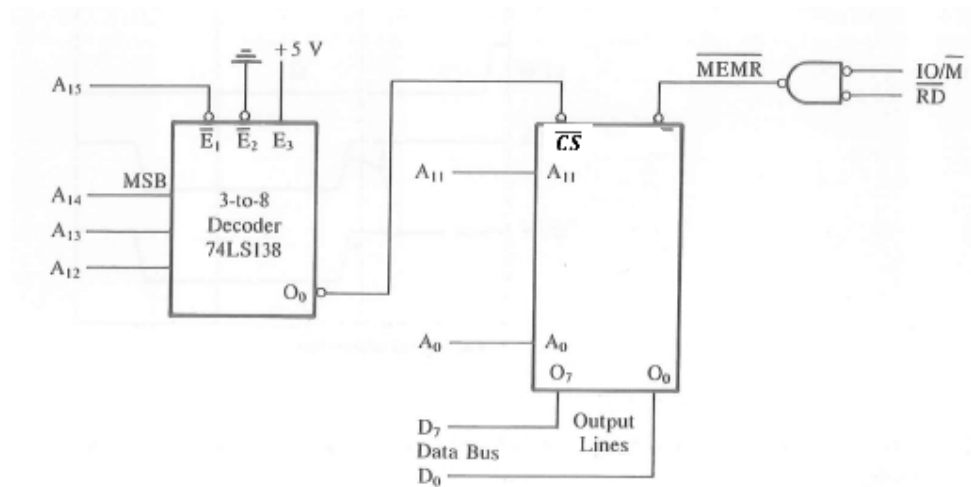
- Each memory location or I/O port corresponds to a unique address value on the address bus
- All the address lines on that mapping mode are used for address decoding. (i.e. 8 lines in I/O mapped I/O, 16 lines in Memory mapped I/O)
- It is expensive and complicated but fault proof in all cases.
- In the above circuit the address of latch (output port) is 01H.
 - The latch is enabled when the address 01H is present in address line (A₇-A₀).
 - Other address can't enable the latch

Non-unique/Partial Address Decoding



- If all the address lines available on that mode are not used in address decoding then that decoding is called non unique address decoding.
- Though it is cheaper there may be a chance of address conflict.
- In the above circuit Address line A_0 and A_1 are not used in address decoding circuit and they can be treated as don't care.
 - This latch can be accessed by address 00H, 01H, 02H and 03H.
- Non unique address decoding is used in small systems where multiple address will not cause any problems, provided these addresses are not assigned to any other output ports.

Address decoding using Decoder



Address	A15	A14	A13	A12	A11	A10	A9	A8	A7	A6	A5	A4	A3	A2	A1	A0	Hex address
Initial	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0000H
Final	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	0FFFH

- The decoder's output line O_0 is used to select the ROM.
- Output O_0 is selected when $A_{15}=0$ and $A_{14} A_{13} A_{12}=000$
- The address of the ROM is from 0000H to 0FFFH

Analysis of above circuit when 8085 places address 0FFFH

- $A_{15} A_{14} A_{13} A_{12} = 0000$ selects the output line O_0 of the decoder i.e $O_0 = 0$
- $O_0 = 0$ enables the \overline{CS} of the decoder
- $A_{11} A_{10} A_9 A_8 A_7 A_6 A_5 A_4 A_3 A_2 A_1 A_0 = 1111 1111 1111$ (FFFH) goes to the address lines of the ROM and the internal decoder of the ROM decodes the address and selects the register with address FFFH.
- When 8085 asserts \overline{RD} signal, the contents of the selected register are placed on the data bus for the processor to read.

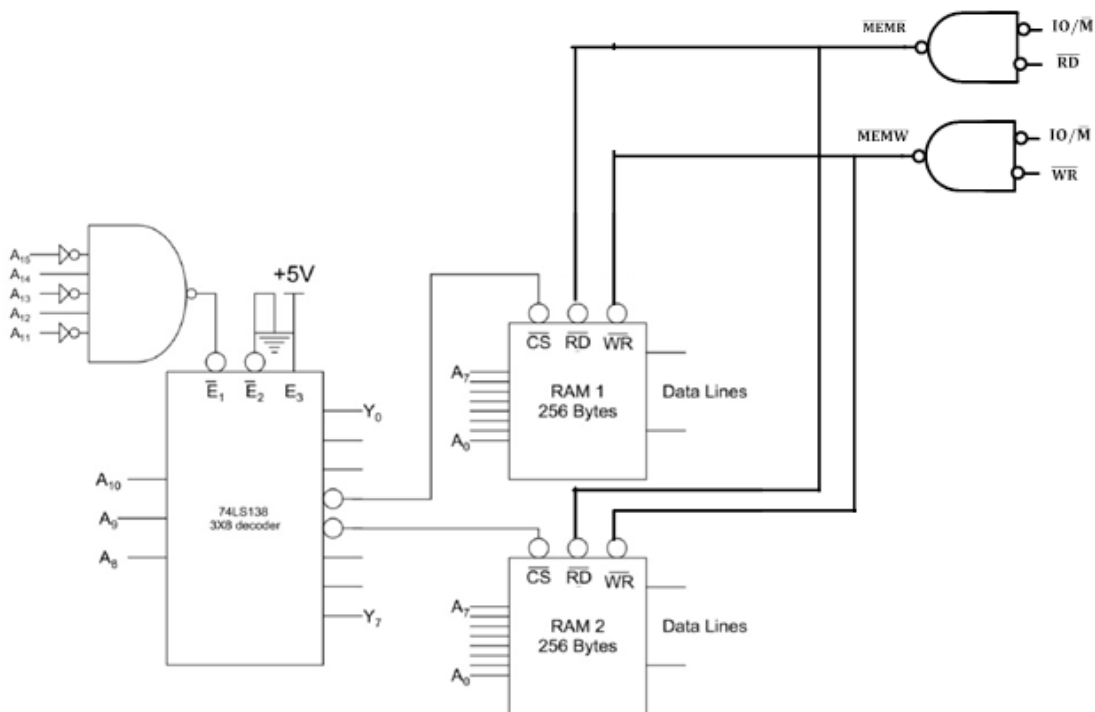
Q.N:01 Design an address decoding circuit for two RAM chips each of 256 bytes at address starting from 5300H.

✓ Required address bit for 256 bytes = $(2^x=256)$, $x = \log(256)/\log(2) = 8\text{bit}$

MEMORY MAPPING																	
Block	Address	A ₁₅	A ₁₄	A ₁₃	A ₁₂	A ₁₁	A ₁₀	A ₉	A ₈	A ₇	A ₆	A ₅	A ₄	A ₃	A ₂	A ₁	A ₀
1	Start	5300H	0	1	0	1	0	0	1	1	0	0	0	0	0	0	0
	End	53FFH	0	1	0	1	0	0	1	1	1	1	1	1	1	1	1
2	Start	5400H	0	1	0	1	0	1	0	0	0	0	0	0	0	0	0
	End	54FFH	0	1	0	1	0	1	0	0	1	1	1	1	1	1	1

Same bits
Decoder Select
Unique bits
Chip Select
8-bit required
Input address lines

- **Same bits** are used in decoder's enable input
- **Unique bits** are used for chip select input of RAM and ROM



Example : Design a minimum system to interface the following specification:

1. 32kB of ROM using 2 x 16kB ROM IC

2. 32kB of RAM using 2 x 16kB RAM IC

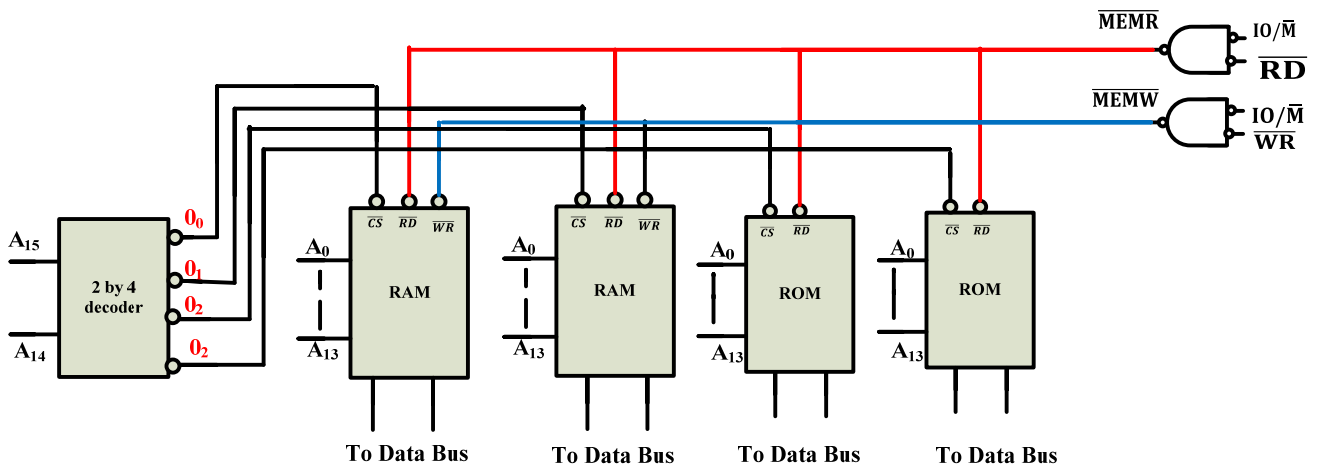
Solution:

Required address bit for 16 kB = ($2^x = 16 \times 1024$)

$x = \log(16384)/\log(2) = 14\text{bit}$

MEMORY MAPPING																	
Block	Address	A ₁₅	A ₁₄	A ₁₃	A ₁₂	A ₁₁	A ₁₀	A ₉	A ₈	A ₇	A ₆	A ₅	A ₄	A ₃	A ₂	A ₁	A ₀
1	Start	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	End	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1
2	Start	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	End	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
3	Start	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	End	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1
4	Start	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	End	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Unique bits Chip Select 14-bit required Input address lines



Example : Design a minimum system to interface the following specification:

i). 74LS138: 3 to 8 decoder

ii). 2732 (4k*8) EP-ROM address range should begin at 0000H and additional 4k memory space should be available for future expansion.


iii). 6116 (2k*8) CMOS R/W Memory

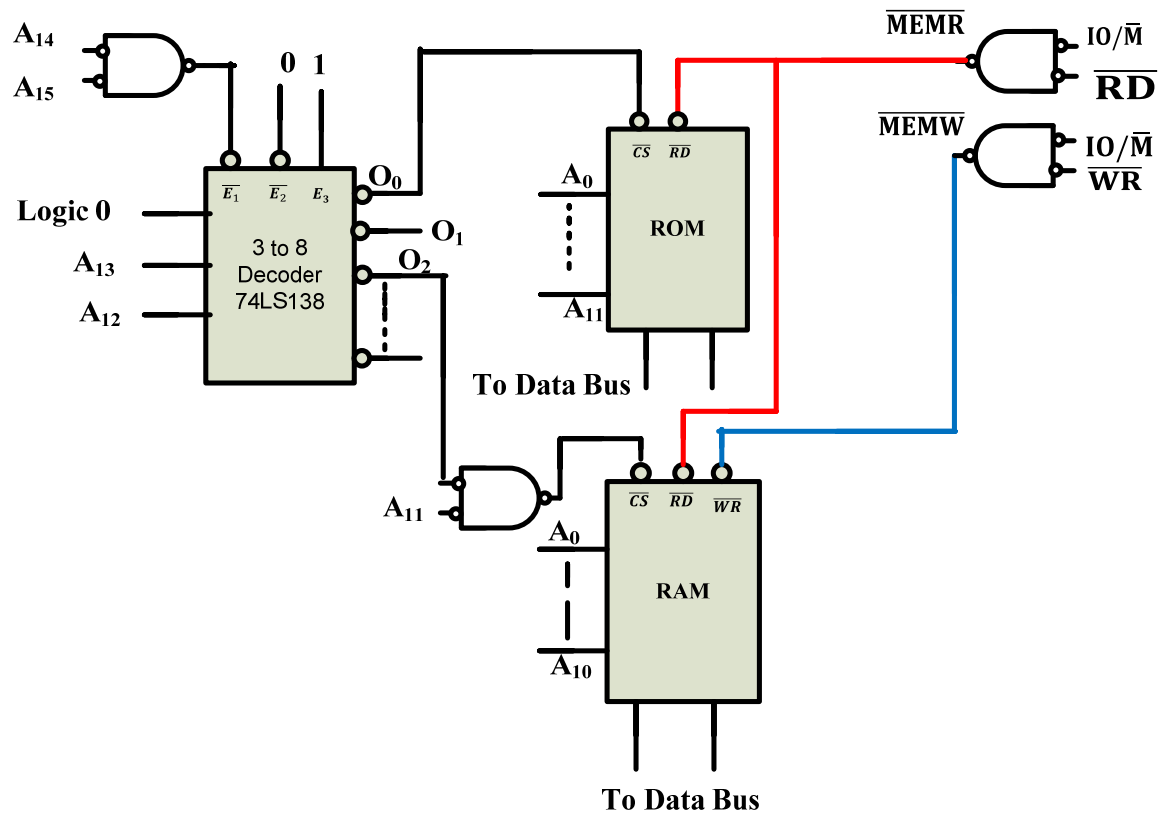
√ required address bit for 4 kB = 12-bit.

√ required address bit for 2 kB = 11-bit.

Block		Address	A15	A14	A13	A12	A11	A10	A9	A8	A7	A6	A5	A4	A3	A2	A1	A0	
1	Start	0000H	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	ROM
	End	0FFFH	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	
2	Start	1000H	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	Future
	End	1FFFH	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	
3	Start	2000H	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	RAM
	End	27FFH	0	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	

unique
 bit chip select 12 bit input address line





Example : Design an address decoding circuit to interface two 4 KB RAM and two 2 KB ROM starting at memory location 1000H.

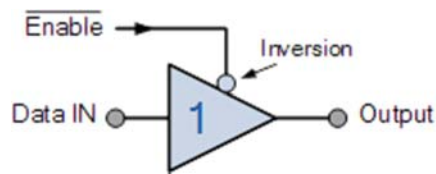
Block		Address	A15	A14	A13	A12	A11	A10	A9	A8	A7	A6	A5	A4	A3	A2	A1	A0	
1	Start	1000H	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	RAM
	End	1FFFH	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	
2	Start	2000H	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	RAM
	End	2FFFH	0	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	
3	Start	3000H	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	ROM
	End	37FFH	0	0	1	1	0	1	1	1	1	1	1	1	1	1	1	1	
4	Start	3800H	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	ROM
	End	3FFFH	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	

Unique
bits



Address lines

Tristate Buffer

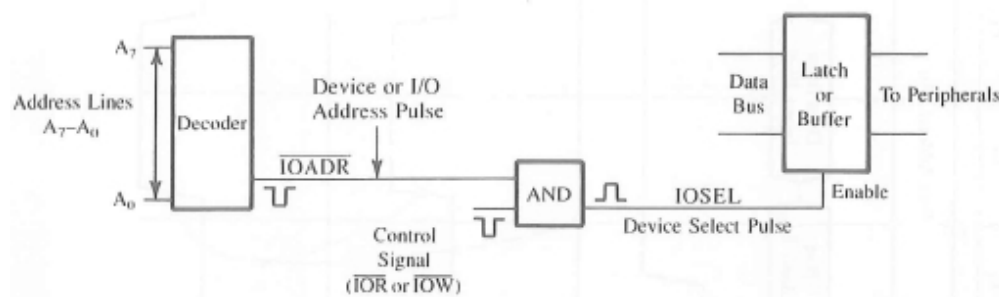


Enable	Data IN	Output
0	0	0
0	1	1
1	0	High impedance
1	1	High impedance

Address decoding for I/O Device

Steps for address decoding

1. Decode the address bus (A_7-A_0) to generate a unique pulse corresponding to the device address on the bus ; this is called **device address pulse or I/O address pulse**
2. Combine (AND) the **device address pulse** with control signal (\overline{IOW} or \overline{IOR}) to generate a device select (**I/O select**) pulse that is generated when both signals are asserted.
3. Use the **I/O select** pulse to activate the I/O port (latch or tri state buffer)



Example: Design an address decoding circuit to interface an input device with 8 input switch and a LED output device. The address of input device is FAH and address of output device is F8H. Design system for following cases

1. Using 3 by 8 decoder
2. Using 2 by 4 decoder
3. Without using decoder

Solution using 3 by 8 decoder

Device	A7	A6	A5	A4	A3	A2	A1	A0
output	1	1	1	1	1	0	1	0
input	1	1	1	1	1	0	0	0

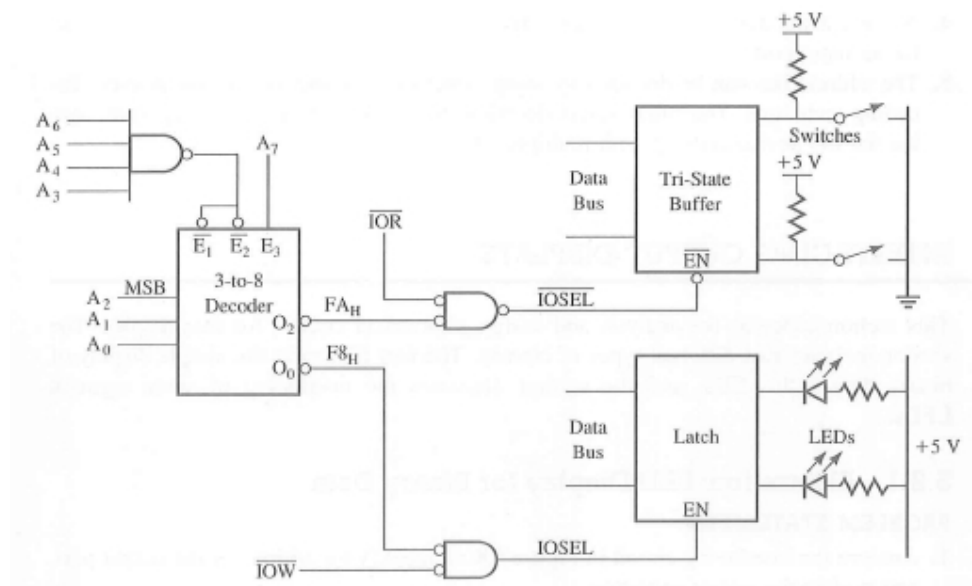
Same bits for decoder

enable

decoder i/p

Since we have to use 3 by 8 decoder

- We use A_2, A_1, A_0 as input of decoder
- We use A_7, A_6, A_5, A_4, A_3 to enable decoder.
- In I/O interfacing we always combine (AND) the decoder output with control signal (\overline{IOR} OR \overline{IOW}) to generate a device (latch or Tri state buffer) select pulse that is generated when both signals are asserted.
- When $O_2 = \overline{IOR} = 0$, Then $\overline{EN} = 0$ and the tri state buffer is activated.
- When $O_0 = \overline{IOW} = 0$, Then $\overline{EN} = 1$ and the latch is activated.

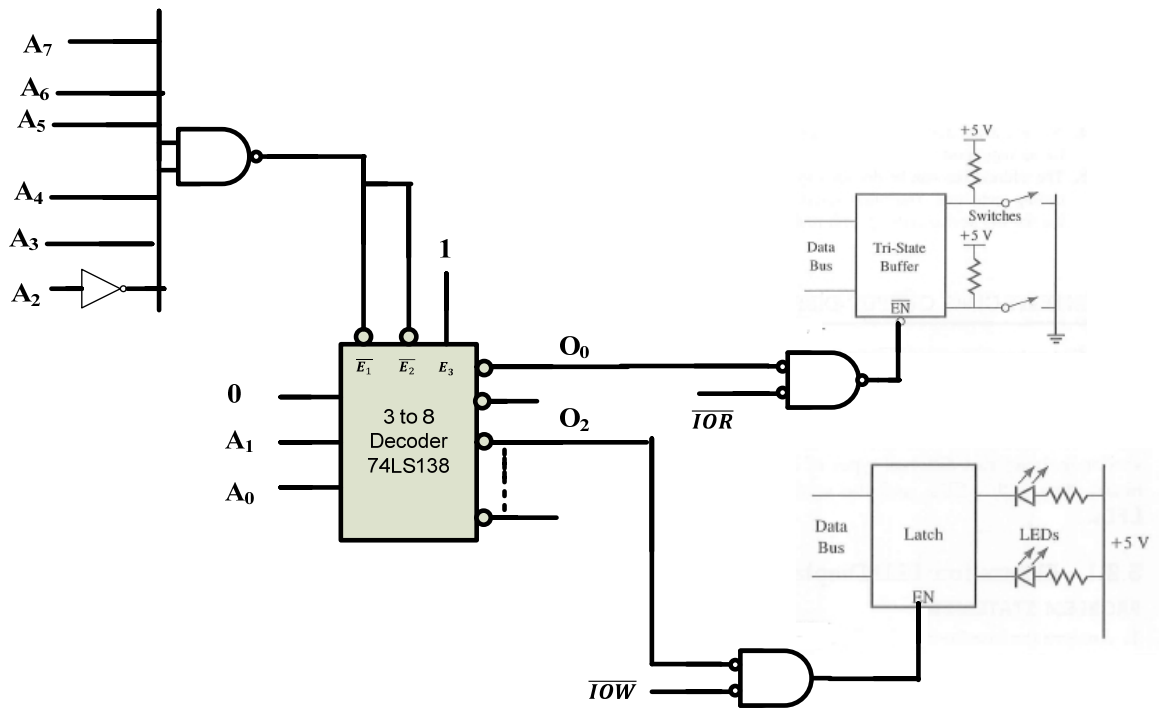


Alternate solution using 3 by 8 decoder

Device	A7	A6	A5	A4	A3	A2	A1	A0
output	1	1	1	1	1	0	1	0
input	1	1	1	1	1	0	0	0

Same bits for decoder
enable

decoder
i/p

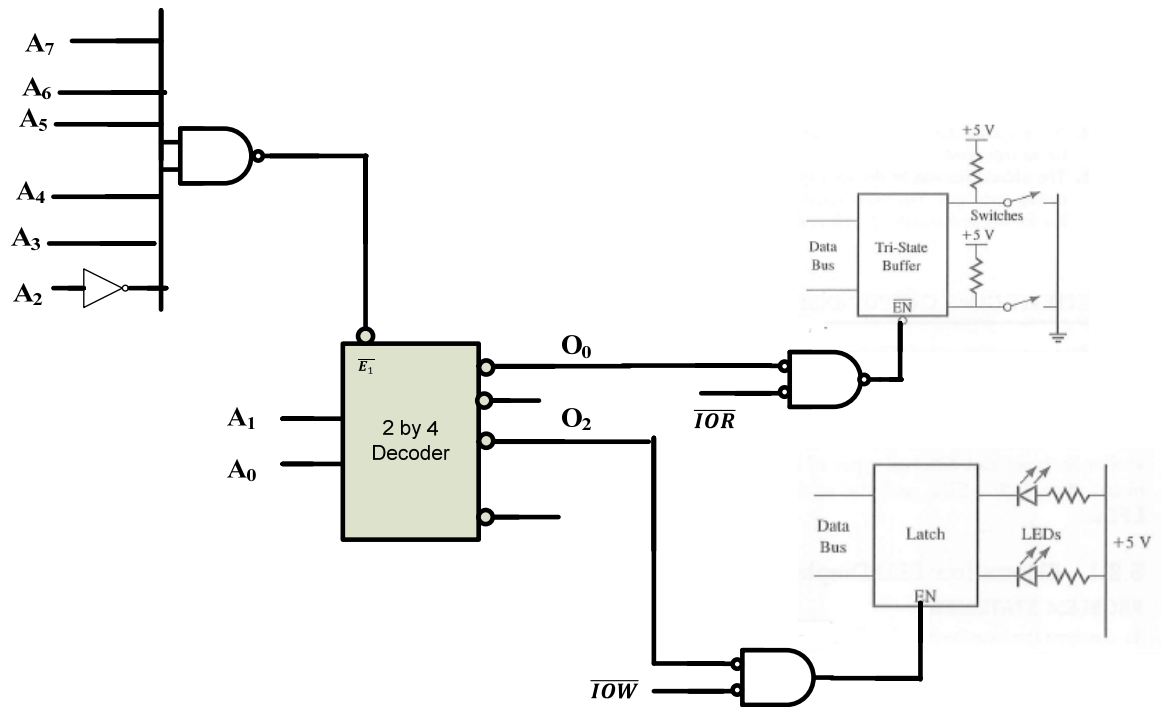


Solution using 2 by 4 Decoder

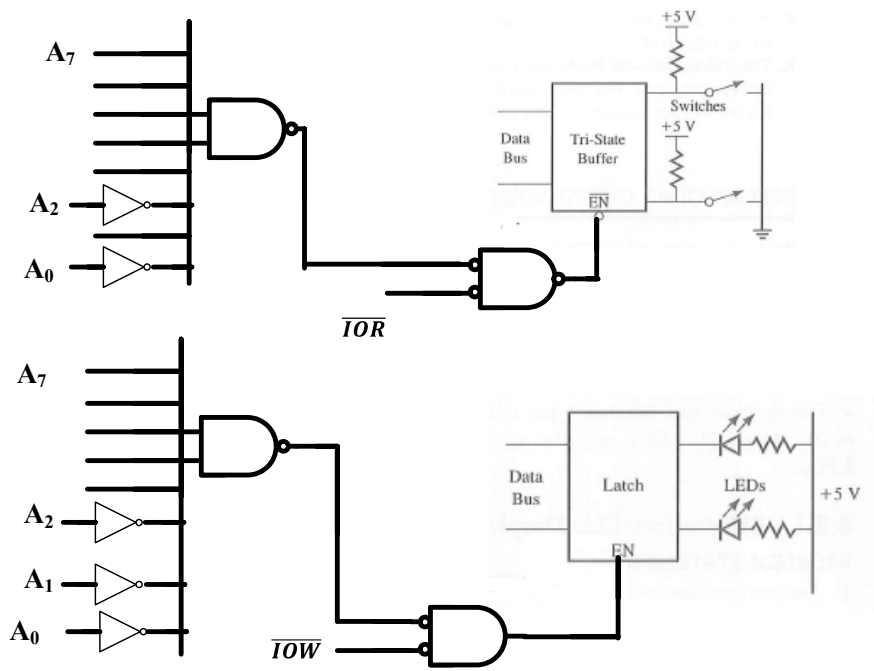
Device	A7	A6	A5	A4	A3	A2	A1	A0
output	1	1	1	1	1	0	1	0
input	1	1	1	1	1	0	0	0

Same bits for decoder
enable

decoder
i/p

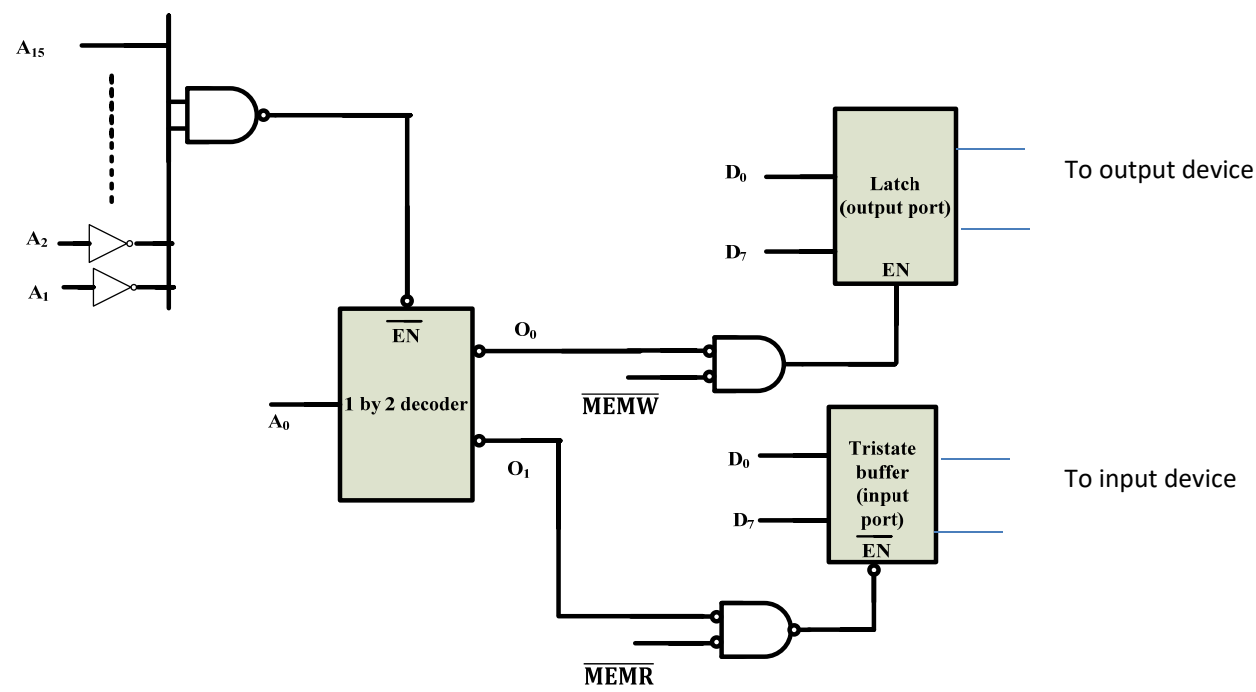


Solution without using Decoder



Example: Design a unique address decoding circuit using memory mapped I/O to interface an input port and an output port with address FFF9H and FFF8H respectively.

Address	A15	A14	A13	A12	A11	A10	A9	A8	A7	A6	A5	A4	A3	A2	A1	A0
FFF9H	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	1
FFF8H	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0



BUS STRUCTURE:

- A bus is a communication pathway connecting two or more devices. A key characteristic of a bus is that it is a shared transmission medium. Multiple devices connect to the bus, and a signal transmitted by any one device is available for reception by all other devices attached to the bus. If two devices transmit during the same time period, their signals will overlap and become garbled. Thus, only one device at a time can successfully transmit.
- A bus actually consists of multiple communication pathway or lines. Each line is capable of transmitting signals representing binary 1 and 0. Several lines of the bus can be used to transmit binary data simultaneously.
- The bus that connects major microcomputer components such as CPU, memory or I/O is called the system bus.
- On any bus the lines can be classified into three functional groups data, address, and control lines

Data bus:

- Group of wires which carries data between the system modules.
- The data bus may consist of 32, 64, 128, or even more separate lines, the number of lines being referred to as the **width of the data bus**. Because each line can carry only one bit at a time, the number of lines determines how many bits can be transferred at a time.

Address bus:

- The address bus is used to designate the source or destination of the data on the data bus.
- For example, if the processor wishes to read a word (8, 16, or 32 bits) of data from memory, it puts the address of the desired word on the address bus.
- the width of the address bus determines the maximum possible memory capacity of the system.
- the address bus is also used to address I/O ports

Control bus:

- Control bus transmit both command and timing information among system modules.
- Timing signals indicate the validity of data and address information.
- Command signals specify operations to be performed.
- Some control bus signals are
 - Memory write: causes data on the bus to be written into the addressed location.
 - Memory read: causes data from the addressed location to be placed on the bus.
 - I/O write: causes data on the bus to be output to the addressed I/O port.
 - I/O read: causes data from the addressed I/O port to be placed on the bus.
 - Bus request: indicates that a module needs to gain control of the bus.
 - Clock: is used to synchronize operations.

Bus Types

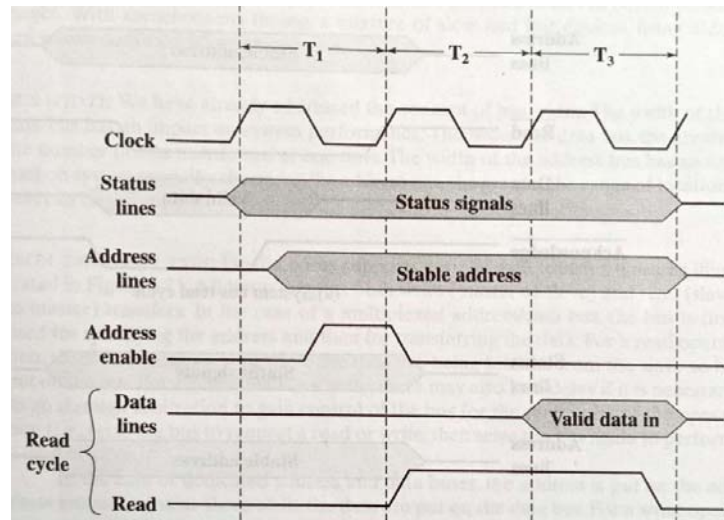
Timing refers to the way in which events are coordinated on the bus. Buses use either synchronous timing or asynchronous timing. Based upon timing bus may be of two types;

- a. Synchronous Bus
- b. Asynchronous Bus

Synchronous Bus

- The occurrence of events on the bus is determined by a clock i.e everything is synchronized to bus clock.
- The bus includes a clock line upon which a clock transmits a regular sequence of alternating 1s and 0s of equal duration.
- A single 1-0 transmission is referred to as clock cycle or bus cycle.
- All devices on the bus can read the clock line.
- All the bus actions are done on fixed clock cycles known in advance to both source and destination units.
- When connecting devices with varying speeds to a synchronous bus, the slowest device will determine the speed of the bus.

Read operation



1. Processor places a memory address on address lines during first clock cycle and may asserts various status lines.
2. Once the address lines have stabilized, the processor issues an address enable signal
3. For a read operation
 - a. Processor issues a read command at the start of second cycle
 - b. After a delay of one cycle, memory module places data on the data lines
 - c. Processor reads the data from data lines and drops the read signal.

Advantage:

- Synchronous bus is simple to implement.
- It is fast

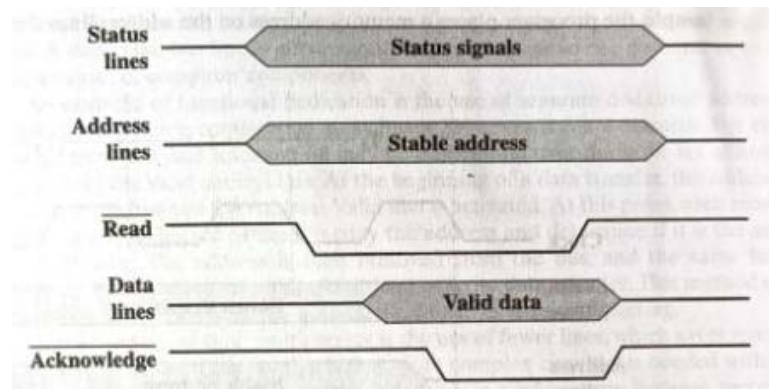
Disadvantage

1. Because all devices are tied to a fixed clock rate, the system can't take advantage of advances in device performance.
2. The bus must be short due to clock skew.

Asynchronous bus

- An asynchronous bus has no system clock
- The occurrence of one event on a bus follows and depends on the occurrence of a previous event.
- Handshaking is done to properly conduct the transmission of data between the sender and the receiver

Read operation



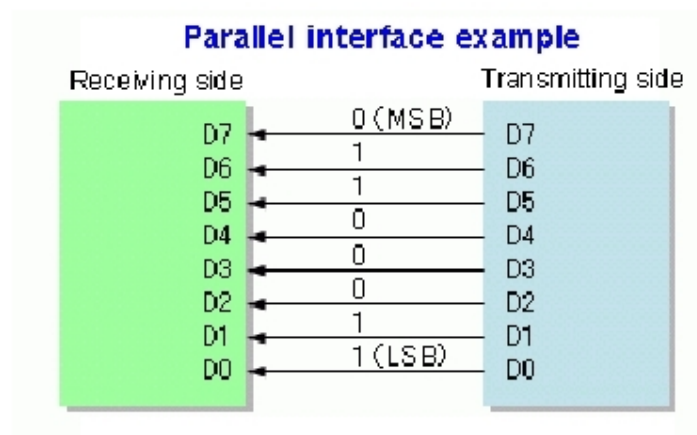
1. Processor places address and status signals on the bus.
 2. After pausing for these signals to stabilize, processor issues a read command, indicating the presence of valid address and control signals
 3. The appropriate memory decodes the address and responds by placing data on data line
 4. Once the data line have stabilized, the memory module asserts the acknowledged line to signal the processor that the data are available.
 5. Once the processor has read the data, it deasserts the read signal.
 6. This causes memory module to drop the data and acknowledge lines.
 7. Finally, once the acknowledge line is dropped, the processor removes address information.
-
- Asynchronous buses work well even when they are long because clock skew problems do not affect them
 - With asynchronous bus, a mixture of slow and fast devices, using older and newer technology, can share a bus.
 - It is slower than synchronous bus.

Modes of data transfer

- The microprocessor receives (or transmits) binary data in either of two modes; **parallel or serial**.
- Devices commonly used for parallel data transfer are keyboards, seven segment LEDs ,data converters and memory
- The serial mode is commonly used with peripherals such as CRT, printers etc.

Parallel Interface

- A parallel interface refers to multiline channel, which is capable of transmitting several bits of data simultaneously.
- N bits of data are transferred simultaneously by using N wires
- Expensive due to need of multiple wires.



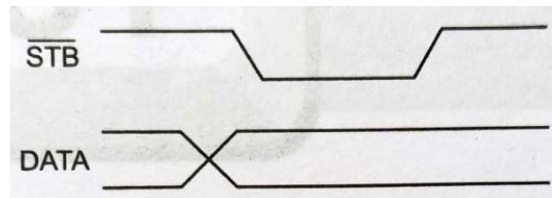
Method of parallel data transfer

Simple input and output

- Microprocessor assumes that the peripherals are always available.
- Example: when you need to output data to a simple display device such as an LED, all you have to do is connect the input of the LED buffer on an output port. The LED is always there and ready, so you can send data to it at any time.
- The devices are always ready to send or receive data.

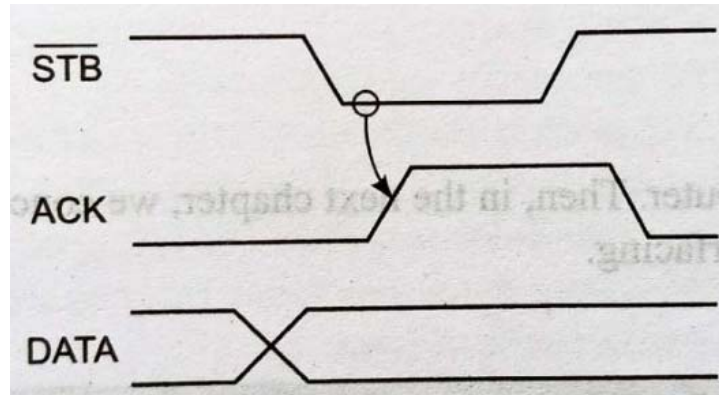


Simple strobe I/O (Wait Interface)



- In many applications, valid data is present on an external device only at a certain time, so it must be read in at that time.
- In this method, the sending device outputs parallel data on data lines and then outputs an \overline{STB} (Strobe) signal to indicate that valid data is present.
- Microprocessor should wait until peripheral asserts an active low strobe signal so it is also known as simple wait I/O.
- This method is suitable for low rates of data transfer such as from a keyboard to microprocessor
- For higher speed data transfer this method does not work because there is no signal which tells the sending device when it is safe to send the next data byte.
- **In such case handshake data transfer is used.**

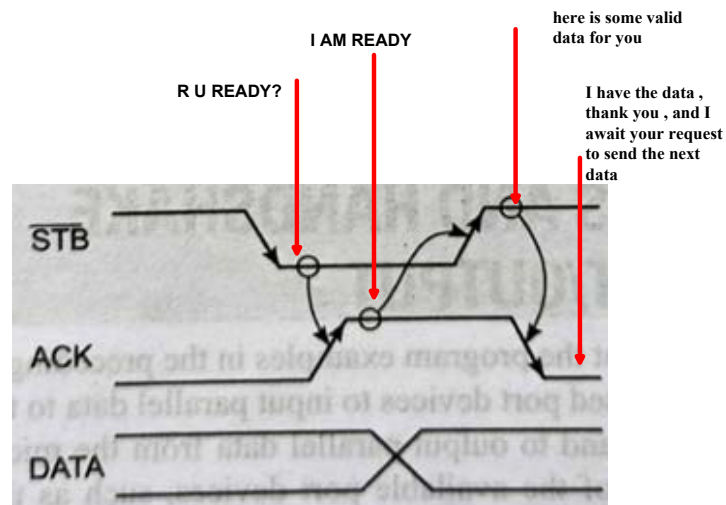
Single handshake I/O



Steps for single handshake

1. Sending device outputs some parallel data and sends a \overline{STB} signal to receiving device.
2. The receiving device detects the asserted \overline{STB} signal and reads in the data.
3. Receiving device sends an acknowledge signal (ACK) to Sending device to indicate that the data has been read and that the Sending device can send the next byte of data.

Double handshake data transfer



Steps in double handshaking

1. The sending device asserts its \overline{STB} line low to ask “Are you ready?”
2. The receiving device raises its ACK line high to say “I am ready”

3. The sending device then sends data and raises its \overline{STB} line high to say “here is some valid data for you”
4. After the receiving device has read in the data , the receiving device drops it ACK line low to say “I have the data , thank you , and I await your request to send the next data”

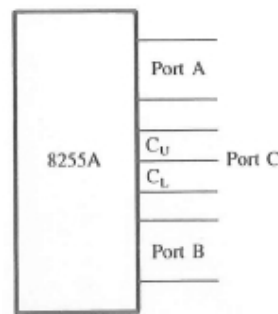
For a handshake output of this type , from a microprocessor to a peripheral, the waveform are the same , but the microprocessor sends the \overline{STB} signal and the data, and the peripheral sends the ACK signal

Programmable peripheral Interface

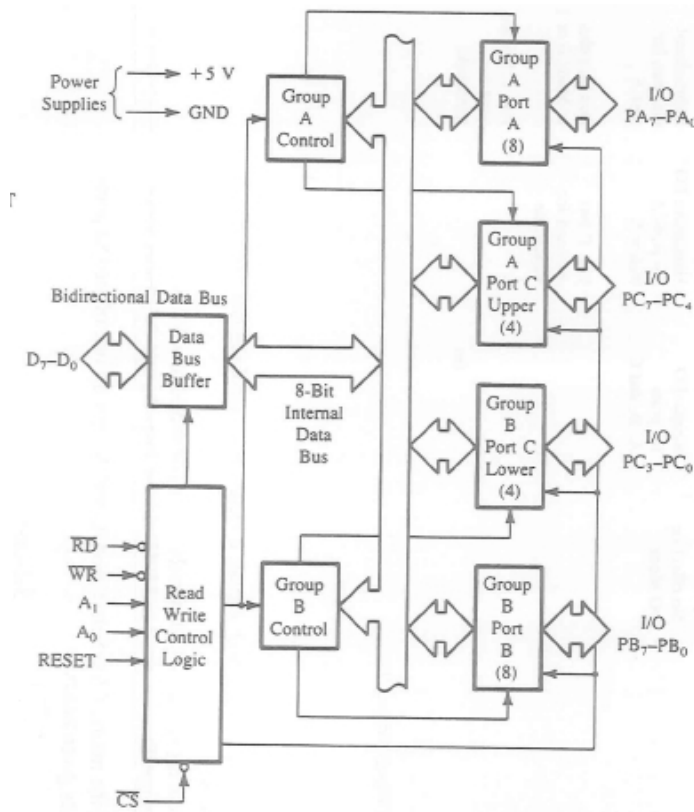
- We can use latch and tri state buffer as output port and input port.
- Latch and tri state can be connected to only one device and they are hardwired.
- A programmable peripheral interface is designed to perform various input output functions.
- It can be set up to perform specific function by writing an instruction (or instructions) in its internal register called control word.
- It is hardware approach through software control to performing the I/O functions.
- A programmable peripheral interface device can be viewed as multiple I/O devices.

8255A (Programmable peripheral Interface)

- It is programmable, parallel I/O device.
- It can be programmed to transfer data under various conditions, from simple I/O to interrupt I/O.
- It has 24 I/O pins that can be grouped as three 8 bit ports
 - **Port A** : can be used as an 8 bit input port or as an 8 bit output port
 - **Port B** : can be used as an 8 bit input port or as an 8 bit output port
 - **Port C**: can be used as an 8 bit input or output port, as two 4 bit port C_U (PC_7-PC_4) and C_L (PC_3-PC_0) or to produce handshake signals for ports A and B.
 - The three ports can be grouped as
 - **Group A** : port A and C_U
 - **Group B** : port B and C_L
 - The function of these ports can be defined by writing control word in control register.



Block Diagram of 8255A



Read write Control Logic:

It manages all of the internal and external transfers of both data and control word. \overline{RD} , \overline{WR} , RESET, A₁ and A₀ are the inputs, provided by microprocessor to the Read Write control logic

\overline{RD} (Read): When the signal is low, the MPU reads data from a selected I/O port of 8255A

\overline{WR} (Write): When the signal goes low, the MPU writes into a selected I/O port or the control register.

RESET: This is active high signal that clears the control register and sets all ports in the input mode.

\overline{CS} , A₀ and A₁:

- These are device select signals
- Asserting \overline{CS} input of the 8255A enables it for reading or writing.
- \overline{CS} is connected to the output of the address decoding circuit to select the device when it is addressed.
- A₀ and A₁ are connected to the MPU address line A₀ and A₁ respectively

\overline{CS} \overline{CS}	A1	A0	Selected
0	0	0	Port A
0	0	1	Port B
0	1	0	Port C
0	1	1	Control register
1	X	X	8255A is not selected

Data Bus Buffer

- It is used to interface the 8255 internal data bus with external system data bus.
- It receives or transmits data upon the execution of input or output instructions by the microprocessor.
- It is controlled by Read Write control logic.

Control word

- 8255 has a control register.
- Functions of the ports of 8255 can be controlled by programming the bits of control register.
- The content of this register is called control word.
- Control word specify an I/O function for each port.
- This register can be accessed to write a control word when A₀ and A₁ are at logic 1.
- Bit D7 of the control register specifies either the BSR mode or I/O mode.

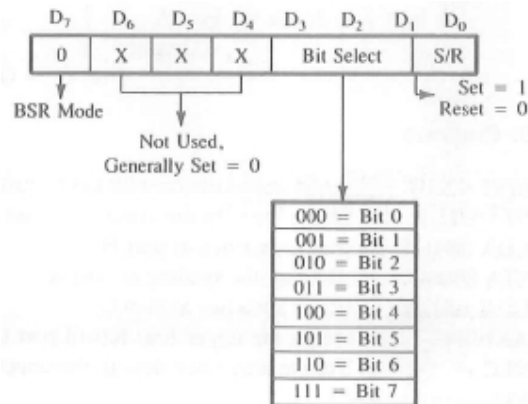
Modes of 8255A

- It operates in two modes
 - **Bit Set/Reset (BSR) mode**
 - **I/O mode:** In this mode the 8255 ports work as programmable I/O ports. I/O mode is divided into 3 modes
 - Mode 0
 - Mode 1
 - Mode 2

BSR mode

- **BSR mode** is concerned only with 8 bits of port C, which can be set or reset by writing appropriate control word in control register.
- A control word with bit **D7=0** is BSR control word and it does not alter any previously transmitted control word with bit D7=1 : thus the I/O operation of port A and B are not affected by a BSR control word.

- In BSR mode, individual bits of port C can be used for applications such as an on/off switch.



BSR control word

- $D_7D_6D_5D_4D_3D_2D_1D_0 = 00001111$ sets the bit C_7
- $D_7D_6D_5D_4D_3D_2D_1D_0 = 00001110$ resets the bit C_7

I/O Mode

- Port A and C_U can be programmed for any of the modes 0 through 2.
- Port B and C_L can be programmed to operate in mode 0 or 1 only.

Mode 0: Simple input or output

- When you want to use for simple input or output without handshaking, you initialize that port in mode 0.
- In this mode, Port A and B are used as two simple 8 bit I/O ports and port C as two 4 bit ports.
- Each port (or half port in case of C) can be programmed to function as simply an input port or an output port.
- The input/output features in Mode 0 are as follows;
 - Output ports are latched
 - Input ports are not latched
 - Ports do not have handshake or interrupt capability.

Mode 1: input or output with handshake

- When you want to use Port A or Port B for a handshake input or output operation, you initialize that port in mode 1.
- In this mode, handshake signals are exchanged between the MPU and peripherals prior to the data transfer.
- The features of this modes includes;

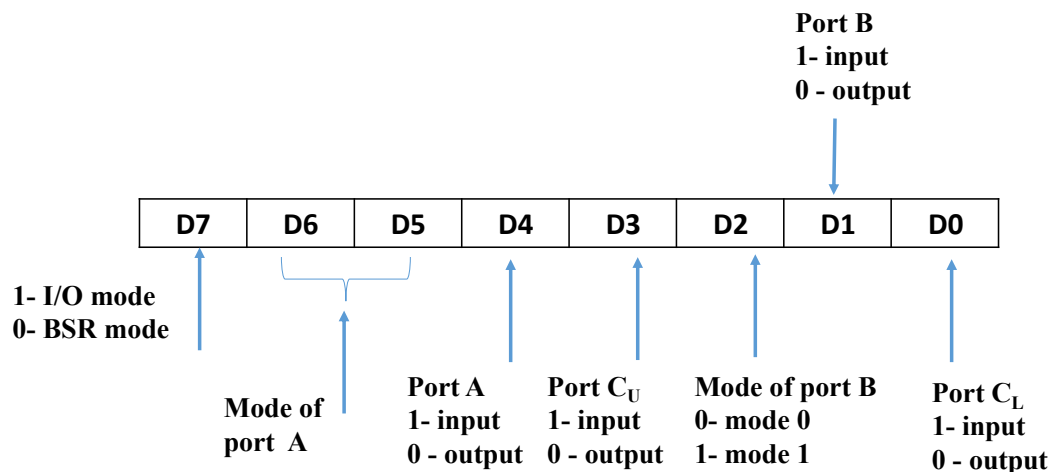
- Ports A and B function as 8 bit I/O ports. They can be configured either as input or output ports.
- Each port uses **three lines** from port C as handshake signals. The remaining two lines of port C can be used for simple I/O functions.
- Input and output data are latched
- Interrupt logic is supported.

Mode 2: Bidirectional data transfer

- Only port A can be initialized in mode 2.
- In this mode, port A can be configured as the bidirectional port.
- Port A uses **5 signals from port C as handshake signals for data transfer**
- **When Port A is in mode 2 , port B can be in mode 0 or in mode 1.**
- The remaining 3 signals from port C can be used either as simple I/O or as handshake for port B.

Control word for I/O mode

- The content of the control register is known as control word.
- Control word specify an I/O function for each port.
- Control register can be accessed to write a control word.
- Bit D7 of control register specify either the I/O or BSR function.
 - If D7=1, the bits D6-D0 determine I/O functions in various mode
 - If D7=0, port C operates in the BSR mode.



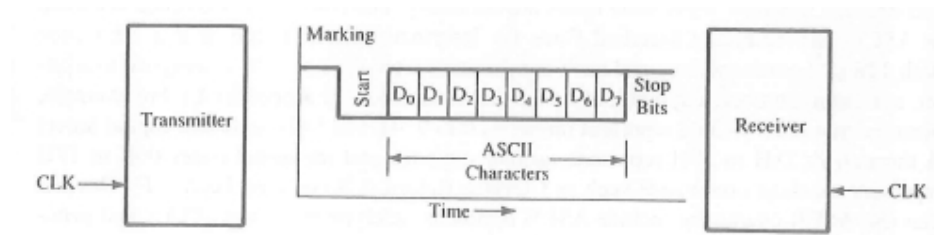
D₆	D₅	Mode of port A
0	0	Mode 0
0	1	Mode 1
1	X	Mode 2

Serial Interface /Serial Data transmission

- In serial data transmission , the data are sent one bit at a time over the transmission channel
- Slow due to single communication link but inexpensive to implement.
- In many situations, the parallel I/O mode is either impractical or impossible.
- Parallel data communication over a long distance can be very expensive.
- In such cases, the serial I/O mode is used, whereby one bit at a time is transferred over a single line.
- In serial transmission (from MPU to peripheral), an 8 bit parallel word should be converted into a stream of eight serial bit; this is known as parallel to serial conversion.
- After conversion, one bit at a time is transmitted over a single line at a given rate called baud rate.
- On the other hand, in serial reception, the MPU receives a stream of 8 bits and they should be converted into an 8 bit parallel word; this is known as serial to parallel conversion.
- The interface requirement for a serial I/O peripheral are the same as for a parallel I/O device. The primary difference is that parallel I/O uses entire data bus and serial I/O uses one data line.
- The serial binary information transmitted consists of binary coded characters.
- The characters may represent alphabets, decimal digits, graphic symbols etc.
- The no. of bits assigned to each character may be between 5 and 8.

Asynchronous serial data transmission

- Clock is not shared with the transmitter and receiver.
- Asynchronous transmission is a character framing scheme i.e the data are transmitted so that each character is framed with markers pointing the beginning or end of the character.
- The framing is done by the use of start and stop bits.
- Transmission begins with one start bit (low), followed by a character and one or two stop bits (high).
- When no data is being sent, the signal line is in a constant high.
- Asynchronous transmission is used in low speed transmission (less than 20 k bits/sec)



- above figure shows transmission of 11 bits for an ASCII character with
 - 1 start bit
 - 8 character bits
 - 2 stop bits

Synchronous serial data transmission

- The start and stop bits of asynchronous communication represent wasted overhead bits that reduce the overall character rate.
- In synchronous transmission, a large block of data characters is sent from transmitter to the receiver.
- Instead of start and stop bits some **control words** such as SYN, STX, SOH etc. are sent to assure orderly data transfer between the transmitter and receiver.

SYN	SYN	SYN	SYN	SOH	ADR	STA	FC1	FC2	STX	block of character	ETB	ETR	LRC	EOT
-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	--------------------	-----	-----	-----	-----

- SYN (synchronizing character):
 - To indicate the start of transmission, the transmitter sends out one or more SYN characters.
 - The receiver uses SYN characters to synchronize its internal clock with that of a transmitter.
- Transmitter and receiver share a common clock.
- In long distant serial transmission, transmitter and receiver is driven by a separate clock of the same frequency.
- Synchronous transmission is used for high speed transmission (**more than 20 k bits/sec**)

Format	Synchronous	Asynchronous
Data Format	Groups of character	One character at a time
Speed	More than 20 k bits/s	Less than 20 k bits
Framing information	Syn characters are sent with each group	Start and stop bits with every character
Implementation	Hardware	Hardware or Software
Data Direction	Simplex , Half and Full duplex	Simplex , Half and Full duplex

Simplex and duplex transmission

Simplex transmission:

- Data are transmitted in only one direction.
- Eg: commercial radio station

Hal duplex transmission:

- Communication takes place in both the directions but not simultaneously.
- Eg: two way radio system

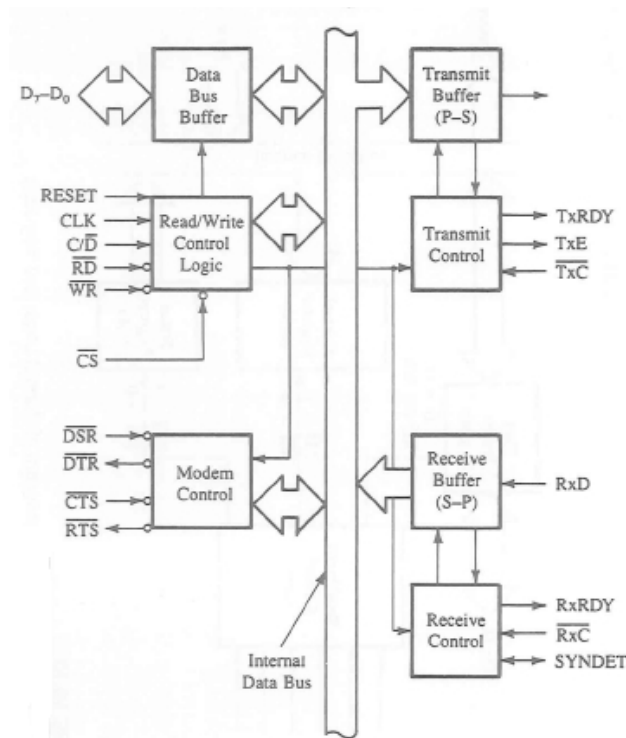
Full duplex transmission:

- Simultaneous transmission in both direction
- Eg: phone conversation

Universal Synchronous/Asynchronous Receiver/Transmitter (USART)

The 8251 is a USART (Universal Synchronous Asynchronous Receiver Transmitter) for serial data communication. As a peripheral device of a microcomputer system, the 8251 receives parallel data from the CPU and transmits serial data after conversion. This device also receives serial data from the outside and transmits parallel data to the CPU after conversion.

- **8251A** is USART device
- Its chip has 28 pin
- The 8251A is a programmable chip designed for synchronous and asynchronous serial data communication.
- Its block diagram includes
 - Read/Write control logic
 - It interfaces the chip with MPU
 - Determines the functions of the chip according to the control word in its register
 - Monitors the data flow
 - Transmitter
 - It converts a parallel word received from the MPU into serial bits and transmits them over the TxD line to a peripheral
 - Receiver
 - It receives serial bits from a peripheral, converts them into a parallel word, and transfer the word to the MPU.
 - Data bus buffer
 - It interfaces the internal bus of circuit with system bus.
 - Modem control
 - It is used to establish data communication through modems over telephone line.



Read/Write Control Logic and Registers

The input to the control logic are;

\overline{CS} (chip select): When this signal goes low, 8251A is selected by MPU for communication.

C/\overline{D} (Control/Data):

- When this signal is high, the **control register or status register** is addressed.
- When this signal is low, the **data buffer** is addressed.

\overline{WR} (Write): When this signal goes low, the MPU either writes in the control register or sends

Output to data buffer.

\overline{RD} (Read): When this signal goes low, the MPU either reads a status from the status register or

accepts data from the data buffer.

RESET: A high on this input resets 8251A and forces it into the idle mode.

CLK (Clock):

- This is clock input
- It is used to generate internal device timings.

Control Register:

- 16 bit register for control word
- To implement serial communication, the MPU must inform 8251 A of all the details. Therefore, prior to the data transfer, a set of control word must be loaded into the control register.
- This register can be accessed as an output port when C/\bar{D} pin is high

Status Register:

- This input register checks the ready status of the peripheral
- It is addressed as an input port when C/\bar{D} pin is high
- It has same port address as the control register

Data buffer:

- This is bidirectional register
- It can be addressed as an input port and an output port when C/\bar{D} pin is low

\overline{CS}	C/\bar{D}	\overline{RD}	\overline{WR}	Function
0	1	1	0	MPU writes instructions in the control register
0	1	0	1	MPU reads status from the status register
0	0	1	0	MPU outputs data to the Data Buffer
0	0	0	1	MPU accepts data from the Data Buffer
1	X	X	X	USART is not selected

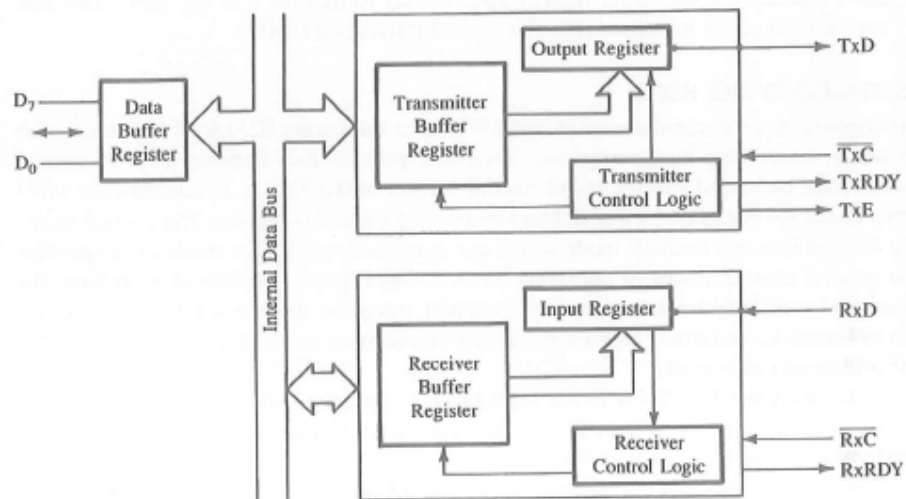
Transmitter Section

- The transmitter section accepts parallel data from MPU and converts them into serial data
- The transmitter section two register
 - Buffer register : holds 8 bits
 - Output register: converts the 8 bits into a stream of serial bits
- MPU writes a byte in the buffer register
- Whenever the output register is empty, the contents of the buffer register are transferred to the output register.
- This section transmits data on TxD pin
- Signals associated with transmitter sections are
 - TxD (Transmit data): serial bits are transmitted on this line
 - $\overline{Tx\bar{C}}$ (Transmitter clock): this input signal controls the rate at which bits are transmitted by USART.
 - TxRDY (Transmitter Ready):

- This is an output signal
- When high: indicates that the buffer register is empty and USART is ready to accept a byte.
- This signal is reset when a data byte is loaded into the buffer.
- TxE (Transmitter empty):
 - This is an output signal
 - When high: indicates that output register is empty
 - This signal is reset when a byte is transferred from the buffer to the output register.

Receiver Section

- The receiver section accepts serial data on RxD line from peripheral and convert them into parallel data.
- This section has two register
 - Receiver input register
 - Buffer register
- In Asynchronous mode following signals are necessary
 - RxD (Receive Data): Bits are received serially on this line and converted into a parallel byte in the receiver input register.
 - $\overline{\text{RxC}}$ (Receiver Clock) :
 - It is a clock signal
 - Controls the rate at which bits are received by USART.
 - RxRDY (Receiver Ready):
 - This is an output signal
 - When high: USART has a character in buffer register and is ready to transfer it to MPU.



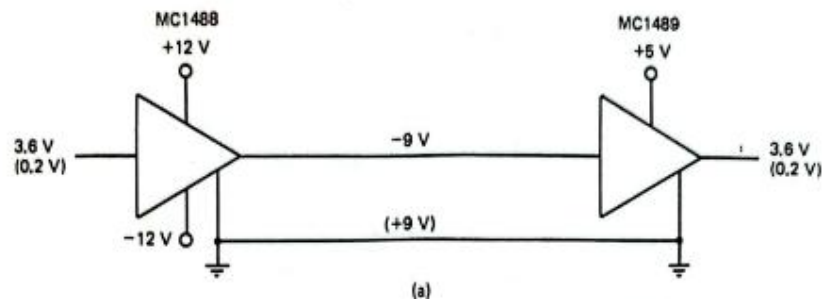
Baud (rate of transmission)

- The rate at which the bits are transmitted (bits/sec) is called a baud.

Standards in serial I/O

- The serial I/O technique is commonly used to interface terminals, printers and modems.
- These peripherals and computers are manufactured by various manufacturers. Therefore, a common understanding must exist, among various manufacturing and user groups that can ensure compatibility among different equipment.
- When this understanding is defined and generally accepted in industry, it is known as a standard.
- In serial I/O, data can be transmitted as either current or voltage.
- When data are transmitted as voltage following standards are used
 - RS 232C
 - RS 422A
 - RS 423A

RS 232C



- Devices used in RS 232 communication are called
 - Data terminal equipment (DTE): The terminals and computers that are sending or receiving serial data.
 - Data Communication Equipment (DCE): Modems and other equipment used to send serial data over long distances.
- Rate of data transmission is restricted to maximum of 20Kbaud and a distance of 50 ft.
- For data line
 - Logic 0 : +3V to +15V
 - Logic 1: -3V to -15V
 - Usually +/- 12 V is used.
- Since the voltage levels are not compatible with TTL logic; line drivers and line receivers are required.
- Line driver MC1488 converts TTL level to RS 232 level
 - TTL Logic 1 into -9V
 - TTL Logic 0 into +9V
- Line Receiver MC1489 converts RS232 to TTL level.
- DTE has a male connector and DCE has a female connector
- Connectors used in RS 232
 - 25 pin male and female connector
 - Many of the 25 pins are not used so 9 pin male and female connectors are also available.

- 6 active low signals are used to control the transfer of data between a DTE and DCE. The definition of each of these signals are as follows;
1. Data terminal ready (\overline{DTR})
 - After the terminal power is turned on and terminal runs any self-check, DTE asserts \overline{DTR} to tell the modem that it is ready.
 2. Data Set Ready (\overline{DSR})
 - This is asserted by DCE when the DCE is ready to transmit or receive data.
 3. Data carrier detect (\overline{DCD})
 - Asserted by the DCE to indicate that it has establish contact with the remote DTE.
 - Modem (DCE) dials up the remote DTE. When the remote DTE answers, it places a high pitched tone (the carrier) on the line, causing the \overline{DCD} signal to become active.
 4. Request to send (\overline{RTS})
 - Asserted by the DTE when it is ready to send a character.
 5. Clear to send (\overline{CTS})
 - This signal is output by DCE and acknowledges \overline{RTS} .
 - \overline{CTS} Indicates that the DCE is ready for transmission..
 6. Ring indicator (\overline{RI}) :
 - This signal is output by the DCE and is active in synchronism with telephone ring signal.
 - It is indication of DCE to the DTE that a ring indicator signal is received.

RS-232C Signals Used with Handshake Data Communication

Signals ^a		Functions
Transmitted Data	TxD	Output; transmits data from DTE to DCE
Received Data	RxD	Input; DTE receives data from DCE
Request to Send	RTS	General-purpose output from DTE
Clear to Send	CTS	General-purpose input to DTE; can be used as a handshake signal
Data Set Ready	DSR	General-purpose input to DTE; can be used to indicate that DCE is ready
Signal Ground	GND	Common reference between DTE and DCE
Data Carrier Detect	DCD	Generally used by DTE to disable data reception
Data Terminal Ready	DTR	Output; generally used to indicate that DTE is ready

^aSignals are referenced to DTE.

Sequence of signals that would occur between a DTE and a DCE when a person at a DTE intends to transfer some information to other DTE via modems and telephone lines.

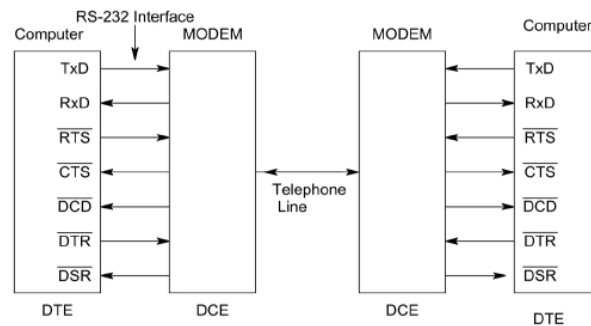
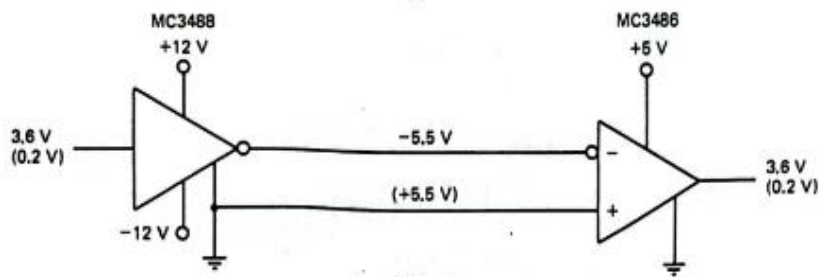


Fig: Digital Data transmission using MODEM and Telephone Line

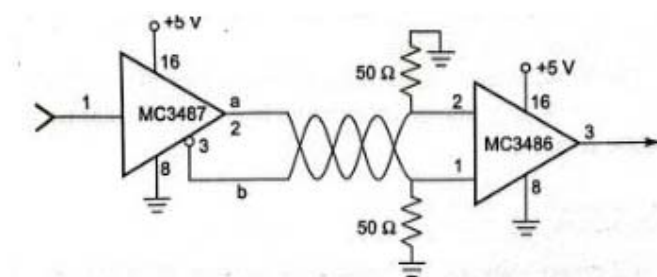
- After the terminal power is turned on and terminal runs any self-check, DTE asserts **DTR** to tell the modem that it is ready.
- If modem is present and turned on, it responds with **DSR**
- Modem dials up the remote DTE. When the remote DTE answers, it places a high pitched tone (the carrier) on the line, causing the **DCD** signal to become active.
- To begin data transfer, the DTE makes **RTS** active and waits for the modem to respond with **CTS**
- When **CTS** is activated by modem the terminal then sends serial data to the modem
- when sending is completed
 - DTE de-asserts **RTS** to indicate all characters have been sent
 - Modem de-asserts **CTS** and stops transmission

RS 423A



- Problem of RS 232
 - A major problem with RS 232 is that it can only transmit data reliably for about 50 ft at its maximum baud rate 20K Bd.
 - If longer lines are used, the transmission rate reduces drastically.
 - One of the reason that RS 232 is restricted to relatively short cable lengths is that the drivers and receivers are **unbalanced (single ended)**. This means the input and output signals are referenced to a common ground.
 - Because it is likely that the receiving and transmitting nodes will be at different ground potential, a current will flow in common ground wire connection causing voltage drop in this wire.
- RS 423A is improvement over RS 232.
- It uses single ended driver with a differential receiver.
- No common ground path exist.
- Logic high : -4 V to -6 V , Logic low: 4V to 6V
- It allows maximum data rate of 100,000 Bd over a 40 ft line or a maximum baud rate of 1000Bd on a 4000 ft line.

RS 422A



- In this standard each signal will be sent **differentially** over two adjacent wires in a ribbon cable or a twisted pair of wires.
- It uses a differential transmitter and receiver, eliminating the common ground.
- The term differential in this standard means that the signal voltage is developed between the two signal lines rather than between a signal line and ground as in RS 232 and RS 423A.
- The receiver detects the difference between its two inputs as positive or negative.
- **A logic high is transmitted by making 'b' line more positive than 'a' line**
- **A logic low is transmitted by making the 'a' line more positive than the b line.**
- The voltage difference between the two signal lines must be greater than **0.4 V but less than 12 V**.
- **Maximum data rate is for RS 422A lines ranges from 10 million Bd on a line 40 ft long to 100,000 Bd on a 4000 ft line.**
- A further advantage of differential signal transmission is that any electrical noise induced in one signal line will be induced equally in the other signal line. A differential line receiver responds only to the voltage difference between its two inputs, so any noise voltage that is induced equally on the two inputs will not have any effect on the output of differential receiver.

Parameter	RS 232	RS 422A	RS 423A
Line length (max)	50 ft	4000 ft	4000ft
Speed (max)	20 Kbaud/50 ft	10 million baud /40ft 100 Kbaud/4000ft	100 Kbaud /40ft 1Kbaud/4000ft
Mode of operation	Single ended input and output	Differential input and output	Single ended output and differential input
No. of receiver allowed on one line	1	10	10
Logic 1	-3 V to -15 V	b>a	-4 V to -6 V
Logic 0	3 V to 15 V	b<a	4 V to 6 V

Universal serial bus (USB)

- It is a serial I/O interface between peripheral devices and a computer.
- Since it enables easy connection of peripherals to a computer, it has replaced the RS 232 interface and old parallel interface.
- **It allows plug and play**; when a device is connected to USB bus, it is recognized by the host computer and configured, without any effort by user.
- **It supplies power**, on the bus, eliminating need for a separate power supply in peripherals.
- The logic levels used for representing the bits use **voltages less than 5 V**.
- USB implements an **asynchronous serial communication**.
- Initially, it used to be **half duplex but the new USB standard supports full duplex communication**.
- The data is transmitted in the form of packet containing **start, data, parity and stop bits**.
- A USB communication system consists of three basic units
 - Host: it detects the connected device, and manage flow of control information and actual data between the system and the device.
 - Cable : it has four conductors
 - The VBUS conductor is a power supply pin and carries voltage from 4.2 V to 5.25 V
 - The GND connector extends the system ground to the device
 - There are two data pins D+ and D- for differential signaling of the logic levels
 - Device
 - A USB device monitors device address in each communication and prepares itself for communication if selected.
 - It responds to all the request from the host.
 - It adds bits to the packets being set to the host for detecting errors.
 - It detects and corrects errors in the data received from the host.
- USB versions
 - USB 1.0 :
 - allows transfer at 12Mbps
 - known as full speed USB
 - USB 1.1:
 - Supports two modes : full speed mode of 12 Mbps and low speed mode of 1.5 Mbps
 - USB 2.0 :
 - It supports 3 modes : 1.5 ,12 and 480 Mbps
 - USB 3.0 :
 - Known as super speed USB
 - Supports data transfer rates of 4.8 Gbps

Direct Memory Access (DMA)

- DMA is a process of data transfer controlled by an external peripheral.
- In situations in which the microprocessor controlled data transfer is too slow, the DMA is used.
- The Direct Memory Access (DMA) mode of data transfer is the fastest amongst all the modes of data transfer.
- In this mode, the device may transfer data directly to/from memory without any interference from MPU.
- In DMA, the MPU releases the control of buses to a device called DMA controller. The controller manage data transfer between memory and the peripheral under its control, thus bypassing the MPU.
- In 8085 microprocessor uses two pin HOLD and HLDA(Hold Acknowledge) for DMA

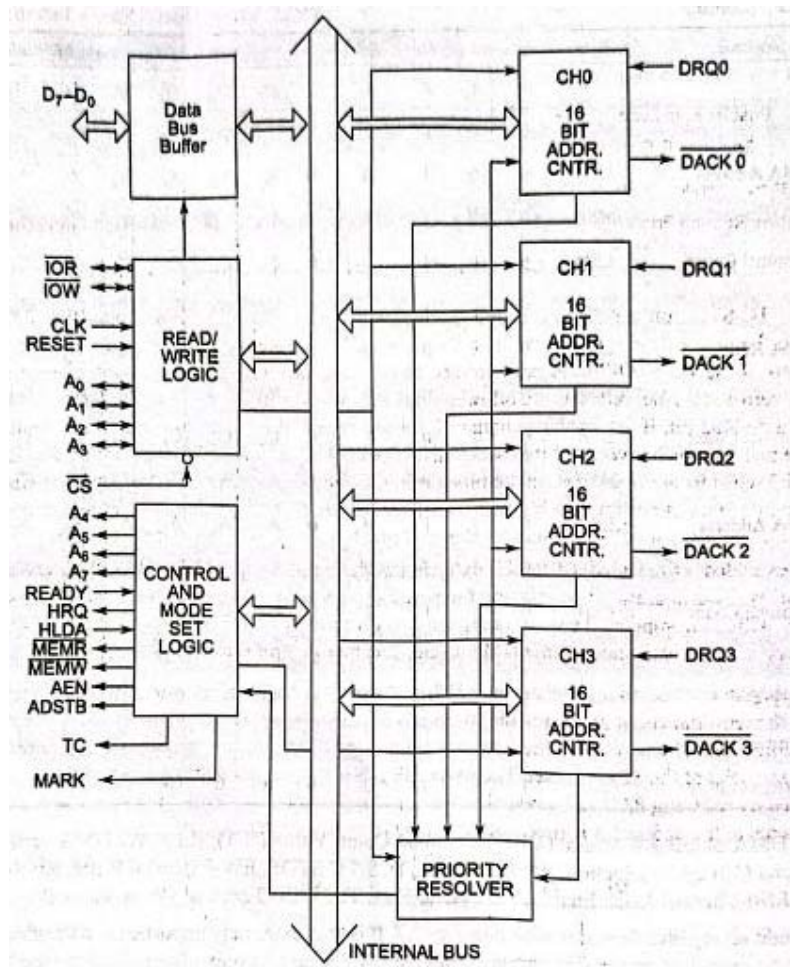
HOLD

- This is an active high input signal to 8085 from the master requesting the use of the address, data and control bus.
- After receiving the HOLD request, the MPU relinquishes the buses in the following machine cycle.
- All buses are tri-stated and the HLDA signal is sent out.
- MPU regains the control of the buses after HOLD goes low.

HLDA (Hold Acknowledge)

- This is an active high output signal indicating that the MPU has relinquished the buses and buses are tri-stated and the requesting unit such as DMA controller can use the address, data and control buses.
- Typically, an external peripheral such as DMA controller sends a high signal to HOLD pin.
- The processor completes the execution of current machine cycle; floats (high impedance state) the address, the data and the control lines; and sends HLDA signal.
- After receiving HLDA, DMA controller takes control of buses and transfer data directly between source and destination, thus by passing the microprocessor.
- At the end of the data transfer, the controller terminates the request by sending a low signal to the HOLD pin and the microprocessor regains control of the buses.

DMA Controller 8257



- Intel's 8257 is a 4 channel DMA controller i.e 4 peripheral devices can independently request for DMA data transfer through these channels at a time.
- The 8257, on behalf of the devices, request the CPU for bus access.

Internal Architecture

- DMA controller has
 - 8 bit internal data buffer
 - Read/write unit
 - Control unit
 - Priority resolving unit
 - 4 DMA channels
- The 8257 performs DMA operation over

- 4 independent DMA channels.
- Each of four channel has a pair of two 16 bit register
 - DMA address register
 - The function of this register is to store the address of the starting memory location, which will be accessed by DMA channel.
 - The starting address of the memory block which will be accessed by the device is first loaded in the DMA address register of the channel.
 - Terminal count register
 - This 16 bit register is used for ascertaining that the data transfer through a DMA channel stops after the required no. of DMA cycles.
 - This register should be written before the actual DMA operation starts.
 - The lower order 14 bits of terminal count register are initialized with the binary equivalent of the number of required DMA cycles minus one.
 - After each DMA cycle, the terminal count register content will be decremented by 1 and finally becomes zero after the required no. of DMA cycles are over.
 - The bits 14 and 15 of this register indicates the type of DMA operation.
 - Bit 15 =0 , Bit 14=1 : DMA write operation
 - Bit 15 =1 , Bit 14=0 : DMA Read operation
- Mode set register and status register are common registers for all channels.
 - Mode set register
 - It is 8 bit register
 - It is used for programming 8257.
 - The function of this register is to enable the DMA channels individually and also set the various modes of operation.
 - Status register
 - It is 8 bit register
- The priority resolver resolves the priority of the four DMA channel depending upon whether normal priority or rotating priority is programmed.

Memory

Memory can be classified in two category: Prime memory and storage memory

Prime memory:

- It is the memory used by microprocessor to execute and store program.
- This memory should be able to respond fast enough to keep up with execution speed of the microprocessor.
- Types of Prime memory
 - Read/Write Memory (W/RM)
 - Read only memory (ROM)

R/WM (Read/Write memory)

- Microprocessor can write into or read from this memory
- It is also known as RAM.
- It is used for information that is likely to be altered, such as writing programs or receiving data.
- It is volatile memory

Types of R/WM

1. Static memory (SRAM)
 - It is made up of flip flops.
 - It stores bits as voltage
 - Each cell requires six transistors. Therefore, the memory chip has low density but high speed
 - This memory is more expensive and consumes more power.
2. Dynamic Memory (DRAM)
 - This memory is made up of MOS transistor gates
 - it stores the bit as charge
 - The advantage of DRAM are it has high density, low power consumption and cheaper than SRAM.
 - The disadvantage is that the charge (bit information) leaks therefore, stored information needs to be read and written again every few milliseconds. This is called refreshing.
 - Refreshing requires extra circuitry adding cost to the system
 - In comparison to processor speed, DRAM is too slow.

ROM (Read only memory)

- It is nonvolatile memory
- This memory is used for programs and data that need not be altered.
- data can be read only
- Once a bit pattern is stored, it is permanent or at least semi-permanent.

Types of ROM

1. Masked ROM: A bit pattern is permanently recorded by the manufactures during production.
2. PROM (Programmable Read Only Memory): This memory can be programmed by the user with a special PROM programmer.
3. EPROM (Erasable Programmable Read only memory) :
 - Information are stored by using EPROM programmer.
 - All the information can be erased by exposing the chip to the **ultraviolet light** and the chip can be re programmed.
 - Disadvantage :
 - i. It must be taken out of the circuit to erase it.
 - ii. The entire chip must be erased
 - iii. Erasing process takes 15 to 20 mins
4. EE-PROM (Electrically Erasable PROM)
 - It is functionally same as EPROM except that information can be altered by using **electrical signal** at the **register level** rather than erasing all the information.
 - It is also possible to **erase entire chip.**
5. Flash Memory
 - This is variation of EE-PROM.
 - Flash memory can be erased either in its **entirety or at the sector (block) level.**

Storage Memory

- This memory is used to store programs and results after the completion of program execution.
- **This is nonvolatile memory.**
- Microprocessor can't directly execute or process programs stored in these memories; programs need to be copied into the R/W prime memory first.

Performance parameter of Memory

Access time (latency):

- For random- access memory, this is the time it takes to perform a read or write operation, that is, the time from the instant that an address is presented to the memory to the instant that data have been stored or made available for use.
- For non-random-access memory, access time is the time it takes to position the read–write mechanism at the desired location.

Memory cycle time:

This concept is primarily applied to random-access memory and consists of the access time plus any additional time required before a second access can commence. This additional time may be required for transients to die out on signal lines or to regenerate data if they are read destructively.

Transfer rate

This is the rate at which data can be transferred into or out of a memory unit. For random-Access memory, it is equal to $1/(\text{cycle time})$.

Self study : Access modes of memory : Random access, sequential access, semirandom access

The Memory Hierarchy

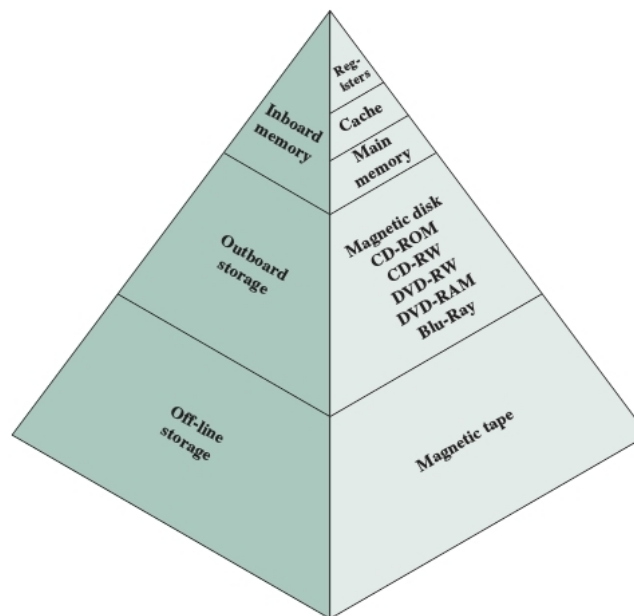
There is a trade-off among the three key characteristics of memory: **capacity, access time, and cost.**

The following relationships hold:

- Faster access time, greater cost per bit
- Greater capacity, smaller cost per bit
- Greater capacity, slower access time

The dilemma facing the designer is clear. The designer would like to use memory technologies that provide for large-capacity memory, both because the capacity is needed and because the cost per bit is low. **However, to meet performance requirements,** the designer needs to use expensive, relatively lower-capacity memories with short access times.

The way out of this dilemma is not to rely on a single memory component or technology, but to employ a **memory hierarchy**. The overall goal of using memory hierarchy is to obtain the highest possible average speed while minimizing the total cost of entire memory system.



As one goes down the hierarchy, the following occur:

- Decreasing cost per bit;
- Increasing capacity;
- Increasing access time;
- Decreasing frequency of access of the memory by the processor.

- Most general computer work more efficiently if they were equipped with additional storage beyond the capacity of main memory.
- There is just not enough space in one memory unit to accommodate all the programs used in a computer.
- Not all the accumulated information is needed by the processor at the same time.
- Therefore, It is economical to use low cost storage device to serve as back up for storing the information that is not currently used by Processor.
- Only the programs and the data currently needed by processor reside in main memory and all other information is stored in **auxiliary memory**.
- Auxiliary memory:
 - Devices that provide backup storage.
 - E.g. Magnetic disk, tape.
 - They are used for storing system programs, large data files and other backup information.
- The memory hierarchy system consists of all storage devices used in computer system from slow but high capacity auxiliary memory to a relatively faster main memory, cache and registers.
- At the bottom of the hierarchy there is relatively slow magnetic tapes used to store removable files.
- The main memory is able to communicate with CPU and with auxiliary memory.
- When programs not residing in main memory are needed by the CPU, they are brought in from auxiliary memory.
- Programs not currently needed in main memory are transferred into auxiliary memory to provide space for currently used programs and data.
- Cache memory:
 - Processor is faster than main memory access time.
 - A technique used to compensate the mismatch in operating speed between Processor and main memory is to employ an extremely fast cache between processor and main memory whose access time is close to processor logic clock cycle time.
 - Cache is used to store program segments currently being executed in the processor and temporary data frequently needed in the present calculation.

Mode of transfer

- Binary information received from an external device is usually stored in memory for later processing.
- Information transferred from processor into an external device originates in the memory unit.
- The CPU merely executes the I/O instructions and may accept the data temporarily, but the ultimate source or destination is the memory unit.
- Data transfer between the processor and I/O devices may be handled using following modes
 - Programmed I/O
 - Interrupt initiated I/O
 - Direct memory access
- Programmed I/O and Interrupt initiated I/O use the processor as an intermediate path whereas DMA transfer data directly to and from memory unit.

Polling (Programmed I/O)

- In this mode during the initial phase, the CPU checks the status of the selected device, issue non data command to the device and prepares it for performing I/O operation.
- In a data transfer operation, the CPU keeps on checking for the device ready condition for data transfer through a program loop.
- Once the device is ready, data flows between device and the CPU.
- This data transfer occurs by executing corresponding instructions by CPU.
- Once one data transfer is complete, the CPU again goes into a loop waiting for the device to be ready for the next data transfer operation.
- In this way, the CPU is tied to the speed of the device till the specified number of bytes of data is transferred between CPU and the device.
- Hence, CPU does not do any other job except waiting for the device to become ready for data transfer.
- Polling is useful in systems that are dedicated to monitor a device continuously.

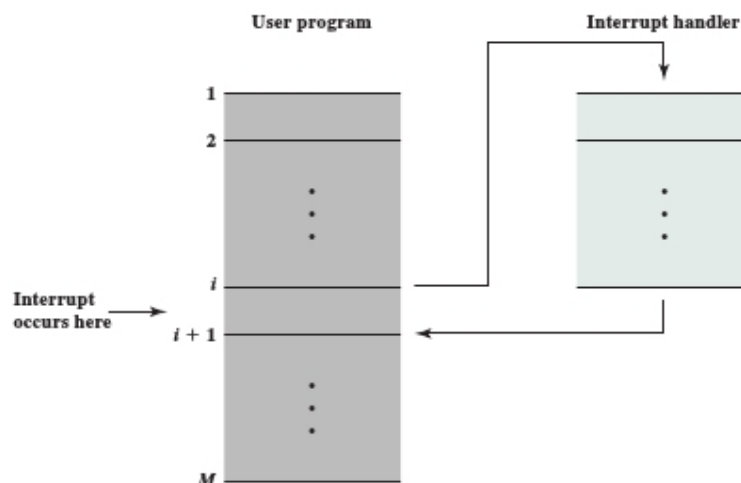
- **Advantage of polling**

- It is a very low cost interface and can meet speeds of most peripherals

- **Disadvantage of polling**
 - During I/O operation, most of the time of the CPU is spent in waiting for the device to become ready and hence it does no useful work.

Interrupt

- Interrupt is a process of data transfer whereby external device or a peripheral can inform the processor that it is ready for communication and requests attention.
- The process is initiated by an external device and is asynchronous i.e. It can be initiated at any time without reference to system clock.
- However, the response to an interrupt request is directed or controlled by microprocessor.
- Interrupts are provided primarily as a way to improve processing efficiency. For example, most external devices are much slower than the processor. Suppose that the processor is transferring data to a printer. After each write operation, the processor must pause and remain idle until the printer catches up. The length of this pause may be on the order of many hundreds or even thousands of instruction cycles that do not involve memory. Clearly, this is a very wasteful use of the processor.
- With interrupts, the processor can be engaged in executing other instructions while an I/O operation is in progress.



Transfer of Control via Interrupts

- When the I/O module sends an *interrupt request* signal to the processor. The processor completes the instruction of the main program it is currently executing and then executes a program written for that interrupt which is called Interrupt service routine

(ISR). After executing ISR, the CPU returns to the next instruction in the main program which it was to execute if interrupt had not occurred.

- **Advantage of interrupt**

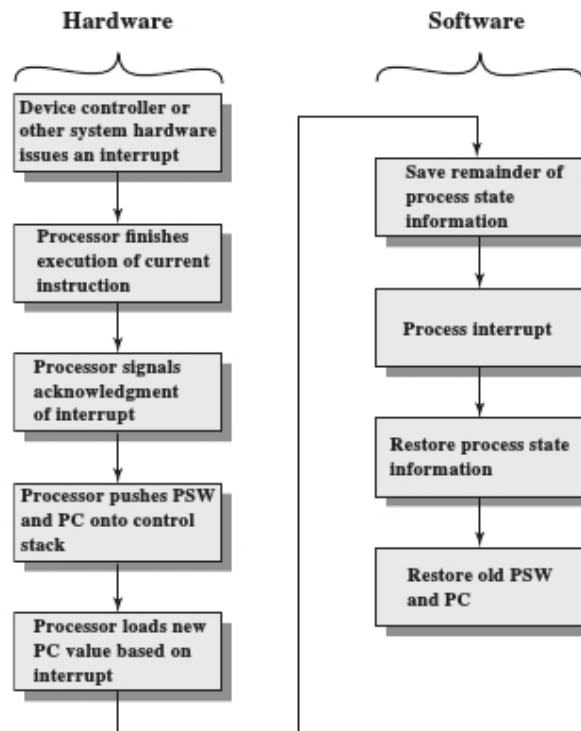
- Data transfer take place only when the device is ready and, therefore, as the CPU is not idle, it can be put to other useful work.

- **Disadvantage of interrupt**

- There is additional cost for interrupt logic
- It may not be suitable for handling very fast devices.

Interrupt Processing Sequence

An interrupt triggers a number of events, both in the processor hardware and in software. The interrupt driven I/O operation takes following steps:



1. The device issues an interrupt signal to the processor.
2. The processor finishes execution of the current instruction before responding to the interrupt.
3. The processor tests for a pending interrupt request, determines that there is one, and sends an acknowledgment signal to the device that issued the interrupt. The acknowledgment **allows the device to remove its interrupt signal**.
4. The processor next needs to prepare to transfer control to the interrupt routine. To begin, it saves information needed to resume the current program at the point of interrupt. The minimum information required is the program status word (PSW) and the location of the next instruction to be executed, which is contained in the program counter (PC). These can be pushed onto a control stack.
5. The processor then loads the program counter with the entry location of the interrupt-handling routine that will respond to this interrupt.
6. At this point, the program counter and PSW relating to the interrupted program have been saved on the control stack. However, in addition, the contents of the processor

registers need to be saved, because these registers may be used by the interrupt handler. Typically, the interrupt handler will begin by saving the contents of all registers on the stack. The stack pointer is updated to point to the new top of stack, and the program counter is updated to point to the beginning of the interrupt service routine.

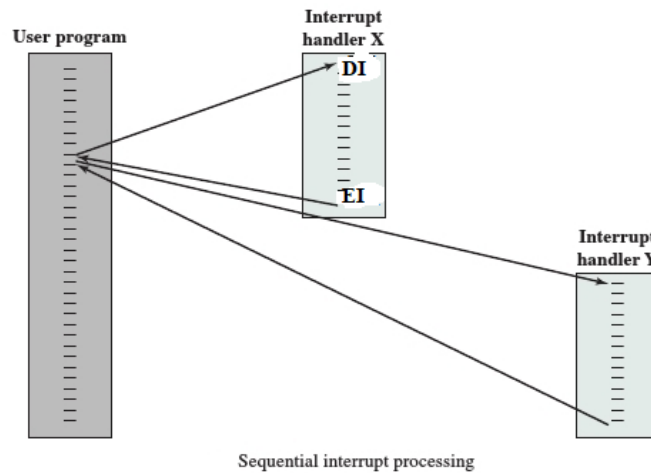
7. The interrupt handler may now proceed to process the interrupt.
8. When interrupt processing is complete, the saved register values are retrieved from the stack and restored to the registers
9. The final act is to restore the PSW and program counter values from the stack. As a result, the next instruction to be executed will be from the previously interrupted program.

Multiple interrupts

- One or more interrupts can occur while an interrupt is being processed.
- Two approaches can be taken to dealing with multiple interrupts.

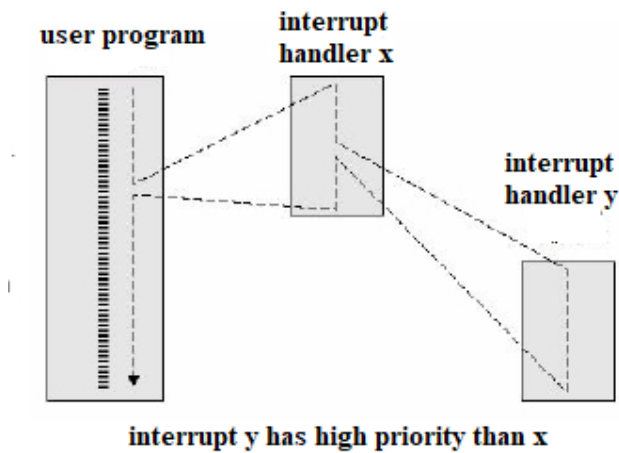
Sequential interrupt processing:

- This method disable interrupts while an interrupt is being processed.
- A **disabled interrupt** simply means that the processor ignores any new interrupt request signal. If an interrupt occurs during this time, it generally remains pending and will be checked by the processor after the processor has re-enabled interrupts.
- If an interrupt occurs when a user program is executing, then interrupts are disabled immediately. After the interrupt-handler routine completes, interrupts are re-enabled before resuming the user program, and the processor checks to see if additional interrupts have occurred.
- This approach is simple, as interrupts are handled in strict sequential order.



Nested interrupt processing (priority wise processing of interrupt):

- The drawback to the preceding approach is that it does not take into account relative priority or time-critical needs.
- A second approach is to define priorities for interrupts and to allow an interrupt of higher priority to cause a lower-priority interrupt handler to be itself interrupted.



Interrupt Service Routine (ISR)

- While the CPU is executing a program, on 'interrupt' breaks the normal sequence of execution of instructions, diverts its execution to some other program called Interrupt Service Routine (ISR).
- ISR, is a special block of code associated with a specific interrupt condition.
- Its central purpose is to process the interrupt and then return control to the main program.
- The service routine must have instructions to perform the following tasks:
 1. Save contents of processor registers
 2. Serve the device
 3. Restore contents of processor registers.
 4. Enable the interrupt.
 5. Return to the running program
- Since the service routine and interrupted program both use same processor registers, it is necessary to save contents of these registers at the beginning of the routine and restore them at the end.
- The occurrence of an interrupt disables the facility from further interrupts. The service routine must turn the interrupt on before the return to the interrupted program. This will enable further interrupts while the computer is executing the interrupted program. The interrupt facility should not be turned on until after the return address is inserted into the program counter.

Interrupt types

1. Hardware interrupt

- When microprocessors receive interrupt signals through pins (hardware) of microprocessor, they are known as Hardware Interrupts.
- An external device initiates the hardware interrupts

2. Software interrupt

- A software interrupt is a particular instructions that can be inserted into the desired location in the program
- They allow the microprocessor to transfer program control from the main program to the subroutine program. After completing the subroutine program, the program control returns back to the main program.

3. Maskable and Non maskable interrupt

- Maskable interrupts can be enabled or disabled by microprocessor by executing instructions.
- Non maskable interrupts cannot be enabled or disabled by microprocessor by executing instructions.
- A non maskable interrupt cannot be ignored by the CPU and must be serviced.

4. Vectored and Non-vectored interrupt

- Vectored Interrupts *are those which have fixed vector address (starting address of ISR)*
- Non vectored interrupts are those in which vector address is not predefined. The interrupting device gives the address of the ISR for these interrupts.

Hardware interrupts in 8085 (Interrupt Pins and Priorities)

- An external device initiates the hardware interrupts and placing an appropriate signal at the interrupt pin of the processor. If the interrupt is accepted then the processor executes an interrupt service routine
- The 8085 has five hardware interrupts: INTR, RST 7.5, RST 6.5, RST 5.5, TRAP

How the 8085 obtains the starting address of the ISR in hardware Interrupts?



Interrupt Type	Trigger	Priority	Maskable	Vector Address
TRAP	Edge and Level	1 st	No	0024H
RST 7.5	Edge	2 nd	Yes	003cH
RST 6.5	Level	3 rd	Yes	0034H
RST 5.5	Level	4 th	Yes	002cH
INTR	Level	5 th	Yes	-

TRAP

- This interrupt is a non-maskable interrupt (NMI) i.e it can't be disabled by any instruction.
- It has highest priority and it need not be enabled and it cannot be disabled.
- It is level- and edge sensitive i.e the input should go high and stay high to be acknowledged.
- **TRAP interrupt is disabled by falling edge of the signal on the pin.**
- When this interrupt is triggered, the program control is transferred to the location 0024H without any external hardware.
- It is generally used for such critical events as power failure and emergency shut off.

RST 7.5

- This interrupt is a maskable interrupt i.e it can be enabled or disabled using instructions.
- It has the second highest priority
- **It can be enabled using EI and SIM instructions.**
- 8085 responds to RST 7.5 interrupt when the signal on RST 7.5 pin has a leading edge.
- When this interrupt is triggered, the program control is transferred to the location 003CH.

RST 6.5 and 5.5

- These interrupts are maskable interrupt i.e it can be enabled or disabled using instructions.
- They can be enabled using EI and SIM instructions.
- The RST 6.5 and RST 5.5 both are level triggered. . ie. Input goes to high and stay high until it is recognized
- The RST 6.5 has the third priority whereas RST 5.5 has the fourth priority.
- When RST 6.5 is triggered, the program control is transferred to the location 0034H.
- When RST 5.5 is triggered, the program control is transferred to the location 002CH.

INTR

- It is maskable interrupt.
- It can be enabled or disabled by using EI and DI instruction.
- It is high level sensitive ie. Input goes to high and it is necessary to maintain high state until it is recognized.
- It has lowest priority
- 8085 respond to INTR interrupt by INTA (interrupt acknowledgement) signal and expects either 1 byte CALL (RST0 through RST7) or a 3 byte CALL.
- It is non vectored interrupt i.e the address of service routine is provided by the external hardware.

The 8085 interrupt process can be described in terms of following steps

1. The interrupt process should be enabled by writing the instruction EI in the main program.
2. When microprocessor is executing a program, it checks the INTR line during execution of each instruction.
3. If the INTR line is high and the interrupt is enabled,
 - a. microprocessor completes current instruction,
 - b. disables the interrupt enable flip flop and
 - c. Sends a \overline{INTA} (interrupt acknowledge) signal.

The processor cannot accept any interrupt request until the interrupt flip flop is enabled again.

4. \overline{INTA} is used to insert a RST instruction (or a CALL instruction) through external hardware.
5. When microprocessor receives an RST instruction (or a CALL instruction) , it saves the memory address of the next instruction on the stack.
6. The ISR is executed from the specified location.
7. The ISR must include the instruction EI to enable interrupt again.
8. At the end of subroutine, the RET instruction retrieves the memory address where the program was interrupted and continue execution.

NOTE: TRAP, RST 7.5, 6.6 and 5.5 does not require \overline{INTA} signal as they are vectored interrupt.

Interrupt instructions

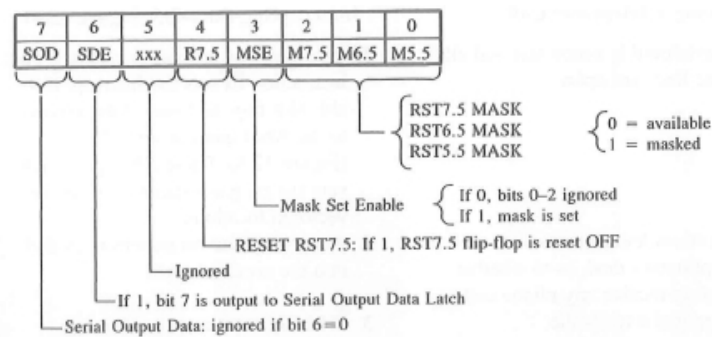
EI (Enable Interrupt)

- It is 1 byte instruction
- It enables all interrupts except TRAP
- The instruction sets the interrupt enable flip flop and enables the interrupt process
- After a system reset or the acknowledgement of an interrupt, the interrupt process is disabled. So this instruction is necessary to re-enable the interrupt.
- No flags are affected

DI (Disable interrupt)

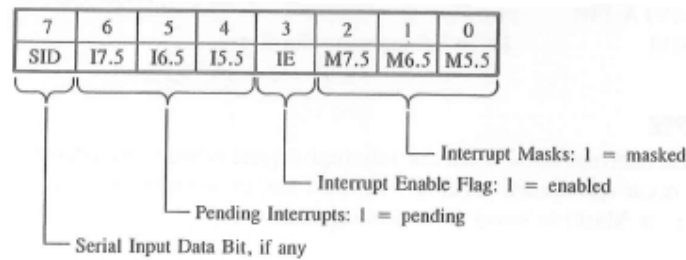
- It is 1 byte instruction
- The instruction resets the interrupt enable flip flop and disables all the interrupt except TRAP.
- No flags are affected

SIM (Set interrupt Mask)



- it is 1 byte instruction
- It has following 3 functions
 1. It is used to set mask for RST 7.5 ,6.5 and 5.5 interrupts
 - SIM reads contents of Accumulator and enables or disables the interrupts.
 - Bit D3 is control bit and should be 1 for bits D0,D1 and D2 to be effective.
 - Logic 0 on D0,D1 and D2 will enable the corresponding interrupts and logic 1 will disable the interrupts.
 2. It reset RST 7.5 flipflop
 - Bit D4 is additional control for RST 7.5
 - D4=1: RST 7.5 is reset
 - It is used to ignore RST 7.5 without serving it
 3. Serial I/O
 - Bits D7 and D6 of accumulator are used for serial I/O and do not affect the interrupts.
 - D6=1: enables the serial I/O
 - When D6=1 , the bit in D7 is output to the SOD pin

RIM (Read interrupt Mask)



- When one interrupt request is being served, other interrupt requests may occur and remain pending.
- RIM instruction is used to sense pending interrupts
- This instruction loads accumulator with 8 bits.
- **The status of pending interrupts can be read from accumulator after executing RIM instruction.**
- It is 1 byte instruction
- It has following functions
 - **To read interrupt mask:** Bits D0,D1 and D2 show the current setting of the mask for each of RST 5.5, RST 6.5 and RST 7.5.
 - **D3 shows the current interrupt enable status. If it is 1, interrupt is enable.**
 - **To identify pending interrupts:** Bits D4, D5 and D6 identify pending interrupts on RST 5.5, RST 6.5 and RST 7.5
 - **To receive serial data:** Bit D7 is loaded with data on the SID input pin.

Software Interrupt in 8085

- The 8085 has 8 software interrupts from **RST 0 to RST 7**.
- These are 1 byte Call instruction that transfer the program execution to a specific location on page 00H.
- **RST (Restart)** instructions are executed in a similar way to that of Call instruction.
 - The address in PC (the address of next instruction to RST instruction) is stored on the stack before program execution is transferred to the RST call location.
 - When processor encounters a **Return** instruction in the subroutine associated with RST instruction, the Program returns to the address that was stored on the stack.
- The vector address of these interrupt can be calculated as follows;
 - $\text{Interrupt no.} \times 8 = \text{vector address}$

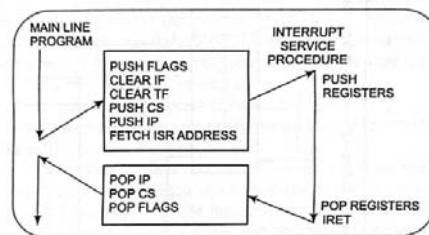
- For RST 5: $5 \times 8 = 40$ (0028H)

Restart instruction	Call location in HEX
RST 0	0000H
RST 1	0008H
RST 2	0010H
RST 3	0018H
RST 4	0020H
RST 5	0028H
RST 6	0030H
RST 7	0038H

8086 Interrupts

- An 8086 interrupt can come from any of following sources
 - **External interrupt of Hardware interrupt:** An external device or a signal interrupts the processor from outside. These interrupts are applied through the NMI and INTR pin of 8086 microprocessor.
 - **Internal interrupts : these interrupts are of two types**
 - **Software interrupt:** interrupt caused by execution of interrupt instruction (INT).
 - **Some error condition produced in 8086 by the execution of an instruction** may generate interrupt. An example of this is the divide by zero interrupt. If you attempt to divide an operand by zero, the 8086 will automatically interrupt the currently executing program.

8086 interrupt process



At the end of each instruction cycle, 8086 checks to see if any interrupts have been requested. If an interrupt has been requested, the 8086 responds to interrupt by following series of major actions;

1. It decrements the Stack pointer by 2 and pushes the flag register on the stack.
2. **It disables the 8086 INTR** interrupt input by clearing the interrupt **flag (IF)** in flag register.
3. It resets the trap flag (TF) in flag register.
4. It decrements stack pointer by 2 and pushes the current code segment register contents on the stack.
5. It decrements stack pointer again by 2 and pushes the current instruction pointer contents on the stack.
6. The control is then transferred to ISR and ISR is executed.

7. At the end of ISR the last instruction should be **IRET**. An IRET instruction at the end of ISR returns execution to the main program by retrieving the contents of IP, CS and flag registers pushed in stack to the respective registers.

Hardware Interrupt

- A hardware interrupt is caused by a device that is external to the processor.
- The two pins that can signal external interrupts are
 - Non maskable interrupt (NMI) pin
 - Interrupt request (INTR) pin

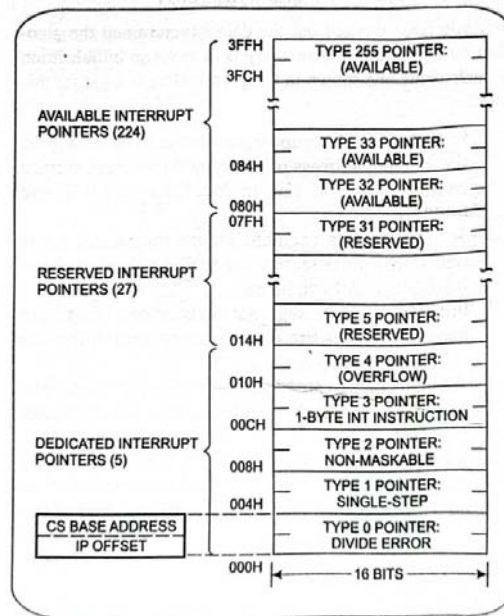
Non maskable interrupt (NMI)

- It has **highest priority** among the hardware interrupts
- It is **not maskable internally by software** i.e it cannot be disabled (masked) by any program instructions.
- It is activated on a **positive edge**.
- 8086 do **a type 2** interrupt response when it receives a low to high transition on its NMI input pin
- When 8086 does a type 2 interrupt , 8086 will
 - Push flags on stack
 - Reset TF and IF
 - Push CS value and IP value
 - 8086 get the **CS** value for the start of the type 2 ISR from address 0000AH and **IP** value for the start of the procedure from address 00008H.

Interrupt request (INTR)

- **It can be masked using Interrupt Flag (IF).**
- If the IF flag is cleared then INTR input is disabled.
- If the IF flag is set INTR input is enabled.
- **IF can be set and cleared using STI and CLI instruction**
- **INTR is of 256 types**
- Its type may be from 00H to FFH.
- **It is level triggered.**
- **For INTR interrupt, the interrupt type is sent to the 8086 from an external hardware device called PIC.**

Interrupt vector table



- When 8086 responds to an interrupt, it goes to 4 memory locations to get the CS and IP values for the start of ISR.
- In 8086, the first 1 KB of memory, from 00000H to 003FFH, is set aside as a table for storing the starting addresses of ISR.
- Since 4 bytes are required to store CS and IP values for each ISP, the table can hold the starting addresses for up to 256 ISR. The 8086 supports a total of 256 types of interrupts i.e from 00H to FFH.
- The starting address of ISR is called **interrupt vector** or interrupt pointer and the table is known as **interrupt vector table**.
- The interrupt vector table contains the IP and CS of all the interrupt types.
- Every external and internal interrupt is assigned with a type (N) that is either implicit (in case of NMI, divide by zero etc.) or specified in the instruction INT N (in case of software interrupt).
- In case of external interrupt through INTR, the type is passed to the processor by an external hardware like PIC.
- For Type N interrupt, the offset address of its IP is calculated as
 - $4 \times N$
 - For Type 32 : offset of its IP is $IP = 4 \times 32 = 128$ (80H) and offset of CS=82H

- The lowest five types are dedicated for interrupts such as divide by zero interrupt, single step interrupt and non-mask able interrupt.
- Type 5 to type 31 are reserved by Intel.
- Type 32 to type 255 are available for you to use for hardware or software interrupts.

Software interrupts

- The 8086 INT instruction can be used to cause the 8086 to do any one of the 256 possible interrupt types.
- The desired interrupt type (N) is specified as part of the instruction.
- Format : INT N
- N can vary from 00H to FFH (0 to 255)
- INT 32 will cause 8086 to do a type 32 interrupt response.
- Execution of software interrupt is same as that of hardware interrupt.
 - Push flag register on stack
 - Reset TF and IF
 - Push CS and IP values of next instruction on stack
 - Get the CS and IP values for the start of ISR from IVT.

INT 00H or Type 0: Divide by zero interrupt

- Type 0 interrupt will generate if the result of a division operation is too large to fit in the destination register.
- The type 0 response is automatic and can't be disabled.
- It gets CS value for the start of ISP from address 00002H in IVT and the IP value for the start of ISP from address 00000H in IVT.

INT 01H or type 1: Single step interrupt

- Used by DEBUG and other debugger to enable single stepping through program execution.
- In single stepping, a system will stop after it executes each instruction and wait for further direction from you.
- If the trap flag is set, the 8086 will automatically do a type 1 interrupt after each instruction executes.
- After execution of each instruction, 8086 automatically jumps to 00004H to fetch 4 bytes for CS: IP of the ISR.

INT 02H or type 2: Nonmaskable interrupt

- 8086 will automatically do a type 2 interrupt response when it receives low to high transition in NMI pin.
- It gets CS value for the start of ISP from address 0000AH in IVT and the IP value for the start of ISP from address 00008H in IVT.

INT 03H or type 3: Breakpoint interrupt

- It is used to implement break point function in a system
- When you insert a break point, the system executes the instructions up to the break point and then goes to the break point procedure.
- A breakpoint ISR usually saves all register contents on stack. Depending upon the system, it may then send the register contents to the CRT display and wait for next command from user or it may just return control to the user.
- It gets CS value for the start of ISP from address 0000EH in IVT and the IP value for the start of ISP from address 0000CH in IVT.

INT 04H or type 4: Overflow Interrupt

- The 8086 overflow flag will be set if the signed result of an arithmetic operation on two signed numbers is too large to be represented in the destination register or memory location.
- There are two ways to detect and respond to an overflow error.
 1. Use jump if overflow instruction (JO), immediately after the arithmetic operation.
 2. Use interrupt on overflow instruction (INTO) after the arithmetic instruction
 - If overflow flag is not set when 8086 executes INTO instruction, the instruction simply function as NOP (No operation).
 - If overflow flag is set then 8086 will do a type 4 interrupt after it executes INTO instruction.
- It gets CS value for the start of ISP from address 00012H in IVT and the IP value for the start of ISP from address 00010H in IVT.

Interrupt priority in 8086

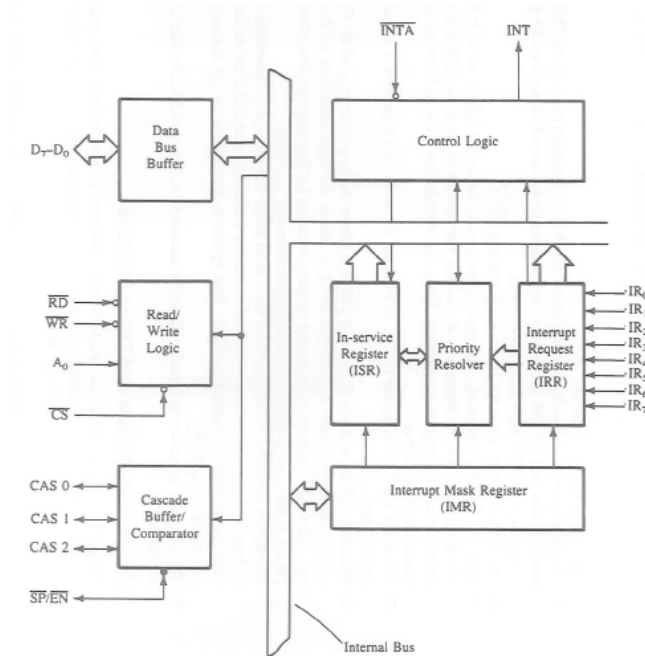
Interrupt	Priority
Divide Error, INT n ,INTO	HIGHEST
NMI	
INTR	
Single step	LOWEST

Programmable interrupt controller (PIC): 8259A

- Consider an application, where a number of I/O devices connected with a CPU desire to transfer data using interrupt driven data transfer mode. In these types of application, more number of interrupt pins are required than available in a typical microprocessor. Moreover, in these multiple interrupt systems, the processor will have to take care of the priorities for the interrupts, simultaneously occurring at the interrupt request pin.
- To overcome all these difficulties, we require a PIC which is able to handle a number of interrupts at a time.
- This controller takes care of a number of simultaneously appearing interrupt request along with their types and priorities.
- For application where we have interrupts from multiple sources, we use an external device called PIC to funnel the interrupts signal into a single interrupt input on the processor.
- **PIC** is designed for use with INTR pin of 8085 and 8086 microprocessor.
- Before using 8259A MPU have to initialize its initialization command word (ICWs) and operation command word(OCWs)
- The 8259A can
 1. Manages 8 interrupt requests according to the instruction written into its control register.
 2. Vector an interrupt request anywhere in the memory map.
 3. Resolve 8 levels of interrupt priorities in a variety of modes.
 4. Mask each interrupt request individually

5. Read the status of pending interrupt , in service interrupts and masked interrupt
 6. be set up to accept either level triggered or edge triggered interrupt request
 7. With additional 8259A devices , the priority scheme can be expanded to 64 levels
- **Operation of 8259A when connected to 8085 microprocessor**
 1. one or more interrupt request line go high requesting the service
 2. 8259A resolves the priorities and sends an INT signal to MPU
 3. The MPU acknowledges the interrupt by sending \overline{INTA}
 4. After the \overline{INTA} has been received, the opcode for CALL instruction (CDH) is placed on data bus.
 5. Because of CALL instruction, the MPU sends two more \overline{INTA} signals.
 6. At the first \overline{INTA} , the 8259A places the low order 8 bit address on data bus and at the second \overline{INTA} , it places the high order 8 bit address of the interrupt vector. This completes the 3 byte CALL instruction
 7. The program sequence of the MPU is transferred to the memory location specified by CALL instruction.

Block diagram of 8259A



Read write logic

- It accepts and decodes commands from CPU
- This block also allows status of 8259A to be transferred on to the data bus.

Data bus buffer

- This bidirectional buffer interfaces internal 8259A bus to the microprocessor system data bus.
- Control word ,status and vector information pass through data buffer during read or write operation

Interrupt Request Register (IRR)

- It has 8 input lines for interrupts.
- If an interrupt input has an interrupt signal on it, then the corresponding bit in the IRR register will be set.

In service register (ISR)

- It keeps track of which interrupt inputs are currently being serviced.
- For each input that is currently being serviced, the corresponding bit will be set in ISR.

Interrupt mask register (IMR)

- It is used to disable (mask) or enable (unmask) individual interrupt inputs.

- Each bit in this register corresponds to interrupt input with the same number.
- You **unmask** an interrupt input by sending a command word **with 0** in the bit position that corresponds to that input.

Priority resolver

- This unit determines the priorities of interrupt requests appearing simultaneously
- It determines if and when an interrupt request on one of the inputs gets serviced.

Cascade buffer /Comparator

- This block is used to expand the number of interrupts levels by cascading two or more 8259As.

Control Logic

- This block has two pins
 - INT as an output :
 - it is connected to interrupt pin of MPU.
 - Whenever a valid interrupt is asserted , this signal goes high
 - \overline{INTA} as an input :
 - it is connected to \overline{INTA} pin of MPU

Multiprocessing Systems

- Multiprocessor systems possess multiple processors and can execute multiple processes simultaneously.
- A multiprocessor system is controlled by one operating system that provides interaction between processors and all the components of the system cooperate in the solution of a problem.
- Multiprocessing improves the reliability of the system because if a fault causes one processor to fail, a second processor can be assigned to perform the functions of disabled processor.
- The benefit derived from a multiprocessor organization is an improved system performance.
- The system derives its high performance from the fact that the computation can proceed in parallel in one of two ways
 - **Multiple independent jobs can be made to operate in parallel**
 - **A single job can be partitioned into multiple parallel tasks:** Multiprocessing can improve performance by decomposing a program into parallel executable tasks. These parallel tasks are executed in different processors.

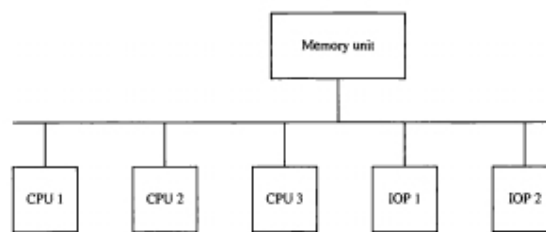
Characteristics of Multiprocessing system

1. There are two or more similar processors of comparable capability.
2. These processors share the same main memory
3. All processors share access to I/O devices, either through the same channels or through different channels that provide paths to the same device.
4. The system is controlled by an integrated operating system

Interconnection structures

- The component that form a multiprocessor system are CPUs, IOPs (input output processor) connected to input-output devices and a memory unit.
- IOP: It is similar to a CPU except that it is designed to handle the details of I/O processing. IOP can fetch and execute its own instructions
- There are several physical forms available for establishing an interconnection network in multiprocessor system. Some of them are;

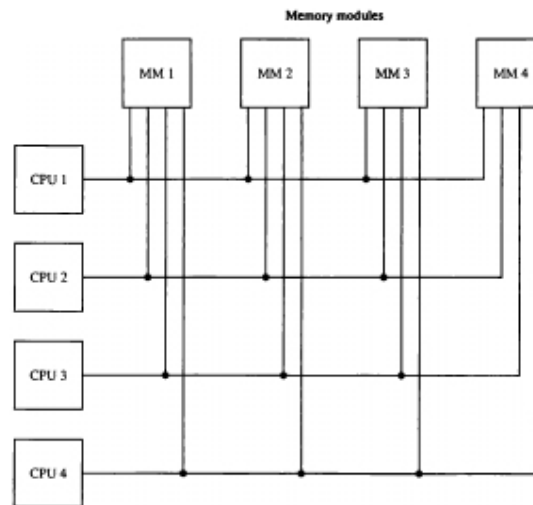
Time shared common bus



- It consists of a number of processors connected through a common path to a memory unit.
- Only one processor can communicate with memory or other processor at any given time.
- Transfer operation are conducted by the processor that is in control of the bus at the time. Any other processor can initiate transfer only when the bus becomes available.
- A time shared common bus must have following features
 - Addressing: It must be possible to distinguish modules on the bus to determine the source and destination of data.
 - Arbitration: A mechanism is provided to arbitrate competing requests for bus control, using some sort of priority scheme.
 - Time sharing: When one module is controlling the bus, other modules are locked out and must, if necessary, suspend operation until bus access is achieved.
- Advantages
 - Simplicity :It is the simplest approach to multiprocessor organization
 - Flexibility: It is generally easy to expand the system by attaching more processors to the bus.

- Reliability : The bus is essentially a passive medium, and the failure of any attached device should not cause failure of the whole system
- Disadvantage :
 - A single common-bus system is restricted to one transfer at a time. This means that when one processor is communicating with the memory, all other processors are either busy with internal operations or must be idle waiting for the bus. As a consequence, the total overall transfer rate within the system is limited by the speed of the single path.

Multiport Memory



- It employs separate buses between each memory module and each CPU.
- Each processor bus is connected to each memory module.
- The memory module have ports and each port accommodates one of the buses.
- The memory module must have an internal control logic to determine which port will have access to memory at any given time.
- Memory access conflicts are resolved by assigning fixed priorities to each memory port.
- Advantage :
 - High transfer rate that can be achieved because of multiple paths between processors and memory.

- Disadvantage :
 - It requires expensive memory control logic and a large no. of cables and connectors.

Real and Pseudo parallelism

- **Pseudo parallelism**
 - **Multiprogramming:** refers to the existence of two or more programs in different parts of the memory at the same time, which are ready for execution.
 - In multiprogramming system, the CPU switches from process to process quickly, running each for tens or hundreds of milliseconds. While, strictly speaking, at any one instant the CPU is running only one process, in the course of 1 second it may work on several of them, giving illusion of parallelism and is termed as pseudo parallelism.
- **Real parallelism**
 - Real parallelism refers to the simultaneous execution of multiple processes by multiple processors in a system. In real parallelism, each processor executes its own process independently, allowing multiple processes to be executed simultaneously.

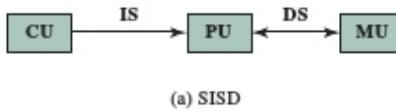
Parallel Processing

- Parallel processing is a term used to denote 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**
- Some methods used for parallel processing
 - While an instruction is being executed in the ALU, the next instruction can be read from memory.
 - The system may have two or more ALUs and be able to execute two or more instructions at the same time.
 - **the system may have two or more processors operating concurrently.**
- The purpose of parallel processing is to speed up the computer processing capability and increase its throughput, that is, the amount of processing that can be accomplished during a given interval of time.

Flynn's Classification

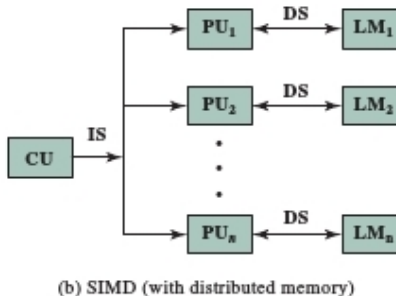
- Flynn's classification is used to categorize systems with parallel processing capability.
- Flynn's considers the organization of a computer system by the number of instructions and data items that are manipulated simultaneously.
- The sequence of instructions read from memory constitutes an **instruction stream**.
- The operations performed on the data in the processor constitutes a **data stream**
- Parallel processing may occur in instruction stream, in the data stream or in both.
- Flynn's classification divides computers into 4 groups as
 - Single Instruction , Single Data (SISD) stream
 - Single Instruction, Multiple Data (SIMD) stream
 - Multiple Instruction, Single Data (MISD) stream
 - Multiple Instruction, Multiple Data (MIMD) stream
- Control unit (CU): provides an instruction stream (IS) to a processing unit (PU).
- Processing unit (PU): operates on a data stream (DS) from a memory unit (MU).

Single Instruction, Single Data (SISD) stream



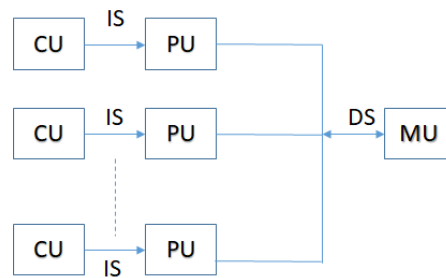
- A single processor executes a single instruction stream to operate on data stored in a single memory.
- Instructions are executed sequentially and the system may or may not have internal parallel processing capabilities.
- Parallel processing in this case may be achieved by means of multiple functional units or by pipelining.
- It has a control unit, a processing unit and a memory unit.
- Uniprocessors fall into this category.

Single Instruction, Multiple Data (SIMD) stream



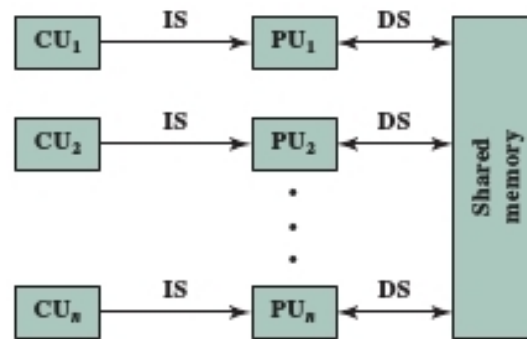
- This organization includes many processing units under the supervision of a common control unit.
- All processors receive the same instruction from the control unit but operate on different data items.
- Each processing element has an associated data memory, so that each instruction is executed on a different set of data by different processors.

Multiple Instruction, Single Data (MISD) stream



- A sequence of data is transmitted to a set of processors, each of which executes a different instruction sequence.
- This structure is not commercially implemented.

Multiple instruction, multiple data (MIMD) stream



(c) MIMD (with shared memory)

- A set of processors simultaneously execute different instruction sequences on different data sets.
- It is capable of processing several programs at the same time.
- Most multiprocessor systems can be classified in this category

Instruction level parallelism

- **Instruction-level parallelism** refers to the degree to which, on average, the instructions of a program can be executed in parallel.
- A combination of compiler-based optimization and hardware techniques can be used to maximize instruction-level parallelism
- **Instruction-level parallelism** exists when instructions in a sequence are independent and thus can be executed in parallel by overlapping.
 - MOV B,C
ADI 01H
MOV H,L
These 3 instructions are independent and all three could be executed in parallel.
 - ADI 01H
ADI 02H
MOV B,A
These instructions can't be executed in parallel because the second instruction uses result of first, and the third uses the result of second.
- The degree of ILP is determined by
 - Frequency of true data dependencies (if an instruction needs data generated by previous instruction)
 - Frequency of procedural dependencies in a code : The instructions following a branch (taken or not taken) have a **procedural dependency** on the branch and cannot be executed until the branch is executed

Process level parallelism

- A process is a program in execution.
- A process requires various system resources like CPU for executing process, memory for storing the code corresponding to the process and associated variables, I/O devices for information exchange etc.
- A process is sequential in execution.
- Process-level parallelism refers to the ability to execute multiple processes simultaneously on a computer system with multiple CPUs.
- Multiprocessor systems possess multiple processors and can execute multiple processes simultaneously.

Thread level parallelism

- A single process can be broken up into multiple, concurrent threads that execute cooperatively to perform work of the process.
- A thread is a single sequential flow of control within a process.
- Thread-level parallelism (TLP) is a technique used in computer architecture to increase the performance of processors by executing multiple threads simultaneously on a computer system with multiple processors.
- Each thread is a separate path of execution within a process, and by running them concurrently, the system can achieve a higher level of performance and efficiency.

Inter-process communication

- Processes that are cooperating to get some job done often need to communicate with one another and synchronize their activities.
- Inter process communication (IPC) is a mechanism which allows processes to communicate each other.

- **Some of the important Methods for IPC are**

- **Shared memory**

- Processes share some area of memory to communicate among them.
 - Information to be communicated by process is written to the shared memory area.
 - Other processes which require this information can read the same from the shared memory area.

- **Messages passing**

- The major difference between shared memory and message passing is that. Through shared memory lots of data can be shared whereas only limited amount of data is passed through message passing.
 - The actual function of message passing is normally provided in the form of a pair of primitives:

send (destination, message)
receive (source, message)

- This is the minimum set of operations needed for processes to engage in message passing. A process sends information in the form of a *message*

to another process designated by a *destination*. A process receives information by executing the receive primitive, indicating the *source* and the message.

○ Remote Procedure Call (RPC)

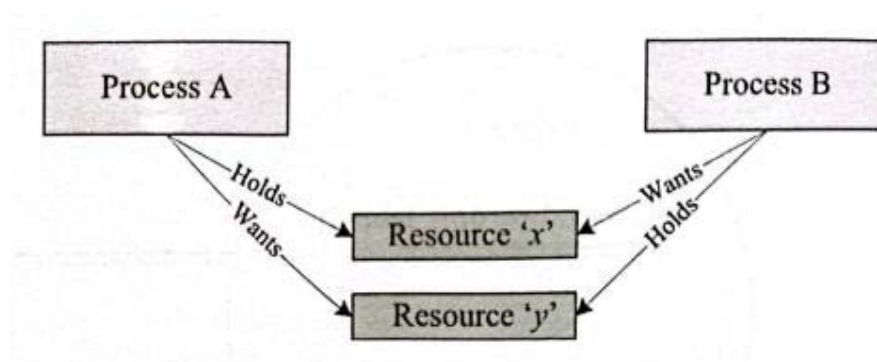
- It is used by a process to call a procedure of another process running on the same processor or on a different processor.

Resource Allocation

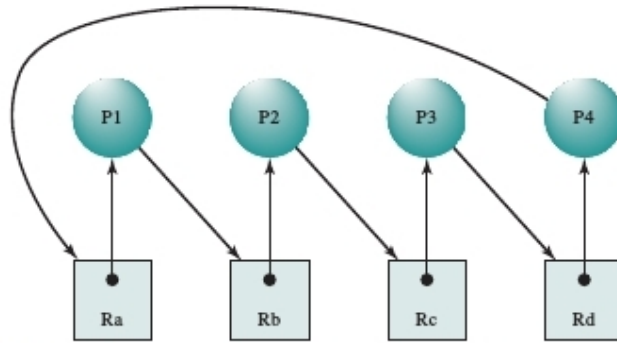
- resource allocation is necessary for any application to be run on the system
- When the user opens any program this will be counted as a process, and therefore requires the computer to allocate certain resources for it to be able to run.
- Such resources could be access to a section of the computer's memory, data in a device interface buffer, one or more files, or the required amount of processing power.
- On a computer with multiple processors different processes can be allocated to different processors so that the computer can truly multitask.
- OS manages the resources for each processes depending upon different algorithm.

Deadlock

- A set of processes is deadlocked if each process in the set is waiting for an event (typically the freeing up of some requested resource) that only another process in the set can cause.
- Because all the processes are waiting, none of them will ever cause any event that could wake up any of the other members of the set, and all the processes continue to wait forever.
- Dead lock is permanent because none of the events is ever triggered.
- A deadlock condition creates a situation where none of the processes are able to make any progress in their execution.



- Process A holds a resource x and it wants a resource y held by process B. Process B is currently holding resource y and it wants the resource x which is currently held by process A. Both hold the respective resources and they compete each other to get the resource held by the respective processes. The result of competition is deadlock. None of the process will be able to access the resources held by other processes since they are locked by the respective processes.
- Conditions for deadlock : four conditions must hold for there to be a deadlock
 - **Mutual exclusion:** Only one process may use a resource at a time. No process may access a resource unit that has been allocated to another process
 - **Hold and wait:** A process may hold allocated resources while awaiting assignment of other resources held by other process.
 - **No preemption:** No resources can be forcibly removed from a process holding it.
 - **Circular wait:** A closed chain of processes exists, such that each process holds at least one resource needed by the next process in the chain.



- All four of these conditions must be present for a resource deadlock to occur. If one of them is absent, no resource deadlock is possible.
- Approaches for dealing with deadlock
 - **Prevent deadlock:** prevent deadlock condition by negating one of the 4 conditions favoring the deadlock situation.
 - **Avoid deadlock :**
 - Avoiding deadlock by making the appropriate dynamics choices based on the current state of resource allocation.
 - With dead lock avoidance, a decision is made dynamically whether the current resource allocation request will, if granted, potentially lead to a deadlock. **Deadlock avoidance thus requires knowledge of future process resource requests.**
 - Two approaches for deadlock avoidance
 1. Do not start a process if its demands might lead to deadlock
 2. Do not grant an incremental resource request to a process if this allocation might lead to deadlock.
 - **Detect and recover :**
 - This approach suggest the detection of a deadlock situation and recovery from it.
 - Deadlock detection involves periodically checking the system for deadlocks
 - Once the deadlock has been detected following methods can be used for recovery
 - a. Abort all the deadlock processes
 - b. Successively abort deadlock process until deadlock no longer exit. The order in which processes are selected for abortion

should be based on some criteria and after each abortion existence of deadlock is checked.

- c. Successively preempt resources until deadlock no longer exist.

RISC (Reduced instruction set computer) Architecture

The concept of RISC architecture involves an attempt to reduce execution time by simplifying the instruction set of the computer.

- The number of instructions is minimized (less than 100)
- The no. of addressing mode is relatively few.(less than 3)
- The no. of memory reference instruction is minimized. Memory is accessed only by Load and Store instruction.
- A relatively large number of registers in the processor unit
- All operations done within the registers of the CPU
- The no. of registers ranges from 32 to more than 100 registers.
- The instruction format is simplified and each instruction is executed in one clock cycle.
This is done by overlapping the fetch, decode, and execute phases of two or three instructions by using a procedure referred to as pipelining
- Hardwired rather than microprogrammed control
- Fixed-length, easily decoded instruction format.

CISC(Complex instruction set Computer) Architecture

- A computer with a large number of instructions is classified as a complex instruction set computer
- The essential goal of a CISC architecture is to attempt to provide a single machine instruction for each statement that is written in a high-level language.
- It uses variable length instruction format.
- The instructions in a typical CISC processor provide direct manipulation of operands residing in memory.
- As more instructions and addressing modes are incorporated into a computer, the more hardware logic is needed to implement and support them, and this may cause the computations to slow down.
- A large number of instructions-typically from 100 to 250 instructions
- Some instructions that perform specialized tasks and are used infrequently
- A large variety of addressing modes-typically from 5 to 20 different modes
- They use microprogramming in order to implement complex instructions.
- It has small number of internal processor register
- Instructions require multiple no. of clock cycles for their execution.

RISC	CISC
Less number of instructions	Greater number of instructions
Instruction pipelining and increased execution speed	Generally no instruction pipelining feature
Allows each instruction to operate on any register.	All instructions are not allowed to operate on any register
Operations are performed on registers only, the only memory operations are load and store.	Operations are performed on registers or memory depending on the instruction.
Programmers need to write more code to execute task since the instructions are simpler	A programmer can achieve the desired functionality with a single instruction which in turn provides the effect of using more simpler single instructions in RISC
Control unit is simple	Control unit is complex
Fixed length instructions	Variable length instructions
Execute one instruction per clock cycle.	CISC processor has complex instructions that require multiple numbers of clock cycles for execution
It has a hard-wired control unit.	It has a microprogramming control unit.
A large no. of registers are available	Limited no. of general purpose registers
Decoding of instructions is simple	Decoding of instructions is complex
Can be implemented using Harvard Architecture	Can be Harvard or Von-Neumann Architecture.

Operating System

- A modern computer consists of one or more processors, some main memory, disks, printers, a keyboard, a mouse, a display, network interfaces, and various other input/output devices. All in all, a complex system. If every application programmer had to understand how all these things work in detail, no code would ever get written. Furthermore, managing all these components and using them optimally is an exceedingly challenging job. For this reason, computers are equipped with a layer of software called the **operating system**.
- An OS is a program that controls the execution of application programs and acts as an interface between applications and the computer hardware.
- OS can be thought of as having two objectives:
 - **Convenience:** An OS makes a computer more convenient to use.
 - **Efficiency:** An OS allows the computer system resources to be used in an efficient Manner.

Features of OS

Process Management

- Process management deals with managing the processes.
- It includes
 - setting up the memory space for process,
 - loading the process's code into the memory space,
 - allocating system resources,
 - Scheduling and managing the execution of process
 - Setting up and managing inter process communication and synchronization, process termination etc.

File System Management

- **File is a collection of related information**
- File could be a program (Source code or executable), text files, image files etc.
- Each of these files differ in the kind of information they hold and the way in which information is stored.
- OS is responsible for
 - Creating , deletion and alteration of files
 - Creating , deletion and alteration of directories

- Saving of files in secondary storage memory (Hard disk)
- Providing automatic allocation of file space based on the amount of free space available.
- Providing a flexible naming convention for the files

I/O system Device Management

- OS is responsible for routing the I/O requests coming from different user applications to the appropriate I/O devices of the system.
- In a well-structured OS, the direct accessing of I/O devices are not allowed and access to them are provided through a set of application programming interfaces (API) exposed by the OS.
- The OS maintains a list of all the I/O devices of the system.
- This list may be available in advance, at the time of building the OS.
- Some OS, dynamically updates the list of available devices as and when a new device is installed.
- The service 'Device manager' is responsible for handling all I/O device related operation.
 - Loading and unloading of device drivers
 - Exchanging information and the system specific control signals to and from the device.

Primary Memory Management

- In a **uniprogramming** system, main memory is divided into two parts: one part for the operating system and one part for the program currently being executed.
- In a **multiprogramming system**, the "user" part of memory must be further subdivided to accommodate multiple processes. The task of subdivision is carried out dynamically by the operating system and is known as **memory management**.
- The OS must prevent independent processes from interfering with each other's memory, both data and instructions.
- Memory management is responsible for
 - keeping track of which part of memory area is currently used by which process
 - Allocating and de-allocating memory space to process on a need basis.

Secondary Storage management

- It deals with managing the secondary storage memory devices connected to the system
- Secondary storage is used as backup medium for programs and data since main memory is volatile.
- In most system secondary storage is kept in disks(Hard Disk)
- Secondary storage management deals with
 - Disk storage allocation
 - Disk scheduling (time interval at which the disk is activated to backup data)
 - Free disk space management.

Resource Management

- Key responsibility of the OS is to manage the various resources available to it (main memory space, I/O devices, and processors) and to schedule their use by the various active processes.
- The main goal of resource management is to maximize the utilization of system resources, prevent resource starvation, and ensure that the system runs smoothly and efficiently.

Protection system

- Most of the OS are designed in such a way to support multiple user with different level of access permission.
- Protection deals with implementing security policies to restrict the access to both user and system resources by different applications or processes or users.
- In multiuser supported OS, one user may not be allowed to view or modify the whole/portions of another user's data or profile details.
- Some applications may not be granted with the permission to make use of some of the system resources.

Error detection and response

- A variety of errors can occur while a computer system is running. These include internal and external hardware errors, such as a memory error, or a device failure or malfunction; and various software errors, such as arithmetic overflow, attempt to access forbidden memory location, and inability of the OS to grant the request of an application. In each case, the OS must make the response that clears the error condition with the least impact on running applications. The response may range from ending the

program that caused the error, to retrying the operation, to simply reporting the error to the application.

Accounting

- A good OS collects usage statistics for various resources and monitors performance parameters such as response time. On any system, this information is useful in anticipating the need for future enhancements and in tuning the system to improve performance.

Digital Signal Processor (DSP)

Digital signal processing system is a specialized system that inputs an analog signal, converts it to digital form, performs specialized arithmetic operations on the data, and then converts the data back to the analog form via a digital to analog converter.

The key element in this system is a new type of microprocessor called the **Digital Signal Processor (DSP)**.

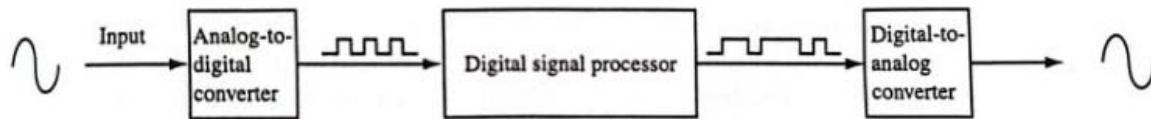


Figure 1.8 In a digital signal processing system, analog input signals are converted to digital form, processed by the DSP, and then converted back to analog form.

- A digital signal processor (DSP) is a specialized microprocessor with its architecture optimized for the operational needs of digital signal processing.
- Digital signal processors are 2 to 3 times faster than the general purpose microprocessors in signal processing applications.
- Digital signal processor implement algorithms in hardware which speeds up the execution.
- The architecture of a DSP is made to differ from that of a conventional microprocessor in several ways;
 1. The data and program instructions are stored in two different memory areas, each with its own bus. This is called Harvard Architecture.
 2. Hardware multiplier and adders are built into the processor and are optimized to perform a calculation in a single clock cycle.
 3. Arithmetic pipelining is used so that several instructions can be operated on at once. For example, two numbers may be multiplied in one part of processor while two other numbers are being added in another part.
 4. Hardware DO loops are provided to speed up repetitive operations.
 5. Multiple (serial) I/O ports are provided for communication with other processor

