ANNAMACHARYA INSTITUTE OF TECHNOLOGY & SCIENCES
(AUTONOMOUS)
UTUKUR, C. K. DINNE (V & M), KADAPA, YSR DIST.
Approved by AICTE, New Delhi & Affiliation to JNTUA, Anantapuramu.
Accredited by NAAC with 'A' Grade, Bangalore & NBA (EEE, ECE & CSE)

DEPARTMENT OF
ELECTRONICS AND COMMUNICATION ENGINEERING

MICROPROCESSORS AND MICROCONTROLLERS

(23HPC0414)

COURSE MATERIAL

B. TECH III YEAR I SEM

23HPC0414	MICROPROCESSORS AND MICROCONTROLLERS	L	T	P	C
		3	0	0	3

Course Objectives:

1. To learn the fundamental architectural concepts of microprocessors.
2. To gain knowledge about assembly language programming concepts.
3. To get familiar about 8086 interfacing.
4. To understand the fundamentals of the 8051 Microcontroller.
5. To learn interfacing with the 8051Microcontroller.

Course Outcomes:

At the end of this course, the students will be able to

1. Learn the fundamental architectural concepts of microprocessors.
2. Gain knowledge about assembly language programming concepts.
3. Understand the concepts of 8086 interfacing.
4. Learn the fundamentals of the 8051Microcontroller.
5. Know the interfacing with the 8051Microcontroller.

UNIT I

8086 Architecture: Introduction to Microprocessors, Evolution of microprocessors, 8086-Main features, pin diagram/description, 8086 microprocessor family, internal architecture, bus interfacing unit, execution unit, interrupts and interrupt response, 8086 system timing, minimum mode and maximum mode configuration.

UNIT II

8086 Programming: Program development steps, instructions, addressing modes, assembler directives, writing simple programs with an assembler, assembly language program development tools.

UNIT III

8086 Interfacing: Semiconductor memories interfacing (RAM,ROM), Intel8255 programmable peripheral interface, Interfacing switches and LEDS, Interfacing seven segment displays, software and hardware interrupt applications, Intel 8251 USART architecture and interfacing, Intel 8237a DMA controller, stepper motor, A/D and D/A converters, Need for 8259 programmable interrupt controllers.

UNIT IV

Microcontroller : Architecture of 8051 – Special Function Registers(SFRs) - I/O Pins Ports and Circuits - Instruction set - Addressing modes - Assembly language programming.

UNIT V

Interfacing Microcontroller:-Programming8051Timers-SerialPort Programming-Interrupts Programming–LCD & Keyboard Interfacing- ADC, DAC & Sensor Interfacing - External Memory Interface- Stepper Motor and Waveform generation - Comparison of Microprocessor, Microcontroller, PIC and ARM processors

Textbooks:

1. Microprocessors and Interfacing–Programming and Hardware by Douglas V Hall, SSSP Rao, Tata McGraw Hill Education Private Limited, 3rdEdition,1994.
2. K M Bhurchandi, A K Ray, Advanced Microprocessors and Peripherals, 3rd edition, McGraw Hill Education, 2017.
3. RajKamal,Microcontrollers:Architecture,Programming,Interfacingand System Design, 2nd

edition, Pearson, 2012.

References:

1. Ramesh S Gaonkar, Microprocessor Architecture Programming and Applications with the 8085, 6th edition, Penram International Publishing, 2013.
2. Kenneth J.Ayala, The 8051Microcontroller, 3rdedition,Cengage Learning,2004.

UNIT I
8086 ARCHITECTURE

8086 Microprocessor Family:-

Processor	Year	Architecture	Data Bus	Address Bus	Max Addressable Memory	Clock Speed	Key Features
8086	1978	16-bit	16-bit	20-bit	1 MB	5 – 10 MHz	First x86, 6-byte queue, min/max mode
8088	1979	16-bit (internal)	8-bit	20-bit	1 MB	4.77 – 8 MHz	Used in IBM PC, 4-byte queue
80186	1982	16-bit	16-bit	20-bit	1 MB	6 – 25 MHz	Built-in peripherals (timer, DMA, etc.)
80188	1982	16-bit	8-bit	20-bit	1 MB	6 – 25 MHz	Embedded use, cost-effective 80186 version
80286	1982	16-bit	16-bit	24-bit	16 MB	6 – 25 MHz	Protected mode, multitasking
80386	1985	32-bit	32-bit	32-bit	4 GB	12 – 40 MHz	Virtual memory, paging, VM86 mode
80486	1989	32-bit	32-bit	32-bit	4 GB	20 – 100 MHz	Integrated FPU, 5-stage pipelining
Pentium	1993	32-bit	64-bit	32-bit	4 GB	60 – 300 MHz	Superscalar, MMX instructions, dual pipelines

Features of 8086-Microprocessor

The 8086 Microprocessor is a 16-bit microprocessor developed by Intel Corporation in the year of 1978, and it was marked as a milestone in the history of computing technology. With being the first x86 architecture microprocessor, it paved the path for the development of modern microprocessors. As it is a 16-bit microprocessor, it can process 16-bits of digital data in a single cycle.

The important features of the 8086-microprocessor are:-

1. 16-Bit Architecture

The 8086 microprocessor has a 16-bit architecture which means it can handle or process 16-bits of digital data simultaneously. Being an advanced version of the 8085-microprocessor, it provides a better performance and computing power.

2. 16-Bit Data Bus

The 8086-microprocessor consists of the 16-bit data bus, thus it can transfer data between the processor and memory and ports either 16-bits or 8-bits at a time.

3. 20-Bit Address Bus

It is another key feature of the 8086-microprocessor that is it comprises a 20-bit address bus. Therefore, it allows for directly accessing up to 2^{20} (1 MB) memory locations.

Each of these 1 MB memory locations is a byte, hence a 16-bit word is stored in two consecutive memory locations.

4. Registers

The 8086-microprocessor comprises fourteen 16-bit registers for various purposes such as data storage, memory addressing, and execution of instructions. These registers are categorized as follows −

- **General Purpose Registers** − These registers include AX, BX, CX, and DX.
- **Segment Registers** − They are CS, DS, SS, and ES.
- **Special Purpose Registers** − These are also called Index and Pointer Registers, and they are SP, BP, SI, DI, and IP
- **Flag Registers** − These are also known as status registers, and they include conditional flags and control flags.

5. Multiplexed Address and Data Bus

The 8086-microprocessor consists of the multiplexed address and data bus (AD0-AD15). This feature reduces the number of pins required in the microprocessor. However, it also introduces a drawback which is it slows down the data transfer.

6. Clock Speed

The 8086-microprocessor requires a single-phase clock with a 33% duty cycle for providing optimized internal timing. Different models of the 8086-microprocessor operate at different clock frequencies, as follows −

- 8086 operates at 5 MHz.
- 8086-2 operates at 8 MHz.
- 8086-1 operates at 10 MHz.

7. Operating Modes

The 8086-microprocessor has two modes of operation namely, minimum mode and maximum mode. In the minimum mode of 8086-microprocessor, only one processor is used and the MN/MX pin is connected to logic 1. The maximum mode is used in multiprocessor systems and it is achieved by connecting the MN/MX pin to logic 0.

8. Instruction Prefetching

This is another unique feature of the 8086-microprocessor. This feature allows the microprocessor to prefetch up to 6 instruction bytes from the memory and queues them to minimize the waiting time for fetching instructions and improve the execution speed of the instructions.

9. Memory Addressing

In the 8086-microprocessor, the memory is byte-addressable which means that each byte in the memory uses a unique memory address. It improves efficiency of data storage and transfer.

10. Power Supply

The 8086-microprocessor requires a power supply of +5V to operate.

11. Powerful Instruction Set

The 8086-microprocessor has a powerful instruction set with the following addressing modes:

i) Direct addressing mode

ii) Immediate addressing mode

iii) Register addressing mode

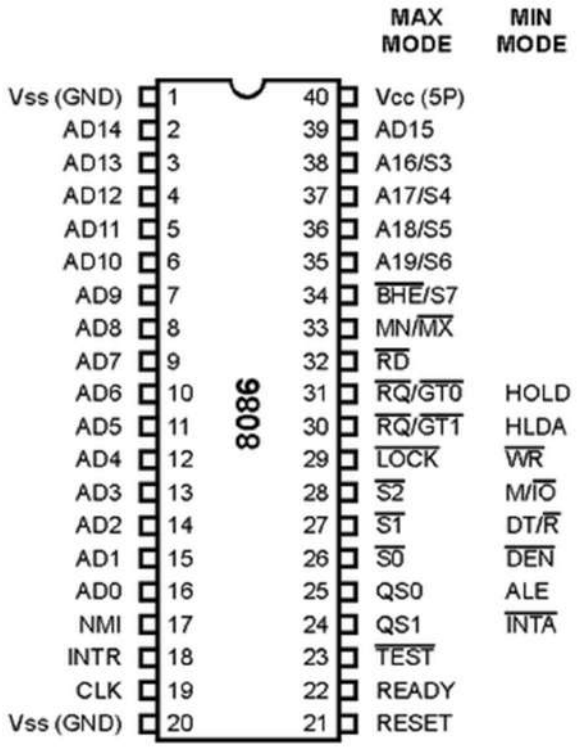
iv) Register indirect addressing mode

v) Based addressing mode

vi) Indexed addressing mode

vii) Based indexed addressing mode

8086 Pin Diagram



Power supply and frequency signals

It uses 5V DC supply at V_{CC} pin 40, and uses ground at V_{SS} pin 1 and 20 for its operation.

Clock signal

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

Address/data bus

AD0-AD15. These are 16 address/data bus. AD0-AD7 carries low order byte data and AD8AD15 carries higher order byte data. During the first clock cycle, it carries 16-bit address and after that it carries 16-bit data.

Address/status bus

A16-A19/S3-S6. These are the 4 address/status buses. During the first clock cycle, it carries 4-bit address and later it carries status signals.

A17/S4	A16/S3	Function
0	0	Extra segment access

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

S7/BHE

BHE stands for Bus High Enable. It is available at pin 34 and used to indicate the transfer of data using data bus D8-D15. This signal is low during the first clock cycle, thereafter it is active.

Read (RD)

It is available at pin 32 and is used to read signal for Read operation.

Ready

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

RESET

It is available at pin 21 and is used to restart the execution. It causes the processor to immediately terminate its present activity. This signal is active high for the first 4 clock cycles to RESET the microprocessor.

INTR

It is available at pin 18. It is an interrupt request signal, which is sampled during the last clock cycle of each instruction to determine if the processor considered this as an interrupt or not.

NMI

It stands for non-maskable interrupt and is available at pin 17. It is an edge triggered input, which causes an interrupt request to the microprocessor.

TEST

This signal is like wait state and is available at pin 23. When this signal is high, then the processor has to wait for IDLE state, else the execution continues.

MN/MX

It stands for Minimum/Maximum and is available at pin 33. It indicates what mode the processor is to operate in; when it is high, it works in the minimum mode and vice-versa.

INTA

It is an interrupt acknowledgement signal and id available at pin 24. When the microprocessor receives this signal, it acknowledges the interrupt.

ALE

It stands for address enable latch and is available at pin 25. A positive pulse is generated each time the processor begins any operation. This signal indicates the availability of a valid address on the address/data lines.

DEN

It stands for Data Enable and is available at pin 26. It is used to enable Transreceiver 8286. The transreceiver is a device used to separate data from the address/data bus.

DT/R

It stands for Data Transmit/Receive signal and is available at pin 27. It decides the direction of data flow through the transreceiver. When it is high, data is transmitted out and vice-a-versa.

M/IO

This signal is used to distinguish between memory and I/O operations. When it is high, it indicates I/O operation and when it is low indicates the memory operation. It is available at pin 28.

WR

It stands for write signal and is available at pin 29. It is used to write the data into the memory or the output device depending on the status of M/IO signal.

HLDA

It stands for Hold Acknowledgement signal and is available at pin 30. This signal acknowledges the HOLD signal.

HOLD

This signal indicates to the processor that external devices are requesting to access the address/data buses. It is available at pin 31.

QS_1 and QS_0

These are queue status signals and are available at pin 24 and 25. These signals provide the status of instruction queue. Their conditions are shown in the following table.

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

S_0, S_1, S_2

These are the status signals that provide the status of operation, which is used by the Bus Controller 8288 to generate memory & I/O control signals. These are available at pin 26, 27, and 28. Following is the table showing their status.

$\overline{S_2}$	$\overline{S_1}$	$\overline{S_0}$	Status
0	0	0	Interrupt acknowledgement
0	0	1	I/O Read
0	1	0	I/O Write
0	1	1	Halt
1	0	0	Opcode fetch
1	0	1	Memory read
1	1	0	Memory write
1	1	1	Passive

LOCK

When this signal is active, it indicates to the other processors not to ask the CPU to leave the system bus. It is activated using the LOCK prefix on any instruction and is available at pin 29.

RQ/GT_1 and RQ/GT0: These are the Request/Grant signals used by the other processors requesting the CPU to release the system bus. When the signal is received by CPU, then it sends acknowledgment. RQ/GT_0 has a higher priority than RQ/GT_1.

Architecture of 8086

The 8086 Microprocessor is a 16-bit microprocessor developed by Intel Corporation in the year of 1978. The architecture of the 8086 microprocessor is based on a complex instruction set computer (CISC) architecture, which means that it supports a wide range of instructions, many of which can perform multiple operations in a single instruction. The 8086 microprocessor has a 20-bit address bus, which can address up to 1 MB of memory, and a 16-bit data bus, which can transfer data between the microprocessor and memory or I/O devices.

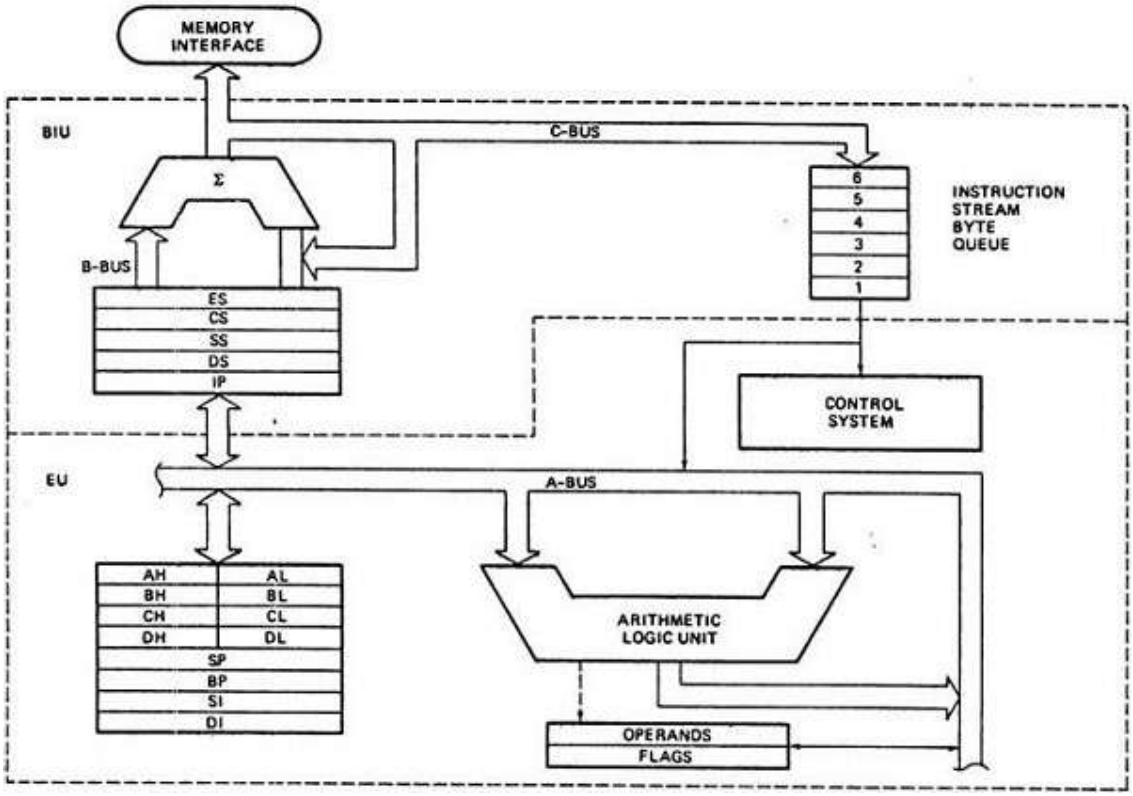


Fig: 8086 Architecture

The internal architecture of Intel 8086 is divided into 2 units: **The Bus Interface Unit (BIU)**, and **The Execution Unit (EU)**.

1. The Bus Interface Unit (BIU):

It provides the interface of 8086 to external memory and I/O devices via the System Bus. The following functions are performed by BIU.

- ✓ It generates the 20-bit physical address for memory access.
- ✓ It fetches instructions from the memory.
- ✓ It transfers data to and from the memory and I/O.
- ✓ Maintains the 6-byte pre-fetch instruction queue.

BIU mainly contains the **4 Segment registers**, the **Instruction Pointer**, a pre-fetch queue and an **Address Generation Circuit**.

Code Segment register: (16 Bit register): CS holds the base address for the Code Segment. All programs are stored in the Code Segment and accessed via the IP.

Data Segment register: (16 Bit register): DS holds the base address for the Data Segment.
Stack Segment register: (16 Bit register): SS holds the base address for the Stack Segment.
Extra Segment register: (16 Bit register): ES holds the base address for the Extra Segment.

Instruction Pointer (IP):

- ✓ It is a 16-bit register. This register is used to hold the address of the next instruction which is to be executed.
- ✓ IP is incremented after every instruction byte is fetched.
- ✓ CS is multiplied by 10H to give the 20-bit physical address of the Code Segment.
- ✓ The address of the next instruction is calculated by using the formula CS x 10H + IP.

Example:

If CS = 4321H, IP = 1000H then CS x 10H = 43210H + offset = 44210H
Here Offset = Instruction Pointer (IP). This is the address of the next instruction.

6 Byte Pre-fetch Queue:

- ✓ It is a 6-byte queue (FIFO).
- ✓ Fetching the next instruction (by BIU from CS) while executing the current instruction is called pipelining.
- ✓ Gets flushed whenever a branch instruction occurs.
- ✓ The pre-Fetch queue is of 6-Bytes only because the maximum size of instruction that can have in 8086 is 6 bytes. Hence to cover up all operands and data fields of maximum size instruction in 8086 Microprocessor there is a Pre-Fetch queue is 6 Bytes.
- ✓ The pre-Fetch queue is connected with the control unit which is responsible for decoding op-code and operands and telling the execution unit what to do with the help of timing and control signals.

Address Generation Circuit:

- ✓ The BIU has a Physical Address Generation Circuit.
- ✓ It generates the 20-bit physical address using Segment and Offset addresses using the formula: **Physical Address = Segment Address * 10H + Offset Address**
- ✓ In Bus Interface Unit (BIU) the circuit shown by the Σ symbol is responsible for the calculation unit which is used to calculate the physical address of an instruction in memory.

2. The Execution Unit (EU): The main components of the EU are General purpose registers, the ALU, Special purpose registers, the Instruction Register and Instruction Decoder, and the Flag/Status Register.

- ✓ Fetches instructions from the Queue in BIU, decodes, and executes arithmetic and logic operations using the ALU.
- ✓ Sends control signals for internal data transfer operations within the microprocessor. (Control Unit)
- ✓ Sends request signals to the BIU to access the external module.

General Purpose Registers:

8086 has four 16-bit general purpose registers AX, BX, CX, and DX which store intermediate values during execution.

- ✓ AX register: (Combination of AL and AH Registers)
  It is used to store operands and result of ALU operations.
- ✓ BX register: (Combination of BL and BH Registers)
  It is used for addressing memory (store memory location addresses) and storing data.
- ✓ CX register: (Combination of CL and CH Registers)
  It holds the count for instructions like a loop, rotate, shift and string operations.
- ✓ DX register: (Combination of DL and DH Registers)
  It is used for data storage and during the multiplication and division operations, if the result is of 32-bits, then 16 bits from MSB is stored in DX register and bits from LSB is stored in AX register. It is also used for input/output port addressing.

Arithmetic Logic Unit (16-bit): Performs 8 and 16-bit arithmetic and logic operations.

Special purpose registers (16-bit): Special purpose registers points to specific memory locations under each segment.

- Stack Pointer (SP): Stack Pointer register is used to hold the topmost address of the stack memory, i.e., it stores the address of the memory location in which data was recently stored.
- Base Pointer (BP): BP register acts as a memory pointer to the stack segment register. The BP register is mainly used to access any location directly in the stack.
- Source Index (SI): It act as a memory pointer for source data.
- Destination Index (DI): It act as a memory pointer for destination data.

Instruction Register and Instruction Decoder: The EU fetches an opcode from the queue into the instruction register. The instruction decoder decodes it and sends the information to the control circuit for execution.

Flag/Status register (16 bits): It has 9 flags that help change or recognize the state of the microprocessor.

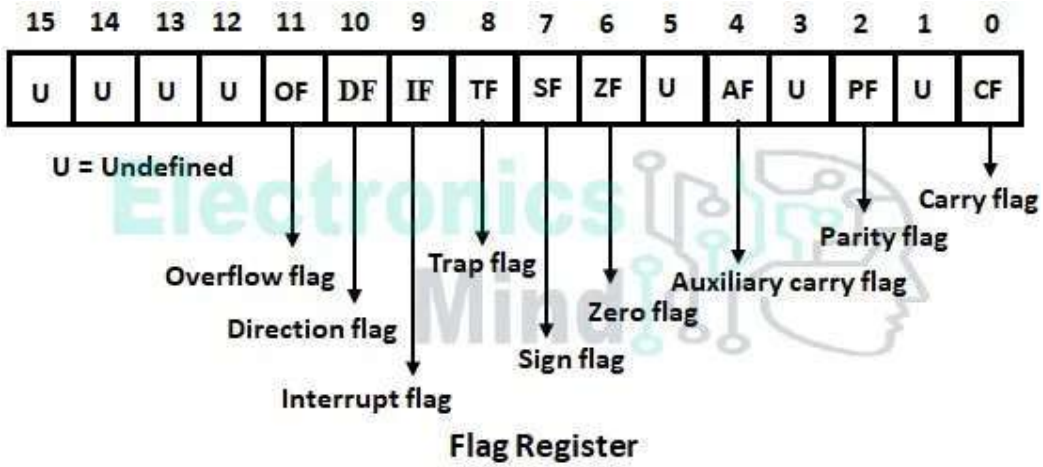
6 Status flags: Status flags are updated after every arithmetic and logic operation.

- Carry flag (CF)
- Parity flag (PF)
- Auxiliary carry flag (AF)
- Zero flag(Z)
- Sign flag(S)
- Overflow flag (O)

3 Control flags:

- Trap flag (TF)
- Interrupt flag (IF)

- Direction flag (DF)



- Carry flag − This flag indicates an overflow condition for arithmetic operations.
- Auxiliary flag − When an operation is performed at ALU, it results in a carry/borrow then this flag is set.
- Parity flag − This flag is used to indicate the parity of the result, i.e. when the lower order 8-bits of the result contains even number of 1s, then the Parity Flag is set. For odd number of 1s, the Parity Flag is reset.
- Zero flag − This flag is set to 1 when the result of arithmetic or logical operation is zero else it is set to 0.
- Sign flag − This flag holds the sign of the result, i.e. when the result of the operation is negative, then the sign flag is set to 1 else set to 0.
- Overflow flag − This flag represents the result when the system capacity is exceeded.
- Trap flag − It is used for single step control and allows the user to execute one instruction at a time for debugging. If it is set, then the program can be run in a single step mode.
- Interrupt flag − It is an interrupt enable/disable flag, i.e. used to allow/prohibit the interruption of a program. It is set to 1 for interrupt enabled condition and set to 0 for interrupt disabled condition.
- Direction flag − It is used in string operation. As the name suggests when it is set then string bytes are accessed from the higher memory address to the lower memory address and vice-a-versa.

3. Decode unit:

The Decode Unit works in parallel with the Prefetch Unit, which fetches instructions from memory and stores them in a queue. The Decode Unit reads the instructions from the queue and translates them into micro-operations that can be executed by the microprocessor.

4. Control Unit:

The Control Unit in the 8086 microprocessor is a component that manages the overall operation of the microprocessor. The control unit is responsible for controlling the flow of instructions through the microprocessor and coordinating the activities of the other components, including the Decode Unit, Execution Unit, and Prefetch Unit.

There are two modes in which a microprocessor 8086 can be operated for each of the mode, timing diagram can be categorized into two. They are read cycle and write cycle.

Let us see the minimum mode configuration system and it's timing diagrams first.

Minimum mode Configuration of 8086 micro processor: (Single processor mode)

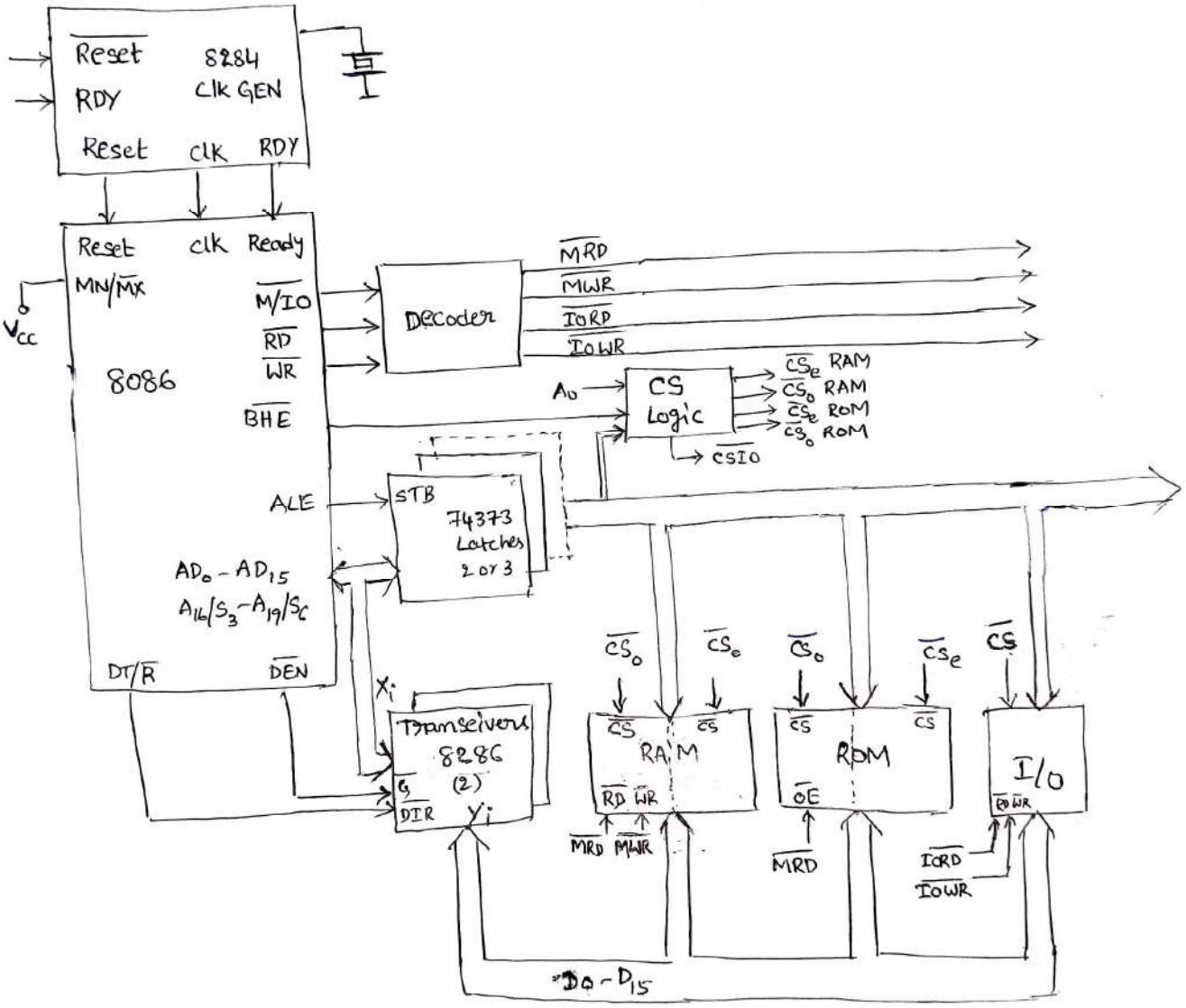


fig: Minimum mode configuration of 8086 system

The microprocessor 8086 can be operated in minimum mode by connecting MN/$\overline{MX}$ pin to V_{cc} i.e at logic 1. In this mode all the control signals are generated by microprocessor itself. In this minimum mode we have a single microprocessor. The remaining components in the system are latches, transceivers, clock generator,

memory and I/O devices. Some type of chip selection logic may also be required for selecting memory or I/O devices.

The latches are generally D-Flipflops like 74LS373 or 8282. They are used for seperating the valid address from the multiplexed address/data signals and are controlled by 'ALE' signal of 8086.

The transceivers are the bidirectional buffers which are required to seperate the valid data from the multiplexed address/data signal. The transceivers are controlled by two signals namely $\overline{DEN}$ and $DT/\overline{R}$. The $\overline{DEN}$ indicates that the valid data is available on the data bus while $DT/\overline{R}$ indicates the direction of data ie from/to the microprocessor.

The system contains RAM and ROM. It may also contain I/O devices for communication with the processor.

The clock generator IC 8284 generates the clock from the crystal oscillator which is used to provide an accurate timing reference for the system. The operation of minimum mode is explained interms of timining diagrams for better understanding purpose.

i) Minimum mode read cycle:

The read cycle begins during T_1 state with ALE signal being at logic '1' for one clock cycle. on the trailing edge of ALE, the address information is latched and is available at the output of latches.

$\overline{BHE}$ and A_0 signals address lower byte or higher byte or whole word of the data to be accessed.

If it is a memory read the $M/\overline{IO}$ signal is at logic '1' from T_1 to T_4. If it is I/O read, $M/\overline{IO}$ signal is at logic '0'. Since it is read cycle $DT/\overline{R}$ should be at logic '0' from T_1 to T_4 indicating that microprocessor is receiving data.

At 'T_2', the address is removed from the AD_0 to AD_{15} and the bus is tristated. The address information from A_{16}/S_3 to A_{19}/S_6 is also removed and these lines now carry the status information.

In T_2 the $\overline{RD}$ signal is made low. when $\overline{RD}$ signal goes low,

the bus $AD_0 - AD_{15}$ comes out of tristate and valid data is available on the data bus.

CS logic is the chip select logic and the suffixes 'e', 'o' indicate even and odd address memory banks.

In T_2 $\overline{DEN}$ is kept at logic '0' to enable the data bus.

During T_2 when $\overline{RD}$ is low, the bringing of data from memory or I/O onto the data bus is done till the end of T_3. At the end of T_3, the data is available on the data bus which can be transferred to 8086 till T_4 state.

In T_4 all the signals are de-activated in preparation for the next bus cycle. Hence this machine cycle ends with T_4.

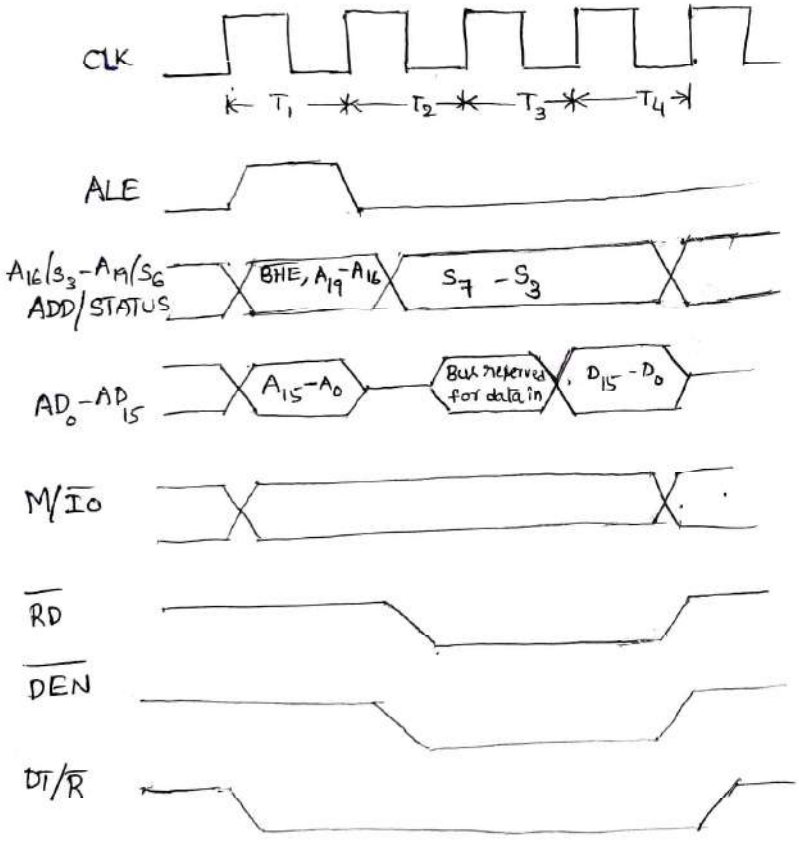


fig: Read cycle Timing diagram for minimum mode of 8086.

ii) Minimum mode write cycle:

The write cycle also begins in T_1 with the ALE signal being at logic '1' during which the address location is placed on the address bus. The $M/\overline{IO}$ signal is again asserted to indicate a memory or I/O operation. The $\overline{WR}$ signal becomes low at the

beginning of the T_2-state. Hence the processor sends the data to be written to the addressed location that was appeared during 'T_1'. The data remains on the data bus till the middle of 'T_4' state.

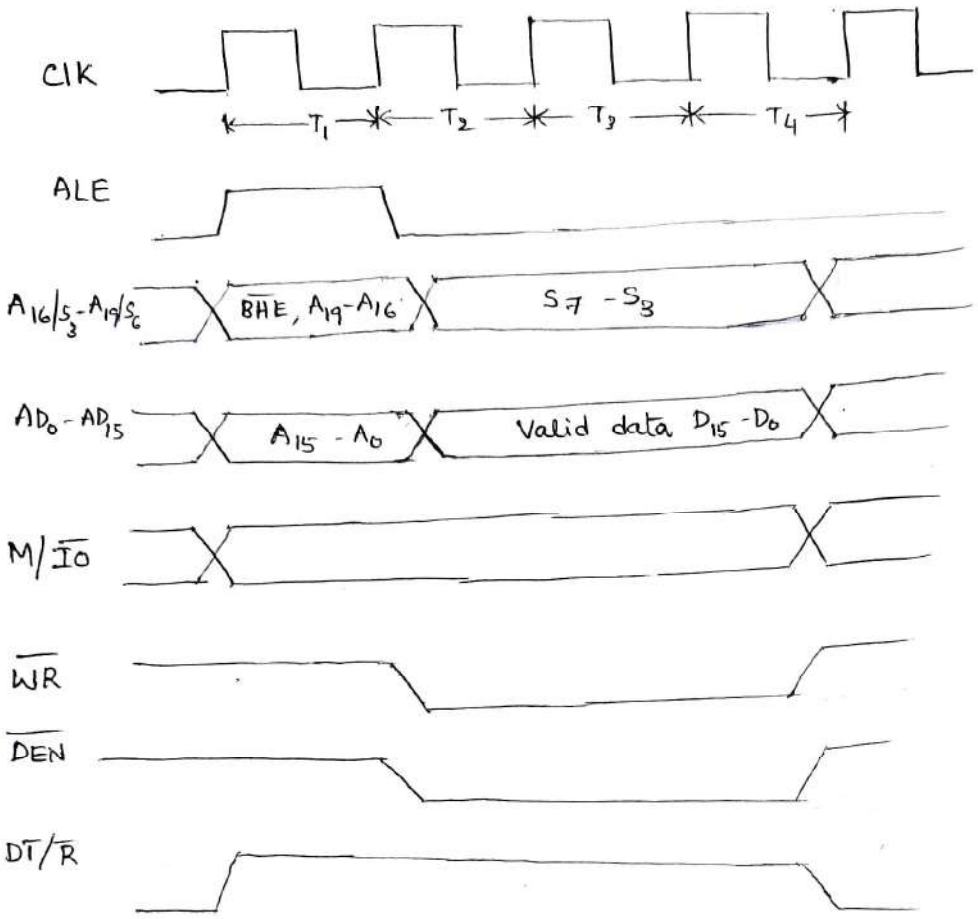


fig: Write cycle Timing Diagram for minimum mode

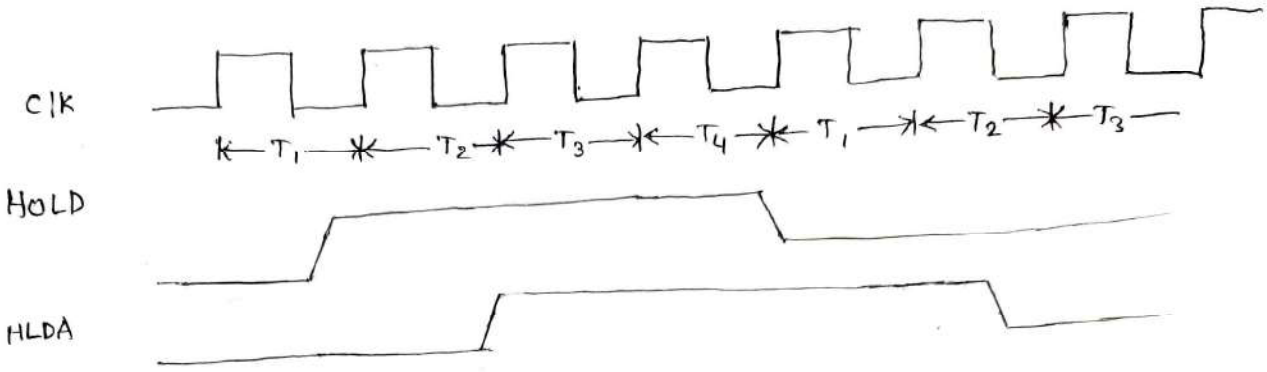
The $\overline{BHE}$, A_0 signals are used to select proper byte (either higher byte or lower byte) or bytes (the whole word) to be read or written.

$M/\overline{IO}$, $\overline{RD}$, and $\overline{WR}$ signals indicate the type of data transfer i.e to or from the microprocessor.

HOLD and HLDA in the timing diagram:

The HOLD and HLDA pins of the 8086 are the bus request and bus grant signals of the minimum mode. The HOLD pin is checked at the end of each bus cycle. If it is received active by the processor before T_4 of the previous cycle or during T_1 of the current cycle, the CPU activates HLDA in the next clock cycle and for the

succeeding bus cycles, the bus will be given to another requesting master. The processor cannot gain it's control on the bus until the requesting master doesnot drop the HOLD pin low. when the request is dropped by the requesting master, the HLDA is dropped by the processor at the trailing edge of the next clock cycle as shown in below.



Maximum mode configuration of 8086 micro processor: (Multiprocessor mode)

The Microprocessor 8086 can be operated in maximum mode by connecting MN/$\overline{MX}$ pin to ground. In this mode an another chip called bus controller is used to derive the control signals using the status signals $\overline{S_2}$, $\overline{S_1}$ and $\overline{S_0}$ of micro processor. The other components of the system in maximum mode is same as in the minimum mode system.

The basic functions of the bus controller chip IC 8288 is to derive control signals like $\overline{RD}$ and $\overline{WR}$, $\overline{DEN}$, DT/$\overline{R}$, ALE etc using $\overline{S_2}$, $\overline{S_1}$ and $\overline{S_0}$. The bus controller chip has input lines $\overline{S_2}$, $\overline{S_1}$ and $\overline{S_0}$ and CLK which are driven by the microprocessor. The outputs of the bus controller are ALE, $\overline{DEN}$, DT/$\overline{R}$, $\overline{MRDC}$, $\overline{MWTC}$, $\overline{AMWC}$, $\overline{IORC}$, $\overline{IOWC}$ and $\overline{AIOWC}$.

$\overline{IORC}$, $\overline{IOWC}$ are I/O read command and I/O write command signals respecively. These signals are used to perform read or write the data from or to the I/O port. $\overline{MRDC}$, $\overline{MWTC}$ are memory read command and memory write command signals respectively and may be used as memory read and memory write signals. For both the write command signals ($\overline{IOWC}$ and $\overline{MWTC}$) the advanced signals namely $\overline{AIOWC}$ and $\overline{AMWTC}$ are available. They are also used for the same purpose but are activated one clock cycle earlier than $\overline{IOWC}$ and $\overline{MWTC}$ signals respectively.

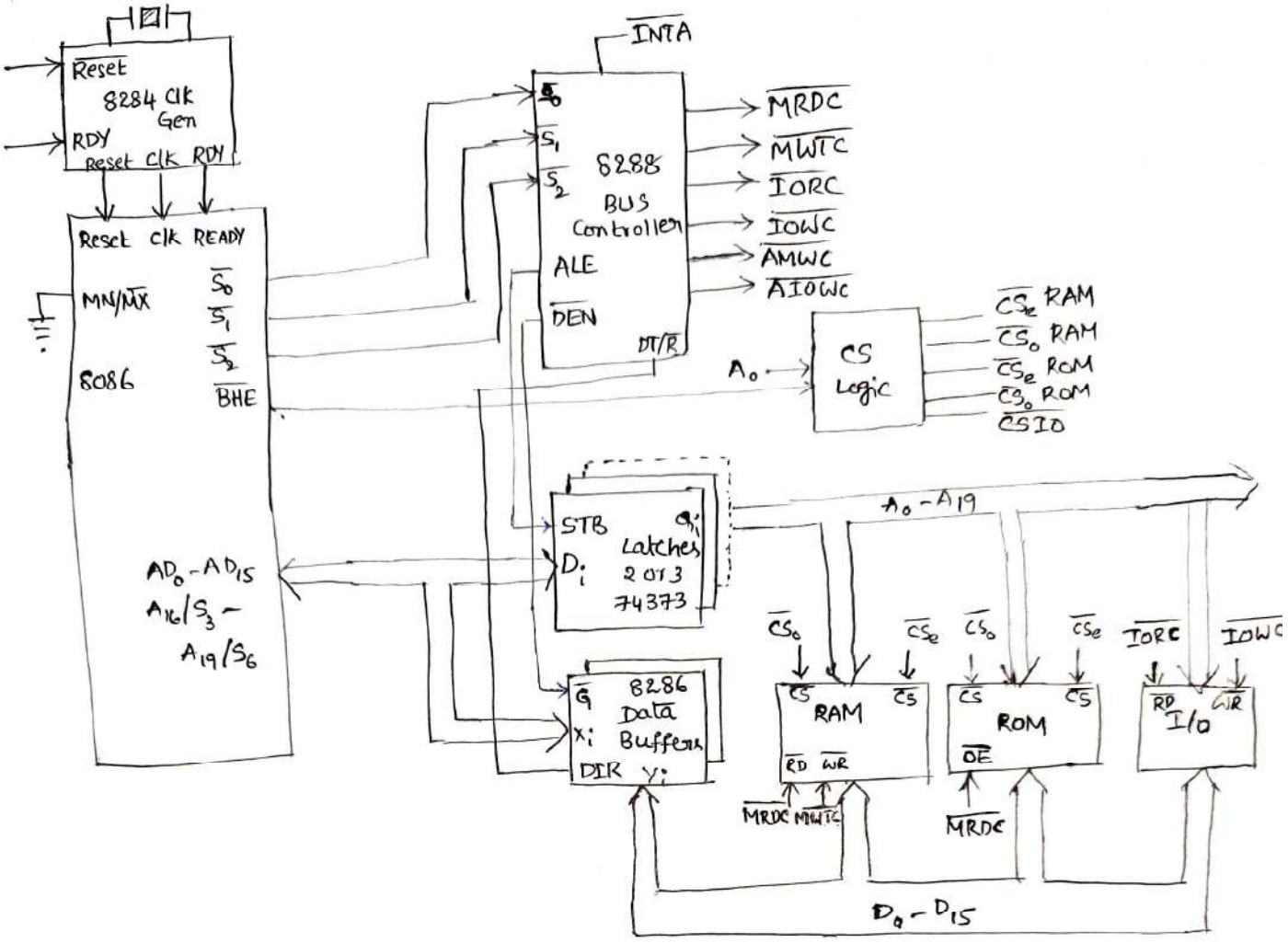


fig:- Maximum mode Configuration of 8086

The system Configuration of 8086 maximum mode is shown in the above figure.

Timing diagrams for maximum mode:

The timing diagrams for maximum mode are divided into two parts read cycle and write cycle. The timing diagrams for maximum mode is similar to that of minimum mode. The only difference is the status signals $\overline{S_2}, \overline{S_1}, \overline{S_0}$ are used here and also the command signals $\overline{MRDC}$, $\overline{MWTC}$, $\overline{IORC}$, $\overline{IOWC}$. In the case of write cycle of maximum mode we have some advanced signals such as $\overline{AMWC}$ or $\overline{AIOWC}$ for memory and I/O respectively. The advanced write command signals are same as that of $\overline{MWTC}$ or $\overline{IOWC}$. The only difference is the advanced write commands $\overline{AMWC}$ and $\overline{AIOWC}$ are made active low one clock cycle earlier than $\overline{MWTC}$ and $\overline{IOWC}$

respectively. The timing diagrams for read cycle of maximum mode are given below.

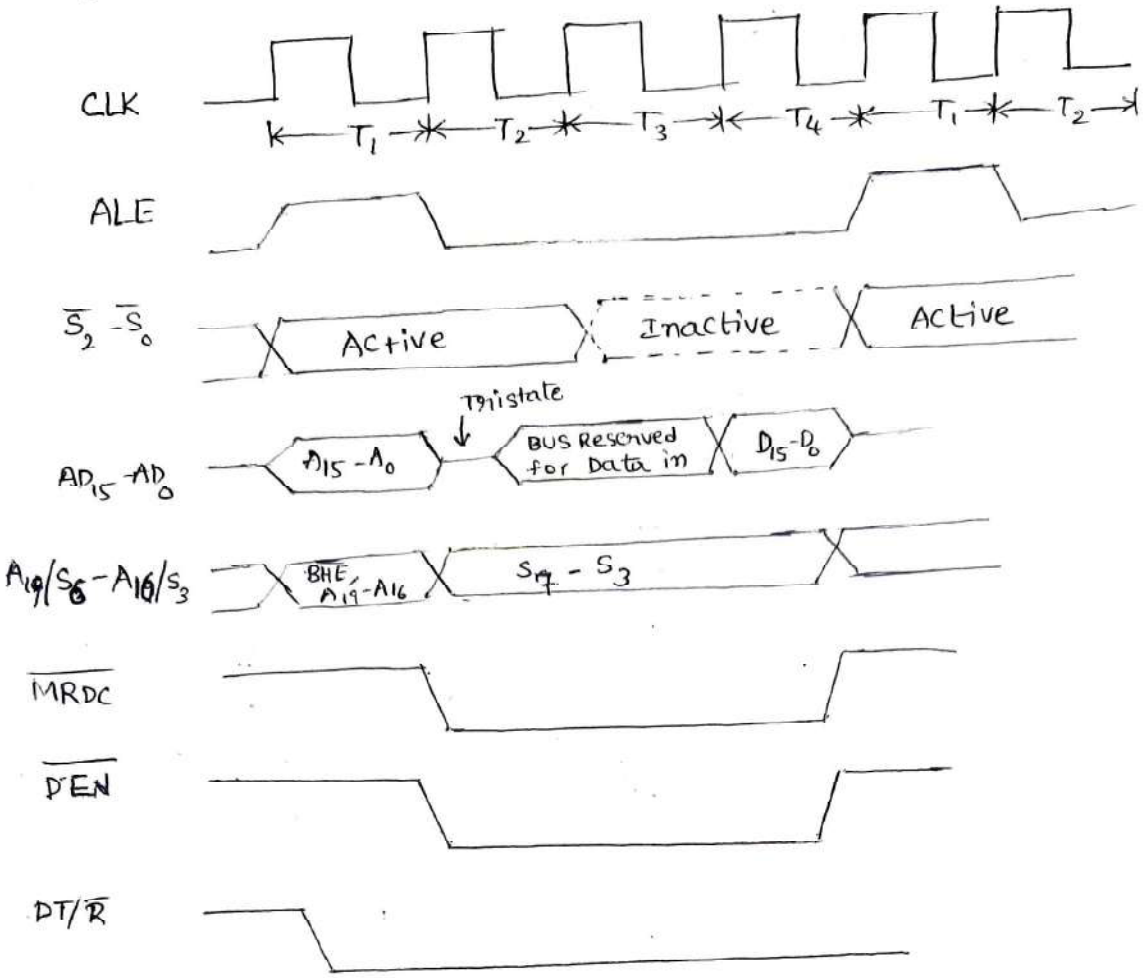


fig:- Memory read Timing diagram in maximum mode

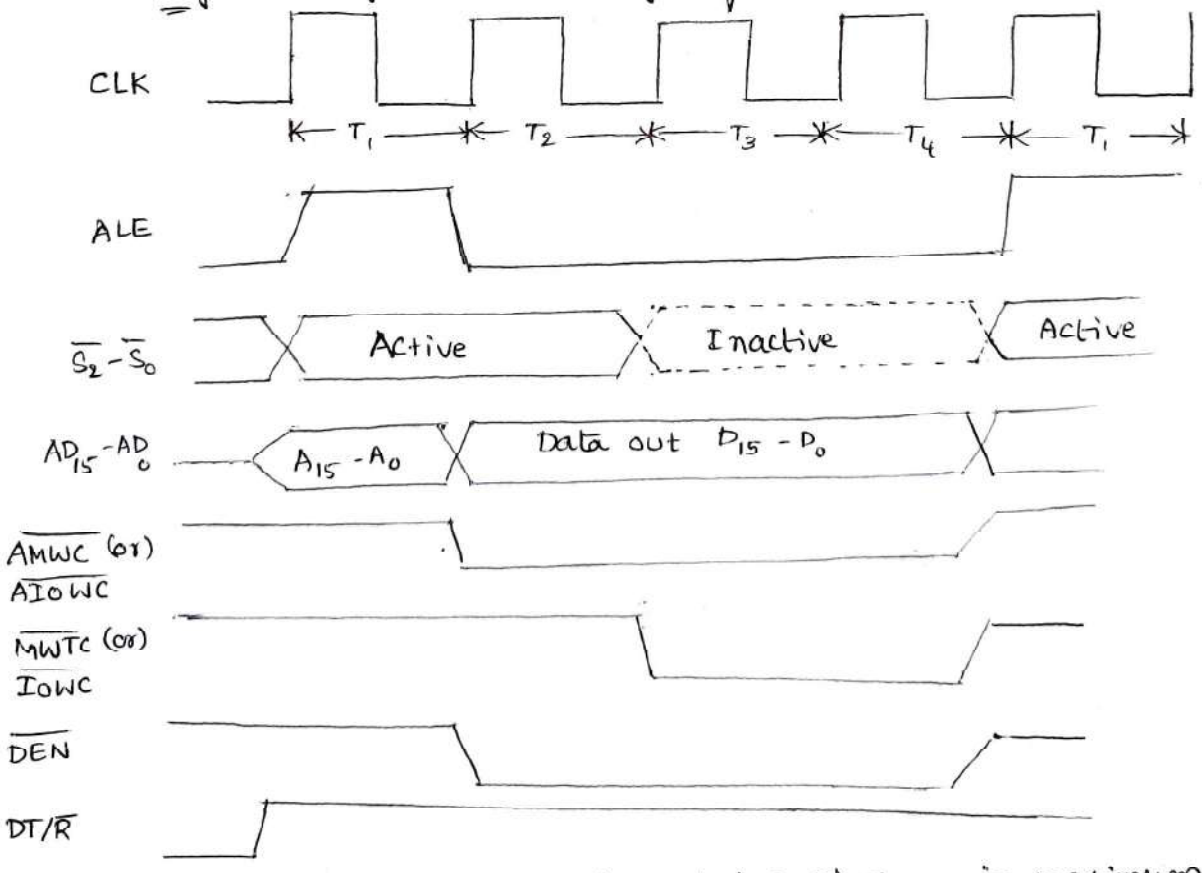


fig:- Memory write Timing Diagram in maximum mode.

State (or) T_1 of the next machine cycle. When the requesting master receives the grant pulse, the master gets the control over the bus. After accessing the bus it sends a release pulse to the processor that indicates the bus is released to the processor.

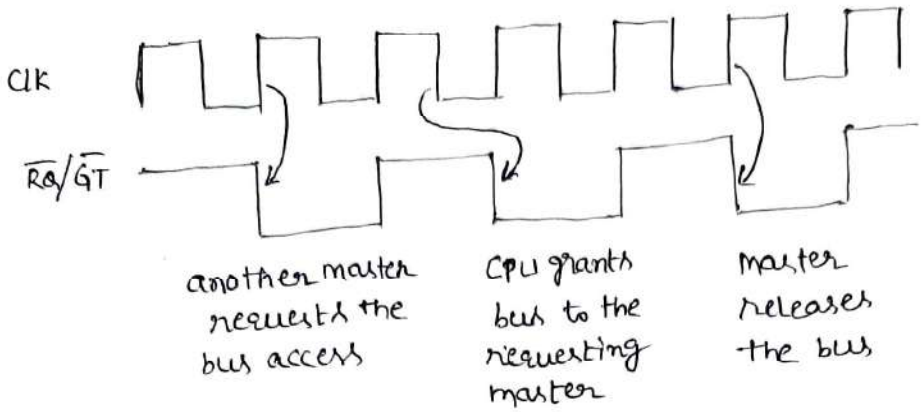


fig:- $\overline{RQ}/\overline{GT}$ Timings in maximum mode.

Interrupt structure of 8086 :

The signal that causes a break in the execution of a program to the microprocessor is known as an interrupt.

The interrupt for 8086 can occur in any one of the following ways.

i) An external signal applied to the non maskable Interrupt (NMI) pin (or) INTR pin of 8086. The interrupt caused by the signal applied to any one of these pins is called hardware interrupt.

ii) The execution of an instruction of the format 'INT n', where 'n' is the type of the interrupt that can be any value between 00 to FF H. This is called software interrupt.

iii) An error condition such as divide by '0' which is produced in 8086 by the execution of the DIV/IDIV Instruction.

iv) A trap interrupt.

Processing of Interrupt by 8086:

After executing each instruction in a program, the 8086 checks if any interrupt has been requested. If an interrupt has been requested the 8086 processes it by performing the following series of steps.

i) Pushes the contents of the flag register onto the stack to preserve the status of the interrupt flag (IF) and trap flag (TF) by decrementing the stack pointer (SP) by 2.

ii) Disables the INTR interrupt by clearing IF in the flag register.

iii) Disables the Trap interrupt by clearing TF in the flag register

iv) Pushes the content of the code segment (CS) register onto the stack by decrementing the stack pointer (SP) by 2.

v) Pushes the content of the Instruction pointer (IP) onto the stack by decrementing SP by '2'.

vi) Transfers the control to a location at which the Interrupt service routine (ISR) is stored by loading CS and IP with ISR starting address.

vii) Executes the ISR. The last instruction in the ISR will be IRET.

viii) Then IP is popped off the stack

ix) The CS is popped off the stack.

x) The flag register is popped off the stack.

xi) The control returns to the point where it had been diverted.

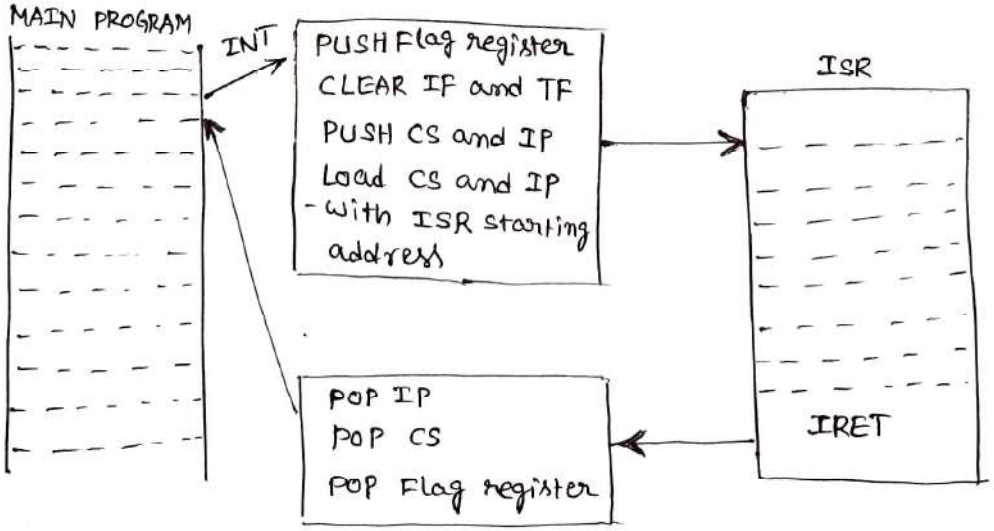


fig: Steps of processing an interrupt request.

Interrupt service Routine (ISR):

when an interrupt occurs, the processor suspends the execution of it's current program and executes another program which is related to the interrupt. This program or routine is known as Interrupt service Routine (ISR).

Interrupt vector:

The address of an ISR is called as interrupt vector of the corresponding interrupt. For an 8086 based system, the address of any program is in the format CS:IP. Thus the interrupt vector for any interrupt has 4 bytes. Two for the CS value and the remaining two for IP value. Thus if the interrupt vector for a particular interrupt is obtained, the control will be transferred to the new location by using the new values of CS and IP specified by the interrupt vector.

Interrupt vector table:

The number of interrupt vectors in a system is equal to the number of interrupts in the system. The 8086 has 256 interrupts. So there are 256 interrupt vectors in 8086. Each interrupt vector requires 4 bytes of locations, it implies that $256 \times 4 = 1024$ bytes = 1KB of memory is allocated to store the interrupt vectors. These 256 interrupt vectors are stored in a table called interrupt vector table (IVT) in the system RAM from the locations 00000H to 003FFH.

The interrupt vector is numbered from 00H to FFH, having the same number as that of the type of the interrupt.

The first entry in the IVT is INT 00H, INT 01H is the second entry, INT 02H is the third entry and so on.

Since each interrupt vector is of 4 bytes long, the memory location (or) the address of the interrupt vector is obtained for 'INT n' by multiplying the type value with 4. For Example

for INT 01H, the vector address is 01×4 = 00004H. In the four consecutive memory locations from 00004H (i.e 00004H, 00005H, 00006H, 00007H), the CS and IP address of ISR are available. i.e in 00007H, 00006H the higher and lower bytes of CS of the ISR are present and in 00005H, 00004H the higher and lower bytes of the IP address are present respectively. The CS and IP registers are loaded with these values and hence the control is transfered to the ISR location.

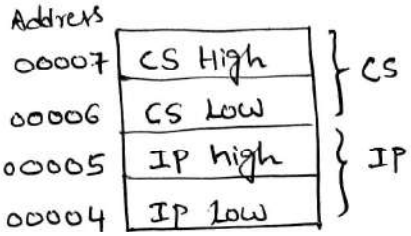


fig: Interrupt vector of INT 01H.

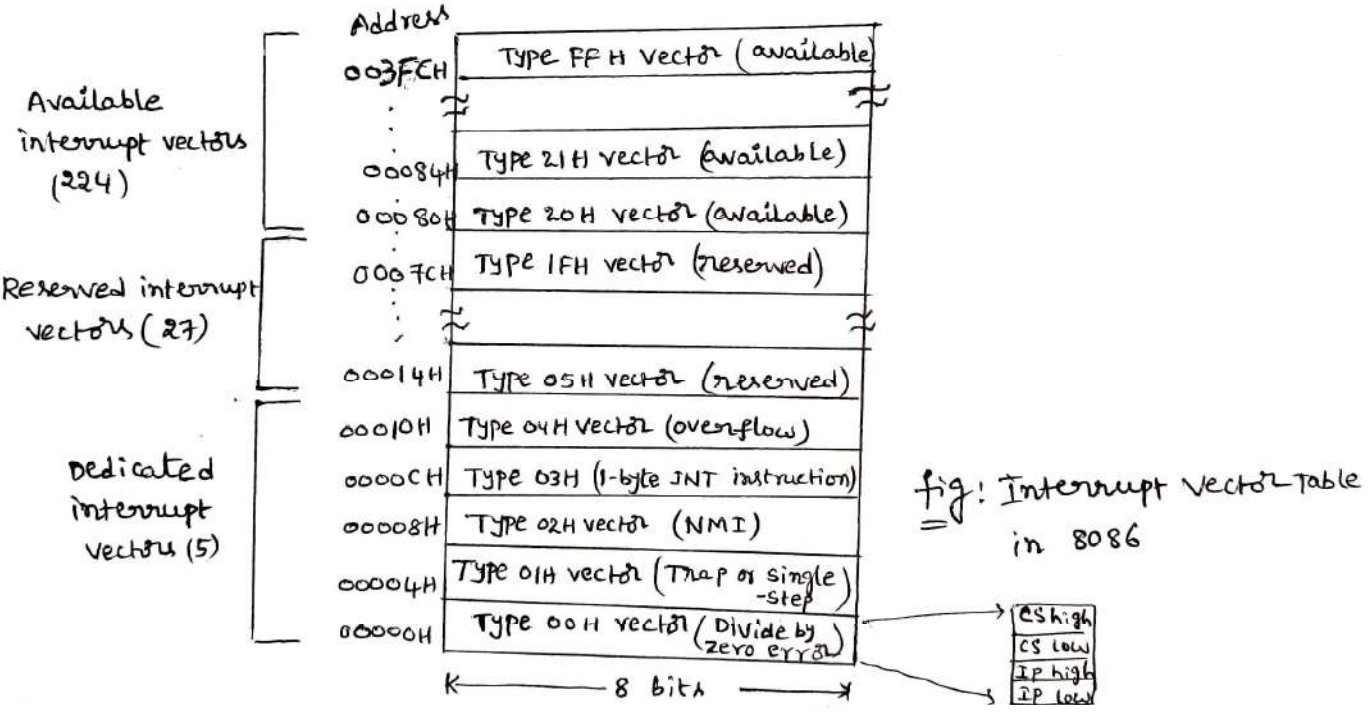


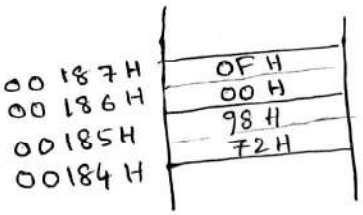
fig: Interrupt vector Table in 8086

problem: Find the address in the IVT of the interrupt vector of INT 61H. Find the physical address of the ISR corresponding to this interrupt if the vector is 0F00:9872H

Sol) The type number of the interrupt = 61H

The address of the interrupt vector = 61H × 4 = 184H

Given the CS of ISR = 0F00H, IP of ISR = 9872H



The base address of Code segment of ISR = 0F00 X10H
= 0F000 H

The offset address (IP) of ISR = 9872H

∴ The physical address of ISR = 0F000 + 9872H
= 18872H

NOTE: The Interrupt vector table (IVT) is available in the microprocessor 8086 memory locations from 00000H to 003FFH. i.e the first 1KB of memory of 8086 is used to store the IVT.

Dedicated Interrupt types in 8086:

The lowest five interrupt types in 8086 i.e Type 00H - 04H are dedicated interrupt types.

Type 00H or Divide by '0' interrupt:

when ever the quotient from division operation (DIV (or) I DIV) is too large to fit in the resultant register, which occurs while dividing a number by '0' or if the divisor is very small compared to the dividend, the 8086 automatically generates a type 00 interrupt.

Type 01 H or single step or trap interrupt:

The type 1 interrupt is used for single step operation, in which the 8086 executes one instruction in the main program and then executes the ISR of trap interrupt. In this ISR, we write the instructions to verify the contents of certain registers and memory locations and display them in an output device such as 7-segment display or monitor. If the expected data is present in the registers (or) memory locations, the 8086 can be allowed to execute the next instruction of the main program. After executing each instruction of the main program the ISR of trap interrupt is executed. If the trap flag in the 8086 is set, the 8086 automatically generates a type 1 interrupt after each instruction in the main

program is executed. After executing the IRET instruction in the ISR, the 8086 again goes to execute the next instruction in the main program.

Type 02 H (or) NMI interrupt:

The 8086 generates type 2 interrupt automatically when it receives logic '1' on NMI pin. The NMI interrupt can not be disabled by using software. this interrupt is also called as a hardware interrupt. NMI interrupt is a type 02 interrupt hence the ISR related to NMI interrupt must be written in the address pointed by interrupt vector of type 02.

NMI interrupt is used to sense hazardous situations such as fire, smoke and unsafe temperature or pressure limits in an industrial environment, when the 8086 is used in the industrial applications. In these applications, an appropriate sensor is used to detect the abnormal condition and it's output is connected to the NMI interrupt. when ever NMI interrupt is activated, the 8086 runs the ISR of NMI interrupt which is used to issue an alarm signal and shut down the system if needed.

Type 03 H (or) one-byte instruction interrupt (or) break point interrupt:

The Type 03 instruction is produced by the execution of INT 03H instruction. It is a single byte instruction which is very useful for debugging a program. when we insert a break point interrupt in the program, the 8086 executes the instructions upto the break point interrupt. and then executes the ISR corresponding to the break point interrupt. Unlike single step technique in which the execution is stopped after each instruction, the break point technique allows us to execute all the instructions upto the break point interrupt.

The opcode of the INT 03H instruction is CC H.

Type 04 H (or) overflow interrupt:

The 8086 overflow flag (OF) is set if the result of an arithmetic operation on signed binary numbers is too large to be stored in the destination register. If the overflow flag is set, the overflow

interrupt occurs but not automatically. An instruction INTO must be written after the instruction at which we expect the overflow flag may set to '1'.

```
MOV AL, NUM1
MOV BL, NUM2
ADD AL, BL
INTO        ; interrupt on overflow
```

The last line of this program (INTO) can pass the control to the ISR written for INT 04H if the overflow flag is set by the result of addition. otherwise INTO acts as a NOP instruction

Software interrupts: (Types 00H - FFH)

The interrupt that is generated by writing an instruction of the format 'INT n' is called as the software interrupt. Since there are 256 possible interrupt types for 8086, the value of 'n' can be any one value between '00H - FFH'.

In general when 8086 executes the INT n instruction, where n is the type value of the interrupt, the 8086 pushes the content of the flag register, CS, and IP values into the stack and clears the flags IF and TF. Then the 8086 goes to the memory address given by 4xn to obtain the interrupt vector for the type 'n' from the IVT and loads in the IP, CS registers. This makes the control of the processor to go to the ISR location and the execution of ISR is performed. At the end of the ISR the 'IRET' instruction will be there which causes the 8086 to pop the flag register content, CS, IP contents from the stack and to load them in the flag register, CS register and IP register respectively. Hence the 8086 returns to the main program where it had been interrupted earlier and continues executing main program.

By using this software interrupts we can execute various ISR's. By executing the instruction INT 02H we can execute the ISR of NMI without giving any input to NMI pin.

INTR interrupts :

The 8086 INTR interrupt allows an external signal to interrupt the execution of a program. The INTR interrupt can be disabled, so as to not cause an interrupt. If IF is set, the INTR interrupt is enabled and if IF is cleared the INTR interrupt is disabled.

The INTR interrupt is activated by a logic '1' signal applied at the INTR Pin. To set or clear the IF we use the instructions STI and CLI respectively.

If the INTR pin is at logic high and the Interrupt Flag (IF) is set [i.e IF = 1], then the 8086 is interrupted and the ISR related to INTR interrupt is executed.

To run the ISR related to the INTR interrupt, an external hardware such as Programmable Interrupt Controller (8259) is to be interfaced with 8086.

When the 8259 receives an interrupt signal on one of its IR inputs ($IR_0 - IR_7$), it sends an interrupt signal to the 'INTR' pin of the 8086 through a pin INT. If the INTR interrupt is enabled by setting IF, the 8086 responds as shown in below timing diagram.

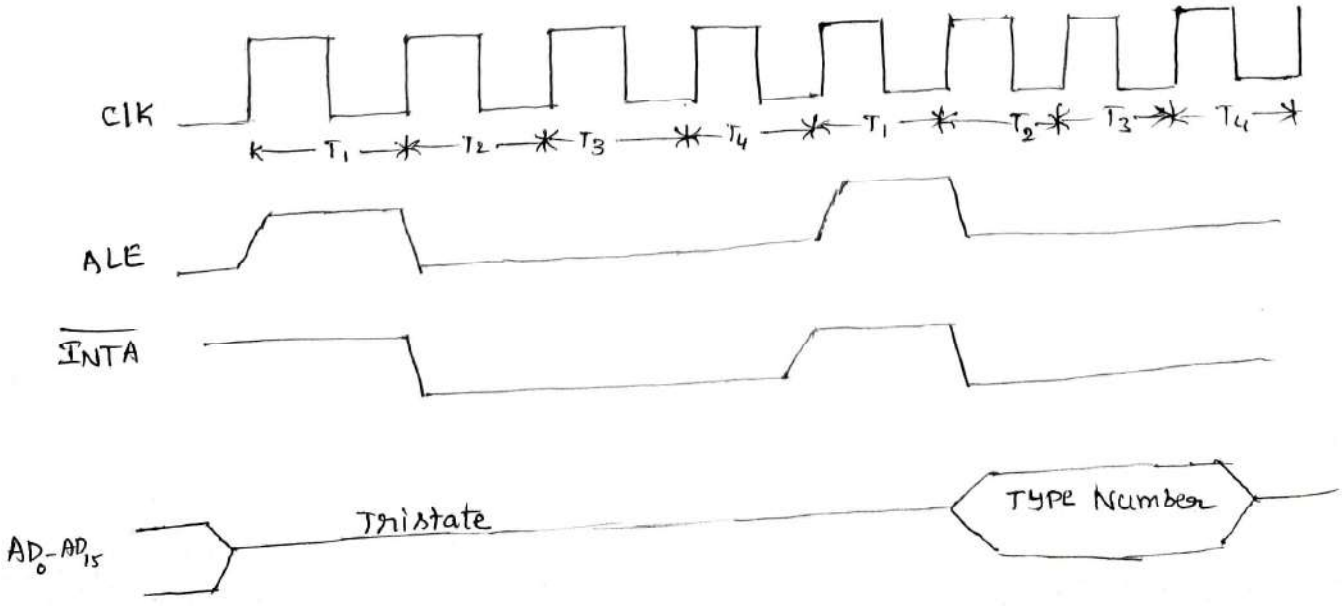


fig : 8086 INTR interrupt's response.

The 8086 generates the acknowledge signal $\overline{INTA}$ in two machine cycles during T_2-T_4 states of each machine cycle. During the first acknowledgement the $AD_0 - AD_{15}$ bus will be in tristate. This $\overline{INTA}$ during first machine cycle indicates the 8259 to perform certain internal operations to get the interrupt type related to the interrupt that is received on one of it's inputs $IR_0 - IR_7$.

The Interrupt type for IR_0 in 8259 is preprogrammed during it's initialization process. The Interrupt type for the successive inputs $IR_1, IR_2 - - - IR_7$ is one greater than the interrupt type of the previous interrupt.

For example if IR_0 is assigned 50H, the IR_1 is assigned 51H, IR_2 is assigned 52H and so on IR_7 is assigned 57H.

During the second acknowledgement of 8086, the 8259 places the type of the interrupt type on lower eight lines of the data bus $AD_7 - AD_0$ which is read by 8086. After receiving the interrupt type the 8086 execute the ISR of the received interrupt type.

The simplified diagram of interfacing the 8259 with 8086 is as shown in below.

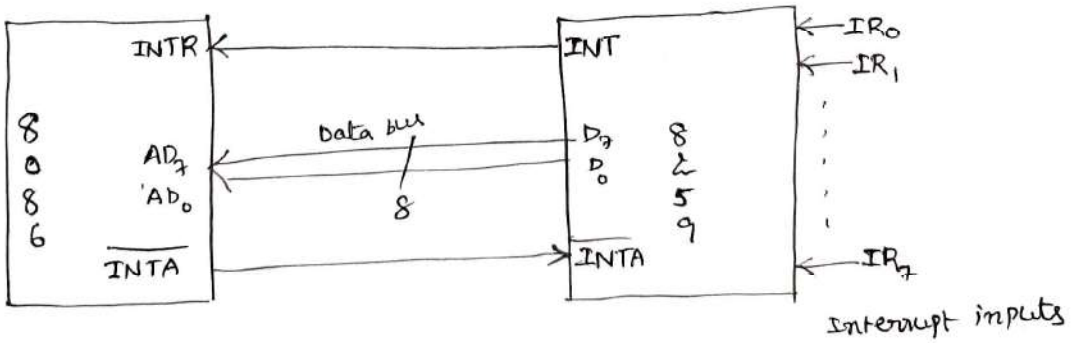


fig: Simplified diagram of interfacing the 8259 with the 8086

The advantage of interfacing 8259 with 8086 is the ability of the 8086 to handle multiple hardware interrupts. Because usually 8086 can handle only two hardware interrupts alone i.e NMI and INTR.

UNIT - II

ADDRESSING MODES OF 8086 :

The addressing mode indicates the way in which the operand or data for an instruction is accessed and the way in which the microprocessor calculates the branch address for the jump, call and return instructions.

The addressing modes of the instructions depends upon their types. According to the flow of instruction execution, the instructions may be categorized as i) sequential control flow instructions
ii) control transfer instructions.

↳ sequential control flow instructions are the instructions for which after execution, the control will be transferred to the next instruction appearing immediately after it in the program.

↳ The control transfer instructions on the other hand transfer the control to the address specified in the instruction, after their execution.

Examples for sequential control flow instructions are arithmetic, logical, data transfer and processor control instructions.

Examples for control transfer instructions are INT, CALL, RET and JUMP instructions.

↳ The addressing modes for sequential control flow instructions are explained as follows.

1) Immediate Addressing mode :

In this type of addressing mode the immediate data of either 8bit or 16 bit is directly given in the instruction.

Ex: i) MOV AL, 50H ; Move the data 50H to AL
ii) MOV BX, 53A9H ; Move the data 53A9H to BX

In the above two examples 50H, 53A9H are the immediate data of 8 bit, 16 bit respectively.

2) Direct addressing mode:

In the direct addressing mode the 16-bit offset address of the data with in the segment is directly given in the instruction

Ex: MOV AX, [2000H] :

In this instruction the offset address is 2000H. Since the destination in the above instruction is an 16-bit register (ie AX), a word is taken from the memory location DS x 10H + Effective address

= 3000 x 10H + 2000H = 32000H (say DS = 3000H)

Since a word contains two bytes, the bytes present in 32000H and 32001H are moved into AL and AH respectively.

3) Register addressing mode:

In the register addressing mode the data is stored in a register and it is referred using another register. All the registers except Instruction pointer (IP), may be used in this mode.

Ex: MOV AX, BX ; Move the data present in BX to AX
MOV CX, DX ; Move the data present in DX to CX
ADD AL, BL ; ADD the Contents of AL, BL and store the result in AL

4) Register indirect addressing mode:

Some times the address of the memory location which contains data (or) operand is determined by an indirect way, using offset registers. This mode is known as register indirect addressing mode. In this addressing mode, the offset address of the data is in either BX (or) SI (or) DI register.

The default segment is either DS (or) ES.

Example: MOV AX, [BX]

Here the data is present in data segment whose offset address is in BX. i.e the effective address = [BX]

Physical address = DS × 10 + [BX]

5) Indexed addressing mode:

In this addressing mode the offset address of the data (or) operand is stored in any one of the index registers. DS is the default segment for offset registers SI and DI. In case of string instructions DS and ES are the default segments for SI and DI respectively.

This mode is a special case of register indirect addressing mode.

Example1: MOV AX, [SI]

Here the data is available in the offset address stored in SI in data segment. Physical address = DS × 10 + [SI]

Example 2: MOV CX, [DI]

6) Register relative addressing mode:

In this addressing mode, the data is available in the effective address formed by adding an 8 bit (or) a 16-bit displacement with the content of any one of the registers BX, BP, SI and DI in the default segment (Either DS (or) ES).

Ex: MOV AL, 50H[BX]

Here the effective address = 50H + [BX]

Physical address = DS × 10 + Effective address

∴ Physical address = DS × 10 + 50H + [BX].

7) Based indexed addressing mode:

In this addressing mode the data is available in the effective address formed by adding the content of the base register (any one of BX (or) BP) to the content of an index register (any one of SI (or) DI). The default segment register is either DS (or) ES.

Example:

MOV AX, [BX][SI]

Here the base register = BX, index register = SI.

The effective address = [BX] + [SI]

Physical address = DS × 10H + Effective Address

= DS × 10H + [BX] + [SI]

8) Relative Based indexed addressing mode:

In this addressing mode the data is available in the effective address formed by adding the content of the base register (any one of BX (or) BP), the content of an index register (any one of SI (or) DI) and an 8-bit (or) 16-bit displacement. The default segment is either DS (or) ES.

Ex: MOV AX, 50H [BX][SI]

Here 50H is an 8-bit displacement, BX is the base register and SI is an Index register. The effective address of data is = 50H + [BX] + [SI]

∴ Physical address = DS × 10H + Effective address

= DS × 10H + 50H + [BX] + [SI].

↦ Addressing modes for the control transfer instructions:

For control transfer instructions, the addressing modes depend upon whether the destination location is with in the same segment or in a different one.

Basically there are two addressing modes for the control transfer instructions i.e intersegment and intrasegment addressing modes.

If the location to which the control is to be transferred lies in a different segment other than the current one, the mode is called intersegment addressing mode. If the destination lies in the same segment, the mode is called intrasegment addressing mode.

Each of these two modes are in turn divided into again two types. They are Intersegment direct, intersegment indirect. similarly intrasegment direct and intrasegment indirect. Let us discuss about each of these four.

1) Intra segment Direct addressing mode:

In this mode, the address to which the control is to be transferred lies with in the same segment in which the control transfer instruction lies and that address appears directly in the instruction as an immediate displacement value.

In this mode the displacement is computed relative to the content of the instruction pointer.

The effective address to which the control will be transferred is given by the sum of 8bit (or) 16-bit displacement and the content of IP.

Ex: JMP SHORT LABEL ; LABEL Lies with in -128 to +127 from the Current Ip Content.

Short Label is an 8-bit displacement.

Similarly Long jump is for 16-bit displacement i.e (-32768 to +32767)

2) Intra segment indirect addressing mode:

In this mode, the displacement to which the control is to be transferred is in the same segment in which the control transfer instruction lies, but that address is passed to the instruction indirectly.

Here the branch address is found as the content of a register.

Ex: i) JMP [BX] ; Jump to effective address stored in BX.

ii) JMP [BX+5000H];

3) Inter Segment inDirect addressing mode:

In this mode the address to which the control is to be transferred lies in a different segment and it is passed to the instruction indirectly, i.e Contents of a memory block Containing four bytes IP(LSB), IP(MSB), CS(LSB) & CS(MSB) sequentially. The starting address of the memory block may be referred using any one of the addressing modes, except immediate mode.

Ex: JMP [2000H]

Jump to an address in the other segment specified by the contents of a memory block of four bytes sequentially from the effective address 2000H.

2000H	IP (LSB)
2001H	Ip (MSB)
2002H	CS (LSB)
2003H	CS (MSB)

4) Intersegment Direct addressing mode :

In this mode the address to which the control is to be transferred is in the different segment. This addressing mode provides a means of branching from one code segment to another code segment. Here the CS and IP of the destination address are specified directly in the instruction.

Ex: JMP 5000H : 2000H ; jump to effective address 2000H in segment 5000H.

problem :

The contents of different registers are given below. Form the effective addresses for different addressing modes also find physical addresses.

[BX] = 2000H , [SI] = 3000H , [DI] = 4000H , [BP] = 5000H

[SP] = 6000H , [CS] = 0000H , [DS] = 1000H , [SS] = 2000H , [IP] = 7000H

Sol

i) Direct addressing mode :

MOV AX, [5000H]

Effective address = 5000H

Physical address = DS x 10H + Effective address

= 1000x10H + 5000H

∴ Physical address = 15000H

ii) Register indirect addressing mode :

MOV AX, [BX]

Effective address = [BX] = 2000H

Physical address = DS x 10H + Effective address

= 1000H x 10H + 2000H

∴ Physical address = 12000H

iii) Register relative addressing mode :

MOV AX, 5000[BX]

Given [BX] = 2000H

Relative displacement = 5000H

∴ Effective address = [BX] + 5000H = 2000H + 5000H = 7000H

Physical address = DS × 10H + Effective address

= 1000H × 10H + 7000H

∴ Physical address = 17000H

iv) Based Indexed addressing mode:

MOV AX, [BX][SI]

Given [BX] = 2000H, [SI] = 3000H

Effective address = [BX] + [SI] = 2000H + 3000H = 5000H

Physical address = DS × 10H + Effective address = 1000H × 10H + 5000H

∴ Physical address = 15000H.

v) Relative based indexed addressing mode:

Ex 1: MOV AX, 5000H [BX][SI]

Given [BX] = 2000H, [SI] = 3000H

Effective address = 5000H + [BX] + [SI]

= 5000H + 2000H + 3000H

= A000H

∴ Physical address = DS × 10H + Effective address

= 1000H × 10H + A000H

∴ Physical address = 1A000H.

Ex 2: MOV AX, -5000H [BX][SI]

Given [BX] = 2000H, [SI] = 3000H

Effective address = -5000H + [BX] + [SI]

= -5000H + 2000 + 3000H

= 0000H

∴ Physical address = DS × 10H + Effective address = 1000H × 10H + 0000H

∴ Physical address = 10000H.

Instruction format of 8086 micro processor:

The instruction format is the representation of an instruction in machine language. The instruction format may have one or more fields associated with it. The first field is called operation code (opcode) field, which indicates the type of the operation to be performed by 8086. The other fields in the instruction format are known as operand fields.

There are six general instruction formats in the 8086. The length of an instruction may vary from one byte to six bytes.

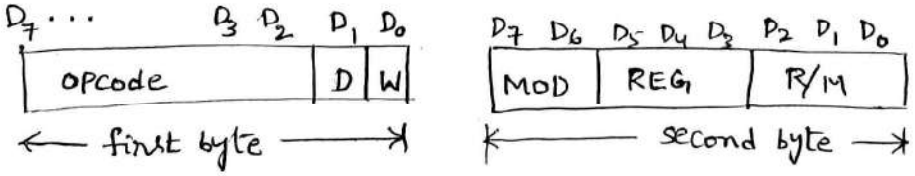
The instruction formats of 8086 are explained here.

1) one byte instruction:

This format is only one byte long, and it may have implied register or data operands. The least significant three bits of the opcode are used to specify the register operand, if any. otherwise all the eight bits in the instruction form an opcode and the operands are implied.

2) Register to Register:

This format is two bytes long. The first byte of the code specifies the opcode. The width of the operand is specified by 'W' bit present in the opcode field. The second byte of the instruction indicates the register operand and R/M (register/memory) field as given below.



The register represented by the field REG is one of the operand and is given in table below. When 'MOD' field's bits (i.e bits D_7 and D_6) are 1, the R/M field is also treated as a REG field. The direction bit 'D' indicates whether the data is transferred from

the register (if D=0) or to the register (if D=1).

W bit	Register code	Register
0	000	AL
0	001	CL
0	010	DL
0	011	BL
0	100	AH
0	101	CH
0	110	DH
0	111	BH

W bit	Register code	Register
1	000	AX
1	001	CX
1	010	DX
1	011	BX
1	100	SP
1	101	BP
1	110	SI
1	111	DI

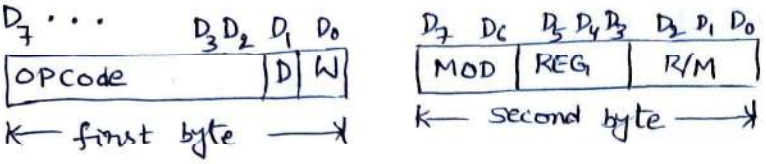
2-bit segment register code	segment register
00	ES
01	CS
10	SS
11	DS

3) Register to / from memory with no displacement :

This format is two bytes long. The MOD field indicates the mode of addressing. The MOD, REG, R/M and W fields are decided as per the following table

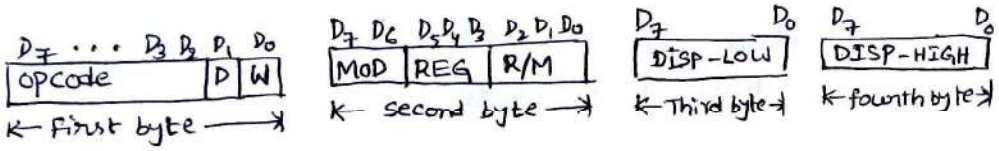
operands	Memory operand			Register operand	
	No Displacement	8-bit Displacement	16-bit Displacement		
R/M \ MOD	00	01	10	11	
				W=1	W=0
000	[BX]+[SI]	[BX]+[SI]+D8	[BX]+[SI]+D16	AX	AL
001	[BX]+[DI]	[BX]+[DI]+D8	[BX]+[DI]+D16	CX	CL
010	[BP]+[SI]	[BP]+[SI]+D8	[BP]+[SI]+D16	DX	DL
011	[BP]+[DI]	[BP]+[DI]+D8	[BP]+[DI]+D16	BX	BL
100	[SI]	[SI]+D8	[SI]+D16	SP	AH
101	[DI]	[DI]+D8	[DI]+D16	BP	CH
110	D16 (Direct address)	[BP]+D8	[BP]+D16	SI	DH
111	[BX]	[BX]+D8	[BX]+D16	DI	BH

In the above table D_{16}, D_8 are 16-bit and 8-bit displacements respectively. The instruction format is given here.



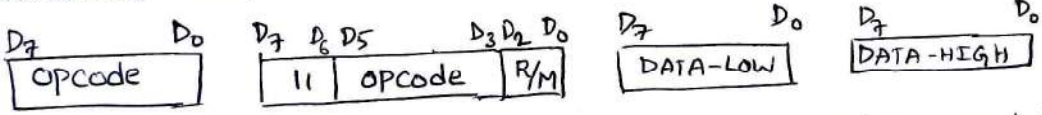
4) Register to/from memory with displacement:

This type of instruction format contains two bytes as in the previous instruction format and one or two additional bytes for displacement namely DISP-LOW (Third byte) and DISP-HIGH (Fourth byte), which are the lower order and higher order bytes of displacement respectively.



5) Immediate operand to Register:

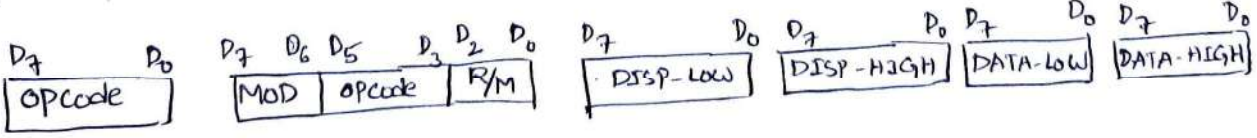
In this format, the first byte and the bits $D_3 - D_5$ in the second byte are used for opcode. It also contains one (or) two bytes of immediate data.



Here DATA-LOW, DATA-HIGH are the lower order and higher order bytes of data respectively.

6) Immediate operand to memory with 16-bit displacement:

This instruction format is five or six bytes long. The first two bytes contain the opcode, MOD, and R/M fields. Then two bytes of displacement and two bytes of data are present.



The opcodes have different single bit indicators which are as follows.

i) W-bit : This bit indicates whether the instruction operates on 8-bit data for which W=0 or 16-bit data for which W=1.

ii) D-bit : This is valid in the case of double operand instructions. One of the operand must be a register, specified by the REG field. The register specified by REG is source operand if D=0, else it is a destination operand. This bit is also called as Direction bit.

iii) S-bit : This bit is called as sign extention bit. The S-bit is used along with W-bit to show the type of the operation.

For example

a) when S=0, W=0, it indicates 8-bit operation with 8-bit immediate data.

b) when S=0 and W=1, it indicates 16-bit operation with 16-bit immediate data.

c) when S=1 and W=1, it indicates 16-bit operation with a sign extended immediate data.

iv) V-bit : This bit is used in shift and rotate instructions. It is set to 0, if shift count is '1' and to 1 if the CL register contains the shift count.

v) Z-bit : This bit is used by REP instruction to control the loop. If the Z-bit is 1, the string instruction with REP prefix is executed until zero flag matches the Z-bit.

Problem Assume DS = 1000H, DI = 2000H, CS = 3000H. When JMP [DI] is executed, find the location to which the control is transferred. Assume the word present in the location 12000H is 4000H.

Sol) Given DS = 1000H, DI = 2000H, CS = 3000H.

Given instruction is JMP [DI]

Effective address = [DI] = 2000H

Physical address = DS x 10H + Effective address = 1000x10H + 2000H

= 12000H.

Now the data present in the physical address 12000H is loaded into instruction pointer IP. i.e in 12000H location as per data the value is given as 4000H. so IP is loaded with 4000H.

Now the physical address to which the control has to be transferred is = CS x 10H + IP = 3000 x 10H + 4000H

= 34000H.

Therefore the control has to be transfered to a physical address location 34000H.

Instruction set of 8086 microprocessor:

The 8086 instructions are categorised into the following types.

i) Data transfer / Data copy instructions
ii) Arithmetic and logical instructions
iii) Branch instructions
iv) Loop instructions
v) Machine control instructions
vi) Flag manipulation instructions
vii) Shift and rotate instructions
viii) String instructions.

i) Data transfer / Data copy instructions:

These types of instructions are used to transfer data from source operand to destination operand. All the store, load, move, exchange, input and output instructions belong to this category.

MOV: (Move) Usage: MOV Destination, Source

This data transfer instruction transfers data from one register/ memory location to another register / memory location. The source may be any one of the general purpose registers (or) segment register (or) other special purpose registers (or) a memory location. The destination may be a register or a memory location.

However in case of immediate addressing mode, a segment register cannot be a destination register. i.e the direct loading of the segment registers with immediate data is not permitted. To load the segment registers with immediate data, one has to load that data into a general purpose register from where it has to be

moved into that particular segment register.

Ex:- 1) MOV DS, 5000H ; (invalid)

2) MOV AX, 5000H
MOV DS, AX

3) MOV AX, BX ; Register addressing mode

4) MOV CX, 0005H ; Immediate addressing mode

5) MOV AX, [SI] ; Register indirect addressing mode

6) MOV AX, [2000H] ; Direct addressing mode

7) MOV AX, 50H[BX] ; register relative (or) Base relative addressing mode.

* NOTE: It is noted that the source operand and destination operand both cannot be memory locations in an instruction except for string instructions.

2) PUSH : (Push to stack) usage: PUSH source

This instruction is used to push the contents of a specified register (or) memory location onto the stack. The stack pointer (SP) is decremented by two each time when 'PUSH' instruction is executed.

Ex: 1) PUSH CX ; PUSH the contents of CX into the stack
2) PUSH DX ; PUSH the contents of DX into the stack.
3) PUSH [BX]; PUSH the word in the memory at [BX] into the stack
4) PUSH DS ; Push the contents of DS into the stack.
5) PUSH AX ;

3) POP: (POP from stack)

usage: POP destination

This instruction is used to load a specified register or memory location with the contents of the memory location whose address is obtained using stack segment address and stack pointer address. Here the stack pointer is incremented by 2. The POP instruction is exactly opposite to PUSH instruction.

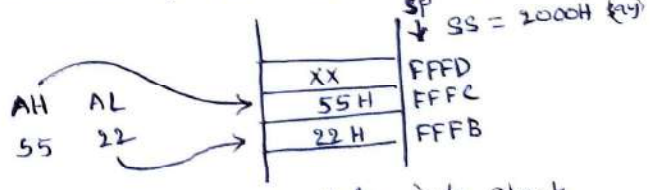
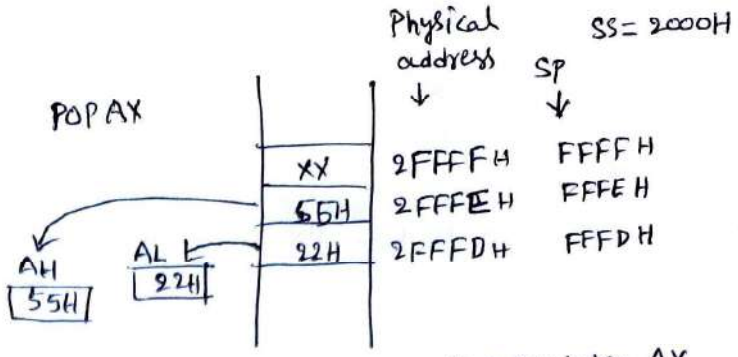


fig: Pushing AX data into stack

Examples:

1) POP AX
2) POP DS
3) POP [5000H]



After executing POP AX, the register AX will have 5522H.

4) XCHG : (Exchange)

Usage : XCHG Destination, source

This instruction is used to exchange the contents of the specified source and destination operands, which may be registers or one of them may be a memory location. However, exchange of contents of two memory locations is not permitted. Immediate data is also not allowed in these instructions.

Ex: XCHG AX, BX ; This instruction exchanges the data between AX and BX.

XCHG [5000H], AX ; Exchange data between AX and memory location [5000H] in the data segment.

5) IN : (Input port)

This instruction is used for reading an input port. The address of the input port may be specified in the instruction directly (or) indirectly. AL and AX are allowed destinations for 8-bit and 16-bit input operations respectively. DX register is the only register (implicit) which is allowed to carry the port address. If the port address is of 16 bits it must be in DX.

Ex: i) IN AL, 03H ; Read data from an 8bit port whose address is 03H and store it in AL.

ii) IN AX, DX ; This instruction reads data from a 16-bit port whose address is in DX and stores in AX register

iii) MOV DX, 0300H ; The 16-bit port address is taken in DX

IN AX, DX ; Read the content of the port whose address is in DX and store it in AX.

6) OUT : (output to the port)

This instruction is used for writing to an output port. The address of the output port may be specified in the instruction directly or indirectly. The contents of AX or AL are transferred to the directly given (or) indirectly given address port after the execution of this instruction.

The data to an odd address port is transferred on D_8-D_{15} while the data to an even address port is transferred on D_0-D_7.

The registers AL and AX are the allowed source operands for 8-bit and 16-bit operations respectively.

If the port address is of 16 bits it must be in DX.

Ex: i) OUT 03H, AL ; Writes data present in AL to a port whose address is 03H.

ii) OUT DX, AX ; Writes the data present in AX to a port whose address is specified in DX.

iii) MOV DX, 0300H ; The 16-bit port address is taken in DX

OUT DX, AX ; Write the data present in AX to a port whose address is specified in DX.

7) XLAT : (Translate) Usage: XLAT

The XLAT instruction is used to translate a byte present in AL from one code to another code. This instruction replaces a byte in the AL register with a byte in the memory whose offset address is present in [BX].

Before XLAT is executed, the look up table containing the desired codes must be stored in the data segment and the offset address of the starting location of that look-up table is stored in BX. The code byte which is to be translated is put in AL.

When XLAT is executed now, it adds the content of AL with BX to find the offset address of the data in the look-up table. Further the byte in that offset address will be copied to AL.

Ex:

```
MOV AL, 37H      ; Move the value 37H to AL (which is to be converted)
MOV BX, 2500H    ; Move the offset address of the look up table to BX.
XLAT ;           AL register is loaded from the look-up table
                 content, whose address is DS : {[BX]+[AL]}
```

8) LEA : (Load Effective Address) Usage: LEA Destination register, memory location

This instruction determines the offset address of a variable or memory location given in source operand place and loads this offset address value in the 16-bit destination register.

Ex:
i) LEA BX, ADR ; Effective address of a label 'ADR' is transferred to register BX

ii) LEA CX, [BX][SI] ; Effective address [BX] + [SI] is transferred to the register CX.

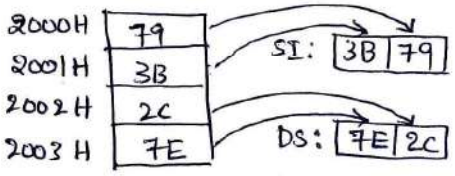
iii) LEA BX, COST[SI] ; offset of the label 'Cost' is added to the content of SI and loaded into BX.

9) LDS/LES: (Load pointer to DS (or) ES) Usage: LDS Destination register, memory location

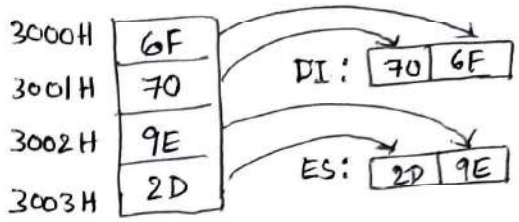
This instruction loads the DS register or ES register and the specified destination register in the instruction with the content of memory location specified as source in the instruction.

LDS (or) LES instruction copies a word from the memory location specified in the instruction to the specified destination register in the instruction and then copies a word from the next memory location to DS (or) ES register.

Ex) i) LDS SI, [2000H]



Ex ii) LES DI, [3000H]



10) LAHF (Load AH from lower byte of the flag register) : usage: LAHF

This instruction loads AH register with the lower byte of the flag register. This command may be used to observe the status of all condition code flags (except overflow flag) at a time.

11) SAHF (store AH to lower byte of the flag register) : usage: SAHF

This instruction sets or resets the conditional code flags (except overflow flag) in the lower byte of the flag register depending upon the corresponding bit positions in AH. If a bit in AH is 1, the flag corresponding to that bit position is set else it is reset.

12) PUSHF : (push flag register onto stack) : usage: PUSHF

This instruction pushes the flag register content onto the stack; firstly the higher byte is pushed and then the lower byte. The stack pointer (SP) is decremented by 2; for each push operation. The operation of this instruction is similar to 'PUSH' instruction operation.

13) POPF : (POP flags from the stack) usage: POPF

This instruction loads the flag register completely (both bytes) from the contents of the memory location whose address is given by SS:SP. After executing the POPF instruction, the stack pointer (SP) is incremented by 2.

ii) Arithmetic Instructions and Logical Instructions:

All the instructions that perform arithmetic, logical, increment, decrement, compare and scan instructions belong to this category. The arithmetic instructions perform the operations such as addition, subtraction, multiplication and division along with the respective ASCII and decimal adjust operations.

The arithmetic and logical instructions affect all the condition code flags depending upon the result.

The instructions under this category are discussed below.

Arithmetic instructions:

1) ADD: (Add) Usage: ADD destination, source.

* This instruction adds the contents of source operand and destination operand and the result is stored in the destination.
* Here the source may be an immediate data (or) a register (or) a memory location. The destination can be a register or a memory location.
* However the source and destination both cannot be memory locations. The data from the source and destination must be of same type (either bytes or words)

Ex: i) ADD AL, 70H
ii) ADD CX, 12A3H
iii) ADD BX, [SI]
iv) ADD AX, BX

* All the condition code flags (AF, CF, OF, PF, SF and ZF) are affected by the execution of the 'ADD' instruction

2) ADC: (Add with Carry) Usage: ADC destination, source.

* This instruction adds the contents of source and destination along with the content of carry flag and stores the result in the destination. All the rules specified for the instruction 'ADD' are

applicable for ADC instruction also.

* All the Condition Code flags (AF, CF, OF, PF, SF and ZF) are affected by the execution of ADC instruction.

Ex: i) ADC AX, BX
ii) ADC AX, 1000H
iii) ADC AX, [SI]
iv) ADC AX, [5000H]

3) INC : (Increment) Usage: INC Destination

* This instruction increases the content of a specified register or memory location by 1. All the Condition Code flags except carry flag are affected by the execution of INC instruction.

* Immediate operand cannot be used with this instruction.

Examples: i) INC AX
ii) INC [BX]
iii) INC [2000H]

4) SUB : (Subtract) Usage: SUB Destination, source.

* This instruction is used to subtract the source operand from the destination operand and the result is stored in the destination operand.

* source operand may be a register (or) a memory location (or) immediate data. The destination can be a memory location (or) register but not an immediate data. And also both the source and the destination can not be memory locations.

* All the Condition Code flags are affected by the execution of 'SUB' instruction.

Examples: i) SUB AX, BX
ii) SUB CX, 0100H

iii) SUB AX, [5000H]

5) SBB (Subtract with borrow) : Usage : SBB Destination, Source

* This instruction subtracts the source operand and the carry flag content, from the content of destination and stores the result in the destination. The rules for the source operand and destination operand are same as that of SUB instruction.

* All the condition code flags are affected by the execution of the SBB instruction. The result is stored in the destination operand.

Ex: i) SBB AX, BX,
ii) SBB AX, [5000H]
iii) SBB AX, 2500H

6) DEC (Decrement) : Usage: DEC Destination.

* This instruction decreases the content of a specified register (or) memory location by 1. All the condition code flags except the carry flag are affected by the execution of DEC instruction.

* Immediate operand cannot be used with this instruction.

Examples : i) DEC SI
ii) DEC [SI]
iii) DEC CX
iv) DEC [1000H]

7) CMP (Compare) : Usage : CMP Destination, Source.

* This instruction compares the source operand with the destination operand. The source operand can be a register (or) memory location (or) an immediate data. The destination operand can be a register (or) a memory location.

* For comparison, it subtracts the source operand from the

destination operand, but does not store the result anywhere.

* All the condition code flags are affected by the execution of 'CMP' instruction.

* If the source operand is greater than the destination operand the carry flag is set or else carry flag is reset.

* If the source operand is equal to the destination operand the zero flag is set, otherwise zero flag is reset.

Examples: i) CMP AX, 1000H ;

ii) CMP BX, CX

iii) CMP AX, [5000H]

iv) CMP AX, [SI]

8) AAA : (ASCII Adjust after addition) Usage: AAA

The AAA instruction is executed after an ADD instruction that adds two ASCII coded operands to give a byte of result in AL. The AAA instruction converts the content of AL to a valid unpacked BCD. After addition AAA instruction examines the lower 4 bits of AL to check whether it contains a valid BCD number in the range 0 to 9. If it is between 0 to 9 and AF=0, AAA instruction makes the 4 bits of higher nibble of AL to 0. AH must be cleared before addition.

If the lower nibble of AL is between 0 to 9 and AF is set (i.e AF=1), 06 is added to AL. The upper 4 bits of AL (i.e higher nibble of AL) are cleared and AH is incremented by one.

If the lower nibble of AL is greater than 9, then 06 is added to AL. The higher nibble of AL is cleared and AH is incremented by one. and AF and CF are set.

'AAA' instruction affects AF and CF and the remaining flags are undefined.

Ex:
```
MOV  AH, 00H
MOV  AL, 36H  ;  ASCII Code of 6 is moved into AL.
MOV  BL, 39H  ;  ASCII Code of 9 is moved into BL
ADD  AL, BL   ;  Add the Contents of AL, BL. Store the result 6F into AL.
AAA   ;  Since the lower nibble of AL is 'F' which is greater than 9, the content of AL is added with 06 and the higher nibble of AL is cleared after adding 06. This leads to the unpacked BCD 05 in AL register and AH is incremented by 1. Before performing addition AH must be cleared.
```

The above said operation is shown below.

Content in AL = 6FH before AAA instruction

Content in AH = 00H before AAA instruction.

Lower nibble of AL = F > 9 and hence 06 is added.

i.e 6F + 06 = 0110 1111
+ 0110
0111 0101

= 75H

Content of AL after AAA : AL = 05 ; (∵ higher nibble of AL '7' is cleared)

Content of AH after AAA : AH = 01 ; AH is incremented by 1 after AAA.

9) AAS (ASCII adjust after subtraction): Usage: AAS

This instruction is similar to AAA instruction. After performing subtraction of two ASCII coded operands only AAS instruction is used. The AAS instruction converts the content of AL to a valid unpacked BCD. AH register has to be cleared before the addition itself. After the subtraction, the AAS instruction examines the lower nibble of AL. If the lower nibble of AL is between 0 to 9 (i.e a valid BCD), AAS makes the higher nibble of AL to 0. AH is remained as 00H only.

If the lower nibble of AL is in between A to F (or) if the auxilliary carry flag is '1' then the higher nibble of AL is cleared and AH is decremented by 1 and AF, CF are set. AAA and AAS affect AF and CF. The flags OF, PF, ZF and SF are un-defined.

Case(i) (Example)

```
MOV AH,00H ; clear AH content.
MOV AL, 37H ; ASCII Code of 7 is moved into AL.
MOV BL, 32H ; ASCII Code of 2 is moved into BL.
SUB AL, BL ; Subtract the contents of BL from AL Content, store the result in AL
AAS
```

Before AAS, i.e just after subtraction AL = 05H, AH = 00H

Since the lower byte of AL = 05H in which lower nibble is 5 < 9.

So After AAS, AL = 05, AH = 00H.

Case ii) (Example)

```
MOV AH,00H ; clear AH Content
MOV AL, 32H ; ASCII Code of 2 is moved into AL
MOV BL, 39H ; ASCII Code of 9 is moved into BL
SUB AL, BL ; Subtract the Contents of BL from AL and store the result in AL
AAS;
```

Before AAS, i.e just after subtraction AL = F9H Since the Auxilliary Carry flag is set (as the borrow is generated from lower nibble to higher nibble position) 06 is subtracted from the Content of AL and from that value higher nibble is cleared. And also AH is decremented by '1'.

$\therefore$ AL = F9 = 1111 1001

0000 0110

1111 0011

After higher nibble 1111 is cleared AL = 0000 0011

$\Rightarrow$ AL = 03H

AH is decremented by 1. Now AH = FFH (2's Complement of -1).

For this case, after AAS AH = FFH, AL = 03H.

∴ The result of AX = FF03H which is the 10's complement

i.e 10 − 7 = 3.

10) AAM: (ASCII adjust after multiplication) : (Usage: AAM)

This instruction is used only after a multiplication instruction that multiplies two unpacked BCD operands. When AAM instruction is executed it converts the content of AL into unpacked BCD format. The higher order digit is placed in AH and the lower order digit in AL.

Ex:
```
MOV AL, 05H ; Move 05H to AL register
MOV BL, 09H ; move 09H to BL register
MUL BL ;  multiply the contents of AL and BL and the
          result 2DH is placed in AL.  (2D)16 = (45)10
AAM ;     AH = 04H, AL = 05H, unpacked BCD form of the result
          i.e AX = 0405H.
```

The AAM instruction affects parity flag, sign flag and ZF flag. AF, CF and OF are undefined.

11) AAD : (ASCII adjust before division) : (Usage: AAD)

This instruction is used before dividing two unpacked BCD digits in AX by an unpacked BCD byte.

AAD converts two unpacked BCD digits in AH and AL to an equivalent binary number in AL.

After division, AL contain the unpacked BCD quotient and AH contains the unpacked BCD remainder.

This instruction affects PF, SF and ZF. The flags AF, CF and OF are undefined.

Ex:
```
MOV AX, 0508H ; Move 0508H to AX register
MOV BL, 09H ; Move 09H to BL register
AAD         ; Result in AX = 003AH = (58)10.
DIV BL ;  Divide AX by BL.
```

Quotient = [AL] = 06H (unpacked BCD)

Remainder = [AH] = 02H (unpacked BCD)

12) DAA : (Decimal adjust after addition) Usage: DAA

This instruction is used only after the addition of two packed BCD numbers. The result of the addition must be present in AL. If the lower nibble of AL is greater than 9 (after the addition) (or) if AF = 1 by the addition, the DAA instruction adds 06 to AL. If the higher nibble of AL is greater than 9 (or) if the carry flag is set by addition, the DAA instruction adds 60H to AL. DAA brings the addition result into BCD form.

Ex) a) Let AL = 98 H (packed BCD)

CL = 35 H (packed BCD)

ADD AL, CL ; Add the contents of AL, CL. The result AL = 8DH

DAA ; Adds 06H to AL since lower nibble in AL is greater than 9. After adding 06, AL = 93 BCD.

b) Let AL = 88H (packed BCD)

CL = 49H (packed BCD)

ADD AL, CL ; Add the contents of AL, CL. The result in AL = D7H. AF = 1.

DAA ; Adds 60H to AL since the higher nibble in AL is greater than 9. So after addition of 60H AL = 37 BCD and CF = 1. The final result in the AL register is 37 and CF = 1, so together 137H is the result.

This instruction affects AF, CF, PF, and ZF.

overflow flag is undefined.

13) DAS : (Decimal adjust after subtraction) Usage : DAS

This instruction is used only after the subtraction of two packed BCD numbers. The result of the subtraction must be present in AL. The DAS instruction brings the result into BCD form.

If the lower - nibble of AL after subtraction is greater than 9 (or) if AF = 1 by subtraction, the DAS instruction subtracts 06H from AL.

If the higher nibble of AL after subtraction is greater than 9 (or) if carry flag = 1 by subtraction, the DAS instruction subtracts 60H from AL.

Ex a)
```
MOV AL, 86H ; Move 86H into AL : 86H (Packed BCD)
MOV CL, 57H ; Move 57H into CL (57H Packed BCD)
SUB AL, CL  ; Subtract CL from AL.
DAS ;  Subtract 06H from the AL as the
       lower nibble in AL after the subtraction
       is greater than 9. So after subtracting
       06H , AL = 29H. (BCD)
```

b)
```
MOV AL, 49H ; Move 49H (Packed BCD) into AL
MOV CL, 72H ; Move 72H (Packed BCD) into CL.
SUB AL, CL  ; Subtract CL from AL.
DAS ;  Subtract 60H from the AL as the higher
       nibble of AL after subtraction is greater than 9.
       So after subtracting 60H from AL , AL = 77H.
                                            (Packed BCD)
```

14) NEG : Negate Usage : NEG Destination

This instruction forms the 2's Complement of the specified destination in the instruction. For obtaining 2's Complement it subtracts the Contents of destination from zero. The result is stored back to the destination operand which can be either a register or a memory location

If OF is set, that indicates the operation could not be Completed

successfully. This instruction affects all the Condition code flags.

Ex: NEG AX ; Replace the content of AX with it's 2's Comple-ment.

NEG [2400H] ; Replace the Content of [2400H] with it's 2's Complement.

15) MUL (unsigned multiplication) Usage: MUL source

The instruction MUL is used for multiplying two unsigned bytes or words. The source can be a register or a memory location which can be considered as the multiplier. The multiplicand is taken by default from AL and AX for byte and word type data respectively. The result of the multiplication of two 8-bit numbers is stored in AX and the multiplication result of two 16-bit numbers is stored in DX-AX. (i.e most significant word in DX and least significant word in AX).

Multiplication of two 8-bit number gives 16-bit result and the multiplication of two 16-bit number gives 32-bit data.

Ex:- i) MUL BL
ii) MUL CX
iii) MUL BX

The source operand cannot be an immediate data in this case.

All Condition code flags are affected. But CF and OF are meaniful with the result. The remaing flags are undefined.

16) IMUL (signed multiplication) Usage: IMUL source.

this instruction IMUL is used to multiply a signed byte in source operand by a signed byte in AL (or) a signed word in source operand by a signed word in AX. The source can be a general purpose register (or) memory location but it cannot be an immediate data. In case if the magnitude of the product does not require all the 16 bit (or) all the 32-bits of the destination (i.e all the magnitude bits are not completely filled), the unused bits are taken as same as that of the sign bit and CF, OF will both be '0'.

The other condition code flags ZF, PF, SF and AF are undefined.

Ex: IMUL BL
IMUL BX

17) DIV : (un-signed division) Usage: DIV source

This instruction is used to perform the unsigned division. It divides an unsigned word or double word by a 16 bit or 8-bit operand. The dividend must be in a AX for 16-bit and in AL for 8 bit.

The source (divisor) can be a register or a memory location. when a double word is divided by a word, the upper 16-bit of the double word is taken in DX and the lower 16-bit in AX. After the division AX contains the 16-bit quotient and DX contains a 16-bit remainder.

when a word is divided by a byte, word is placed in AX register. The divisor can be a register (or) a memory location. After the division the AL contains the quotient and AH contains the remainder.

After the execution of DIV instruction all flags are undefined.

Ex: DIV CL ; Divide a word in AX by a byte in CL. Quotient in AL and remainder in AH.

DIV BX ; Divide a double word in DX and AX by BX. Quotient in AX and remainder in DX.

18) IDIV : (signed division) Usage : IDIV source.

This instruction performs the same operation as that of DIV instruction, but with signed operands. The result is stored in the same manner as that of DIV instruction for both cases of word and double word divisions. The result will also be signed numbers.

The operands are also specified in the same way as that of DIV instruction. If the result is too big to fit in AX (16-bit dividend operation) (or) to fit in DX-AX (in 32-bit dividend operation), a divide by 0 interrupt is generated.

After IDIV operation all the flags are undefined.

Ex: IDIV CL

IDIV BX.

19) CBW (Convert signed byte to word): Usage: CBW

This instruction converts a signed byte to a signed word. In other words, it copies the sign bit of a byte to be converted to all the higher byte of the resultant word. The byte to be converted must be in AL. The result will be in AX. It doesnot affect any flag.

20) CWD (Convert signed word to double word): Usage: CWD

This instruction copies the sign bit of AX to all the bits of the DX register. This operation is to be done before signed division. It doesnot affect any flag.

Logical Instructions:

Under this category of instructions include shift, rotate and basic logical operations.

The basic logical operations available with 8086 instruction set are AND, OR, NOT and XOR.

AND: (Logical AND) Usage: AND destination, source

This instruction performs the logical AND operation between the corresponding bits in the source and destination and stores the result in destination.

The source operand can be a register (or) a memory location (or) immediate data, and the destination operand can be a register (or)

a memory location but not be an immediate data. Both the source and destination operands cannot be memory locations.

The carry flag and overflow flag are both '0' after each AND instruction and PF, SF and ZF are updated after AND operation. But AF is undefined.

Ex i) AND AX, 3030H iii) AND AX, [5000H]
ii) AND AX, BX iv) AND [5000H], CX.

OR: (Logical OR). Usage: OR Destination, source.

The 'OR' instruction performs the logical OR operation between the corresponding bits in the source and destination and the result is stored in the destination.

The source operand can be a register (or) a memory location (or) an immediate data. The destination operand can be a register (or) a memory location but not be an immediate data. Both the source and destination cannot be memory locations.

The flags CF and OF both are '0' and PF, SF, ZF are updated after each AND operation. AF is undefined.

Ex i) OR AX, 0F0FH iii) OR AX, [2000H]
ii) OR AX, BX iv) OR CX, [BX]

NOT: (Logical invert) Usage: NOT destination.

The NOT instruction complements the contents of a destination operand bit by bit. The destination operand can be a register or a memory location. But it can not be an immediate data.

Ex: i) NOT AX
ii) NOT [BX]
iii) NOT [5000H].

The NOT instruction doesnot affect any flags.

XOR: (Logical Exclusive OR) Usage: XOR destination, Source.

The XOR instruction performs the logical XOR operation between the corresponding bits in the source and destination and stores the

result in the destination.

The source operand can be a memory location (or) a register (or) an immediate data. The destination operand can be a register (or) memory location but not immediate data. Both the source and destination cannot be the memory locations.

After XOR instruction execution CF = 0, OF = 0 and PF, SF, ZF are updated based on result. AF is undefined.

Ex: i) XOR AX, 1500H
ii) XOR CX, [BX]
iii) XOR [1000H], AX
iv) XOR AX, BX

TEST: (Logical Compare) Usage: TEST destination, source

The TEST instruction performs a bit by bit logical AND operation between the corresponding bits of source and destination, but the result is not stored in the destination. But the flags are affected. The flags CF and OF are '0', and PF, SF, ZF are updated based on the result but AF is undefined after the execution of the TEST instruction. The source operand can be a register (or) a memory location (or) immediate data and the destination operand can be a register (or) a memory location but not an immediate data. Both source and destination cannot be the memory locations.

Ex: i) TEST AX, BX
ii) TEST BX, 1050H
iii) TEST CX, [BX]
iv) TEST AX, [1800H]

Shift and rotate instructions.

SAL / SHL (Arithmetic left shift / logical left shift)

Usage: SAL Destination, Count
SHL Destination, Count.

The SAL (or) SHL instructions are used to shift the contents of the given destination operand bit by bit left side by the number of bit positions specified by CL. If the count is '1' it can be specified directly in the instruction directly without using CL.

The destination operand can be either a register (or) a memory location. The result after executing SAL (or) SHL is stored in destination operand. In the case of all shift and rotate instructions the count is either '1' or it is specified by CL.

This instruction (SAL/SHL) shifts the contents of destination operand bit by bit left side and insert zeros in least significant bit. MSB will be shifted to carry flag.

Ex: i) SAL AX, 01H

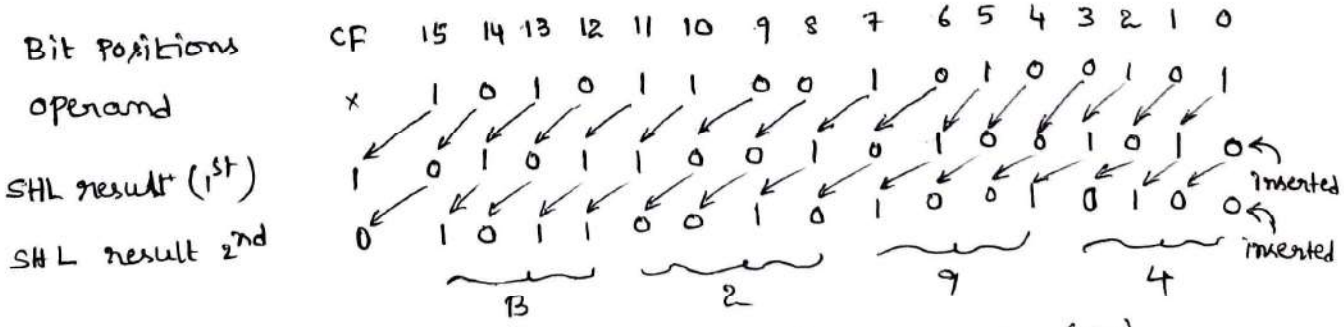
ii) SHL AX, 01H

iii) MOV CL, 05H
SAL AX, CL

iv) MOV CL, 07H
SHL AX, CL.

v) MOV AX, ACA5H
MOV CL, 02
SHL AX, CL

This example has been explained below.



The result B294 is stored in destination operand (AX).

This instruction affects all the condition code flags except AF. AF is undefined.

SHR : (Shift logical right) Usage : SHR Destination, Count.

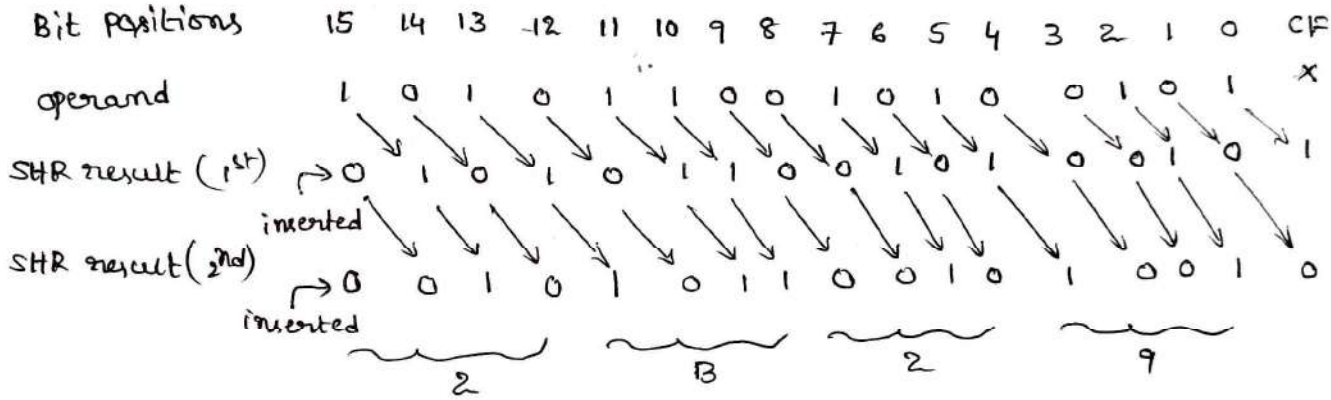
The SHR instruction is used to shift the Content of the given destination operand bit by bit right side by the number of bit positions specified by CL. If Count is '1', it can be specified directly in the instruction without using CL. The destination operand may be a memory location (or) a register. The result after executing SHR, is stored in the destination operand.

The SHR instruction Shifts the Contents of the destination operand bit by bit rightside and insert Zeros in the MSB Position. LSB bit is shifted to Carry flag.

Ex: i) SHR AL, 01

ii) MOV CL, 03H
SHR AX, CL

Let AX = ACA5H , CL=02H the execution of SHR AX, CL is explained below



The result 2B29H is stored in destination operand AX. SHR instruction affects all the condition code flags OF, SF, PF, ZF, and CF except AF. AF is undefined.

SAR : (Shift Arithmetic Right) Usage : SAR Destination, Count.

The SAR instruction is used to shift the Content of the given destination operand bit by bit right side by the number of bit positions specified by CL. If the count is '1', it can be specified directly in the instruction without using CL. The destination operand may be a memory location (or) a register. The result after 'SAR' execution, is stored in the destination operand.

The SAR instruction shifts the contents of destination operand bit by bit right side and inserts the MSB of the given destination operand after each shift. The LSB after each shift is shifted to Carry flag

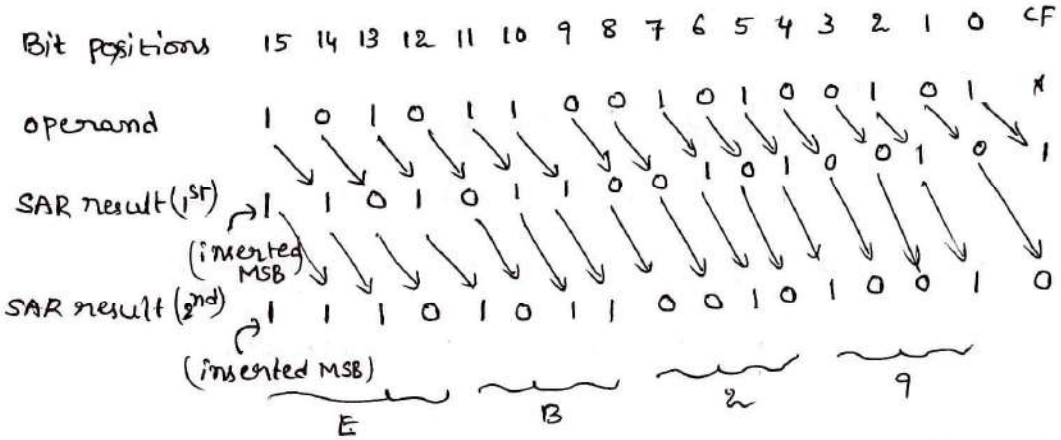
This instruction affects all the condition code flags (PF, SF, ZF, OF, and CF) except AF. AF is undefined.

Ex: i) SAR AX, 01H

ii) SAR AL, 01H

iii) MOV CL, 02H
SAR AX, CL

Let AX = AC A5H, CL = 02, The execution of SAR AX, CL is as shown below.



After executing SAR instruction in this example, the result is stored in AX is EB29H.

ROL : Rotate Left without Carry Usage : ROL Destination, count.

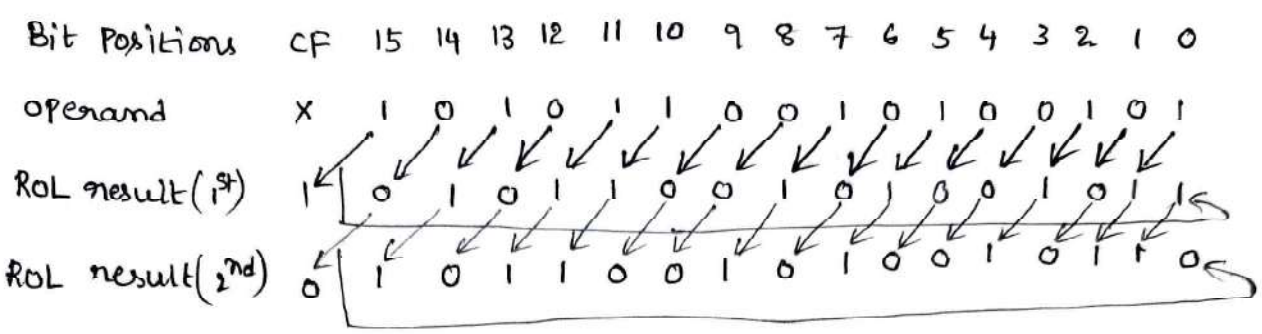
The ROL instruction rotates the contents of the given destination operand bit by bit left side by the count specified by CL. If the count is 1 it can be directly specified in the instruction directly without using CL. The destination operand can be a memory location or a register.

The MSB is pushed into Carry flag as well as the LSB position at each operation. The Carry flag and over-flow flag are affected. The remaining flags are un affected. Remaining bits of destination are shifted bit by bit left side.

Ex: i) ROL AL, 01

ii) MOV CL, 05H
ROL AX, CL

Let AX = ACA5H, CL = 02, the operation of ROL AX, CL is explained below.



The result of ROL AX, CL in this example is B296H, stored in AX.

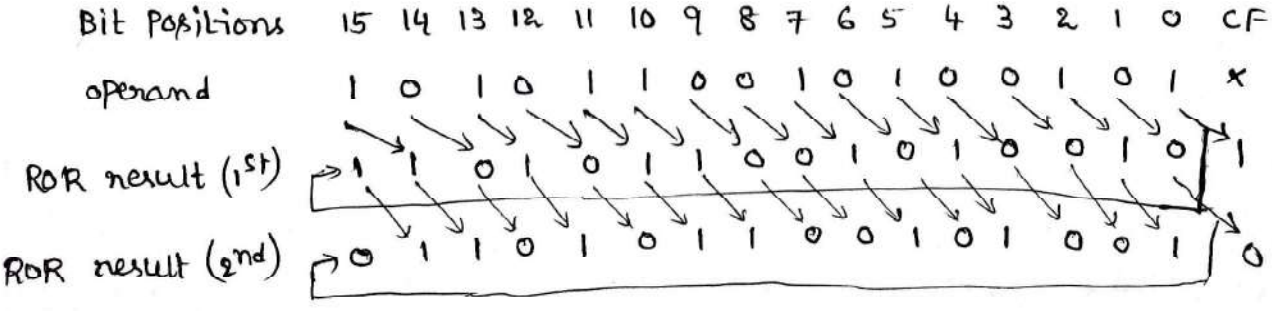
ROR (Rotate right without carry): Usage: ROR Destination, Count

The ROR instruction rotates the contents of the destination operand bit by bit right side by the count specified by CL. If the count is '1', it can be directly specified in the instruction without using CL. The destination operand can be either a register or a memory location.

The LSB is pushed into carry flag as well as the MSB position at each operation. The carry flag and overflow flag are affected. The remaining flags are unaffected. The remaining bits are shifted bit by bit right side.

Ex: i) ROR AL, 01
ii) MOV CL, 08H
ROR AX, CL.

Let AX = ACA5H, CL = 02, the operation of ROR AX, CL is explained below.



The result of ROR AX, CL in this example is 6B29H, stored in AX.

RCL (Rotate left through Carry) Usage: RCL Destination, Count.

This instruction rotates the contents of the destination operand bit by bit left side through Carry, by the Count specified by CL. If the Count is 1, it can be directly specified in the instruction without using CL. The destination operand can be either a register or a memory location.

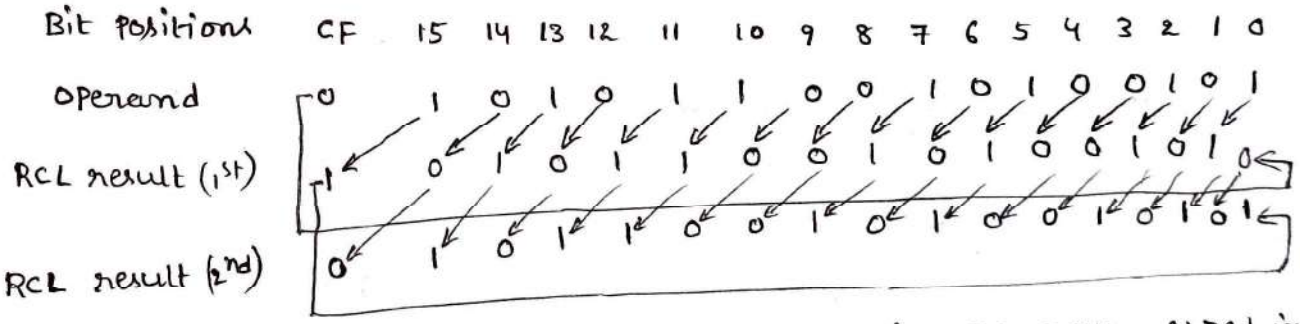
At each operation, the carry flag is pushed into LSB, the MSB bit is pushed into carry flag. The remaining bits of destination operand are shifted left by the specified positions.

This instruction affects CF, OF. The remaining flags are un-affected.

Ex: i) RCL AL, 01H

ii) MOV CL, 05H
RCL AX, CL

Let AX = ACA5H, CL = 02, the operation of RCL AX, CL is explained below.



The result of ROR AX, CL in this example is B295H, stored in AX.

RCR (Rotate right through Carry) usage: RCR destination, Count.

This instruction rotates the contents of the destination operand bit by bit right side through Carry, by the Count specified by CL. If the Count is '1', it can be directly specified in the instruction without using CL. The destination operand can be a register or a memory location.

At each operation the carry flag is pushed into MSB, the LSB bit is pushed into Carry flag. The remaining bits are shifted right by the specified positions.

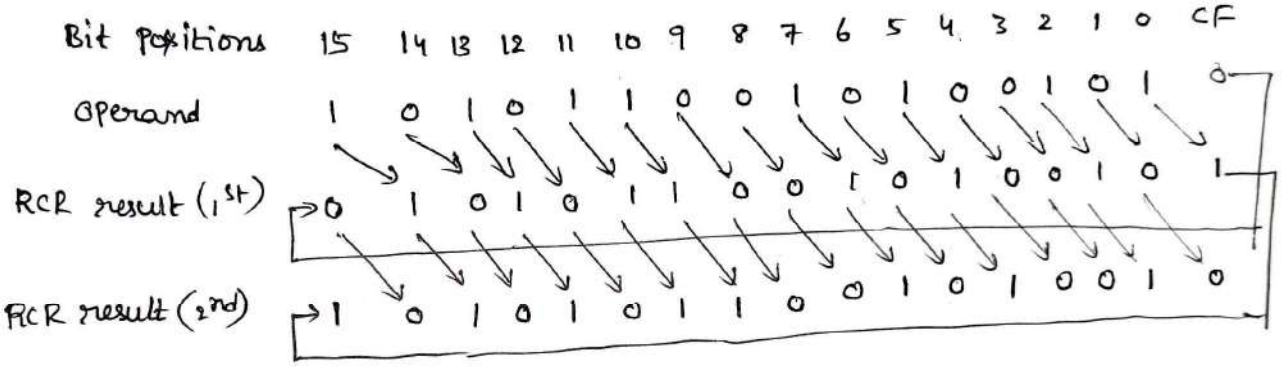
This instruction affects CF, OF. The remaining flags are unaffected.

Ex: i) RCR AL, 01

ii) MOV CL, 07H

RCR AX, CL.

Let AX = ACA5H, CL=02, the operation of RCR AX, CL is explained below.



The result of RCR AX, CL in this example is AB29H, stored in AX.

iii) Branch instructions (or) Control transfer instructions:

After executing a branch instruction the control of execution is transferred to a new location whose address is specified in the instruction directly or indirectly.

The branch instructions are of two types. They are unconditional branch instructions and Conditional branch instructions. In the case of unconditional branch instructions, the control of execution is transferred to a specified location with out the necessity of satisfying any condition.

In a conditional branch instruction, the control is transferred to a new location only if a particular condition is satisfied.

Unconditional Branch instructions:

CALL (unconditional call): Usage: CALL destination.

This instruction is used to call a sub-routine (procedure) from the main program. The address of the subroutine is specified in the call instruction either directly or in-directly.

There are two types of CALL. They are Near Call and Far Call.

A Near call is a call which calls a subroutine which is in the same segment of Call instruction. The near call requires intra segment addressing modes.

A far call is a call which calls a subroutine which is in the different segment from the segment that contains CALL instruction. The Far call requires Intersegment addressing modes.

When a CALL instruction is executed, it stores the CS content and IP content onto the stack and the CS and IP registers are loaded with the segment and offset addresses of subroutine.

Ex: CALL Multi ; Call procedure with a name Multi.

CALL BX ; BX content specifies the offset of the first instruction of the procedure. This is a near call.

RET: (Return from procedure)

At each CALL instruction the CS and IP contents of the next instruction is pushed onto the stack, before the control is transferred to the procedure. At the end of the procedure, the RET instruction is present, executing which the previously loaded contents of IP and CS are retrieved into the IP register and the CS registers.

If the procedure is a near procedure, then the RET instruction replaces the IP with a word from the top of the stack.

If the procedure is a far procedure, then the RET instruction replaces the IP and CS registers with the words from the stack top.

INT N: (Interrupt of Type N).

8086 supports 256 interrupts each of which has a specific type value. The type value refers to a number in the range 00H-FFH. When INT instruction is executed the flag register contents are pushed onto the stack by decrementing SP by 2. Again SP is decremented by 2 but now CS content is loaded onto the stack. similarly IP content is loaded onto the stack by decrementing SP by 2.

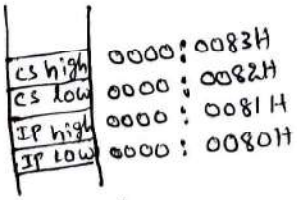
Now the type value N is multiplied by 4 which gives the offset address of the interrupt vector in 0000H segment. From that address, in 4 consecutive locations among which the top two locations have the higher and lower bytes of CS address and the lower two locations that have higher and lower bytes of IP address of the ISR. with these values the CS and IP registers are loaded, hence the control is transferred to ISR.

When INT instruction is executed IF and TF are cleared. All the remaining flags are unaffected.

Ex: INT 20H

Type = 20H so 20X4 = 80H

The four locations from 0000:0080H contains IP, CS.



IRET: (Return from ISR)

when an ISR is to be called, before transferring the control of execution the flag register contents, CS contents and IP contents are loaded into the stack to indicate the location from where the execution is to be continued after the execution of ISR. At the

end of ISR, when IRET is executed the values of IP, CS and flag register contents are retrieved from the stack to continue the execution of the main program.

INTO: (Interrupt on overflow) usage: INTO

This instruction is executed when OF = 1. This is equivalent to a type 4 interrupt instruction. when OF = 1 and INTO is executed, the IP and CS are loaded with the contents of 0000:0010H and the control is transferred to the respective ISR.

JMP: (Jump unconditionally) usage: JMP Destination

This instruction transfers the control of execution to a location specified directly or indirectly in the instruction rather than the next location just after the JMP instruction.

If the destination is in the same segment in which the JMP instruction is present, this is referred as near jump (or) intrasegment jump. In this case only IP is changed to get the destination location.

If the destination is in the different segment from that of the segment containing JMP instruction, this is referred as far jump (or) intersegment jump. In this case both CS and IP have to be changed to get the destination location.

No flags are affected by JMP instruction.

Ex: JMP BX

JMP label

Conditional branch instructions:

when these instructions are executed, the execution control is transferred to the address specified instruction, provided if a condition is satisfied. If not the execution continues sequentially.

The address has to be specified in the instruction relatively must lie with in -80H to 7FH locations from the address of the

branch instructions. That means, only short jumps are allowed for conditional branch instructions. The following instructions are various Condition -al jump instructions

Mnemonics	Description
1) JC / JB / JNAE Label	Jump if Carry / Jump if below / Jump if not above or equal Transfers the control of execution to Label if CF=1
2) JNC / JNB / JAE Label	Jump if no carry / Jump if not below / Jump if above or equal Transfers the control to Label address if CF=0
3) JP / JPE Label	Jump if Parity / Jump if parity even Transfers the control to Label address if PF=1
4) JNP / JPO Label	Jump if no parity / Jump if parity odd. Transfers the control to Label address if PF=0
5) JA / JNBE Label	Jump if above / Jump if not below or equal. Transfers the control to label address if CF=0 & ZF=0
6) JNA / JBE Label	Jump if not above / Jump if below or equal Transfers the control to label address if CF=1 (or) ZF=1
7) JZ / JE Label	Jump if zero / Jump if equal Transfers the control to label address if ZF = 1.
8) JNZ / JNE Label	Jump if not zero / Jump if not equal Transfers the control to label address if ZF = 0
9) JS Label	Jump if sign Transfers the control to label address if SF = 1
10) JNS Label	Jump if no sign Transfers the control to label address if SF = 0
11) JO Label	Jump if overflow Transfers the control to label address if OF = 1
12) JNO Label	Jump if no overflow Transfers the control to label address if OF = 0.

Mnemonics	Description
13) JG/JNLE Label	Jump if Greater / Jump if not lesser or equal Transfers the control to label address if ZF=0 and SF=OF
14) JNG/JLE Label	Jump if not greater / Jump if lesser or equal Transfers control to label address if ZF=1 or SF≠OF.
15) JL/JNGE Label	Jump if lesser / Jump if not greater or equal Transfers control to label address if SF≠OF
16) JNL/JGE Label	Jump if not lesser / Jump if greater or equal Transfers control to label address if SF=OF.
17) JCXZ Label	Jump if CX=0. Transfers the control to label address if CX=0

The instructions JG/JNLE, JNG/JLE, JL/JNGE, JNL/JGE are used in case of decisions based on signed binary operations. All the remaining instructions are used for unsigned binary operations

iv) Loop instructions:

These type of instructions are mainly two types such as unconditional Loop., conditional Loop.

LOOP (unconditional loop): LOOP label.

This instruction executes a sequence of instructions from a label or address specified in Loop instruction to Loop instruction, CX number of times. At each operation, CX is decremented automatically and the loop continues if ZF=0.

If CX≠0 and ZF=0 the execution of loop is continued. otherwise the next instructions following LOOP instruction are executed.

No flags are affected with the execution of this instruction.

Conditional loop:

LOOPZ/LOOPE (Loop while zero (or) Loop while equal)

This instruction loops through a sequence of instructions

from the address specified at label while ZF=1 and CX≠0

This instruction (LOOPE/ LOOPZ) repeats execution of a loop until ZF=0. The number of times the loop has to be repeated is placed in the CX register. Each time when loop is executed CX is decremented by one.

If CX=0 and ZF=0, the control of execution goes to the next instruction following LOOPE/LOOPZ instruction.

LOOP NE/ LOOP NZ. (Loop while not equal/Loop while not zero)

This instruction loops through a sequence of instructions from the address specified at label, while ZF=0 and CX≠0.

This instruction repeats the execution of a loop until ZF=1. The number of times the loop has to be repeated is placed in CX register. Each time when the loop is executed CX is decremented by one.

v) Machine control instructions:

The machine control instructions supported by 8086 are HLT, LOCK, NOP, ESC and WAIT.

1) HLT: (Halt the processor)

This instruction stops the execution of all the instructions and places the processor in the halt state. The reset signal or an interrupt can cause the processor to resume the execution from the halt state.

2) LOCK: (Bus Lock instruction prefix)

Under multiprocessor environment, the LOCK prefix allows one microprocessor to make sure that another coprocessor doesnot take the control of the system bus, while it is in the execution of a critical instruction using system bus. No flags are affected.

3) NOP: (No operation)

When NOP instruction is executed the processor doesnot perform any operation till 4 T-states. This instruction is used to create a time delay. No flags are affected.

4) ESC : (Escape)

When the ESC instruction is executed, it frees the bus for an external master like a coprocessor or peripheral devices.

Every instruction after ESC instruction, is executed by the coprocessor.

Generally the 8086 fetches the instruction bytes from the memory, the coprocessor catches these bytes from the data bus and put them in a queue. How ever the co processor treats all the instructions as NOP instructions. When 8086 fetches the instruction 'ESC', the coprocessor decodes the instruction and executes it.

5) WAIT : (Wait for $\overline{\text{TEST}}$ pin goes low)

When this instruction is executed, the 8086 micro processor checks the status of $\overline{\text{TEST}}$ pin and if it is at high the processor remains idle until $\overline{\text{test}}$ pin becomes zero. once $\overline{\text{TEST}}$ pin goes low, it continues further execution.

vi) Flag manipulation instructions:

The flag manipulation instructions directly modify some of the flags of 8086. The flag manipulation instructions of 8086 are listed below.

CLC - Clear Carry flag
STC - Set Carry flag
CMC - Complement Carry flag.
CLD - Clear direction flag
STD - Set direction flag
CLI - Clear Interrupt flag
STI - Set interrupt flag.

vii) Shift and rotate instructions:

for these instructions refer logical instructions.

viii) String manipulation instructions:

A series of data bytes or words available in memory locations (at consecutive locations), to be collectively or individually referred as

byte strings or word strings. In each memory location the characters of the string are stored with their equivalent ASCII.

For referring a string, two parameters are required generally They are a) starting or ending address of the string b) length of string. The length of the string is stored in CX register.

SI and DI registers act as pointer registers to DS and ES respectively. That means SI should contain the offset of the first location in DS and DI should have the offset of the first location in ES.

There is a control flag called direction flag, for string operations. If this flag is set the pointer registers SI and DI get automatically decremented and if reset the reverse occurs. So whenever string instructions are being used DF should be set or reset depending on the requirement.

The string manipulation instructions of 8086 are listed below.

REP: (Repeat instruction prefix)

This instruction is used as a prefix to other instructions. The instructions to which REP prefix is provided, is executed until the CX register becomes zero (at each iteration CX is decremented by '1'). When CX becomes zero, the execution proceeds to the next instruction in sequence.

There are two more options of REP instructions. The first one is REPE/REPZ (i.e repeat while equal / repeat while zero), The second one is REPNE/REPNZ (i.e repeat while not equal/ repeat while not zero). These options are used for instructions CMPS, SCAS instruction only, as prefixes.

MOVSB / MOVSW: (Move string byte / Move string word)

Suppose a string of bytes or a string of words are stored in a set of consecutive locations is to be moved to another set of destination locations, this can be done using MOVSB or MOVSW instructions.

source string is located in Data Segment whose offset address (SI) gives the starting address of the source string.

The starting address of the destination string is given by ES : DI.

The MOVSB/MOVSW instruction moves a string byte / a string word pointed by DS:SI to a location pointed by ES:DI. The REP instruction is used with MOVSB/MOVSW to repeat it by a value given in counter CX. The length of the byte string (or) word string must be stored in CX. No flags are affected by this instruction.

Based on the status of direction flag, after MOVSB/MOVSW is executed, SI and DI are automatically up dated and CX is decremented. If DF = 0 then SI, DI are automatically incremented otherwise they are decremented.

Ex:

```
MOV  DS, 5000H
MOV  DS, AX
MOV  AX, 6000H
MOV  ES, AX
MOV  CX, OFFOH
MOV  SI, 1000 H
MOV  DI, 2000H
CLD          ;  DF=0
REP  MOVSB   ;  Move String bytes from source address to
                destination. The number of bytes to be moved
                is OFFOH.
```

CMPSB/ CMPSW - Compare string byte (or) Compare string word:

The CMPSB (or) CMPSW instructions can be used to compare two strings of bytes or words. The length of the string must be stored in the register CX.

The flags are affected in the same manner as that of CMP.

The source string offset should be held by SI which is present in Data segment. (i.e DS:SI)

The destination string is pointed by ES:DI.

The comparison of the string starts from initial byte or initial word of the string. After each comparison the index registers are updated depending on direction flag.

This comparison of byte by byte (or) word by word is done until any mismatch is found. when a mismatch is found the CF and ZF are affected according to the result.

REPE (or) REPNE Prefix instructions can be used with this instruction.

Ex:

```
MOV AX, 5000H
MOV DS, AX
MOV AX, 6000H
MOV ES, AX
MOV CX, 0050H
MOV SI, 1000H
MOV DI, 2000H
CLD
REPE CMPSB ; Compare 50H bytes of 5000H:1000H with 6000H:2000H, repeat while they are equal
```

SCASB/SCASW: (scan string byte (or) string word)

This instruction scans a string of bytes (or) a string of words for a particular byte (or) a word specified in AL (or) AX. The string is pointed by ES:DI. The length of the string must be stored in CX. Based on DF value DI can be incremented or decremented by 1 or 2 for byte and word strings respectively

whenever a match to the specified operand is found in the string the execution is stopped and zero flag is set. If no match is found the zero flag is reset. The REPNE prefix is used with the SCASB / SCASW instruction.

Ex:
```
MOV AX, 5000 H
MOV DS, AX
MOV AX,6000H
MOV ES,AX
MOV CX 0010H
MOV AX,0027H
CLD
REPNE SCASW
```

LODSB / LODSW : (Load string byte or Load string word)

The LODS instruction loads AL/AX register by the content of the string byte or string word pointed by DS:SI. SI is modified automatically based on DF. No flags are affected with this instruction.

If It is LODSB, SI is modified by 1, if it is LODSW, SI is modified by 2.

STOSB/ STOSW :

The STOSB/STOSW is used to store the contents of AL/AX register to a location pointed by ES:DI. The is modified based on DF. No flags are affected with this .ruction.

If it is STOSB, DI is modified by '1', if it is STOSW DI is modified by '2'.

NOTE : If the ending address of the string is available, the auto decrementing mode is preferrable. If the starting address of string is available the auto incrementing mode is preferrable.

ASSEMBLER DIRECTIVES AND OPERATORS :

An assembler is a program that convert an assembly language program into the equivalent machine language program.

The assembler decides the address of each label and substitutes the value for each variable and constants. It then forms the machine language equivalent, for the mnemonics and given data in assembly language program. while doing this the assembler may get some syntax errors. The assembler cannot find logical (or) other programming errors. To do this the assembler needs some hints from the programmer. These type of hints are given to the assembler using some predefined alphabetical strings called Assembler directives, which help the assembler to correctly understand the assembly language programs to prepare machine codes.

Another type of hints that helps the assembler is operator. But the operators perform the arithmetic and logical tasks & unlike assembler directives.

DB, DW, DD, DQ and DT : (Define byte, Define word, Define doubleword, Define quad word and Define Ten bytes)

The assembler directives DB, DW, DD, DQ and DT are used to reserve one byte, one word (2 bytes), one double word (i.e 2 words (or) 4 bytes) one quad word (4 words (8 bytes)) and ten bytes in the memory respectively for storing the constants, variables or strings.

Example

```
DATA1   DB  20H ; reserve one byte for storing DATA1 and
                  assign the value 20H to it.
Array1  DB  10H, 20H, 50H ; reserve three bytes for storing
                  array1 and initialise them with the
                  values 10H, 20H, 50H.
DATA2   DW  5090H ; reserve one word for storing DATA2 and
                  assign the value 5090H to it.
```

```
Array2   DW  513AH, 957BH, 9BC9H ; Reserve three words for storing
                                   Array2 and initialize them with
                                   specified values.

DATA3    DQ  1756A9BED75F123CH ; reserve 4 words for storing Data3
                                 and assign a value to it.

DATA4    DD  127F3A9BH ; reserve 2 words for storing data4 and assign a
                         value 127F3A9BH to it.

DATA5    DT  12345789 9F5E7B8C 5F2AH ; reserve ten bytes for storing
                                       DATA5 and assigns a value to it.
```

The directive `DUP`: is used to reserve a series of bytes (or) words (or) double words (or) ten bytes and is used with DB, DW, DD and DT respectively. The reserved area can be either filled with a specific value or left uninitialized.

Ex:

```
Array   DB  20 DUP (?) ; reserves 20 duplicate locations of bytes
                         and left uninitialized.

Array1  DB  25 DUP (0) ; reserves 25 bytes of locations of bytes
                         each location is initialized with 0.

Array2  DB  50 DUP (64H); reserves 50 bytes of locations of bytes
                          each location is initialized with 64H.
```

ASSUME : (Assume logical segment name)

The `ASSUME` assembler directive is used to inform the assembler, the name of the logical segments to be assumed for different segments used in the program.

```
ASSUME CS: CODE1 , DS: DATA1
```

This statement informs the assembler that the segment addresses of the segments with names CODE1 , DATA1 are stored in CS and DS registers respectively.

END : (End of Program)

The END directive marks the end of an assembly language program when the assembler comes across this END directive, it ignores the next instructions to `END`. So it should be ensured that the

END statement should be the last statement in a program and no useful program should lie in the file after END statement.

ENDP: (END of procedure)

The ENDP directive is used to indicate the end of a procedure. To represent the end of a particular procedure, the name of the procedure should appear as a prefix with the directive ENDP.

Example:

```
Average   PROC NEAR
    :
    :
Average   ENDP
```

ENDS: (END of segment)

This directive indicates the end of a logical segment. The logical segments are assigned with the names using the ASSUME directive. To represent the end of a particular segment, the name of the segment should appear as a prefix with the directive ENDS. What ever be the contents of the segment they should appear in the program before ENDS. Any statement appearing after ENDS will be neglected from the segment.

Ex:

```
DATA SEGMENT
    :
    ;
DATA ENDS
ASSUME CS: CODE, DS: DATA
CODE SEGMENT
    :
    :
CODE ENDS
END
```

EVEN: (Align on Even memory address)

The assembler while starting the assembling procedure of any program, it initializes the location counter and goes on updating it. It goes on assigning the available addresses to the program variables, constants and subroutines as per the requirements.

The directive EVEN updates the location counter to the next even address, if the current location counter content is not even.

Ex: EVEN
Array2 DW 20 DUP(0)

The EVEN directive can also be used at the beginning of a procedure so that the instructions can be fetched quickly by 8086 during execution

```
EVEN
INTEREST PROC NEAR
   :
   :
INTEREST ENDP.
```

EQU: (Equate)

The EQU directive can be used to assign a name to constants. For example the statement NUM EQU 50H directs the assembler to assign a value 50H every time it finds the NUM in the program.

Now if the assembler come across a statement for example MOV AL, NUM, Then it will code it as MOV AL, 50H.

The advantage of using EQU directive is to reduce the recurrence of the numerical values or constants in the program.

EQU directive must be placed at the start of the user program. i.e Assigning values to the constants must be declared at the beginning of the program.

EXTRN: (External) and PUBLIC (public).

The directive EXTRN informs the assembler that the names, procedures and the labels declared after this directive have already been defined in some other segments. While in the other segments where the names, procedures and labels actually appear must be declared as public using the directive 'PUBLIC'.

If one wants to call a procedure Factorial appearing in module1 from module2; in module1 it must have been declared PUBLIC using the statement PUBLIC FACTORIAL and in module2, it must be declared as EXTRN FACTORIAL.

The statement declaration of EXTRN must be accompanied

by SEGMENT and ENDS directives of Module1, before it is called in Module2.

```
Module1  SEGMENT
PUBLIC   FACTORIAL  FAR
Module1  ENDS
Module2  SEGMENT
EXTRN    FACTORIAL  FAR
Module2  ENDS
```

GROUP: (Group the related segments)

This directive is used to form a logical group of segments and it informs the assembler to form a logical group of the following segment names. the assembler passes an information to the linker/loader to form a code such that the group declared segments must lie with the 64 KB of memory segment.

```
Program Group  code, data, stack
```

The above statement directs the loader/linker to prepare an EXE File such that code, data and stack segments must lie with in 64 KB memory segment that is named as program. Now for the Assume statements, one can use that Label program rather than CODE, DATA and STACK.

Ex: ASSUME CS: program, DS: program, SS: program

LABEL: (Label)

The directive LABEL is used to assign a name to the current content of the location counter. When the assembly process starts, the assembler initialises the location counter and it's contents are updated as the assembler goes through the process of assembly. During this if the assembler comes across the directive LABEL, it assigns the declared label with current contents of the location counter.

The type of the label must be specified i.e whether it is a NEAR (or) FAR label, Byte (or) word label etc.

A LABEL directive may be used to make a FAR jump. A FAR jump cannot be made at a label with colon.

Ex: CONTINUE LABEL FAR.

LENGTH: This assembler directive is used to determine the length of a data array or string. This directive is not available in MASM.

MOV CX, LENGTH Array.

This statement substitutes the length of the array in bytes in the instruction.

LOCAL: The labels, variables, constants or procedures declared as LOCAL in a module are to be used only by that particular module. If an another module declares a data type LOCAL which was declared previously in other module, with the same label it may serve for different purposes. with in a single statement a number of variables can be declared as local as shown below.

LOCAL a, b, data, array, routine

NAME: (Logical name of a module)

The NAME directive is used to assign a name to an assembly language program module. The module may now be referred to by it's declared name. If the names are given based on the function of the module, then it will be very helpful in documentation.

OFFSET: (Offset of a label)

when the assembler comes across the operator OFFSET along with a label, it first computes the 16-bit displacement of the particular label and replaces it by the computed displacement This operator can be used with arrays, strings, tables, and procedures to decide their offset addresses in their respective segments.

Ex:

```
DATA SEGEMENT
LIST  db  20H , 30H, 40H , 50H
DATA ENDS
CODE SEGMENT
MOV SI, offset List
CODE ENDS.
```

SEG:

This operator is used to determine the segment address of a label, variable, procedure and substitute the segment address in the place of SEG label.

Ex:

```
MOV AX, SEG LIST ;  Load the segment address in which LIST. is present into AX
MOV DS, AX ;  Move the Content of AX into DS .
```

ORG: (origin)

The ORG directive directs the assembler to start the memory allocation for a particular segment (Data segment (or) CS (or) SS) from the declared offset address in the ORG statement. while starting the assembly process for a module, the assembler initialises the location counter and keeps the track of allotting addresses for the module. If the ORG statement is not written the location Counter is initialized to 0000H. If an ORG 2000H is present at the starting of the code segment of the module, then the code will start from 2000H address in the code segment instead of 0000H.

The ORG directive can also be used with Data segment also.

PROC (procedure) : The PROC directive is used indicate the starting of a named procedure. The procedure whether to be called from the same segment or from the other segment is

specified using NEAR and FAR respectively.

Ex: i) Palindrome PROC NEAR
ii) FIBONACCI PROC FAR

PTR: (The pointer)

The PTR operator is used to declare the type of a label, variable or a memory operand. The operator PTR is is prefixed by BYTE (or) WORD. If the prefix is BYTE, it indicates a particular label (or) variable (or) memory operand as an 8 bit quantity. while if WORD is the prefix, then it is treated as a 16-bit quantity.

In otherwords, the PTR operator is used to specify the data type - byte or WORD

Ex: MOV AL, BYTE PTR [SI] ; Moves 8 bit Content of memory location addressed by SI to AL

MOV CX, WORD PTR [BX] ; - Moves the 16 bit Contents of memory location addressed by BX to CX.

INC BYTE PTR [BX] ; Increment the Content of memory location addressed by BX.

In the case of JMP instructions, the PTR operator is used to specify the type of the JUMP i.e near or far.

JMP NEAR PTR [BX] ; Near Jump

JMP FAR PTR [BX] ; far jump.

PUBLIC: This directive is used along with EXTRN directive. This informs the assembler that the labels, variables, Constants, or procedures declared as PUBLIC may be used in another source modules also, but while using the labels declared as PUBLIC, the user must declare them using EXTRN directive.

Ex:

```
MODULE1  SEGMENT
  PUBLIC   FACTORIAL FAR
MODULE1  ENDS
MODULE2  SEGMENT
  EXTRN    FACTORIAL FAR
MODULE2  ENDS .
```

SEGMENT : (Logical segment)

This directive indicates the starting of a logical segment. The Each logical segment is identified by a name preceeding the SEGMENT directive.

For example DATA SEGMENT, indicates, the assembler the starting of a logical segment whose name is DATA.

SEGMENT and ENDS directives are used at the beginning and ending of a logical segment.

SHORT : The short operator indicates to the assembler that only one byte is required to get the displacement for jump (i.e displacement is within -128 to +127 bytes from the address of Jump instruction).

This method of specifying the displacement saves memory. otherwise the assembler may require two bytes for the displacement.

Ex: JMP SHORT AGAIN.

TYPE : The type operator directs the assembler to decide the data type of the specified label and replaces the 'TYPE Label' by data type value. For word type variable, data type value is 02, for double word type the data type value is 4 and for byte type the data type value is '1'.

Ex: MOV AX, TYPE NUM ; moves the value 0002 in AX as the variable NUM belongs to word in this example

GLOBAL: The labels, variables, constants or procedures declared as GLOBAL may be used by other modules of the program. once variable is declared GLOBAL, it can be used by any module in the program.

The following statement declares a procedure of name ROUTINE as a global procedure.

```
ROUTINE PROC GLOBAL.
```

'+ & -' operators: These operators represent arithmetic addition and subtraction respectively and are typically used to add or subtract displacements of 8bit or 16 bit to base (or) to index registers etc.

Ex:
```
MOV AL, [SI+2]
MOV DX, [BX-5]
```

FAR PTR: This directive indicates the assembler that the label following FAR PTR is not available with in the same segment and the address of the label is of 4 bytes (2 bytes of offset followed by 2 bytes of segment address) i.e 32 bit address.

Ex:
```
JMP FAR PTR OVER.
CALL FAR PTR ROUTINE
```

Both the above instructions indicate the assembler that the target address is going to require 4 bytes lower byte of offset, higher byte of offset, lower byte of segment and higher byte of segment. This indicates the mode of addressing is intersegment addressing mode.

NEAR PTR: This directive indicates that the label following NEAR PTR is available with in the same segment and the address of the label is of 2 bytes (16 bit). i.e offset address.

Ex: JMP NEAR PTR ROOT

CALL NEAR PTR SQUARE .

NOTE: If a label is not preceded by NEAR PTR, then it is by default considered as NEAR PTR and two bytes of address is required for that label.

8086 Programming Development Steps:-

The development steps for an 8086 program involve problem definition, creating an algorithm and flowchart to represent the solution, writing the assembly language source code using the 8086 instruction set, assembling it into machine-readable object code, linking to combine different code modules, loading into memory, and finally executing and debugging the program.

1. Problem Definition & Algorithm Development

Define the Problem: Clearly understand the task the program needs to accomplish.

Develop an Algorithm: Create a step-by-step logical procedure to solve the problem.

Create a Flowchart: Visually represent the algorithm's steps using standard symbols.

2. Coding & Tool Usage

Write Source Code: Use a text editor to write the 8086 assembly language program.

Choose the Right Instructions: Select appropriate 8086 instructions (e.g., data transfer, arithmetic, logic, control flow) to implement your algorithm.

Use an Assembler: An assembler translates your assembly source code into machine-readable object code.

3. Linking & Loading

Link Object Modules:

If your program consists of multiple modules, a linker combines them into a single executable file.

Load into Memory:

A loader places the executable program into the computer's memory for execution.

4. Execution & Debugging

Execute the Program: The processor runs the program from memory.

Debug: Use a debugger to identify and correct errors (bugs) in the program.

5. Documentation

Document the Program: Properly comment and document your assembly language code to explain its functionality.

Assembly Language Program Development Tools:-

Developing assembly language programs typically involves a suite of specialized tools that facilitate the entire development cycle, from writing the code to debugging and execution. These tools include:

- **Editor:**

A text editor is used to create and modify the source code file containing the assembly language instructions. These files are typically saved with an extension like .asm.

- **Assembler:**

This is a crucial tool that translates the assembly language mnemonics and directives into machine code (binary instructions) that the processor can understand. Popular assemblers include NASM (Netwide Assembler), MASM (Microsoft Macro Assembler), and TASM (Turbo Assembler). The assembler generates an object file (.obj) and often a listing file (lst) which shows the assembly code alongside its corresponding machine code.

- **Linker:**

When a program is composed of multiple object files or utilizes external libraries, the linker combines these separate modules and resolves references between them to create a single, executable file (.exe).

- **Locator (or Loader):**

For embedded systems or specific memory models, a locator assigns absolute memory addresses to the program's code and data segments, determining where the program will reside in memory when executed.

- **Debugger:**

A debugger is an essential tool for identifying and resolving errors in a program. It allows developers to execute the program step-by-step, set breakpoints, examine the contents of registers and memory locations, and trace the program's execution flow.

UNIT-3: 8086 Interfacing

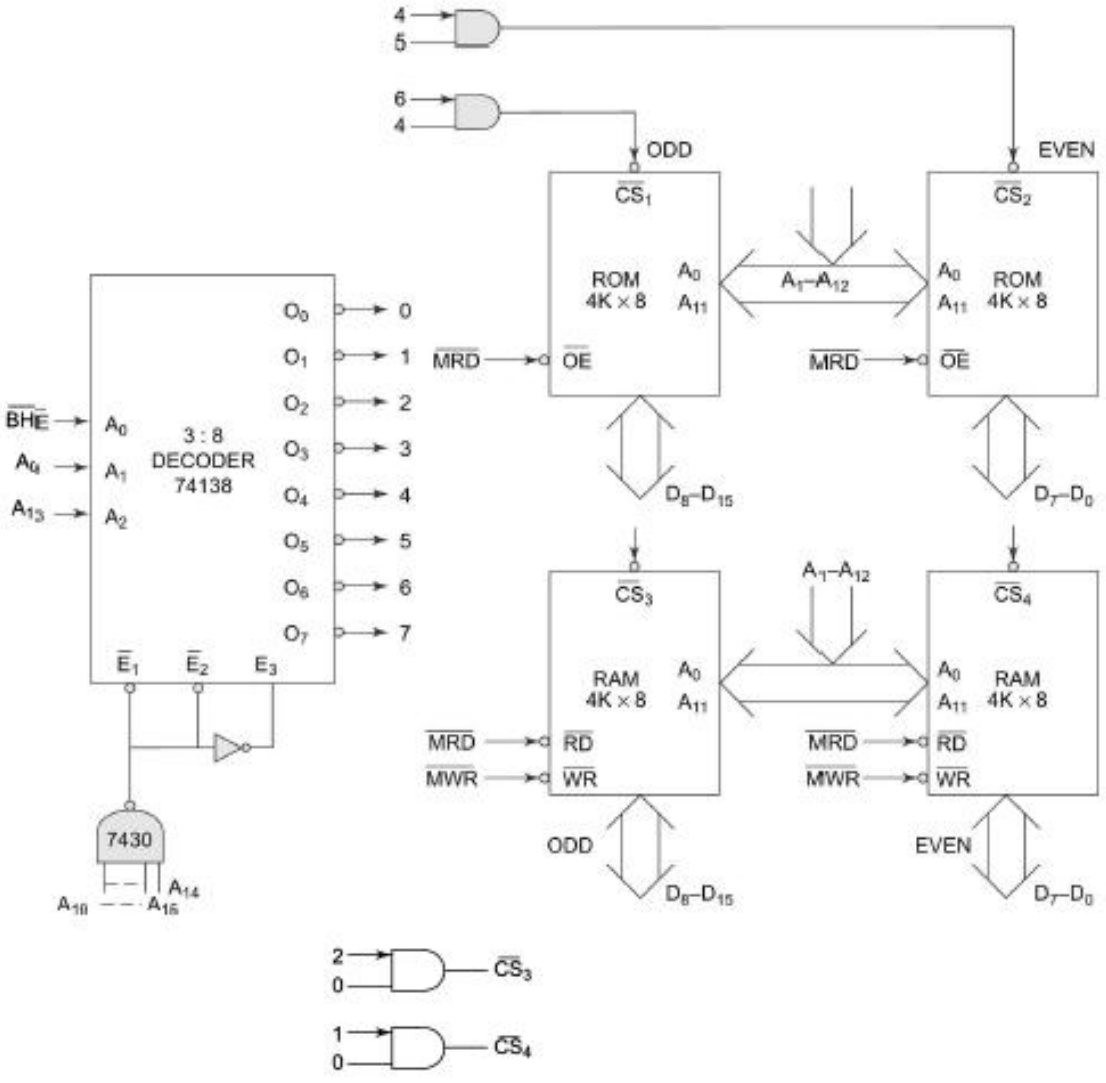
Semiconductor Memories Interfacing:-

Interface two 4K × 8 EPROMS and two 4K × 8 RAM chips with 8086. Select suitable maps.

Solution We know that, after reset, the IP and CS are initialised to form address FFFF0H. Hence, this address must lie in the EPROM. The address of RAM may be selected anywhere in the 1MB address space of 8086, but we will select the RAM address such that the address map of the system is continuous, as shown in Table.

Table *Memory Map for Problem*

Address	A_{19}	A_{18}	A_{17}	A_{16}	A_{15}	A_{14}	A_{13}	A_{12}	A_{11}	A_{10}	A_{09}	A_{08}	A_{07}	A_{06}	A_{05}	A_{04}	A_{03}	A_{02}	A_{01}	A_{00}
FFFFFH	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
						EPROM							8K × 8							
FE000H	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
FDFFFH	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1
						RAM							8K × 8							
FC000H	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0



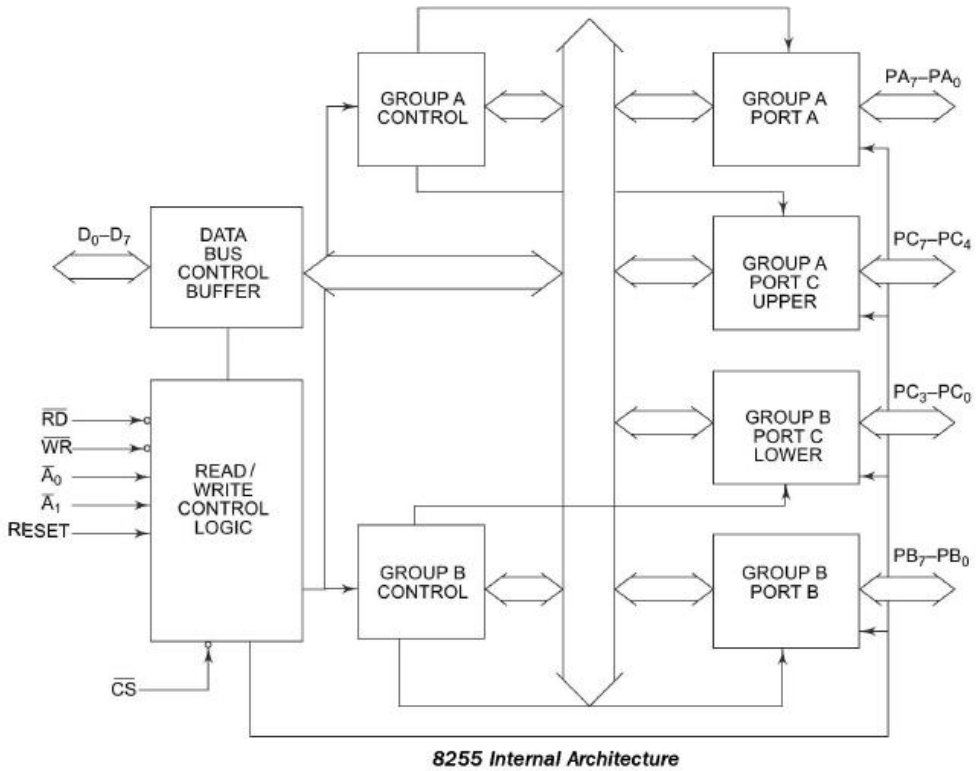
Total 8K bytes of EPROM need 13 address lines A0-A12 (since 213 = 8K). Address lines A 13-A 19 are used for decoding to generate the chip select. The BHE signal goes low when a transfer is at odd address or higher byte of data is to be accessed. Let us assume that the latched address, BHE and demultiplexed data lines are readily available for interfacing. Figure shows the interfacing diagram for the memory system.

The memory system in this example contains in total four 4K × 8 memory chips. The two 4K × 8 chips of RAM and ROM are arranged in parallel to obtain 16-bit data bus width. If Ao is 0, i.e. the address is even and is in RAM, then the lower RAM chip is selected indicating 8-bit transfer at an even address. If A_0 is 1, i.e. the address is odd and is in RAM, the BHE goes low, the upper RAM chip is selected, further indicating that the 8-bit transfer is at an odd address. If the selected addresses are in ROM, the respective ROM chips are selected. If at a time A, and BHE both are 0, both the RAM or ROM chips are selected, i.e. the data transfer is of 16 bits. The selection of chips here takes place as shown in Table.

Table *Memory Chip Selection*

Decoder I/P → *Address/ $\overline{BHE}$ →*	A_2 A_{13}	A_1 A_0	A_0 $\overline{BHE}$	*Selection/* *Comment*
Word transfer on $D_0 - D_{15}$	0	0	0	Even and odd addresses in RAM
Byte transfer on $D_7 - D_0$	0	0	1	Only even address in RAM
Byte transfer on $D_8 - D_{15}$	0	1	0	Only odd address in RAM
Word transfer on $D_0 - D_{15}$	1	0	0	Even and odd addresses in ROM
Byte transfer on $D_0 - D_7$	1	0	1	Only even address in ROM
Byte transfer on $D_8 - D_{15}$	1	1	0	Only odd address in ROM

8086 Interfacing with Intel 8255 Programmable Peripheral Interface:-



8255 Internal Architecture

The Intel's 8255 is designed for use with Intel's 8-bit, 16-bit and higher capability microprocessors. It has 24 input/output lines which may be individually programmed in two groups of twelve lines each, or three groups of eight lines. The two groups of I/O pins are named as Group A and Group B. Each of these two groups contain a subgroup of eight I/O lines called as 8-bit port and another subgroup of four I/O lines or a 4-bit port. Thus, Group A contains an 8-bit port A along with a 4-bit port, C upper. The port A lines are identified by symbols PA7- PA0 while the port C lines are identified as PC4-PC7. Similarly, Group B contains an 8-bit port B, containing lines PB7- PB0 and a 4-bit port C with lower bits PC-PC3. The port C upper and port C lower can be used in combination as an 8-bit port C. All of these ports can function independently either as input or as output ports. This can be achieved by programming the bits of an internal register of 8255 called as Control Word Register (CWR).

The 8-bit data bus buffer is controlled by the read/write control logic. The read/write control logic man-ages all of the internal and external transfers of both data and control words. $\overline{RD}$, $\overline{WR}$, $\overline{A1}$, $\overline{A0}$ and RESET are the inputs, provided by the microprocessor to the READ/WRITE control logic of 8255. The 8-bit, 3-state bidirectional buffer is used to interface the 8255 internal data bus with the external system data bus. This buffer receives or transmits data upon the execution of input or output instructions by the microprocessor. The control words or status information is also transferred through the buffer.

PA7-PA0: These are eight port A lines that act as either latched output or buffered input lines depending upon the control word loaded into the control word register.

PC7-PC4: Upper nibble of port C lines. They may act as either output latches or input buffers lines. This port also can be used for generation of handshake lines in mode l or mode 2.

PC3-PC0: These are the lower port C lines, other details are the same as PC7-PC4 lines.

PB7-PB0: These are the eight port B lines which are used as latched output lines or buffered input lines in the same way as port A.

$\overline{RD}$: This is the input line driven by the microprocessor and should be low to indicate read operation to 8255.

$\overline{WR}$: This is an input line driven by the microprocessor. A low on this line indicates write operation.

$\overline{CS}$: This is a chip select line. If this line goes low, it enables the 8255 to respond to RD and WR signals, otherwise RD and WR signals are neglected.

$\overline{A1}$-$\overline{A0}$: These are the address input lines and are driven by the microprocessor. These lines (A1-A0) with $\overline{RD}$, $\overline{WR}$ and $\overline{CS}$ form the following operations for 8255. These address lines are used for addressing any one of the four registers, i.e. three ports and a control word register.

$\overline{RD}$	$\overline{WR}$	$\overline{CS}$	A_1	A_0	*Input (Read) cycle*
0	1	0	0	0	Port A to data bus
0	1	0	0	1	Port B to data bus
0	1	0	1	0	Port C to data bus
0	1	0	1	1	CWR to data bus

$\overline{RD}$	$\overline{WR}$	$\overline{CS}$	A_1	A_0	*Output (Write) cycle*
1	0	0	0	0	Data bus to Port A
1	0	0	0	1	Data bus to Port B
1	0	0	1	0	Data bus to Port C
1	0	0	1	1	Data bus to CWR

Modes of Operation of 8255:

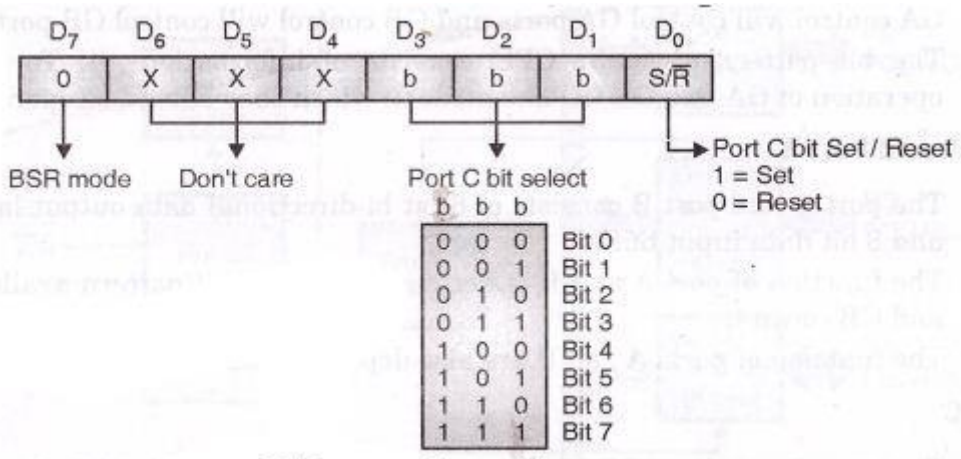
There are two basic modes of operation of 8255:

1. Bit Set-Reset mode (BSR).
2. I/O mode

In the I/O mode, the 8255 ports work as programmable I/O ports, while in BSR mode only port C(PC-PC7) can be used to set or reset its individual port bits. Under the IO mode of operation, further there are three modes of operation of 8255, so as to support different types of applications, viz, mode 0, mode 1 and mode 2.

1. Bit Set-Reset mode (BSR)

In this mode, any of the 8-bits of port C can be set or reset depending on D0 of the control word. The bit to be set or reset is selected by bit select flags D3, D2 and D1 of the CWR as shown in below figure.



BSR control word format

2. I/O Mode

This mode is selected when the D7 bit of the control register is 1. This mode has also three different modes. These modes are Mode 0 and Mode 1 and Mode 3.

Mode 0 - Simple or basic I/O Mode

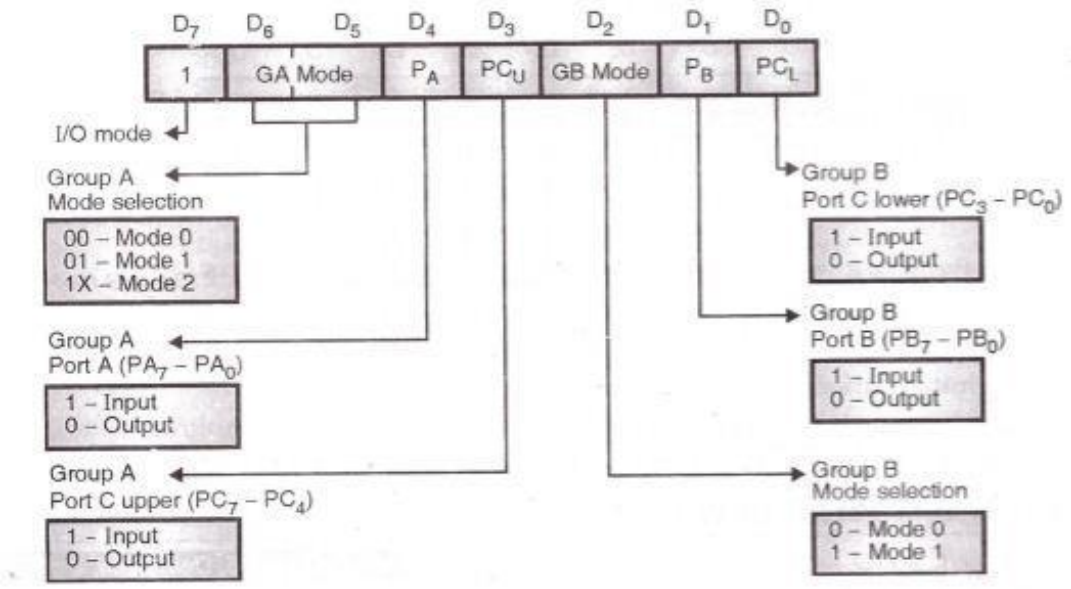
In this mode all of the ports A, B and C can be used as input or output mode. The outputs are latched, but inputs are not latched. This mode has interrupt handling capability.

Mode 1 - Handshake or Strobed I/O

In this mode the Port A and Port B can be used as input or output ports, the port C are used for handshaking. In this mode the inputs and outputs are latched. This mode also has the interrupt handling capability, and signal control to match the speed of CPU and IO devices.

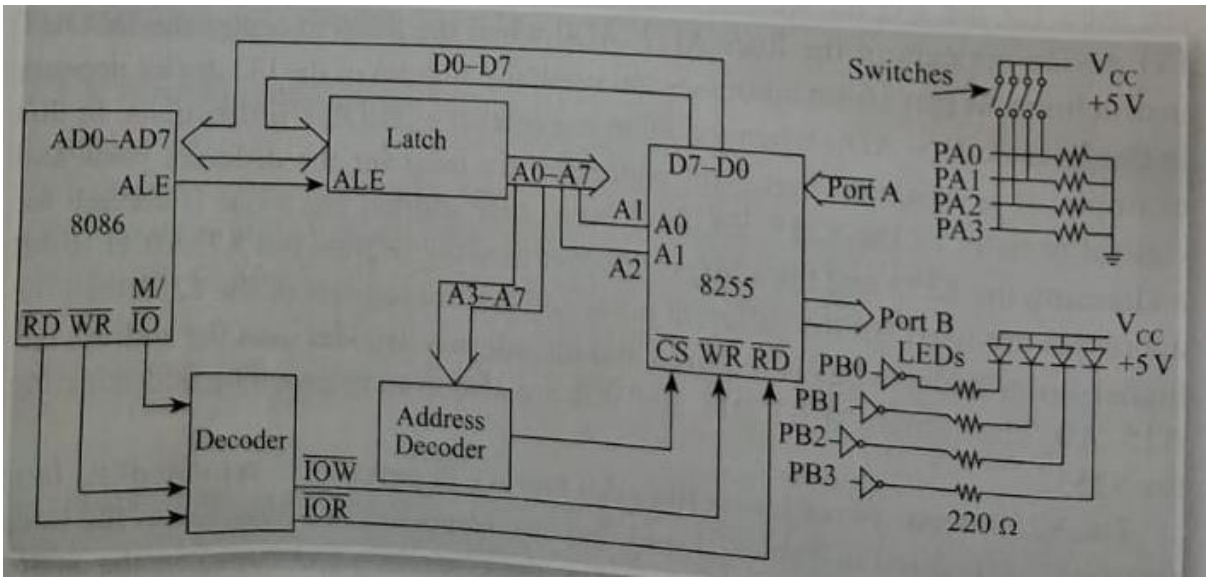
Mode 2 - Bidirectional I/O

In this mode only Port A can work, and port B can either be in mode 0 or mode 1, and the port C are used for handshaking. In this mode the inputs and outputs are latched.



I/O modes control word format

<u>Interfacing Switches and LED's</u>:-



In this, we discuss the interfacing of four switches and four LEDs with the 8086 through the 8255. Data is obtained from the switches and displayed using the LEDs.

The 8255 is interfaced with the 8086, with the 8255 ports connected to the switches and LEDs. A latch is used to demultiplex the lower address bus and the data bus (AD7-AD0). In the 8086-based system, either 8-bit or 16-bit addresses are used for the I/O devices. When 8-bit addresses are used, the address of the I/O device appears in the lines AD7-AD0 when the 8086 executes the IN/OUT instructions. When 16-bit addresses are used, the address of the I/O device appears in the lines AD15-AD0 when the 8086 executes the IN/OUT instructions. In this example, it is assumed that 8-bit addresses are used for the different ports and control register of the 8255. So, the lower-order address bus alone is enough for addressing the 8255 and the address decoder uses the address bus A7-A0. If 16-bit addresses are used for the different ports and control register of the 8255 then the higher-order address bus is required and the address decoder uses the address bus A15-A0. The signals M/IO, RD, and WR are also used in decoding and selecting the 8255.

The lines A0 and Al of the 8255 are connected to lines Al and A2 of the 8086. The IOR and IOW signals from the 8086 are connected to the RD and WR control signals of the 8255, respectively.

The above figure shows the interfacing of the switches and LEDs with the 8255 through the 8086. The four switches in figure are connected to the lower-order four bits of port A of the 8255. The switch connection is such that when it is open, it connects logic 0, that is, 0 volts to the port and when it is closed, it connects logic 1, that is, 5 volts to the port pins. These connections ensure that the port is not damaged and also not sourcing over current. This ensures safe operation of the ports and switches. The interfacing of four LEDs through an inverter (which acts as a driver) to the ports is shown in figure. When logic 1 is given on the port pin, it will be inverted by the inverter and will connect ground (logic 0) to the cathode of the LED. This will forward bias the LED and light will be emitted by the LED. This connection ensures that the port pin is not sourcing enormous current and also the current required for the LED illumination is from the supply and the driver IC.

Interfacing Seven-Segment Displays:

Seven-segment light emitting diode displays are the most commonly used low-cost displays and are easy to interface with microprocessors. Seven-segment displays consist of seven LED segments. The below figure shows the arrangement of these seven and the appearance of the various digits. Seven-segment displays are available in a single dual in-line package (DIP). There is one pin for each segment and these pins are named from 'a' to 'f' and another LED is available for the decimal point (dp). In addition to these eight pins, the seven-segment displays have one more pin for power supply. Seven-segment displays come in two types common anode and common cathode.

In common anode display, the anodes of all segment LEDs are connected together. So, to illuminate a segment, the common anode is connected to the supply and then the segment input, that is, 'a' to 'f' is connected to a low-level voltage or logic 0. In common cathode display,

the cathodes of all the LEDs are connected together. So, to illuminate a segment, the corresponding segment input is connected to the high-level voltage or logic 1 and the common cathode is connected to the ground. This forward biases the LEDs and illuminates them.

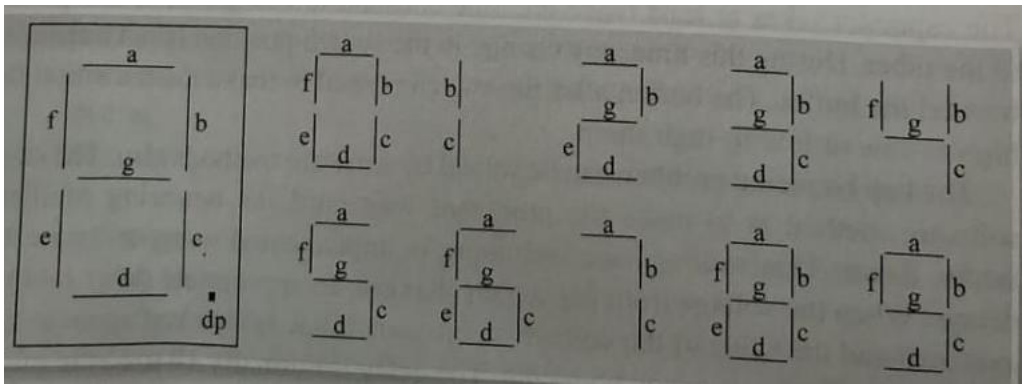


Fig: Arrangement of LEDs and appearance of digits in seven-segment displays

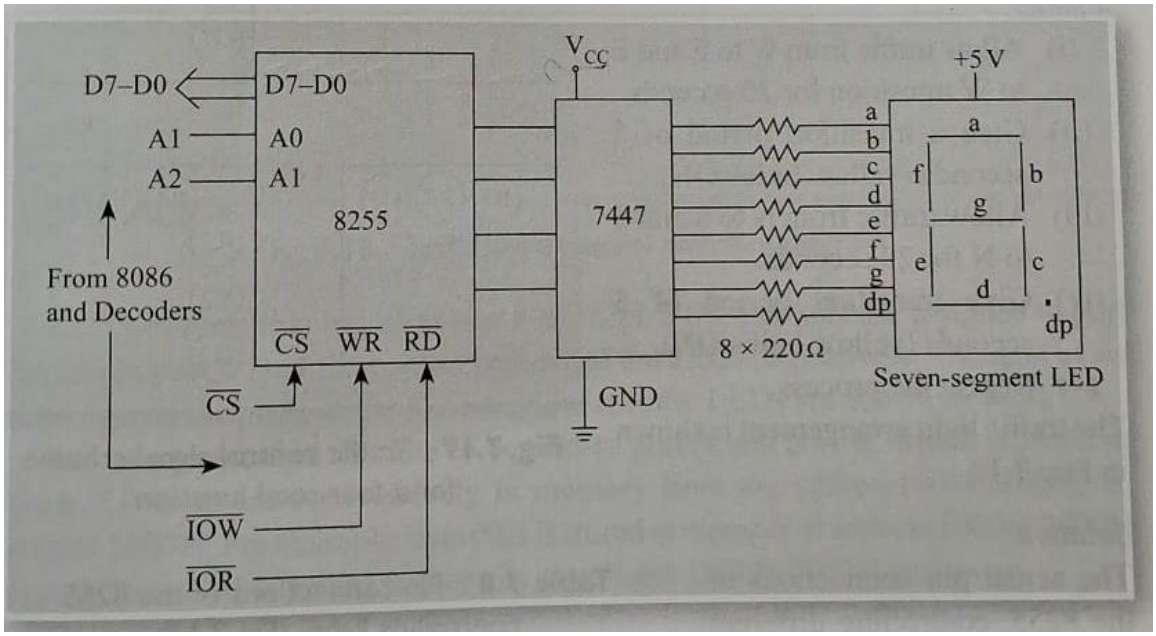


Fig: Interfacing seven -segment display

Intel 8251 USART Architecture and interfacing: -

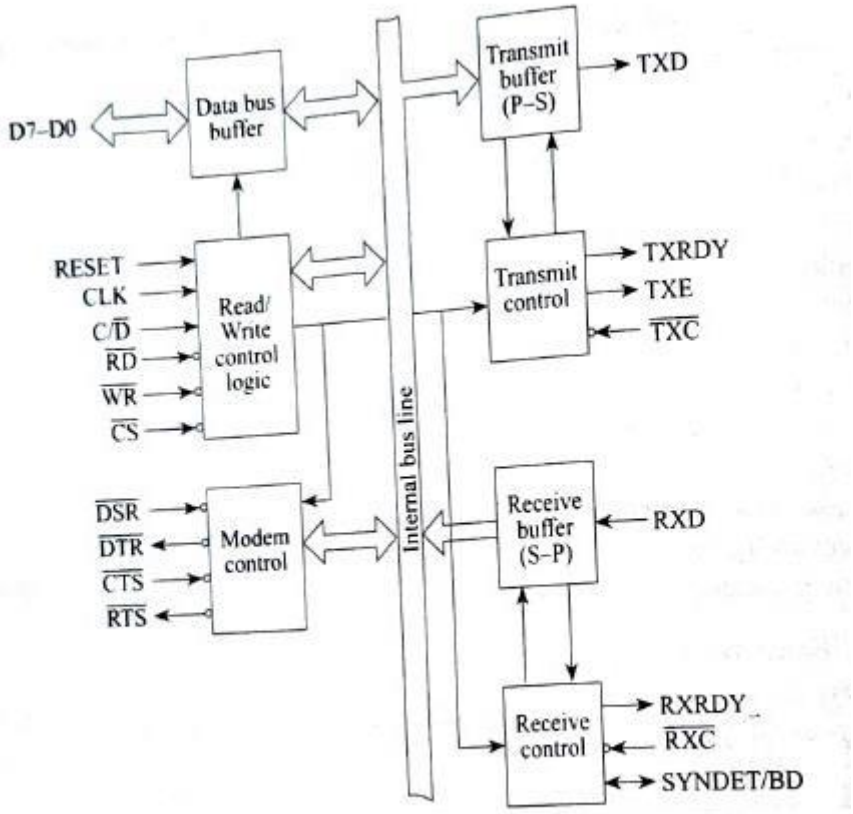


Fig. Block diagram of the 8251 USART

Table Basic operations of 8251 and related control signals

$\overline{CS}$	C/$\overline{D}$	$\overline{RD}$	$\overline{WR}$	Function
1	X	X	X	Chip not selected; data bus in high impedance state
0	X	1	1	Data bus in high impedance state
0	1	0	1	Status word read by CPU from status register
0	1	1	0	Control word written into control register by CPU
0	0	0	1	Data read by CPU from data register
0	0	1	0	Data written into data register by CPU

The 8251 has 28 pins. The details and functions of these pins are listed as follows:

(i) Data bus (D0-D7): A group of bi-directional lines that are used for data and control word transfer between the CPU and the 8251.

(ii) Reset: An active high signal applied on this pin brings the 8251 into reset state. The device after reset waits for the writing of the mode instruction.

(iii) CLK: A clock signal is used to generate internal device timing. It is independent of receive clock ($\overline{RXC}$) or transmit clock ($\overline{TXC}$) In general, the CLK frequency must be much higher than the $\overline{RXC}$ and $\overline{TXC}$ frequencies.

(iv) Write data/command ($\overline{WR}$) It is an active low input signal for writing data and control words from the CPU into the 8251.

(v) Read data (RD): It is an active low input signal for reading data and status words from the 8251.

(vi) Control/data (C/ $\overline{D}$) It is an input signal for selecting data or command/status words when the 8251 is accessed by the CPU. If the C/D data are accessed. If the C/$\overline{D}$=1 command word or status word are accessed.

(vii) Chip select (CS): It is an active low input signal, which selects the 8251 for CPU accesses.

(viii) Transmit data line (TXD): It is an output signal for transmitting converted serial data from the 8251. The device is in mark status (high-level) after resetting or during a status, when transmit is disabled.

(ix) Transmitter ready (TXRDY): It is an output signal, which indicates that the 8251 is ready to accept a data character for transmission. However, the terminal is always at a low level if clear to send ($\overline{CTS}$)=1 or the device is set in transmitter disable status by a command.

(x) Transmitter empty (TXEMPTY): It is an output signal that indicates that the 8251 has transmitted all the characters and had no data character for transmission.

(xi) Transmitter clock ($\overline{TXC}$) This is a clock input signal that determines the transfer speed of the transmitted data or in other words, the baud rate for transmission.

(xii) Receive data (RXD): A signal line that receives serial data.

(xiii) Receiver ready (RXRDY): It is a signal that indicates that the 8251 contains a character that is ready to be read and the CPU can read the data.

(xiv) Receiver clock ($\overline{RXC}$): This is a clock input signal that determines the transfer speed of received data or the baud rate of reception.

(xv) Sync detect/break detect (SYNDET/BD): It is an active high output signal. In asynchronous mode, it is used to indicate a data break. In synchronous mode, it is used to indicate the correct receipt of synchronous characters and the next data is to be received.

The following signals are used with a modem for handshaking and establishing connection.

(i) Data set ready ($\overline{DSR}$): This is an input signal to the 8251 from the modem interface. $\overline{DSR}$ indicates that the modem is powered up.

(ii) Data terminal ready ($\overline{DTR}$): This is an output signal from the 8251 for the modem interface. $\overline{DTR}$ indicates that the modem is powered up.

(iii) Clear to send data ($\overline{CTS}$): This is an input signal to the 8251 from the modem interface, which is used for controlling the transmit circuit.

(iv) Request to send ($\overline{RTS}$): Active low output signal sent to the modem by the 8251, indicating that it is ready to send data.

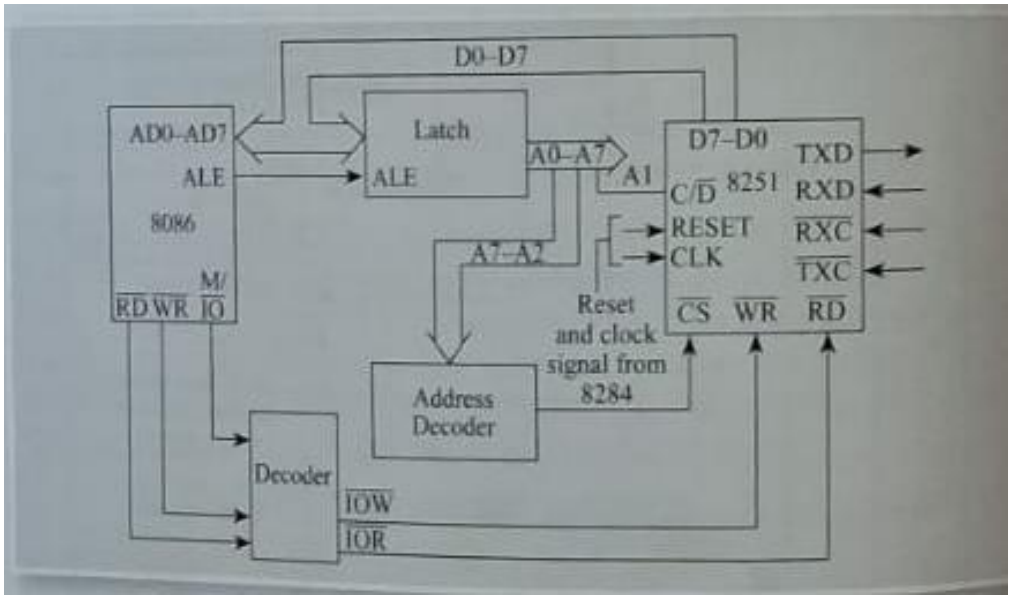


Fig: Interfacing the 8251 with the 8086

The basic interfacing diagram for interfacing the serial port IC 8251 with the processor 8086 is shown in above figure.

The data bus lines D0-D7 are connected to the data lines of the 8251. The higher address lines are used for address decoding and selection. The chip selection signal $\overline{CS}$ is generated using a proper address decoder. The A0 line is connected to the C/$\overline{D}$ line of the 8251 to select either a control word or a data word. The read and write control signals are connected to the corresponding signals of the 8251 The reset and clock output signals from the 8284 are connected to the reset and clock inputs of the 8251. In addition, the 8251 needs separate clock signals for transmission and reception. These RXC and TXC clock signals can be obtained by dividing the clock output from the 8284 or 8253. Normally, two computer systems can be interconnected using the serial port. In such communication, the TXD signal of one system is connected to the RXD line of another system and vice versa. Care must be taken to ensure that the transmit clock of the transmitting computer is the same as the receive clock of the receiving computer.

The serial transmission is started by writing the data to be transmitted into the data register of the 8251. The serial shifting of data into the TXD line starts Immediately. Once the transmission of data is over, the 8251 asserts the TXRDY signal informing the processor that the 8251 is ready for transmission of next data. The programmer can also read the status word for checking whether the next data can be written in the data register for transmission.

Interfacing Stepper Motor: -

A stepper motor is a special motor that rotates in incremental steps, unlike other motors that run continuously. They find application in printers, plotters, robots, etc. Stepper motors are excited by pulses to get incremental displacements. Stepper motors are made of permanent magnet rotors with stator field excitation. Two-phase excitation and four-phase excitation are common. A four-phase stepper motor has four stator poles, which are excited by pulses. Each pole winding can be excited such that the pole can be made either a north pole or a south pole. In this arrangement, the poles should be properly excited in a particular sequence so that the rotor rotates in a particular direction. If the excitation sequence is reversed, the rotor rotates in the reverse direction.

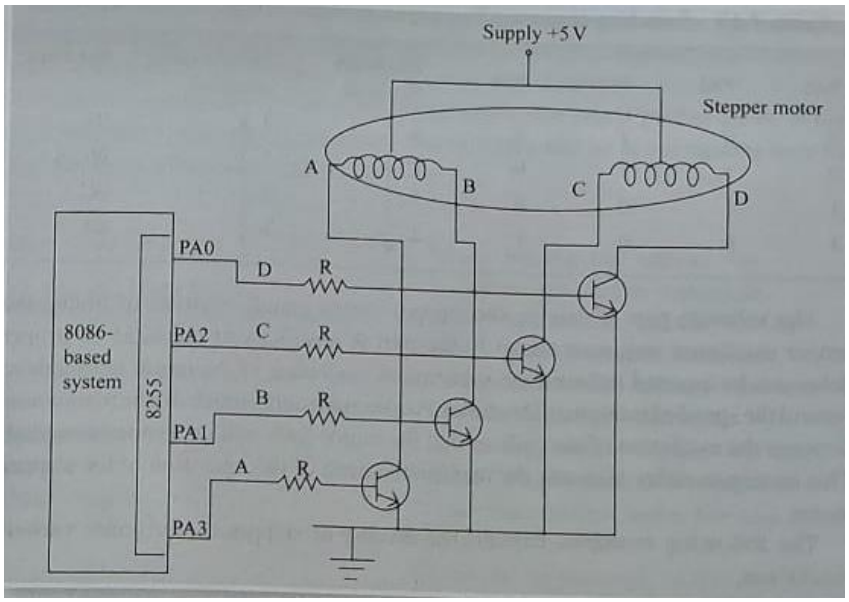


Fig: Interfacing of Stepper Motor using 8255

The stepper motor has six terminals-four terminals A, B, C and D for excitation and two more terminals for power supply. The above figure shows the four terminals A, B, C, and D connected to the 8255 ports through the transistor drivers. The transistor drivers or buffers are essential as the port pins cannot directly source the current required for the motor drive. As explained earlier, the motor terminals have to be excited in a proper sequence, so that rotor will have continuous rotation in one direction. Two types of excitation are possible with a four-phase motor-one-phase excitation and two-phase excitation. In one-phase excitation, only one phase of the stepper motor is excited at a time and in two-phase excitation, two phases are excited at a time. The exciting sequence is fixed for a rotation in a particular direction. The excitation sequence for the interface diagram in above figure is given in below tables. The single phase excitation results in low current through the motor windings and it is also called wave mode. In two-phase excitation, the excitation current through the motor winding is high and so it is called high-torque excitation. The following tables also show the corresponding hexadecimal bytes value to be given to the port A assuming that the higher-order Four bits of the data are zero.

Table : Switching sequence: One-phase excitation (Wave mode)

PA3	PA2	PA1	PA0	Clockwise sequence	Anti-clockwise sequence	Hex value
0	0	0	1	1	4	01
0	0	1	0	2	3	02
0	1	0	0	3	2	04
1	0	0	0	4	1	08

Interfacing Analog to Digital Converters: -

The basic function of the analog-to-digital converter (ADC) is to convert the input analog voltage levels into corresponding discrete digital signals. An ADC is essential in a microprocessor-based system as the microprocessor can only handle digital data, though the real-world signals are all in analog form only.

There are many types of ADC. The major ones are counter ramp type ADC, dual slope ADC, flash type ADC, and successive approximation type ADC. Each type of ADC has its own advantages and disadvantages. Successive approximation type ADC is a commonly available ADC. This ADC has fixed conversion time for any analog input voltage level.

The specifications of the ADC are the range of analog input voltage, number of digital bits at the output, resolution, the conversion time, and the number of analog input channels. The analog input voltage can be either unipolar or bipolar. Unipolar means that the input voltage can have only one polarity such as 0 to +5V or 0 to +10 V. Bipolar means that the input voltage can range from one polarity to the other such as -5 to +5 V or 10 to +10 V. Most of the ADC chips come with an option of selecting one of these voltage ranges using the Vref input pins. The ADC chips are available for different number of output binary bits. ADCs are available with 8, 10,12 or 16-bit digital outputs. The number of bits will decide the number of voltage levels sensed. For example, an 8-bit ADC will have 28 possible levels, that is, 256 levels. The number of bits and the input voltage range will decide the resolution. The resolution of an ADC is defined as the smallest change in the input voltage that can be detected at the output.

We discuss the operation and interfacing of ADC 0816 with the 8086 microprocessor through the 8255 PPI. ADC 0816 is an 8-bit successive approximation type ADC chip with an in-built analog multiplexer, which can select one of 16 analog inputs for conversion into digital format. One of 16 analog inputs IN0-IN15 in the ADC0816 chip can be selected by the select lines A, B, C. and D. The analog to digital conversion can be started by using the active high control signal Start Conversion (SC). The conversion of the analog voltage on the input channel selected, will then take place based on the clock signal applied to the ADC chip. After the conversion is over, the ADC chip will issue an active high end of conversion (EOC) signal on the EOC line. The digital output can then be read from the data lines after issuing an active high Output Enable (OE) signal on the OE line in the ADC chip.

The interfacing of ADC 0816 with the 8255 is shown in below figure. The 8255 PPI is in turn interfaced with the 8086 as shown in figure. In the interfacing diagram shown in figure, it can be noted that the port A of the 8255 is used to output or send the channel select lines and the related control signals. The port B lines are used to get or input the digital result data from the ADC chip. The LSB of port C (i.e., PC0) is used to check the end of conversion signal. With this hardware arrangement, the ADC chip can only be interfaced with software polling method. For interrupt driven interface, the EOC signal can be connected to any interrupt input. Analog inputs can be applied to the analog input pins of the ADC 0816.

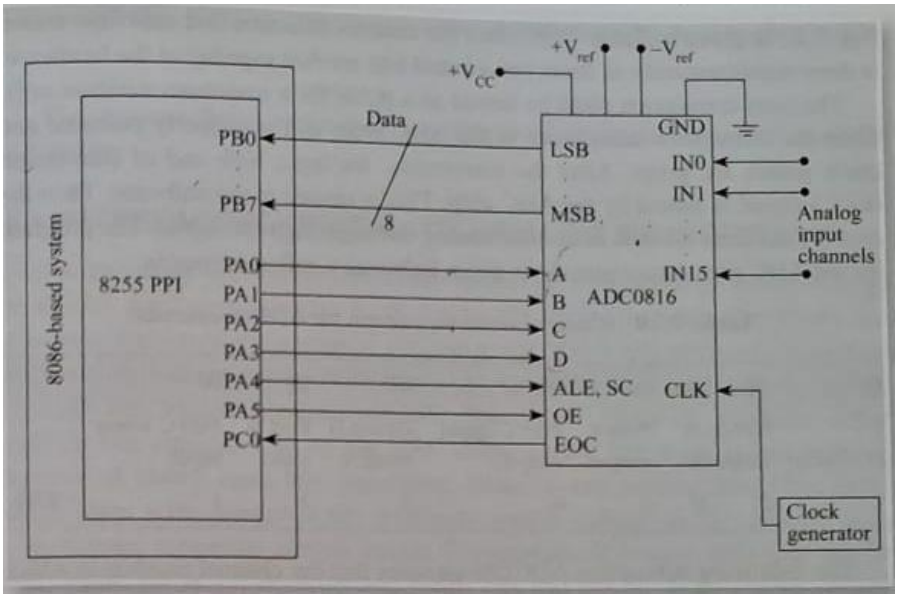


Fig: Interfacing ADC0816 with 8255

Interfacing Digital to Analog Converters: -

Digital-to-analog converters (DACs) are used to get a proportional analog voltage or current for the digital data given out by the microprocessor. The DACs are essential in microprocessor-based systems as the real-world applications operate with analog data. Basically, there are two types of DACs. They are R-2R ladder network and weighted resistor network. Many DAC chips are available in the market. The specifications of the DAC chips are the full scale output voltage, number of binary input bits, resolution, linearity, and settling time. The DAC chips come with choices in the maximum output voltage as 5V, 10V, or with a predefined maximum current output. The number of binary input bits can be 4, 8, 10, or 12. Both the number of bits and the full scale output voltage will determine the resolution.

The DAC 0800 is a common digital to analog converter chip that can be easily interfaced to the 8086 through the 8255 as shown in the below figure. This section will describe the interfacing of the DAC 0800 with the 8086 processor. As the DAC chips can be connected only to a port, there is a need for an output port connected between the processor and DAC. The 8255 can act as an output port to give data from the processor to the DAC chip. One port is enough to interface an 8-bit DAC with the 8255. The interfacing diagram in Fig. 7.21 uses port A of the 8255 for connecting the data to DAC 0800. The other control signals are directly correspondingly connected to either logic 0 or logic 1. The DAC chip gives a proportional

current output. This current output in most cases is difficult to measure and so a current to voltage (I to V) converter is used at the output. DAC chips have an in-built latch. This latch stores the digital input given by the port A and the DAC gives out a proportional voltage.

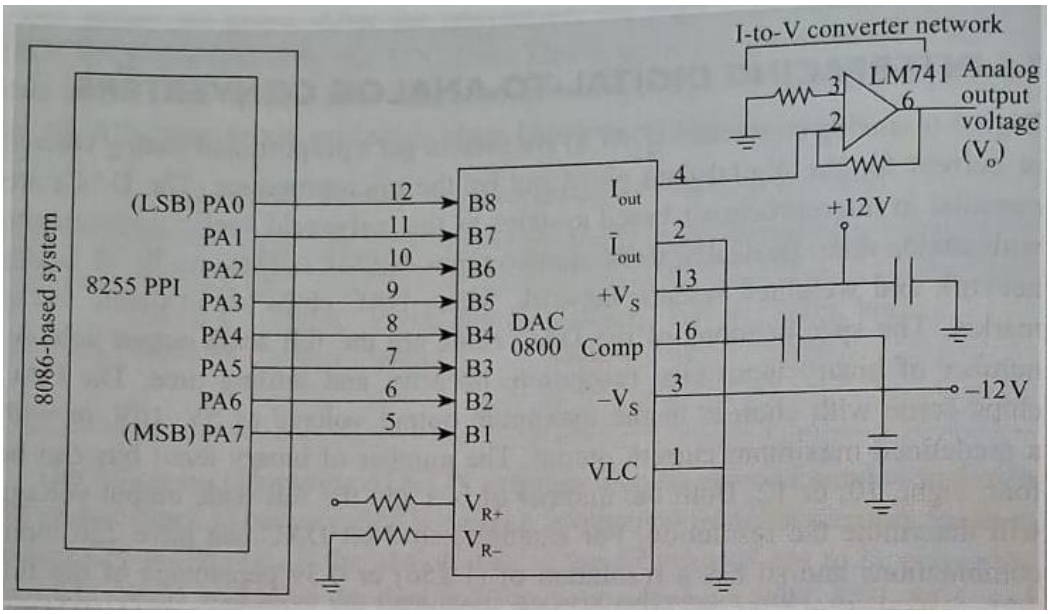


Fig: Interfacing DAC 0800 with the 8086

Need for 8259 Programmable Interrupt Controller:-

The processor has limited number of hardware interrupts. For applications that use interrupts from multiple sources, the processor can use external device called programmable Interrupt Controller (PIC). The Intel 8259 is a PIC designed and developed for use with the Intel 8086 microprocessor.

Operation of 8255:

The following steps show how interrupt handling is done when an external device places an interrupt request on the IR lines of the 8259. It is assumed that the system has a single 8259 chip.

(i) One or more of the IR lines in the 8259 may go high.

(ii) Corresponding IRR bit bits is/are set in the 8259.

(n) The 8259 evaluates the interrupt request based on masking and priority.

(iv) If no higher priority interrupt other than the currently received interrupt interrupts is currently processed then the 8259 sends out interrupt request (INT) signal to the 8086.

signal from the 8259. (v) The 8086 microprocessor sends out two INTA pulses in response to the INT

(vi) The bit corresponding to the highest priority interrupt among the currently received interrupts in the ISR is set and its corresponding bit in IRR is reset in the 8259.

(vii) After receiving the first INTA pulse from the 8086, the 8259 does the above internal operation and after receiving the second INTA pulse from the 8086, the 8259 sends the desired interrupt type or interrupt vector number of the interrupt currently to be processed by the 8086 through the data bus D7-DO of the 8259.

(viii) The interrupt service routine (ISR) is executed in the 8086 processor with the following steps:

- The flags are pushed onto the stack.
- The interrupt and trap flags of the processor are cleared.
- The return address is pushed onto the stack.
- The starting address of the interrupt service routine obtained from the interrupt vector table corresponding to the currently received interrupt type or vector number in the program counter and instruction pointer is loaded.
- The ISR is executed.

(ix) At the end of the ISR, instructions are written for sending a command word to reset the bit corresponding to the currently processed interrupt in the ISR of the 8259 so that lower priority interrupts can be processed.

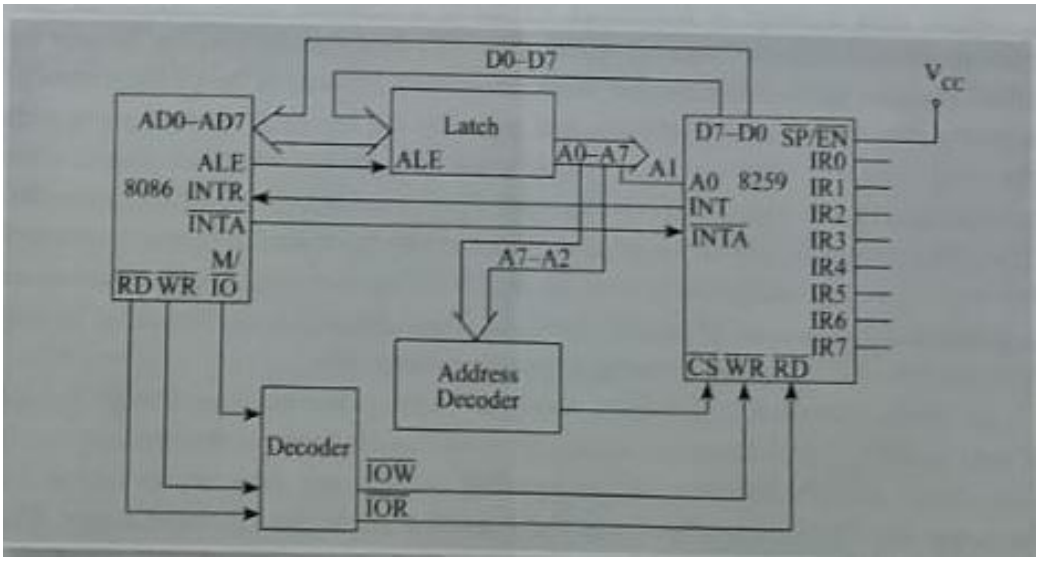


Fig: Interfacing of 8259 to 8086

The PIC 8259 requires two addresses with A0 being 0 and 1. The Al line from the address bus is connected to the A0 line in the 8259. The higher-order address bus is used to select the particular chip through a proper address decoder. Read and write control signals of the 8086 are connected to the corresponding signals of the 8259. The data lines of IC 8259 are connected to the lower-order address and data bus of the 8086.The multipurpose SP/EN pin is tied to logic high because only one 8259 is used in the system. The interrupt request line, INT of the 8259 is connected to the 8086's interrupt line INTR. The INTA of the 8086 is connected to the INTA of the 8259. The eight IR inputs of the 8259 can be connected to the interrupt sources from different external devices.

Interfacing Intel 8237 DMA Controller:-

DMA stands for Direct Memory Access. It is one of the ways to accomplish high speed data transfers, directly between memory and peripheral devices. The DMA is a method of data transfer between memory and I/O devices without the intervention of microprocessor. DMA data transfer is controlled by suing a separate DMA controller. The microprocessor must be disabled during the DMA data transfer process.

Operation of 8237 with 8086:

The following figure shows how a DMA transfer takes place between a memory and an I/O device with the help of a DMA controller. Here, the microprocessor and the DMA controller time share the use of address, data and control buses. The 8237 address, control signals and data bus pins are connected in parallel with the system buses. An external latch is required for the upper address byte. While inactive, the controller's outputs are in a high impedance state. When activated by a DMA request and bus control is surrendered by the host, the 8237 drives the buses and generates the control signals to perform the data transfer.

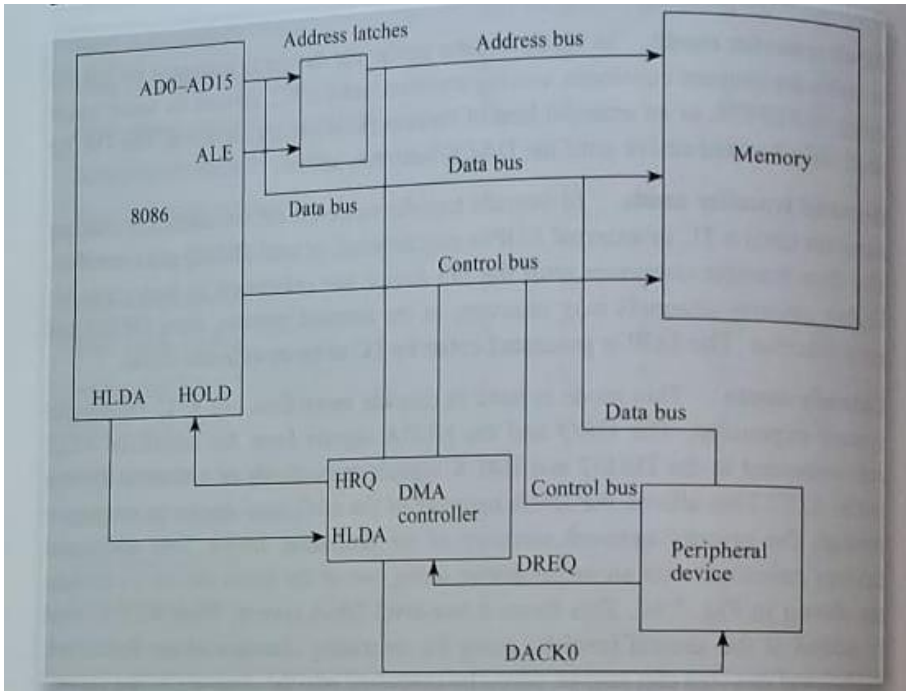


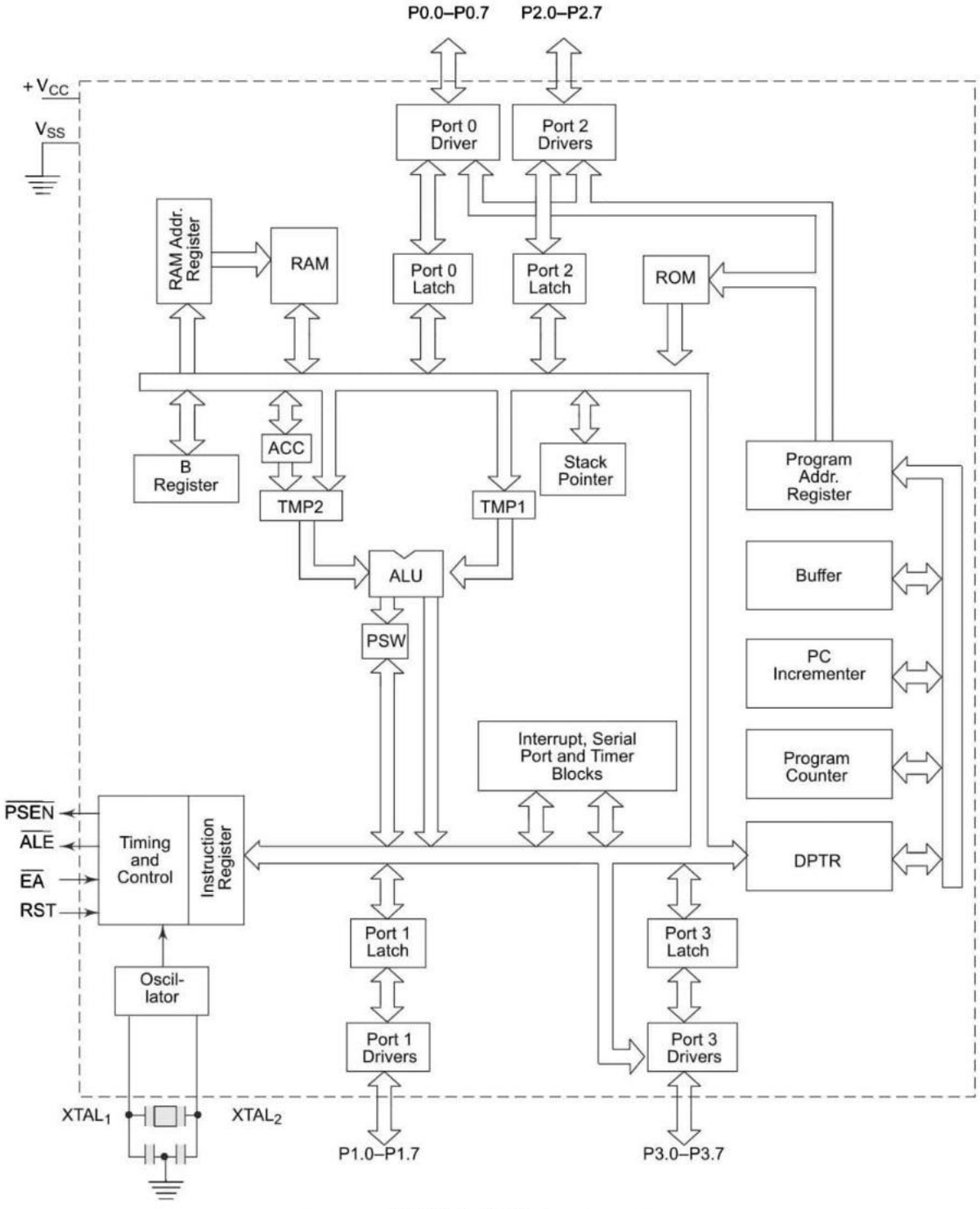
Fig: Interfacing DMA controller with 8086

When the system is first turned on, the buses are connected to the microprocessor to system memory and peripherals. Then all the programmable devices in the system are initialized and then the normal routine program is executed until it is needed for data transfer.

The operation performed by activating one of the four DMA request inputs has to be programmed first into the controller via the command, mode, address, and word count registers.

UNIT-4

Architecture of 8051:-



8051 Block Diagram

The internal architecture of 8051 is presented in Fig. 17.2. The functional description of each block is presented briefly below.

- **Accumulator (ACC)**: The accumulator register (ACC or A) acts as an operand register, in case of some instructions. This may either be implicit or specified in the instruction. The ACC register has been al-lotted an address in the on-chip special function register bank.
- **B Register:** This register is used to store one of the operands for multiply and divide instructions. In other instructions, it may just be used as a scratch pad. This register is considered as a special function register.
- **Program Status Word (PSW):** This set of flags contains the status information and is considered as one of the special function registers.
- **Stack Pointer (SP)**: This 8-bit wide register is incremented before the data is stored onto the stack using push or call instructions. The 8051 stack is not a top-down data structure, like other Intel processors. This register has also been allotted an address in the special function register bank.
- **Data Pointer (DTPR)**: This 16-bit register contains a higher byte (DPH) and the lower byte (DPL) of a 16-bit external data RAM address. It is accessed as a 16-bit register or two 8-bit registers as specified above. It has been allotted two addresses in the special function register bank, for its two bytes DPH and DPL.
- **Port 0 to 3 Latches and Drivers:** These four latches and driver pairs are allotted to each of the four on-chip I/O ports. These latches have been allotted addresses in the special function register bank. Using the allotted addresses, the user can communicate with these ports. These are identified as P0, P1, P2 and P3.
- **Serial Data Buffer:** The serial data buffer internally contains two independent registers. One of them is a transmit buffer which is necessarily a parallel-in serial-out register. The other is called receive buffer which is a serial-in parallel-out register. Loading a byte to the transmit buffer initiates serial transmission of that byte. The serial data buffer is identified as SBUF and is one of the special function registers. If a byte is written to SBUF, it initiates serial transmission and if the SBUF is read, it reads received serial data.
- **Timer Registers:** These two 16-bit registers can be accessed as their lower and upper bytes. For ex-ample, TLO represents the lower byte of the timing register 0, while THO represents higher bytes of the timing register 0. Similarly, TL1 and TH1 represent lower and higher bytes of timing register 1. All these registers can be accessed using the four addresses allotted to them which lie in the special function registers SFR address range, i.e. 80 H to FF.
- **Control Registers**: The special function registers IP, IE, TMOD, TCON, SCON and PCON contain control and status information for interrupts, timers/counters and serial port. All of these registers have been allotted addresses in the special function register bank of 8051.
- **Timing and Control Unit:** This unit derives all the necessary timing and control signals required for the internal operation of the circuit. It also derives control signals required for controlling the external system bus.
- **Oscillator**: This circuit generates the basic timing clock signal for the operation of the circuit using crystal oscillator.

- **Instruction Register**: This register decodes the opcode of an instruction to be executed and gives information to the timing and control unit to generate necessary signals for the execution of the instruction.
- **EPROM and Program Address Register:** These blocks provide an on-chip EPROM and a mechanism to internally address it. Note that EPROM is not available in all 8051 versions.
- **RAM and RAM Address Register:** These blocks provide internal 128 bytes of RAM and a mechanism to address it internally.
- **ALU**: The arithmetic and logic unit performs 8-bit arithmetic and logical operations over the operands held by the temporary registers TMP1 and TMP2. Users cannot access these temporary registers.
- **SFR Register Bank** This is a set of special function registers, which can be addressed using their respective addresses which lie in the range 80H to FFH.

Special Function Registers (SFR):-

8051 has two 8-bit registers, registers A and B, which can be used to store operands, as allowed by the instruction set. Internal temporary registers of 8051 are not user accessible. Including these A and B registers, 8051 has a family of special purpose registers known as, Special Function Registers (SFRs). There are, in total, 21-bit addressable, 8-bit registers. ACC (A), B, PSW, P0, P1, P2, P3, IP, IE, TCON and SCON are all 8-bit, bit-addressable registers. The remaining registers, namely, SP, DPH, DPL, TMOD, THO, TLO, TH1, TL1, SBUF and PCON registers are to be addressed as bytes, i.e. they are not bit-addressable. The registers DPH and DPL are the higher and lower bytes of a 16-bit register DPTR, i.e. data pointer, which is used for accessing external data memory. Starting 32-bytes of on-chip RAM may be used as general purpose registers. They have been allotted addresses in the range from 0000H to 001FH. These 32, 8-bit registers are divided into four groups of 8 registers each, called register banks. At a time only one of these four groups, i.e. banks can be accessed. The register bank to be accessed can be selected using the RS1 and RSO bits of an internal register called program status word.

The registers THO and TLO form a 16-bit counter/timer register with H indicating the upper byte and L indicating the lower byte of the 16-bit timer register TO. Similarly, TH1 and TL1 form the 16-bit count for the timer T1.

The four port latches are represented by P0, P1, P2 and P3. Any communication with these ports is established using the SFR addresses to these registers.

Register SP is a stack pointer register. Register PSW is a flag register and contains status information. Register IP can be programmed to control the interrupt priority.

Register IE can be programmed to control interrupts, i.e. enable or disable the interrupts.

TCON is called timer/counter control register. Some of the bits of this register are used to turn the timers on or off. This register also contains interrupt control flags for external interrupts INT, and INT.

The register TMOD is used for programming the modes of operation of the timers/counters. The SCON register is a serial port mode control register and is used to control the operation of the serial port.

The SBUF register acts as a serial data buffer for transmit and receive operations.

The PCON register is called power control register. This register contains power down bit and idle bit which activate the power down mode and idle mode. In the idle mode, the oscillator continues to run and the interrupt, serial port and timer blocks are active but the clock to the CPU is disabled. The CPU status is preserved. This mode can be terminated with a hardware interrupt or hardware reset signal. After this, the CPU resumes program execution from where it left off.

In power down mode, the on-chip oscillator is stopped. All the functions of the controller are held maintaining the contents of RAM. The only way to terminate this mode is hardware reset.

Register	*Bit Addressable*	*Address (SFR)*
ACC	Y	0E0H
B	Y	0F0H
PSW	Y	0D0H
SP	N	81H
DPH	N	82H
DPL	N	83H
P0	Y	80H
P1	Y	90H
P2	Y	0A0H
P3	Y	0B0H
IP	Y	0B8H
IE	Y	0A8H
TMOD	N	89H
TCON	Y	88H
TH0	N	8CH
TL0	N	8AH
TH1	N	8DH
TL1	N	8BH
SCON	Y	98H
SBUF	N	99H

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
CY	AC	F0	RS_1	RS_0	OV	—	P

CY	D_7	Carry Flag.
AC	D_6	Auxiliary carry Flag.
F0	D_5	Flag 0 is available to the user for general purpose.
RS_1	D_4	Register Bank selector bit 1.
RS_0	D_3	Register Bank selector bit 0.

The value presented by RS_0 and RS_1 bits select the corresponding register bank as shown below.

RS_1	RS_0	Register Bank	Address
0	0	0	00H-07H
0	1	1	08H-0FH
1	0	2	10H-17H
1	1	3	18H-1FH

OV	D_2	Overflow Flag.
—	D_1	User definable flags (Reserved for future use)
P	D_0	Parity flag is set/cleared by hardware in each instruction cycle to indicate an odd/even number of '1' bits in the accumulator

Addressing Modes of 8051:-

8051 instruction set supports six addressing modes listed below.

1. Immediate Addressing Mode
2. Direct Addressing Mode
3. Register Addressing Mode
4. Register indirect Addressing Mode
5. Register Specific Addressing Mode
6. Indexed Addressing Mode

The addressing modes supported by 8051 are discussed further in brief.

Direct Addressing: In this mode of addressing, the operands are specified using the 8-bit address field, in the instruction format.

Ex: MOV RO. 89H.

Here 89H is address of a special function register TMOD.

Indirect Addressing: In this mode of addressing, the 8-bit address of an operand is stored in a register and the register, instead of the 8-bit address, is specified in the instruction. The registers Ro and R_1, of the selected bank of registers or stack pointer can be used as address registers for storing the 8-bit addresses.

Ex: ADD A, @RO

Register Addressing Mode: In this addressing mode, operands are stored in the registers Ro-R_1, of the selected register bank. One of these eight registers (R0-R7) is specified in the instruction. A register bank can be selected using the two bank select bits of the PSW.

Ex: ADD A. R7.

Register Specific Addressing Mode: In this type of instructions, the operand is implicitly specified using one of the registers. Some of the instructions always operate only on a specific register. These types of instructions fall under this category.

Ex: RLA

Immediate Addressing Mode: In this mode, an immediate data, i.e. a constant is specified in the instruction.

Ex: ADD A, #100

Indexed Addressing Mode: Only program memory can be accessed using this addressing mode. Basically, this mode of addressing is accomplished in 8051 for look-up table manipulations. Program counter or data pointer are the allowed 16-bit address storage registers, in this mode of addressing.

Ex: MOVC A, @A+DPTR

JMP @A+DPTR

8051 Instruction Set:-

The 8051 instruction set consists of 5 categories:

1. Data transfer instructions
2. Arithmetic instructions
3. Logical instructions
4. Boolean (bit manipulation) instructions
5. Control Transfer (or) Branch instructions

1. Data transfer instructions

These instructions move data between internal memory, external memory, and registers.

Mnemonic	Description	Example
MOV	Moves a byte of data between a register and a memory location.	MOV A, R0 (Moves the content of register R0 to the Accumulator A).

MOVC	Moves a byte from code (program) memory into the accumulator.	MOVC A, @A+DPTR (Loads the accumulator with a byte from code memory, addressed by the sum of the accumulator and the Data Pointer).
MOVX	Moves a byte of data between the accumulator and external data memory.	MOVX @DPTR, A (Moves the content of the accumulator to the external RAM address pointed to by the Data Pointer).
PUSH	Increments the stack pointer and copies the source byte onto the stack.	PUSH 40H (Pushes the content of internal RAM location 40H onto the stack).
POP	Copies the byte from the stack to the destination and decrements the stack pointer.	POP A (Pops a byte from the stack into the accumulator).
XCH	Exchanges the contents of the accumulator with a specified byte.	XCH A, R1 (Exchanges the contents of the accumulator with register R1).

2. Arithmetic instructions

These instructions perform arithmetic operations such as addition, subtraction, multiplication, and division.

Mnemonic	Description	Example
ADD	Adds the source byte to the accumulator.	ADD A, #25H (Adds the immediate value 25H to the accumulator).
ADDC	Adds the source byte and the carry flag to the accumulator.	ADDC A, R2 (Adds the content of R2 and the carry flag to the accumulator).
SUBB	Subtracts the source byte and the borrow (carry) flag from the accumulator.	SUBB A, 50H (Subtracts the content of internal RAM address 50H and the carry flag from the accumulator).
INC	Increments the operand by one.	INC R0 (Increments the content of register R0 by 1).

DEC	Decrements the operand by one.	DEC A (Decrements the content of the accumulator by 1).
MUL	Multiplies the accumulator by the B register.	MUL AB (Multiplies the 8-bit unsigned integers in registers A and B. The 16-bit result is stored in B (high byte) and A (low byte)).
DIV	Divides the accumulator by the B register.	DIV AB (Divides the 8-bit unsigned integer in A by the 8-bit unsigned integer in B. The integer result is stored in A, and the remainder is stored in B).

3. Logical instructions

These instructions perform bitwise logical operations.

Mnemonic	Description	Example
ANL	Performs a bitwise AND operation between the source and destination bytes.	ANL A, #0FH (Masks the upper 4 bits of the accumulator by performing a bitwise AND with 0FH).
ORL	Performs a bitwise OR operation between the source and destination bytes.	ORL P1, #01H (Sets the least significant bit of Port 1).
XRL	Performs a bitwise XOR operation between the source and destination bytes.	XRL A, 20H (XORs the accumulator with the content of internal RAM address 20H).
CLR A	Clears the accumulator to zero.	CLR A
CPL A	Complements (inverts) each bit of the accumulator.	CPL A
SWAP A	Swaps the upper and lower nibbles (4 bits) of the accumulator.	SWAP A (If A=A5H, after execution A=5AH).

4. Boolean (bit manipulation) instructions

These instructions perform single-bit operations, affecting individual bits in the internal RAM and Special Function Registers (SFRs).

Mnemonic	Description	Example
SETB	Sets the specified bit (sets to 1).	SETB P1.0 (Sets the pin 0 of Port 1).
CLR	Clears the specified bit (sets to 0).	CLR C (Clears the Carry flag).
CPL	Complements (inverts) the specified bit.	CPL 50H (Complements the bit at address 50H in the bit-addressable area of internal RAM).
MOV	Moves a bit value to or from the carry flag.	MOV C, P2.3 (Copies the bit state of pin P2.3 to the Carry flag).

5. Control Transfer (or) Branch instructions

These instructions alter the flow of program execution by jumping to a different memory address.

Mnemonic	Description	Example
LJMP	Performs a long jump to a 16-bit absolute address.	LJMP 2050H (Jumps unconditionally to program memory address 2050H).
SJMP	Performs a short jump using an 8-bit relative address.	SJMP LOOP (Jumps to the instruction at the label LOOP).
JNC	Jumps if the carry flag is not set.	JNC NEXT_INST (If the carry flag is clear, jump to NEXT_INST).
JZ	Jumps if the accumulator is zero.	JZ ZERO_FOUND (If the accumulator holds a value of 0, jump to ZERO_FOUND).
DJNZ	Decrements the specified byte and jumps if the result is not zero.	DJNZ R7, LABEL (Decrements R7, then if R7 is not zero, jumps to LABEL).
ACALL	Performs an absolute call to a subroutine within the current 2K-byte page.	ACALL SUBROUTINE (Calls the subroutine at the address of SUBROUTINE).

LCALL	Performs a long call to a subroutine at a 16-bit address.	LCALL MY_FUNC (Calls the subroutine at the address of MY_FUNC).
RET	Returns from a subroutine.	RET (Pops the return address from the stack and resumes execution there).

I/O Pins Ports and Circuits:-

The 8051 has four I/O ports-port 0, port 1, port 2, and port 3. The major constraint in microcontroller chips is the number of pins. To reduce the number of pins, the pins allotted for parallel ports are assigned some alternative functions as well. Out of the four parallel ports available in the 8051, port 1 is used exclusively for input and output functions. The other port pins have functions other than input and output. So, the 24 pins of ports 0, 2, and 3 perform functions other than parallel data transfer. These functions are decided by the hardware interfaced and the instructions being executed.

All the four ports are bidirectional, that is, they can be programmed to perform input or output operation. The eight port pins are connected through eight D-type port latches. A D-type latch connects the data in it to a port pin, when the port is used as an output port.

Port pins	Bit address	Port pins	Bit address
Port 0.0	80	Port 2.0	A0
Port 0.1	81	Port 2.1	A1
Port 0.2	82	Port 2.2	A2
Port 0.3	83	Port 2.3	A3
Port 0.4	84	Port 2.4	A4
Port 0.5	85	Port 2.5	A5
Port 0.6	86	Port 2.6	A6
Port 0.7	87	Port 2.7	A7
Port 1.0	90	Port 3.0	B0
Port 1.1	91	Port 3.1	B1
Port 1.2	92	Port 3.2	B2
Port 1.3	93	Port 3.3	B3
Port 1.4	94	Port 3.4	B4
Port 1.5	95	Port 3.5	B5
Port 1.6	96	Port 3.6	B6
Port 1.7	97	Port 3.7	B7

Fig: Ports in 8051 and their Pins

Port 1: Port 1 is the only port in the 8051 that is used exclusively for input and output operations. The structure of port 1 is shown in figure. The output of the port latch is connected to the port pin

through a transistor driver with an internal pull up resistor. The port can be operated as an input port after writing the data '1' in all the bits of the port 1 latch.

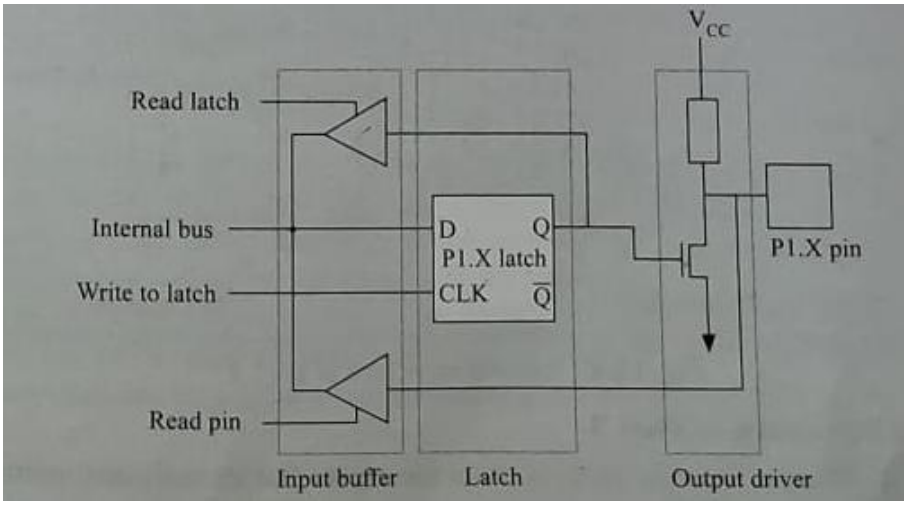


Fig: Internal Circuit of Port 1

Ports 0 and 2:

Pins of ports 0 and 2 can be used as input port pins if a '1' is written in the corresponding port latches by the programmer. For using the pins of ports 0 and 2 as output pins, a pull-up resistor may have to be connected to the corresponding port pins. The internal structure of port 0 is shown in figure.

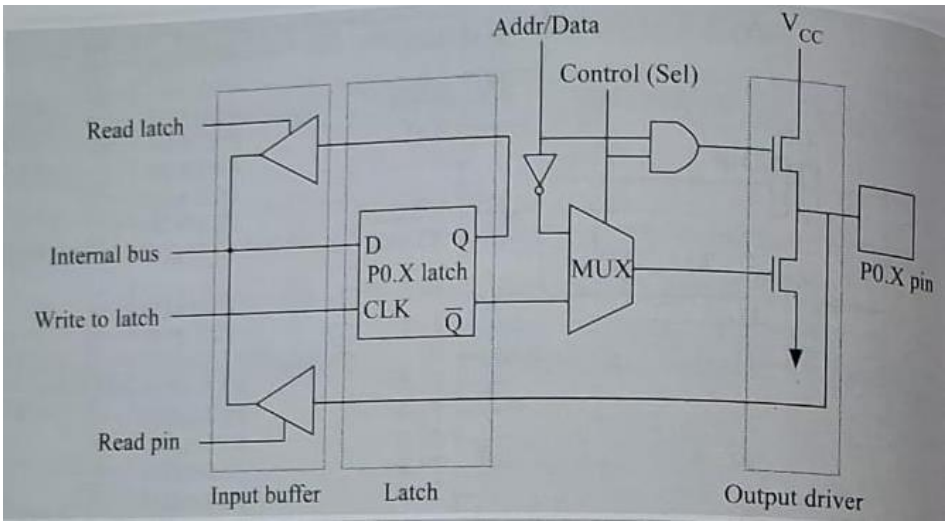


Fig: Internal Circuit of Port 0

Besides input and output, ports 0 and 2 have an alternative function they can be used as address/data bus when external memory or I/O devices are accessed. Port 0 acts as the lower-order address bus and port 2 acts as the higher-order address bus. The drivers of ports 0 and 2 have an internal multiplexer for this purpose. The internal structure of port 2 is shown in figure.

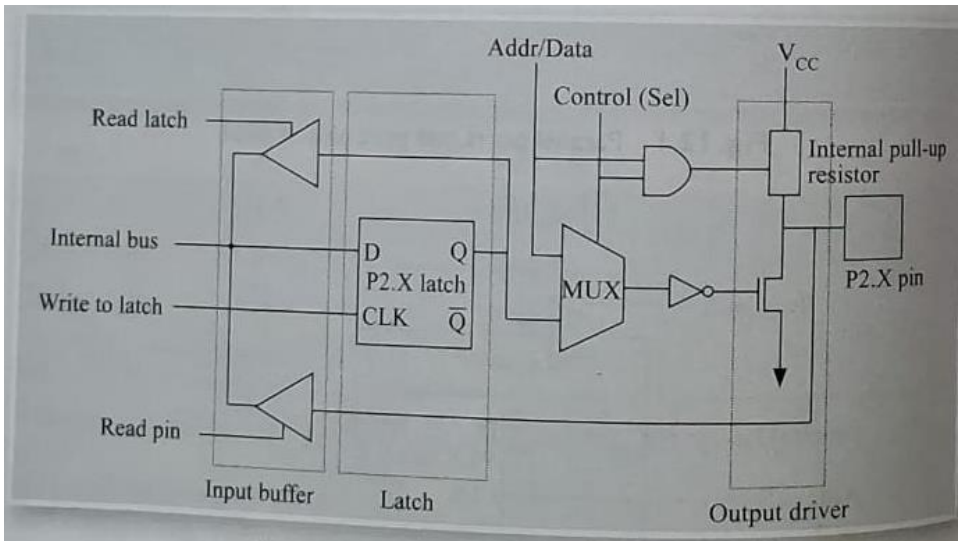


Fig: Internal Circuit of Port 2

Port 3:

Port 3 is different from the other ports in the aspect that its individual port pins programmed separately for input operation, output operation or other alternative can be programmed for input and output operations. Each pin of port 3 is given in the table. All the port 3 pins serve an alternative function, based on the hardware functions.

Port pin	Alternative function
P3.0	RXD—Serial input data line
P3.1	TXD—Transmit data line
P3.2	INT0—External interrupt 0
P3.3	INT1—External interrupt 1
P3.4	T0—Timer 0 external input
P3.5	T1—Timer 1 external input
P3.6	WR—External data memory write strobe
P3.7	RD—External data memory read enable

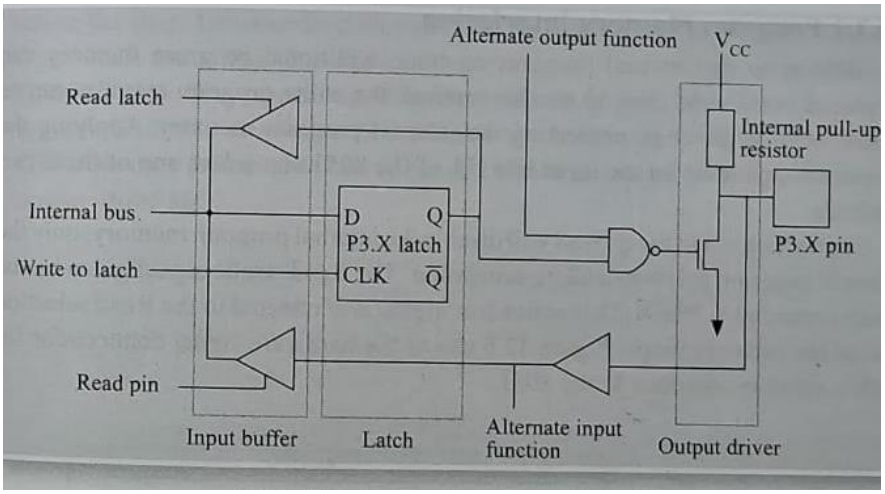
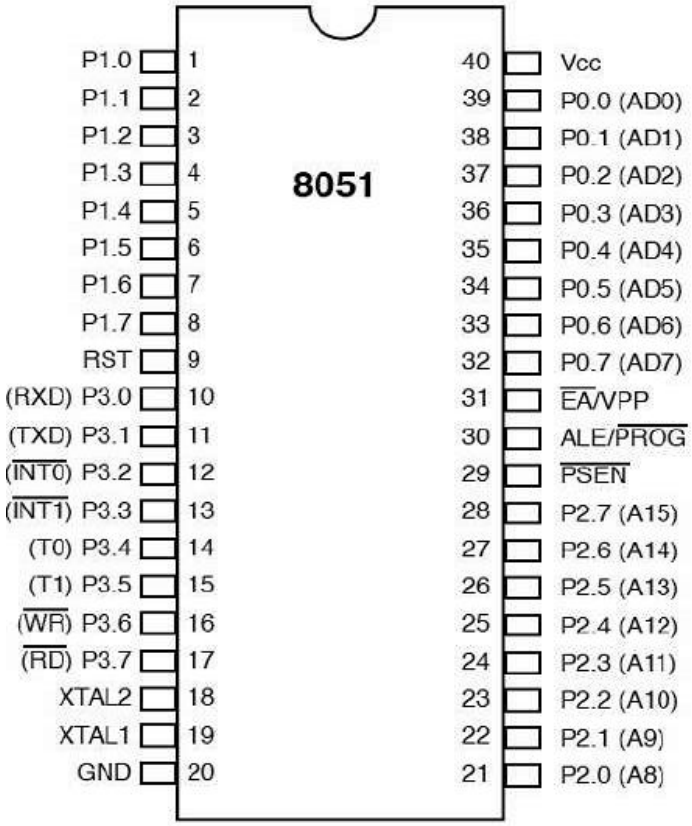


Fig: Internal Circuit of Port 3

Pin Diagram of 8051:-



Unit-1: 8051 Interfacing

Interfacing LCD with 8051:-

Modern LCD devices are becoming popular and are finding an increasing number of applications. The main advantage of LCDs over seven segment displays is their ability to display numbers, characters, and graphics; the latter can display only numbers and a limited set of characters. LCDs come with an internal controller and refreshing circuit. So, the CPU is relieved of the task of refreshing the display The programming of LCD devices is easier, since they have predefined control words and addresses. The cost of LCDs is also declining.

Table 13.4 LCD display pin configuration

Pin number	Symbol	Level	I/O	Function
1	V_{SS}	–	–	Power supply (GND)
2	V_{CC}	–	–	Power supply (+5 V)
3	V_{EE}	–	–	Contrast adjust (Vo)
4	RS	0/1	I	Command or data register select line 0 = Instruction input 1 = Data input
5	R/W	0/1	I	0 = Write to LCD module 1 = Read from LCD module
6	E	1 to 0	I	Enable signal
7	DB0	0/1	I/O	Data bus line 0 (LSB)
8	DB1	0/1	I/O	Data bus line 1
9	DB2	0/1	I/O	Data bus line 2
10	DB3	0/1	I/O	Data bus line 3
11	DB4	0/1	I/O	Data bus line 4
12	DB5	0/1	I/O	Data bus line 5
13	DB6	0/1	I/O	Data bus line 6
14	DB7	0/1	I/O	Data bus line 7 (MSB)

Eight data lines are used for interfacing the LCD with the processor. The three control signals are RS, R/W and E.

(i) RS is used to select a control register or a data register.

(ii) R / overline W indicates the direction of data flow between the processor and the display.

(iii) The E signal is used to enable data transfer.

The three control signals and eight data lines are interfaced with the two ports of the 8051. In the interface diagram shown in figure, the port 1 lines are used to transfer data and the port 2 lines are used to issue the control signals from the microcontroller to the LCD.

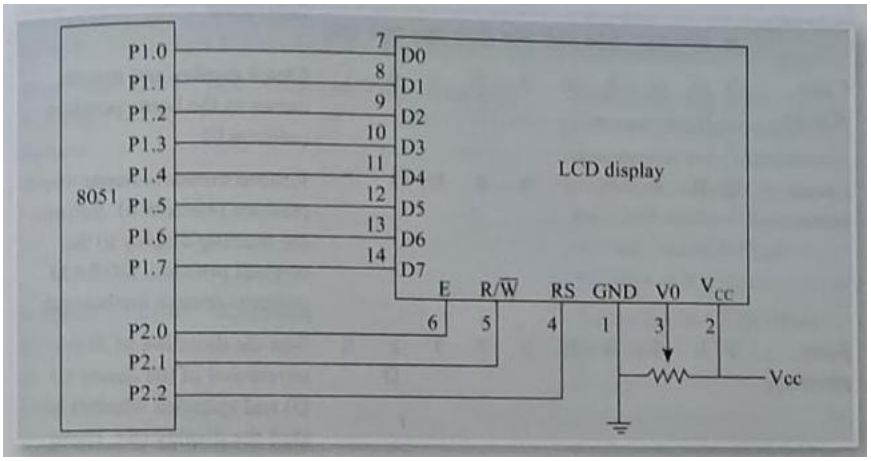


Fig: LCD interfaced with 8051

The programming of the LCD is done with the appropriate initialization word and command words. Every time a control or command word is written into the LCD, the RS signal input must be made 0; R/W must be made 0 for the write operation. After setting these two signals, the enable signal E must be applied as a pulse. Similarly, when data is written into the LCD device, the RS signal must be made 1and R/W must be made 0.

Hex code	Instruction
01	Clear display and return cursor to home position
02	Return cursor to home position
04	Shift cursor left (Decrement)
06	Shift cursor right (Increment)
05	Shift display right
07	Shift display left
08	Display off and cursor off
0A	Display off and cursor on
0C	Display on and cursor off
0E	Display on and cursor character not blinking
0F	Display on and cursor character blinking

Fig: LCD Command Codes

Interfacing Keyboard with 8051:-

The matrix keypad is organized as a matrix connection of switches. Mechanical switches have a problem called contact bounce because of their construction. The pressing of a mechanical switch must produce a single pulse output. Practically, instead of producing a single clean pulse output, the switches generate a series of pulses because the switch contacts do not come to rest immediately.

As the microprocessor is faster than a manual key press, the single key press will be registered as multiple key presses. The signal from the key falls and rises a few times within a period of about 5ms. This is the main consequence of key bouncing as the contact bounces. So, the signal from the key must be made free from key bouncing transients. This technique is called key de-bouncing.

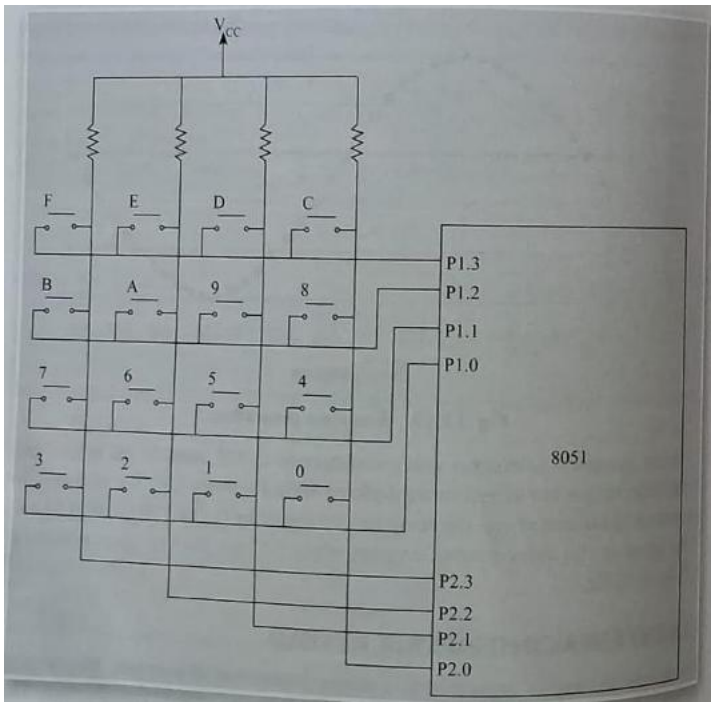


Fig: Interfacing Keyboard with 8051

In the software, a column scan is done by giving an output of 0 on the lines P1.0-P1.3 of port 1. A key press is detected by checking for the row (P2.0-P2.3 of port 2) in which the zero level appears. The scanning of a row is achieved by applying OV to one port 1 pin and 5 V to the other three port 1 pins, then scanning each individual port 2 pin to see if one of them is low. If it is, the key at the junction between the row and column being scanned is the pressed key.

The procedure for finding a key press involves the following steps:

(i) Clear P1.0, set the other three bits
   a. Scan P2.0
   b. Scan P2.1
   c. Scan P2.2
   d. Scan P2.3

(ii) Clear P1.1, set the other three bits
   a. Scan P2.0
   b. Scan P2.1
   c. Scan P2.2
   d. Scan P2.3

(iii) Clear P1.2, set the other three bits
   a. Scan P2.0
   b. Scan P2.1
   c. Scan P2.2
   d. Scan P2.3

(iv) Clear P1.3, set the other three bits
   a. Scan P2.0
   b. Scan P2.1
   c. Scan P2.2
   d. Scan P2.3

Interfacing ADC with 8051:-

The basic function of the analog-to-digital converter (ADC) is to convert the input analog voltage levels into corresponding discrete digital signals. An ADC is essential in a microprocessor or microcontroller system.

The specifications of the ADC are the range of analog input voltage, number of digital bits at the output, resolution, the conversion time, and the number of analog input channels. The analog input voltage can be either unipolar or bipolar. Unipolar means that the input voltage can have only one polarity such as 0 to +5V or 0 to +10 V. Bipolar means that the input voltage can range from one polarity to the other such as -5 to +5 V or 10 to +10 V. Most of the ADC chips come with an option of selecting one of these voltage ranges using the Vref input pins. The ADC chips are available for different number of output binary bits. ADCs are available with 8, 10,12 or 16-bit digital outputs. The number of bits will decide the number of voltage levels sensed. For example, an 8-bit ADC will have 28 possible levels, that is, 256 levels. The number of bits and the input voltage range will decide the resolution. The resolution of an ADC is defined as the smallest change in the input voltage that can be detected at the output.

ADC 0808/0809 is a commonly used ADC chip with eight analog input channels and an 8-bit digital output. The eight analog inputs are multiplexed and selected using the three select lines, A, B, and C. The select lines are connected to the least significant three bits of port 0. The additional inputs +Vref, -Vref and CLK and the supply inputs must be given to the ADC chip. The ALE and SC pins are tied together and connected to the port pin P0.3. The OE pin is connected to another port 0 pin, P0.4, and the end of conversion signal (EOC) is sensed through P0.5.

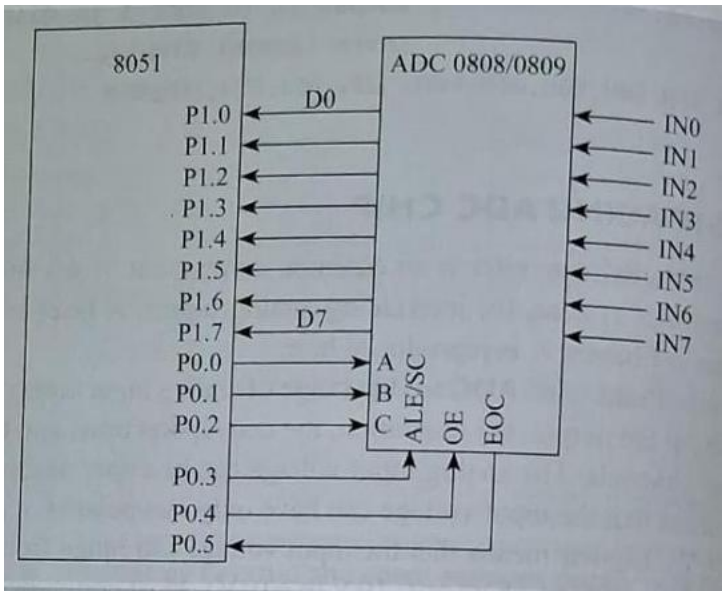


Fig: Interfacing ADC with 8051

Interfacing DAC with 8051:-

Digital-to-analog converters are used to get a proportional analog voltage or current for the digital data sent by the microprocessor.

Let us consider the common digital-to-analog converter chip DAC 0800 in this interface example. This section will describe the interfacing of the DAC 0800 with the 8051 microcontroller. Any one port is enough to interface an 8-bit DAC with the 8051. The interface diagram is shown in figure. The data lines of the DAC chip are connected to the port 1 lines of the microcontroller. The port 1 lines should now be made output lines, to send digital output to the DAC. The output from the DAC chip is a variable current and this is converted into a variable voltage using a current-to-voltage converter.

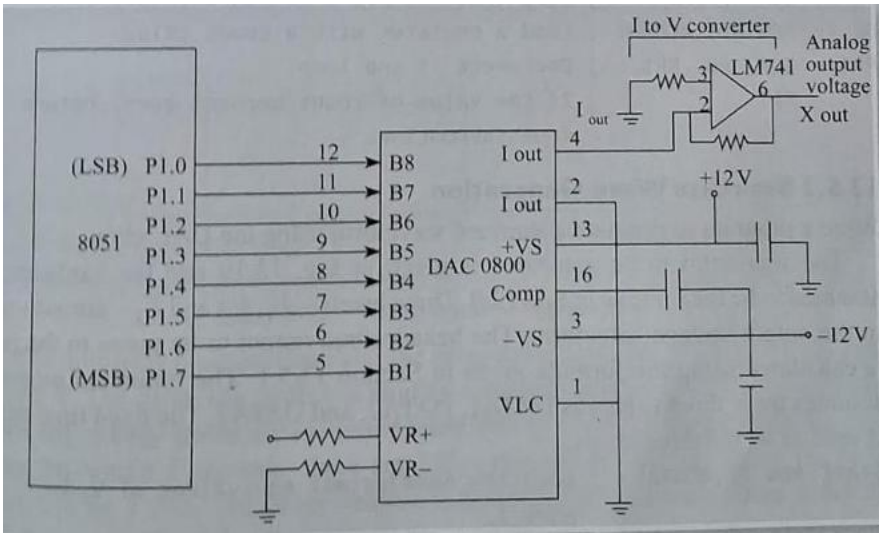


Fig: Interfacing DAC with 8051

Interfacing Stepper Motor:-

A stepper motor is a special motor that rotates in incremental steps, unlike other motors that run continuously. They find application in printers, plotters, robots, etc. Stepper motors are excited by pulses to get incremental displacements. Stepper motors are made of permanent magnet rotors with stator field excitation. Two-phase excitation and four-phase excitation are common. A four-phase stepper motor has four stator poles, which are excited by pulses. Each pole winding can be excited such that the pole can be made either a north pole or a south pole. In this arrangement, the poles should be properly excited in a particular sequence so that the rotor rotates in a particular direction. If the excitation sequence is reversed, the rotor rotates in the reverse direction.

The interfacing of a four-phase stepper motor with the 8051 is given in figure. The figure shows the four terminals A, B, C, and D of the motor connected to the port 0 pins through the transistor drivers. The common terminal of the motor is connected to the supply. The excitation sequence for the stepper motor is given in Tables. The single-phase excitation results in low current through the motor windings and is also called wave drive mode. In two-phase excitation, the excitation current through the motor windings is high and so it is called high-torque excitation mode.

Table Switching sequence: One-phase excitation (Wave mode)

PA3	PA2	PA1	PA0	Clockwise sequence	Anti-clockwise sequence	Hex value
0	0	0	1	1	4	01
0	0	1	0	2	3	02
0	1	0	0	3	2	04
1	0	0	0	4	1	08

Table Switching sequence: Two-phase excitation (high-torque excitation)

P0.3	P0.2	P0.1	P0.0	Clockwise	Anti-clockwise	Hex value
0	0	1	1	1	4	03
0	1	1	0	2	3	06
1	1	0	0	3	2	0C
1	0	0	1	4	1	09

External Memory Interfacing in 8051:-

External memory interfaced with the 8051 can be of two types external program memory and external data memory. External memory accesses are accomplished with ports 0 and 2 of the 8051, as they serve as the multiplexed address/data buses The external memory in the 8051 is always accessed with 16-bit addresses. The 8051 outputs the ALE (address latch enable) signal to de-multiplex the lower-order address and data bus. In addition, the microcontroller sends the control signals on the port 3 lines.

Program Memory Interfacing:

In addition to the internal program memory, additional program memory can be placed outside the chip. In another method, the entire program memory can be placed outside the chip, neglecting the internal program memory. Applying the proper voltage level on the input line EA of the 8051 can select one of these two methods.

Connecting EA to the ground will disable the internal program memory and only the external program memory will be accessible. The Read strobe signal given by the microcontroller is PSEN. This active low signal is connected to the Read selection line of the memory chips. The following figure shows the hardware signal connections for such a memory interface to the 8051.

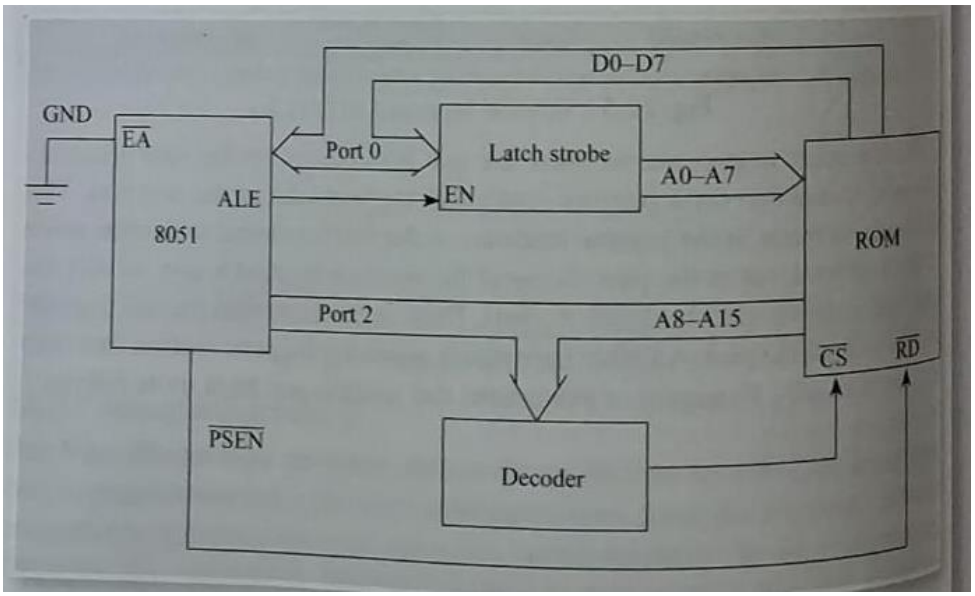


Fig: Interfacing external program memory to 8051

Connecting the EA pin of the 8051 to logic 1 or +5V will program the microcontroller to use the internal program memory for the addresses starting at 0000H. After the available internal memory is used completely, the external memory is accessed.

- If $\overline{EA}$ = 0 the internal program memory is not accessed.

- If $\overline{EA}$ = 1 the internal program memory is accessed for the address range 0000H-OFFFH (or the available range) and the external program memory is accessed for addresses greater than OFFFH.

Data Memory Interfacing:

The data memory in the system should be random access memory, as it should facilitate both read and write operations on data. The external data memory is interfaced in the same way as the program memory, as shown in figure. The major difference between data memory and program memory is that read and write operations can be done in random access data memory, whereas only read operation is possible in program memory. The control signals for reading from and writing into data memory are obtained from the port 3 pins. P3.6 pin sends the data memory write enable signal and P3.7 pin sends the RAM read enable signal. A decoder logic circuit is necessary to select the RAM chip, based on the higher-order address lines.

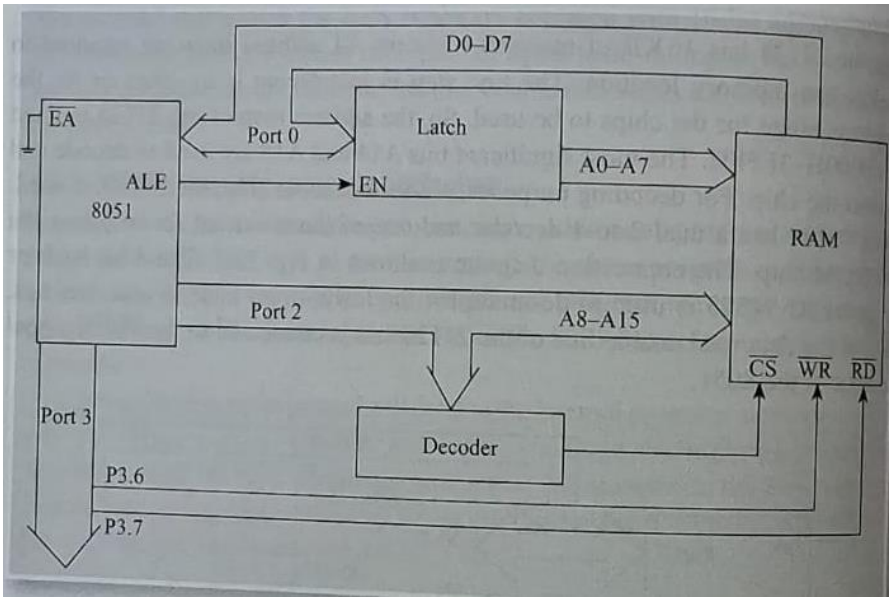


Fig: Interfacing external data memory to 8051

Serial Port Programming:-

One of the 8051's many powerful features is its integrated Universal Asynchronous Receiver Transmitter (UART), otherwise known as serial port. With the integrated serial port of the 8051, data can be transmitted and received easily by reading it from and writing it to the serial port registers. The features of the 8051 serial ports are as follows:

(i) Full-duplex operation
(ii) Receive-buffered
(iii) Access using single double-buffered register SBUF

(iv) Four different modes of operation
(v) Option to use fixed or programmable baud rate

Serial Port Control SFRs:

The serial port of the 8051 is controlled by two registers in the SFR area, as shown in Table. The two registers are serial port control register SCON and serial port buffer register SBUF.

SFR name	Description	SFR address
SCON	Serial port control register	98H
SBUF	Serial port buffer register	99H

The individual bits of SCON have the functions shown in Table. As the SCON register has many individual status bits, the individual bits of this register are bit-addressable. The bit address is also given in Table. The programmer can use these bit addresses to check the status of the serial port and set the mode individually.

The D7 and D6 bits of the SCON register define the operating modes of the serial port; the basic operating modes are given in Table. The SMO and SM1 bits can select any one of the four operating modes.

Bit	Name	Bit address	Explanation of function
D7	SM0	9FH	Serial port mode select bits
D6	SM1	9EH	
D5	SM2	9DH	Multiprocessor communications enable bit
D4	REN	9CH	Receiver enable—this bit must be set, to receive characters
D3	TB8	9BH	Transmit bit 8—the 9th bit to be transmitted in modes 2 and 3
D2	RB8	9AH	Receive bit 8—the 9th bit received in modes 2 and 3
D1	TI	99H	Transmit Interrupt flag—set when a byte has been completely transmitted
D0	RI	98H	Receive Interrupt flag—set when a byte has been completely received

Fig: Bit Patterns for SCON SFR

Table Definition of bits SM0 and SM1 in SCON SFR

SM0	SM1	Serial mode	Explanation	Baud rate
0	0	0	8-bit shift register	FOSC/12
0	1	1	8-bit UART	Variable
1	0	2	9-bit UART	FOSC/64 or FOSC/32
1	1	3	9-bit UART	Variable

Programming the Serial Port:

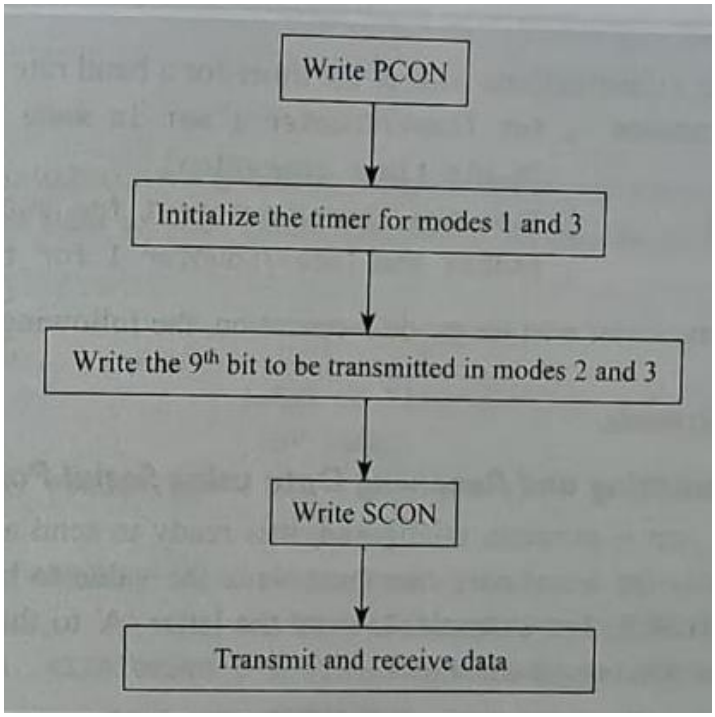


Fig: Flowchart for Programming Serial Port of 8051

Programming 8051 Timers:-

The basic Intel 8051 has two 16-bit timers. These timers are accessible to the programmer through the corresponding SFR registers. They can be initialized, run, read, and controlled by these registers. The 8051 timers have the following three general functions:

(i) Producing a delay for a definite time and then issuing an interrupt request
(ii) Counting the transitions on an external pin
(iii) Generating baud rates for the serial port

Basically, timers are digital counters that are incremented when a pulse is given to them. They can be controlled to do any of their functions, using four SFRs-TMOD, TCON, TH0/TL0, and THI/TL1. A timer overflows when it counts to the highest value and resets to 0 on next count. The overflow in the timers sets the two bits in the TCON SFR.

If the timer registers are incremented by the internal clock pulses from the microcontroller, the operation is called timing. If the timer registers get their clock pulses from an external device through the port 3 pins of the 8051, the operation is called counting. Timer 0 external input pin P3.4 (TO) is used to give clock input to timer 0 to act as a counter. Timer 1 external input pin P3.5 (T1) is used to give clock input to timer 1.

Timer SFRs:

The 8051 has two timers, both of which have similar operations and functions. The timer in the 8051 is basically a 16-bit register, which can be incremented based on the clock pulses applied to it. These timer registers are configured as SFRs. One timer is Timer 0 and the other is Timer 1. Each timer has two 8-bit SFRS. THO and TLO form the higher- and lower-order bytes of Timer 0; TH1 and TL1 form the higher- and lower-order bytes of Timer 1. These SFRs, at any time, have the timer/counter register content. So, the timers can be stopped at any time and the contents can be read from these registers. As the timers have 16 bits, the timer content/count value can vary from 0 to 65,535. The timer content will become 0 after counting up to 65,535. This is called as overflow. The overflow will be indicated by the setting of the timer overflow interrupt flag.

The two timers share two SFRs for control-the TMOD and TCON registers. The following Table shows the timer-related SFRs and their addresses.

SFR name	Description	SFR address
TH0	Timer 0 higher-order byte	8CH
TL0	Timer 0 lower-order byte	8AH
TH1	Timer 1 higher-order byte	8DH
TL1	Timer 1 lower-order byte	8BH
TCON	Timer control	88H
TMOD	Timer mode	89H

TMOD SFR:

The TMOD SFR in the 8051 controls the timer operation. The TMOD register bits are used to select the timer operating modes, counting or timing operation, and gate control. The higher-order four bits are used for Timer 1 and the lower-order four bits are used for Timer 0. The individual bits of TMOD have the functions shown in Table.

Fig: TMOD Register

Bit	Bit Name	Description
7 or 3	Gate (0 or 1)	Gate Control Bit
6 or 2	C/Tx	Counter/Timer Select Bit
5 or 1	TxM1	Timer Select Bits for Mode1
4 or 0	TxM0	Timer Select Bits for Mode0

Timer Operating Modes:

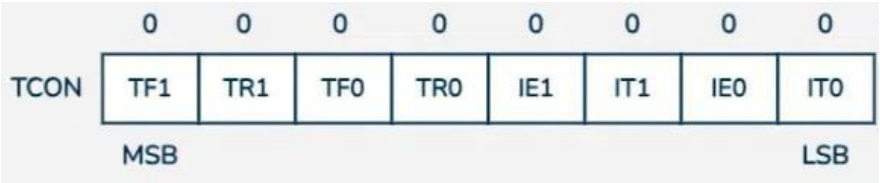
The two 16-bit timers of the 8051 can be operated in any one of four modes. Mode selection can be done by setting the proper bits in the TMOD SFR. The basic modes of operation are tabulated in Table.

TxM1	TxM0	Mode	Description
0	0	Mode 0	**13-bit Timer mode** (8 Bit of THx and 5 Bit of TLx).
0	1	Mode 1	**16-bit Timer mode**.
1	0	Mode 2	**8-bit Auto-reload mode** (TLx reload with the THx value each time when TLx overflows)**.**

TxM1	TxM0	Mode	Description
1	1	Mode 3	**Split Timer mode** - Split 16 bit timer into two 8 bit Timers (THx and TLx).

TCON SFR:

The SFR that controls the two timers and provides valuable information about them is TCON. The TCON SFR bit pattern is given in Table.



Bit	Bit Name	Description
7	TF1 (Timer 1 Overflow Flag)	Set to 1 when Timer 1 overflows else 0.
6	TR1 (Timer 1 Run Control Bit)	Set to 1 to start Timer 1, and set to 0 to stop Timer 1.
5	TF0 (Timer 0 Overflow Flag)	Set to 1 when Timer 0 overflows else 0.
4	TR0 (Timer 0 Run Control Bit)	Set to 1 to start Timer 0, and set to 0 to stop Timer 0.
3	IE1 (Interrupt 1 Edge Flag)	Set to 1 when an external interrupt 1 occurs else 0.
2	IT1 (Interrupt 1 Type Control Bit)	Set to 1 to configure external interrupt 1 as edge-triggered, and set to 0 for level-triggered.
1	IE0 (Interrupt 0 Edge Flag)	Set to 1 when an external interrupt 0 occurs else 0.
0	IT0 (Interrupt 0 Type Control Bit)	Set to 1 to configure external interrupt 0 as edge-triggered, and set to 0 for level-triggered.

Comparison of Microprocessor, Microcontroller, PIC and ARM Processors:-

Feature	Microprocessor	Microcontroller	PIC Microcontroller	ARM Processor
Definition	A CPU on a single chip used for general-purpose computing.	A single chip with CPU, memory, and peripherals for control applications.	A family of microcontrollers by Microchip Technology for embedded control.	A family of high-performance RISC-based processors by ARM Ltd.
Architecture Type	Von Neumann architecture (shared memory).	Harvard architecture (separate program and data memory).	Harvard architecture.	Modified Harvard architecture (optimized RISC).
Data Bus Width	8-bit, 16-bit, 32-bit, or 64-bit.	8-bit, 16-bit, or 32-bit.	8-bit, 16-bit, or 32-bit (varies by family).	32-bit or 64-bit.
Integration Level	CPU only (memory, I/O external).	CPU + Memory + I/O on single chip.	CPU + Flash + RAM + Peripherals on single chip.	Integrated core + memory + peripherals in System-on-Chip (SoC).
Program Memory	External ROM/EPROM/Flash.	On-chip Flash or ROM.	On-chip Flash memory.	On-chip or external Flash memory.
Data Memory (RAM)	External RAM.	On-chip RAM.	On-chip RAM.	On-chip or external RAM.
Clock Speed	High (up to several GHz).	Moderate (few MHz to hundreds of MHz).	Moderate (up to 40–200 MHz).	High (hundreds of MHz to several GHz).

Feature	Microprocessor	Microcontroller	PIC Microcontroller	ARM Processor
Power Consumption	High.	Low.	Very low.	Very low (optimized for battery devices).
Performance	High for data processing and computation.	Moderate, optimized for control tasks.	Moderate to high for embedded systems.	Very high, optimized for speed and efficiency.
Peripheral Devices	External (timers, I/O ports, ADC, DAC).	Built-in peripherals (timers, I/O ports, serial ports).	Built-in peripherals (timers, ADC, DAC, UART, SPI, I²C).	Built-in advanced peripherals (DMA, ADC, timers, communication interfaces).
Interrupt Handling	Limited support.	Multiple interrupts available.	Multiple programmable interrupt sources.	Advanced Nested Vector Interrupt Controller (NVIC).
Instruction Set	Complex (CISC).	Simple (RISC/CISC hybrid).	Simple (RISC-based).	Simple and efficient (RISC-based).
Speed of Execution	Slower per instruction (due to CISC).	Moderate.	Fast (RISC pipeline).	Very fast (multi-stage pipeline).
Programming Languages	Assembly, C, C++.	Assembly, Embedded C.	Assembly, Embedded C (MPLAB IDE).	C, C++, Embedded C, Python (for some ARM MCUs).
Cost	Expensive (requires external components).	Low-cost.	Low-cost and compact.	Moderate cost, depending on core.
Applications	Personal computers, servers, laptops, complex systems.	Embedded systems, industrial control, appliances, robotics.	Small embedded applications, IoT devices, home automation.	Smartphones, tablets, automotive systems, IoT, advanced embedded systems.

Feature	Microprocessor	Microcontroller	PIC Microcontroller	ARM Processor
Examples	Intel 8086, Pentium, AMD Ryzen.	Intel 8051, AVR ATmega, Arduino.	PIC16F877A, PIC18F4550, dsPIC33.	ARM Cortex-M (microcontrollers), Cortex-A (smartphones), Cortex-R (real-time systems).
Advantages	High processing power, supports multitasking.	Compact, low power, single-chip control solution.	Cost-effective, reliable, good documentation and support.	High speed, energy-efficient, widely adopted.
Disadvantages	Requires many external components, high power use.	Limited processing speed, smaller memory.	Limited computational power, simple architecture.	Complex architecture, higher cost for high-end variants.