CHAPTER 1: Introduction to Microprocessors: (4hrs)

Microprocessor:

A microprocessor is a multipurpose, programmable, clock-driven, register-based electronic devices that reads binary instructions from a storage device called memory, accepts binary data as input and processes data according to those instructions, and provides results as output.

- Input – digital input from IO or memory
- Process – process the instruction
- Output – digital output to IO or memory

In general microprocessor performs 3-basic task:

1. Data transfer between itself and memory or IO unit.
2. Arithmetical or Logical operations on data, and
3. Switching and decision making.

Operation on above task are based on the instruction provided to the microprocessor. So, a critical step in performing any task is retrieval of instruction, understanding the command and executing it. This can be listed as:

1. Fetch instruction from memory
2. Decode instruction
3. Execute the command from the instruction.

Microprocessor Based system:

The microprocessor is a clock-driven semiconductor device consisting of electronic logic circuits manufactured by using either a large-scale integration (LSI) or very-Large-scale integration (VLSI) technique. The microprocessor is capable of performing various computing functions and making decisions to change the sequence of program execution.

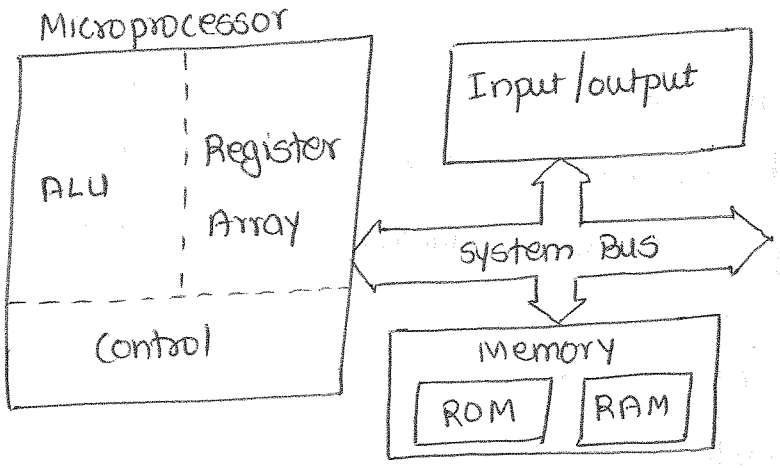


Fig: Microprocessor based system with Bus architecture

Arithmetic/Logic unit:

The ALU unit performs such arithmetic operations as addition and subtraction, and such logic operations as AND, OR and exclusive OR.

Register Array:

Registers are primarily used to store data temporarily during the execution of a program and are accessible to the user through instructions.

Control unit:

The control unit provides the necessary timing and control signals to all the operations in the microcomputer. It controls the flow of data between the microprocessor and memory and peripherals.

Memory:

Memory stores such binary information as instructions and data, and provides that information to the microprocessor whenever necessary. The memory block has two sections:

ROM (Read-only memory): is used to store programs that do not need alterations.

RAM (Random-Access memory) (Read/write memory): is also known as

user memory and used to store user programs and data.

I/O (Input/output):

I/O (Input/output) is used to communicate with the outside world. These I/O devices are also known as peripherals.

System Bus:

The system bus is a communication path between the microprocessor and peripherals; it is nothing but a group of wires to carry bits.

Microcontroller:

A microcontroller is essentially an entire computer on a single chip.

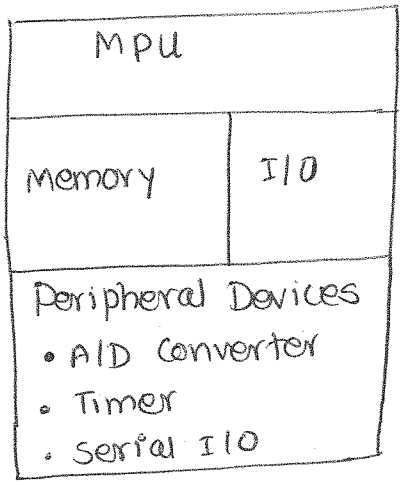


fig: Block diagram of a microcontroller:

Advantages of microprocessor:

* Small in size / Less space consumption.
* High speed
* Multiple operations can be performed.
* Low cost
* Efficiency (power efficiency, computational efficiency)
* Greater Reliability.
* Programmability

Disadvantages:

Uses of micro-processor is disadvantageous typically in areas where the device is simple and low-end. In such types, they may become costly and complex, with most of the resources unused. In such systems simple logic gate implementation can be more efficient.

Applications:

i) Programmable system
ii) Embedded system

i) Programmable system:

These are programmable in nature i.e. both hardware and program can be changed according to task and user requirement. These are capable of handling large number of data memory and peripheral devices. They are used in computing and data processing.

Eg: Microcomputer.

ii) Embedded system:

These are designed to perform specific task such as printer, washing machine, traffic light controller etc. and microprocessor used in them provide programbility for such task.

Microcomputer Architecture (3 bus Architecture):

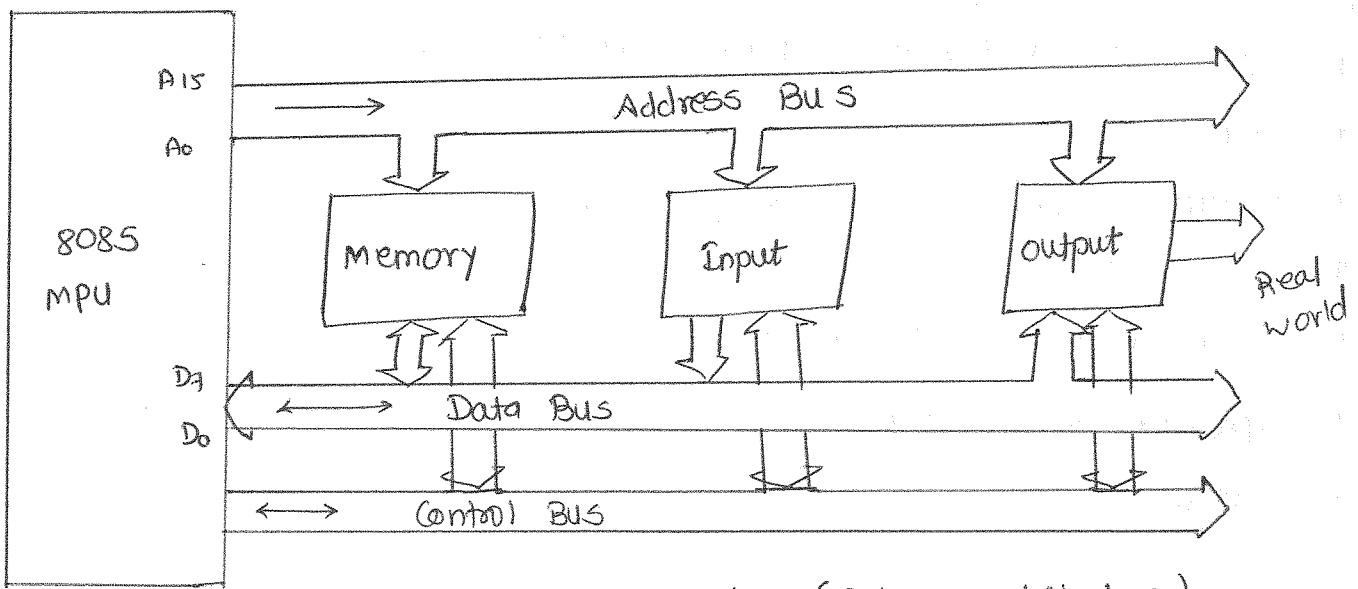


Fig: Microcomputer architecture (3 bus architecture)

1) Address Bus:

It is a unidirectional Bus used to carry address location of memory unit where data is written to or read from. It also carries the location or port address of the I/O devices. The transfer of address is from microprocessor to peripheral devices only.

2) Data Bus:

It is a bidirectional Bus that carries data from microprocesser to peripheral devices and viceversa. It is a parallel Bus that carries multiple bits of data in a single instance.

3) Control Bus:

It is a bidirectional Bus that is used to carry control signal between microprocessor and peripheral devices. In control Bus, rather than using whole bus to carry control signals individual wires are used to carry each control signal.

Evolution of microprocessor:

First generation:

→ Built upon PMOS that was low cost, low speed and not comparable with TTL (Transistor - Transistor Logic)

→ Used in calculator, home appliances, low and special purpose application.

S.N.	Date	model	features
1.	1971	4004	• 4 bit address / data multiplexed • 740 KHz clock rate • PMOS • Addressable memory upto 640 byte and program memory 4KB. • Used in first electronic calculator • can only add or subtract

2.	1972	8008	• 8 bit address/data multiplexed • 500 KHz clock rate • PMOS • Addressable memory 16 KB.

Second generation:

→ Built upon NMOS that was fast, high packing density and better reliability.
→ has large chip size
→ More addressable memory and IO port.
→ Fast and powerful instruction set
→ Better interrupt handling
→ used in industrial control, process control, military application.

3.	1973	8080	• Bus width 8 bit data, 16 bit address • clock rate 2 MHz • NMOS • Addressable memory 64 KB • Used in Altair 8800, Traffic light controller, cruise missile. • Other manufactures - Motorola MC 6800, Zilog Z8, Fairchild F8
4.	1976	8085	• Bus width 8 bits data, 16 bits address • clock rate 3 MHz • Nmos / CMOS • Used in computer peripheral controller's modems, harddisks, printers etc. • High level of integration, operating for the first time on a single 5V power supply from 12V previously. • 2^8 I/O devices and 2^{16} memory locations.

Third generation:

→ Used HMOS (High density MOS) that provided high packing density and high speed.

→ 40 | 48 | 64 pins
→ High speed
→ High processing
→ registors are of 8 | 16 | 32 bits
→ able to execute multiply and divide operations
→ Memory space 1 to 16 GB.
→ Used in Business and data acquision system, real time control.

5.	1978	8086	• Bus width 16 bits data, 20 bits address • clock rates: 4.77, 8, 10 MHz. • Addressable memory 1 MB • The memory is divided into odd and even banks. It accesses both the banks simultaneously in order to read 16 bits of data in one clock cycle. • Used in IBM-PC
6.	1978	8088	• Internal architecture 16 bits • External bus width: 8 bits data, 20 bits address • Addressable memory 1 MB. • clock rates 4.77, 8 MHz. • Used in IBM, PCs and PC clones.
7.	1982	80186	• Clockrate 6 MHz • Included two timers, a DMA controller, and an interrupt controller on chip. • Used mostly in embedded applications controllers, point-of-cell systems, etc.
8.	1982	80286	• Bus width: 16 bit data, 24 bit address • clock rate upto 25 MHz • Addressable memory 16 MB • Included memory protection hardware to support multitasking operating systems with pre-process address space.

Fourth generation:

→ 32 bit architecture

→ used low power (HMOS)

→ used in multiuser and multiapplication enviroment.

9.	1985	80386	• Bus width : 32 bits data, 32 bits address • clock rate upto 33 MHz. • Reworked and expanded memory protection support including paged virtual memory and virtual 86 mode. • used in desktop computing.
10.	1989	80486	• Bus width : 32 bits • clock rate upto 50 MHz • Addressable memory 4 GB • Level 1 cache of 8 kB on chip • Math co-processor on chip • used in desktop computing and servers.

Fifth generation:

→ 32/64 bit architecture

→ had varieties on speed, power consumption, size etc.

→ Commonly used in personal computers, smart phones, servers, etc.

11.	1993	pentium I	• Bus width 32 bits
12.	1997	pentium II	• Bus width 32 bits
13.	1999	pentium III	• Bus width 32 bits
14.	2000	pentium IV	• Bus width 32 bits
15.	2001	Itanium	• Bus width 64 bits • clock rate upto 1.66 GHz.
16.	2006 onwards	Core processor eg: Core2 duo, solo, quad, i3, i5, i7	• Bus width 32/64 bit • single core, dual core, quad core and hex core processor families.

The von- Neumann Architecture:

The programming process could be facilitated if the program could be represented in a form suitable for storing in memory alongside the data. Then a computer could get its instruction by reading them from the memory and a program could be set or altered by setting the values of a portion of memory. This approach known as 'stored- program concept' was first adopted by John von- Neumman's and hence the architecture of computer he proposed is name as von- Neumman's architecture.

The main memory is used to store both data and instructions. The arithmetic logic unit is capable of performing arithmetic and logical operations on binary data. The program control unit, interprets the instructions in memory and causes them to be executed. The I/O unit gets operated from the control unit.

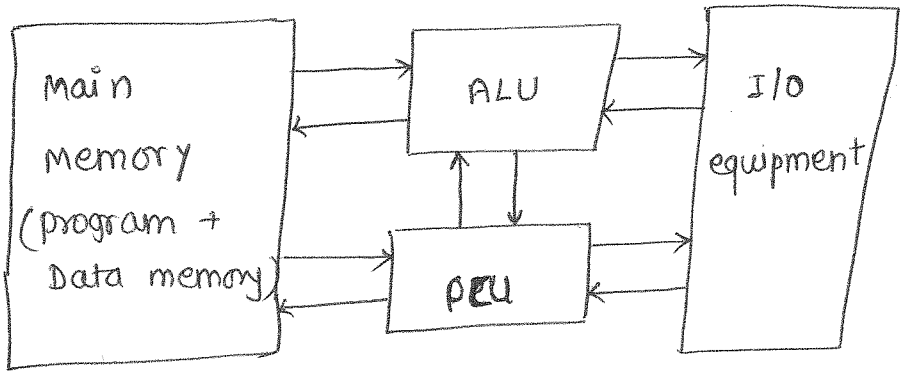


Fig: Von- Neumann Architecture

The memory of the Neumann's architecture consists of 1000 storage locations, called words of 40 binary digits. Both data and instructions are stored in it.

The control unit operates the computer by fetching instructions from memory and executing them one at a time. The storage locations of the control unit and ALU are called Registers. The various register of this models are:

MBR - Memory Buffer Register - It consists of a word to be stored in memory or is used to receive a word from memory.

MAR - Memory Address Register - It contains the address in memory of the word to be written from or read into the MBR.

IR - Instruction Register - Contains the 8-bit op code instruction being executed.

IBR - Instruction Buffer Register - It is used to temporarily hold the instruction from a word in memory.

PC - program counter - It contains the address of the next instruction to be fetched from memory.

Advantages:

1. Computer can handle instructions as easily as data
2. Ease of loading program into memory
3. Efficient use of memory.
4. Cost effective due to same program and data memory.

Disadvantages:

1. Required special hardware protection mechanism to protect instruction and data being overlapped by each other.
2. Low speed because concurrent fetching of data and instruction was not possible.

Harvard Architecture:

The Harvard architecture based computer consists of separate memory spaces for the programs (instructions) and data. Each memory space has its own address and data buses. As a result of this both instruction and data can be fetched from memory concurrently. This feature provides a significant processing

speed improvement when compared with microprocessor devices based on Von-Neumann type architecture.

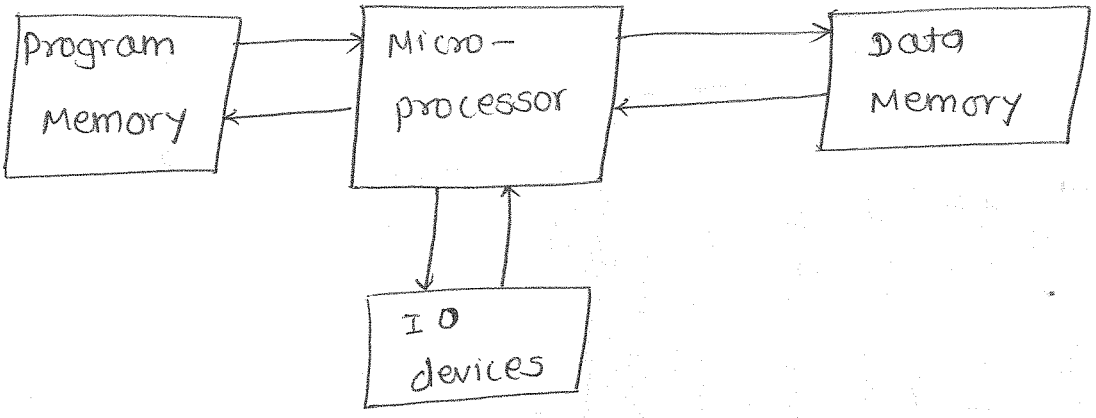


Fig: Harvard architecture

Advantages:

→ Concurrent fetching of data and instruction was possible so it provide higher speed.

→ No overwriting of program and data

Disadvantages:

→ Methods or mechanism of storing program into program memory and data into data memory had to be developed.

→ Higher cost due to separate program and data memory.

→ No optimum use of memory.

Internal architecture of 8 bit Microprocessor 8085

Features of 8085:

Bus width: 8 bit data bus, 16 bit address

Maximum I/O devices: 2^8 and maximum address memory location: 2^{16}

Clock rate: 3 MHz

40 pin IC.

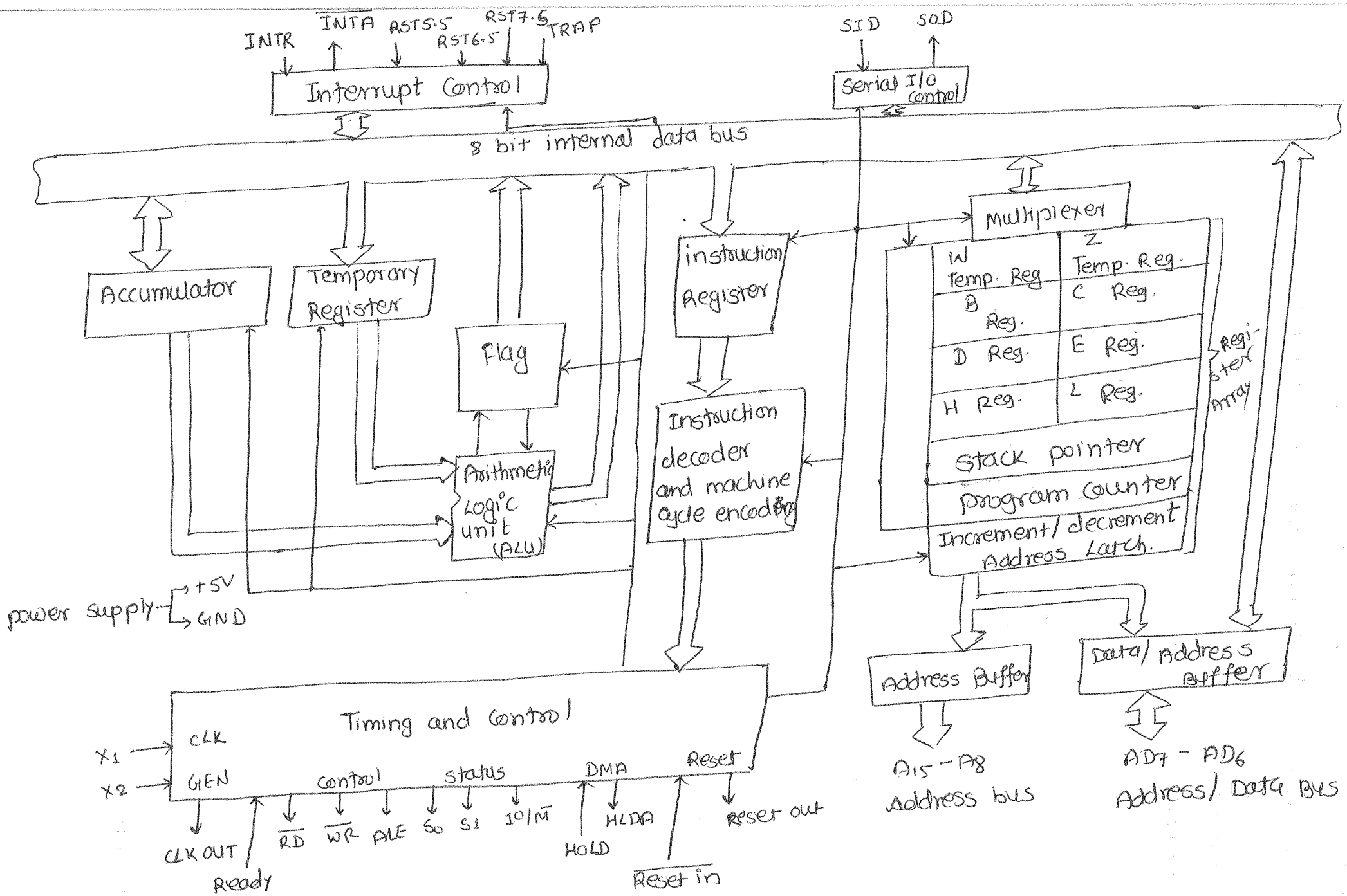


Fig: 8085 Microprocessor: Functional Block diagram

Figure above shows the internal architecture of the 8085. It includes the ALU (Arithmetic/Logic Unit), Timing and Control unit, Instruction Register and Decoder, Register Array, Interrupt control and serial I/O control.

The ALU:

The arithmetic/logic unit performs the computing functions; it includes the accumulator, the temporary register, the arithmetic and logic circuits, and five flags. The temporary register is used to hold data during an arithmetic/logic operation. The result is stored in the accumulator, and the flags (flip-flops) are set or reset according to the result of the operation.

Timing and control unit:

This unit synchronizes all the microprocessor operations with the clock and generates the control signals necessary for communication between the microprocessor and peripherals. The $\overline{RD}$ and $\overline{WR}$ signals are sync pulses indicating the availability of data on the data bus.

Instruction register and decoder:

When an instruction is fetched from memory, it is loaded in the instruction register. The decoder decodes the instruction and establishes the sequence of events to follow. The instruction register is not programmable and can not be accessed through any instructions.

Register array:

These registers are used to hold 8-bit data during the execution of some instructions. However, because they are used internally, they are not available to the programmer.

Phase of Assembly Language program execution:

1) Fetch
2) Decode
3) Execute

1) Fetch:

In fetch phase microprocessor places the content of program counter on address Bus and gets the opcode from the address. Micro-processor then saves the upcode in the instruction register.

2) Decode:

In this phase opcode from instruction register is decoded to identify the task to be performed.

3) Execute:

In this phase micro-processor generates appropriate control signal for the decoded opcode and executes.

END OF CHAPTER 1

- Prabhat chaudhary

CHAPTER 2: ASSEMBLY LANGUAGE PROGRAMMING:

The 8085 Microprocessor pin configuration:

Figure below shows the logic pinout of the 8085 microprocessor. All the signals can be classified into six groups: 1) address bus, 2) data bus, 3) control and status signals, 4) power supply and frequency signals, 5) externally initiated signals and 6) serial I/O ports.

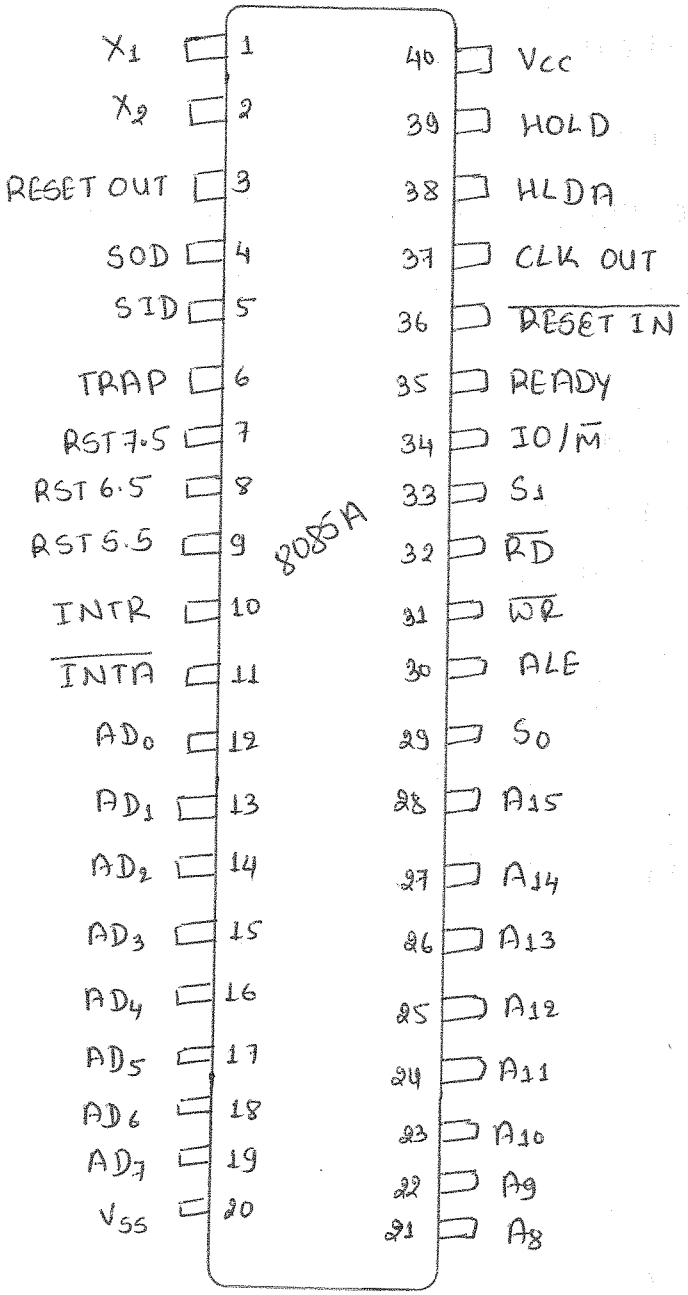


Fig: The 8085 Microprocessor pinout

Address Bus:

The 8085 has 16 signal lines that are used as the address bus; however, these lines are split into two segments: A_{15} - A_8 and AD_7-AD_0

$A_{15} - A_8$, are unidirectional and used for the most significant bit, called the high-order address.
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 the data bus. [8 bit address and data are multiplexed].

Control and status signals:

This group of signals include two control signals ($\overline{RD}$ and $\overline{WR}$), three status signals ($IO/\overline{M}$, S_1 and S_0) to identify the nature of the operation, and one special signal (ALE) to indicate the beginning of the operation.

ALE: (Address Latch Enable)

This is a positive going pulse generated every time the 8085 begins an operation (machine cycle); it indicates that the bits on $AD_7 - AD_0$ are address bits.

$\overline{RD}$ (Read): This is a Read control signal (active low). This signal indicates that the selected I/O or memory device is to be read and data are available on the data bus.

$\overline{WR}$ (write): This is a write control signal (active low). This signal indicates that the data on the data bus are to be written into a selected memory or I/O location.

$IO/\overline{M}$: This is a status signal used to differentiate between I/O and memory operations. When it is high, it indicates an I/O operation; when it is low, it indicates a memory operation. This signal is combined with $\overline{RD}$ and $\overline{WR}$ to generate I/O and memory control signals.

S_1 and S_0 : These status signals can identify various operations, but they are rarely used in small systems.

S_0	S_1	operation
0	0	Halt (stop)
0	1	Read
1	0	Write
1	1	fetch

power supply and clock frequency:

V_{CC} : +5V power supply

V_{SS} : Ground Reference

X_1, X_2 : A crystal (or RC, LC n/w) is connected at these two pins. The frequency is internally divided by two; therefore, to operate a system at 3 MHz, the crystal should have a frequency of 6 MHz.

CLK (OUT) : Clock output : This signal can be used as the system clock for other devices.

Externally Initiated signals, including interrupts:

INTR (input) : Interrupt Request : This is used as a general-purpose interrupt; it is similar to the INT signal for the 8080A.

INTA (output) : Interrupt Acknowledge : This is used to acknowledge an interrupt.

RST 7.5 (inputs) :
RST 6.5
RST 5.5 : Restart interrupts : These are vectored interrupts that transfer the program control to specific memory locations. They have higher priorities than the INTR interrupt. Among these three, the priority order is 7.5, 6.5 and 5.5.

TRAP (Input) : This is a nonmaskable interrupt and has the highest priority.

HOLD (Input) : This signal indicates that a peripheral such as a DMA (Direct memory access) controller is requesting the use of the address and data buses.

HLDA (Output) : Hold acknowledge : This signal acknowledges the HOLD request.

READY (input) : This signal is used to delay the microprocessor Read or Write cycle 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.

RESET IN : When the signal on this pin goes low, the program counter is set to zero, the buses are tri-stated, and the 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 : serial input data. This pin receives data serially from peripheral serial devices.

SOD : serial output data. This pin outputs data serially to peripheral serial devices.

Registers in 8085 :

1) General purpose register
2) Temporary Register
3) Special purpose Register
4) 16 bit Register

1) General purpose Register :

B, C, D, E, H, L are general purpose register. They can store 8 bit of data at a time. 16 bit of data can also be stored by using combination of BC, DE and HL. The general purpose register is available for the user to manipulate.

2) Temporary Register :

W and Z are each of 8 bit and are used by processor for its internal operation and not available to the user or programmer.

3) Special purpose Register :

Accumulator and Flag are the two special purpose register each of 8 bit. Accumulator is extensively used to store

results of an arithmetic and logical operation. It is also used in input output operations.

Flag Register holds the properties of an arithmetical and logical operation. It is also, 8 bit register.

S	Z	x	Ac	x	P	x	Cy

Fig: 8 bit flag register
[program status word (PSW)]

Sign (S) bit (Flag):

After the execution of an arithmetic or logic operation, if bit D_7 of the result is 1, the sign flag is set. This flag is used with signed numbers.

If S = 1, the result is -ve

If S = 0, the result is +ve.

Z - Zero flag:

The zero flag is set if the ALU operation results in 0, and the flag is reset if the result is not 0.

i.e. If Z = 0, the result is not zero

If Z = 1, the result is zero

Ac - Auxiliary Carry Flag:

In an arithmetic operation, when a carry is generated by digit D_3 and passed on to digit D_4, the AC flag is set. The flag is used only internally for BCD operations.

If AC = 0, there is no carry from D_3 bit

If AC = 1, there is carry from D_3 bit

P - Parity flag:

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

i.e. If P = 0, there is odd number of 1's in the result

If P=1, there is even number of 1s in the result.

Cy - Carry flag:

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

i.e. If, Cy = 0, there is no carry from D_7 bit

If, Cy = 1 there is carry from D_7 bit

4) 16-bit register:

Stack / stack pointer and program counter are 16 bit registers. Stack stores the value of flag and program counter during interrupt operation or subroutines call. Stack pointer holds the address of top of the stack. Program counter stores the address of next instruction to be executed.

8085 instruction set:

The following notations are used in the description of the instructions.

R = 8085 8-bit register (A, B, C, D, E, H, L)

M = Memory Register (location)

Rs = Register source

Rd = Register destination

Rp = Register Pair

( ) = Contents of

1. Data Transfer (copy) Instructions:

These instructions perform the following six operations.

- Load an 8-bit number in a register
- Load 16-bit number in a register pair
- Copy from register to register
- Copy between register and memory
- Copy between I/O and accumulator
- Copy between registers and stack memory.

S.N.	Mnemonics	Examples	Operation
1.1	MVI R,8-bit	MVI B, 4F H	Load 8-bit data (byte) in a Register
1.2	MOV Rd, Rs	MOV B, A MOV C, A	Copy data from source register Rs into destination register Rd.
1.3	LXI Rp, 16-bit	LXI B, 2050H	Load 16 bit number in a register pair
1.4	OUT 8-bit (port address)	OUT 01 H	Send (write) data byte from the accumulator to an output device.
1.5	IN 8-bit (port address)	IN 07 H	Accept (read) data byte from an input device and place it in the accumulator.
1.6	LDA 16-bit	LDA 2050 H	Copy the data byte into A from the memory specified by 16-bit address.
1.7	STA 16-bit	STA 2070 H	Copy the data byte from A into the memory specified by 16-bit address
1.8	LDAX Rp	LDAX B	Copy the data byte into A from the memory specified by the address in the register pair.
1.9	STAX Rp	STAX D	Copy the data byte from A into the memory specified by the address in the register pair.
1.10	MOV R, M	MOV B, M	Copy the data byte into register from the memory specified by the address in HL register.
1.11	MOV M, R	MOV M, C	Copy the data byte from the register into memory specified by the address in HL register.

2. Arithmetic Instructions:

The frequently used arithmetic operations are:

- Add
- Subtract
- Increment (Add 1)
- Decrement (subtract 1)

2.1	ADD R	ADD B	Add the content of a register to the content of A and stores result at A
2.2	ADI 8-bit	ADI 37 H	Add 8-bit data to the content of A
2.3	ADD M	ADD M	Add the contents of memory to A; the address of memory is in HL register

2.4	SUB R	SUB C	Subtract the content of a register from the content of A and stores result at A
2.5	SUI 8-bit	SUI 7F H	subtract 8-bit data from the contents of A.
2.6	SUB M	SUB M	subtract the content of memory from A; the address of memory is in HL register.
2.7	INR R	INR D	Increment the content of a register
2.8	INR M	INR M	Increment the content of memory, the address of which is in HL.
2.9	DCR R	DCR E	Decrement the content of a register
2.10	DCR M	DCR M	Decrement the content of a memory, the address of which is in HL.
2.11	INX Rp	INX H	Increment the contents of a register pair
2.12	DCX Rp	DCX B	Decrement the contents of a register pair
2.13	ACI 8-bit	ACI 55 H	ADD 8-bit data ~~with~~ to content of A and carry out
2.14	ADC r	ADC B	ADD the content of register, carry out and content of A and stores result at A.
2.15	ADC M	ADC M	ADD the content of memory, carry out and content of A.
2.16	SBI 8-bit	SBI 0A H	subtracts 8-bit value and borrow value from content of A, and stores the result at A.
2.17	SBB r	SBB C	subtracts the content of C and borrow value from content of A
2.18	SBB M	SBB M	subtracts the content of memory and borrow value from content of A
2.19	DAD Rp	DAD B	Add the content of register pair and content of HL pair and stores the result into HL pair.

2.20	DAA (Decimal adjust accumulator)	DAA	Converts the hexadecimal value of accumulator to decimal value.

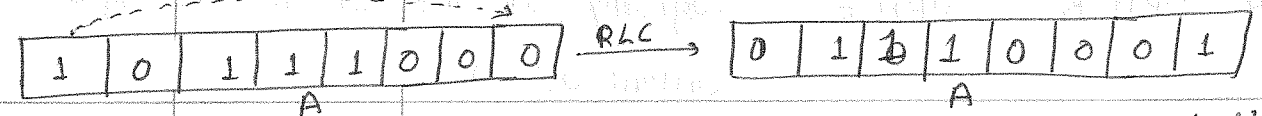
3. Logic and Bit Manipulation Instructions

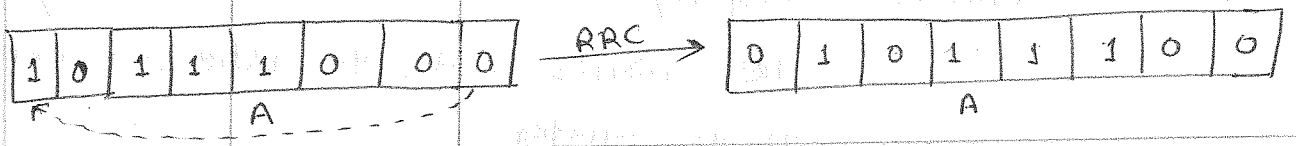
These instructions include the following operations:

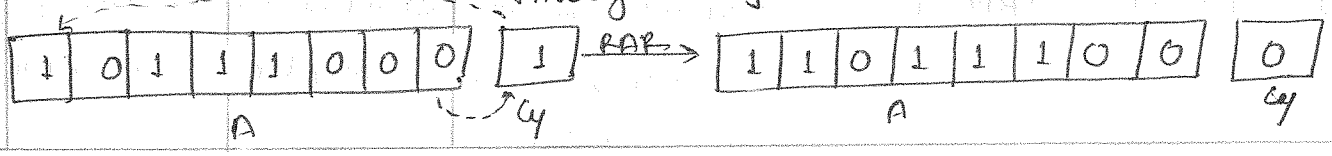
• AND • OR • X-OR (Exclusive OR) • Compare • Rotate Bits

3.1	ANA R	ANA B	Logically AND the contents of a register with the contents of A
3.2	ANI 8-bit	ANI 2FH	Logically AND 8-bit data with the content of A
3.3	ANA M	ANA M	Logically AND the contents of memory with the contents of A; the address of memory is in HL register.
3.4	ORA R	ORA E	Logically OR the content of register with the content of A
3.5	ORI 8-bit	ORI 3FH	Logically OR 8-bit data with the contents of A.
3.6	ORA M	ORA M	Logically OR the contents of memory with the contents of A; the address of memory is in HL register.
3.7	XRA R	XRA B	Exclusive-OR the contents of a register with the contents of A
3.8	XRI 8-bit	XRI 6A H	Exclusive-OR the 8-bit data with the contents of A.
3.9	XRA M	XRA M	Exclusive-OR the contents of memory with the contents of A; the address of memory is in HL register.
3.10	CMP R	CMP B	Compare the contents of register with the contents of A for less than, equal to or greater than.

3.11	CPI 8-bit	CPI 4FH	Compare 8-bit data with the contents of A for less than, equal to, or greater than
3.12	CMP M	CMP M	Compare the content of memory with the content of A for less than, equal to or greater than.
		If, A > 8-bit data/R/M, S=0 Z=0 Cy=0 A = 8-bit data/R/M, S=0 Z=1 Cy=0 A < 8-bit data/R/M, S=1 Z=0 Cy=1	
3.13	STC	STC	This instruction sets the carry flag (Cy).
3.14	CMC	CMC	This instruction complements the value of carry flag (Cy)
3.15	RLC	RLC	Rotates the value of accumulator to the left by 1 bit.
3.16	RRC	RRC	Rotates the value of accumulator to the right by 1 bit
3.17	RAL	RAL	Rotate the value of accumulator to the left through carry out flag by 1 bit.
3.18	RAR	RAR	Rotate the value of accumulator to the right through carry out flag by 1 bit







4. Branch Instruction:

A. Jump Instruction:

i) Unconditional Jump:

JMP 16-bit address:

This instruction makes the program jump i.e. change the value of program counter to the address location specified in the instruction.

eg: JMP D000H.

ii) Conditional Jump:

This instruction changes the value of program counter (i.e. jumps) as specified in the instruction and according to the condition specified.

S.N.	Instruction	Description	Condition
1.	JC	Jump on carry	$C_y = 1$
2.	JNC	Jump on no carry	$C_y = 0$
3.	JP	Jump on plus	$S = 0$
4.	JM	Jump on minus	$S = 1$
5.	JPE	Jump on parity even	$P = 1$
6.	JPO	Jump on parity odd	$P = 0$
7.	JZ	Jump on zero	$Z = 1$
8.	JNZ	Jump on no zero	$Z = 0$

eg: JNZ C000H, JC C123H

B) Call instruction:

i) Unconditional call:

CALL 16-bit address:

This instruction calls the next program with the address specified in the instruction, from the program currently being executed.

eg: CALL D123H.

ii) Conditional call:

This instruction calls the next program (subroutine) based on the condition specified in the instruction.

S.N.	Instruction	Description	condition
1.	CC	call on carry	$C_y = 1$
2.	CNC	call on no carry	$C_y = 0$
3.	CP	call on plus	$S = 0$
4.	CM	call on Minus	$S = 1$
5.	CPE	call on parity Even	$P = 1$
6.	CPO	call on parity odd	$P = 0$
7.	CZ	call on zero	$Z = 1$
8.	CNZ	call on no zero	$Z = 0$

Eg: CZ F12F H

CP D000 H

c) Return instruction:

i) Unconditional return:

RET

This instruction returns the program from subroutine to main/calling program unconditionally.

ii) Conditional return:

It returns the program from subroutine to main/calling program according to the condition specified.

S.N.	Instruction	Description	Condition
1.	RC	Return on carry	$C_y = 1$
2.	RNC	Return on no carry	$C_y = 0$
3.	RP	Return on plus	$S = 0$
4.	RM	Return on Minus	$S = 1$

5.	RPE	Return on parity even	P = 1
6.	RPO	Return on parity odd	P = 0
7.	RZ	Return on zero	Z = 1
8.	RNZ	Return on no zero	Z = 0

eg: RZ F12F H

RP C010 H

Subroutine:

Subroutine is a group of instructions that performs sub-task of repeated occurance. It is written as separate unit apart from the main program and the Microprocessor transfers the program execution from the main program to the subroutine whenever it is called to perform the task. After the execution of the task completes the microprocessor returns to main program. Re-calling of sub-routine is carried out by call instruction CALL. After the CALL the value of program counter is stored into the stack and the program counter now stores with value of initial location of the subroutine. After the subroutine is executed it returns to the main program, at this point the value of the program counter is now loaded from the stack and the main program continues.

Stack:

Stack is the portion of read/write memory setaside by user to store data or address, temporarily during subroutine or interrupt function call. There are two types of stack operation: PUSH and POP

PUSH operation:

When the information is written into the stack the operation is called PUSH operation.

POP operation:

When the information is retrived from the stack then it is

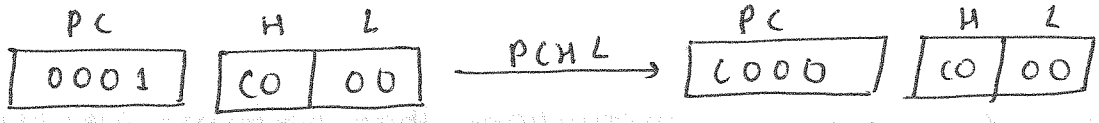
called POP operation.

5. Stack related instructions:

i) PCHL:

This instruction copies the 16 bit data of 'HL' pair to the program counter.

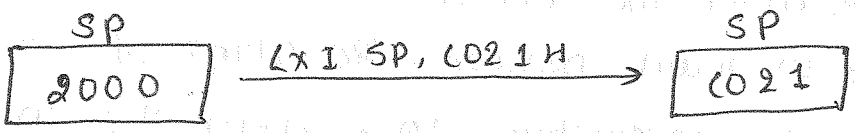
i.e. HL → PC



ii) LXI SP, 16-bit data/address:

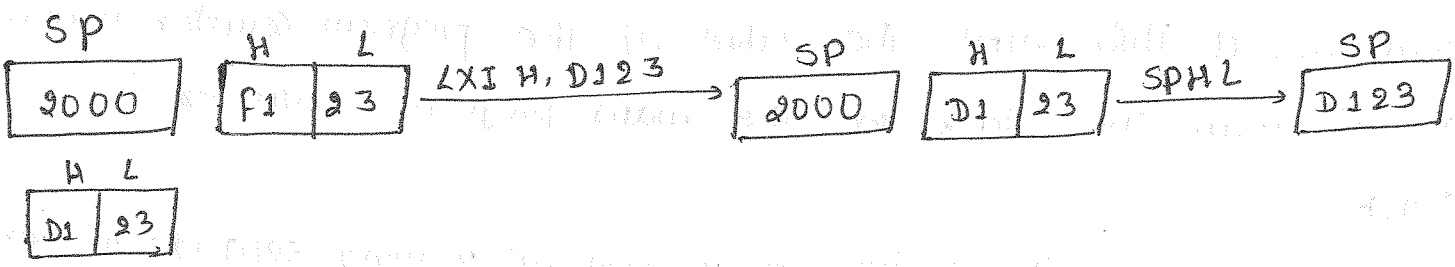
It copies the 16-bit immediate data to the stack pointer.

Eg: LXI SP, C021 H



iii) LXI H, 16-bit data and SPHL

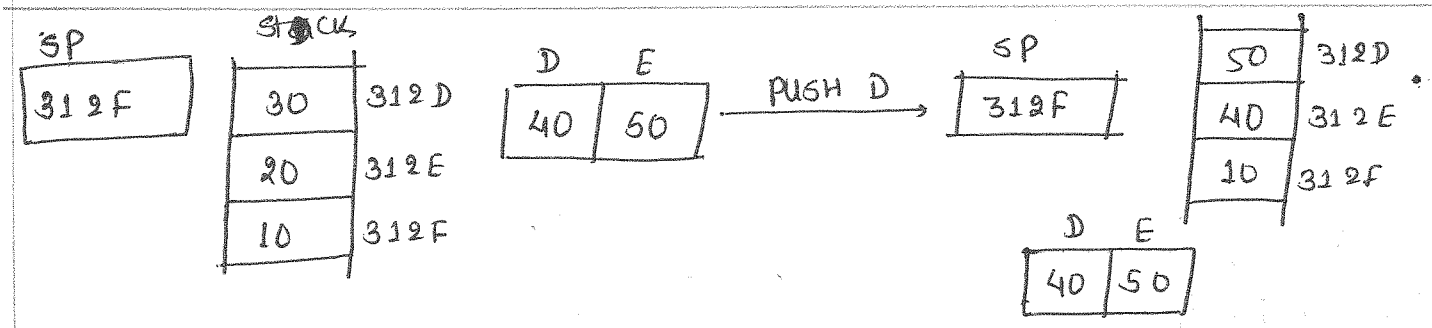
The 16 bit data is first copied in 'HL' pair register and then moved to stack pointer by SPHL



iv) PUSH rp:

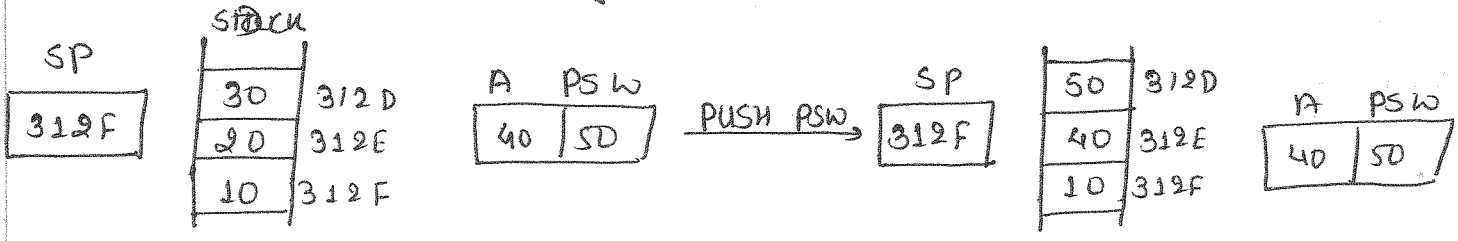
This instruction pushes the data temporarily into stack. For this first SP is decremented by 1 and higher order byte is loaded into the stack. Again the value of SP is decremented by 1 and lower order byte is loaded into the stack.

Eg: PUSH D.



v) PUSH PSW :

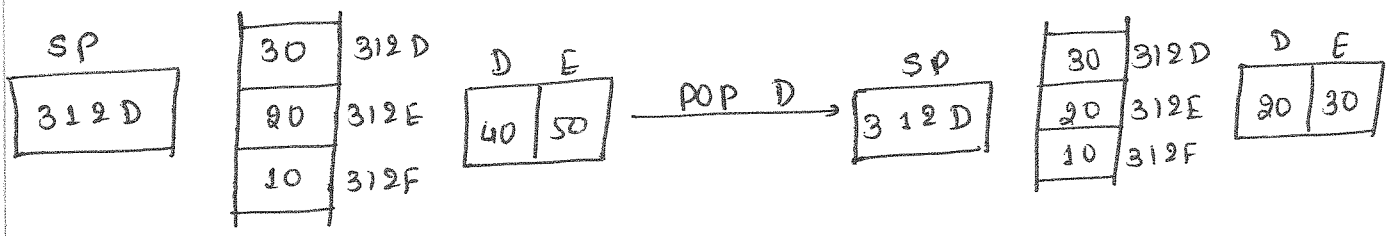
This instruction pushes the value of accumulator into (SP-1) location and that of flag register into (SP-2) location.



vi) POP rp :

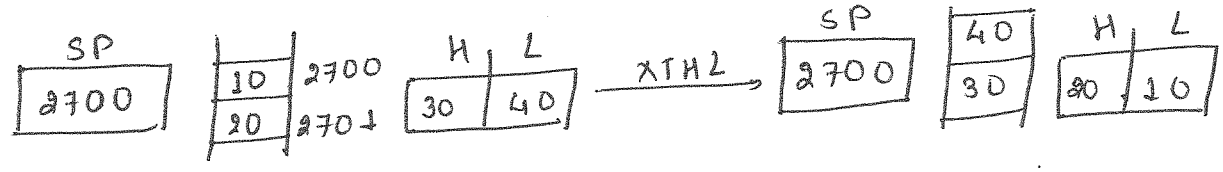
During execution of this instruction, the value in the location pointed by SP is popped into 2nd register of the register pair. The SP is now incremented by 1 and the value in the stack of this location is popped into 1st register of the register pair.

eg: POP D.



vii) XTHL

This instruction exchanges the value of HL pair with the stack location pointed by SP.



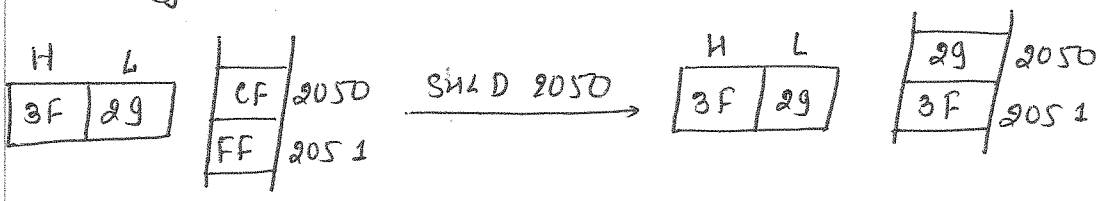
Remaining instructions from Data transfer group:

- SHLD address :

This instruction copies the value of 'L' register to the

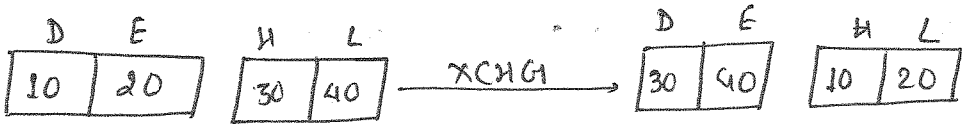
given address location and value of 'H' register to one location above the given location.

Eg: SHLD 2050



• XCHG:

This instruction exchanges the value of DE and HL pair such that value of D is exchanged with H and that of E is exchanged with L.



• HLT:

This instruction stops the program currently being executed.

• NOP:

This instruction temporarily pauses execution of program for 1 instruction cycle.

$$1 \text{ instruction cycle} = 4 \text{ T state} = 4 \times \frac{1}{\text{clock frequency}}$$

programming [Assembly Language]

Q.1 Wap to copy 1BH to register D and copy the value of D to register B.

Soln:

```
MVI D, 1BH    // Copy value 1BH to register D
MOV B, D      // Move value of register D to register B
HLT           // terminate program.
```

Q.2 Write a program to read content of location C000H mask the lower 4-bit (nibble) and stores the result into location C001H.

Soln: LDA C000H // Load content of C000H to accumulator

```
ANI FOH     // ANDing with FO H
STA C001 H  // Store the content of ACC. into C001H.
HLT         // terminate the program.
```

Q.3 WAP to do following

ⓐ Load A with byte 1

ⓑ Load B with byte 2

ⓒ Compare the content of A and B

ⓓ If two numbers are equal, display 00H at port 1, otherwise display 01H.

Soln:

```
    MVI A, byte 1
    MVI B, byte 1
    CMP B
    JNZ down
    MVI A,00H
    OUT port 1
    JMP Exit
down: MVI A,01H
    OUT port1
Exit: HLT
```

NOTE:

LHLD 16-bitaddress:

This instruction loads the contents of the 16-bit memory location into the HL register pair

LHLD 3000

content of 3000 → L

Content of 3001 → H

Q.4 Load the data 91H into register B and 87H into C. Mask all bits except D_0 from register B and C. If D_0 is logic 1 in both registers then output 01H at port 1.

Soln:

```
    MVI B, 91H
    MVI C, 87H
    MOV A, B
    ANI 01H
    MOV B, A
```

```
MOV A,C
ANI 01H
ANA B
CPI 01 H
JNZ Exit
MVI A,01H
Out port 1
Exit: HLT.
```

Q.5 WAP to find the larger of two numbers stored in C001H and C002H. The result should be stored in C003H.

```
Soln:  LDA C001H
       MOV B, A
       LDA C002H
       CMP B
       JNC down
       MOV A, B
       STA C003H
       Jmp Exit
down:  STA C003H
Exit : HLT
```

Q.6 WAP to add two 8 bit numbers : 1st number is at C000H and and number is at C001H. The result are to be stored at C002H and C003H.

```
       MVI B, 00H
Soln:  LXI H, C000H
       MOV A, M
       INX H
       ADD M
       JNC down
       INR B
down:  INX H
       MOV M, A
       INX H
       MOV M, B
       HLT.
```

phase of Assembly Language program execution:

i) Fetch
ii) Decode
iii) Execute

i) Fetch:

In fetch phase microprocessor places the content of program counter on address Bus and gets the opcode from the address. Micro-processor then saves the opcode in the instruction register.

ii) Decode:

In this phase opcode from instruction register is decoded to identify the task to be performed.

iii) Execute:

In this phase micro-processor generates appropriate control signal for the decoded opcode and executes.

Addressing modes of 8085:

The different ways in which micro processor excess the data is called addressing mode. There are five different types of addressing modes in 8085.

i) Immediate addressing mode
ii) Register addressing mode
iii) Direct addressing mode
iv) Indirect addressing mode
v) Implied addressing mode

i) Immediate ~~adding~~ addressing mode:

In this mode 8 bit of data or operand itself is provided in the instruction.

Eg: MVI C, 40H

LXI B, D000H

ii) Register addressing mode:

In this mode, both source and destination of data are registers.

Eg: MOV A, B

MOV H, L

iii) Direct addressing mode:

In this mode 16 bit address from where data is to be read from or written to is directly provided in the instruction.

Eg: STA C000H

LDA D001H.

iv) Indirect addressing mode:

In this mode address is provided indirectly through M register or register pairs.

Eg: MOV B, M : move content of M into register B

STAX B : store value of Acc into location pointed by Bc pair

v) Implied or Inherent addressing mode:

These are the instructions whose meaning are implied by themselves. The instruction of this mode do not have operands.

Eg: NOP : No operation

HLT : halt

Classification of instruction based on size:

The 8085 instruction set is classified into the following

Three groups according to word size or byte size.

1. 1-byte instructions
2. 2-byte instructions
3. 3-byte instructions

One-BYTE instruction:

A 1-byte instruction includes the opcode and the operand in the same byte. for example:

Task	Address	opcode	operand	Hex code
Copy the content of ACC in reg. C	C000H	MOV	C, A	4F H
Add the contents of register B to the content of accumulator	C001H	ADD	B	80H.

These instructions are stored in 8 bit binary format in memory; each requires one memory location.

TWO BYTE instruction:

In a 2-byte instruction, the first byte specifies the operation code and the second byte specifies the operand. For example:

Task	Address	opcode	Operand	Hex code
Load an 8-bit data byte in the accumulator	2000H 2001H	MVI	A, 32H	3E 32
Load an 8-bit data byte in register B	2002H 2003H	MVI	B, F2H	06 F2

These instructions would require two memory locations each to store the binary codes.

THREE-BYTE instruction:

In a 3-byte instruction, the first byte specifies the opcode,

and the following two bytes specify the 16-bit address. Note that the second byte is the low-order address and the third byte is the high-order address. For example:

TASK	Address	op code	operand	Hex value
Load content of memory 2050 H into accumulator	3000 H	LDA	2050 H	3A H
	3001 H			50 H
	3002 H			20 H
Transfer the program sequence to Memory location 2085 H	8000 H	JMP	2085 H	C3 H
	8001 H			85 H
	8002 H.			20 H

The instruction would require 3 memory location each to store the binary codes.

Instruction timing diagram:

The execution of any program consist of a sequence of read/write operation on which each operation transfers a byte of data between micro-processor and particular memory location. These read/write operations are only communication between microprocessor and other components and all that is necessary to execute any instruction or program.

i) Instruction cycle:

It is the time required to complete the execution of any instruction. The 8085 consists of 1 to 6 Machine cycle.

ii) Machine cycle:

It is defined as the time required to complete one operation such as memory read/write, I/O read/write or acknowledging external request. It consist of 3 to 6 T-states.

iii) T-state:

It is the time required to perform a machine cycle in one clock period called sub-division or sub-task. This sub-division is synchronized with system clock, and each T-state is equal to one clock period.

Op-code fetch Machine cycle:

The first operation in any instruction is op-code fetch, The microprocessor needs to get (fetch) this machine code from the memory register where it is stored before the microprocessor can begin to execute the instruction. Op-code fetch cycle is of 4 clock cycle.

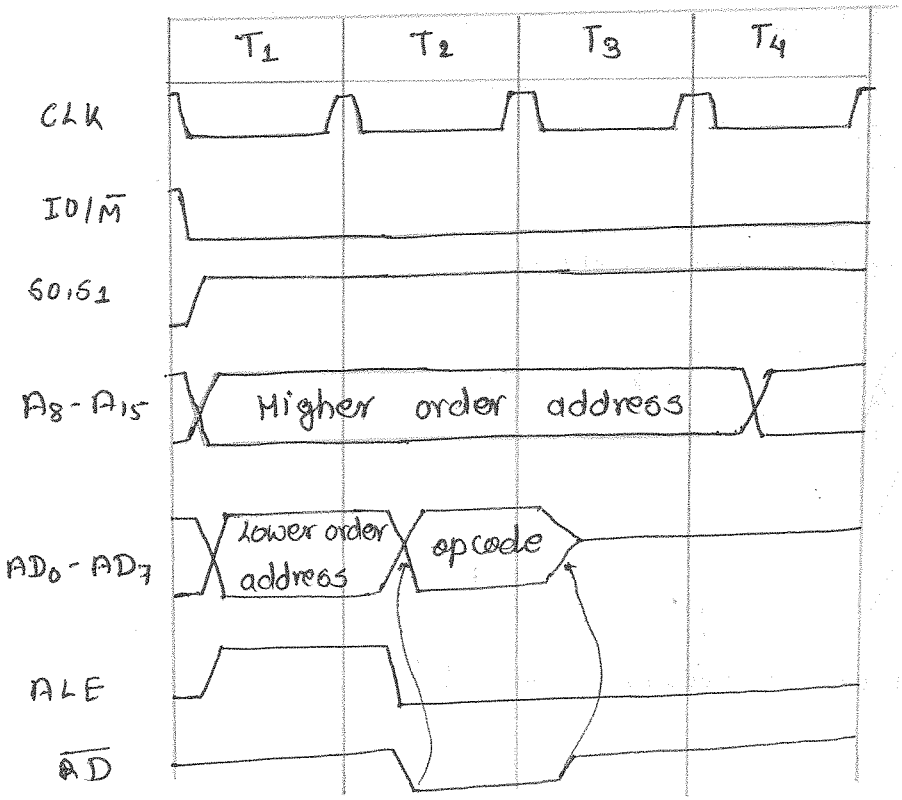


Fig: Timing diagram of opcode fetch.

Step1: Microprocessor places the 16 bit memory address from program counter on the address bus. At T_1, high order address is placed at $A_8 - A_{15}$ and lower order address is placed at $AD_0 - AD_7$. ALE signal goes high. IO/$\bar{M}$ goes low and both S_0 and S_1 goes high for op-code fetch.

Step2: The control unit sends the control signal $\bar{RD}$ to enable the memory chip and active during T_2 and T_3.

Step3: The byte from the memory location is placed on the data bus.

Step4: The instruction is decoded and executed.

Memory Read cycle:

- The memory Read cycle is executed by the processor to read a data byte from memory.
- The processor takes 3T states to execute this machine cycle.

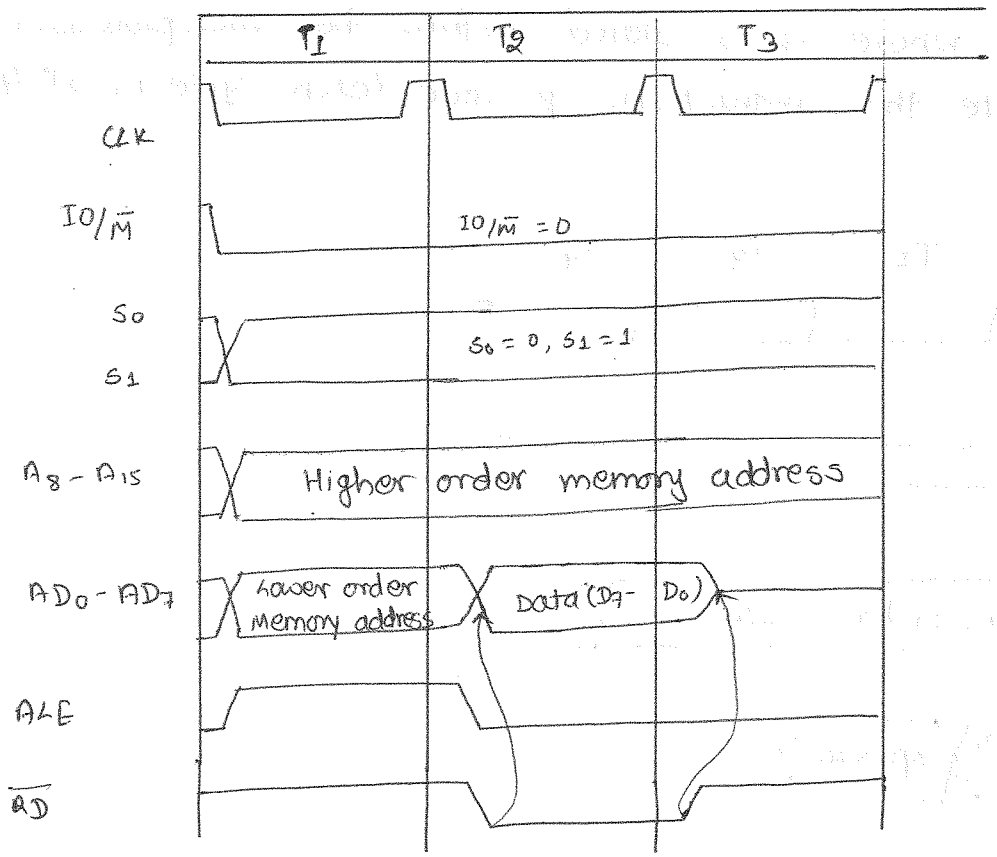


Fig: Timing diagram of Memory Read cycle.

Memory write cycle:

- The memory write machine cycle is executed by the processor to write a data byte in a memory location.
- The processor takes, 3T states to execute this machine cycle.

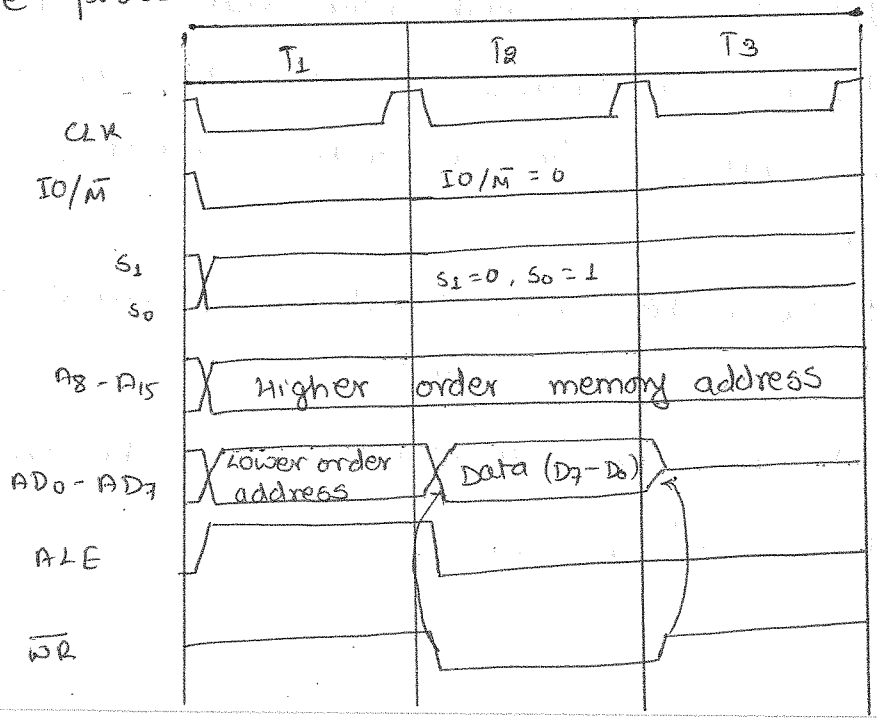


Fig: Timing diagram of memory write cycle.

IO Read cycle:

- The I/O Read cycle is executed by the processor to read a data byte from I/O port or from the peripheral, which is I/O, mapped in the system.
- The processor takes 3T states to execute this machine cycle.
- The IN instruction uses this machine cycle during the execution.

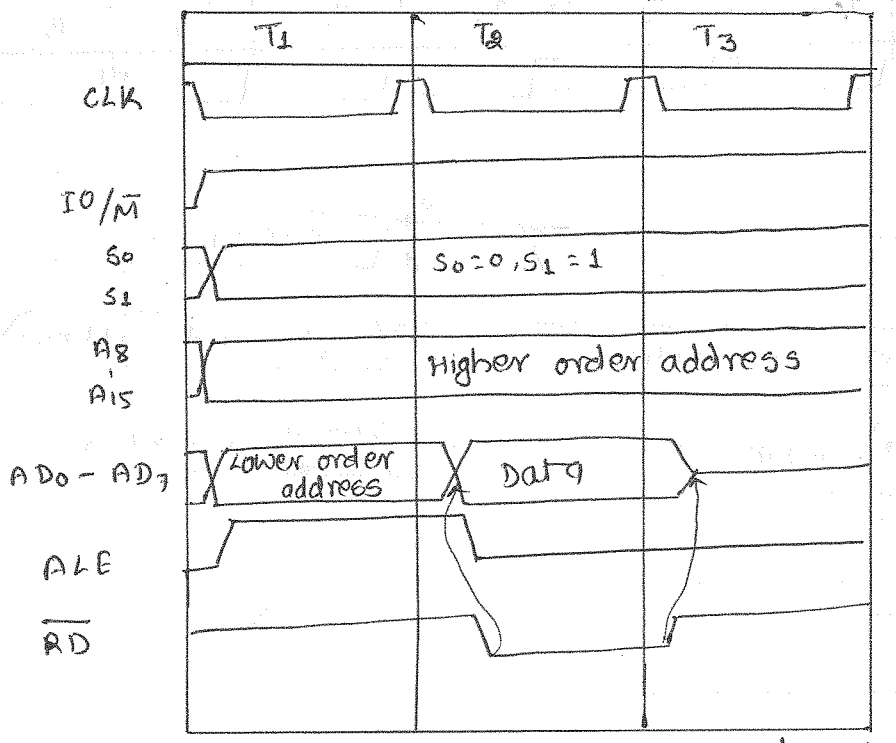


Fig: Timing diagram of IO Read cycle.

IO Write cycle:

- The I/O write cycle is executed by the processor to write a data byte into I/O port.
- The processor takes 3T states to execute this machine cycle.
- The OUT instruction uses this machine cycle during the execution.

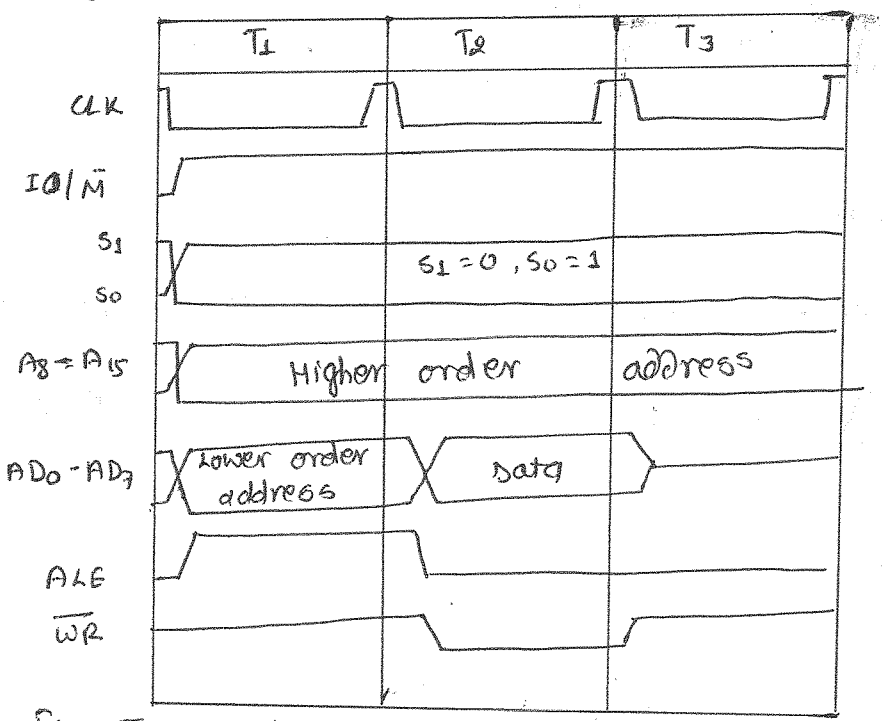


Fig: Timing diagram of IO Write cycle.

Q. Draw the timing diagram for MVI B, 04H.

Soln: This instruction copies the value 04H into register B. So, this instruction includes Op-code fetch and Memory read to reed data 04H. So, MVI B,04H has 7T states.

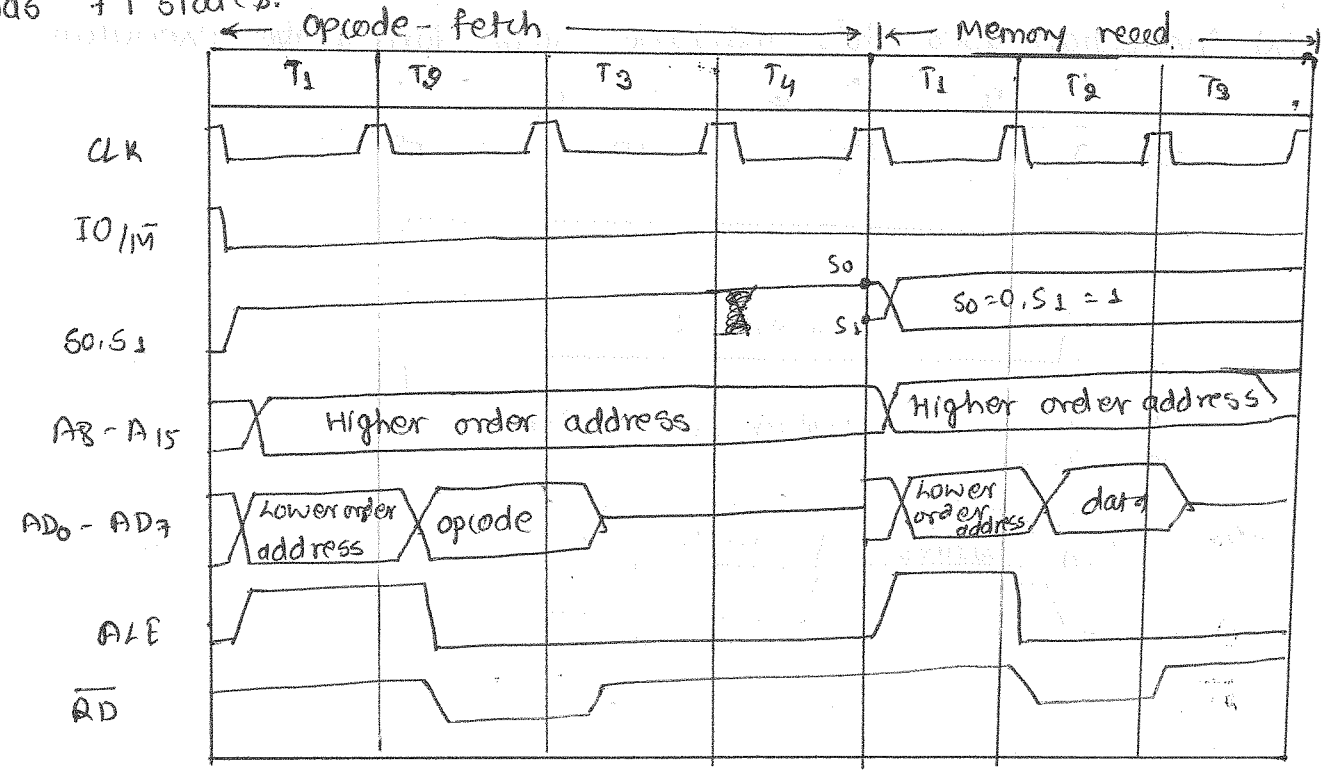


Fig: Timing diagram for MVI B, 04H.

Q. Draw a timing diagram for IN 60H.

Soln: IN 60H has Opcode fetch, Memory read and I/O read machine cycle.

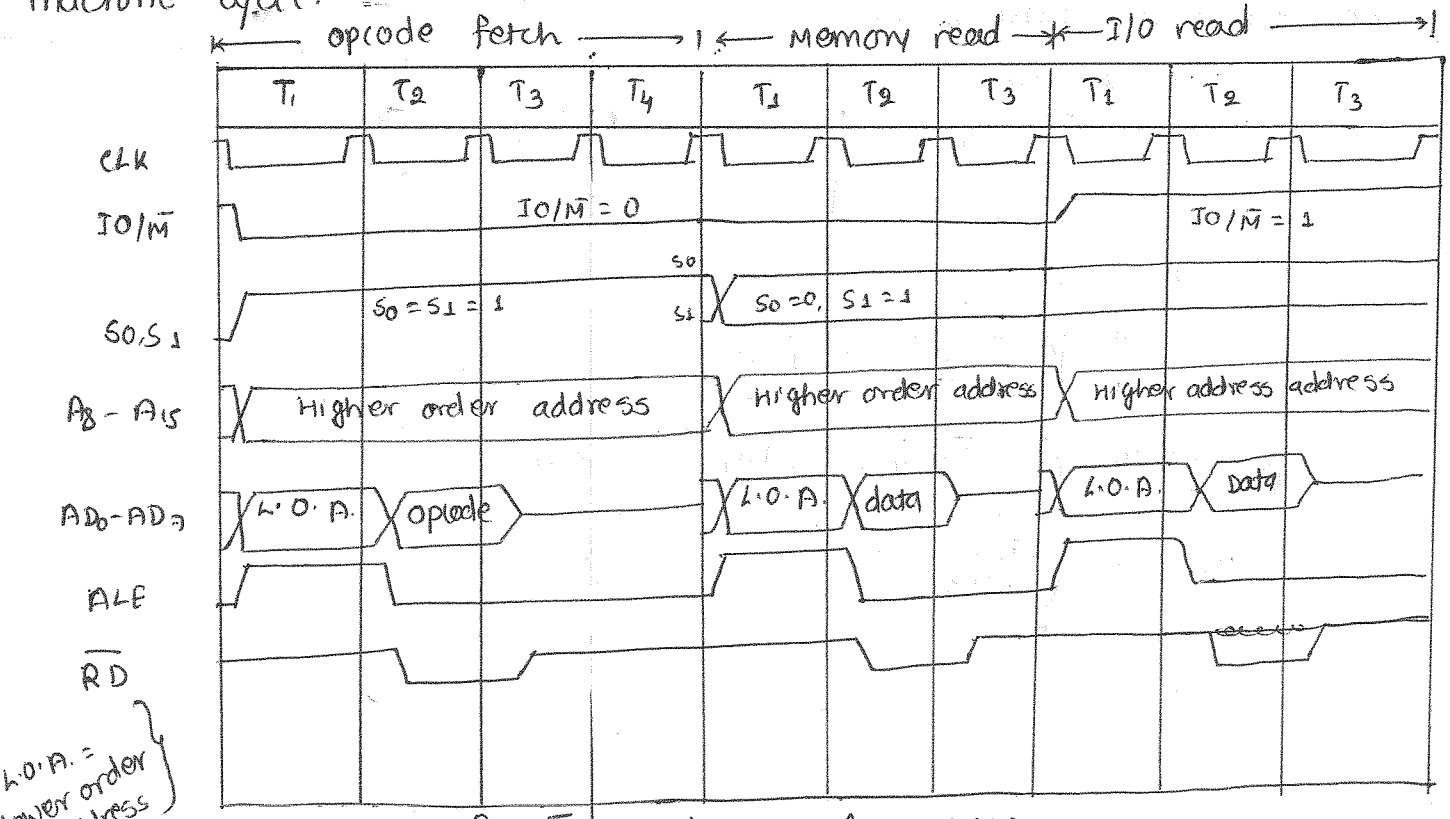


Fig: Timing diagram for IN 60H.

Q. Draw a Timing diagram for INR M.

Soln:
- Fetching the opcode 34H from the memory 4105H. (OF cycle)
- Let the memory address (M) be 4250H (MR cycle - To read memory address and data)
- Let the content of that memory is 12H.
- Increment the memory content from 12H to 13H. (MW machine cycle)

Address	Mnemonics	op-code
4105 H	INR M	34H.

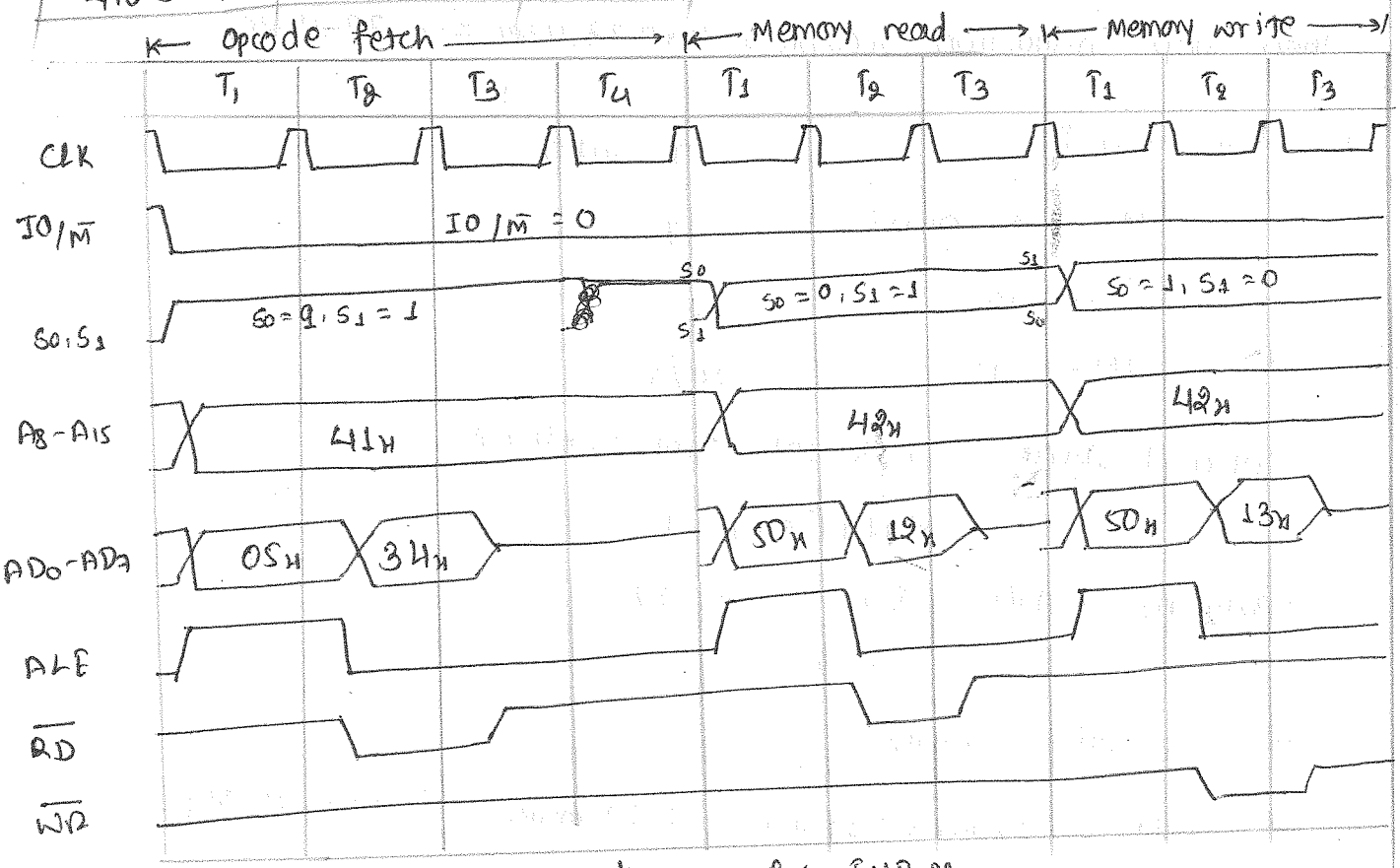


Fig: Timing diagram for INR M.

NOTE:

STA 2000H → Opcode fetch + Memory Read + Memory Read + Memory write

LDA 2000H → Opcode fetch + Memory Read + Memory Read + Memory read

MOV A, B → Opcode fetch

OUT 60H → Opcode fetch + Memory read + IO write

LXI H, C000H → Opcode fetch + memory read + Memory read

DCX H, INX H → opcode fetch (6-T states)

MVI M, 30H → Opcode fetch + memory read + memory write

Time delay generation and counter program:

1) NOP

Total T-state occupied by 1 NOP = 4T state

For a crystal oscillator of 6MHz,

Operating frequency, $f = \frac{6}{2} = 3\,MHz$

Clock period, $T = \frac{1}{f} = \frac{1}{3\,MHz} = 0.33\,\mu sec.$

∴ 1-T state occupies 0.33 μsec.

Then, NOP instruction occupies $4 * 0.33\,\mu sec = 1.32\,\mu sec.$

2) 8-bit Counter:

	T-states
MVI B, count	7
Up: DCR B	4
JNZ Up	10/7

Total T-state = $7 + (4+10)(count-1) + 4 + 7$

$= 18 + 14(count - 1)$

Maximum T-state = $18 + 14(255 - 1)$

$= 3574$

for a crystal of 6MHz,

Max. delay = $3574 \times 0.33\,\mu sec = 1.19\,msec.$ $\left\{3574 \times \frac{1}{3}\,\mu sec\right\}$

3) 16-bit Counter:

	T-states
LXI B, count (16-bit)	10
Up: DCX B	6
MOV A, C	4
ORA B	4
JNZ Up	10/7

Total T-state = $10 + (6+4+4+10)(count-1) + 6+4+4+7$

$= 31 + 24(count - 1)$

Max. T-state = $31 + 24(65535 - 1) = 1572847$

For a crystal of 6 MHz,

Maximum delay = $1572847 \times \frac{1}{6}$ μsec. = 0.52 sec.

Q1. WAP to count continuously in hexadecimal from FF to 00 in a system with 0.5μsec clock period. Set up 1 msec delay between each count and display the numbers at port 30H.

soln: Main program:

```
      MVI  A, FF H
up:   OUT  30 H
      CALL delay    ; generates 1 msec delay
      DCR  A
      JNZ  up
      HLT
```

Delay program

```
delay: MVI  B, 8FH.
up:    DCR  B
       JNZ  up
       RET
```

Delay calculation:

Total T-state = 18 + 14 (count - 1)

Total delay = [18 + 14 (count - 1)] × 0.5μs

or, $\frac{1\,ms}{0.5\mu s} - 18 = 14\,(count - 1)$

or, count = 142 9.57

= 143

= 8F

Q2. WAP to continuously generate a square wave with a period of 500μsec. Assume that the system clock period is 325 nanosec. Use the port 30 to generate the square wave.

Soln:

Main program:

```
up: MVI A,01 H
    OUT 30 H
    CALL delay    ; 500 usec. delay
    MVI A,00 H
    CALL delay    ; 500 usec. delay
    JMP up
    HLT
```

Delay program:

```
delay: MVI B, 6E H
  up: DCR B
      JNZ up
      RET
```

Delay calculation:

Total T state = 18 + 14 (count - 1)

Total delay = [18 + 14 ((count - 1)] * 325 x 10^{-9}

or, $\frac{500 \times 10^{-6}}{325 \times 10^{-9}} - 18 = 14 (count - 1)$

count = 109.6

= 110 = 6E H

Q3. WAP to continuously generate a rectangular wave with a period of 500 usec. 'on' period and 1 msec. 'off' period. Assume the system clock period is 325 nsec. use port 30 to generate the ~~square~~ rectangular wave.

Soln:

Main program:

```
up: MVI A, 01 H
    OUT 30 H
    CALL delay    ; 500 usec. delay
    MVI A,00H
    OUT 30 H
    CALL delay  }
    CALL delay  }  ; 1 msec delay
    JMP up
    HLT
```

Delay subroutine:

```
      MVI  B, 6E H
up:   DCR  B
      JNZ  up
      RET
```

Q4. WAP to continuously generate a rectangular wave with a period of 500 usec. 'On' period and 95% duty cycle. Assume the system clock period is 325 nsec. Use port 30 to generate the rectangular wave.

soln: $\text{duty cycle} = \frac{T_{on}}{T_{on} + T_{off}} \times 100\%$

or, $95\% = \frac{T_{on}}{T_{on} + T_{off}} \times 100\%$

or, $4\,T_{on} = T_{on} + T_{off}$

or, $T_{off} = 3 \times T_{on} = 3 \times 500\,\text{usec}$

$= 1500\,\text{usec}$.

Main program:

```
Up:  MVI  A, 01 H
     OUT  30 H
     CALL delay
     MVI  A, 00 H
     OUT  30 H
     CALL delay
     CALL delay
     CALL delay
     JMP  up
     HLT
```

Delay program:

```
      MVI  B, 6E H
up:   DCR  B
      JNZ  up
      RET
```

RTL (Register Transfer Logic)

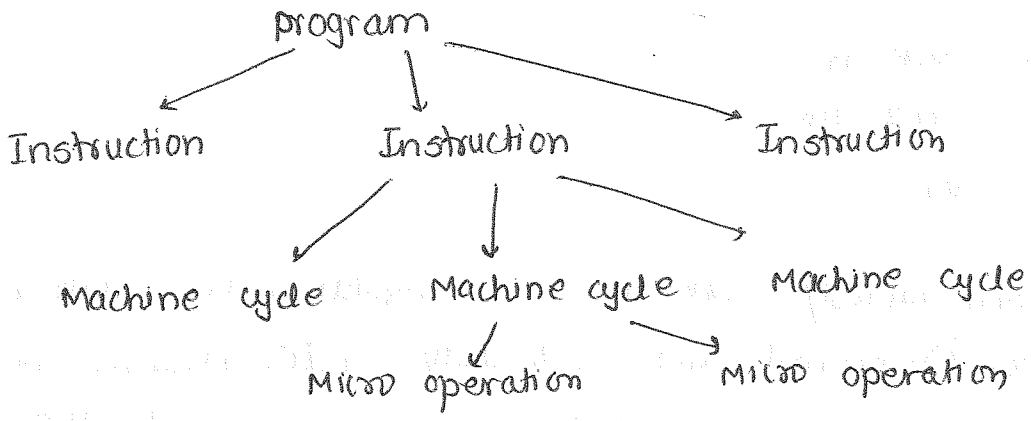


Fig: Program heirarchy

Operation that are performed in a single time cycle (clock pulse) are called microoperation. Each micro-operation transfers data into or out of data bus. Language representing such data flow is known as register transfer language.

Eg: MOV A, B

Machine cycle: Opcode fetch

T_1 : [MAR] ← [PC]

T_2 : [MBR] ← [[MAR]], PC ← PC+1

T_3 : [IR] ← [MBR]

T_4 : decode and execute

Q. MVI A, 20H

Machine cycle: Opcode fetch

T_1 : [MAR] ← [PC]

T_2 : [MBR] ← [[MAR]], PC ← PC+1

T_3 : [IR] ← [MBR]

T_4 : Decode

Machine cycle: Memory read

T_5 : [MAR] ← [PC]

T_6 : [MBR] ← [[MAR]], PC ← PC+1

T_7 : [A] ← [MBR]

Q. LXI H, 2050 H

Machine cycle: Opcode fetch

T_1 : [MAR] ← [PC]

T_2 : [MBR] ← [[MAR]], PC ← PC+1

T_3 : [IR] ← [MBR]

T_4 : Decode

Machine cycle: Memory read

T_5 : [MAR] ← [PC]

T_6 : [MBR] ← [[MAR]], PC ← PC+1

T_7 : [L] ← [MBR]

Machine cycle: Memory read

T_8 : [MAR] ← [PC]

T_9 : [MBR] ← [[MAR]], PC ← PC+1

T_{10} : [H] ← [MBR]

Q. MOV M, A

Machine cycle:

T_1 : [MAR] ← [PC]

T_2 : [MBR] ← [[MAR]], PC ← PC+1

T_3 : [IR] ← [MBR]

T_4 : decode

Machine cycle: Memory write

T_5 : [MAR] ← [HL]

T_6 : [MBR] ← [A]

T_7 : [[MAR]] ← [MBR]

CHAPTER: 3 BUS structure, Memory and I/O Interfacing

Bus structure:

→ Bus refer to common communication path between microprocessor and peripherals.
→ It is actually a group of wires to carry bits.
→ Fundamentally, in any system bus, the lines can be classified into three functional groups: Data bus, Address bus and control bus.

1. Data Bus:

→ The data bus provides a path for monitoring data between the system modules.
→ The data bus consist of a number of separate lines: 8, 16, 32 or 64.
→ The number of lines is refered to as the width of the data bus.
→ Since, each line can carry only one bit at a time, the number of lines determines how many bits can be transmitted at a time.
→ The width of the data bus is a key factor in determining the overall system performance.
→ Data bus is a bi-directional bus for data transfer to and from the microprocessor.
→ If the data bus is 8 bit wide and each instruction is 16 bits long as in the case of Intel 8088 then the CPU must access the memory module twice during each instruction code.

2. Address Bus:

→ It is unidirectional bus from microprocessor to the peripherals.
→ The address bus which consists of a number of separate lines are used to designate the source or destination of the data on data bus.
→ For example, if the CPU requires to read a word (8, 16 or 32) bits of data from memory, it put the address of the desired word on the

address bus.

→ The width of the address bus clearly determines the maximum possible memory capacity of the system.

→ If there are 'N' no. of address lines (wires), the number of addresses that microprocessor can access is 2^N, or maximum addressing capability of such microprocessor is also 2^N.

- A memory location or an I/O device can be represented by each of those addresses.

3. Control Bus:

→ The control bus is comprised of various signal lines that carry synchronization signals.

→ These are not groups of lines like address or data buses, but individual lines that provides a pulse to indicate the microprocessor operation.

→ The microprocessor unit (MPU) generates specific control signals for every operation (such as memory Read or I/O write, etc) it performs.

→ The control signals transmit both command and timing information between the system module.

→ The timing signals indicate the validity of data and address information, where as command signals specify operations to be performed.

→ Some of the control signals are:

Memory write:
- data on the bus to be loaded into the address location.

Memory Read:
- data from the addressed location to be placed on the data bus.

I/O write:
- data on the bus to be output to the addressed I/O port

I/O read:
- data from the addressed I/O port to be placed on the bus

Transfer ACK:

- Indicates that data have been accepted from or placed on the bus.

Bus request:

- Indicates that a module wants to gain control of the bus.

Bus grant:

- Indicate that a requesting module has been granted for the control of bus.

Interupt Request:

- Indicates that an interupt has been pending

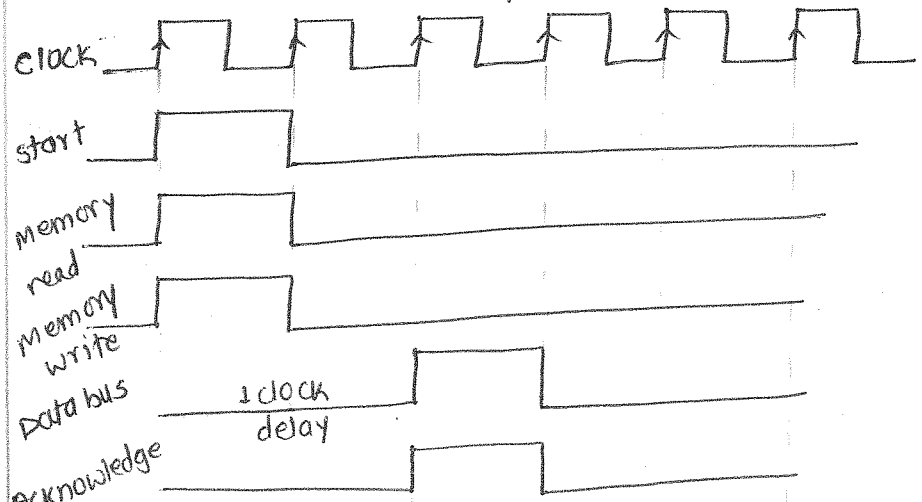
Interupt Ack:

- Indicates that the pending interupt has been recognized.

synchronous and Asynchronous Bus:

1) Synchronous Bus:

→ Occurrence of each event in the bus is determined by clock.
→ A regular sequence of 1's and 0's are transmitted by the clock.
→ A single 1 or 0 is called a clock cycle or bus cycle.
→ All devices on the bus can read clock signal and all events starts at the beginning of clock cycle.
→ All devices operates at a fixed rate.
→ System can not take advantage of device performance i.e. speed is limited to clock period.



→ CPU issues start signal to indicate presence of address and control information on bus.
→ It then issues memory read signal and places memory address on address bus
→ Memory gets the address after one clock delay and it places data on data bus and acknowledge signal to indicate data has been sent.

2. Asynchronous Bus:

→ Occurrence of one event on the bus depends on the occurrence of previous event
→ system speed can be faster than clock speed.

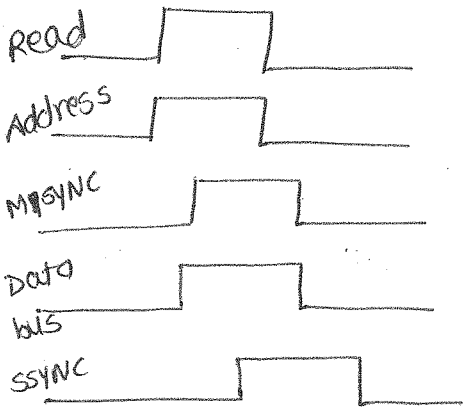


Fig: Asynchronous Bus

→ CPU issuess read signal and places address on address bus.
→ After allowing these two signals to stabilize, it issues MSYNC signal to indicate valid address on bus.
→ Addressed memory responds with valid data and SSYNC signal.

MSYNC = Master synchronous
SSYNC = Slave Synchronous.

Memory:

→ A memory unit is a collection of storage cells together with associated circuits needed to transfer the information in and out of storage. A memory stores binary information in groups of bits called word.
→ A memory is a group of 1's and 0's and may represent a number, an instruction or any other binary coded information. The capacity of memory is usually stated as the total number of bytes that can be stored.
→ A memory word is identified by an address. The 8 bit Microprocessor uses 16 bit address to access memory word. This provides a maximum

of $2^{16} = 65536 = 64$ KB memory address ranging from 0000H to FFFFH and we call it 64 KB Memory.

MEMORY
- Volatile : data loss after power OFF
  - Eg: RAM, CCD [charge coupled device]
- Non volatile : No data loss even after power off.
  - Eg: ROM, FLOPPY disk, HDD etc.

Classification of memory:

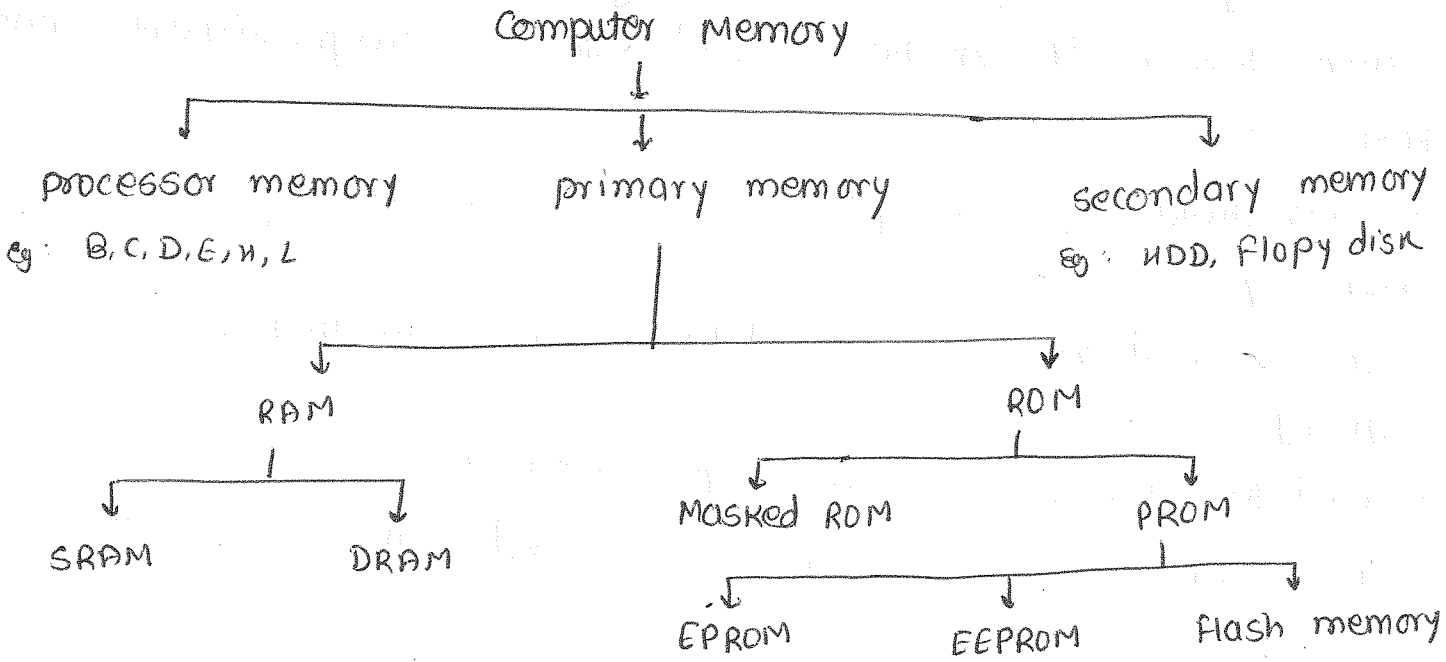


Fig: Classification of memory

processor memory:

→ processor memory refers to set of microprocessor registers. They are used to hold temporary results when computation is in progress.

→ High execution speed so cost is also high. For this reason manufacturer manufacture microprocessor with low processor memory.

→ Eg: Registers (A, B, C, D. etc.)

Secondary Memory:

→ It is the storage area used to hold large data like compiler, data base, OS etc. That are not needed for the processor continuously.

→ Slow devices eg: HDD, Flopy disk etc.

→ Content of secondary memory is 1st copied to primary memory and then executed by the processor.

primary memory:

→ It is the storage area where all the programs are executed. That means all program and data must be within primary memory prior to execution. Usually the size of primary memory is much more larger than processor memory and its execution speed is much slower. It can be divided into two groups: ① ROM ② RAM

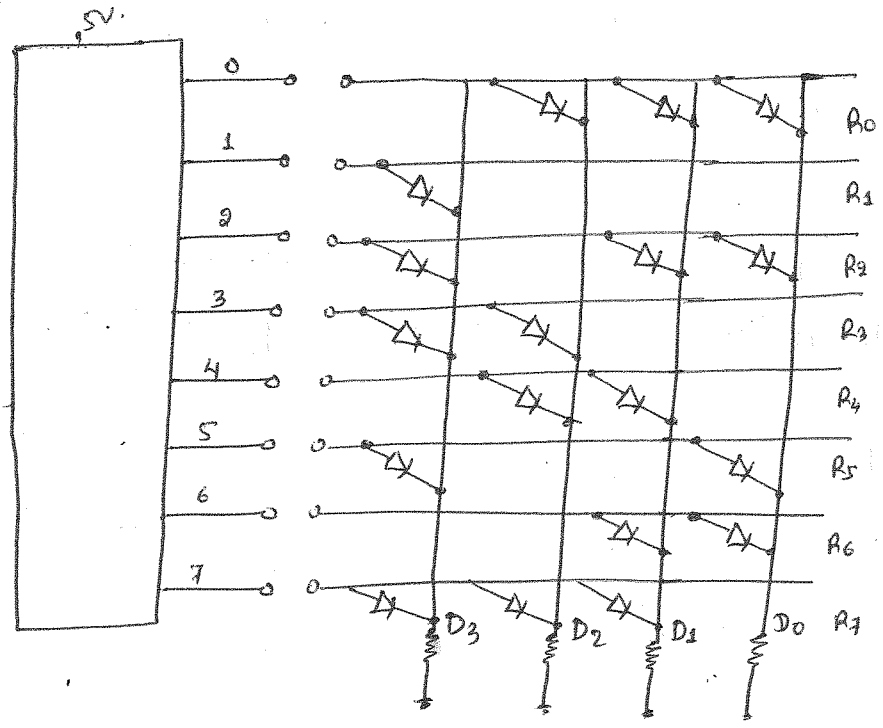
ROM:

→ ROM (Read only memory) is non volatile memory and can be read only.

→ It is used to store the data and program that are not to be altered.

→ Used to store BIOS, Boot strap loader etc.

→ The operation of ROM can be described with following figure.



→ Here each horizontal row is a register or memory location. The R_0 contains three diods, R_1 contains one diode and so on. The output of ROM is:

$$D = D_3\ D_2\ D_1\ D_0$$

→ In switch position 0, high voltage turns 'ON' diodes in R_0 register; all other diodes are 'OFF'. Thus the word stored at the memory 0 is,

$$D = 0111$$

→ Similarly at memory location 1,

$$D = 1000$$

→ Similarly, data are in other registers are stored by the manufacturer during the time of IC manufacture.

→ The data are permanently stored once that diodes are wired in place.

→ Transistors and MOSFETs can also be used in place of diode, but the basic principle is same.

→ ROM can be classified as:

1. Masked ROM:

→ In this ROM, a bit pattern is permanently stored and user cannot change it.

→ Marking is done during manufacturing and the manufacturers are equipped with this.

2. PROM (Programmable ROM):

→ This memory has nichrome or polysilicon wires arranged in a matrix.

→ These wires can be functionally viewed as diode or fuse.

→ The memory can be programmed by the programmer with a special PROM programmer that selectively burns the fuses on applying high current according to the bit patterns to be stored.

→ The process is known as burning of ROM and the information stored is permanent.

3) EPROM [Erasable PROM]:

→ EPROM uses MOSFETS

→ Data is stored with PROM programmer

→ Later data can be erased with uv light

→ The light passes through the window on the chip. The effect is to wipe out stored contents. i.e. EPROM is uv erasable and electrically programmable.
→ Once the chip is programmed, it is covered with opaque tape to avoid accidental erasing.

ii) EEPROM [Electrically Erasable PROM]:

→ Non volatile like EPROM but does not require UV-light to erase.
→ It can be completely erased or have certain bytes changed using electrical pulses.
→ Writing to EEPROM is much slower than RAM thus it can not be used in high speed circuits.

ii) FLASH Memory:

→ Modified EEPROM
→ Only difference is erasure procedure.
→ EEPROM can be erased at register level but Flash memory must be erased either entirely or at the sector (block) level.

② Random Access memory or Read write Memory:

→ It is a volatile memory and is used to store programs and data for the immediate use of the processor.
→ Data can be read from or written into the selected address at any sequence.
→ Two types:

ⓐ SRAM (Static RAM):

→ made up of flipflops and stores bits as voltage
→ Each cell has 6 transistors
→ Expensive and power consuming
→ called static because data does not need a constant update
→ Fasthy but physically bulky.

ⓑ Dynamic RAM (DRAM):

→ uses capacitor to store bits
→ bits are stored as charge
→ IF charge exists in capacitor, it is interpreted as 1 otherwise 0.
→ There is a chance of charge leakage from the capacitor thus needs constant refresh.
→ faster than ROM but slower than SRAM
→ High packing density
→ low cost
→ requires additional circuitry for refresh.

MEMORY

depending upon access

Random access
→ data can be read or written in any order.
→ Initially RAM was only considered random access.
→ Nowadays both RAM and ROM are random accesses.

sequential access
→ We need to access every memory location between the present memory location and the location we want to access.
→ One can't jumps to desired location
eg: reel of magnetic tape

Semiconductor technology

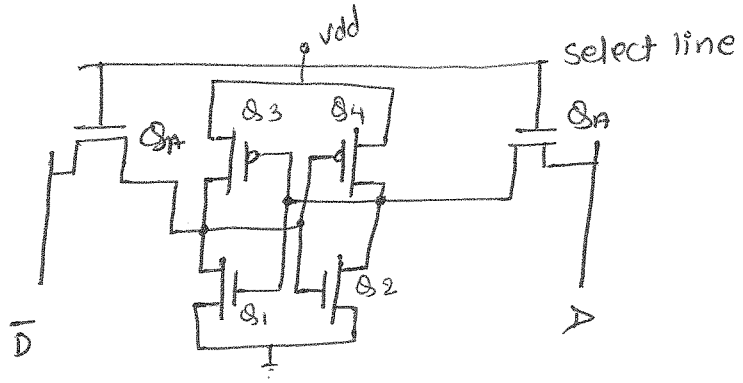
BJT
- fast access time
→ large chip area
→ power consuming
→ expensive

MOSFET
→ simple and economical
→ less chip area
→ consume less power.

MOS RAM cell:

① Static RAM cell:

→ Figure shows NMOS static Memory cell which consists of 6 NMOS transistor.
→ Transistor Q_A is called access transistor and used to access the data during read and write operation.



→ The transistor Q_3 and Q_4 are pull up load transistors and their cross connection with Q_2 and Q_1 respectively forms a flip flop.

→ The D and $\bar{D}$ are data lines. There is external driving circuitry which forces D and $\bar{D}$ to opposite values. When reading the bit from cell the D and $\bar{D}$ serves as o/p lines, while writing to cell, they serve as i/p.

→ Before reading or writing on cell, the cell must be selected which is done by making select line high.

Write on cell:

→ Access transistor Q_A is turned on via row select line [high]

→ The data lines are forced with D high and $\bar{D}$ low.

→ Q_1 is turned on and Q_2 off

→ When forcing signals are removed, the flip flop will continue to hold bits.

→ The forced state is thus self sustaining and stable

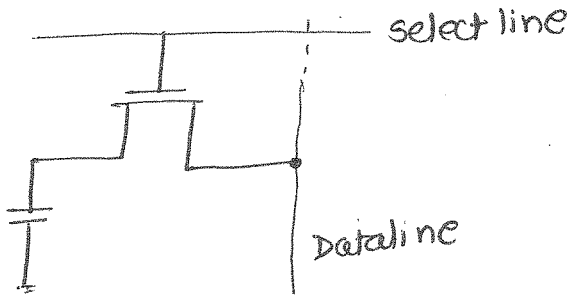
→ Similar stable state exists when, D = low and $\bar{D}$ = high

Reading from cell:

→ Q_A are turned on via row select line (high)

→ The stored bits appears on D and $\bar{D}$

2) Dynamic RAM cell:



→ The capacitor is to store the bit

→ Transistor acts as switch.

Write operation:

→ The data line is high (to store 1)

→ The select line is high.

- MOSFET is turned on and capacitor is charged
- When data and slection line goes low, capacitor retains its charge.

Read operation:

→ Select line is high

→ Data on capacitor is received via data line.

Internal structure of memory:

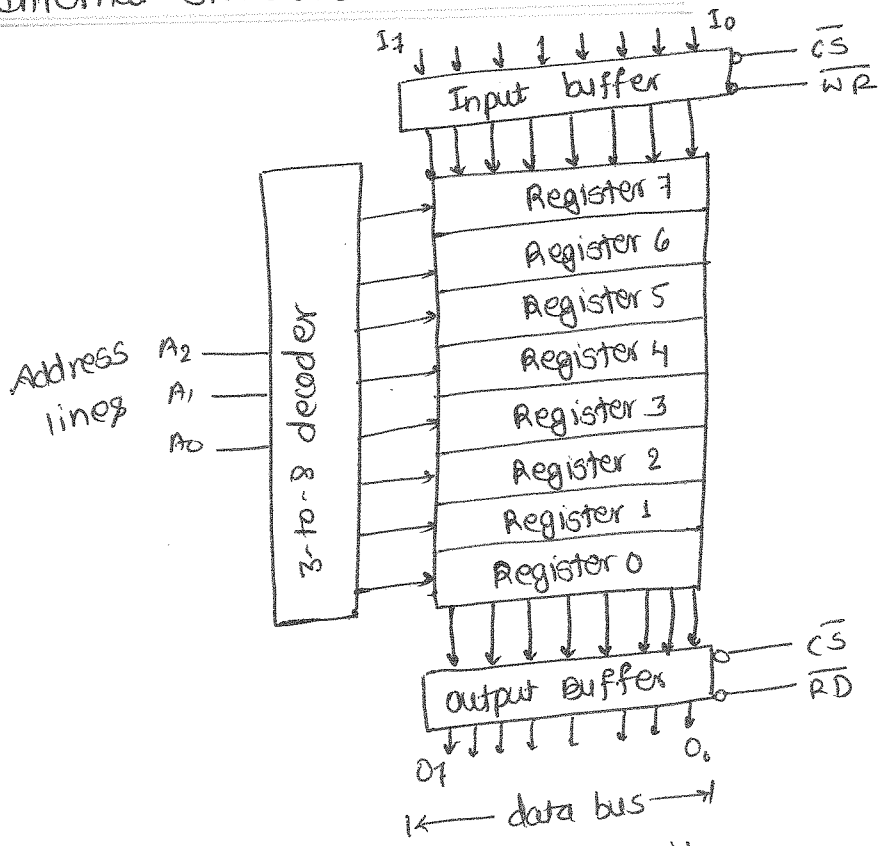


fig: 8x8 memory unit

→ Internally a memory consists of address decoder, input buffer, output buffer, registers with address lines, data lines and $\overline{RD}$, $\overline{WR}$, $\overline{CS}$ control lines.

→ The number of address lines will be determined by the memory capability.

→ The number of data lines will be determined by memory size.

→ for $2^n \times m$ memory capacity, the number of address lines = n and the number of data lines = m.

→ Let's consider the 8x8 memory device with 8-registers, 3 to 8 decoder, an input buffer and an output buffer. The device will have 3 address lines and 8 data lines. It will also have control lines $\overline{RD}$, $\overline{WR}$ and $\overline{CS}$.

- To write an 8-bit word, the microprocessor places the register address on the address line.
  - The decoder decodes the address and selects the register.
  - Then the microprocessor places the data on the data bus and sends the active low $\overline{WR}$ control signal.
  - The control signal enables the input buffer and data are placed in the selected register.
- To read from memory, the process is similar to write operation except that output buffer is enabled with $\overline{RD}$ signal.
- The remaining address lines of the microprocessor address bus are used to select the chip ($\overline{CS}$).

Basic concept in memory interfacing:

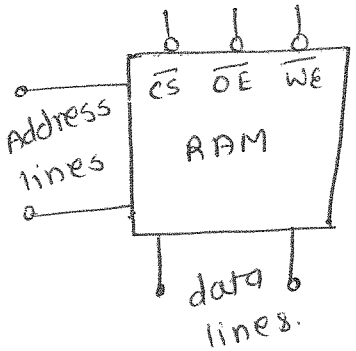
→ The primary function of memory interfacing is that the microprocessor should be able to read from and write into a given register of a memory chip.

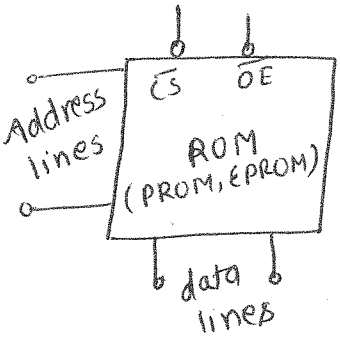
→ To perform this operation, the microprocessor should

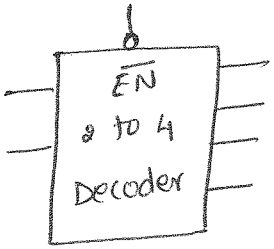
① be able to select the chip.

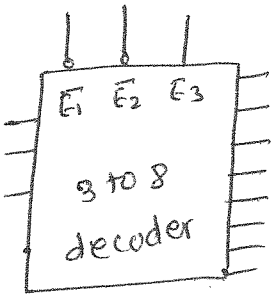
② Identify the register

③ Enable the appropriate buffer.









Here, $\overline{OE}$ → Output enable buffer

$\overline{WE}$ → write enable buffer

$\overline{CS}$ → chip selection enable

Address Decoding:

- Address decoding is a process which identifies a register for a given address i.e. to identify only one location for a given address.

→ In other words, the method used to select the correct location on the correct chip is called address decoding and we use the address bus to accomplish the selection.

→ The address of a device is determined from the way in which the address lines from the processor are used to derive a special device selection signal known as chip select ($\overline{CS}$).

→ There are 2 common methods for mapping address of these devices:

① I/O mapped I/O ② Memory-mapped I/O

Difference between I/O mapped I/O and Memory-mapped I/O:

I/O mapped I/O	Memory mapped I/O
1. 8 bit address for I/O devices for 8085. Eg: OUT 30H	1. 16 bit address for I/O devices for 8085. Eg: STA C000H
2. $2^8 = 256$ I/O device can be connected.	2. $2^{16} = 65536$ I/O devices can be connected.
3. Less hardware is required to decode 8 bit address.	3. More hardware required to decode the 16 bit address.
4. Control signals used are: $\overline{IOR}$ and $\overline{IOW}$	4. Control signal used are $\overline{MEMR}$ and $\overline{MEMW}$
5. The execution speed is always same since IN and OUT has 10 T-state	5. Execution speed depends upon the instructions used.
6. Data transfer between I/O and accumulator	6. Data transfer can be between any resig register memory locations & I/O.
7. The memory map is independent of I/O map	7. The memory map is shared between memory and I/O
8. Arithmetic and logical operation can't be performed directly with I/O data.	8. Arithmetic and logical operation can be directly performed with I/O data.

Address decoding types:

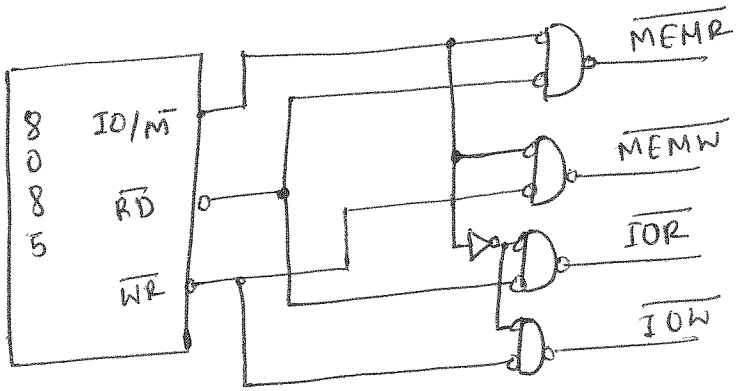
① Absolute:

In this address decoding technique, all address lines that are not used for memory register must be decoded so, chip select ($\overline{CS}$) can be asserted by only one address.

② Partial:

In this address decoding technique, some address lines left don't care. This technique reduces hardwork but generates multiple addresses resulting a foldback memory space.

Generation of control signals:



Note: 

Memory interfacing examples:

Q# Interface 2K×8 ROM with 8085 up at starting address 0800H

Sol^n: Here, size of memory = 2K×8

To find no of address lines,

$$2^x = 2 \times 1024 = 2 \times 2^{10} = 2^{11}$$

$$\therefore x = 11.$$

→ We need 11 address lines.

Thus, out of 16 address lines available, we use 11 as address lines for memory and rest of lines i.e. 5 are used to generate chip enable signal.

→ We have to interface only one memory, so we do not use decoder.

	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	
start. address:	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	= 0800H
						1	1	1	1	1	1	1	1	1	1	1	= 0FFFH
end. address	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	

Thus, $A_0 - A_{10}$ are used for memory address and $A_{11} - A_{15}$ are used to derive $\overline{CS}$ signal.

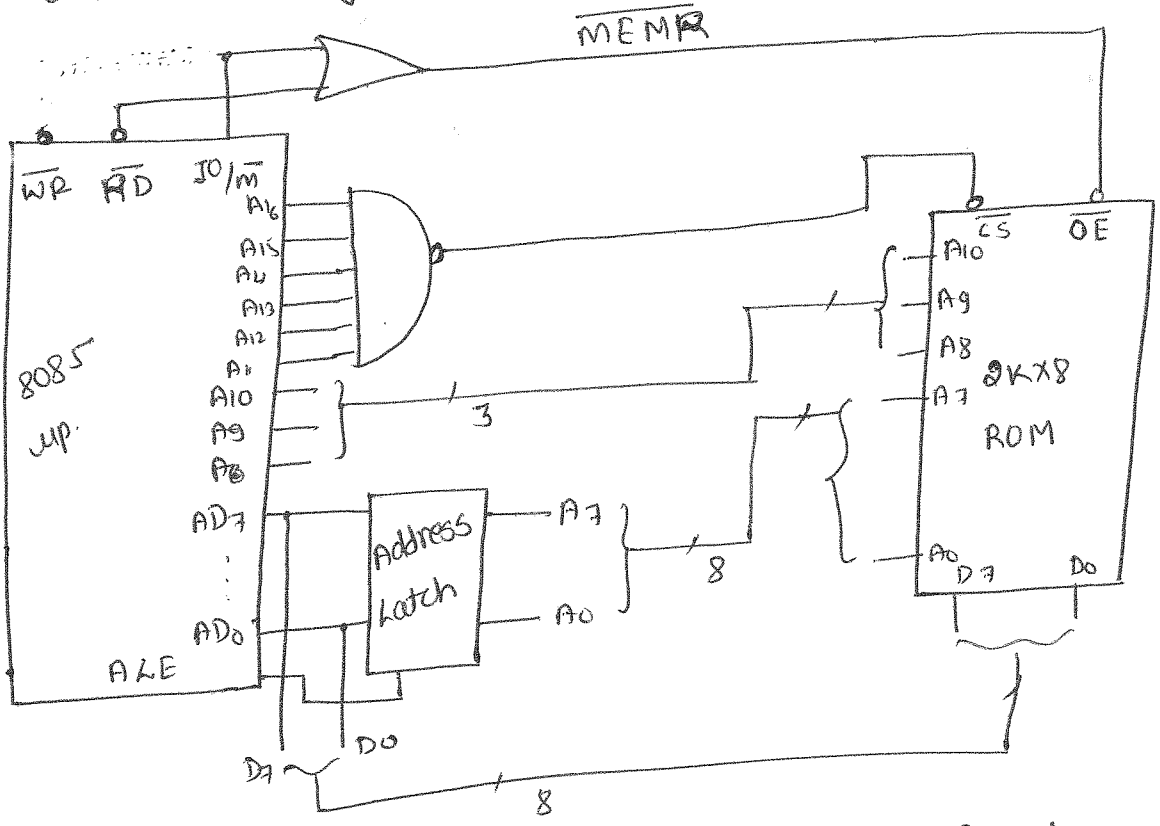


Fig: Circuit interfacing 2KB RAM with 8085 µp.

Q# Design an address decoding circuit to interface two RAM chips/blocks of 64 bytes each at address 2200H and 3300H resp.

Soln: Size of RAM1 = 64 bytes

$\Rightarrow 2^6 = 64$

∴ no. of address lines = 6

Size of RAM2 = 64 bytes

so, no. of address lines = 6

Address decoding:

	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	
For RAM1																	
st. address	0	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	= 2200H
											1	1	1	1	1	1	-
ending address	0	0	1	0	0	0	1	0	0	0	1	1	1	1	1	1	= 223FH
For RAM2																	
st. address	0	0	1	1	0	0	1	1	0	0	0	0	0	0	0	0	= 3300H
											1	1	1	1	1	1	
ending address	0	0	1	1	0	0	1	1	0	0	1	1	1	1	1	1	= 333FH

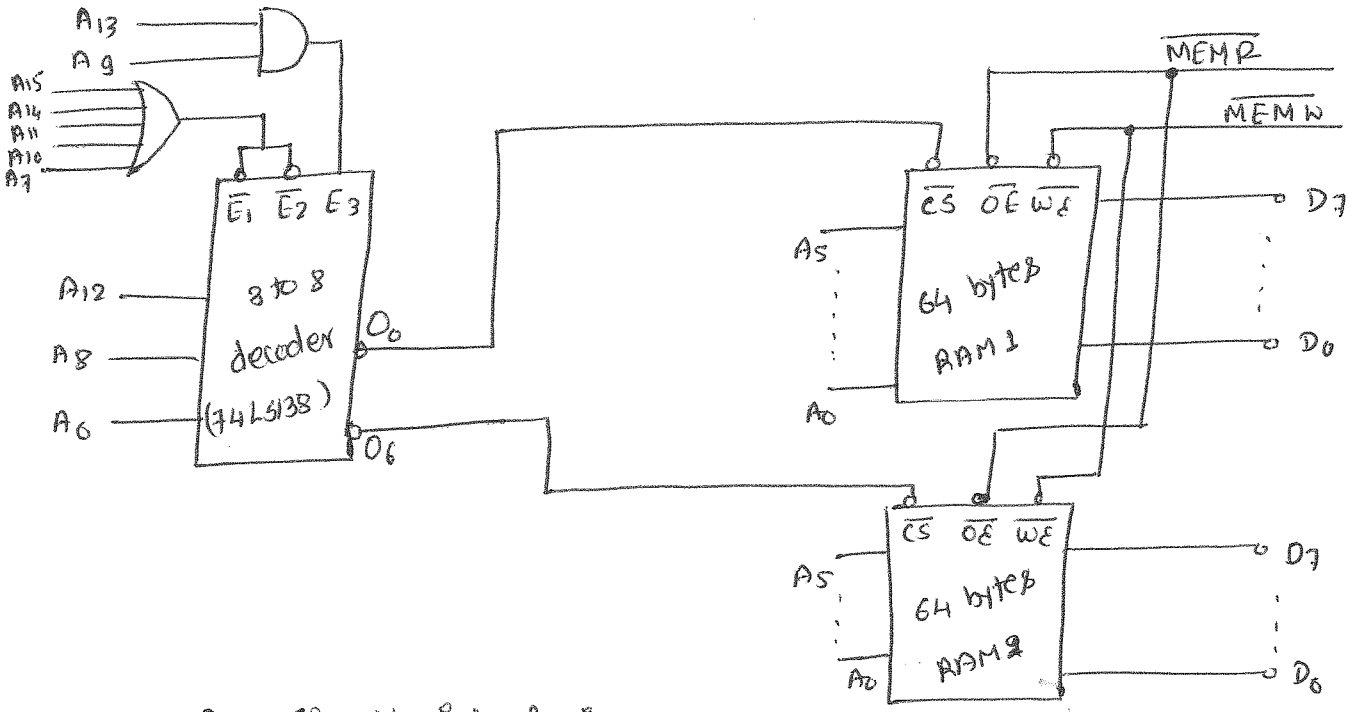


Fig: Circuit interfacing two 64 bytes RAM with 8085 uP.

Q3# Interface three 2KB RAM with 8085 microprocessor.

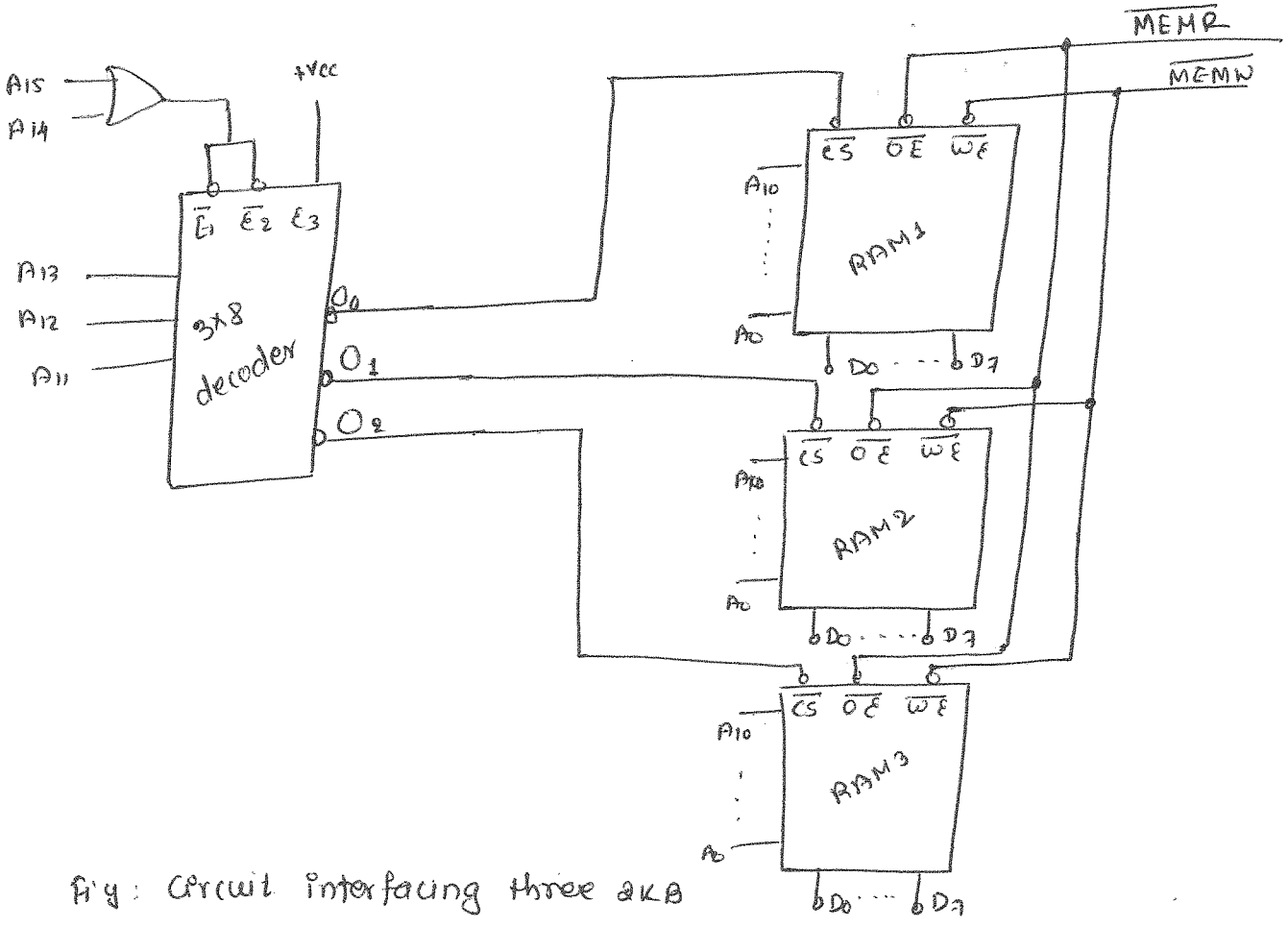


Fig: Circuit interfacing three 2KB RAM with 8085 uP.

Here, Size of RAM = 2KB

$\Rightarrow 2 \times 8^{10} = 2KB = 2^{11}$

∴ Total address lines = 11 for all three RAM.

Address decoding

	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	
For RAM 1																	
St. address	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	= 0000H
						1	1	1	1	1	1	1	1	1	1	1	
Ending address	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	= 07FFH
For RAM 2																	
St. address	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	= 0800H
						1	1	1	1	1	1	1	1	1	1	1	
Ending address	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	= 0FFFH
For RAM 3																	
St. address	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	= 1000H
						1	1	1	1	1	1	1	1	1	1	1	
Ending address	0	0	0	1	0	1	1	1	1	1	1	1	1	1	1	1	= 17FFH.

Q4# Interface 8085 microprocessor with 8 LEDs and 8 switch for memory mapped I/O.

NOTE: In peripheral-mapped I/O, a device is identified with an 8-bit address and enabled by I/O-related control signals. In memory-mapped I/O, a device is identified with a 16-bit address and enabled by memory related control signals.

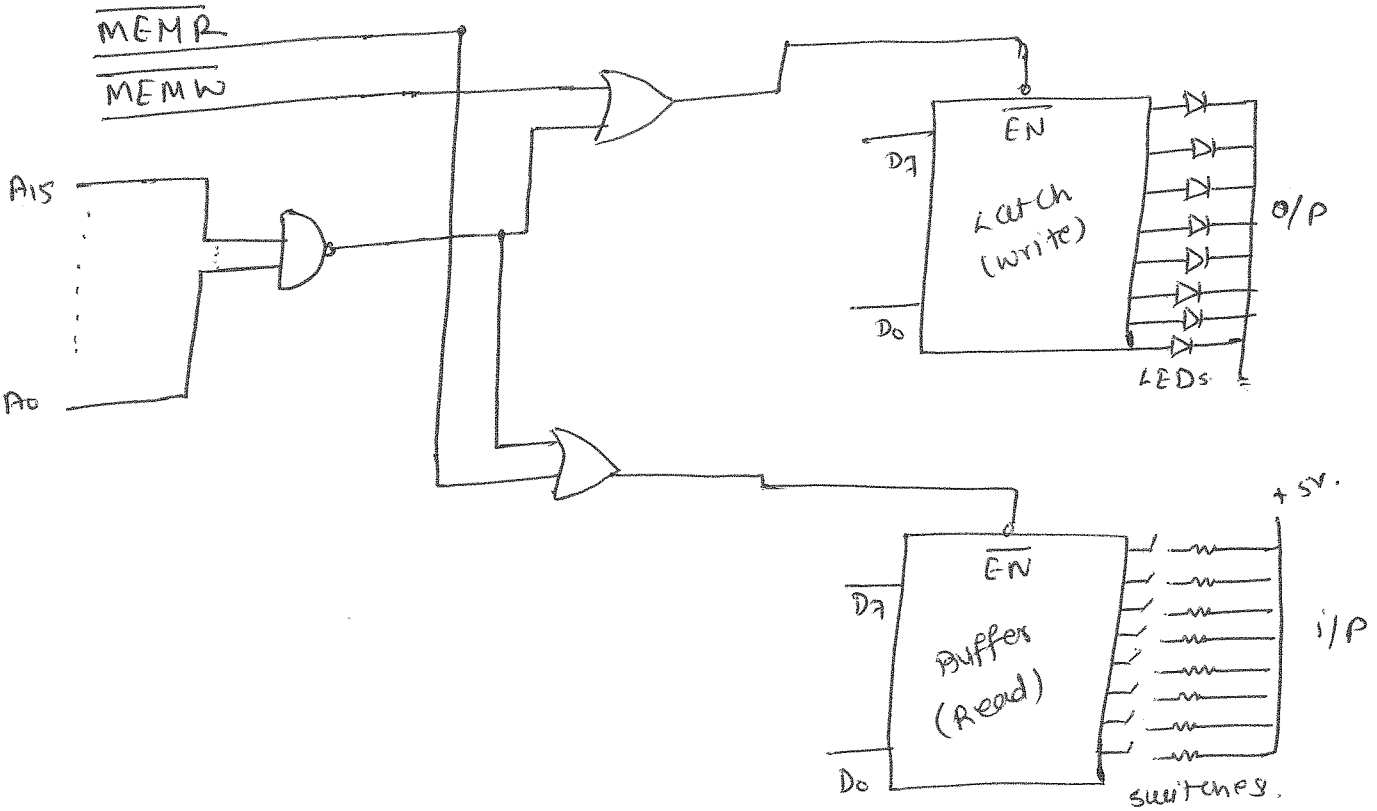


Fig: Interfacing 8085 up with 8 LEDs and 8 switches for memory mapped I/O.

→ For memory mapping,

port address for latch i.e. 8 LEDS is → A_{15} A_{14} . . . . . A_0

1 1 . . . 1 = FFFFH

and,

port address for 8 switches is → A_5 A_{14} . . . . . A_0

1 1 1 = FFFFH

so, there is not address range in this case.

-Value at $\overline{MEMW}$ and $\overline{MEMR}$ helps to determine whether Read or write operation is performed and thus Latch or buffer is selected.

Qs.# Interface 1KB RAM, 8 LEDS and 8 switches for all memory mapped I/O configuration:

Soln:

Address decoding table

For Buffer:

	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	
Address is,	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	= F000H

For Latch :

Address is,	1	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	= F400H

For RAM

St. address	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	= F800H
							1	1	1	1	1	1	1	1	1	1	
Ending address	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	= FBFFH

Here, size of RAM = 1KB

$\Rightarrow 2^{10}$ = 1KB

Thus address lines = 10.

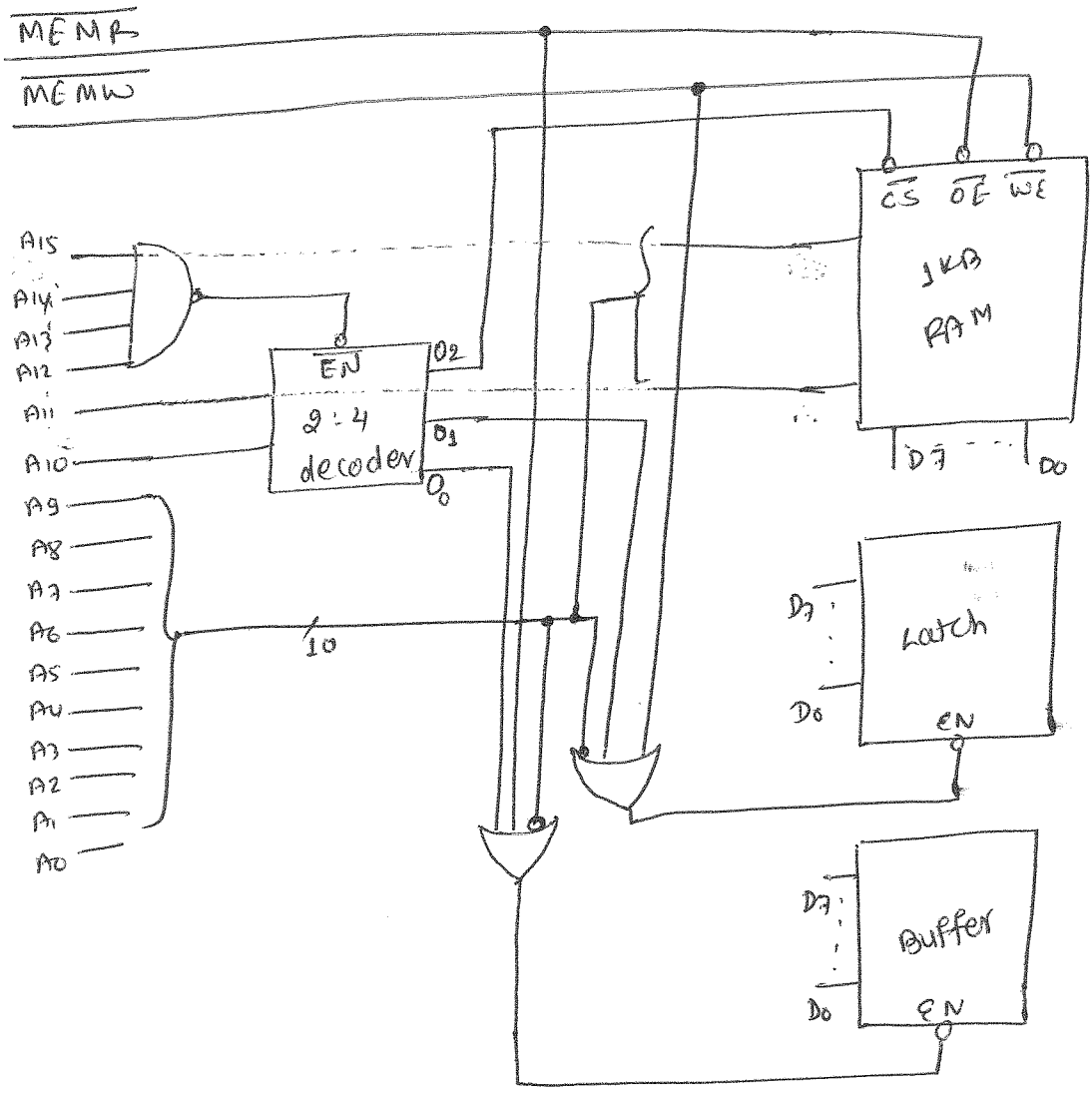


Fig: Interfacing 1KB RAM, 8 LEDS and 8 switches for all memory mapped I/O configuration.

Interrupts:

programmed I/O

programmed I/O (PIO) refers to data transfers initiated by a CPU under driver software control to access registers or memory on a device.

The CPU issues a command then waits for I/O operations to be complete. As the CPU is faster than the I/O module, the problem with programmed I/O is that the CPU has to wait a long time for the I/O module of concern to be ready for either reception or transmission of data. The CPU, while waiting must repeatedly check the status of the I/O module, and this process is known as polling. As a result, the level of the performance of the entire system is severely degraded.

Programmed I/O basically works in these ways:

- CPU requests I/O operation
- I/O module performs operation
- I/O module sets status bits
- CPU checks status bits periodically
- I/O module does not inform CPU directly
- I/O module does not interrupt CPU
- CPU may wait or come back later

Interrupt driven I/O:

The CPU issues commands to the I/O module then proceeds with its normal work until interrupted by I/O device on completion of its work.

For input, the device interrupts the CPU when new data has arrived and is ready to be retrieved by the system processor. The actual actions to perform depend on whether the device uses I/O ports, memory mapping.

For output, the device delivers an interrupt either when it is ready to accept new data or to acknowledge a successful data transfer. Memory-mapped and DMA-capable devices usually generate interrupts to tell the system they are done with the buffer.

Although Interrupt relieves the CPU of having to wait for the devices, but it is still inefficient in data transfer of large amount because the CPU has to transfer the data word by word between I/O module and memory.

The Basic operations of Interrupt:

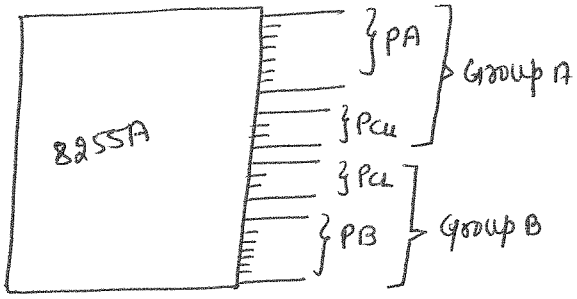
- CPU issues read command
- I/O module gets data from peripheral whilst CPU does other work
- I/O module interrupts CPU
- CPU requests data
- I/O module transfer data.

8255 A (programmable peripheral interface):

The 8255A is a widely used programmable parallel I/O device. It is flexible, versatile and economical when multiple I/O ports are required, but somewhat complex. It is compatible with almost all microprocessor.

The 8255A has 24 I/O pins [40 pins in total] which are grouped into two 8-bit ports: port A and port B. The remaining 8-bits are collectively known as port C out of which each bit of port C can be used as individual bit or grouped into two 4-bit ports or used as a single 8-bit port.

The function of these ports are defined by writing a control word line into the control register.



Block diagram of 8255A:

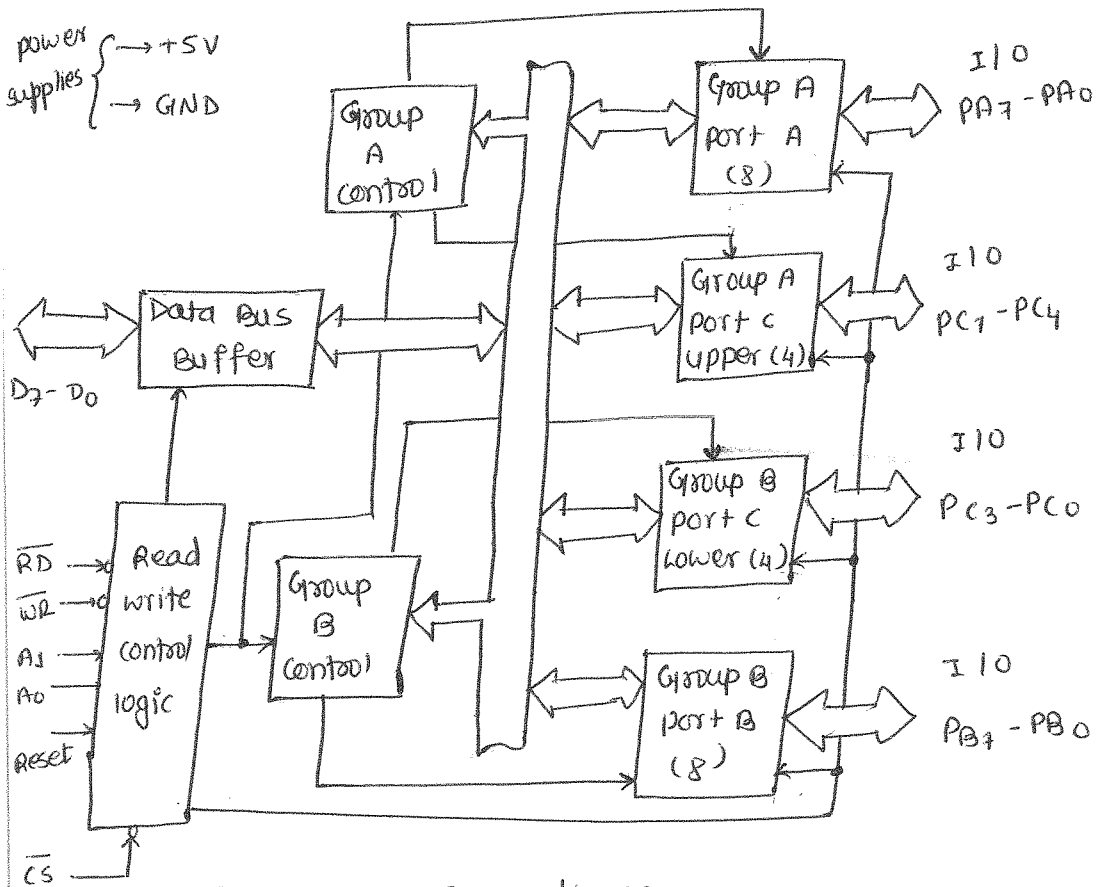


Fig: 8255A Block diagram.

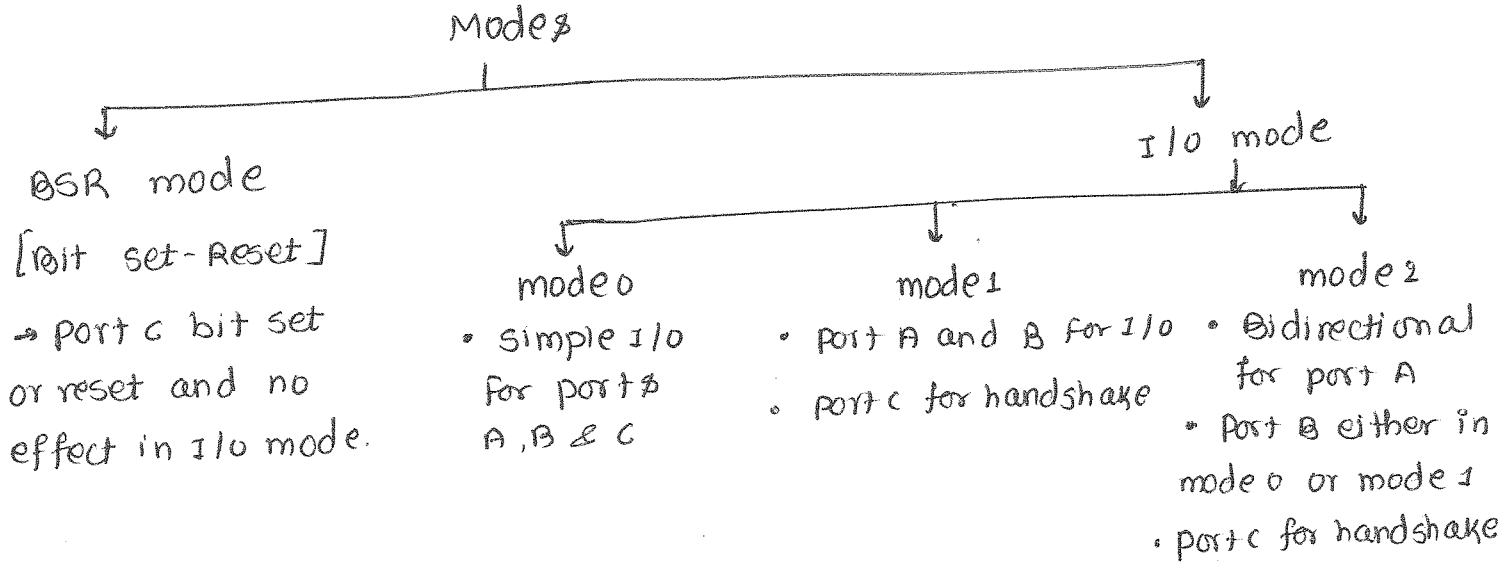
$\overline{RD}$: When this signal is low, the micro-processor unit reads data from a select I/O port of the 8255A

$\overline{WR}$: When low, the MPU writes data into a selected I/O port of 8255A

Reset: When high, it clears the control register and sets all ports into the input mode.

$\overline{CS}$, A_0, A_1: $\overline{CS}$ is the master chip select and A_0 and A_1 specify one of the I/O ports or control register is given below:

$\overline{CS}$	A_1	A_0	selected
0	0	0	port A
0	0	1	port B
0	1	0	port C
0	1	1	control Register (R)
1	x	x	None



Control word:

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
IO/$\overline{BSR}$	mode select ←	→	PA	PCU	mode select	PB	PCL

P_7: IO/$\overline{BSR}$ = 0; BSR mode is selected

= 1; IO mode is selected

D_6	D_5	mode	
0	0	0	} for group A
0	1	1	
1	x	2	

$D_4 = 0$: port A in o/p mode
$= 1$: port A in i/p mode

$D_3 = 0$: port C_U in o/p mode
1 : port C_U in i/p mode

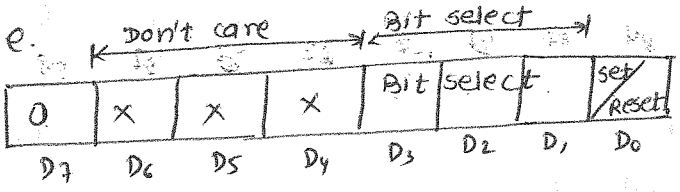
$D_2 = 0$: mode 0 } for group B
1 : mode 1 }

$D_1 = 0$: PB in o/p mode
1 : PB in i/p mode

$D_0 = 0$: PC_L in o/p mode
1 : PC_L in i/p mode

BSR mode:

The BSR mode is only used to set/Reset, one of the 8 bits of portc at a time by writing a BSR control word into the control register. Before using BSR mode we must use I/O mode control word to configure portc in o/p mode.



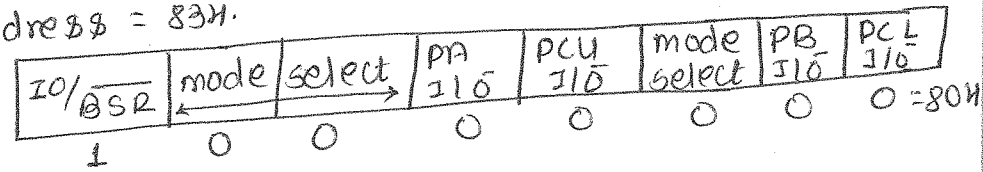
D_3	D_2	D_1	portc bit-select
0	0	0	PC_0
0	0	1	PC_1
0	1	0	PC_2
0	1	1	PC_3
1	0	0	PC_4
1	0	1	PC_5
1	1	0	PC_6
1	1	1	PC_7

Q.# Write a BSR control word sub-routine to set bits PC_7 and PC_3 and reset them after 10ms (Assume control register address = 83H and delay subroutine is available)

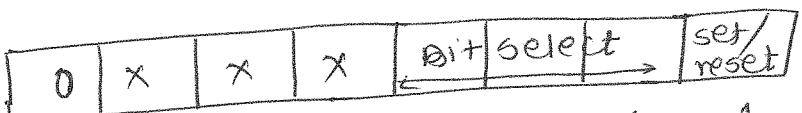
Soln:

Control Register Address = 83H.

IO mode CW =



BSR control words:



To set PC_7 0 0 0 0 1 1 1 1 = 0F H

To Reset PC_7 0 0 0 0 1 1 1 0 = 0E H

To set PC_3 0 0 0 0 0 1 1 1 = 07 H

To Reset PC_3 0 0 0 0 0 1 1 0 = 06H

program:

```
MVI A, 80    ; load CW to CR
OUT 83H
MVI A, 0F H  ; Load byte in accumulator to set PC7
OUT 83H      ; set PC7 = 1
MVI A,07 H   ; Load byte in accumulator to set
OUT 83H      ; set PC3 = 1
CALL delay   ; this is a 10 ms delay
MVI A,06 H   ; Load accumulator with the byte to reset PC3
OUT 83 H     ; PC3 reset
MVI A, 0E    ; Load accumulator with the byte to reset PC7
OUT 83H      ; PC7 reset
HLT.
```

I/O mode 0:

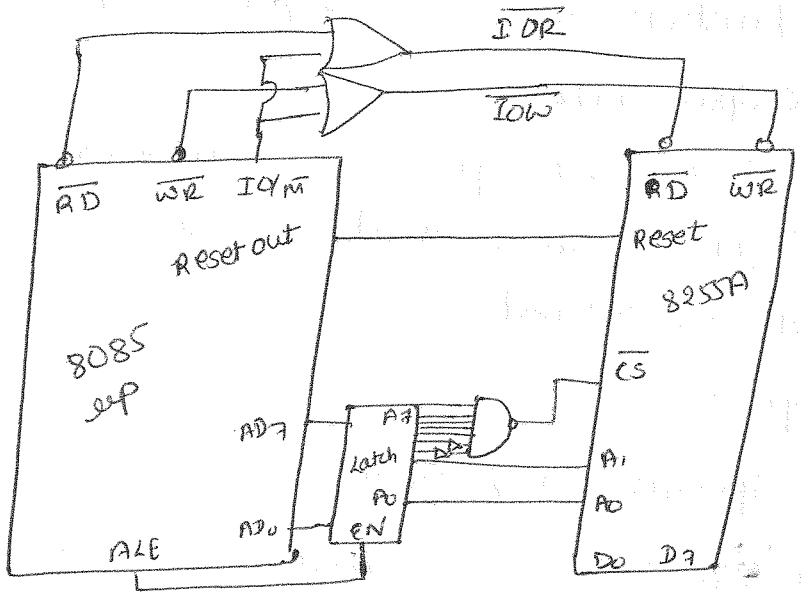
In this mode, PA and PB are used as two simple I/O ports and port C as two 4-bit simple I/O port, where each port or half of port can be programmed as an input or output port individually.

The input and output feature of mode 0 are:

1. Output ports are latched [the bits are put into a storage register which holds its output constant even if the inputs chang after being latched].
2. Input ports are buffered, not latched [. the cpu only has to read their current values, then store the data in a cpu register or memory if it needs to be referenced at a later time.]
3. ports do not have handshake or interrupt capability.

8 Interface a 8255 A PPI with 8085 up such that its port address starts at FOH. Configure PA and PCU as output ports. And the remaining port as input port. Then WAP to read the I/O ports and display the reading at o/p ports.

soln:



I/O control word:

PCU

IO/$\overline{BSR}$	Mode select	PA I/O	Mode select	PB I/O	PCL I/O
1	0 0	0	0	0	1 1 = 83 H.

	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0	
PA =	1	1	1	1	0	0	0	0	= F0H
PB =	1	1	1	1	0	0	0	1	= F1H
PC =	1	1	1	1	0	0	1	0	= F2H
CR =	1	1	1	1	0	0	1	1	= F3H

program:

```
up: MVI A, 83 H     ; load cw to CR
    OUT F3 H
    IN F1 H         ; input from port B
    OUT F0 H        ; load port B to port A
    IN F2 H         ; i/p port C
    RLC
    RLC
    RLC
    RLC
    OUT F2 H        ; load port C to port C
    JMP up
```

I/O mode 1 (Input/output with handshake)

In mode 1, handshake signals are exchanged between the MPU and peripherals before data transfer. The feature of this mode include the following:

→ 2 ports A and B function as 8-bit i/o port. They can configured either input or output ports.

→ Each port uses three lines from port C is handshake signals. Remaining two lines of port C can be used for I/O functions.

→ Input and output data are latched

→ Interupt logic is supported.

In 8255A, the specific lines from port C used for handshake signals vary according to I/O functions of a port. Therefore input and output functions in mode 1 are discussed separately.

Input handshake signals:

$I/\overline{O}$	$I/\overline{O}$	IBF_A	$INTE_A$	$INTR_A$	$INTE_B$	IBF_B	$INTR_B$

fig: mode 1 input status word.

1. $\overline{STB}$ (strobe input):

This signal is active low and is generated by a peripheral device to indicate that it has transmitted a byte of data.

2. IBF (Input Buffer Full)

This signal is an acknowledgement by the 8255A to indicate that the i/p latch has received the data byte. This is reset when the MPU reads the data.

3. INTR (Interrupt request):

This is an output signal that may be used to interrupt the MPU. This signal is generated if $\overline{STB}$, IBF and INTE are all at logic 1. This is reset by the falling edge of $\overline{RD}$ signal.

4. INTE (Interrupt enable):

This is an internal flipflop used to enable or disable the generation of INTR signal. The two flipflops $INTE_A$ and $INTE_B$ are set/Reset using the BSR mode. The $INTE_A$ is enabled or disabled through PC_4 and $INTE_B$ is enabled or disabled through PC_2.

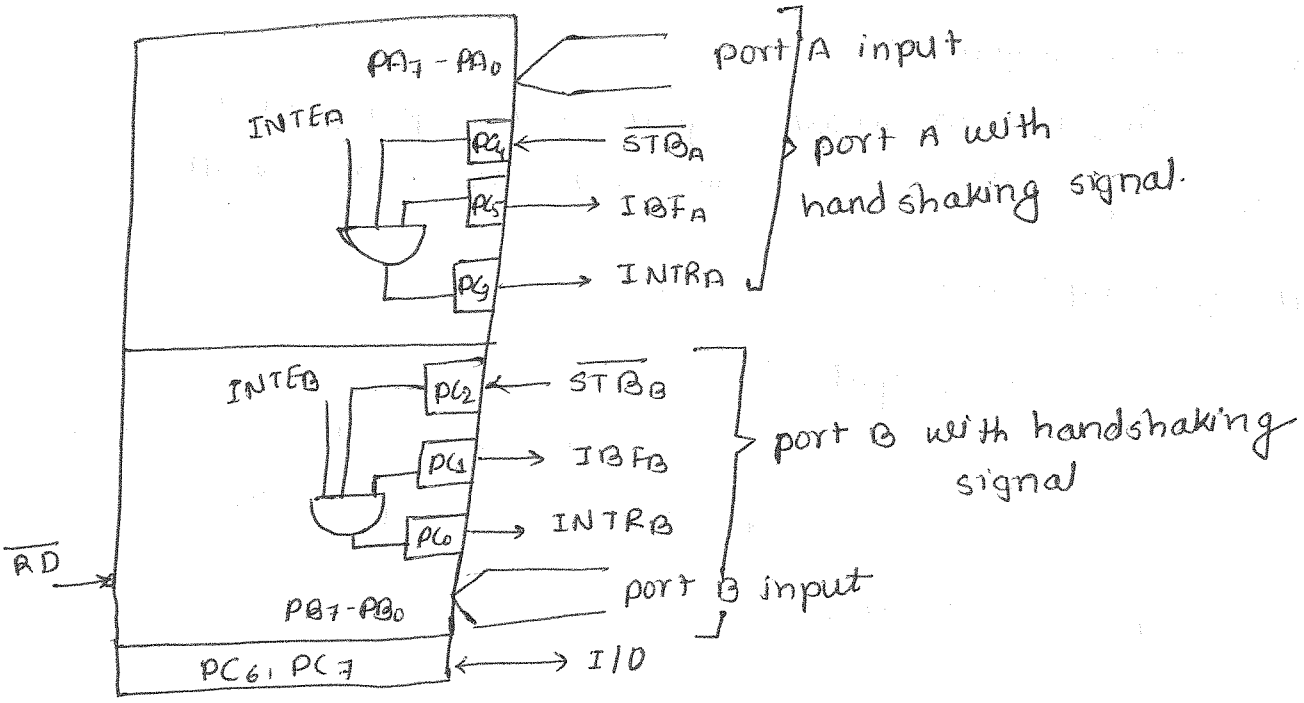


Fig - 8255A mode 1 input configuration.

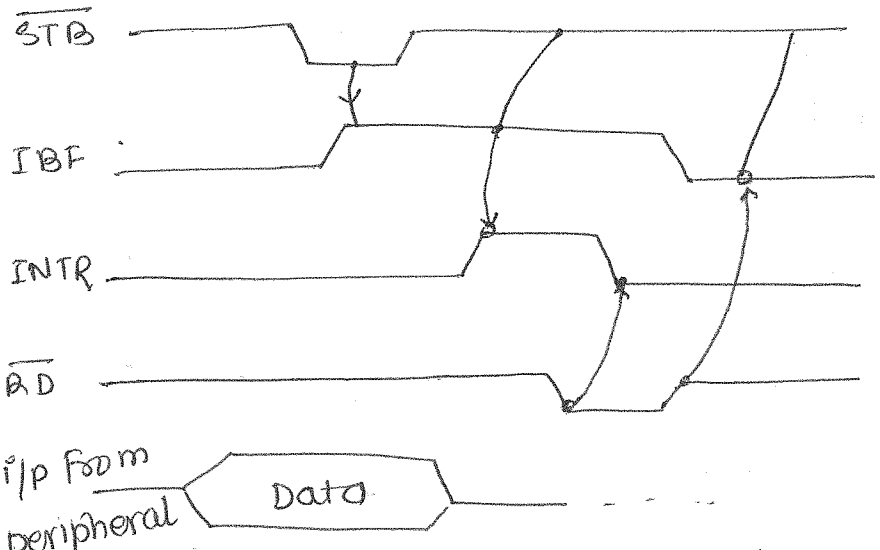


Fig: mode 1 i/p timing diagram.

Output handshake signals:

$\overline{OBF_A}$	$INTE_A$	I/O	I/O	$INTR_A$	$INTE_B$	$\overline{OBF_B}$	$INTR_B$

Fig: mode 1 output status word.

1. $\overline{OBF}$ [Output buffer full]:

This is an output signal that goes low when the MPU writes data into the output latch of 8255A. This signal indicates to an output peripheral that new data are ready to be read. It goes high again after the 8255A receives an $\overline{ACK}$ from the peripheral.

2. $\overline{ACK}$ [acknowledgement]:

This is an input signal from a peripheral that goes low when the peripheral receives the data from the 8255A.

3. INTR [interrupt request]:

This is an output signal and it is set by the rising edge of $\overline{ACK}$ signal. This signal can be used to interrupt the MPU to request the next datatype for output. The INTR is set when $\overline{OBF}$, $\overline{ACK}$ and INTE are all at logic 1 and reset by falling edge of $\overline{WR}$.

4. INTE [interrupt enable]:

This is an internal flip-flop and needs to be set to generate the INTR signal. $INTE_A$ and $INTE_B$ are controlled by PC6 and PC2 respectively.

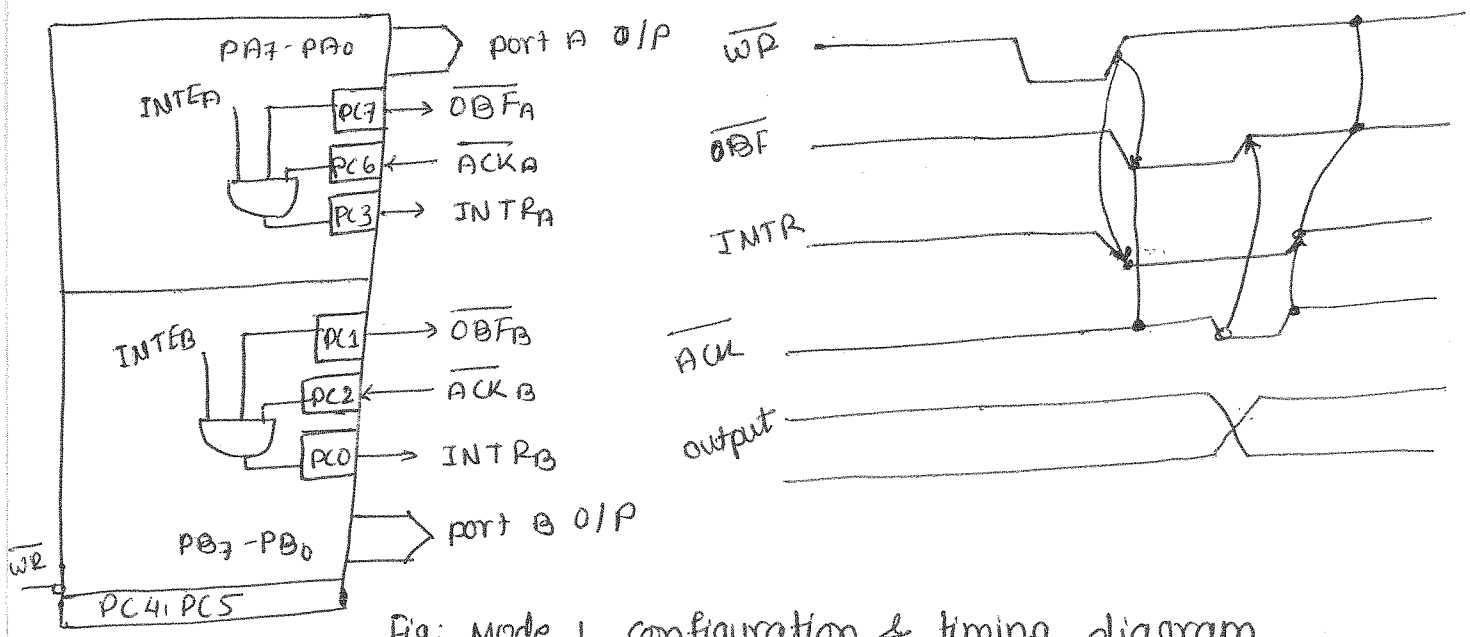


Fig: Mode 1 configuration & timing diagram

I/O mode 2 (bidirectional data transfer):

This mode is used primarily in application such as data transfer between two computers or floppy disk controller interface. In this mode port A can be used as bi-directional port and port B either in mode 0 or mode 1. Port A uses 5 signals from port C as handshake signals with remaining three being used as simple I/O port or handshake for port B.

Data transfer between master and slave MPU:

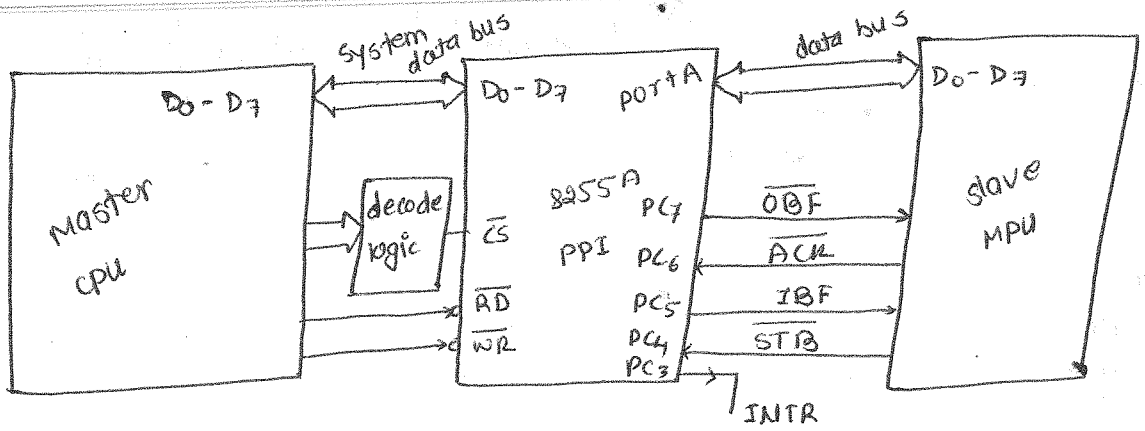


Fig: Master - slave configuration using 8255A I/O mode 2.

Master to slave:

step1: The master MPU reads the status of $\overline{OBF}$ to varify whether the previous byte has been received by the slave MPU.

step2: The master writes data into port A and the 8255A informs the slave by causing the signal $\overline{OBF}$ to go low.

step3: The slave checks by causing the $\overline{OBF}$ signal from the master for data availability.

step4: The slave MPU reads data from port A and acknowledges the reading at the same time by making the $\overline{ACK}$ low.

Slave to master:

step1: The slave MPU checks the handshake signal IBF to findout whether the port A is available to transfer a databyte.

step2: The slave MPU places data byte on the databus and informs the 8255 A by enabling the $\overline{STB}$.

step3: The 8255A causes the IBF to go high and the master MPU reads the signal to find out whether a databyte is available.

step4: Finally the master reads the databyte.

programmable Interrupt controller [8259]

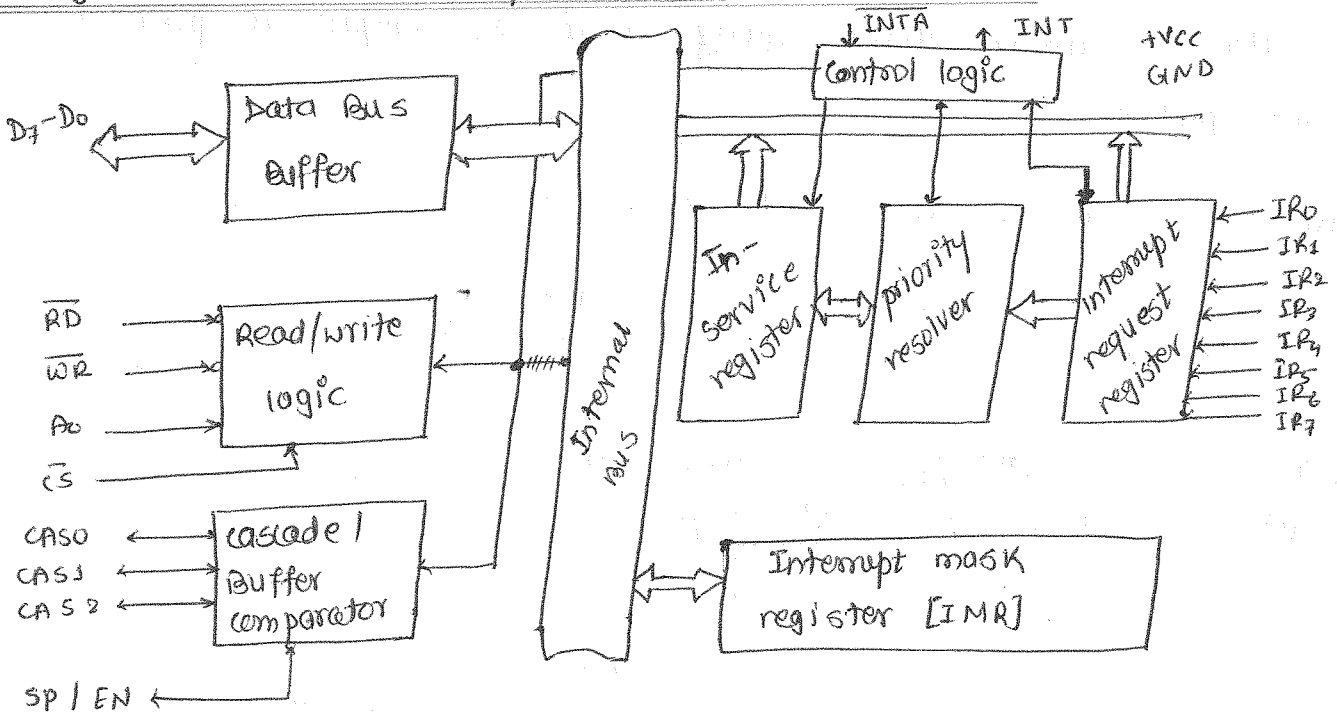


Fig: 8259 Block diagram.

The 8259 is a 28 pin chip which can manage 8-interupts according to the instruction written in its control register. This is equivalent to providing 8 interrupt pins on the processor instead of one. The priority of these 8 plines can also be defined. In cascaded mode this chip can be used to define 64 priority levels.

A0: If A0 = 0, ICW1 is selected
= 1, ICW2, ICW3, ICW4 and OCW are selected.

CAS0, CAS1 and CAS2 are used in cascade mode. In cascade one chip is configured as master with atmost 8 slaves. These pins are used as i/p from master to slave.

In cascade mode, $\overline{SP}/\overline{EN}$ is used to determine whether the chip is mastered or slave.

If $\overline{SP}/\overline{EN}$ = 0, it is slave
= 1, it is master

In buffer mode, $\overline{SP}/\overline{EN}$ is used to enable data bus buffer.

IF $\overline{SP}/\overline{EN}$ = 0, buffer is enabled

= 1, buffer is disabled

INT is the interrupt o/p connected to the microprocessors interrupt pin. $\overline{INTA}$ is the interrupt acknowledgement i/p from the microprocessor

IRR [Interrupt Request register]:

The 8 interrupt i/p sets are connected to the IRR. When the i/p lines goes high, the corresponding request are stored there.

priority Resolver:

It determines the priority of the interrupts according to the information provided on OCW.

ISR [In-service Register]:

It stores the information about the interrupts currently being serviced.

IMR [Interrupt Mask Register]:

This Register can be programmed by OCW to store the bits which mask the specific interrupts.

Interrupt Operation:

1. The IRR stores the requests.
2. The priority resolver checks three registers: the IRR for interrupt request, The IMR for masking bit, and the ISR for interrupt request being served. It resolves the priority and sets the INT high when appropriate.
3. The MPU acknowledges the interrupt by sending $\overline{INTA}$.
4. After the $\overline{INTA}$ is received, the appropriate priority bit in the ISR is set to indicate which interrupt level is being served, and the corresponding bit in the IRR is reset to indicate that the request is accepted. Then, the opcode for the CALL instruction is placed on the data bus.
5. When the MPU decode the CALL instruction, it places two more $\overline{INTA}$

signals on the data bus.

6. When the 8259A receives the second $\overline{INTA}$, it places the low-order byte of the CALL address on the data bus. At the third $\overline{INTA}$, it places the high-order byte on the data bus. The CALL address is the vector memory location for the interrupt; this address is placed in the control register during the initialization.

7. During the third $\overline{INTA}$ pulse, the ISR bit is reset either automatically or by a command word that must be issued at the end of the service-routine. This option is determined by the initialization of command word.

8. The program sequence is transferred to the memory location specified by the CALL instruction.

priority modes:

1. Fully nested mode:

In this mode the interrupt request are arranged from highest to lowest with IR0 as the highest and IR7 as the lowest priority.

IR_0	IR_1	IR_2	IR_3	IR_4	IR_5	IR_6	IR_7
4	5	6	7	0	1	2	3
			↑	↑			
			lowest priority	highest priority			

In addition, any IR can be assigned the highest priority in this mode.

2. Automatic Rotation mode:

In this mode, a device, after being serviced, receives the lowest priority. Assuming that the IR_3 has just been serviced, it will receive the seventh priority as shown below:

IR_0	IR_1	IR_2	IR_3	IR_4	IR_5	IR_6	IR_7
5	6	7	0	1	2	3	4

3. Specific Rotation mode:

This mode is similar to the automatic rotation mode, except that the user can select any IR for the lowest priority, by fixing all other priority.

End of interrupt Types:

1. Non-specific EOI command:

When this command is sent to the 8259, it resets highest priority ISR bit.

2. Specific EOI command:

This command specifies which ISR bit to reset

3. Automatic EOI:

In this mode, no command is necessary. During the third INTA, ISR bit is reset. The major drawback of this mode is that the ISR does not have information on which IR is being serviced. Thus any IR can interrupt the service routine.

Command words:

Initialization command words (ICW):

1) ICW1

A7	A6	A5	1	LTIM	ADI	SNGL	ICW4

IF ICW_4 = 1, ICW_4 is needed
= 0, ICW_4 is not needed

IF SNGL = 1, 8259 is in single mode
= 0, 8259 is in cascade mode

IF ADI = 0, interval of 8 is used (i.e. 0, 8, 16, ...)
= 1, interval of 4 is used (i.e. 0, 4, 8, ...)

IF LTIM = 1, the interrupt operation takes place at level trigger
= 0, the interrupt operation takes place at edge trigger.

Interval of 4:

	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
7	A7	A6	A5	1	1	1	0	0
6	A7	A6	A5	1	1	0	0	0
5	A7	A6	A5	1	0	1	0	0

4	A_7	A_6	A_5	1	0	0	0	0
3	A_7	A_6	A_5	0	1	1	0	0
2	A_7	A_6	A_5	0	1	0	0	0
1	A_7	A_6	A_5	0	0	1	0	0
0	A_7	A_6	A_5	0	0	0	0	0

Interval of 8:

	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
7	A_7	A_6	1	1	1	0	0	0
6	A_7	A_6	1	1	0	0	0	0
5	A_7	A_6	1	0	1	0	0	0
4	A_7	A_6	1	0	0	0	0	0
3	A_7	A_6	0	1	1	0	0	0
2	A_7	A_6	0	1	0	0	0	0
1	A_7	A_6	0	0	1	0	0	0
0	A_7	A_6	0	0	0	0	0	0

2) ICW2:

A_{15}	A_{14}	A_{13}	A_{12}	A_{11}	A_{10}	A_9	A_8

ICW 2 holds the higher 8-bit of interrupt vector address.

3) ICW3

ICW3 is only used in cascade mode

for master:

S_7	S_6	S_5	S_4	S_3	S_2	S_1	S_0

If S_0 to S_7 = 1, corresponding i/p has a slave

= 0, corresponding i/p doesn't have a slave

for slave:

0	0	0	0	0	ID_2	ID_1	ID_0

slave ID

ID2	ID1	ID0	
0	0	0	→ Connected to master's IR0
0	0	1	→ Connected to master's IR1

0	1	0	connected to master's IR2
0	1	1	connected to master's IR3
1	0	0	connected to master's IR4
1	0	1	connected to master's IR5
1	1	0	connected to master's IR6
1	1	1	connected to master's IR7

4) ICW4:

0	0	0	SFNM	BUF	M/S̄	AEOI	uPM

IF uPM = 0, uP model used is 8085/8080
= 1, uP model used is 8086/8088

IF AEOI = 1, interrupt is ended automatically
= 0, interrupt is not ended automatically

IF M/S̄ = 1, the 8259 is master
= 0, the 8259 is slave

IF BUF = 1, the 8259 is in buffered mode
= 0, the 8259 is not in buffered mode.

IF SFNM = 1, the 8259 is specially fully nested mode
= 0, the 8259 is not special fully nested mode.

operational command word (OCW):

1. OCW1:

m_7	m_6	m_5	m_4	m_3	m_2	m_1	m_0

IF m_0 to m_7 = 1, the corresponding interrupt mask is set i.e. corresponding interrupt is disabled.
= 0, the interrupt mask is reset, i.e. corresponding interrupt is enabled.

2. OCW2:

It is used only when automatic end of interrupt (AEOI) is not selected. It is used to set the type of EOI and the priority of the interrupt.

8. OCW 3:

It is used for polling mode.

9. Explain the following initialization instruction for 8259.

```
MVI  A, 79H   } ICW1
OUT  80 H
MVI  A, 20 H  } ICW2
OUT  8Q H
MVI  A, 04H   } ICW3
OUT  8Q H
MVI  A, 16 H  } ICW4
OUT  83 H
```

Ans:

interval of 8 is used.

ICW1:

A_7	A_6	A_5	1	LTIM	ADI	SNGL	ICW4
0	1	1	1	1	0	0	1

= 7g H

↑ Level trigger ↑ cascade mode ↑ ICW4 needed

ICW2:

A_{15}	A_{14}	A_{13}	A_{12}	A_{11}	A_{10}	A_9	A_8
0	0	1	0	0	0	0	0

= 20 H

ICW3:

S_7	S_6	S_5	S_4	S_3	S_2	S_1	S_0
0	0	0	0	0	1	0	0

= 04H

↑ IR 2 has a slave

ICW4:

0	0	0	SFNM	BUF	M/S	AEOI	uPM
0	0	0	1	0	1	1	0 ← 8085/8080 uP used

↑ special fully nested mode ↑ not buffer ↑ master mode ↑ auto end of interrupt

Interrupt vector Table:

	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
IR_7	0	0	1	0	0	0	0	0	0	1	1	1	1	0	0	0
IR_6	0	0	1	0	0	0	0	0	0	1	1	1	0	0	0	0
IR_5	0	0	1	0	0	0	0	0	0	1	1	0	1	0	0	0
IR_4	0	0	1	0	0	0	0	0	0	1	1	0	0	0	0	0
IR_3	0	0	1	0	0	0	0	0	0	1	0	1	1	0	0	0
IR_2	0	0	1	0	0	0	0	0	0	1	0	1	0	0	0	0
IR_1	0	0	1	0	0	0	0	0	0	1	0	0	1	0	0	0
IR_0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0

Direct memory Access (DMA)

→ Direct memory Access (DMA) is an I/O technique commonly used for high-speed data transfer, for example, data transfer between system memory and floppy disk.

→ In status checked I/O and interrupt I/O, data transfer is relatively slow because each instruction needs to be fetched and executed.

→ In DMA, the MPU releases the control of the buses to a device called a DMA controller.

→ The controller manages data transfer between memory and a peripheral under its control, thus bypassing the MPU.

→ DMA transfer uses two signals - HOLD and HLDA in 8085 microprocessor.

HOLD:

→ This is an active high input signal to 8085 from another master requesting the use of address and data buses.

→ After receiving the Hold request, the MPU relinquishes (releases) the buses in the following machine cycle.

→ All - buses are tri - stated and Hold Acknowledge (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 MPU is relinquishing control of the buses.

→ A DMA controller uses these signals - as if it were a peripheral requesting the MPU for the control of the buses.

Concept of DMA:

The sequence of DMA transfer:

1) Originally, microprocessor is connected to the memory as shown in fig. below. with switches closed for address, data and control buses. When peripheral wants to transfer data using DMA transfer, it sends DMA request, DREQ, signal to the DMA controller.

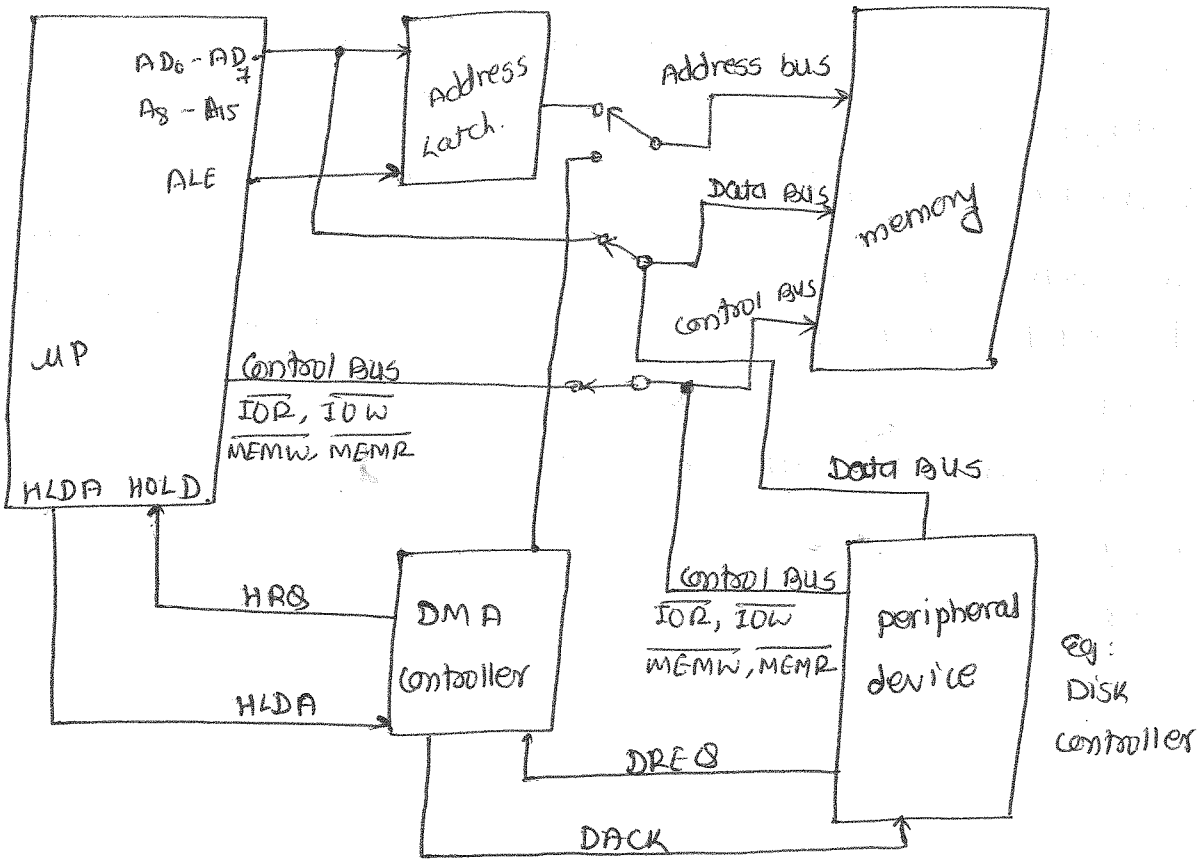


Fig: Block diagram showing DMA transfer

2) If the input (channel) of the DMA controller is unmasked, the DMA controller will send a hold-request, HRQ signal to the microprocessor's HOLD input.

3) The microprocessor finishes the current machine cycle and floats its buses, sending out a hold-acknowledge signal, HLDA, to the DMA controller.

4) When DMA controller receives HLDA signal, it will send out a control signal which throws the 3 bus switches down to their DMA position. This disconnects the processor from buses and connects DMA controller to the buses. Now DMA controller sends out the address of the byte to be transferred and sends out DMA-acknowledge (DACK) signal to the peripheral devices to tell it to get ready to output the byte.

5) Then the DMA transfer begins and finally when the data transfer is complete, the DMA controller unasserts its hold-request signal to the processor and releases the buses.

8237 DMA controller:

→ The DMA controller should have:
- A data bus
- An address bus
- Read/write control signals, and
- Control signals to disable its role as a peripheral and to enable its role as a peripheral.

→ DMA controller is a processor capable only of copying data at high speed from one location to another location.

→ 8237 is a programmable Direct memory Access controller (DMA) with 40 pin

→ It has four independent channels with each channel capable of transfering 64K bytes.

→ It must interface with MPU and a peripheral device

→ It is an I/O device to MPU

→ It is a data transfer processor to peripheral device.

→ Many of its signals that are input in the I/O mode become outputs in the processor mode.

→ 8237 has four independent channels CH0 - CH3. Two 16-bit registers are internally associated with each channel.

→ Count Register: It is used to load a count of the number of bytes to be copied.

→ Memory address Register: It is used to load the starting address of the byte to be copied.

→ The address of these registers are determined by four address lines: A_3 to A_0 and the chip select ($\overline{CS}$) signal.

DMA signals:

→ To obtain DMA service, a request is generated by activating the DREQ line of the channel.

→ DACK are output lines to inform the individual peripherals that DMA

is granted. DREQ and DACK are equivalent to handshake signal in I/O devices.

→ AEN and ADSTB - Address enable and Address strobe are used to latch a high-order byte to generate a 16-bit address.

→ After receiving the HRQ (Hold request), the MPU completes the bus cycle in process and issues the HLDA (Hold Acknowledgement) signal.

System Interface :

→ When a transfer begins, the DMA places the low-order byte on the address bus and high-order byte on the data bus.

→ Then 8237 asserts AEN (Address Enable) and ADSTB (Address strobe)

→ These two signals are used to latch the high-order byte from the data bus and 8237 places the 16-bit address on the system bus.

Serial I/O (8251/8250)

→ Specially, there are two basic approaches for serial data communication. They are:

- Software - controlled serial I/O and
- Hardware - controlled serial I/O

→ The 8085 microprocessor has two pins specially designed for software-controlled serial I/O. One is called SOD (serial output data) and the other is called SID (serial input data).

→ But for the hardware-controlled serial I/O, different integrated circuits are used. The two basic chips used for serial communication are as follows:

1. 8250 UART (Universal Asynchronous Receiver Transmitter)
2. 8251 - USART (Universal Synchronous Asynchronous Receiver Transmitter)

8250 UART

→ 8250 UART is used as an interface by 8086 microprocessor, to perform

serial communication.

- While transmitting, it converts data from parallel to serial. While receiving, it converts the data from serial to parallel.
- It supports signal for RS-232 which is a popular serial communication bus standard.
- Moreover, it was commonly used in PCs and related equipments such as printers or modems.

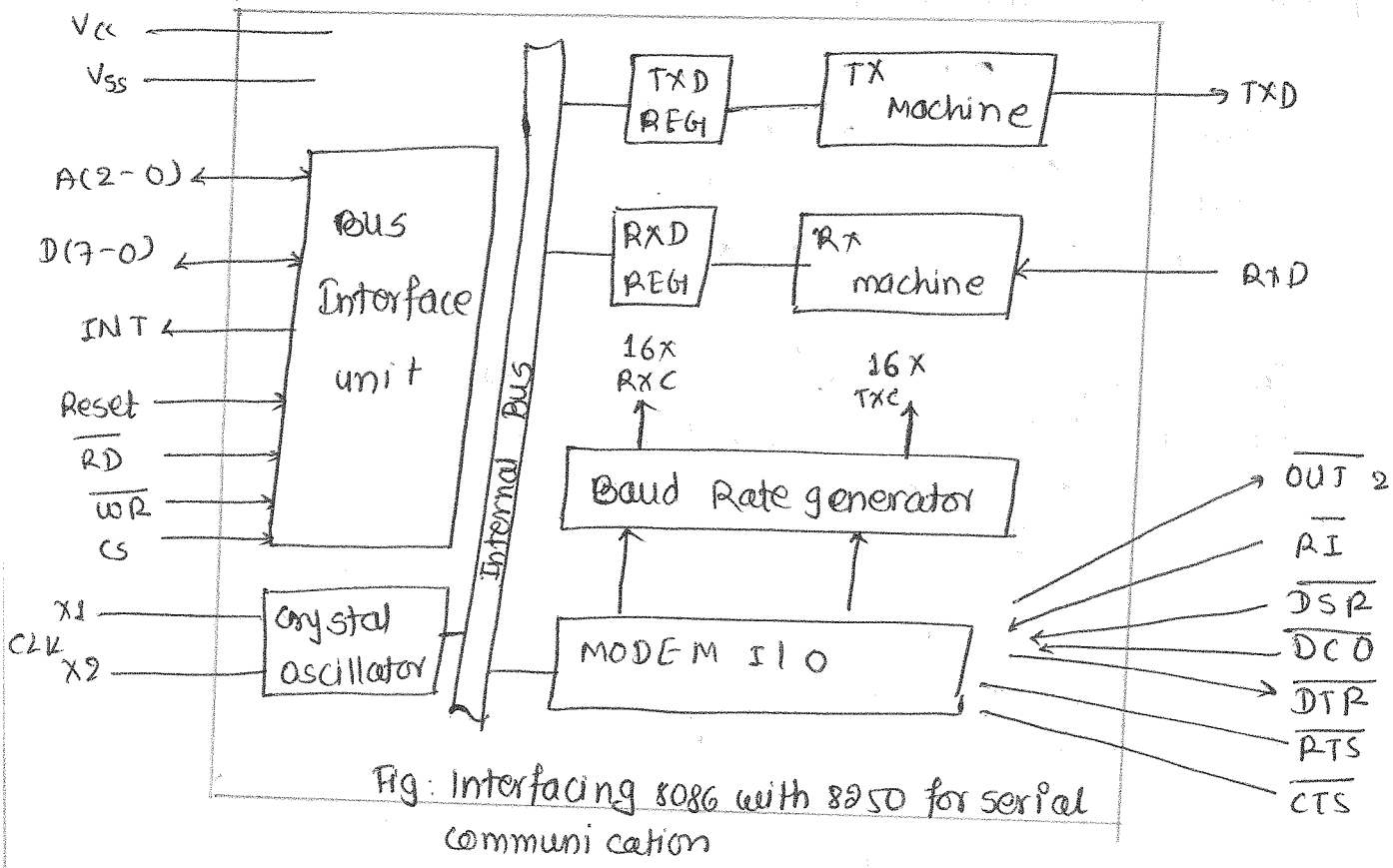


Fig: Interfacing 8086 with 8250 for serial communication

- In this block diagram, the internal bus of 8250 UART transfers data between the transmit/receive registers and the bus interface unit.
- Generally, 8250 UART can be used for serial communication using modem, so the pins required for serial communication through modem are also available in 8250 UART.
- For asynchronous serial data transfer, 8250 UART has Baud Rate Generator that selects appropriate Baud Rate of data transmission.

The 8251 USART:

- The 8251 is a programmable chip designed for synchronous and asynchronous serial data communication, packaged in a 28-pin DIP.
- It is compatible with large/wide range of microprocessor such as 8048,

8080, 8085, 8086 and 8088.

→ The 8251A is used as a peripheral device and is programmed by the CPU to operate using virtually any serial data transmission technique presently in use.

→ The USART accepts data characters from the CPU in parallel format and then converts them into a continuous serial data stream for transmission.

→ Simultaneously, it can receive serial data streams and converts them into parallel data characters for the CPU.

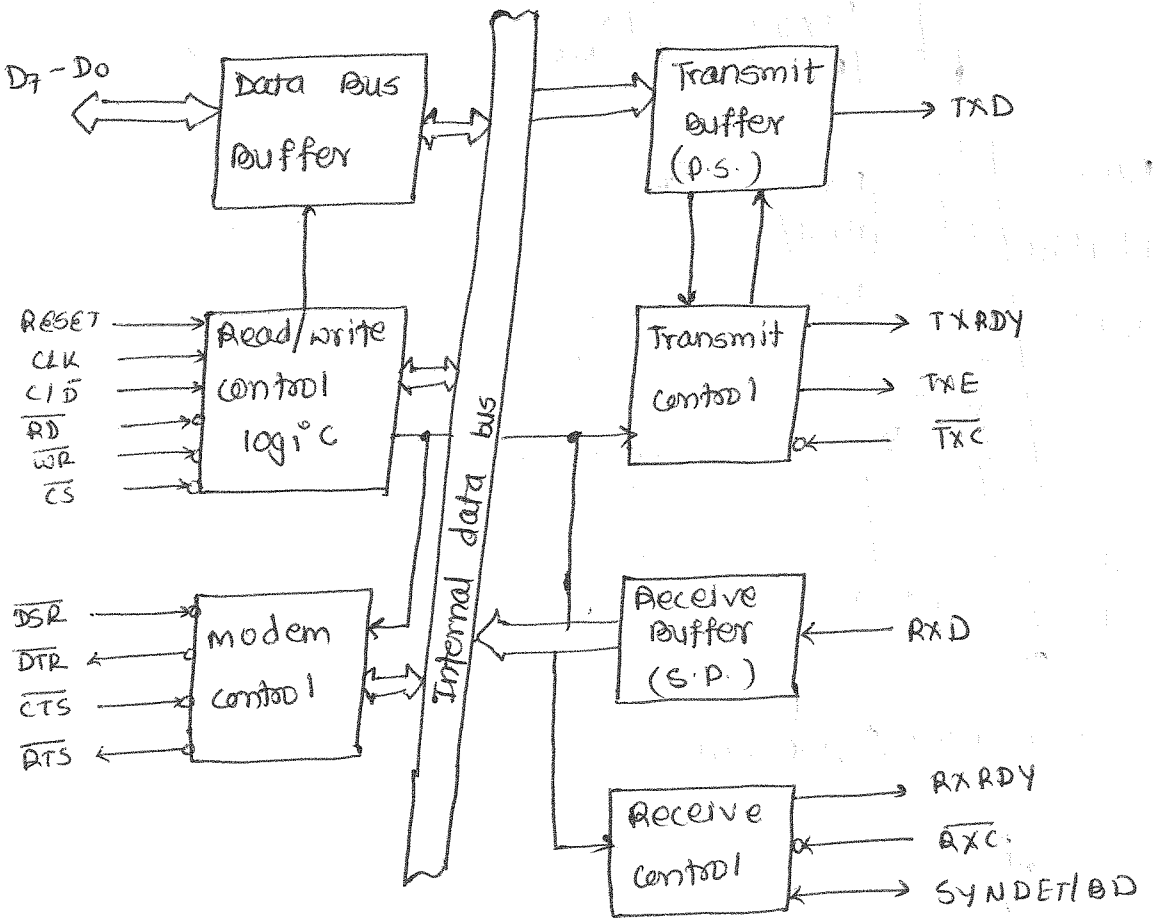


Fig: Functional block diagram of 8251A USART

→ Fig. above shows the functional block diagram of 8251A USART. It includes five sections which are:

- Read/write control logic and registers
- Transmitter section
- Receiver section
- Data Bus Buffer section, and
- Modem control section.

Chapter 4: 16-bit Microprocessor and programming:

Salient features:

8086 is the first 16-bit microprocessor from INTEL, released in the year 1978. It is a 40 pin DIP chip based on HMOS. The term 16 bit means that its ALU, its internal registers and most of the instructions are designed to work with 16 bit binary words. 8086 is available at different clock frequencies, 5 MHz, 8 MHz and 10 MHz. 8086 Microprocessor has a 16-bit data bus and 20-bit address bus. So, it can address any one of 2^{20} = 1 mega byte memory locations.

Architecture of 8086

To improve the performance by implementing the parallel processing concept the CPU of 8086 is divided into two independent sections. They are Bus interface unit (BIU) and execution unit (EU). The BIU sends out addresses, fetches instructions, read data from ports and memory and writes data to ports and memory i.e. BIU handles all transfers data and addresses on the buses required by the execution unit. Whereas the execution unit decodes the instructions and executes the instructions.

The execution unit:

The execution unit consists of a control system, a 16-bit ALU, 16-bit flag register and four general purpose registers (AX, BX, CX, DX), pointer registers (SP, BP) and index registers (SI, DI) of each 16-bits.

The control circuitry controls the internal operations. The decoder in the execution unit decodes the instructions fetched from the memory into a series of actions. The ALU can add, subtract, logical AND, OR, XOR, increment, decrement, complement and shifting the binary numbers.

Bus Interface unit:

The BIU consists of a 6-byte long instruction register called queue. And four ~~stack~~ segment registers (ES, CS, SS, DS), one instruction pointer and adder circuit to calculate the 20 bit physical address of a location. This bus interface unit will perform all the external bus operations. They are fetching the instructions from the memory, read/write data from/into memory or port and also supporting the instruction queue etc. The BIU fetches up to six instruction bytes from the memory and stores these pre-fetched bytes in a first-in first out register set called Queue. When the execution unit is ready for the execution of the instruction, instead of fetching the byte from the memory, it reads the byte from the queue. This will increase the overall speed of microprocessor. Fetching the next instruction while the current instruction executes is called pipelining or parallel processing.

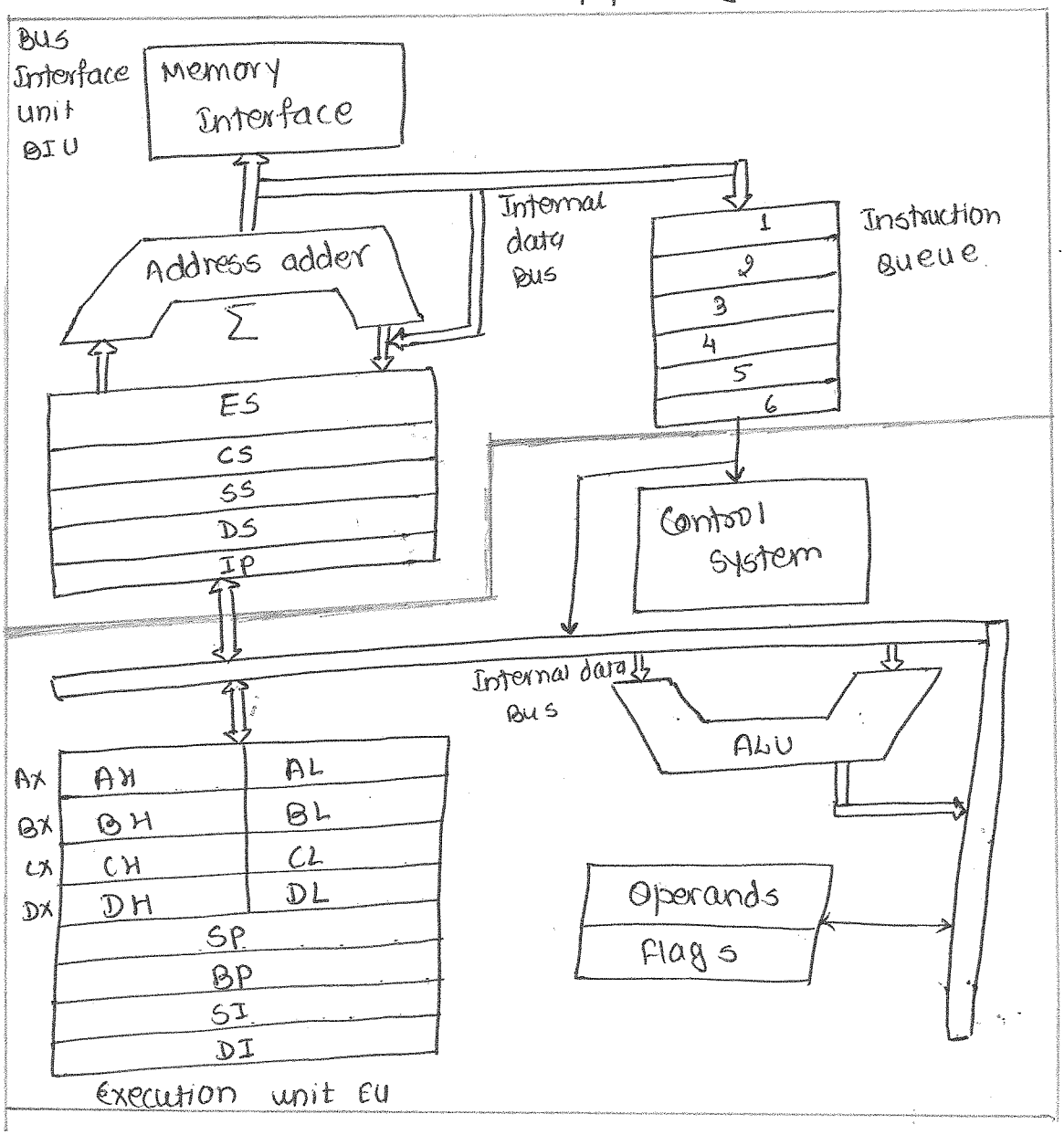


Fig: Architecture of 8086 Microprocessor

Register organization:

The 14 Registers of 8086 Microprocessor are categorized into different groups. They are general purpose data registers, pointers and index registers, segment register and flag register as shown in the table below:

S.N.	Type	Register width	Name of the Registers
1.	General purpose Registers	16-bit	AX, BX, CX, DX
		8-bit	AL, AH, BL, BH, CL, CH, DL, DH
2.	pointer Registers	16-bit	stack pointer (SP) Base pointer (BP)
3.	Index Registers	16-bit	Source Index (SI) Destination Index (DI)
4.	segment Registers	16-bit	Code segment (CS) Data segment (DS) stack segment (SS) Extra segment (ES)
5.	Instruction	16-bit	Instruction pointer (IP)
6.	flag (PSW)	16-bit	Flag Register

1. Code segment Register (CS) and Instruction pointer (IP):

→ Code segment Register (CS) holds the starting address of memory from where the code segment starts.

→ Instruction pointer (IP) contains the distance or offset, which denotes how far the code is located from the start.

→ Both are 16 bit register

→ BIU computes the 20 bit physical address using the content of CS and IP.

→ Content of CS is shifted by 4 bits to the left (or multiplied by $(16)_{10} = (10)_H$) and 16 bit offset address is added to produce 20 bit physical address.

Eg: [IP] = 0050H and [CS] = 0010H

Then physical address = 0010 * 10 + 0050
= 00100 + 0050 = 00150H

2. Stack segment register (SS) and stack pointer register (SP) or Base pointer (BP)

→ Stack segment register contains the starting address of stack segment (current stack).
→ Stack pointer holds the 16 bit offset address from the start of the segment.
→ 20 bit physical address is computed using SS and SP.
→ PUSH and POP instructions use this.
→ BP is used instead of SP when based addressing mode is used. In this mode 20 bit physical address is computed from SS and BP.

3. Data segment Register (DS)

→ Data segment Register points to the starting address of data segment.
→ Operands or data are fetched from the segment.
→ 16 bit contents of SI (Source Index) or DI (Destination Index) are combined with DS register content to obtain 20 bit physical address.

4. Extra segment register (ES)

→ If the data segment memory (64 KB) is not sufficient then extra segment is used.
→ Extra segment register (ES) contains the starting address of extra segment.
→ String operations utilize ES where 20 bit physical address is computed from ES and DI.

5 General purpose Registers:

a) AX Register:

→ It is also called 16-bit accumulator.
→ Can be divided into 2, 8 bit registers AH and AL.
→ AL is called 8 bit accumulator.
→ I/O operations used AX or AL registers
→ Multiplication and division instructions used AX or and AL.
→ AL is similar to accumulator of 8085.

b) BX Register:

→ Also called Base register.

→ The only general purpose register whose contents can be used for addressing 8086 memory. All memory reference utilizing this register content for addressing uses the DS as the default segment register.
→ It is similar to the 8085, H, L Register.
→ 16 bit register divided into BH and BL each of 8-bit.

c) CX Register:

→ 16 bit counter Register
→ SHIFT, ROTATE and LOOP use the content of CX as a counter
→ Example: The instruction LOOP START will automatically decrement CX by 1 without affecting flags and will check if contents of CX becomes zero. If it is zero, the 8086 executes the next instruction; otherwise the 8086 branches to the label 'START'.

d) DX Register:

→ 16 bit data register
→ 16X16 bit multiplication, 32 and 16 bit division make use of this register.

6. Flag register:

→ Flag register in the EU (execution unit), holds the status flags, typically after an ALU operation.
→ 16 bit register which contains Nine different flags.
→ Out of 9 flags 6 are status and 3 are control flags.
→ Control flag can be set or reset by the programmer.

X	X	X	X	O	D	I	T	S	Z	X	Ac	X	P	X	Cy

Fig: Flag of 8086 registers

Status Flag:

i) OF (Over flow flag): is set if there is an arithmetic over flow i.e. if the size of the result exceeds the capacity of the destination location.
ii) SF (Sign flag): is set if the most significant bit of the result is 1; otherwise it is zero.
iii) ZF (zero flag): is set if the result is zero. ZF is zero for a non-zero result.

iv) AcF (Auxillary carry flag): is set if there is a carry from the lower nibble into the higher nibble or a borrow from higher nibble into the lower nibble of the low or high 8 bits of 16 bit number. This flag is used by BCD arithmetic instruction; otherwise it remains reset.

v) PF (Parity Flag): is set if the results has even parity. If remains zero for odd parity of the result.

vi) CF (Carry Flag): is set if there is a carry from addition or borrow from subtraction.

Control Flags:

i) DF (Direction Flag): setting DF causes string instructions to auto decrement; clearing DF causes string instruction to auto increment.
i.e. If DF = 0, string is process from lowest address to highest address
If DF = 1, string is process from highest address to lowest address.

ii) IF (Interrupt Flag): Setting IF causes 8086 to recognize external maskable interrupts; clearing IF disables these interrupts.

iii) TF (Trap Flag): Setting TF puts the 8086 in the single step mode. In this mode, the 8086 generates an internal interrupt after execution of each instruction. The user can write a service routine at the interrupt vector address to display the designed registers and memory locations.

8086 Addressing modes:

The different ways in which a source operand is denoted in an instruction are known as the addressing modes. There are 10 different addressing modes in 8086 programming. They are:

1. Immediate addressing mode
2. Register addressing mode
3. Direct addressing mode
4. Register indirect addressing mode
5. Based addressing modes
6. Indexed addressing mode

7. Based Indexed Addressing mode
8. Based, Indexed with displacement
9. String address
10. IO addressing mode.

1. Immediate addressing mode:

The addressing mode in which the data operand is a part of the instruction itself is called Immediate addressing mode.

For Ex: MOV CX, 4847H
ADD AX, 2056H
MOV AL, FFH.

MOV al, 2400H X
MOV ds, 2400H X

2. Register addressing mode:

In Register addressing mode, both source and destination operands are register. We can use 16-bit registers or 8-bit registers both but 8-bit and 16 bit source and destination are not allowed.

For Ex: MOV AX, BX
ADD CX, DX

MOV CS, ax X ; CS can not be destination
MOV ds, cs X ; both can not be segment register.

3. Direct addressing mode:

→ The addressing mode in which the effective address of the memory location at which the data operand is stored is given in the instruction.
→ Physical address is calculated using segment register and displacement value in the operand.
→ unless specified segment register is ds.

Eg: MOV bx, start ; Moves the content of memory location whose physical address = [DS] x 10 + start

MOV ax, [1234H] ; Moves the content of memory location whose physical address = [DS] x 10 + 1234

4. Register indirect addressing mode:

→ Register indirect addressing allows data to be addressed at any memory location through an offset address held in any of the following registers: BP, BX, DI and SI.

Eg: MOV AX, [BX] ∵ [BX] points the address of BX

i.e. suppose the register BX contains 4675 H, the content of the 4675 H are moved to AX.

→ Physical address is calculated from segment register and base or index register.

→ By default BP uses SS and others use DS.

5. Based Addressing mode:

→ Effective address (offset) is the sum of displacement and the content of BX or BP.

→ The segment register may DS or SS.

→ If stack is used physical address is computed from bp and ss otherwise bx and ds.

eg:
```
MOV AX, [BX+04]
ADD CL, [BX+08]
```

6. Indexed Addressing mode:

The operands offset address is found by adding the contents of SI or DI register and 8-bit or 16-bit displacements.

Ex:
```
MOV BX, [SI+06]
ADD AL, [DI+08]
```

7. Based-Indexed addressing mode:

The offset address of the operand is computed by summing the base register to the contents of an Index register.

Ex:
```
ADD CX, [BX+SI]
MOV AX, [BX+DI]
```

8. Based-Indexed with displacement mode:

The operand offset is computed by adding the base register contents, an index registers contents and 8 or 16-bit displacement.

Ex:
```
MOV AX, [BX+DI+08]
ADD CX, [BX+SI+16]
```

9. String address:

→ SI is used to point first byte (word) of source string and DI is used to point first byte (word) of destination string.
→ DS is default segment for source and ES for destination.
→ SI and DI are increased or decreased depending upon direction flag (DF).

Eg: MOVS WORD

Before its execution, Suppose

[DF] = 0 [DS] = 4000 H [SI] = 0040 H [ES] = 8000 H

[DI] = 0080 H [40040] = FDH [40041] = AB H

[80080] = 54 H and [80081] = 77 H.

After execution.

Content of 4000 x 10 + 0040 = 40040 i.e. FDH is moved to the address 8000 x 10 + 0080 = 80080 ∴ [80080] = FDH.

∵ DF = 0, SI is increased by 1, di is increased by 1.

∴ Content of (40041 H) = AB H is moved to [80081 H]

∴ [80081 H] = AB H

SI and DI are again increased by 1.

∴ SI = 0042 H and di = 0082 H

10. IO addressing:

Eg: IN ax, [dx]

IN al, 8-bit-port-address

Assembler:

An assembler translates program written in assembly language into a machine language program (object code). Assembly language programs are called 'source codes'. Machine language programs are known as object codes. A translator converts source code to object code and then into executable format. The process of converting source code into object code is called compilation and

assembler does it.

The assembler generates two files. The first file is the object file with the extension .OBJ. The object file consists of the binary codes for the instructions and information about the addresses of the instructions. After further processing, the contents of the file will be loaded into memory and run. The second file is the assembler list file with the extension .LST.

The process of converting object codes into executable format is called linking and linker does it.

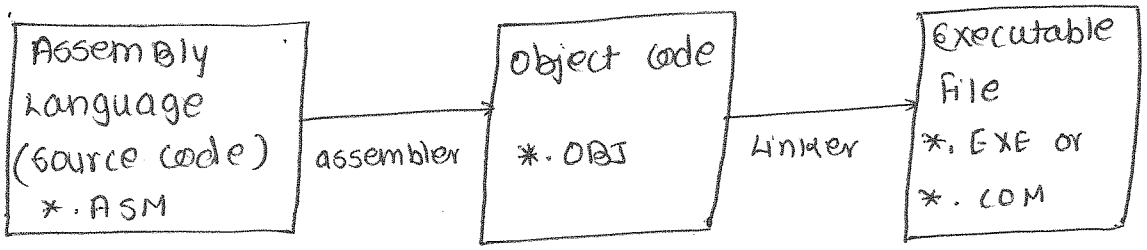


Fig: process of compilation and linking Assembly Language

Each instruction in an assembly language is composed of three or four fields as follows:

- Label field
- Instruction, Mnemonic or op-code field
- Operation field / operand field
- Comment field

Eg: L1 : MOV A, B ; Moves content of Register B into register A.

Label field — op-code field — operand field — Comment field.

There are Two ways of converting an assembly language program into machine language:

① Manual assembly

programmer translates each mnemonic into its numerical machine language representation by looking up a table of the microprocessor's instruction set, which contains both assembly and machine

language instruction.

② Assembly using assembler

When an assembler is used, the assembler program reads each assembly instruction of a program as ASCII characters and translates them into respective binary op-codes.

Types of Assemblers

There are various types of assemblers:

1. One pass/single pass Assembler:

This assembler goes through the assembly language program once and translates the assembly language program. This assembler has the program of defining forward references. This means that a JUMP instructions using an address that appears later in the program must be defined by the programmer after the program is assembled.

2. Two Pass assembler:

This assembler scans the assembly language program twice In the first pass, the assembler generates the table of the symbols. A symbol table consists of labels with addresses assigned to them. On the second pass, the assembler translates the assembly language program into the machine code. The two pass assembler is hence more efficient and easier to use.

3. Macro Assemblers:

This type of assembler translates a program written in macro language into the machine language. A macro language is the one in which all the instruction sequence can be defined using macros. A macro is an instruction sequence that appears repeatedly in a program assigned with a specific name. The macro assembler replaces a macro name with the appropriate instruction sequence each time it encounters a macro name.

Difference between subroutine and macro: (X not required)

A specific subroutine occurs once in a program. A subroutine is executed by calling it from a main program. The program execution jumps out of the main program and then executes the subroutine. At the end of the subroutine, a RET instruction is used to resume program execution following the CALL SUBROUTINE instruction in the main program. A macro, on the other hand, does not cause the program execution to branch out of the main program. Each time a macro occurs, it is replaced with the appropriate sequence in the main program. The advantages of using macros are that the source programs becomes shorter and program documentation becomes better.

Conditional macro assembly is very useful in determining whether or not an instruction sequence shall be included in the assembly depending on a condition that is true or false. If two different programs are to be executed repeatedly based on a condition that can be either true or false. It is convenient to use conditional macros. Based on each condition, a particular program is assembled.

Debugger:

A debugger is a program which allows to load your object code program into system memory, execute the program and trouble-shoot or debug it. The debugger allows to look into the contents of registers and memory locations after the program runs.

Some debugger allows to stop the program after each instruction so that you can check or alter memory and register contents. This is called single step debug. A debugger also allows to set a break-point at any point in the program. If we insert a break point, the debugger will run the program upto the instruction where the breakpoint is put and then stop the execution.

Linker:

A Linker is a program used to connect several object files into one large object file while writing large programs it is better to divide the large program into smaller modules. Each module can be individually written, tested and debugged. Then all the object modules are linked together to form one functioning program. These object module can also be kept in library file and linked into other programs as needed. A linker produces a link file which contains the binary codes for all the combined modules. The linker also produces a link map file which contains the address information about the linked files. The linker which come with MASM assemblers produce link file with the .EXE extension.

Assembly language features:

The main features of ALP are program comments, reserved words, identifiers & statements, which provide the basic rules and framework for the language.

program comments:

- The use of comment throughout a program can improve its clarity
- It starts with semicolon (;) and terminates with a new line.
- e.g: `ADD AX,BX ; Adds AX and BX`

Reserved words:

- Certain names in assembly language are reserved for their own purpose to be used only under special conditions and includes
- Instructions: such as MOV and ADD (Operation to execute)
- Directives : such as END, SEGMENT (information to assembler)
- Operators : such as SIZE
- predefined symbols: such as @DATA, @MODEL

Identifiers:

- An identifier (or symbol) is a name that applies to an item in

the program that expects to reference.

- Two types of identifiers are Name and Label.
- Name refers to the address of a data item such as NUM1 DB 5, COUNT DB 0.
- Label refers to the address of an instruction
  L1 : ADD BL, 73

statements:

- ALP consists of a set of statements with two types.

(i) Instruction:

Instruction which the assemblers translate into machine code.

(ii) Directives:

- The directives are the number of statements that enables us to control the way in which the source program assembles and lists.
- These statements act only during the assembly of program and generate no machine-executable code.
- The different types of directives are:

1. TITLE:

It defines the title of the program so that the next typed to the right of these directives is printed at the top of each page in the listing file as comment.

2. DOSSEG:

There is a standard order for placing the stack, code and data segments. One can place the segments in any order as well. This directive tell the assembler to place the segments in standard order.

3. MODEL:

This directive determines the size of each segment. All of the program models except tiny result in the creation of .EXE program. The tiny

model creates a .COM program.

model	Description
TINY	code and data together may not be greater than 64K
SMALL	Neither code nor data may be greater than 64K
MEDIUM	Only the code may be greater than 64K
COMPACT	Only the data may be greater than 64K
LARGE	Both code and data may be greater than 64K
HUGE	All available memory may be used for code and data.

.STACK:

The STACK directive sets the size of the program stack, which may be any size up to 64K. The stack segment is addressed by SS and SP registers.

⑤ .CODE:

The CODE directive identifies the part of the program that contains instruction (code). The code segment is addressed by CS and IP registers.

⑥ PROC:

The PROC directive creates a name and an address for the beginning of a procedure. A procedure is a group of instructions that is given a common name.

⑦ ENDP:

The ENDP directive marks the end of a procedure. Its name must match the name used by the PROC directive.

⑧ .DATA:

All variables are defined in the area following the .DATA directive called the data segment.

⑨ END:

The END directive terminates assembly of the program and any statements after it are ignored.

2. Defining data types:

→ EQU, DB, DW, DD and DUP directives

ⓐ EQU:

It is used to define constants.

eg: const_name EQU 5000H

so, MOV ax, const_name ≈ MOV ax, 5000H

const_name = 5000H

ⓑ DB, DW and DD:

→ Used to define variable of different types

Var_name Dn expression

where, Var_name ⇒ program reference the data item by means of variable name ⇒ optional

→ Dn may be:

Directives	Description	NO. of bytes
DB	Define Byte	1
DW	Define word	2
DD	Define double word	4
DQ	Define Quad word	8
DT	Define Ten bytes	10

Expression: Defines the value with which variable is to be defined.

eg:

① array1 dB 5;

array1

05

② array2 dw 10;

array2

0A
00

③ array3 dB 2,3,5,6;

array 3

02
03
05
06

PAGE Directives:

→ It helps to control the format of a listing of an assembled program.

→ It is optional directives

→ Its format is

PAGE [LENGTH], [WIDTH]

The .startup & .exit directives:

→ .startup generates the instruction to initialize the segment register

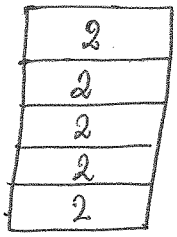
→ .exit generates the INT21H function. 4Ch instruction for existing program.

→ MASM 6.0 introduce these directives to simplify program initialization and Termination

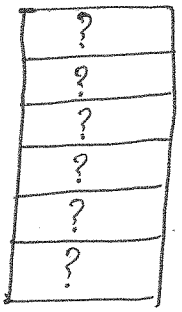
⑥ DUP directives:

Dup defines multiple variables initialize with same value

var DB 5 dup (2)



array db 6 dup(?)



Some commonly used instructions of 8086:

MOV

syntax: MOV reg/memory, memory/reg/immediate

eg: MOV ax, 1234
MOV bl, cl
MOV al, value1
MOV [di], 16H

XCHG

syntax: XCHG register/memory, register/memory

eg: XCHG al, ah

LEA (Load effective address)

Syntax: LEA register, memory

• data

data1 db 5

Lea bx, data1 ;

equivalent code; MOV ax, offset data1.

LOOP

syntax: loop label (value of cx decremented to 0)

INC and DEC

syntax: INC/DEC reg/memory

→ Flag affected: OF, SF, ZF

ADD and SUB, ADC and SBB

ADD/SUB reg/mem., reg/mem./immediate_value

ADC/SBB reg/mem., reg/mem./immediate

eg: ADC ah, cl

MUL/IMUL/AAM:

→ MUL: multiply unsigned byte by byte or unsigned word by word

→ IMUL: Multiply signed byte by byte or signed word by word

→ AAM: ASCII adjust after multiplication

→ flag affected: OF, CF

DIV/IDIV/AAD:

→ DIV: Divide unsigned word by byte or unsigned double word by word

→ IDIV: Divide signed word by byte or signed double word by word

→ AAD: ASCII adjust after division

Shift Instruction:

→ SHL/SAL: shift bits of a word or byte left, put zero(s) in LSBs

→ SHR: shift bits of word or byte right, put zeros in MSBs.

→ SAR: shift bits of a word or byte right, copy old MSB into New MSB.

JUMP

① unconditional jump: JMP address

② conditional jump

JE/JZ: jump on equal or zero

JNE/JNZ: Jump on not equal or non zero

JA/JNBE: Jump on above / Jump on not below or equal

JAE/JNB: Jump on above or equal / Jump on not below

JB/JNAE: Jump on below / Jump on not above or equal

JBE/JNA: Jump on below or equal / Jump on not above.

Flag transfer registers:

LAHF: Load AH with the low byte of the flag register

SAHF: Store AH register to low byte of flag register

PUSHF: Copy flag register to top of the stack

POPF: Copy word at top of the stack to flag register.

Flag set/clear instructions:

STC: Set carry flag CF to 1

CLC: clear carry flag CF to 0

CMC: Complement the state of the carry flag CF.

STD: set direction flag DF to 1 (decrement string pointer)

CLD: clear direction flag DF to 0

STI: Set interrupt enable flag to 1 (enable INTR input)

CLI: Clear interrupt enable flag to 0 (disable INTR input)

programming:

Q1. WAP to add 34 and 36

⇒
```
TITLE program to add 34 and 36
   dosseg      ; auto formating
 .model small
 .stack 64H
 .code
    main proc
    mov ax, 34
    mov bx, 36
    ADD ax, bx
   ; Termination of program
    MOV ax, 4C00H
    int 21H
    main endp
    end main
```

Q2. WAP to add two data defined in data segment

⇒ TITLE program to add two data from data segment

```
dosseg
.model small
.stack 100
.data
value1 db 30H
value2 db 40H
.code
main proc
mov ax, @data
mov ds, ax
lea si, value1
lea di, value2
MOV ax, [si]
MOV bx, [di]
add ax, bx
MOV ax, 4C00H
int 21H
main endp
end main
```

Q3. WAP to add elements between 50 and 150 and store the result in one of the registers.

Soln:

```
TITLE program to add
      elements between
      50 and 150
dosseg
.model small
.stack 100
.data
value1 dw 500, 100, 600, 60, 50, 550,
          150, 75, 125, 130
.code
main proc
mov ax, @data
mov ds, ax
lea si, value1
MOV ax, 00
MOV cx, 10
label: MOV dx, [si]
CMP dx, 50
JBE skip
CMP dx, 150
JAE skip
add ax, dx
skip: inc si
      inc si
      loop label
MOV ax, 4C00H
int 21H
main endp
end main
```

Q4. WAP to add three numbers stored in memory location and store the result at another memory location.

⇒

```
dosseg
.model small
.stack 100
.data
value1 db 10,60,70
value 2 db ?
.code
main proc
MOV ax,@ data
MOV ds, ax
lea si, value1
lea di, value2
MOV cx,02
MOV ax, [si]
L1: inc si
MOV bx,[si]
ADD ax, bx
loop L1
MOV [di], ax
MOV ax, 4C00 H
int 21 H
main endp
end main
```

Q5. WAP to find the smallest data among the four data stored in the table.

Sol^n:

```
dosseg
.model small
.stack 100
.data
value1 db 4, 8, 10, 12
value 2 db ?
```

```
• code
  main proc
  mov ax, @ data
  Mov ds, ax
  Mov cx, 03
  lea si, value 1
  lea di, value2
  mov [ax], [si]
L1: inc si
  mov bx, [si]
  CMP ax, bx
  JB SKIP
  XCHG ax, bx
SKIP: loop L1
  Mov [di], ax
  Mov ax, 4C00h
  int 21 h
  main endp
  end main
```

Q6. WAP TO copy the data from one table to another.

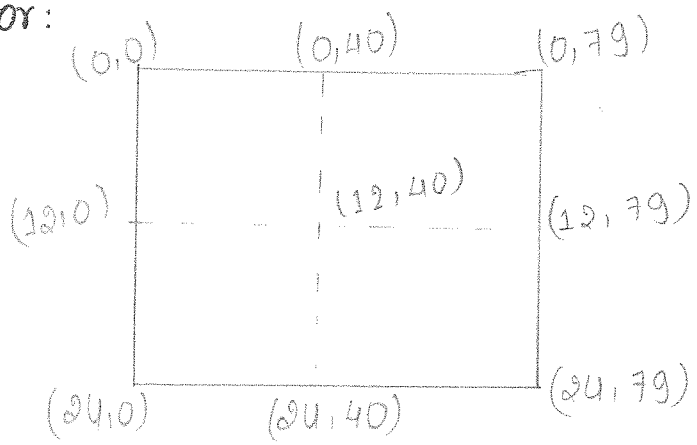
Sol^n:

```
  dosseg
  .model small
  .stack 100
  .data
    Table dB 4, 8, 12, 16
    Table dB 4 dup(?)
  .code
  main proc
  mov ax, @data
  mov ds, ax
  mov cx, 4
```

```
lea si, table
lea di, table1
L2: mov al, [si]
    mov [di], al
    inc si
    inc di
    loop L2
  mov ax, 4c00h
  int 21h
 main endp
 end main
```

Introduction to screen and keyboard processing:

Typical monitor:



ROW = 25

Column = 80

Tasks to do:

- → cursor locate
- → clear screen
- → char display
- → string display
- → keyboard input

Screen location	Decimal format		Hex format	
	ROW	column	ROW	column
upper left corner	00	00	00	00
upper right corner	00	79	00	4F
Center of screen	12	40	0C	28

Lower Left corner	24	00	18	00
Lower right corner	24	79	18	4F
Left middle	12	00	0C	00
Right middle	12	79	0C	4F

① setting the cursor:

→ Interrupt 10 H (i.e. INT 10 H)
→ Function 02H in AH register
→ Required page number in BH register
→ ROW in DH register
→ Column in DL register

Eg: Set the cursor to row 8 and column 15.

```
MOV AH, 02H
MOV BH, 00H
MOV DH, 08H
MOV DL, 0FH
INT 10H
```

② Clearing the screen:

→ Function 06H in ah
→ number of lines to scroll up [00 from for full screen] in al
→ attribute value [color, reserve video, blinking] in bh.
→ starting row in cx
→ Final row in dx
→ Interrupt 10h

Eg: Clear the screen, set background = White, foreground = blue

```
MOV ah, 06
MOV al, 00
MOV bh, 71 ; [7 = White, 1 = blue]
MOV cx, 0000H
MOV dx, 184FH
INT 10H.
```

R	G	B	color
0	0	0	Black
0	0	1	Blue
0	1	0	Green
1	0	0	Red
1	1	1	white

③ Screen display (string):

Eg: Display the string "RAM"

⇒
```
.data
  str db 'RAM','$'
.code
 MOV ah, 09H
 lea dx, str
 int 21 H
```

→ Interrupt 21 H

→ function 09H in ah register

→ required defination of a ~~string~~ display. string in data area are immediately followed by '$' sign.

→ dx → load display string address.

Eg: Display your name with white background and blue foreground at cursor location (8,5)

soln: ⇒
```
dosseg
 .model small
 .stack 100
 .data
   name db 'Ramesh','$'
 .code
 main proc
 mov ax, @data
 MOV ds, ax
 ; clear the screen
 MOV ah, 06h
 MOV al, 00h
 MOV bh, 71h
 MOV cx, 0000h
 MOV dx, 184Fh
 INT 10h
 ; set the cursor
 MOV ah, 02H
 MOV bh, 00H
 MOV dh, 08H
 MOV dl, 05H
 int 10H
 ; display string
 MOV ah, 09H
 lea dx, name
 int 21h
 ; termination of program
 mov ax, 4C00h
 int 21h
 main endp
 end main
```

Write a program to display ASCII characters.

⇒

```
dosseg
.model small
.stack 100
.data
   ASCII db 00,'$'
.code
   main proc
   MOV ax, @data
   mov ds, ax
;clear the screen
   MOV ah,06h
   mov al,00h
   MOV bh, 74h
   MOV cx, 0000h
   MOV dx, 18UFh
   int 10h.
;set the cursor
   MOV ah, 02H
   MOV bh,00H
   MOV dh, 10H
   MOV dL, 05H
   int 10H.
;display string
   MOV cx, 256
L2: MOV ah, 09
   lea dx, ASCII
   int 21H
   inc ASCII
   loop L2
; termination of program
   MOV ax,4C00h
   int 21h
   main endp
   end main.
```

④ Keyboard input:

→ Function 0A H in ah

→ int 21H

parameters for input area

```
paralst     label    byte
maxlen      db       20
actlen      db       ?
kbdata      db       20 dup(?)
```

```
mov ah, 0A
lea dx, paralst  ; mov dx, maxlen
int 21h.
```

WAP to read string of maximum length 70 characters and display it at the centre of the screen.

⇒ dosseg

```
.model small
.stack 100h
.data
 maxlen  db 70
 actlen  db ?
 string  71 dup ('$')
.code
 main proc
 mov ax, @data
 mov ds, ax
; keyboard i/p
 mov ah, 0AH
 lea dx, maxlen
 int 21 H
; clear screen
 mov ah, 06H
 mov al, 00H
 mov bh, 07 H
 mov cx, 0000 H
 mov dx, 184F H
 int 10 H
; set the cursor
 mov ah, 02H.
 mov al ; actlen
 SH R al, 01 H
 mov dl, 40
 sub dl, al
 mov dh, 12
 mov bh, 00
 int 10h
; termination of program
 mov ax, 4c00h
 int 21 h
main endp    end main.
```

⑤ character display

→ Function 02 H in ah register
→ Interrupt 21 H
→ character to display in dl register

display your name using character display function.

⇒
```
dosseg
.model small
.stack 100h
.data
   var1 db 'Ramesh'
.code
main proc
mov ax, @data
mov ds, ax
; display
mov ah, 02h
mov cx, 06h
lea si, var1
up: mov dl, [si]
int 21h
inc si
loop up
; termination of program
mov ax, 4c00h
int 21h
main endp
end main.
```

Some common DOS function and interrupts:

→ DOS functions are interrupts ~~are~~ used for IO services.
→ Interrupt occurs when currently running program is interrupted.
→ The microprocessor recognizes two types of interrupt.

① Hardware : → This interrupt is generated when peripheral needs attentation.

② Software : → It is the call of sub-routine located in OS, generally IO routine
→ Commonly used software interrupts are int 10h for video serives and int 21h for DOS services.

int 10h:

→ DOS provides interrupt 10h for video display control
→ This interrupt is used to control the screen format, color text style, scroll etc.
→ The corresponding function number is loaded in AH register with other parameters.

Function No.	Description
0	Set video mode.
1	Set Cursor Lines
2	Set Cursor position
3	Get cursor position
4	Redd light pen
5	Set display page
6	Scroll window up
7	Scroll window down
8	Read character and attribute at cursor position
9	Write character and attribute at cursor position
0A H	Write character only
0B H	Set color palette
0C H	Write dot
0D H	Read dot
0E H	Write character
0F H	Get video mode

int 21h:

It is a DOS function call. There are some 87 different functions supported by this interrupt, identified by a function number placed in the AH register. Function 00h through 0gh are listed below

Function No.	Description
0	Terminate the current program
1	Console input with echo
2	character output
3	Auxiliary Input
4	Auxiliary output
5	printer output
6	Direct console input-output
7	Console input
8	Console input without echo
9	string output

Interrupt:

Interrupt is the process in which microprocessor is diverted from its running work to another.

process of Interrupt Operation:

1. I/O device issues interrupt signal to the microprocessor
2. Processor finishes current instruction before responding to interrupt.
3. Microprocessor then tests for an interrupt, if it gets high, then sends ACK signal to the device that issued the interrupt signal.
4. Processor now transfer its control to the routine that serves the interrupt request and is called interrupt service routine that resides on specific memory.

⑤ For this process, microprocessor needs to save information needed to resume the current program at the point of interrupt. The minimum information required is:

ⓐ Status of the microprocessor that is the content of PSW register.

ⓑ Location of next instruction to be executed.

⑥ The microprocessor now loads its program counter with the location of interrupt service routine so that the control is now transferred to interrupt handler program.

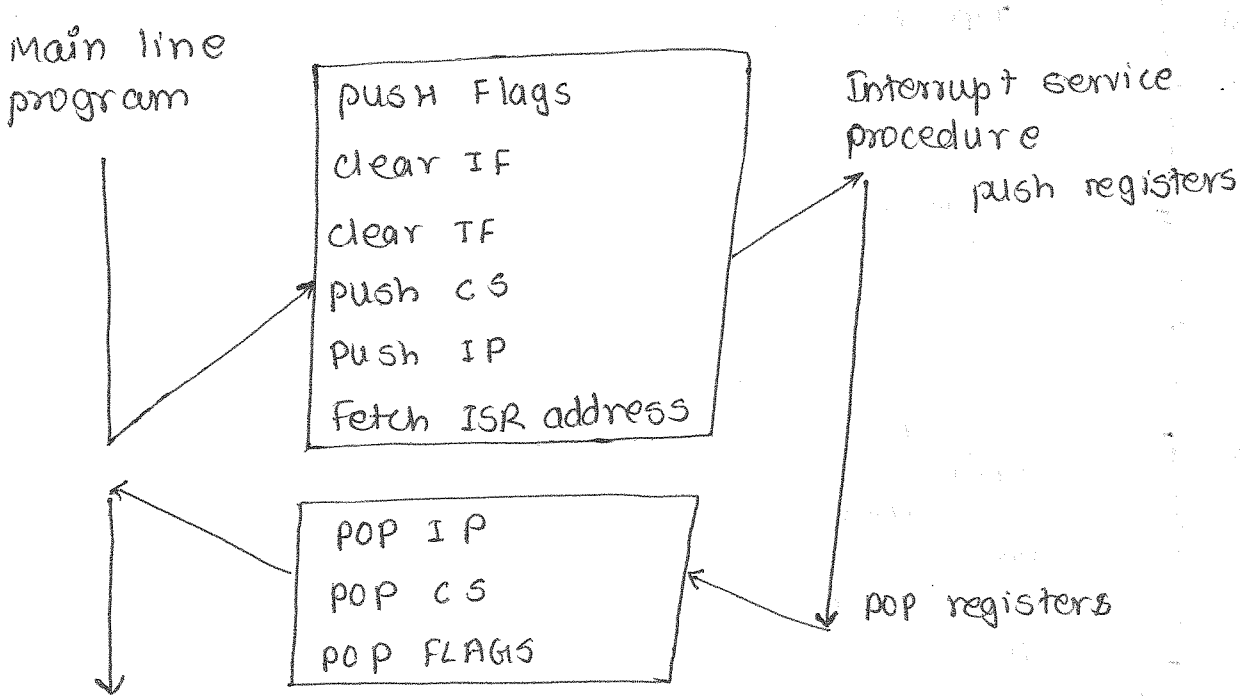


Fig: Interrupt Response

⑦ ISR first saves the content of all the registers

⑧ Now interrupt is processed

⑨ After completing, interrupt processing register contents are retrives from stack finally. PSW and PC contents are retrived to return to the main program.

Interrupt service Routine (ISR):

→ To serve the interrupt request from the device ISR is used by the processor.

→ ISR is situated at some specific memory location.

→ ISR consist of code to serve interrupting device.

Interrupt vector table:

The starting address of an ISR is known as Interrupt vector or Interrupt pointer and the table containing the starting address of all the interrupts is known as Interrupt vector table or Interrupt pointer table.

In 8086, there are 256 types of interrupts (type 0 to type 255) so there are 256 interrupt vectors arranged in the table in memory as shown in the figure below:

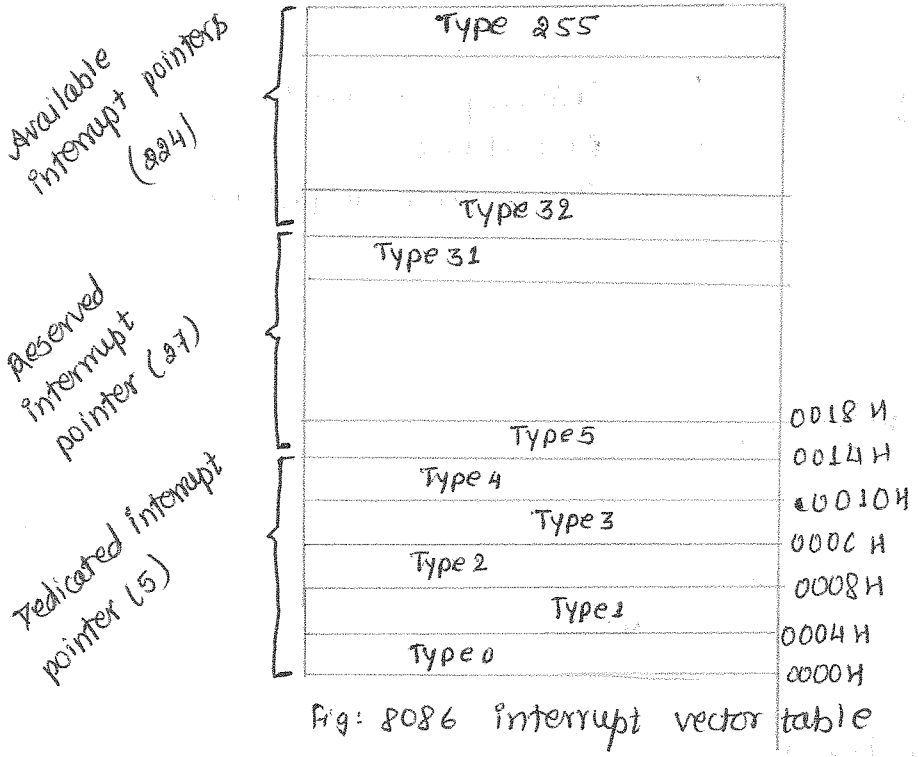


Fig: 8086 interrupt vector table

Each vector is a 4 byte long and contains the starting address of the interrupt service routine. The first 2 bytes of the vector contain the offset address (IP) and the last two bytes contain the segment address (CS).

Types of interrupt:

1. software interrupt:

→ The software interrupts are program instruction which are inserted at desired location in the program.

→ The 8085 has 8 software interrupts from RST 0 to RST 7. The vector address for these interrupts can be calculated as follows:

Vector address = Interrupt number * 8

Eg: For RST 5, vector address = 5*8 = 40 = 28H

Vector address for RST 5 is 0028H.

Interrupt	vector address
RST 0	0000H
RST 1	0008H
RST 2	0010H
RST 3	0018H
RST 4	0020H
RST 5	0028H
RST 6	0030H
RST 7	0038H

2. Hardware interrupt:

→ An external device initiates the hardware interrupts by placing an appropriate signal at the interrupt pin of the microprocessor.

→ If the interrupt is accepted then processor executes an interrupt service routine.

→ 8085 has 5 hardware interrupts:

Interrupt	vector address
TRAP	0024H
RST 5.5	002CH
RST 6.5	0034H
RST 7.5	003CH
INTR	Non-vectored

ⓐ TRAP

→ This is non-maskable interrupt. It is unaffected by mask or interrupt enable

→ Has highest priority

→ When TRAP goes high, it remains high until it is acknowledged.

→ If HOLD and TRAP signals are received at the same time, HOLD is recognized first then TRAP

→ TRAP can be cleared by:
  * Giving TRAP Acknowledge signal
  * Resetting microprocessor.

(d) RST 7.5, RST 6.5, RST 5.5

→ Maskable interrupts
→ priority: RST 7.5 has 2nd, RST 6.5 has 3rd and RST 5.5 has 4th
→ enable by EI
→ Disable by DI, reset or recognization

(e) INTR

→ Maskable
→ Lowest priority
→ Enabled by EI and disabled by DI, reset or recognization
→ Non vectored.

Interrupt can also classified as:

(1) External

→ Generated by external devices via the interrupt pins of microprocessor
→ Two types:

i) Maskable:

→ Can be enabled or disabled using EI and DI instruction
→ If microprocessor's Interrupt flag is disabled, it ignores maskable interrupts.
→ EI enables and DI disables the interrupt.

ii) Non-Maskable:

→ Can not be enabled or disabled by instructions
→ Have highest priority i.e. served first

(2) Internal:

→ Activated internally by exceptional conditions such as overflow, divide by zero, execution of illegal opcode.
→ Handled in similar way as external interrupt
→ solved using service routine.

Prioritising Interrupts:

A processor is provided with more than one interrupt pins on the chip. Therefore a special mechanism is necessary to handle interrupts from several devices that share one of these pins. There are mainly two ways of serving multiple interrupts.

1. Polled interrupts
2. Vectored interrupts (chained)

1. Polled interrupts:

Polled interrupts are handled using mostly software and are therefore slower compared to vectored (hardware) interrupts. The processor responds to an interrupt by executing one general service routine for all devices. The priorities of these devices are determined by the order in which the routine polls each devices.

In order to service an interrupt, the processor checks the starting with the highest priority device. Once the processor determines the source of the interrupt, it branches to the service routing for that device.

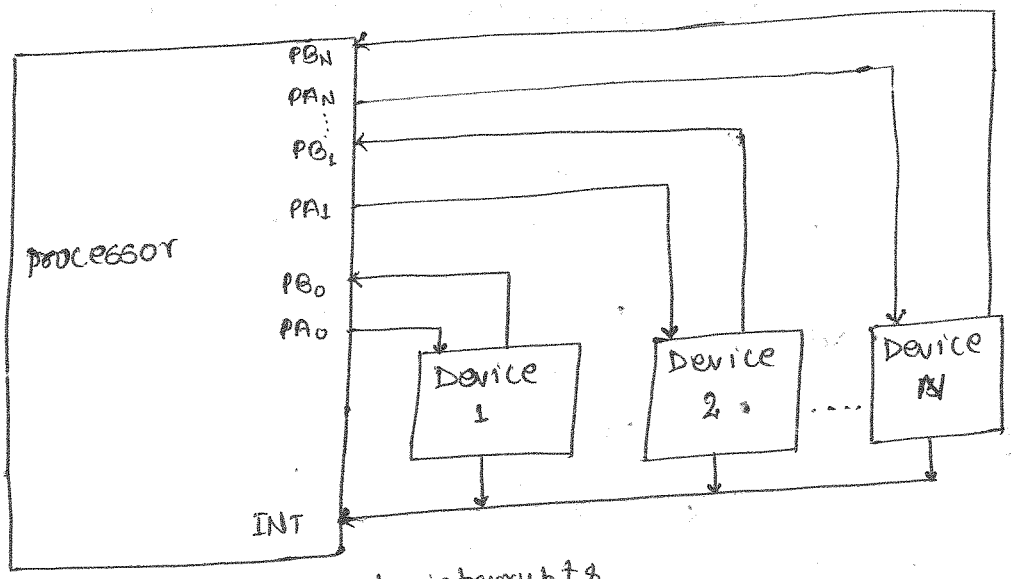


Fig: Polled interrupts.

As shown in figure, several external devices (Device 1, Device 2, ... Device N) are connected to a single interrupt line (INT) of the processor. When one or more device activate the INT line high, the processor saves the contents of the PC and other register and then branches to an address defined by the manufacturer of the processor.

The user can write a program at this address in order to poll each device starting with highest priority device in order to find the source of the interrupt.

polled interrupts are very simple. But for a large number of devices, the time required to poll each device may exceed the time to service the device. In such a case, the faster mechanism called vectored interrupt is used.

2. Chained (vectored) Interrupts:

When one or more devices interrupt the processor at a time, In response, the processor saves its current status and then generates an interrupt acknowledge (INTA) signal to the highest priority device. IF this device has generated the interrupt, it will accept the INTA signal from the processor; otherwise; it will pass INTA on to the next device until the INTA is accepted by the interrupting device. Once accepted, the device provides a means to the processor for finding the interrupt address vector using ~~external~~ external hardware. Usually the requesting device responds by placing a word on the data lines.

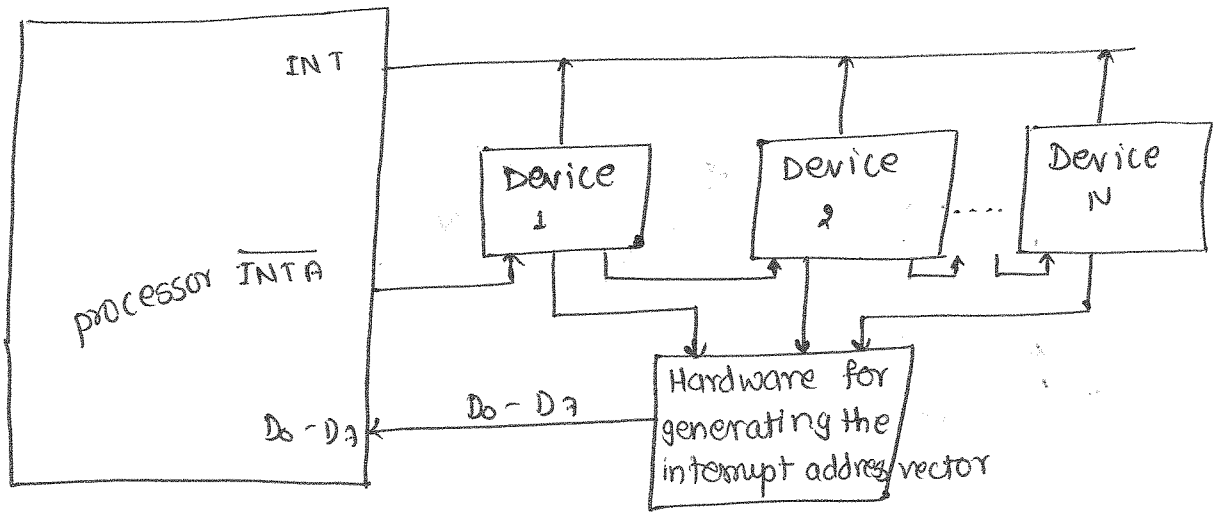


Fig: Vectored Interrupt

With the help of hardware, it generates interrupt vector address. This word is referred to as vector, which the processor uses as a pointer to the appropriate device service routine. This avoids the need to

execute a general interrupt service routine first. So, this technique is also referred to as vectored interrupts.

8086 Interrupt Types:

1. Software Interrupt:

→ Initiated by executing an interrupt instruction:-

INT interrupt-type

where, interrupt type is an integer in the range 0 to 255.

→ Each interrupt type can be parameterized to provide several services - for example, DOS interrupt service, INT 21H provides more than 80 different services.

Application:

→ can be used to insert break-point [INT 3] in the program for debugging.

→ Various interrupt service routine can be tested and each routine can be used to perform some specific task such as reading a character from keyboard, writing a character on CRT etc.

2. Hardware interrupt:

→ Initiated by applying an electrical signal to the processor chip.

→ Hardware interrupts can be maskable and Non-maskable.

→ INTR pin of microprocessor is used for maskable interrupt.

→ Non-maskable interrupt can be triggered by applying an electrical signal to the NMI pin of microprocessor.

Application:

→ Counting, Timing, Real time clock

→ NMI can be used for catastrophic failures such as power failure.

→ A large number of devices can be serviced using hardware interrupts with the help of 8259 PIC.

3. Pre-defined Interrupts:

→ These includes First five interrupts, which are reserved for

specific purpose.

a) Type 0: Divide - error Interrupt:

- Issued by INT 00H instruction
- Type 0 on INTR pin occurs, whenever the result of the division overflows or whenever an attempt is made to divide by zero.

b) Type 1 : Single - step Interrupt:

→ For single-step interrupt, Trap flag (TF) should be set.
→ CPU automatically generates a type 1 interrupt after executing each instruction, if TF is set.
→ Thus, single-step interrupt is useful in monitoring and debugging program.

c) Type 2: Non-maskable Interrupt:

→ This interrupt is initiated when NMI pin of MPU receives a low-to-high transition. This interrupt is normally used for catastrophic condition such as power failure.
→ Whenever, there is a failure of ac power to the system, it is detected by some external circuitry and an interrupt signal is sent to NMI pin of MPU.

d) Type 3: Break-point Interrupt:

→ This interrupt is used for break-point and is non-maskable.
→ When we insert a break point, the system executes the instruction upto the break point and then goes to the breakpoint subroutine.
→ Break-point interrupt can be generated by executing INT 3 instruction.
→ It is useful in debugging.

e) Type 4 : Overflow interrupt

→ Overflow interrupt can be generated by two ways:-
  - by executing INT 4 (unconditionally generates type-4 interrupt)
  - By executing INTO instruction

(Interrupt is generated only if the overflow flag is set)

Interrupt priority:

→ To handle the situation of occurance of two or more interrupts at the same time, priority has been fixed for the interrupts. 8086 MPU has the following interrupt priorities:-

Divide error, INT 0 — Highest priority
NMI
INTR
single-step — Lowest priority

Functional chips:

1. 8284 A : CLOCK GENERATOR:

→ 8284 A is an additional component to the 8086 MPU.

→ The 8284 A provides the following basic functions or signals: clock generation, RESET synchronization, READY synchronization etc.

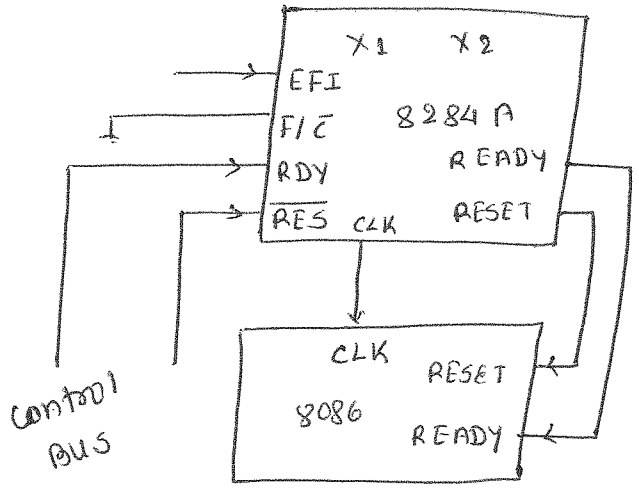


fig: Connection of 8284 A with 8086 microprocessor

→ 8284 A can produce clock in two ways:

- By applying output of pulse generator at EFI (external frequency input). Now we have to keep F/C̄ equal to 1.
- By connecting crystal oscillator across X1 and X2. Now we have to keep F/C̄ (frequency / crystal) = 0.

→ In both the cases, the output clock frequency of 8284 is one-third of input frequency.

8. 8282 : ADDRESS LATCH (20 pin DIPIC)

→ In 8086 the address Bus is multiplexed with the data bus and status signals. To demultiplex this Bus 8282 address latch is used.

→ As the address bus is of 20 bit, three latches are required, each of 8-bit.

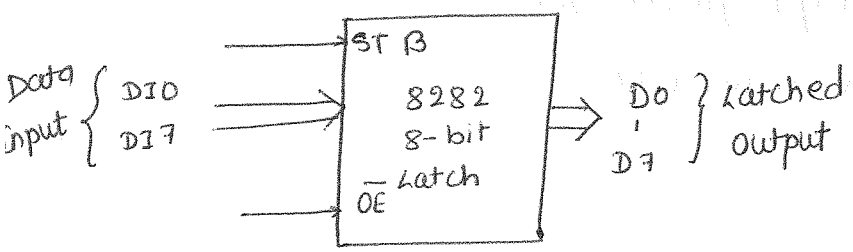


Fig: Block diagram of 8282 address latch.

→ 8282 address latch is selected, when $\overline{OE}$ is low and whenever $\overline{STB}$ goes high, input is transferred to output. And when $\overline{STB}$ remains low, output remains previous value.

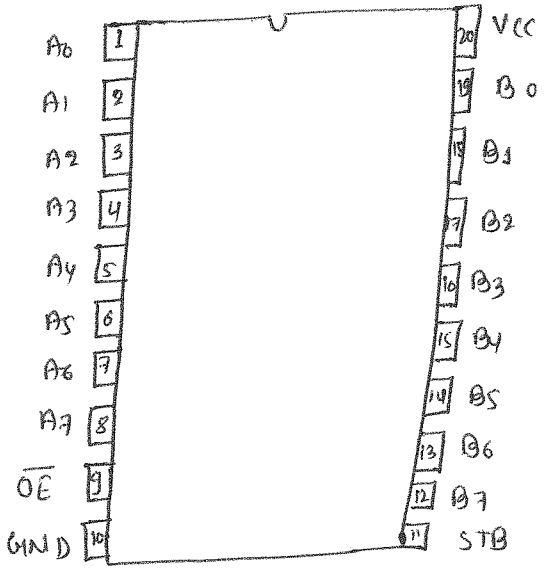


Fig: 8282 Address Latch.

→ pinouts and functions

A0 - A7
- Data input pins
- Data are inputted to 8282 address latch from data input pins.

B0 - B7
- Data output pins
- Data are outputted through Data output pins.

STB
- When STB = 0, then it holds Data

• When STB = 1 then data bits are transferred from the Bus.

$\overline{OE}$
• Output Enable.

3. 8286 : Transceiver:

→ 8286 Transceiver allows two way communications.
→ It acts as bi-directional buffer and increases the driving capability of data bus.
→ It is enabled when $\overline{OE} = 0$
→ T (connected to DT/$\overline{R}$) controls the direction of data:
If T = 1, data is transmitted
If T = 0, data is received
→ As the data bus of 8086 is of 16 bits, two transceivers are required.

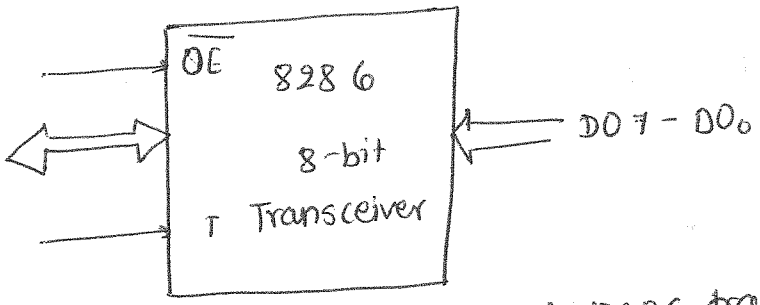


Fig: Block diagram of 8286 transceiver

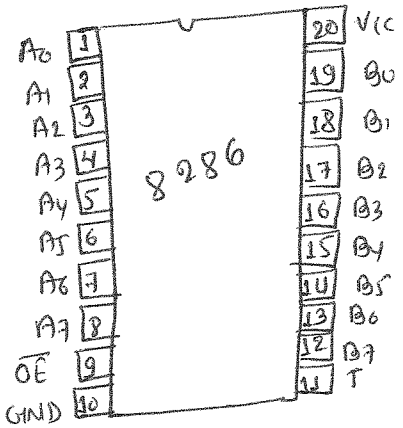


Fig: 8286 Transceiver pin configuration

→ pin out and functions:

A0 - A7
• Local Bus Data I/O pins

B0 - B7
• System Bus Data I/O pins

$\overline{OE}$ (output enable)
• Output Enable

#T (Transmit control pin)

- Transmit control pin. When T = 1, Data are transmitted
- Data are inputted from local Bus Data I/O pins and are inputted by system Bus Data I/O pins.
- Data are outputted by system Bus Data I/O pins and are outputted from Local Bus data I/O pins
- When T=0, Data are received.

4. 8288 : Bus controller

→ It is used to produce different bus control signals like $\overline{MRDC}$ (Memory Read command), $\overline{MWTC}$ (Memory write command), $\overline{IORC}$ (Input output Read command), $\overline{IOWC}$ (Input output write command) etc depending on the condition of status signals ($\overline{S_2}$, $\overline{S_1}$ and $\overline{S_0}$) during maximum mode operation.

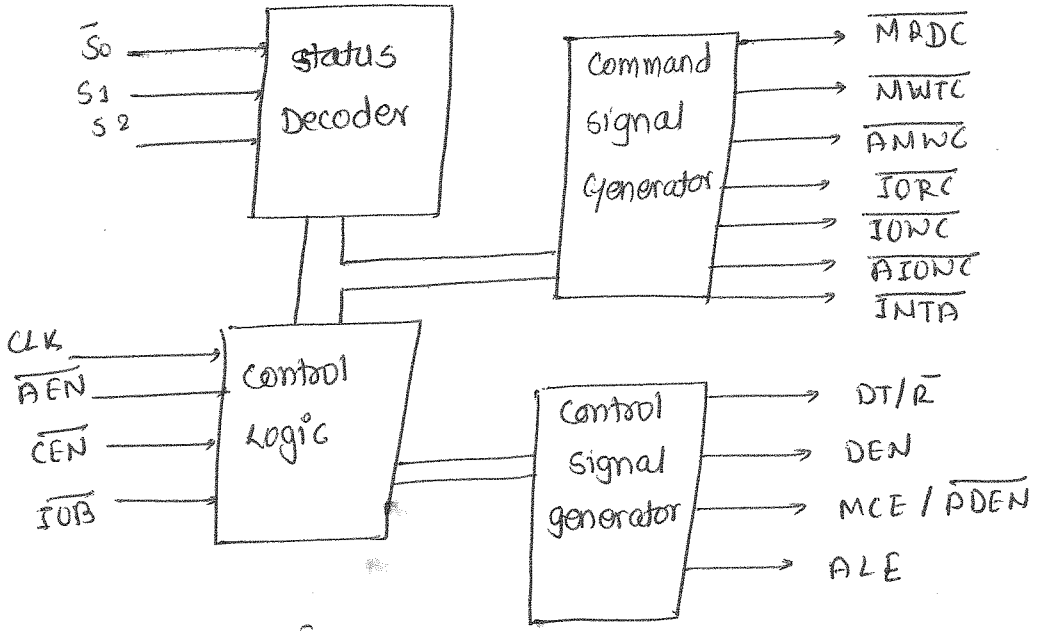


Fig: Block diagram of 8288 Bus controller

- $\overline{MRDC}$: The memory Read control pin provides memory with a read control signal
- $\overline{MWTC}$: The memory write control pin provides memory with its normal write control signal.
- $\overline{IORC}$: The I/O Read command o/p provides I/O with its read control signal.
- $\overline{IOWC}$: The I/O write command o/p provides I/O with its main write signal.

Introduction to Intel 80386:

→ Intel 80386, also known as i386 or 386 is a 32-bit processor introduced in 1985.

→ As the original implementation of the 32-bit extension of the 80286 architecture, the 80386 instruction set, programming model, and binary encodings are still the common denominator for all 32-bit x86 processors, which is termed as i386-architecture, x86 or IA-32.

→ It has 32-bit address bus.

→ The 32-bit ALU allows the 80386 to process data faster and the 32-bit address bus allows the 80386 to address upto 4GB of memory.

→ The 80386 featured three operating modes: real mode, protected mode and virtual mode.

→ The protected mode, which was present in 80286 was extended to allow the 386 to address upto 4GB, so the memory segments can be as large as 4GB.

→ The memory management circuitry and protection circuitry in 80386 are improved over that in 80286, so 80386 is much more versatile as a CPU in a multiuser system.

→ The 80386 has a "virtual 8086" mode, which allows it to easily switch back and forth between 80386 protected-mode tasks and 8086 real mode tasks.

→ The 80386 processor is available in two different versions:
- the 386 DX and
- the 386 SX

→ The 386 DX has a 32-bit address bus and 32-bit data bus. It is packaged in the 132 pins ceramic pin grid array package.

→ The 386 SX, which is packaged in the 100-pin flat has the same internal architecture as 386 DX, but it has only 24-bit address bus and 16-bit data bus.

Assembly Language program development tools:

1. EDITOR

→ An editor is a program which allows to create a file containing the assembly language statements for program.

2. ASSEMBLER

→ An Assembler program is used to translate the assembly language mnemonics for instruction to the corresponding binary codes.

3. LINKER

→ A Linker is a program used to join several files into one large .obj file. It produces .exe file so that the program becomes executable.

4. LOCATOR

→ A Locator is a program used to assign the specific address of where the segment of object code is to be loaded into memory.

→ It usually converts .exe file to .bin file

5. DEBUGGER

→ A debugger is a program which allows to load .obj code program into system memory, execute program and troubleshoot.

→ It allows to look at the content of registers and memory locations after program runs

→ It allows setting the break-points.

6. EMULATOR

→ An Emulator is mixture of hardware and software

→ It is used to test and debug the hardware and software of an external system such as the prototype of a microprocessor based system.