UNIT-5: MICROPROCESSOR (ARCHITECTURE AND PROGRAMMING -8086-16 BIT)

5.1 INTRODUCTION:

- 8086 Microprocessor is an enhanced version of 8085Microprocessor that was designed by Intel in 1976.
- It is a 16-bit Microprocessor having 20 address lines and16 data lines that provides up to 1MB storage.
- It consists of powerful instruction set, which provides operations like multiplication and division easily.
- It supports two modes of operation, i.e. Maximum mode and Minimum mode.
- Maximum mode is suitable for system having multiple processors and Minimum mode is suitable for system having a single processor.

5.2 8086 MICROPROCESSOR FEATURES:

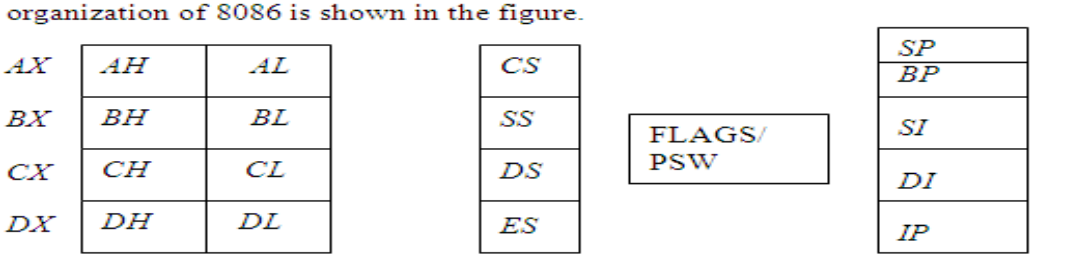
- It is 16-bit microprocessor
- It has a 16-bit data bus, so it can read data from or write data to memory and ports either 16-bit or 8-bit at a time.
- It has 20 bit address bus and can access up to 2^{20} memory locations (1 MB).
- It can support up to 64K I/O ports
- It provides 14, 16-bit registers
- It has multiplexed address and data bus AD_0-AD_{15} & A_{16}-A_{19}
- It requires single phase clock with 33% duty cycle to provide internal timing.
- Pre fetches up to 6 instruction bytes from memory and queues them in order to speed up the processing.
- 8086 supports 2 modes of operation
    1. Minimum mode
    2. Maximum mode
- It is available in 3 versions based on the frequency of operation –
    1. 8086 → 5MHz
    2. 8086-2 → 8MHz
    3. (c)8086-1 → 10 MHz
- It uses two stages of pipelining, i.e. fetch Stage and Execute Stage, which improves performance.
- Fetch stage can pre fetch up to 6 bytes of instructions and stores them in the queue.
- Execute stage executes these instructions.
- It has 256 vectored interrupts.
- It consists of 29,000 transistors.

5.3 REGISTER ORGANIZATION:

- 8086 has a powerful set of registers known as general purpose registers and special purpose registers.
- All of them are 16-bit registers.
- **General purpose registers:**
  - ✓ These registers can be used as either 8-bit registers or 16-bit registers.
  - ✓ They may be either used for holding data, variables and intermediate results temporarily or for other purposes like a counter or for storing offset address for some particular addressing modes etc.
- **Special purpose registers:**
  - ✓ These registers are used as segment registers, pointers, index registers or as offset storage registers for particular addressing modes.

- **The 8086 registers are classified into the following types:**
  - ✓ General Data Registers
  - ✓ Segment Registers
  - ✓ Pointers and Index Registers
  - ✓ Flag Register

The register set of 8086 can be categorized into 4 different groups. The register organization of 8086 is shown in the figure.



Register organization of 8086

1. General Data Registers:

- The registers AX, BX, CX and DX are the general purpose 16-bit registers.
- AX is used as 16-bit accumulator. The lower 8-bit is designated as AL and higher 8-bit is designated as AH.
- AL Can be used as an 8-bit accumulator for 8-bit operation.
- All data register can be used as either 16 bit or 8 bit. BX is a 16 bit register, but BL indicates the lower 8-bit of BX and BH indicates the higher 8-bit of BX.
- The register BX is used as offset storage for forming physical address in case of certain addressing modes.
- The register CX is used default counter in case of string and loop instructions.
- DX register is a general purpose register which may be used as an implicit operand or destination in case of a few instructions.

2. Segment Registers:

- There are 4 segment registers. They are:
  - ✓ Code Segment Register(CS)
  - ✓ Data Segment Register(DS)
  - ✓ Extra Segment Register(ES)
  - ✓ Stack Segment Register(SS)

- The 8086 architecture uses the concept of **segmented memory**. 8086 able to address a memory capacity of 1 megabyte and it is byte organized. This 1 megabyte memory is divided into 16 logical segments. Each segment contains 64 Kbytes of memory.

- **Code segment register (CS):**
  It is used for addressing memory location in the code segment of the memory, where the executable program is stored.

- **Data segment register (DS):**
  It points to the data segment of the memory where the data is stored.

- **Extra Segment Register (ES) :**
  It also refers to a segment in the memory which is another data segment in the memory.

- **Stack Segment Register (SS):**
  - ➢ It is used for addressing stack segment of the memory. The stack segment is that segment of memory which is used to store stack data.
  - ➢ While addressing any location in the memory bank, the **physical address** is calculated from two parts:

    Physical address= segment address + offset address
  - ➢ The first is segment address, the segment registers contain 16-bit segment base addresses, related to different segment.
  - ➢ The second part is the offset value in that segment.

3. Pointers and Index Registers:

- The index and pointer registers are given below:
  - ✓ IP—Instruction pointer-store memory location of next instruction to be executed
  - ✓ BP—Base pointer
  - ✓ SP—Stack pointer
  - ✓ SI—Source index
  - ✓ DI—Destination index

- The pointers registers contain offset within the particular segments.
  - ✓ The pointer register IP contains offset within the code segment.
  - ✓ The pointer register BP contains offset within the data segment.
  - ✓ Thee pointer register SP contains offset within the stack segment.

- The index registers are used as general purpose registers as well as for offset storage in case of indexed, base indexed and relative base indexed addressing modes.
- The register SI is used to store the offset of source data in data segment.
- The register DI is used to store the offset of destination in data or extra segment.
- The index registers are particularly useful for string manipulation.

4. 8086 flag register and its functions:

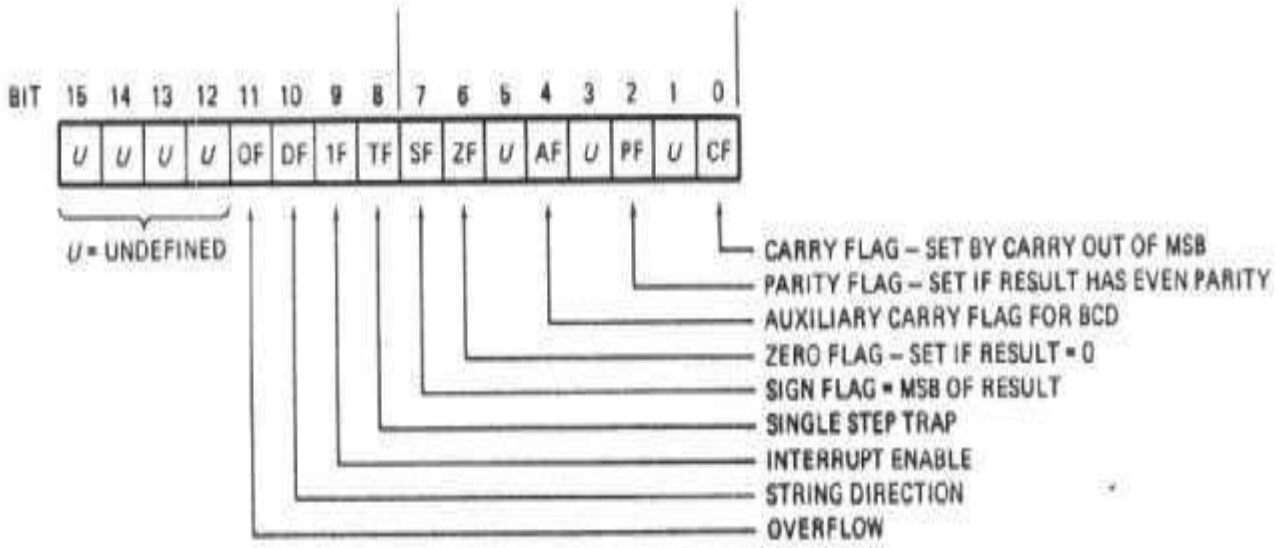
- The 8086 flag register contents indicate the results of computation in the ALU. It also contains some flag bits to control the CPU operations.

- A 16 bit flag register is used in 8086. It is divided into two parts.
  - ✓ Condition code or status flags
  - ✓ Machine control flags

- The **condition code flag register** is the lower byte of the 16-bit flag register. The condition code flag register is identical to 8085 flag register, with an additional overflow flag.

- The **control flag register** is the higher byte of the flag register. It contains three flags namely direction flag (D), interrupt flag (I) and trap flag (T).

Flag register configuration



The description of each flag bit is as follows:

SF (Sign Flag):
This flag is set, when the result of any computation is negative. For signed computations the sign flag equals the MSB of the result.

ZF (Zero Flag):

This flag is set, if the result of the computation or comparison performed by the previous instruction is zero.

PF (Parity Flag):
This flag is set to 1, if the lower byte of the result contains even number of 1's.

CF (Carry Flag):
This flag is set, when there is a carry out of MSB in case of addition or a borrow in case of subtraction.

AF (Auxiliary Carry Flag):
This is set, if there is a carry from the lowest nibble, i.e., bit three during addition, or borrow for the lowest nibble, i.e., bit three, during subtraction.

OF (Over flow Flag):
This flag is set, if an overflow occurs, i.e., if the result of a signed operation is large enough to accommodate in a destination register. The result is of more than 7-bits in size in case of 8-bit signed operation and more than 15-bits in size in case of 16-bit sign operations, and then the overflow will be set.

TF (Tarp Flag):

If this flag is set, the processor enters the single step execution mode. The processor executes the current instruction and the control is transferred to the Trap interrupt service routine.

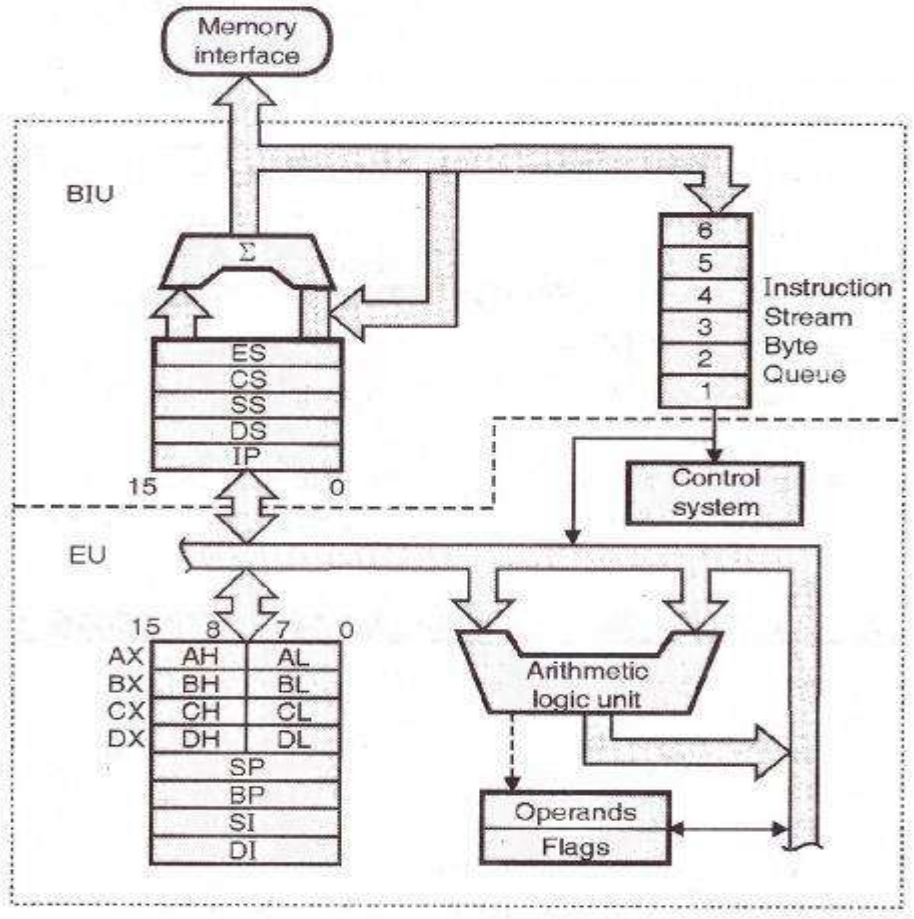
IF (Interrupt Flag):
If this flag is set, the mask able interrupts are recognized by the CPU, otherwise they are ignored.

D (Direction Flag):
This is used by string manipulation instructions. If this flag bit is '0', the string is processed beginning from the lowest address to the highest address, i.e., auto incrementing mode. Otherwise, the string is processed from the highest address towards the lowest address, i.e., auto decrementing mode.

5.4 ARCHITECTURE OF 8086 MICROPROCESSOR:

- As shown in the below figure, the 8086 CPU is divided into two independent functional parts
  - ✓ Bus Interface Unit(BIU)
  - ✓ Execution Unit(EU)

Dividing the work between these two units' speeds up processing.



8086 internal architecture

The Execution Unit (EU):

- The execution unit of the 8086 tells the BIU where to fetch instructions or data from, decodes instructions, and executes instructions.
- The EU contains **control circuitry,** which directs internal operations.
- A decoder in the EU translates instructions fetched from memory into a series of actions, which the EU carries out.
- The EU has a 16-bit **arithmetic logic unit** (ALU) which can add, subtract, AND, OR, XOR, increment, decrement, complement or shift binary numbers.

- **The main functions of EU are:**
  - ✓ Decoding of Instructions
  - ✓ Execution of instructions

- ✓ **Steps:**
- ✓ EU extracts instructions from top of queue in BIU
- ✓ Decode the instructions
- ✓ Generates operands if necessary
- ✓ Passes operands to BIU & requests it to perform read or write bus cycles to memory or I/O
- ✓ Perform the operation specified by the instruction on operands

Or

- Execution unit gives instructions to BIU stating from where to fetch the data and then decode and execute those instructions. Its function is to control operations on data using the instruction decoder & ALU. EU has no direct connection with system buses as shown in the above figure, it performs operations over data through BIU.

Bus Interface Unit (BIU):

- The BIU sends out addresses, fetches instructions from memory, reads data from ports and memory, and writes data to ports and memory.
- In simple words, the BIU handles all transfers of data and addresses on the buses for the execution unit.

Or

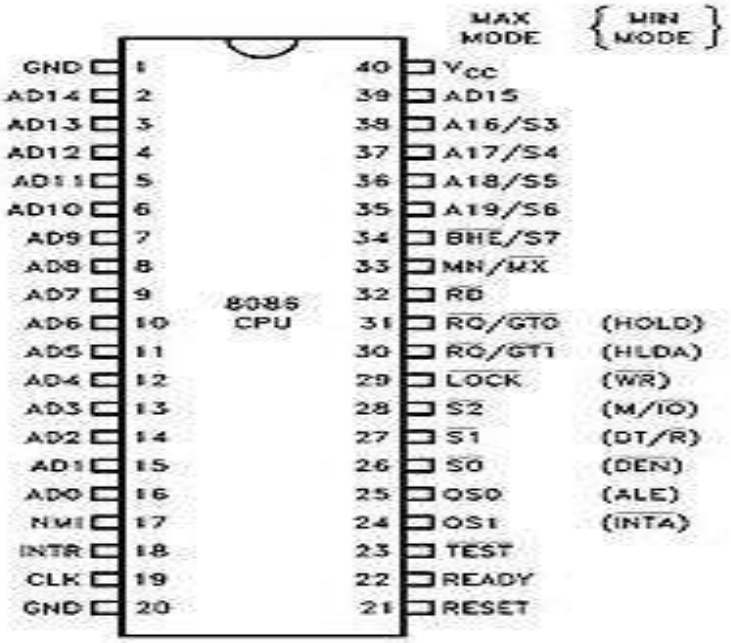
- BIU takes care of all data and addresses transfers on the buses for the EU like sending addresses, fetching instructions from the memory, reading data from the ports and the memory as well as writing data to the ports and the memory.

- EU has no direction connection with System Buses so this is possible with the BIU. EU and BIU are connected with the Internal Bus.

8086 HAS PIPELINING ARCHITECTURE:

- While the EU is decoding an instruction or executing an instruction, which does not require use of the buses, the BIU fetches up to six instruction bytes for the following instructions.
- The BIU stores these pre-fetched bytes in a first-in-first-out register set called a queue.
- When the EU is ready for its next instruction from the queue in the BIU. This is much faster than sending out an address to the system memory and waiting for memory to send back the next instruction byte or bytes.
- Except in the case of JMP and CALL instructions, where the queue must be dumped and then reloaded starting from a new address, this pre-fetch and queue scheme greatly speeds up processing.
- Fetching the next instruction while the current instruction executes is called **pipelining**.

5.5 PIN DIAGRAM OF 8086:



Intel 8086 is a 16-bit HMOS microprocessor. It is available in 40 pin DIP chip. It uses a 5V DC supply for its operation. The 8086 uses 20-line address bus. It has a 16-line data bus. The 20 lines of the address bus operate in multiplexed mode. The 16-low order address bus lines have been multiplexed with data and 4 high-order address bus lines have been multiplexed with status signals.

AD0-AD15:
Address/Data bus. These are low order address bus. They are multiplexed with data. When AD lines are used to transmit memory address the symbol A is used instead of AD, for example A0-A15. When data are transmitted over AD lines the symbol D is used in place of AD, for example D0-D7, D8-D15 or D0-D15.

A16-A19:
High order address bus. These are multiplexed with status signals.

S2, S1, S0:
Status pins. These pins are active during T4, T1 and T2 states and is returned to passive state (1, 1, 1 during T3 or Tw (when ready is inactive). These are used by the 8288 bus controller for generating all the memory and I/O operation) access control signals. Any change in S2, S1, and S0 during T4 indicates the beginning of a bus cycle.

S2	S1	S0	Characteristics
0	0	0	Interrupt acknowledge
0	0	1	Read I/O port

0	1	0	Write I/O port
0	1	1	Halt
1	0	0	Code access1 0 1 Read memory
1	1	0	Write memory
1	1	1	Passive State

A16/S3, A17/S4, A18/S5, A19/S6:

- The specified address lines are multiplexed with corresponding status signals.
- These are the 4 address/status buses. During the first clock cycle, it carries 4-bit address and later it carries status signals.

BHE'/S7:

- Bus High Enable/Status. During T1 it is low. It is used to enable data onto the most significant half of data bus, D8-D15.
- 8-bit device connected to upper half of the data bus use BHE (Active Low) signal.
- It is multiplexed with status signal S7.
- S7 signal is available during T2, T3 and T4.

RD':

This is used for read operation. It is an output signal. It is active when low.

READY:

This is the acknowledgement from the memory or slow device that they have completed the data transfer. The signal made available by the devices is synchronized by the 8284A clock generator to provide ready input to the microprocessor. The signal is active high (1).

INTR:

Interrupt Request. This is triggered input. This is sampled during the last clock cycles of each instruction for determining the availability of the request. If any interrupt request is found pending, the processor enters the interrupt acknowledge cycle. This can be internally masked after resulting the interrupt enable flag. This signal is active high (1) and has been synchronized internally.

NMI:

Non maskable interrupt. This is an edge triggered input which results in a type II interrupt. A subroutine is then vectored through an interrupt vector lookup table which is located in the system memory. NMI is non-maskable internally by software. A transition made from low (0) to high (1) initiates the interrupt at the end of the current instruction. This input has been synchronized internally.

INTA:
Interrupt acknowledge. It is active low (0) during T2, T3 and Tw of each interrupt acknowledge cycle.

MN/MX':
Minimum/Maximum. This pin signal indicates what mode the processor will operate in.

RQ'/GT1', 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.

LOCK':
- It's an active low pin. It indicates that other system bus masters have not been allowed to gain control of the system bus while LOCK' is active low (0). The LOCK signal will be active until the completion of the next instruction.
- 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

RESET:
This pin requires the microprocessor to terminate its present activity immediately. The signal must be active high (1) for at least four clock cycles.

TEST':
This examined by a 'WAIT' instruction. If the TEST pin goes low (0), execution will continue, else the processor remains in an idle state. The input is internally synchronized during each of the clock cycle on leading edge of the clock.

CLK:
- Clock Input. The clock input provides the basic timing for processing operation and bus control activity. It's an asymmetric square wave with a 33% duty cycle.

Vcc:
Power Supply (+5V D.C.)

GND:
Ground

QS1, QS0:
Queue Status. These signals indicate the status of the internal 8086 instruction queue according to the table shown below

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

DT/R:
Data Transmit/Receive. This pin is required in minimum systems that want to use an 8286 or 8287 data bus transceiver. The direction of data flow is controlled through the transceiver.

DEN:
Data enable. This pin is provided as an output enable for the 8286/8287 in a minimum system which uses transceiver. DEN is active low (0) during each memory and input-output access and for INTA cycles.

HOLD/HOLDA:
HOLD indicates that another master has been requesting a local bus .This is an active high (1). The microprocessor receiving the HOLD request will issue HLDA (high) as an acknowledgement in the middle of a T4 or T1 clock cycle.

ALE:
Address Latch Enable. ALE is provided by the microprocessor to latch the address into the 8282 or 8283 address latch. It is an active high (1) pulse during T1 of any bus cycle. ALE signal is never floated, is always integer.

5.6 GENERAL BUS OPERATION OF 8086:

- The 8086 has a combined address and data bus commonly referred as a time multiplexed address and data bus.
- The main reason behind multiplexing address and data over the same pins is the maximum utilization of processor pins and it facilitates the use of 40 pin standard DIP package.
- The bus can be de multiplexed using a few latches and transceivers, whenever required.
- Basically, all the processor bus cycles consist of at least four clock cycles. These are referred to as T1, T2, T3, and T4. The address is transmitted by the processor during T1. It is present on the bus only for one cycle.
- The negative edge of this ALE pulse is used to separate the address and the data or

status information. In maximum mode, the status lines S0, S1 and S2 are used to indicate the type of operation.

- Status bits S3 to S7 are multiplexed with higher order address bits and the BHE signal. Address is valid during T1 while status bits S3 to S7 are valid during T2 through T4.

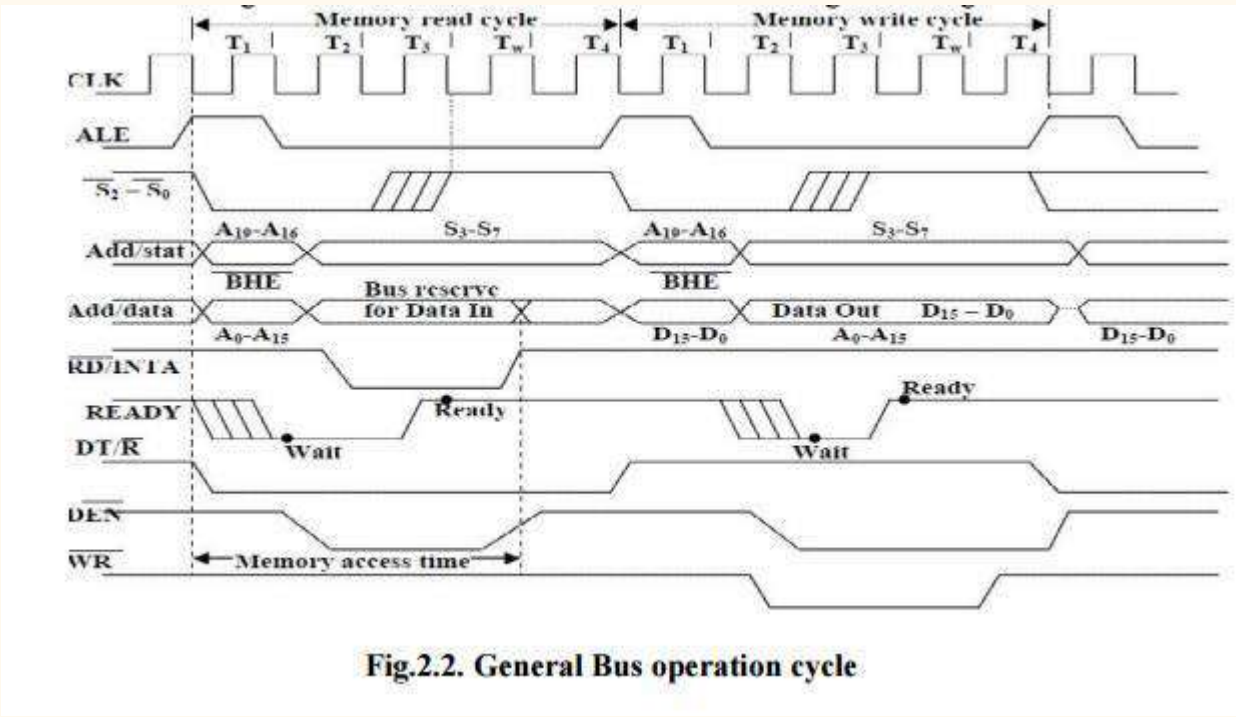


Fig.2.2. General Bus operation cycle

Maximum mode

- In the maximum mode, the 8086 is operated by strapping the MN/MX pin to ground.
- In this mode, the processor derives the status signal S2, S1, S0. Another chip called bus controller derives the control signal using this status information.
- In the maximum mode, there may be more than one microprocessor in the system configuration.

Minimum mode

- In a minimum mode 8086 system, the microprocessor 8086 is operated in minimum mode by strapping its MN/MX pin to logic 1.
- In this mode, all the control signals are given out by the microprocessor chip itself.
- There is a single microprocessor in the minimum mode system.

5.7 8086 MEMORY ORGANIZATION:

- Segmented Memory Two types of memory organization are used:
- Linear addressing where the entire memory is available to the processor at all the times (Motorola 68000 family).
- Segmented addressing where the memory space is divided into several segments and the processor is limited to access program instructions and data in specific segments.
- 8086 Memory Organization Each memory location 8086 is a byte while the 8086 is a 16-bits microprocessor.

Memory Segmentation:

- The memory in an 8086 based system is organized as segmented memory.
- The CPU 8086 is able to access 1MB of physical memory. The complete 1MB of memory can be divided into 16 segments, each of 64KB size and is addressed by one of the segment register.
- The 16-bit contents of the segment register actually point to the starting location of a particular segment. The address of the segments may be assigned as 0000H to F000h respectively.
- To address a specific memory location within a segment, we need an offset address. The offset address values are from 0000H to FFFFH so that the physical addresses range from 00000H to FFFFFH.

Advantages of the segmented memory scheme are as follows:

- Allows the memory capacity to be 1MB although the actual addresses to be handled are of 16-bit size.
- Allows the placing of code, data and stack portions of the same program in different parts (segments) of memory, for data and code protection.
- Permits a program and/or its data to be put into different areas of memory each time the program is executed, i.e., provision for relocation is done.

Overlapping and Non-overlapping Memory segments:

- In the overlapping area locations physical address = CS1+IP1 = CS2+IP2. Where '+' indicates the procedure of physical address formation.

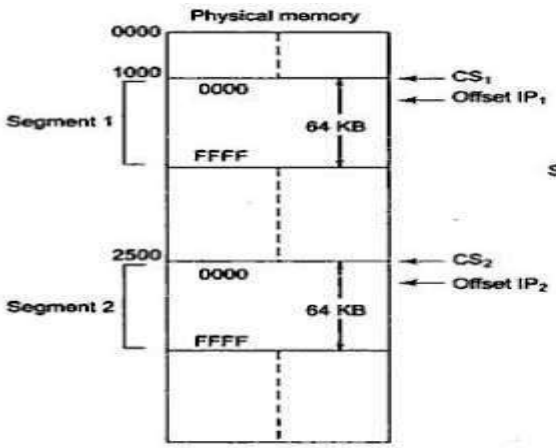


Fig. 1.3(a) *Non-overlapping Segments*

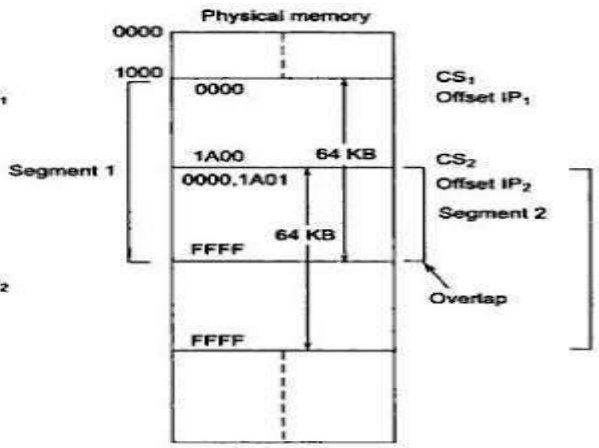
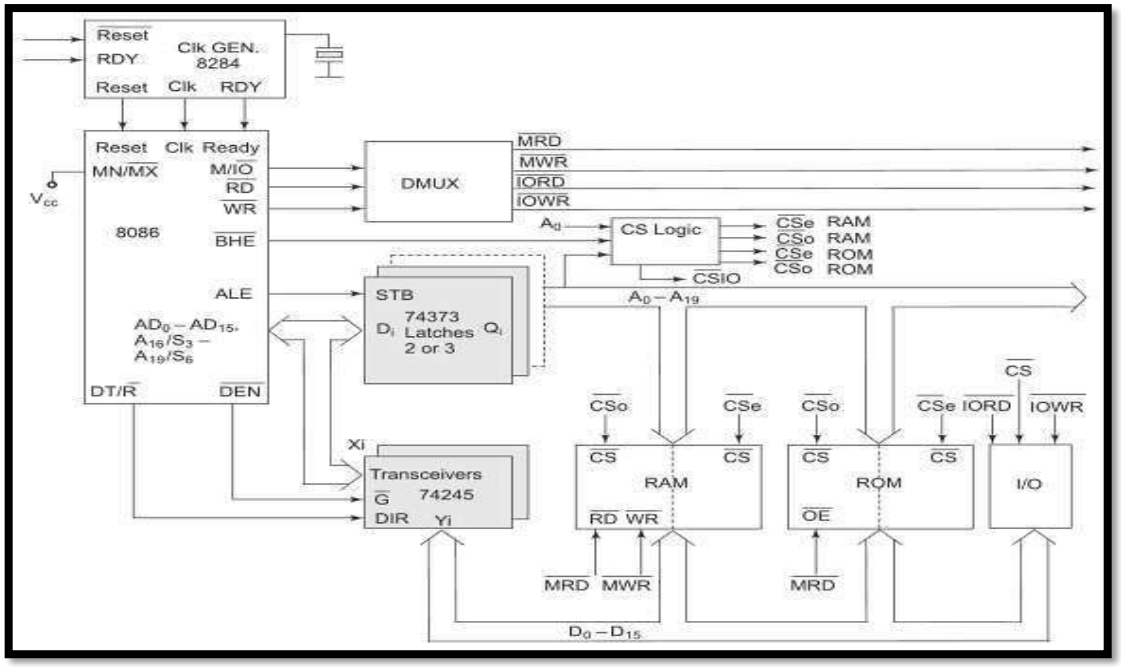


Fig. 1.3(b) *Overlapping Segments*

5.8 MINIMUM MODE & TIMINGS:

Minimum Mode 8086 System:

- The microprocessor 8086 is operated in minimum mode by strapping its MN/MX pin to logic 1.
- In this mode, all the control signals are given out by the microprocessor chip itself. There is a single microprocessor in the minimum mode system.
- The remaining components in the system are latches, transreceivers, clock generator, memory and I/O devices.
- Some type of chip selection logic may be required for selecting memory or I/O devices, depending upon the address map of the system
- Latches are generally buffered output D-type flip-flops like 74LS373 or 8282. They are used for separating the valid address from the multiplexed address/data signals and are controlled by the ALE signal generated by 8086.



Minimum Mode Configuration for 8086

- Since it has 20 address lines and 16 data lines, the 8086 CPU requires three octal address latches and two octal data buffers for the complete address and data separation.
- Transceivers are the bidirectional buffers and sometimes they are called as data amplifiers. They are required to separate the valid data from the time multiplexed address/data signal.
- They are controlled by two signals, namely, DEN' and DT/R'. The DEN' signal indicates that the valid data is available on the data bus, while DT/R' indicates the direction of data, i.e. from or to the processor.
- The system contains memory for the monitor and users program storage. Usually, EPROMS are used for monitor storage, while RAMs for users program storage.
- A system may contain I/O devices for communication with the processor as well as some special purpose I/O devices.
- The clock generator generates the clock from the crystal oscillator and then shapes it and divides to make it more precise so that it can be used as an accurate timing reference for the system.
- The clock generator also synchronizes some external signals with the system clock.
- The working of the minimum mode configuration system can be better described in terms of the timing diagrams rather than qualitatively describing the operations.
- The opcode fetch and read cycles are similar. Hence the timing diagram can be categorized in two parts, the first is the timing diagram for read cycle and the second is the timing diagram for write cycle.

Timing Diagrams:

- Timing diagram is graphical representation of the operations of microprocessor with respect to the time.
- **State**: one cycle of the clock is called state.
- **Machine cycle:** The basic microprocessor operation such as reading a byte from memory or writing a byte to a port is called machine cycle and made up of more than one state.
- **Instruction cycle:** The time required for microprocessor to fetch and execute an entire instruction is called Instruction cycle and made up of more than one machine cycle.

Note: An instruction cycle is made up of machine cycles, and a machine cycle is made up of states. The time for a state is determined by the frequency of the clock signal.

Read cycle timing diagram for Minimum mode:

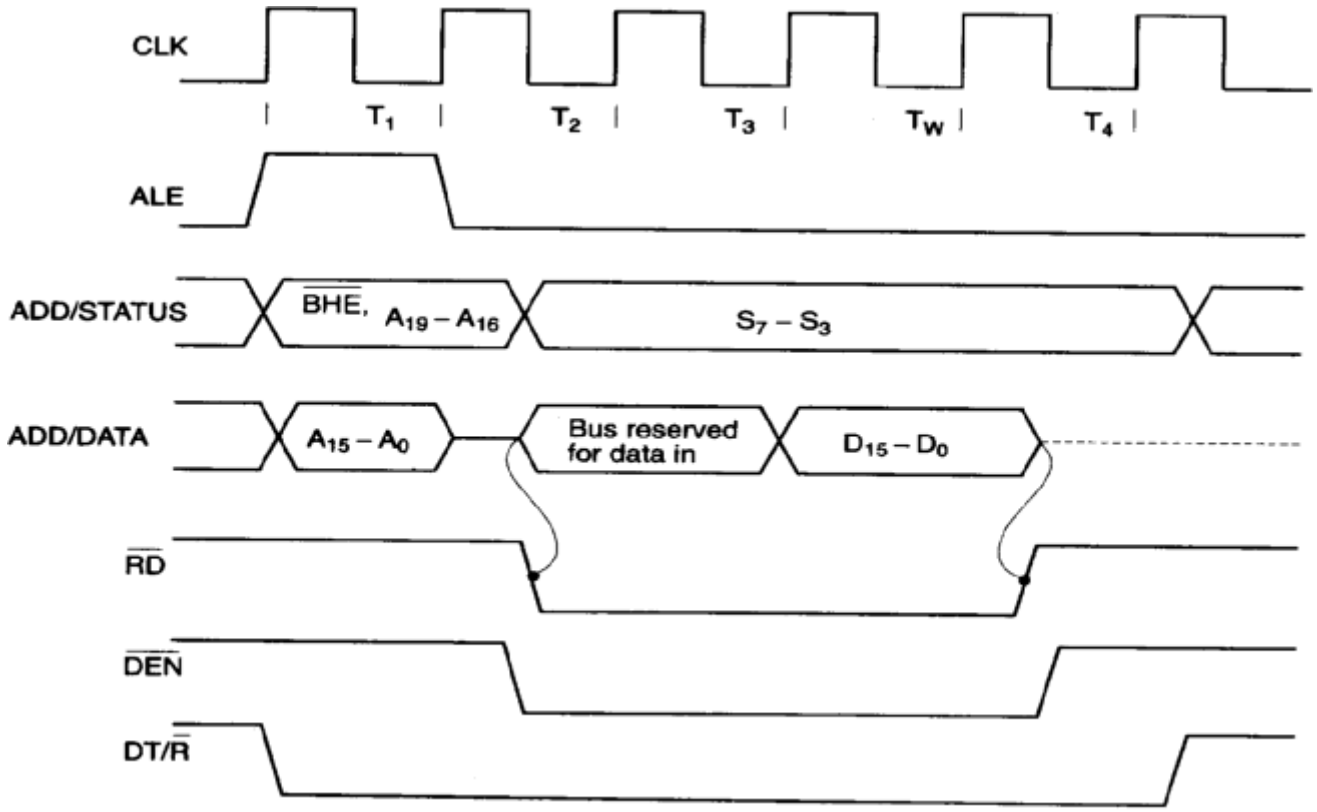


Fig. 1.9(a) *Read Cycle Timing Diagram for Minimum Mode*

- The best way to analyze a timing diagram such as the one to think of time as a vertical line moving from left to right across the diagram.
- **The read cycle** begins in T_1 with the assertion of the address latch enable (ALE) signal and also M/IO' signal.
- During the negative going edge of this signal, the valid address is latched on the local bus. The BHE' and A0 signals address low, high or both bytes.
- From T_1 to T_4, the M/IO' signal indicate a memory or I/O operation. At T_2, the address is removed from the local bus and is sent to the output. The bus is then tristated. The read ($\overline{RD}$') control signal is also activated in T_2.
- The read ($\overline{RD}$') signal causes the addressed device to enable its data bus driver. After goes low, the valid data is available on the data bus. The addressed device will drive the READY line high. When the processor returns the read signal to high level, the addressed device will again tristate its bus drivers.

Write cycle timing diagram for Minimum mode:

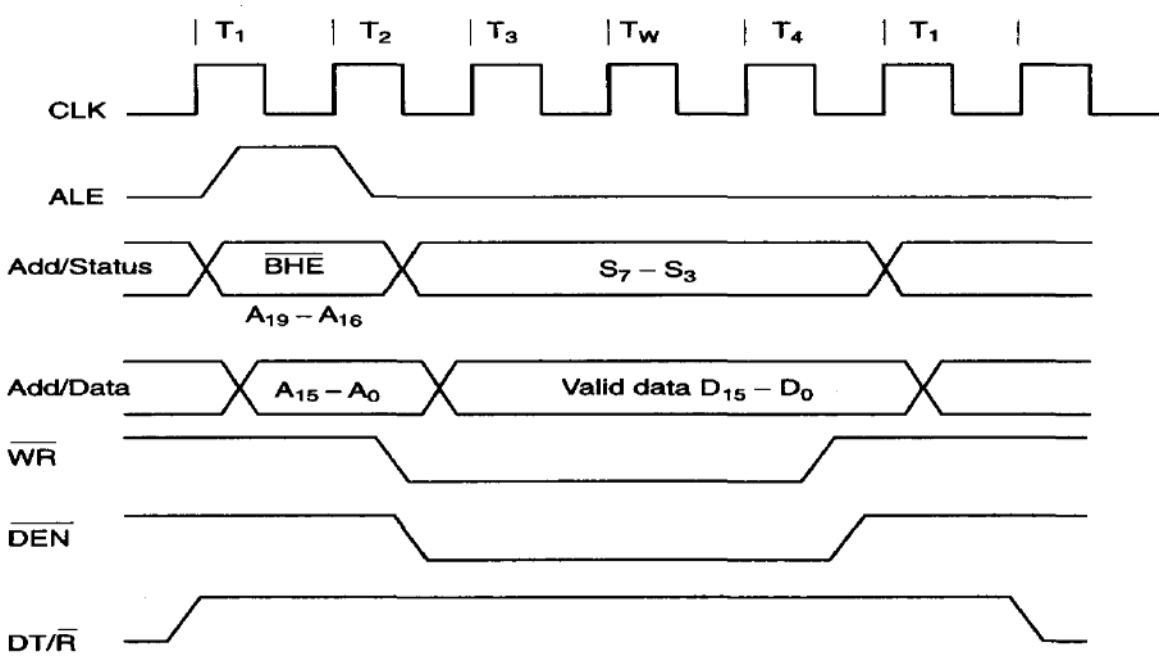


Fig. 1.9(b) *Write Cycle Timing Diagram for Minimum Operation*

- **A write cycle** also begins with the assertion of ALE and the emission of the address. The M/IO' signal is again asserted to indicate a memory or I/O operation.
- In T_2, after sending the address in T_1, the processor sends the data to be written to the addressed location. The data remains on the bus until middle of T_4 state. The WR' becomes active at the beginning of T_2 (unlike RD' is somewhat delayed in T_2 to provide time for floating).
- The BHE' and A_0 signals are used to select the proper byte or bytes of memory or I/O word to be read or written.

M/IO	RD	WR	Transfer Type
0	0	1	I/O read
0	1	0	I/O write
1	0	1	Memory read
1	1	0	Memory write

- The M/IO', RD' and WR' signals indicate the types of data transfer as specified in Table.

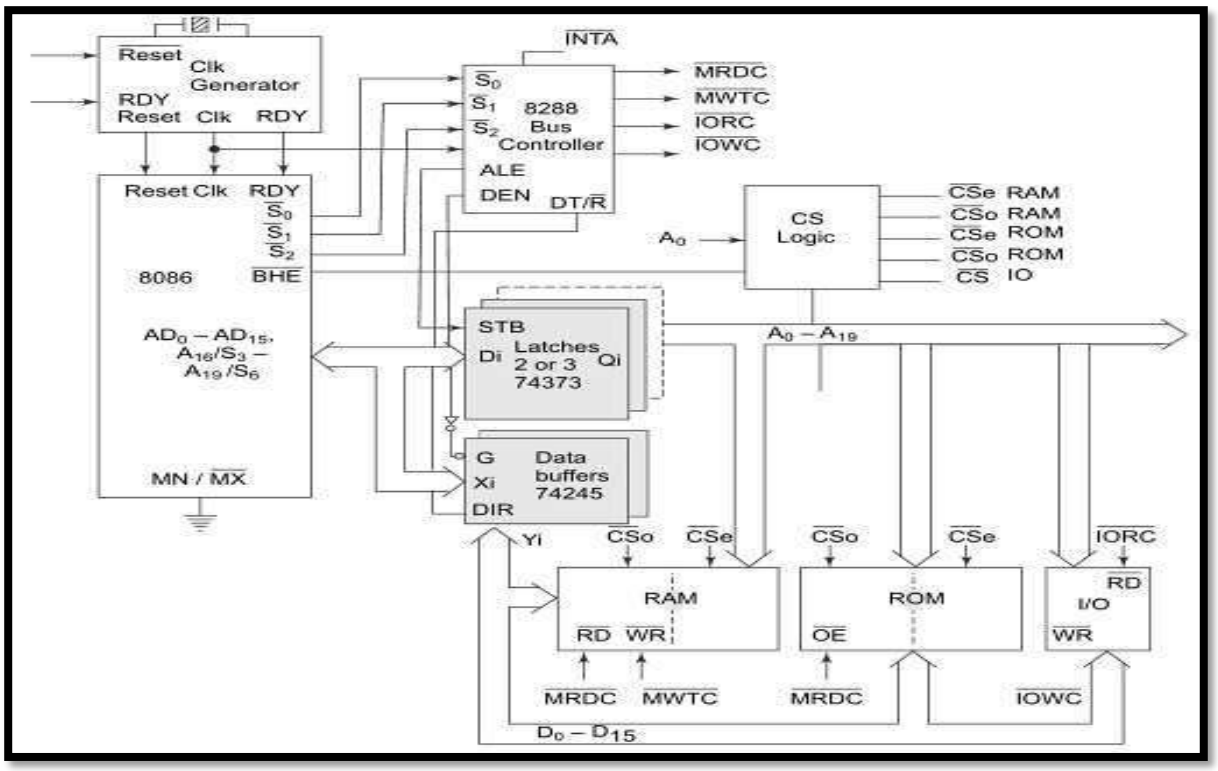
5.9 MAXIMUM MODE &TIMINGS

- In the maximum mode, the 8086 is operated by strapping the MN/MX' pin to ground. In this mode, the processor derives the status signals S2', S1' and S0'. Another chip called bus controller derives the control signals using this status information.
- In the maximum mode, there may be more than one microprocessor in the system configuration. The other components in the system are the same as in the minimum mode system. The general system organization is as shown in the below figure. The basic functions of the bus controller chip IC8288, is to derive control signals like RD' and WR' (for memory and I/O devices), DEN, DT/R', ALE, etc. using the information made available by the processor on the status lines.

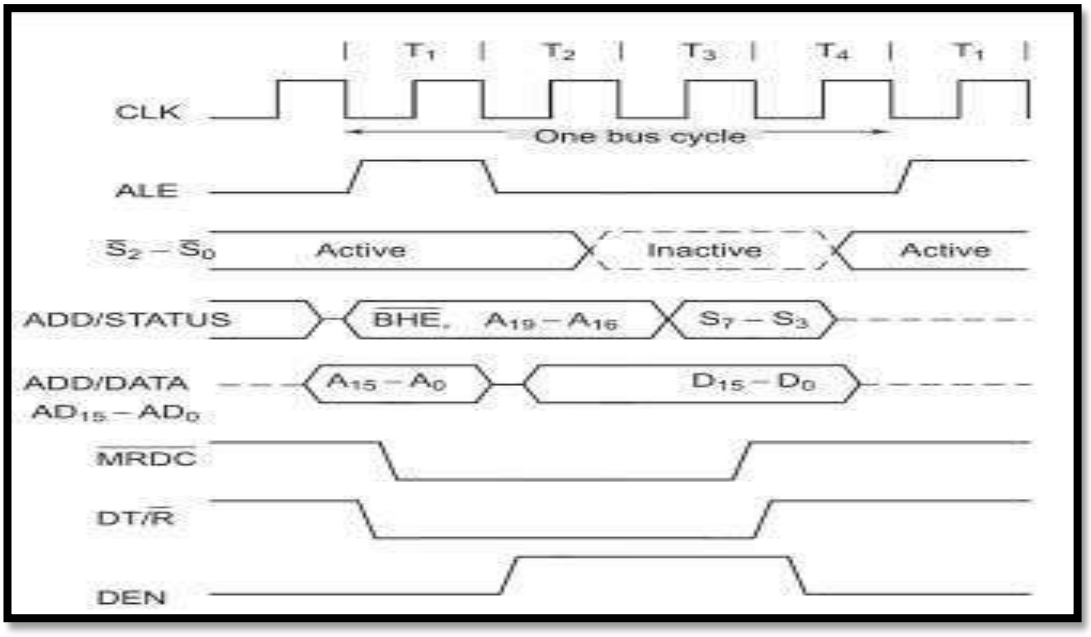
- The bus controller chip has input lines S2', S1' and S0' and CLK. These inputs to 8288 are driven by the CPU. It derives the outputs ALE, DEN, DT/R', MWTC', MRDC', IORC', IOWC' and INTA'.

- INTA' pin is used to issue two interrupt acknowledge pulses to the interrupt controller or to an interrupting device.

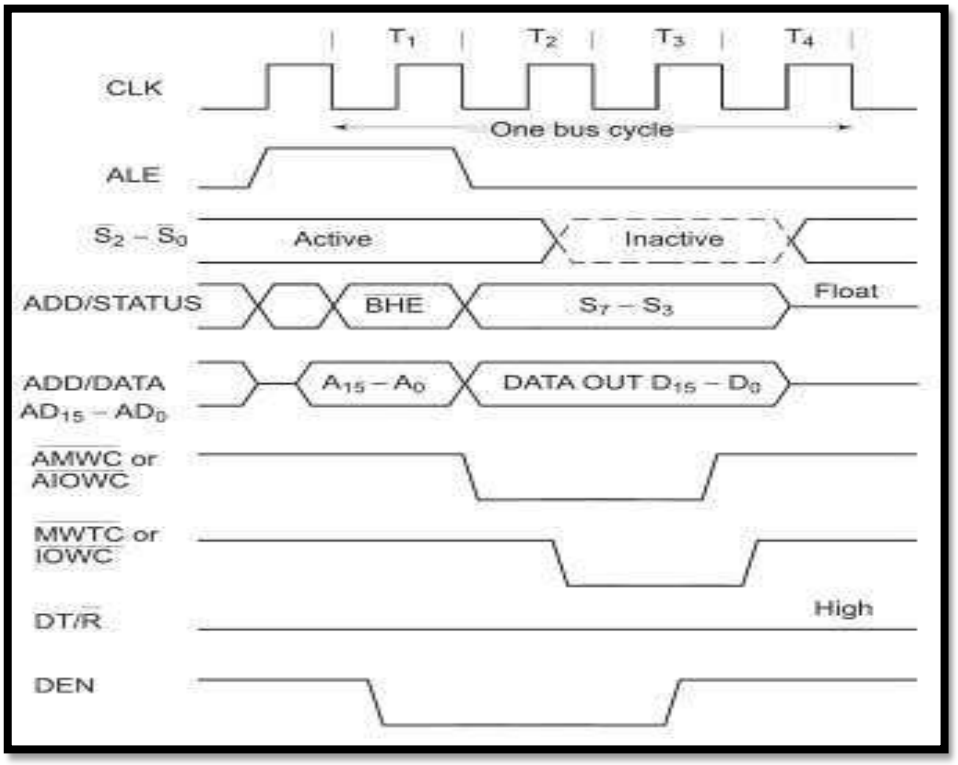
- IORC*, IOWC* are I/O read command and I/O write command signals respectively. These signals enable an IO interface to read or write the data from or to the addressed port. The MRDC*, MWTC* are memory read command and memory write command signals respectively and may be used as memory read and write signals. All these command signals instruct the memory to accept or send data from or to the bus.
- The maximum mode system timing diagrams are also divided in two portions as read (input) and write (output) timing diagrams. The address/data and address/status timings are similar to the minimum mode. ALE is asserted in T1, just like minimum mode. The only difference lies in the status signals used and the available control and advanced command signals.



Read cycle timing diagram for Maximum mode:



Write cycle timing diagram for Maximum mode:



5.10 INTERRUPTS:

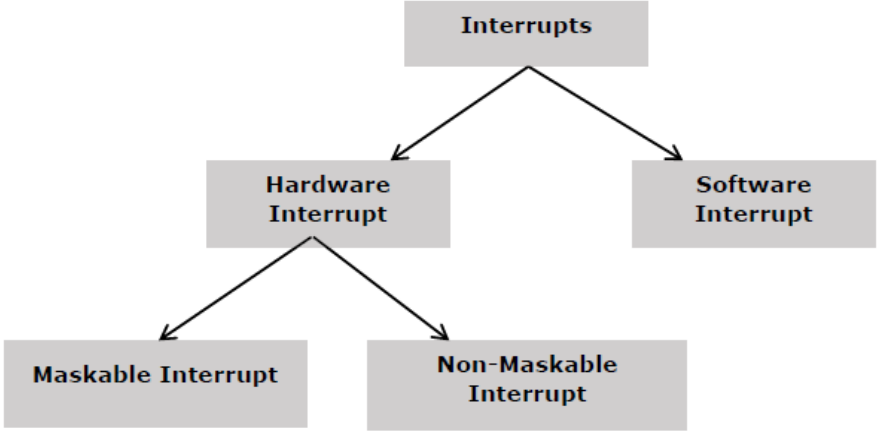
Definition:

The meaning of „interrupts" is to break the sequence of operation. While the CPU is executing a program, on „interrupt" breaks the normal sequence of execution of instructions, diverts its execution to some other program called Interrupt Service Routine (ISR).After executing ISR , the control is transferred back again to the main program. Interrupt processing is an alternative to polling.

Or

Interrupt is the method of creating a temporary halt during program execution and allows peripheral devices to access the microprocessor. The microprocessor responds to that interrupt with an **ISR** (Interrupt Service Routine), which is a short program to instruct the microprocessor on how to handle the interrupt.

The following image shows the types of interrupts we have in a 8086 microprocessor −



Need for Interrupt:
Interrupts are particularly useful when interfacing I/O devices that provide or require data at relatively low data transfer rate.

Types of Interrupts:
There are two types of Interrupts in 8086.

1. Hardware Interrupts
2. Software Interrupts

HARDWARE INTERRUPTS:

- Hardware interrupt is caused by any peripheral device by sending a signal through a specified pin to the microprocessor.
- The 8086 has two hardware interrupt pins, i.e. NMI and INTR. NMI is a non-maskable interrupt and INTR is a maskable interrupt having lower priority. One more interrupt pin associated is INTA called interrupt acknowledge.

NMI (non-maskable):

It is a single non-maskable interrupt pin (NMI) having higher priority than the maskable interrupt request pin (INTR) and it is of type 2 interrupt.

When this interrupt is activated, these actions take place: –

- Completes the current instruction that is in progress.
- Pushes the Flag register values on to the stack.
- Pushes the CS (code segment) value and IP (instruction pointer) value of the return address on to the stack.
- IP is loaded from the contents of the word location 00008H.
- CS is loaded from the contents of the next word location 0000AH.
- Interrupt flag and trap flag are reset to 0.

INTR (Maskable):

- The INTR is a maskable interrupt because the microprocessor will be interrupted only if interrupts are enabled using set interrupt flag instruction. It should not be enabled using clear interrupt Flag instruction.
- The INTR interrupt is activated by an I/O port. If the interrupt is enabled and NMI is disabled, then the microprocessor first completes the current execution and sends '0' on INTA pin twice. The first '0' means INTA informs the external device to get ready and during the second '0' the microprocessor receives the 8 bit, say X, from the programmable interrupt controller.

These actions are taken by the microprocessor: –

- First completes the current instruction.
- Activates INTA output and receives the interrupt type, say X.
- Flag register value, CS value of the return address and IP value of the return address are pushed on to the stack.

- IP value is loaded from the contents of word location X × 4
- CS is loaded from the contents of the next word location.
- Interrupt flag and trap flag is reset to 0

SOFTWARE INTERRUPTS:

- Some instructions are inserted at the desired position into the program to create interrupts. These interrupt instructions can be used to test the working of various interrupt handlers. **It includes: –**

INT- Interrupt instruction with type number

- It is 2-byte instruction. First byte provides the op-code and the second byte provides the interrupt type number. There are 256 interrupt types under this group.

→Its execution includes the following steps: –

- Flag register value is pushed on to the stack.
- CS value of the return address and IP value of the return address are pushed on to the stack.
- IP is loaded from the contents of the word location 'type number' × 4
- CS is loaded from the contents of the next word location.
- Interrupt Flag and Trap Flag are reset to 0
- The starting address for type0 interrupt is 000000H, for type1 interrupt is 00004H similarly for type2 is 00008H and ……so on. The first five pointers are dedicated interrupt pointers. i.e. –
- **TYPE 0** interrupt represents division by zero situation.
- **TYPE 1** interrupt represents single-step execution during the debugging of a program.
- **TYPE 2** interrupt represents non-maskable NMI interrupt.
- **TYPE 3** interrupt represents break-point interrupt.
- **TYPE 4** interrupt represents overflow interrupt.

- The interrupts from Type 5 to Type 31 are reserved for other advanced microprocessors, and interrupts from 32 to Type 255 are available for hardware and software interrupts.

INT 3-Break Point Interrupt Instruction

- It is a 1-byte instruction having op-code is CCH. These instructions are inserted into the program so that when the processor reaches there, then it

stops the normal execution of program and follows the break-point procedure.

→Its execution includes the following steps: –

- Flag register value is pushed on to the stack.
- CS value of the return address and IP value of the return address are pushed on to the stack.
- IP is loaded from the contents of the word location 3×4 = 0000CH
- CS is loaded from the contents of the next word location.
- Interrupt Flag and Trap Flag are reset to 0

INTO - Interrupt on overflow instruction

- It is a 1-byte instruction and their mnemonic **INTO**. The op-code for this instruction is CEH. As the name suggests it is a conditional interrupt instruction, i.e. it is active only when the overflow flag is set to 1 and branches to the interrupt handler whose interrupt type number is 4. If the overflow flag is reset then, the execution continues to the next instruction.

→Its execution includes the following steps: –

- Flag register values are pushed on to the stack.
- CS value of the return address and IP value of the return address are pushed on to the stack.
- IP is loaded from the contents of word location 4×4 = 00010H
- CS is loaded from the contents of the next word location.
- Interrupt flag and Trap flag are reset to 0

5.11 ADDRESSING MODES:

The way of specifying data to be operated by an instruction is known as **addressing modes**. This specifies that the given data is an immediate data or an address. It also specifies whether the given operand is register or register pair.

1. Immediate addressing mode:

- The addressing mode in which the data operand is a part of the instruction itself is known as immediate addressing mode.
- **Example**

2. Register mode:

- In this type of addressing mode both the operands are registers.

Or

- It means that the register is the source of an operand for an instruction.

- **Example:**

✓ MOV CX, AX ; copies the contents of the 16-bit AX register into the 16-bit CX register

✓ ADD BX, AX

3. Displacement or direct mode:

- In this type of addressing mode the effective address is directly given in the instruction as displacement.

- **Example:**

✓ MOV AX, [DISP]

✓ MOV AX, [0500]

4. Register indirect addressing mode:

- This addressing mode allows data to be addressed at any memory location through an offset address held in any of the following registers: BP, BX, DI & SI.

- **Example**

✓ MOV AX, [BX] ; Suppose the register BX contains 4895H, then the contents 4895H are moved to AX

✓ ADD CX, [BX]

✓ ADD AL, [BX]

5. Based addressing mode:

- In this addressing mode, the offset address of the operand is given by the sum of contents of the BX/BP registers and 8-bit/16-bit displacement.

- **Example:**

✓ MOV DX,[BX+04]

✓ ADD CL, [BX+08]

6. Indexed addressing mode:

- In this addressing mode, the operands offset address is found by adding the contents of SI (Index register) or DI (displacement) register and 8-bit/16-bit displacements.

Example:

✓ MOV BX, [SI+16]

✓ ADD AL, [DI+16]

7. Based-index addressing mode:

- In this addressing mode, the offset address of the operand is computed by summing the base register (BX or BP) to the contents of an Index register (SI or DI).

- Offset= [BX or BP]+[SI or DI]
- BX is used as a base register for data segment, and BP is used as a base register for stack segment.
- **Example:**

✓ ADD AX, [BX+SI]
✓ MOV CX,[BX+SI]
✓ MOV AX,[AX+DI]

8. Based indexed with displacement mode:

- In this type of addressing mode the effective address is the sum of index register, base register and displacement.
- Offset= [BX+BP] + [SI or DI] +8-bit or 16-bit displacement.
- **Example:**

✓ MOV AX, [BX+SI+05]→ an example of 8-bit displacement.
✓ MOV AX, [BX+SI+1235H]→ an example of 16-bit displacement.
✓ MOV AL, [SI+BP+2000]

5.12 INSTRUNCTION SET:

The 8086 instructions are categorized into the following main types

- Data transfer instructions
- Arithmetic instructions
- Program control transfer instructions
- Machine control instructions
- Shift/rotate instructions
- Flag manipulation instructions
- String instructions

1. **<u>DATA COPY /TRANSFER INSTRUCTIONS:</u>**

- These type 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 instruction

- It is a general purpose instruction to transfer byte or word from register to register, memory to register, register to memory or with immediate addressing
- MOV destination, source
- Here the source and destination needs to be of the same size that is both 8-bit and both 16-bit.
- MOV instruction does not affect any flags.

MOV BX, 00F2H ;	load the immediate number 00F2H in BX Register
MOV CL, [2000H] ;	Copy the 8 bit content of the memory Location, at a displacement of 2000H from data segment base to the CL register
MOV [589H], BX ;	Copy the 16 bit content of BX register on to the memory location, which at a displacement of 589H from the data segment base.
MOV DS, CX ;	Move the content of CX to DS

PUSH instruction:

- The PUSH instruction decrements the stack pointer by two and copies the word from source to the location where stack pointer now points. Here the source must of word size data. Source can be a general purpose register, segment register or a memory location.
- The PUSH instruction first pushes the most significant byte to sp-1, then the least significant to the sp-2.
- Push instruction does not affect any flags.
- **Example:-**
- PUSH CX ; Decrements SP by 2, copy content of CX to the stack (figure shows execution of this instruction)
- PUSH DS ; Decrement SP by 2 and copy DS to stack

POP instruction:

- The POP instruction copies a word from the stack location pointed by the stack pointer to the destination. The destination can be a General purpose register, a segment register or a memory location. Here after the content is copied the stack pointer is automatically incremented by two.
- The execution pattern is similar to that of the PUSH instruction.
- **Example:**
- POP CX; Copy a word from the top of the stack to CX and increment SP by 2.

IN & OUT instructions

The IN instruction will copy data from a port to the accumulator. If 8 bit is read the data will go to AL and if 16 bit then to AX. Similarly OUT instruction is used to

copy data from accumulator to an output port.

Both IN and OUT instructions can be done using direct and indirect addressing modes.

- **Example:**

• IN AL, 0F8H	;	Copy a byte from the port 0F8H to AL
• MOV DX, 30F8H	;	Copy port address in DX
• IN AL, DX	;	Move 8 bit data from 30F8H port
• IN AX, DX	;	Move 16 bit data from 30F8H port
• OUT 047H, AL	;	Copy contents of AL to 8 bit port 047H
• MOV DX, 330F8H	;	Copy port address in DX
• OUT DX, AL	;	Move 8 bit data to the 30F8H port
• OUT DX, AX	;	Move 16 bit data to the 30F8H port

XCHG instruction

- The XCHG instruction exchanges contents of the destination and source. Here destination and source can be register and register or register and memory location, but XCHG cannot interchange the value of 2 memory locations.
- XCHG Destination, Source
  - **Example:**

• XCHG BX, CX	;	exchange word in CX with the word in BX
	;	exchange byte in CL with the byte in AL
	;	Here physical address, which is DS +SUM+ [BX]. The content at physical address and the content of AX are interchanged

2. **Arithmetic and Logical instructions:**
   All the instructions performing arithmetic, logical, increment, decrement, compare and ASCII instructions belong to this category.

ADD instruction:

- Add instruction is used to add the current contents of destination with that of source and store the result in destination. Here we can use register and/or memory locations.
- AF, CF, OF, PF, SF, and ZF flags are affected.

- ADD Destination, Source
- **Example:**
    - ADD AL, 0FH ; Add the immediate content, 0FH to the content of AL and store the result in AL
    - ADD AX, BX ; AX <= AX+BX
    - ADD AX,0100H – IMMEDIATE
    - ADD AX,BX – REGISTER
    - ADD AX,[SI] – REGISTER INDIRECT OR INDEXED
    - ADD AX, [5000H] – DIRECT
    - ADD [5000H], 0100H – IMMEDIATE
    - ADD 0100H – DESTINATION AX (IMPLICT)

ADC: ADD WITH CARRY

This instruction performs the same operation as ADD instruction, but adds the carry flag bit (which may be set as a result of the previous calculation) to the result. All the condition code flags are affected by this instruction. The examples of this instruction along with the modes are as follows:

Example:

- ADC AX,BX – REGISTER
- ADC AX,[SI] – REGISTER INDIRECT OR INDEXED
- ADC AX, [5000H] – DIRECT
- ADC [5000H], 0100H – IMMEDIATE
- ADC 0100H – IMMEDIATE (AX IMPLICT)

SUB instruction:

- SUB instruction is used to subtract the current contents of destination with that of source and store the result in destination. Here we can use register and/or memory locations. AF, CF, OF, PF, SF, and ZF flags are affected
- SUB Destination, Source

- **Example:**

    - SUB AL, 0FH ; subtract the immediate content, 0FH from the content of AL and store the result in AL
    - SUB AX, BX ; AX <= AX-BX
    - SUB AX,0100H – **IMMEDIATE (DESTINATION AX)**
    - SUB AX,BX – **REGISTER**
    - SUB AX,[5000H] – **DIRECT**
    - SUB [5000H], 0100H – **IMMEDIATE**

SBB: SUBTRACT WITH BORROW:

- To subtract with borrow instruction subtracts the source operand and the borrow flag (CF) which may reflect the result of the previous calculations, from the destination operand. Subtraction with borrow, here means subtracting 1 from the subtraction obtained by SUB, if carry (borrow) flag is set.

- The result is stored in the destination operand. All the flags are affected (condition code) by this instruction. The examples of this instruction are as follows:

- **Example:**

- SBB AX, 0100H – **IMMEDIATE (DESTINATION AX)**
- SBB AX, BX – **REGISTER**
- SBB AX,[5000H] – **DIRECT**
- SBB [5000H], 0100H – **IMMEDIATE**

CMP: COMPARE:

- The instruction compares the source operand, which may be a register or an immediate data or a memory location, with a destination operand that may 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. The flags are affected depending upon the result of the subtraction.
- If both of the operands are equal, zero flag is set. If the source operand is greater than the destination operand, carry flag is set or else, carry flag is reset. The examples of this instruction are as follows:

- **Example:**
- CMP BX, 0100H – IMMEDIATE
- CMP AX, 0100H – IMMEDIATE
- CMP [5000H], 0100H – DIRECT
- CMP BX,[SI] – REGISTER INDIRECT OR INDEXED

- CMP BX, CX – REGISTER

INC & DEC instructions:

1. INC and DEC instructions are used to increment and decrement the content of the specified destination by one. AF, CF, OF, PF, SF, and ZF flags are affected.
2. **Example:**

INC AL	;	AL←AL + 1
INC AX	;	AX←AX + 1

DEC AL	;	AL← AL – 1
DEC AX	;	AX←AX – 1

AND instruction:

- This instruction logically ANDs each bit of the source byte/word with the corresponding bit in the destination and stores the result in destination. The source can be an immediate number, register or memory location, register can be a register or memory location.

- The CF and OF flags are both made zero, PF, ZF, SF are affected by the operation and AF is undefined.
- AND Destination, Source

- **Example:**

- **AND BL, AL;** suppose BL=1000 0110 and AL = 1100 1010 then after the operation BL would be BL= 1000 0010.
- **AND CX, AX;** CX ← CX AND AX
- **AND CL, 08;** CL← CL AND (0000 1000)

OR instruction:

- This instruction logically ORs each bit of the source byte/word with the corresponding bit in the destination and stores the result in destination. The source can be an immediate number, register or memory location, register can be a register or memory location.

- The CF and OF flags are both made zero, PF, ZF, SF are affected by the operation and AF is undefined.

- OR Destination, Source
- **Example:**
- **OR BL, AL** ; suppose BL=1000 0110 and AL = 1100 1010 then after the operation BL would be BL= 1100 1110.
- OR CX, AX ; CX←AX AND AX
- OR CL, 08 ; CL←CL AND (0000 1000)

NOT instruction:

- The NOT instruction complements (inverts) the contents of an operand register or a memory location, bit by bit. The examples are as follows:
  1. **Example:**
  2. NOT AX (BEFORE AX= $(1011)_2$= (B) $_{16}$ AFTER EXECUTION AX= $(0100)_2$= $(4)_{16}$).
  3. NOT [5000H]

XOR instruction:

- The XOR operation is again carried out in a similar way to the AND and OR

operation. The constraints on the operands are also similar. The XOR operation gives a high output, when the 2 input bits are dissimilar. Otherwise, the output is zero. The example instructions are as follows:

- **Example:**

- XOR AX, 0098H
- XOR AX, BX
- XOR AX, [5000H]

3. Shift / Rotate Instructions:

1) Shift instructions move the binary data to the left or right by shifting them within the register or memory location. They also can perform multiplication of powers of 2^{+n} and division of powers of 2^{-n}.

2) There are two type of shifts logical shifting and arithmetic shifting, later is used with signed numbers while former with unsigned.

SHL/SAL instruction:

- Both the instruction shifts each bit to left, and places the MSB in CF and LSB is made 0. The destination can be of byte size or of word size, also it can be a register or a memory location. Number of shifts is indicated by the count.

- All flags are affected.
- SAL/SHL destination, count

→Example:

```
MOV BL, B7H     ;     BL is made B7H
SAL BL, 1       ;     Shift the content of BL register one place to
                      left.
```

Before Execution

CY	B7	B6	B5	B4	B3	B2	B1	B0
0	1	0	1	1	0	1	1	1

After Execution,

CY	B7	B6	B5	B4	B3	B2	B1	B0
1	0	1	1	0	1	1	1	0

SHR instruction:

- This instruction shifts each bit in the specified destination to the right and 0

is stored in the MSB position. The LSB is shifted into the carry flag. The destination can be of byte size or of word size, also it can be a register or a memory location. Number of shifts is indicated by the count.

- All flags are affected
- **Before execution,**

B7	B6	B5	B4	B3	B2	B1	B0
CY 1	0	1	1	0	1	1	1 0

- **After execution,**

B7	B6	B5	B4	B3	B 2	B 1	B0	CY
0	1	0	1	1	0	1	1	1

ROL instruction:

- This instruction rotates all the bits in a specified byte or word to the left some number of bit positions. MSB is placed as a new LSB and a new CF. The destination can be of byte size or of word size, also it can be a register or a memory location. Number of shifts is indicated by the count.
- All flags are affected
- ROL destination, count
  →**Example:**

```
MOV BL, B7H    ;    BL is made B7H

ROL BL, 1      ;    rotates the content of BL register one place to the left.
```

Before Execution:

CY	B7	B6	B5	B4	B3	B2	B1	B0
0	1	0	1	1	0	1	1	1

After the execution,

CY	B7	B6	B5	B4	B3	B2	B1	B0
1	0	1	1	0	1	1	1	1

ROR instruction:

- This instruction rotates all the bits in a specified byte or word to the right

some number of bit positions. LSB is placed as a new MSB and a new CF. The destination can be of byte size or of word size, also it can be a register or a memory location. Number of shifts is indicated by the count.

- All flags are affected.
- ROR destination, count
- **Example:**

```
MOV BL, B7H     ;       BL is made B7H

ROR BL, 1       ;       shift the content of BL register one place to the right.
```

- **Before execution,**

B7	B6	B5	B4	B3	B2	B 1	B0	CY
1	0	1	1	0	1	1	1	0

- **After execution,**

B7	B6	B5	B4	B3	B2	B 1	B0	CY
1	1	0	1	1	0	1	1	1

RCR instruction

- This instruction rotates all the bits in a specified byte or word to the right some number of bit positions along with the carry flag. LSB is placed in a new CF and previous carry is placed in the new MSB. The destination can be of byte size or of word size, also it can be a register or a memory location. Number of shifts is indicated by the count.
- All flags are affected
- **General Format:** RCR destination, count
- **Example:**

```
MOV BL, B7H     ;       BL is made B7H

RCR BL, 1       ;       shift the content of BL register one place to the
                                                                right.
```

- **Before execution,**

B7	B6	B5	B4	B3	B2	B 1
B0	CY	1	0	1	1	0	1	1
1	0							

- **After execution,**

B7	B6	B5	B4	B3	B2	B 1

B0 CY 0 1 0 1 1 0 1
1 1

4. PROGRAM CONTROL TRANSFER INSTRUCTIONS:

- These instructions transfer control of execution to the specified address. All the call, jump, interrupt and return instruction belong to this class.
- There are 2 types of such instructions.

1. Unconditional transfer instructions – CALL, RET, JMP
2. Conditional transfer instructions – J condition

CALL instruction:

- The CALL instruction is used to transfer execution to a subprogram or procedure. There are two types of CALL instructions, near and far.
- A **near CALL** is a call to a procedure which is in the same code segment as the CALL instruction. 8086 when encountered a near call, it decrements the SP by 2 and copies the offset of the next instruction after the CALL on the stack. It loads the IP with the offset of the procedure then to start the execution of the procedure.
- A **far CALL** is the call to a procedure residing in a different segment. Here value of CS and offset of the next instruction both are backed up in the stack. And then branches to the procedure by changing the content of CS with the segment base containing procedure and IP with the offset of the first instruction of the procedure.
- **Example:**

Near call

CALL PRO ; PRO is the name of the procedure

CALL CX ; Here CX contains the offset of the first instruction
of the procedure, that is replaces the content
IP with the content of CX

Far call

CALL DWORD PTR [8X]; New values for CS and IP are fetched from four Memory locations in the DS. The new value for CS is fetched from [8X] and [8X+1], the new IP is fetched from [8X+2] and [8X+3].

RET instruction:

- RET instruction will return execution from a procedure to the next instruction after the CALL instruction in the calling program. If it was a near call, then IP is replaced with the value at the top of the stack, if it had been a far call, then another POP of the stack is required. This second popped data from the stack is

put in the CS, thus resuming the execution of the calling program.

- RET instruction does not affect any flags.

JMP INSTRUCTION:

- This is also called as unconditional jump instruction, because the processor jumps to the specified location rather than the instruction after the JMP instruction. Jumps can be **short jumps** when the target address is in the same segment as the JMP instruction or **far jumps** when it is in a different segment.

Conditional Jump (J cond)

- Conditional jumps are always short jumps in 8086. Here jump is done only if the condition specified is true/false. If the condition is not satisfied, then the execution proceeds in the normal way.

Iteration control instructions:

- These instructions are used to execute a series of instructions some number of times. The number is specified in the CX register, which will be automatically decremented in course of iteration. But here the destination address for the jump must be in the range of -128 to 127 bytes.

Example:

LOOP : loop through the set of instructions until CX is 0
LOOPE/LOOPZ : here the set of instructions are repeated until CX=0 or ZF=0 LOOPNE/LOOPNZ: here repeated until CX=0 or ZF=1

5. MACHINE CONTROL INSTRUCTIONS:

- These instructions control the machine status. NOP, HLT, WAIT and LOCK instructions belong to this class.

HLT instruction

- The HLT instruction will cause the 8086 microprocessor to fetching and executing instructions.
- The 8086 will enter a halt state. The processor gets out of this Halt signal upon an interrupt signal in INTR pin/NMI pin or a reset signal on RESET input

WAIT instruction

- When this instruction is executed, the 8086 enters into an idle state. This idle state is continued till a high is received on the TEST input pin or a valid interrupt signal is received. Wait affects no flags. It generally is used to synchronize the 8086 with a peripheral device(s).

ESC instruction

- This instruction is used to pass instruction to a coprocessor like 8087. There is a 6 bit instruction for the coprocessor embedded in the ESC instruction. In most cases the 8086 treats ESC and a NOP, but in some cases the 8086 will access data items in memory for the coprocessor

LOCK instruction

- In multiprocessor environments, the different microprocessors share a system bus, which is needed to access external devices like disks. LOCK Instruction is given as prefix in the case when a processor needs exclusive access of the system bus for a particular instruction.
- It affects no flags.
- **LOCK XCHG SEMAPHORE, AL** : The XCHG instruction requires two Bus accesses.
- The lock prefix prevents another processor from taking control of the system bus between the 2 accesses

NOP instruction

- At the end of NOP instruction, no operation is done other than the fetching and decoding of the instruction. It takes 3 clock cycles. NOP is used to fill in time delays or to provide space for instructions while trouble shooting. NOP affects no flags.

6. FLAG MANIPULATION INSTRUCTIONS:

- All the instructions which directly affect the flag register come under this group of instructions. Instructions like CLD, STD, CLI, STI etc.., belong to this category of instructions.

- **STC instruction**
  This instruction sets the carry flag. It does not affect any other flag.

- **CLC instruction**
  This instruction resets the carry flag to zero. CLC does not affect any other flag.

- **CMC instruction**
  This instruction complements the carry flag. CMC does not affect any other flag.

- **STD instruction**
  This instruction is used to set the direction flag to one so that SI and/or DI can be decremented automatically after execution of string instruction. STD does not affect any other flag.

- **CLD instruction**
  This instruction is used to reset the direction flag to zero so that SI and/or DI can be incremented automatically after execution of string instruction. CLD does not affect any other flag.

- **STI instruction**
  This instruction sets the interrupt flag to 1. This enables INTR interrupt of the 8086. STI does not affect any other flag.

- **CLI instruction**
  This instruction resets the interrupt flag to 0. Due to this the 8086 will not respond to an interrupt signal on its INTR input. CLI does not affect any other flag.

7. STRING MANIPULATION INSTRUCTIONS:

These instructions involve various string manipulation operations like Load, move, scan, compare, store etc.

- **MOVS/MOVSB/MOVSW**
  - These instructions copy a word or byte from a location in the data segment to a location in the extra segment. The offset of the source is in SI and that of destination is in DI. For multiple word/byte transfers the count is stored in the CX register.
  - When direction flag is 0, SI and DI are incremented and when it is 1, SI and DI are decremented.
  - MOVS affect no flags. MOVSB is used for byte sized movements while MOVSW is for word sized.

Example:

```
CLD             ;       clear the direction flag to auto increment SI
andDI MOV AX, 0000H;

MOV DS, AX      ;       initialize data segment
register to 0 MOV ES, AX  ;       initialize extra
segment register to 0 MOV SI, 2000H  ;       Load the
offset of the string1 in SI MOV DI, 2400H  ;  Load the
offset of the string2 in DI MOV CX, 04H      ;
                load length of the string in CX
```

REP MOVSB ; decrement CX and MOVSB until CX will be 0

- **REP/REPE/REP2/REPNE/REPNZ**
  - REP is used with string instruction; it repeats an instruction until the specified condition becomes false.

Example:	**Comments**
REP	CX=0
REPE/REPZ	CX=0 OR ZF=0
REPNE/REPNZ	CX=0 OR ZF=1

- **LODS/LODSB/LODSW**
  - This instruction copies a byte from a string location pointed to by SI to AL or a word from a string location pointed to by SI to AX.LODS does not affect any flags. LODSB copies byte and LODSW copies word.

Example:

```
CLD                    ; clear direction flag to auto increment SI MOVSI,
OFFSET S_STRING        ; point SI at string
LODS S_STRING          ;
```

- **STOS/STOSB/STOSW**
  - The STOS instruction is used to store a byte/word contained in AL/AX to the offset contained in the DI register. STOS does not affect any flags. After copying the content DI is automatically incremented or decremented, based on the value of direction flag.

Example:

- MOV DL, OFFSET D_STRING; assign DI with destination address.
- STOS D_STRING ; assembler uses string name to determine byte or Word, if byte then AL is used and if of word size, AX is used.

5. CMPS/CMPSB/CMPSW

- CMPS is used to compare the strings, byte wise or word wise. The comparison is affected by subtraction of content pointed by DI from that pointed by SI. The AF, CF, OF, PF, SF and ZF flags are affected by this instruction, but neither operand is affected.

Example:		Comments
MOV SI, OFFSET STRING_A	;	Point first string
MOV DI, OFFSET STRING_B	;	Point second string
MOV CX, 0AH	;	Set the counter as 0AH
CLD	;	Clear direction flag to auto increment
REPE CMPSB	;	Repeatedly compare till unequal or counter =0

5.13 ASSEMBLER DIRECTIVES AND OPERATOR:

- There are some instructions in the assembly language program which are not a part of processor instruction set. These instructions are instructions to the assembler, linker and loader. These are referred to as pseudo-operations or as assembler directives. The assembler directives enable us to control the way in which a program assembles and lists. They act during the assembly of a program and do not generate any executable machine code.
- There are many specialized assembler directives. Let us see the commonly used assembler directive in 8086 assembly language programming.

ASSUME:

- It is used to tell the name of the logical segment the assembler to use for a specified segment.
- E.g.: ASSUME CS: CODE tells that the instructions for a program are in a logical segment named CODE.

DB -Define Byte:

- The DB directive is used to reserve byte or bytes of memory locations in the available memory. While preparing the EXE file, this directive directs the assembler to allocate the specified number of memory bytes to the said data type that may be a constant, variable, string, etc. Another option of this directive also initializes the reserved memory bytes with the ASCII codes of the characters specified as a string. The following examples show how the DB directive is used for different purposes.
  - ✓ **RANKS DB 01H, 02H, 03H, 04H**
    This statement directs the assembler to reserve four memory locations for a list named RANKS and initialize them with the above specified four values.

- ✓ **MESSAGE DB „GOOD MORNING"**
  This makes the assembler reserve the number of bytes of memory equal to the number of characters in the string named MESSAGE and initializes those locations by the ASCII equivalent of these characters.

- ✓ **VALUE DB 50H**
  This statement directs the assembler to reserve 50H memory bytes and leave them uninitialized for the variable named VALUE.

DD:

- Define Double word - used to declare a double word type variable or to reserve memory locations that can be accessed as double word.

  E.g.: ARRAY _POINTER DD 25629261H declares a double word named ARRAY_POINTER.

DQ -Define Quad word:

- This directive is used to direct the assembler to reserve 4 words (8 bytes) of memory for the specified variable and may initialize it with the specified values.

DT -Define Ten Bytes:

- The DT directive directs the assembler to define the specified variable requiring 10-bytes for its storage and initialize the 10-bytes with the specified values. The directive may be used in case of variables facing heavy numerical calculations, generally processed by numerical processors.

DW -Define Word:

- The DW directives serves the same purposes as the DB directive, but it now makes the assembler reserve the number of memory words (16-bit) instead of bytes. Some examples are given to explain this directive.

  - ✓ **WORDS DW 1234H, 4567H, 78ABH, 045CH**
    This makes the assembler reserve four words in memory (8 bytes), and initialize the words with the specified values in the statements. During initialization, the lower bytes are stored at the lower memory addresses, while the upper bytes are stored at the higher addresses.

  - ✓ **NUMBER1 DW 1245H**
    This makes the assembler reserve one word in memory.

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 source lines available

later on. Hence, it should be ensured that the END statement should be the last statement in the file and should not appear in between. Also, no useful program statement should lie in the file, after the END statement.

ENDP:

- End Procedure - Used along with the name of the procedure to indicate the end of a procedure.
- E.g.: SQUARE_ROOT PROC: start of procedure
  SQUARE_ROOT ENDP: End of procedure

ENDS-End of Segment:

- This directive marks the end of a logical segment. The logical segments are assigned with the names using the ASSUME directive. The names appear with the ENDS directive as prefixes to mark the end of those particular segments. Whatever are the contents of the segments, they should appear in the program before ENDS. Any statement appearing after ENDS will be neglected from the segment. The structure shown below explains the fact more clearly.

EQU:

- Equate - Used to give a name to some value or symbol. Each time the assembler finds the given name in the program, it will replace the name with the vale.
- E.g.: CORRECTION_FACTOR EQU 03H MOV
  AL, CORRECTION_FACTOR

EVEN:

- Tells the assembler to increment the location counter to the next even address if it is not already at an even address.
- Used because the processor can read even addressed data in one clock cycle

EXTRN:

- Tells the assembler that the names or labels following the directive are in some other assembly module.
- For example if a procedure in a program module assembled at a different time from that which contains the CALL instruction ,this directive is used to tell the assembler that the procedure is external

GLOBAL:

- Can be used in place of a PUBLIC directive or in place of an EXTRN directive.
- It is used to make a symbol defined in one module available to other modules.
- E.g.: GLOBAL DIVISOR makes the variable DIVISOR public so that it can be accessed from other modules.

GROUP:

- Used to tell the assembler to group the logical statements named after the directive into one logical group segment, allowing the contents of all the segments to be accessed from the same group segment base.
- E.g.: SMALL_SYSTEM GROUP CODE, DATA, STACK_SEG

INCLUDE:

- Used to tell the assembler to insert a block of source code from the named file into the current source module.
- This will shorten the source code.

LABEL:

- Used to give a name to the current value in the location counter.
- This directive is followed by a term that specifies the type you want associated with that name.
- E.g: ENTRY_POINT LABEL FAR
- NEXT: MOV AL, BL

NAME:

- Used to give a specific name to each assembly module when programs consisting of several modules are written.
- E.g.: NAME PC_BOARD

OFFSET:

- Used to determine the offset or displacement of a named data item or procedure from the start of the segment which contains it.
- E.g.: MOV BX, OFFSET PRICES

ORG:

- The location counter is set to 0000 when the assembler starts reading a segment. The ORG directive allows setting a desired value at any point in theprogram.
- E.g.: ORG 2000H

PROC:

- Used to identify the start of a procedure.
- E.g.:SMART_DIVIDE PROC FAR identifies the start of a procedure named SMART_DIVIDE and tells the assembler that the procedure is far

PTR:

- Used to assign a specific type to a variable or to a label.
- E.g.: NC BYTE PTR[BX]tells the assembler that we want to increment the byte pointed to by BX

PUBLIC:

- Used to tell the assembler that a specified name or label will be accessed from other modules.

- E.g.: PUBLIC DIVISOR, DIVIDEND makes the two variables DIVISOR and DIVIDEND available to other assembly modules.

SEGMENT:

- Used to indicate the start of a logical segment.
- E.g.: CODE SEGMENT indicates to the assembler the start of a logical segment called CODE

SHORT:

- Used to tell the assembler that only a 1 byte displacement is needed to code a jump instruction.
- E.g.: JMP SHORT NEARBY_LABEL

TYPE:

- Used to tell the assembler to determine the type of a specified variable.
- E.g.: ADD BX, TYPE WORD_ARRAY is used where we want to increment BX to point to the next word in an array of words.

5.14 SIMPLE ASSEMBLY LANGUAGE PROGRAMMING USING 8086 INSTRUCTIONS:

PROGRAM-1: (ADDITION)

EFFECTIVE ADDRESS	MNEMONIC CODES	LABLE	MNEMONICS	OPERANDS	COMMENTS
2000	8B,06,00,17		MOV	AX,[1700]	Move the contents of 1700 in register AX
2004	8B,1E,02,17		MOV	BX, [1702]	Move the contents of 1702 in register BX
2008	01,D8		ADD	AX,BX	Data of AX and BX are added and result stored in AX
200A	CC		INT 3		Interrupt program

PROGRAM-2: (SUBTRACTION)

EFFECTIVE ADDRESS	MNEMONIC CODES	LABLE	MNEMONICS	OPERANDS	COMMENTS
2000	8B,06,00,17		MOV	AX,[1700]	Move the contents of 1700 in register AX
2004	8B,1E,02,17		MOV	BX, [1702]	Move the contents of 1702 in register BX
2008	29,D8		SUB	AX,BX	Data of AX and BX are added and result stored in AX
200A	CC		INT 3		Interrupt program

PROGRAM-3: (MULTIPLICATION)

EFFECTIVE ADDRESS	OPCODES	MNEMONICS	OPERANDS	COMMENTS
1100	BE 00 15	MOV	SI,1500	Load 1500 into SI
1103	AD	LOD	SW	Load the multiplicand value
1104	89 C3	MOV	BX, AX	Load AX value into BX
1106	AD	LOD	SW	Load the multiplier value
1107	F7 E3	MUL	BX	Multiply two data
1109	BF 0 5 15	MOV	DI, 1520	Load 1520 address into DI
110C	89 05	MOV	[DI], AX	Store AX value into DI
110E	47	INC DI		
110F	47	INC	DI	Increment the DI
1110	89 15	MOV	[DI], BX	Store BX value into DI
1112	CC	INT 3		Break point

PROGRAM-4: (DIVISION)

EFFECTIVE ADDRESS	OPCODES	MNEMONICS	OPERANDS	COMMENTS
1100	BA 00 00	MOV	DX, 0000	Clear DX registers
1103	B8 83 00	MOV	AX, 0083	Load the dividend in AX
1106	B9 00 02	MOV	BX, 02	Load the divisor value in BX
1109	F7 F1	DIV	BX	Divide the two data's
110B	BF 20 15	MOV	DI, 1520	Load 1520 address into DI
110E	88 05	MOV	[DI], AL	Load AL value into DI
1110	47	INC	DI	Increment DI

1111	88 25	MOV	[DI], AH	Load AH value into DI
1113	47	INC	DI	Increment DI
1114	89 15	MOV	[DI], DX	Load DX value into DI
1116	CC	INT3		Break point

PROGRAM-5: (LARGEST NUMBER IN DATA ARRAY)

EFFECTIVE ADDRESS	**MNEMONIC CODES**	**LABLE**	**MNEMONICS**	**OPERANDS**	**COMMENTS**
0101	B8, 00, 00		MOV	AX, 0000	;Initial value for comparison
0104	BE, 00, 02		MOV	SI, 0200	;memory address in SI
0107	8B, 0C		MOV	CX, [SI]	;count in CX
0109	46	BACK	INC	SI	;increment SI
010A	46		INC	SI	;increment SI
010B	3B, 04		CMP	AX, [SI]	;compare previous largest number with next number
010D	73, 02		JAE	GO	;Jump if number in AX is greater i.e. CF = 0
010F	8B, 04		MOV	AX, [SI]	;save next larger number in AX
0111	E2, F6	GO	LOOP	BACK	;jump to BACK until CX become zero
0113	A3, 51, 02		MOV	[0251], AX	;store largest number in memory
0116	CC		INT3		;interrupt program

PROGRAM-6: (SMALLEST NUMBER IN DATA ARRAY)

EFFECTIVE ADDRESS	MNEMONICS CODES	LABEL	MNEMONICS	OPERANDS	COMMENTS
0101	B8,FF,FF		MOV	AX,FFFF	Initial value for comparison.
0104	BE,00,02		MOV	SI,0200	Memory address in SI.
0107	8B,0C		MOV	CX,[SI]	Count in CX
0109	46	BACK	INC	SI	Increment SI
010A	46		INC	SI	Increment SI
010B	3B,04		CMP	AX,[SI]	Compare previous smallest with next number
010D	72,02		JB	GO	Jump if number in AX is smaller i.e. CF=1
010F	8B,04		MOV	AX,[SI]	Save next smaller
0111	E2,F6	GO	LOOP	BACK	Jump to back until CX becomes zero.
0113	A3,51,02		MOV	[0251],AX	Store smallest number in memory
0116	CC		INT 3		Interrupt program.

Branch Instructions:

- These instructions transfer control of execution to the specified address. All the call, jump, interrupt and return instruction belong to this class.

Loop instructions:

- These instructions can be used to implement unconditional and conditional loops. The LOOP, LOOP NZ, LOOP Z instructions belong to this category.

Machine control instructions:

- These instructions control the machine status. NOP, HLT, WAIT and LOCK instructions belong to this class.

Flag manipulation instructions:

- All the instructions which directly affect the flag register come under this group of instructions. Instructions like CLD, STD, CLI, STI etc.., belong to this category of instructions.

Shift and Rotate instructions:

- These instructions involve the bit wise shifting or rotation in either direction with or without a count in CX.

String manipulation instructions:

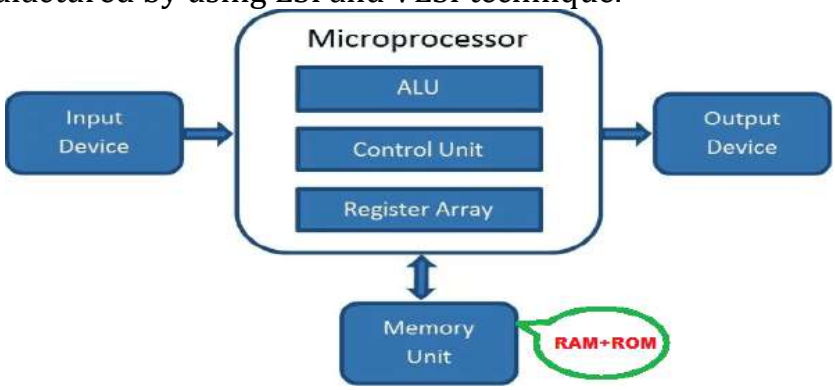
- These instructions involve various string manipulation operations like Load, move, scan, compare, store etc..,

UNIT-6 MICROCONTROLLER (ARCHITECTURE AND PROGRAMMING-8 BIT)

6.1 DISTINGUISH BETWEEN MICROPROCESSOR & MICROCONTROLLER

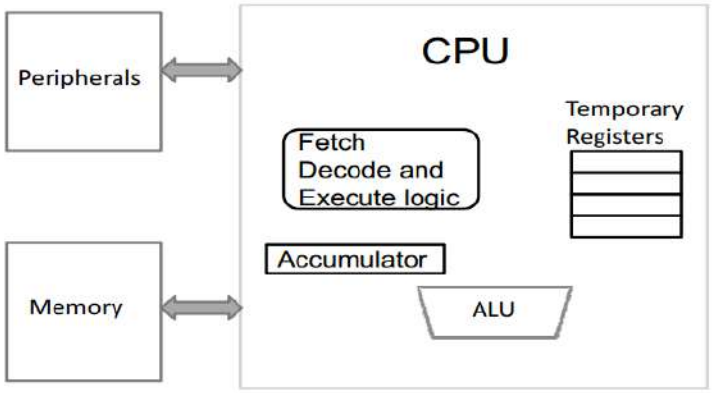
MICROPROCESSOR:

- A Microprocessor is a multipurpose, Programmable clock driven, register based electronic device,
- That read binary instruction from a storage device called memory, accepts binary data as input and processes data according to those instructions and provides results as outputs.
- Microprocessor is clock driven semiconductor device which for is manufactured by using LSI and VLSI technique.



MICROCONTROLLER:

- Microcontroller is like a mini computer with a CPU along with RAM, ROM, serial ports, timers, and IO peripherals all embedded on a single chip.
- It's designed to perform application specific tasks that require a certain degree of control such as a TV remote, LED display panel, smart watches, vehicles, traffic light control, temperature control, etc.
- It's a high-end device with a microprocessor, memory, and input/output ports all on a single chip.
- It's the brains of a computer system which contains enough circuitry to perform specific functions without external memory.
- Since it lacks external components, the power consumption is less which makes it ideal for devices running on batteries.
- Simple speaking, a microcontroller is complete computer system with less external hardware.



DIFFERENCE BETWEEN MICROPROCESSOR AND MICROCONTROLLER:

MICROPROCESSOR	MICROCONTROLLER
Microprocessor contains ALU, General purpose registers, stack pointer, program counter, clock timing circuit, interrupt circuit	Microcontroller contains the circuitry of microprocessor, and in addition it has built in ROM, RAM, I/O Devices, Timers/Counters etc.
It has many instructions to move data between memory and CPU	It has few instructions to move data between memory and CPU
Few bit handling instruction	It has many bit handling instructions
Less number of pins are multifunctional	More number of pins are multifunctional
Single memory map for data and code (program)	Separate memory map for data and code (program)
Access time for memory and IO are more	Less access time for built in memory and IO.
Microprocessor based system requires additional hardware	It requires less additional hardware's
More flexible in the design point of view	Less flexible since the additional circuits which is residing inside the microcontroller is fixed for a particular microcontroller
Large number of instructions with flexible addressing modes	Limited number of instruction with few addressing modes

6.2 8 BIT & 16 BIT MICROCONTROLLER:

8 bit Microcontroller:

- 8 bit microcontroller is type of microcontroller having all traits of microcontroller and its information gadgets are largely 8 bits big.
- 8 bits big means your CPU can use 8 bit information bus or pipe and can entry the similar dimension information by a single machine instruction.
- For every cycle of instruction its fluctuate is zero to 255. It requires 20mA current to work. Intel 8008 was the first model having 8 bit micro-controller.

16 bit Microcontroller:

- 16 bit microcontroller is additional superior than 8 bit microcontroller.
- It is additional right and precise in performing mathematical and technical duties.

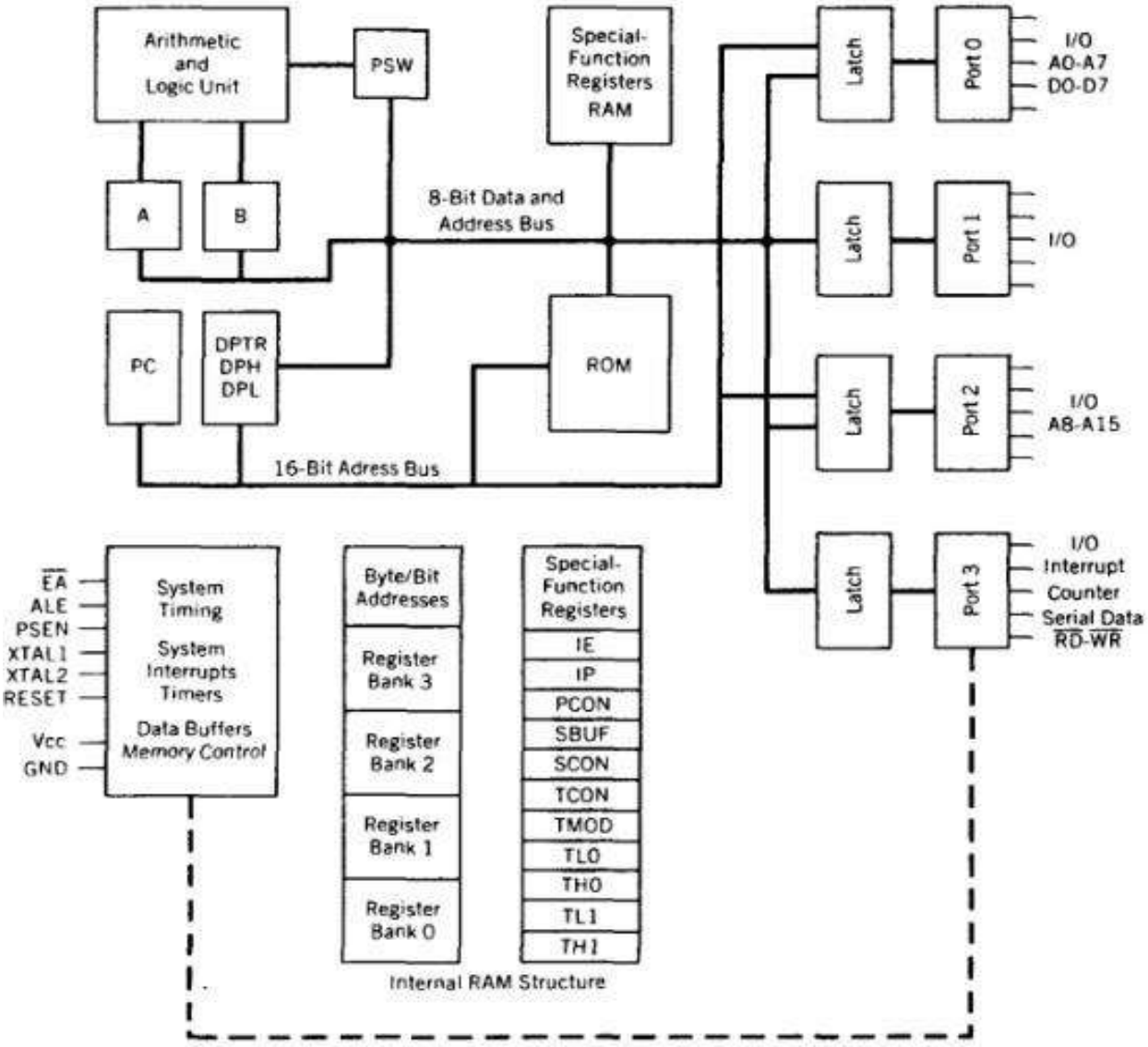
- Unlike 8 bit microcontroller it makes use of 16 bits information bus or pipe for a single instruction.
- For every cycle of instruction its bit fluctuate is extended from zero to 65,535. As 16 bit controller is 2 time better than 8 bit controller, it would probably work on two 16 bit numbers. It requires 10mA current to hold out.

6.3 CISC AND RISC CPU ARCHITECTURES:

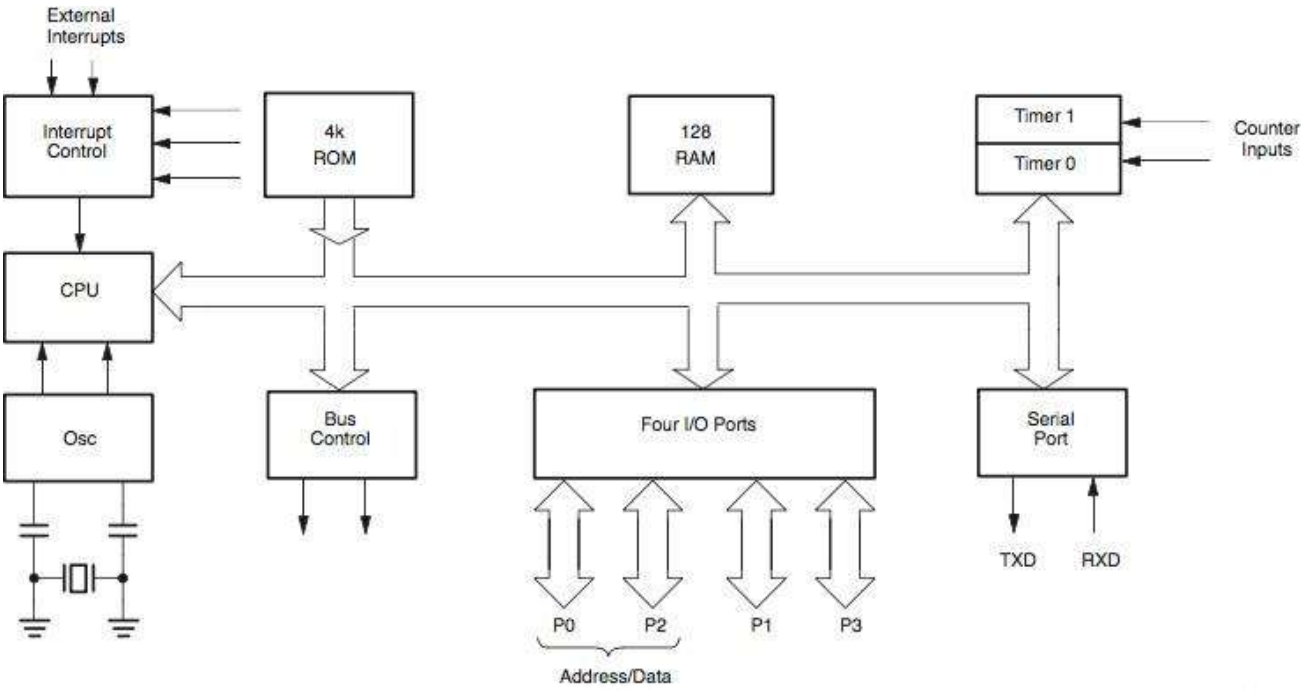
- Microcontrollers with small instruction set are called reduced instruction set computer (RISC) machines and those with complex instruction set are called complex instruction set computer (CISC).
- Intel 8051 is an example of CISC machine whereas microchip PIC 18F87X is an example of RISC machine.

RISC	CISC
Instruction takes one or two cycles	Instruction takes multiple cycles
Only load/store instructions are used to access memory	In additions to load and store instructions, memory access is possible with other instructions also.
Instructions executed by hardware	Instructions executed by the micro program
Fixed format instruction	Variable format instructions
Few addressing modes	Many addressing modes
Few instructions	Complex instruction set
Most of the have multiple register banks	Single register bank
Highly pipelined	Less pipelined
Complexity is in the compiler	Complexity in the microprogram

6.4 ARCHITECTURE OF 8051 MICROCONTROLLER:



Alternate diagram …



✓ **8051** is a microcontroller. This means it has an internal processor, internal memory and an I/O section. The architecture of 8051 is thus divided into three main sections:

- The CPU
- Internal Memory
- I/O components.

CPU:

- 8051 has an 8 bit CPU.
- This is where all 8-bot arithmetic and logic operations are performed.
- It has the following components.

ALU – ARITHMETIC LOGIC UNIT:

- It performs 8-bit arithmetic and logic operations.
- It can also perform some bit operations.

- **Example:**
  ADD A, R0 ; Adds contents of A register and R0 register and stores the result in A register.

ANL A, R0 ; Logically ANDs contents of A register and R0 register and stores the result in A register.

CPL P0.0 ; Complements the value of P0.0 pin.

A – REGISTER (ACCUMULATOR):

- It is an 8-bit register.
- In most arithmetic and logic operations, A register hold the first operand and also gets the result of the operation.
- Moreover, it is the only register to be used for data transfers to and from external memory.

- **Example:**

  ADD A, R1 ; Adds contents of A register and R1 register and stores the result in A register.

  MOVX A, @DPTR ; A gets the data from External RAM location pointed by DPTR

B – REGISTER:

- It is an 8-bit register.
- It is dedicated for Multiplication and Division.
- It can also be used in other operations.

- **Example:**

  MUL AB; Multiplies contents of A and B registers. Stores 16-bit result in B and A registers.

  DIV AB; Divides contents of A by those of B. Stores quotient in A and remainder in B.

PC – PROGRAM COUNTER

- It is a 16-bit register.
- It holds address of the next instruction in program memory (ROM).
- PC gets automatically incremented as soon as any instruction is fetched.
- That's what makes the program move ahead in a sequential manner.

- In the case of a branch, a new address is loaded into PC.

DPTR – DATA POINTER

- It is a 16-bit register.
- It holds address data in data memory (RAM).
- DPTR is divided into two registers DPH (higher byte) and DPL (lower byte).
- It is typically used by the programmer to transfer data from External RAM.
- It can also be used as a pointer to a look up table in ROM, using Indexed addressing mode.
- **Example:**

MOVX A, @DPTR ; A gets the data from External RAM location pointed by DPTR

MOVC A, @A+DPTR ; A gets the data from ROM location pointed by A + DPTR

SP – STACK POINTER

- It is an 8-bit register.
- It contains address of the top of stack. The Stack is present in the Internal RAM.
- Internal RAM has 8-bit addresses from 00H… 7FH. Hence SP is an 8-bit register.
- It is affected during Push and Pop operations. During a Push, SP gets incremented.
- During a Pop, SP gets decremented.

PSW – PROGRAM STATUS WORD

- It is an 8-bit register.
- It is also called the "Flag register", as it mainly contains the status flags. These flags indicate status of the current result.
- They are changed by the ALU after every arithmetic or logic operation. The flags can also be changed by the programmer.
- PSW is a bit addressable register.
- Each bit can be individually set or reset by the programmer.

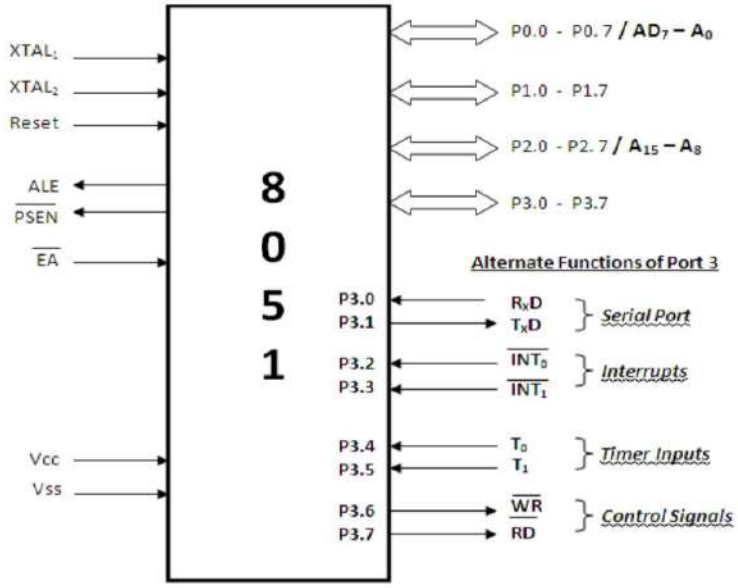
- The bits can be referred to by their bit numbers (**PSW.4**) or by their name (**RS1**).

- **Example:**

  SETB PSW.3 ; Makes PSW.3 ← 1

  CLR PSW.4 ; Makes PSW.4 ← 0

6.5 SIGNAL DESCRIPTION OF 8051:



- 8051 has 40 pins.
  The function of these pins is briefly explained as follows.

XTAL1 & XTAL2:

- These are connected to the crystal oscillator.
- The typical operate in frequency is 12 MHz
- In Serial communication based applications, the operating frequency is chosen to be 11.0592 MHz, in order to derive the standard universal baud rates. This will be discussed in detail in the further chapters.

RESET:

- It is used to reset the 8051 microcontroller. On reset PC becomes 0000H.
- This address is called the reset vector address.
- From here, 8051 executes the BIOS program also called the Booting program or the monitor program.
- It is used to set-up the system and make it ready, to be used by the end-user.

ALE:

- It is used to enable the latching of the address. The address and data buses are multiplexed.
- This is done to reduce the number of pins on the 8051 IC.
- Once out of the chip, address and data have to be separated that is called de-multiplexing.
- This is done by a latch, with the help of ALE signal. ALE is "1" when the bus carries address and "0" when the bus carries data.
- This informs the latch, when the bus is carrying address so that the latch captures only address and not the data.

EA'

- It decides whether the first 4 KB of program memory space (0000H… 0FFFH) will be assigned to internal ROM or External ROM.
- If EA = 0, the External ROM begins from 0000H.
- In this case the Internal ROM is discarded. 8051 now uses only External ROM.
- If EA = 1, the External ROM begins from 1000H.
- In this case the Internal ROM is used. It occupies the space 0000H…0FFFH.
- In modern **FLASH ROM versions**, this pin also acts as **VPP** (12 Volt programming voltage) to write into the FLASH ROM.

PSEN'

- 8051 has a 16-bit address bus (A_{15}-A_0).
  This should allow 8051 to access 64 KB of external Memory as 2^{16} = 64 KB. Interestingly though, 8051 can access 64 KB of External ROM and 64 KB of External RAM, making a total of 128

KB.

- Both have the same address range 0000H to FFFFH.
- This does not lead to any confusion because there are separate control signals for External RAM and External ROM.
- RD and WR are control signals for External RAM.
- PSEN is the READ signal for External ROM.
- It is called Program Status Enable as it allows reading from ROM also known as Program Memory. Having separate control signals for External RAM and External ROM actually allows us to double the size of the external memory to a total of 128 KB from the original 64 KB.

VCC & GND:

- These are power supply pins.
- 8051 works at +5V / 0V power supply.

P0.0-P0.7

- These are 8 pins of Port 0.
- We can perform a byte operation (8-bit) on the whole port 0.
- We can also access every bit of port 0 individually by performing bit operations like set, clear, complement etc.
- The bits are called P0.0… P0.7.
- Additionally, Port 0 also has an alternate function.
- It carries the multiplexed address data lines.
- A0-A7 (the lower 8 bits of address) and D0-D7 (8 bits of data) are multiplexed into AD0-AD7.
- In any operation address and data are not issued simultaneously. First, address is given, then data is transferred. Using a common bus for both, reduces the number of pins.
- To identify if the bus is carrying address or data, we look at the ALE signal. If ALE = 1, the bus carries address,
- If ALE = 0, the bus carries data.

P1.0-P1.7

- These are 8 pins of Port 1.
- We can perform a byte operation (8-bit) on the whole port 1.
- We can also access every bit of port 1 individually by performing bit operations like set, clear, complement etc. on P1.0… P1.7.
- Port 1 also has NO alternate function

P2.0-P2.7

- These are 8 pins of Port 2.
- We can perform a byte operation (8-bit) on the whole port 2.
- We can also access every bit of port 2 individually by performing bit operations like set, clear, complement etc. on P2.0… P2.7.
- Additionally, Port 2 also has an alternate function. It carries the higher order address lines A8-A15.

P3.0-P3.7

- These are 8 pins of Port 3.
- We can perform a byte operation (8-bit) on the whole port 3. We can also access every bit of port 3 individually.
- The bits are called P0.0… P0.7.
- The various pins of Port 3 have a lot of alternate functions.

P3.0 (RXD) and P3.1 (TXD):

- They are used to receive and transmit serial data.
- This forms the serial port of 8051.

P3.2 (INT0) and P3.3 (INT1):

- They are external hardware interrupts of 8051.
- If they occur simultaneously, INTO is by default higher priority.

P3.4 (T0) and P3.5 (T1):

- They are used **timer clock inputs**.
- They provide external clock inputs to Timer 0 and Timer 1.

P3.6 (WR) and P3.7 (RD):

- They are used as **control signals for External RAM**.
- 8051 can access 64 KB External RAM from 0000H to FFFFH.

6.6 MEMORY ORGANISATION-RAM STRUCTURE:

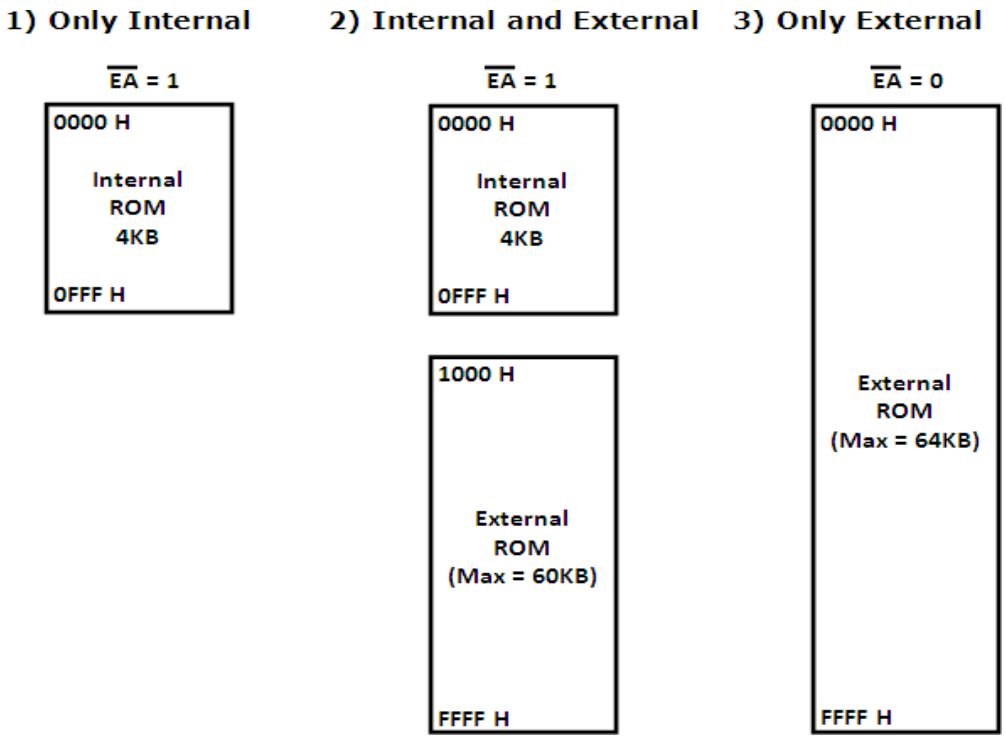
✓ 8051 operates with 4 different memories:

- Internal ROM

- External ROM
- Internal RAM
- External RAM

- Being based on Harvard Model, 8051 stores programs and data in separate memory spaces. Programs are stored in ROM, whereas data is stored in RAM.
- Microcontrollers are used in appliances.
- Washing machines, remote controllers, microwave ovens are some of the
- **EXAMPLES:**
  Here programs are generally permanent in nature and very rarely need to be modified. Moreover, the programs must be retained even after the device is completely switched off. Hence programs are stored in permanent (non-volatile) memory like ROM.
- Data on the other hand is continuously changed at runtime. For example current temperature, cooking time etc. in an oven.
- Such data is not permanent in nature and will certainly be modified in every usage of the device.
- Hence Data is stored in writeable memory like RAM.
- However, sometimes there is permanent data, such as ASCII codes or 7-segment display codes. Such data is stored in ROM, in the form of Look up tables and is accessed using a dedicated addressing mode called Indexed Addressing mode. We will discover this in more depth in further topics.
- We are now going to take a closer look at all four memories.

ROM ORGANIZATION / CODE MEMORY / PROGRAM MEMORY



✓ We can implement ROM in three different ways in 8051.

1. ONLY INTERNAL ROM:

- 8051 has 4 KB internal ROM.
- In many cases this size is sufficient and there is no need for connecting External ROM. Such systems use only Internal ROM of 8051.
- All addresses from 0000H… 0FFFH will be accessed from Internal ROM. Any address beyond that will be invalid.
- In such systems **EA** will be "1" as Internal ROM is being used.

2. INTERNAL AND EXTERNAL ROM:

- 8051 has 4 KB internal ROM.
- In many cases this size is may be insufficient and we may need to add some External ROM. Such systems use a combination of Internal ROM and External ROM.

- The "total" ROM that can be accessed is 64 KB.
- Since we are using the Internal ROM of 4 KB, the maximum amount of External ROM that can be connected is 60 KB.
- All addresses from 0000H… 0FFFH will be accessed from Internal ROM. Addresses 1000H… FFFH will be accessed from External ROM.
- In such systems EA will be "1" as Internal ROM is being used.

3. ONLY EXTERNAL ROM:

- This is the most interesting case.
- Though 8051 has 4 KB of Internal ROM, the user may choose the discard it and connect only External ROM.
- This may happen due to several reasons.
- The program stored in the Internal ROM may have become invalid or outdated, or the system may need to be upgraded etc.
- Such systems use only External ROM, and the Internal ROM is discarded. Here we can connect up to 64 KB of External ROM.
- All addresses from 0000H… FFFFH will be accessed from External ROM. But do keep in mind, that the Internal ROM is still present in 8051.
- We need to clearly indicate to 8051 that the Internal ROM must be ignored and every address from 0000H… FFFFH must be accessed externally. This is indicated by us to 8051 using **EA.**
- By making **EA = 0**, we inform 8051 that the Internal ROM must be discarded and all ROM must be accessed externally.

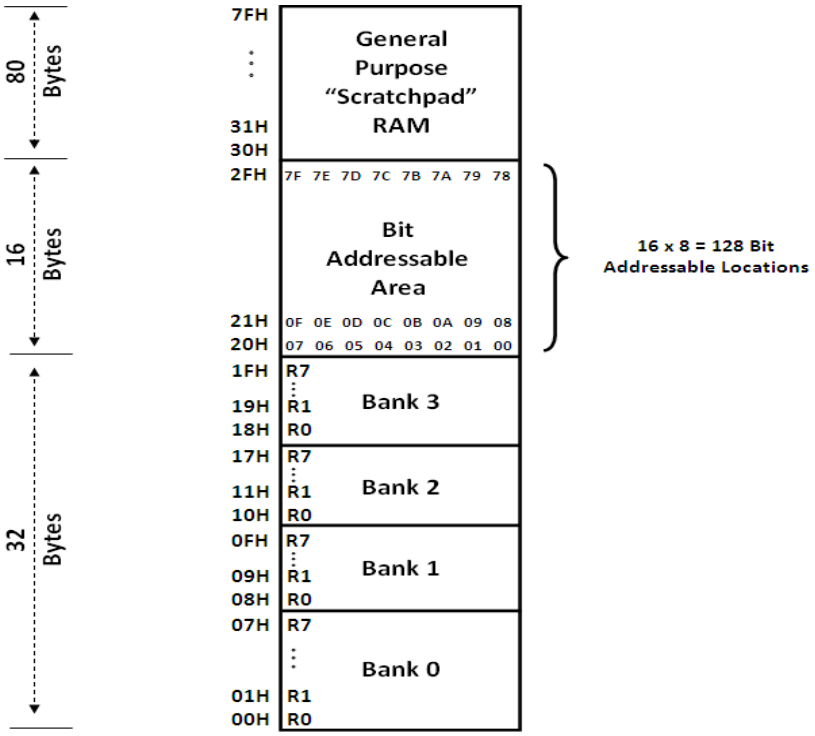
✓ **Use of EA pin of 8051:**

- **The EA pin of 8051 decides whether the Internal ROM will be used or not.**
- If the Internal ROM has to be used we must make **EA** = 1.
- Now 8051 will Access the internal ROM for all addresses from 0000H to 0FFFH and will only access external ROM for addresses 1000H and beyond.

- But if **EA** = 0, then the Internal ROM is completely discarded.
- Now 8051 will access the External ROM for all addresses from 0000H to FFFFH, hence discarding the internal ROM.

- 8051 checks **EA** pin during every ROM operation where the address is 0000H…
- 0FFFH. If **EA = 1,** this location is accessed from internal ROM.
- If **EA = 0,** this location is accessed from external ROM.
- If the address is 1000H or more, 8051 does not check **EA** as this location can only be present in External ROM.

STRUCTURE OF INTERNAL RAM:



- 8051 has a 128 Bytes of internal RAM. These are 128 locations of 1 Byte each.
- The address range is 00H… 7FH.
- This RAM is used for storing data.
- It is divided into three main parts: Register Banks, Bit addressable area and a general purpose area.

REGISTER BANKS:

- The first 32 locations (Bytes) of the Internal RAM from 00H… 1FH, are used by the programmer as general purpose registers.
- Having so many general purpose registers makes programming easier and faster.

- But as a downside, this also vastly increases the number of opcodes (refer my class lectures for detailed understanding of this).
- Hence the 32 registers are divided into 4 banks, each having 8 Registers R0… R7.
- The first 8 locations 00H… 07H are registers R0… R7 of bank 0.
- Similarly locations 08H… 0FH are registers R0… R7 of bank 1 and so on. A register can be addressed using its name, or by its address.
- E.g. Location 00H can be accessed as R0, if Bank 0 is the active bank.
  MOV A, R0; "A" register gets data from register R0.
  It can also be accessed as Location 00H, irrespective of which bank is the active bank.
  MOV A, 00H; "A" register gets data from Location 00H.
- The appropriate bank is selected by the RS1, RS0 bits of PSW. Since PSW is available to the programmer, any Bank can be selected at run-time.
- Bank 0 is selected by default, on reset.

BIT ADDRESSABLE AREA:

- The next 16-bytes of RAM, from 20H… 2FH, is available as Bit Addressable Area.
- We can perform ordinary byte operations on these locations, as well as bit operations.
- As each location has 8-bits, we have a total of →16 X 8 = 128 Addressable Bits.
- These bits can be addressed using their individual address 00H … 7FH. SETB 00H; Will store a "1" on the LSB of location 20H
  CLR 07H; Will store a "0" on the MSB of location 20H

- Normal "BYTE" operations can also be performed at the addresses: 20H … 2FH. MOV 20H, #00H; Will store a "0" on all 8-bits of location 20H.

- Here is something very interesting to know and will also help you understand further topics. The entire internal RAM is of 128 bytes so the address range is 00H… 7FH.
- The bit addressable area has 128 bits so its bit addresses are also 00h… 7FH.

- This means every address 00H… 7FH can have two meanings, it could be a byte address or a bit address.

- This does not lead to any confusion, because the instruction in which we use the address, will clearly indicate whether it is a bit operation or a byte operations.

- SETB, CLR etc. are bit ops whereas ADD, SUB etc. are byte operations.
- SETB 00H; this is a bit operation. It will make Bit location 00H contain a value "1".
- MOV A, 00H; this is a byte operation. A" register will get 8-bit data from byte location 00H.

GENERAL PURPOSE AREA

- The general-purpose area ranges from location 30H … 7FH.
- This is an 80-byte area which can be used for general data storage.

STACK OF 8051:

- Another important element of the Internal RAM is the Stack.
- Stack is a set of memory locations operating in Last in First out (LIFO) manner.
- It is used to store return addresses during ISRs and also used by the programmer to store data during programs.
- In 8051, the Stack can only be present in the Internal RAM.
- This is because, SP which is an 8-bit register, can only contain an 8-bit address and External RAM has 16-bit address. (#Viva)
- On reset SP gets the value 07H.
- Thereafter SP is changed by every PUSH or POP operation in the following manner:

PUSH: **POP:**

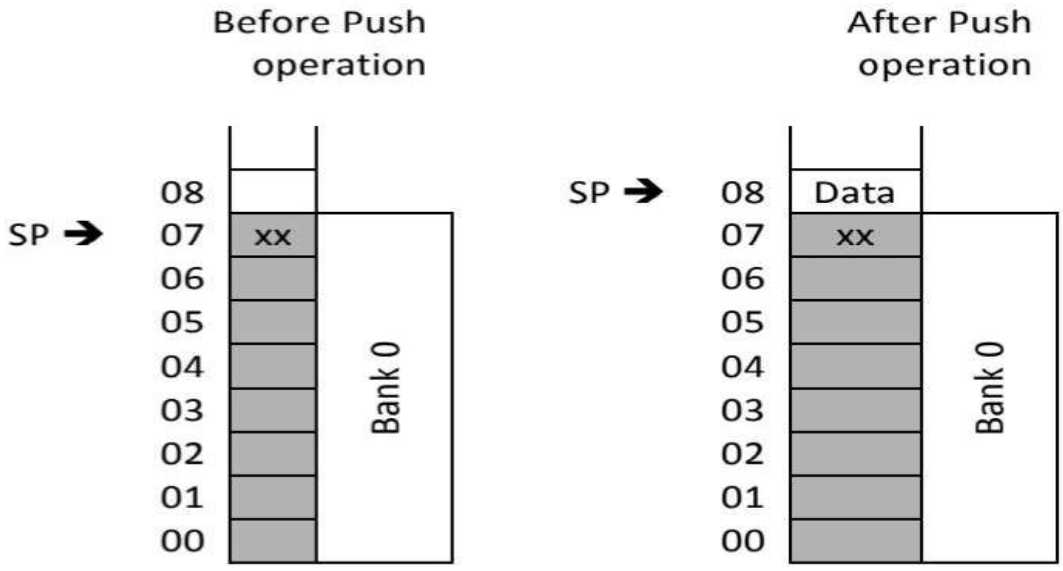
SP → **SP + 1** Data→ [SP]

[SP] → New data **SP→SP – 1**

- The reset value of SP is 07H because, on the first PUSH, SP gets

incremented and then data is pushed on to the stack. This means the very first data will be stored at location 08H.

- This does not affect the default bank (0) and still gives the stack, the maximum space to grow.



- The programmer can relocate the stack to any desired location by simply putting a new value into SP register.

6.7 SFR (SPECIAL FUNCTION REGISTER):

- 8051 has 21, 8-bit Special Function registers.

	NAME	FUNCTION	BYTE ADDRESS	BIT ADDRESS
Used for holding data and status during Programming	A*	Accumulator	0E0H	0E7H...0E0H
	B*	Arithmetic	0F0H	0F7H...0F0H
	PSW*	Program Status Word	0D0H	0D7H...0D0H
Used in instructions to point to memory	SP	Stack Pointer	81H	NA
	DPL	Address External Memory	82H	NA
	DPH	Address External Memory	83H	NA
Used by the respective I/O Ports	P0*	I/O Port latch	80H	87H...80H
	P1*	I/O Port latch	90H	97H...90H
	P2*	I/O Port latch	0A0H	0A7H...0A0H
	P3*	I/O Port latch	0B0H	0B7H...0B0H
Used by the Serial Port	SCON*	Serial Port Control	98H	9FH...98H
	SBUF	Serial Port Data Buffer	99H	NA
Used for Timer Control	TCON*	Timer/Counter Control	88H	8FH...88H
	TMOD	Timer/Counter Mode Control	89H	NA
	TL0	Timer 0 Low Byte	8AH	NA
	TL1	Timer 1 Low Byte	8BH	NA
	TH0	Timer 0 High Byte	8CH	NA
	TH1	Timer 1 High Byte	8DH	NA
Used for Interrupt Control	IE*	Interrupt Enable	0A8H	0AFH...0A8H
	IP*	Interrupt Priority	0B8H	0BFH...0B8H
Used for Power Control	PCON	Power Control	87H	NA

- SFRs are 8-bit registers. Each SFR has its own special function.
- They are placed inside the Microcontroller.
- They are used by the programmer to perform special functions like controlling the timers, the serial port, the I/O ports etc.
- As SFRs are available to the programmer, we will use them in instructions. This causes another problem. SFRs are registers after all, and hence using them would tremendously increase the number of opcodes to reduce the number of opcodes, SFRs are allotted addresses. These addresses must not clash with any other addresses of the existing memories
- Incidentally, the internal RAM is of 128 bytes and uses addresses only from 00H… 7FH. This gives an entire range of addresses from 80H… FFH completely unused and can be freely allotted to the SFRs.
- Hence SFRs are allotted addresses from 80H… FFH.
- It is not a co-incidence that these addresses are free. The Internal RAM was restricted to 128 bytes instead of 256 bytes so that these addresses are free for SFRs.
- To avoid this problem, even the bits of the SFRs are allotted addresses. These are bit addresses, which are different from byte addresses. These bit

addresses must not clash with those of the bit addressable area of the Internal RAM. Amazingly, even the bit addresses in the Internal RAM are 00H… 7FH (again 128 bits), keeping bit addresses 80H… FFH free to be used by the SFR bits.

- So bit addresses 80H… FFH are allotted to the bits of various SFRs.
- Port 0 has a byte address of 80H and its bit addresses are from 80H… 87H.
- A byte operation at address 80H will affect entire Port0.
- **E.g.-**MOV A, P0; this refers to Byte address 80H that's whole Port 0. 12) A bit

Operation at 80H will affect only P0.0.

- E.g. SETB P0.0; this refers to bit address 80H that's Port0.0

6.8 REGISTERS OF 8051 MICROCONTROLLER:

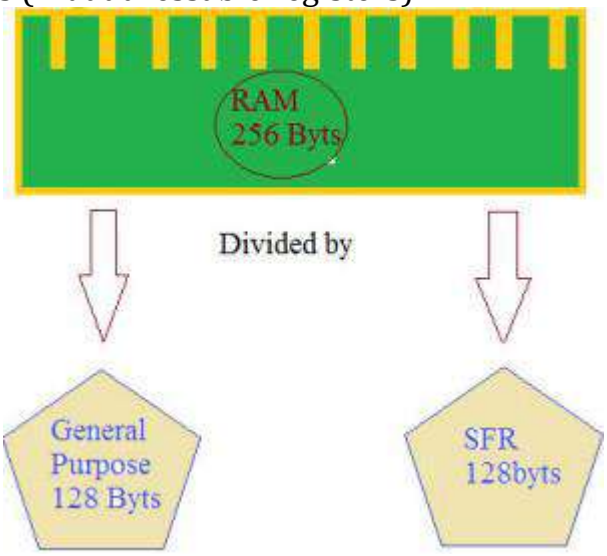
- In the CPU, registers are used to store information temporarily.
- That information could be a byte of data to be processed, or an address pointing to the data to be fetched.
- The vast majority of 8051 registers are 8-bit registers.in the 8051 there is only one data type 8-bits.
- With an 8-bit data type, any data larger than 8-bits must be broken into 8-bit chunks before it is processed.
- Since there are a larger number of registers in the 8051.
- The most widely used registers of the 8051 are A (accumulator), B, R0, R1, R2, R3, R4, R5, R6, R7, DPTR (data pointer) and PC (program counter).
- All of the above registers are 8-bits except DPTR and the program counter (PC).
- The accumulator, register A is used for all arithmetic and logic instructions.

Types of Registers:

The 8051 microcontroller contains mainly two types of registers:

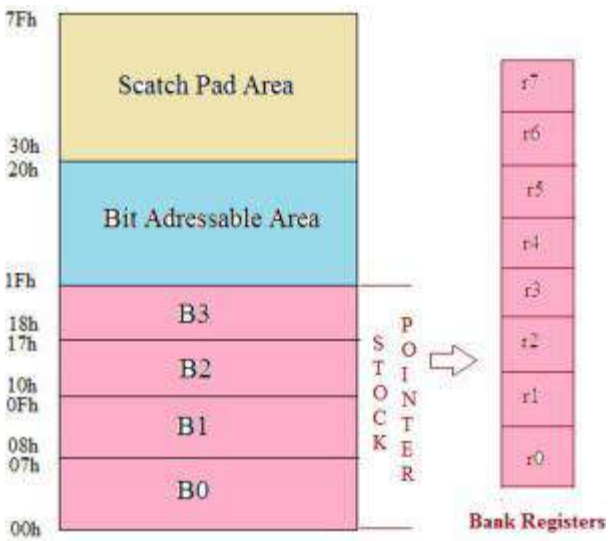
- General purpose registers (Byte addressable registers)

- Special function registers (Bit addressable registers)



- The 8051 microcontroller consists of 256 bytes of RAM memory, which is divided into two ways, such as 128 bytes for general purpose and 128 bytes for special function registers (SFR) memory.
- The memory which is used for general purpose is called as RAM memory, and the memory used for SFR contains all the peripheral related registers like Accumulator, 'B' register, Timers or Counters, and interrupt related registers.

General Purpose Registers:

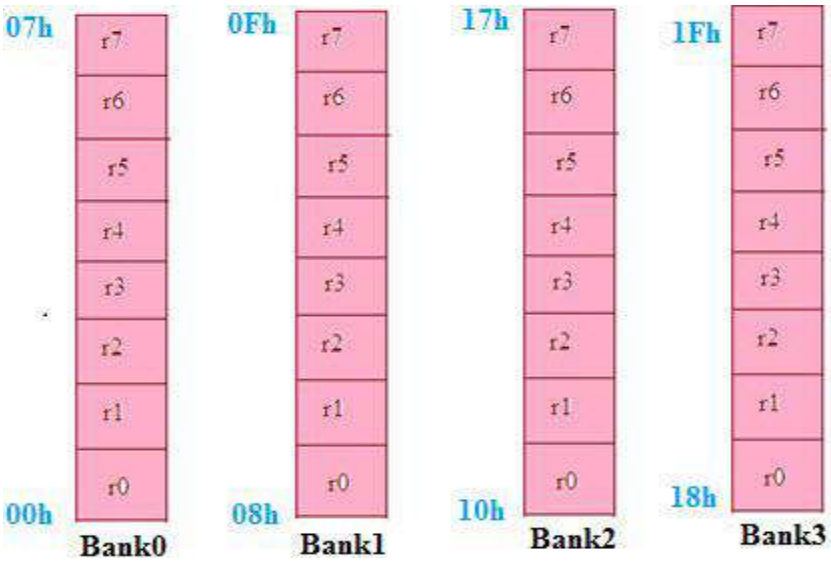


General Purpose Memory

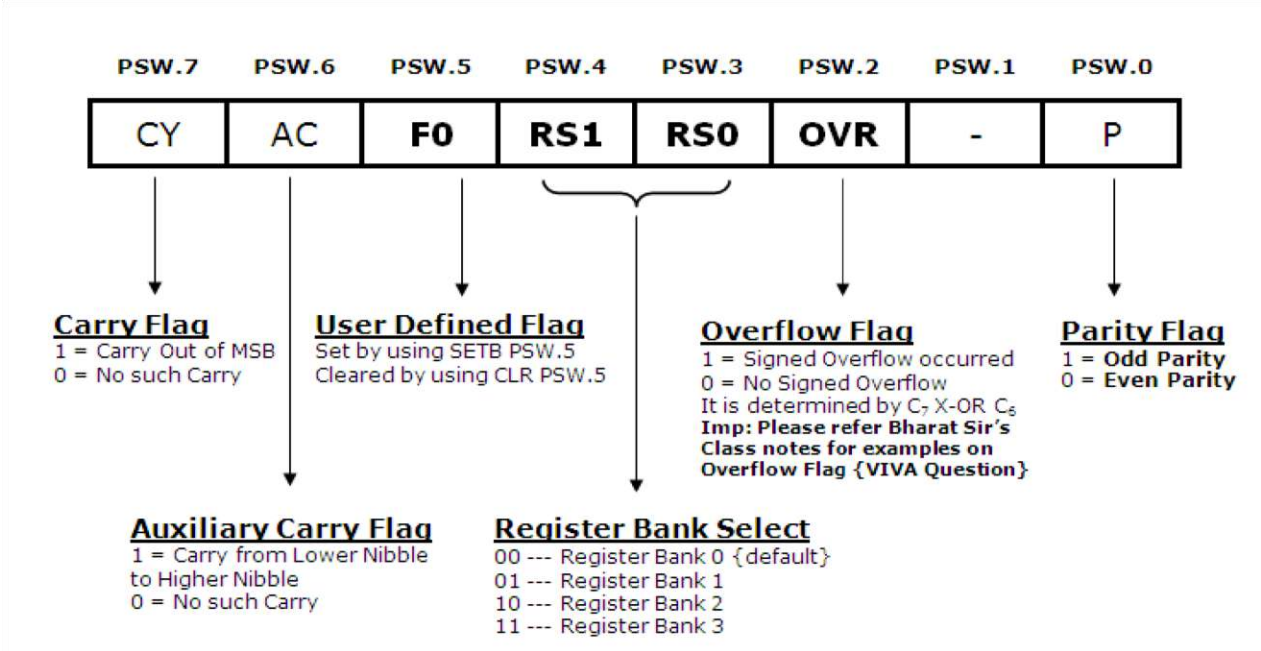
- The general purpose memory is called as the RAM memory of the 8051 microcontroller, which is divided into 3 areas such as **banks**, **bit-addressable area,** and **scratch-pad area.**
- The banks contain different general purpose registers such as R0-R7, and all such registers are byte-addressable registers that store or remove only 1-byte of data.

Banks and Registers:

- The B0, B1, B2, and B3 stand for banks and each bank contains eight general purpose registers ranging from 'R0' to 'R7'.
- All these registers are byte-addressable registers. Data transfer between general purpose registers to general purpose registers is not possible. These banks are selected by the Program Status Word **(PSW**) register.



FLAG REGISTER (PSW) OF 8051:



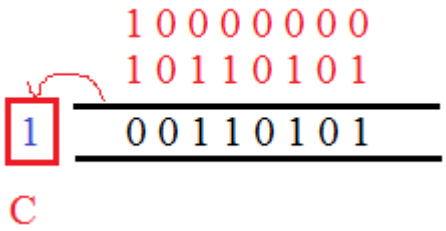
PSW – PROGRAM STATUS WORD

- It is an 8-bit register.
- It is also called the "Flag register", as it mainly contains the status flags.

- These flags indicate status of the current result.
- They are changed by the ALU after every arithmetic or logic operation.
- The flags can also be changed by the programmer.
- PSW is a bit addressable register.
- Each bit can be individually set or reset by the programmer.
- The bits can be referred to by their bit numbers (PSW.4) or by their name (RS1).

CY - CARRY FLAG

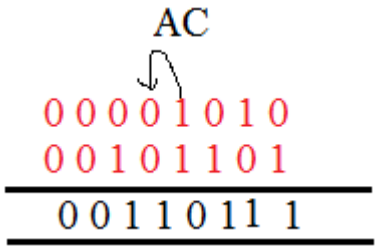
- It indicates the carry out of the MSB, after any arithmetic operation.
- If CY = 1, There was a carry out of the MSB
- If CY = 0, There was no carry out of the MSB



AC – AUXILIARY CARRY FLAG

- It indicates the carry from lower nibble (4-bits) to higher nibble.
- If the 8bits are numbered Bit 7 --- Bit 0, this is the carry from Bit 3 to Bit 4.
- If AC = 1, There was an auxiliary carry
- If AC = 0, There was no auxiliary carry

✓ Note: It is particularly useful in an operation called DA A (Decimal Adjust after Addition).

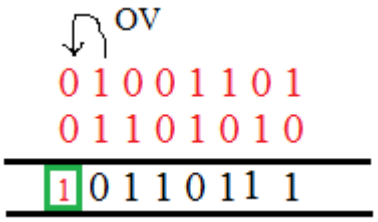


OVR - OVERFLOW FLAG

- It indicates if there was an overflow during a signed operation.
- An 8-bit signed number has the range -80H… 00H… +7FH. Any result, out of this range causes an overflow.

- If OVR = 1, There was an overflow in the result If OVR = , There was no overflow in the result
- Overflow is determined by doing an Ex-Or between the 2nd last carry (C_6) and the last carry (C_7)

✓ Note: After an overflow, the Sign (MSB) of the result becomes wrong.



P - PARITY FLAG

- It indicates the Parity of the result.
- Parity is determined by the number of 1's in the result.
- If PF = 1, The result has ODD parity
- If PF = 0, The result has EVEN parity

F0 – USER DEFINED FLAG

- This flag is available to the programmer.
- It can be used by us to store any **user defined information**.
- For example: In an Air Conditioning unit, programmer can use this flag indicate whether the compressor is ON or OFF (1 or 0).
- This flag can be changed by simple instructions like SETB and CLR.
- SETB PSW.5; This makes F0 bit →1
- CLR PSW.5; This makes F0 bit→0

RS1, RS0 – REGISTER BANK SELECT

- The initial 32 locations (bytes) of the Internal RAM are available to the programmer as registers.
- Having so many registers makes programming easier and faster.
- Naming R0… R31, would tremendously increase the number of opcodes.
- Hence the registers are divided into 4 banks: Bank0… Bank3.
- Each bank has 8 registers named R0… R7.
- At a time, only of the four banks is the "active bank".
- RS1 and RS0 are used by the programmer to select the active bank.

RS1 RS0	REGISTER BANK	SELECTED BY INSTRUCTIONS
0 0	Bank 0	CLR PSW.4 CLR PSW.3
0 1	Bank 1	CLR PSW.4 SETB PSW.3
1 0	Bank 2	SETB PSW.4 CLR PSW.3
1 1	Bank 3	SETB PSW.4 SETB PSW.3

NUMERICAL EXAMPLES FOR FLAG REGISTER:

Example 1:

32 H→0011 0001

23 H→0010 0011

54 H→0101 0100

- Flag Affected: CY=0, AC=0, OVR=0, P=1

Example 2:

39 H→0011 1001

27 H→0010 0111

60 H→0110 0000

- Flag Affected: CY=0, AC=1, OVR=0, P=0

Example 3:

42 H→0100 0010

44 H→0100 0100

86 H→1000 0110

- Flag Affected: CY=0, AC=0, OVR=1, P=1

- The result 86H is out of range for a “Signed” Number as it has become greater than +7FH.
- Such an event is called a “Signed Overflow”.
- In such a case the MSB of the result gives a wrong sign.
- Though the result is +ve (+86H) the MSB is “1” indicating that the result is –ve.
- Overflow is determined by doing an Ex-Or between the 2nd last Carry and the last Carry.
- Here the 2nd last Carry (the one coming into the MSB) is “1”.
- The final carry (The one going out of the MSB) is “0”. As “1” Ex-Or “0” = “1”, the Overflow flag is “1”.

Example 1:

Assembly program to move 6 natural numbers in bank0 register R0-R5

```
Org 0000h (starting addresses declaration)
MOV PSW, #00h (open the bank0 memory)
MOV r0, #00h (starting address of bank0 memory)
MOV r1, #01h
MOV r2, #02h
MOV r2, #03h
MOV r3, #04h
MOV r4, #05h
END
```

Example 2:

Assembly program to move 6 natural numbers in bank1 register R0-R7

```
Org 0000h (starting addresses declaration)
MOV PSW, #08h (open the bank1 memory)
MOV r0, 00h (value send to the bank1 memory)
```

```
MOV r1, 02h
MOV r2, 02h
MOV r2, 03h
MOV r3, 04h
MOV r4, 05h
MOV r5, 06h
MOV r6, 07h
MOV r7, 08h
END
```

6.9 INTERRUPTS OF 8051:

- Interrupts are the events that temporarily suspend the main program, pass the control to the external sources and execute their task. It then passes the control to the main program where it had left off.
- 8051 has 5 interrupt signals, i.e. INT0, TFO, INT1, TF1, and RI/TI. Each interrupt can be enabled or disabled by setting bits of the IE register and the whole interrupt system can be disabled by clearing the EA bit of the same register.

Or

- An interrupt is an event that occurs randomly in the flow of continuity. It is just like a call you have when you are busy with some work and depending upon call priority you decide whether to attend or neglect it.
- Same thing happens in microcontrollers. 8051 architecture handles **5 interrupt sources,** out of which **two are internal (Timer Interrupts),** two are **external** and one is a **serial interrupt**. Each of these interrupts has their interrupt vector address. Highest priority interrupt is the Reset, with vector address 0x0000.

Vector Address:

- This is the address where controller jumps after the interrupt to serve the ISR (interrupt service routine).

Interrupt	Flag	Interrupt vector address
Reset	-	0000H
INT0 (Ext. int. 0)	IE0	0003H
Timer 0	TF0	000BH
INT1 (Ext. int. 1)	IE1	0013H
Timer 1	TF1	001BH
Serial	TI/RI	0023H

Reset

- Reset is the highest priority interrupt, upon reset 8051 microcontroller start executing code from 0x0000 address.

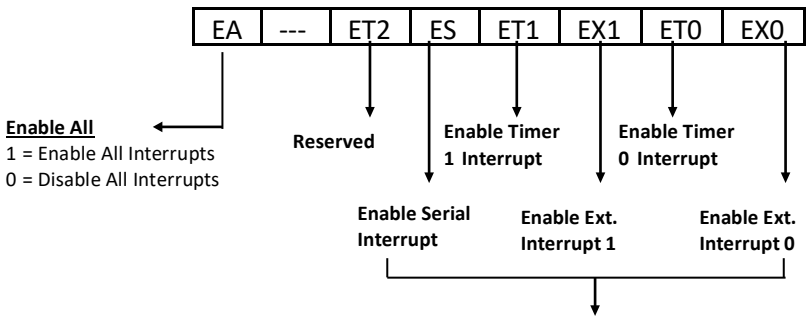
Internal interrupt (Timer Interrupt)

- 8051 has two internal interrupts namely **timer0** and **timer1**. Whenever timer overflows, timer overflow flags (TF0/TF1) are set. Then the microcontroller jumps to their vector address to serve the interrupt. For this, global and timer interrupt should be enabled.

Serial Port Interrupt (Common for RI or TI)

- All interrupts are **vectored** i.e. they cause the program to execute an ISR from a pre-determined address in the Program Memory.
- Interrupts are controlled mainly by **IE** and **IP** SFR's and also by some bits of **TCON** SFR.

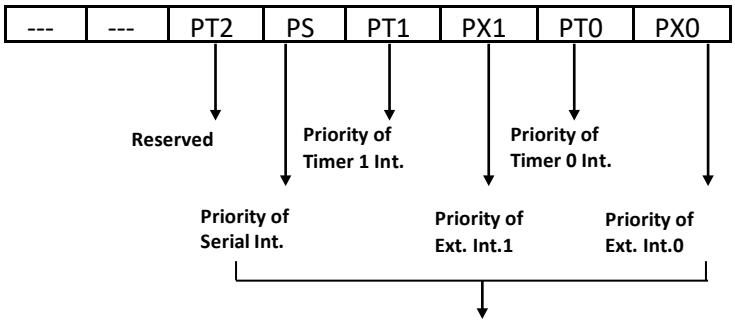
- **IE - Interrupt Enable (SFR) [Bit-Addressable As IE.7 to IE.0]**



1 = Enable respective Interrupt

0 = Disable respective Interrupt

- **<u>IP - Interrupt Priority (SFR)</u> [Bit-Addressable As IP.7 to IP.0]**



- 10 = Priority of respective Interrupt = Priority of respective Interrupt i.e. HIGHLOW

Timer Overflow Interrupts (TF1 and TF0)

- When any of the 2 Timers overflow, their respective bit TFX (TF1 or TF0) is set in TCON SFR.
- If Timer Interrupts are enabled then the timer interrupt occurs. The TFX bits are cleared when their respective ISR is executed.

Serial Port Interrupt (RI or TI)

- While receiving serial data, when a complete byte is received the RI (receive interrupt) bit is set in the SCON.
- During transmission, when a complete byte is transmitted the TI (transmit interrupt) bit is set in the SCON.
- ANY of these events can cause the Serial Interrupt (provided Serial Interrupt is enabled).
- The RI/TI bit is not cleared automatically on executing the ISR. The program should explicitly clear this bit to allow further Serial Interrupts.

IE register: Interrupt Enable Register

- IE register is used to enable/disable interrupt sources.

- **Bit 7 – EA:** Enable All Bit

  1 = Enable all interrupts

  0 = Disable all interrupts
- **Bit 6,5 –** Reserved bits
- **Bit 4 – ES:** Enable Serial Interrupt Bit

  1 = Enable serial interrupt

  0 = Disable serial interrupt
- **Bit 3 – ET1:** Enable Timer1 Interrupt Bit

  1 = Enable Timer1 interrupt

  0 = Disable Timer1 interrupt
- **Bit 2 – EX1:** Enable External1 Interrupt Bit

  1 = Enable External1 interrupt

  0 = Disable External1 interrupt
- **Bit 1 – ET0:** Enable Timer0 Interrupt Bit

  1 = Enable Timer0 interrupt

  0 = Disable Timer0 interrupt
- **Bit 0 – EX0:** Enable External0 Interrupt Bit

  1 = Enable External0 interrupt

  0 = Disable External0 interrupt

Interrupt priority

- Priority to the interrupt can be assigned by using **interrupt priority register (IP)**

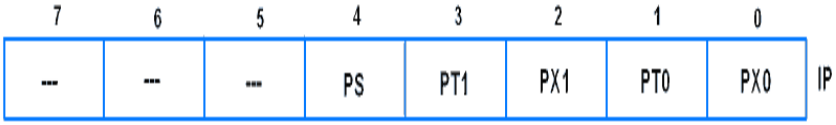
Interrupt priority after Reset:

Priority	Interrupt source	Intr. bit / flag
1	External Interrupt 0	INT0
2	Timer Interrupt 0	TF0
3	External Interrupt 1	INT1
4	Timer Interrupt 1	TF1
5	Serial interrupt	(TI/RI)

- In the table, interrupts priorities upon reset are shown. As per 8051 interrupt priorities, lowest priority interrupts are not served until microcontroller is finished with higher priority ones. In a case when two or more interrupts arrives microcontroller queues them according to priority.

IP Register: Interrupt priority register

- 8051 has interrupt priority register to assign priority to interrupts.



- **Bit 7, 6, 5 –** Reserved bits.
- **Bit 4 – PS:** Serial Interrupt Priority Bit

  1 = Assign high priority to serial interrupt.
  0 = Assign low priority to serial interrupt.

- **Bit 3 – PT1:** Timer1 Interrupt Priority Bit

  1 = Assign high priority to Timer1 interrupt.
  0 = Assign low priority to Timer1 interrupt.

- **Bit 2 – PX1:** External Interrupt 1 Priority Bit

1 = Assign high priority to External1 interrupt.
0 = Assign low priority to External1 interrupt.

- **Bit 1 – PT0:** Timer0 Interrupt Priority Bit

1 = Assign high priority to Timer0 interrupt.
0 = Assign low priority to Timer0 interrupt.

- **Bit 0 – PX0:** External0 Interrupt Priority Bit

1 = Assign high priority to External0 interrupt.
0 = Assign low priority to External0 interrupt.

Or

External Interrupts (INT1 and INT0):

- Pins INT1 and INT0 are inputs for external interrupts.
- These interrupts can be -ve edge or low-level triggered depending upon the IT0 and IT1 bit in
- TCON SFR. (ITX = 1 è -ve edge triggered)
- Whew any of these interrupts occur the respective bits TE1 or IE0 are set in the TCON SFR. If External Interrupts are enabled then the ISR is executed from the respective address.

Interrupt Sequence

- The following sequence is executed to service an interrupt:
- Address of next instruction of the main program i.e. PC is pushed into the Stack.
- All interrupts are disabled, by making EA bit in IE SFR←0.
- Program Control is shifted to the Vector Address (location) of the ISR. The ISR begins.

Returning Sequence

- RETI instruction denotes the end of the ISR.
- It causes the processor to POP the contents of the Stack Top into the PC.
- It also re-enables interrupts by making EA bit in IE SFR←1. The main program resumes.

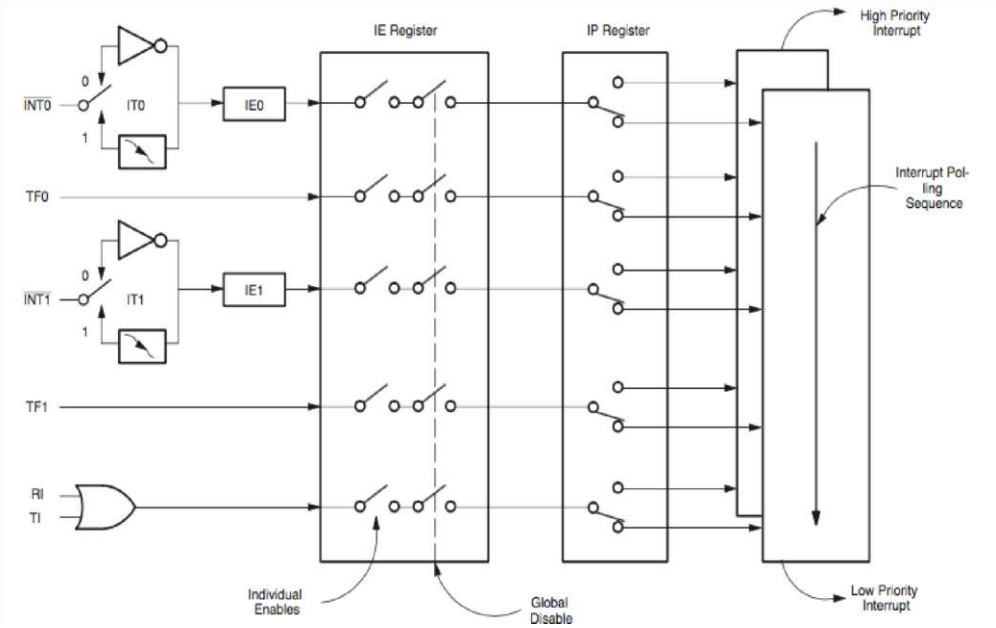
Interrupt Priorities

- 8051 has only two priority levels for the interrupts: Low and High.

- Interrupt priorities are set using the IP SFR.
- As the name suggests, a high priority interrupt can interrupt a low priority interrupt.
- It two or more interrupts at the same level occur simultaneously then priorities are decided as follows:

INTERRUPT	PRIORITY	VECTOR ADDRESS
$\overline{\text{INT0}}$	1	0003H
TF0	2	000BH
$\overline{\text{INT1}}$	3	0013H
TF1	4	001BH
Serial (RI or TI)	5	0023H

DIAGRAM FOR INTERRUPTS



OR

External interrupts in 8051

- 8051 has two external interrupt **INT0 and INT1.**

- 8051 controller can be interrupted by external Interrupt, by providing level or edge on external interrupt pins PORT3.2, PORT3.3.
- External peripherals can interrupt the microcontroller through these external interrupts if global and external interrupts are enabled.
- Then the microcontroller will execute current instruction and jump to the Interrupt Service Routine (ISR) to serve to interrupt.
- In polling method microcontroller has to continuously check for a pulse by monitoring pin, whereas, in interrupt method, the microcontroller does not need to poll. Whenever an interrupt occurs microcontroller serves the interrupt request.
- **External interrupt has two types of activation level**

1. Edge triggered (Interrupt occur on rising/falling edge detection)
2. Level triggered (Interrupt occur on high/low-level detection)

- **In 8051, two types of activation level are used. These are,**

<u>Low level triggered</u>

- Whenever a low level is detected on the INT0/INT1 pin while global and external interrupts are enabled, the controller jumps to interrupt service routine (ISR) to serve interrupt.

<u>Falling edge triggered</u>

- Whenever falling edge is detected on the INT0/INT1 pin while global and ext. interrupts are enabled, the controller jumps to interrupt service routine (ISR) to serve interrupt.
- There are lower four flag bits in **TCON register** required to select and monitor the external interrupt type and ISR status.

TCON: Timer/ counter Register

7	6	5	4	3	2	1	0
TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0

✓ **<u>Bit 3- IE1</u>:**

- External Interrupt 1 edge flag, set by hardware when interrupt on INT1 pin occurred and cleared by hardware when interrupt get processed.

✓ **Bit 2- IT1:**

- This bit selects external interrupt event type on INT1 pin,
- 1= sets interrupt on falling edge
- 0= sets interrupt on low level

✓ **Bit 1- IE0:**

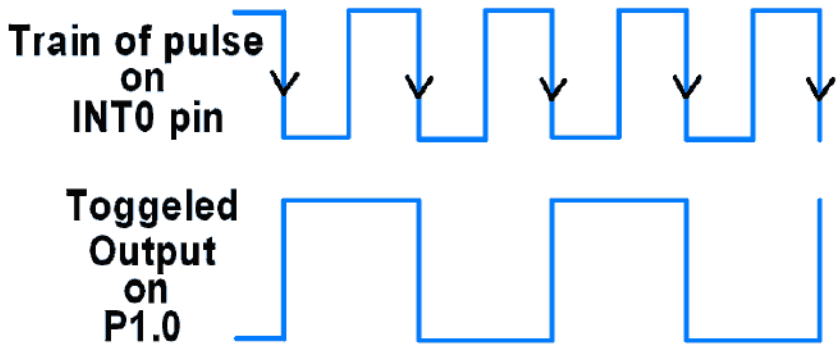
- Interrupt0 edge flag, set by hardware when interrupt on INT0 pin occurred and cleared by hardware when an interrupt is processed.

✓ **Bit 0 - IT0:**

- This bit selects external interrupt event type on INT0 pin.
- 1= sets interrupt on falling edge
- 0= sets interrupt on low level

Example

Let's program the external interrupt of AT89C51 such that, when falling edge is detected on INT0 pin then the microcontroller will toggle the P1.0 pin.



6.10 TIMER SECTION OF 8051:

- The 8051 has two timers: timer0 and timer1. They can be used either as timers or as counters. Both timers are 16 bits wide. Since

the 8051 has an 8-bit architecture, each 16-bit is accessed as two separate registers of low byte and high byte.

- There are two 16-bit timers and counters in 8051 microcontroller: **timer 0 and timer 1**. Both timers consist of 16-bit register in which the lower byte is stored in TL and the higher byte is stored in TH. Timer can be used as a counter as well as for timing operation that depends on the source of clock pulses to counters.
- 8051 has 2, 16-bit Up Counters T1 and T0.
- If the counter counts internal clock pulses it is known as timer.
- If it counts external clock pulses it is known as counter.
- Each counter is divided into 2, 8-bit registers TH1 - TL1 and TH0 - TL0.
- The timer action is controlled mainly by the TCON and the TMOD registers.

TCON - Timer Control (SFR) [Bit-Addressable As TCON.7 to TCON.0]

TF1	**TR1**	**TF0**	**TR0**	IE1	IT1	IE0	IT0

TF1 and TF0: (Timer Overflow Flag)

- Set (1) when Timer 1 or Timer 0 overflows respectively i.e. its bits roll over from all 1's to all 0's.
- Cleared (0) when the processor executes ISR (address 001BH for Timer 1 and 000BH for Timer 0).

TR1 and TR0: (Timer Run Control Bit)

- Set (1) - Starts counting on Timer 1 or Timer 0 respectively.
- Cleared (0) - Halts Timer 1 or Timer 0 respectively.

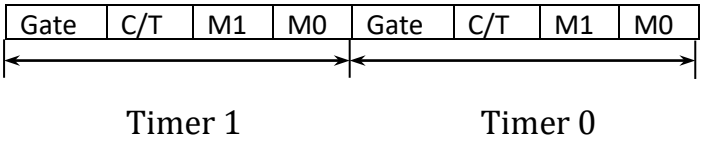
IE1 and IE0: (External Interrupt Edge Flag)

- Set (1) when external interrupt signal received at INT1 or INT0 respectively.
- Cleared (0) when ISR executed (address 0013H for Timer 1 and 0003H for Timer 0).

IT1 and IT0: (External Interrupt Type Control Bit)

- Set (1) - Interrupt at INT1 or INT0 must be -ve edge triggered.
- Cleared (0) - Interrupt at INT1 or INT0 must be low-level triggered.

TMOD - Timer Mode Control (SFR) [NOT Bit-Addressable]



C/T: (Counter/Timer)

- Set (1) - Acts as Counter (Counts external frequency on T1 and T0 pin inputs).
- Cleared (0) - Acts as Timer (Counts internal clock frequency, fosc/12).

Gate: (Gate Enable Control bit)

- Set (1) - Timer controlled by hardware i.e. INTX signal.
- Cleared (0) – Counting independent of INTX signal.

M1, M0: (Mode Selection bits)

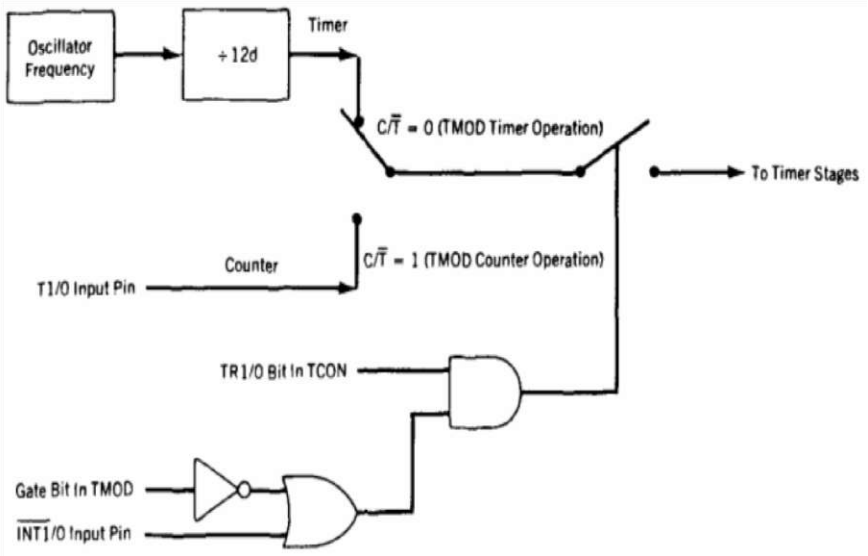
- Used to select the operational modes of the respective Timer.

M1 M0	Timer Mode
0 0	Mode 0
0 1	Mode 1
1 0	Mode 2
1 1	Mode 3

Timer Counter Interrupts

- To use the timer, a certain count value is placed in the Count Register.
- This value is the →**Max count—desired count+1**
- On each count (rising edge of the input clock) the counter increments its value.
- When the counter rolls over (i.e. form all 1's to all 0's) it is said to overflow.
- Thus the Timer Overflow Flag, TFX (TF1 or TF0) is set.
- If timer interrupt is enabled then the Timer Interrupt will occur on overflow.

Timer Counter Logic

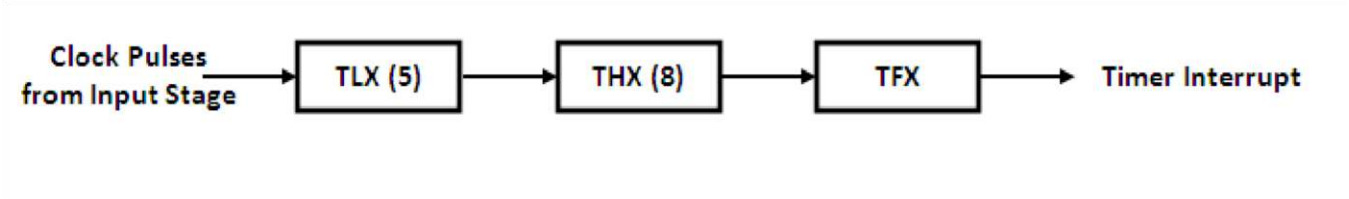


- As shown above, based on C/T bit the timer functions as a Counter or as a Timer.
- If it is a Timer, it will count the internal clock frequency of 8051 divided by 12_d (f/12).
- If it is a Counter, the input clock signal is applied at the TX (T1 or T0) input pins for Timer1 or
- Timer0 respectively. #please refer Bharat Sir's Lecture Notes for this...
- As shown the Timer is running only if the TRX bit (TR1 or TR0) is set.

- Also if the Gate bit is set in the TMOD then the INTX (INT1 or INT0) pin must be "high (1)" for the timer to count.

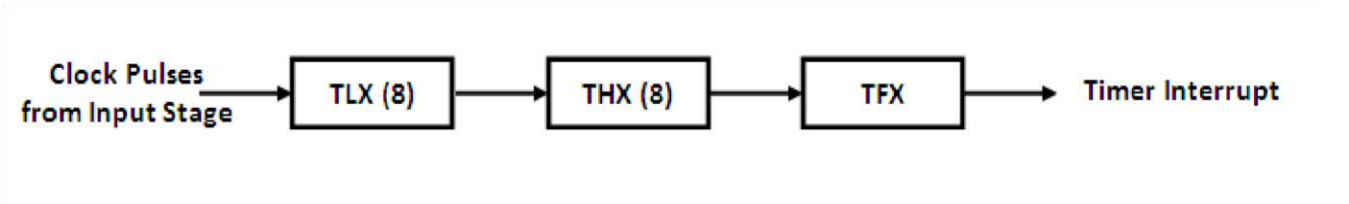
TIMER MODES:

a) Timer Mode 0 (13-bit Timer/Counter)



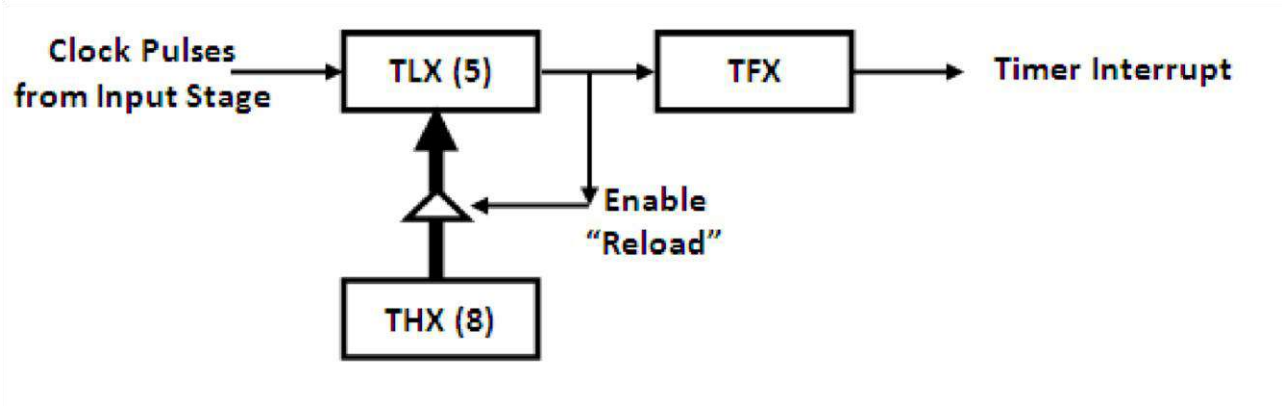
- THX is used as an 8-bit counter.
- TLX is used as a 5-bit pre-set. Hence 13-bits are used for counting.
- On each count the TLX increments.
- Each time TLX rolls-over, THX increments.
- Thus the input frequency is divided by 32 (5-bits of TLX and 2^5 = 32).
- The timer overflow flag TFX is set only when THX overflows i.e. rolls from FFH to 00H. Max Count = 2^{13} = 8K = 8192 (1FFFH). Hence Max Delay→ 8192(12/f)

b) Timer Mode 1 (16-bit Timer/Counter)



- All 16-bits of the Counter are used (8 bits of THX and 8 bits of TLX).
- On each count the 16-bit Timer increments.
- The timer overflow flag TFX is set when the Timer rolls-over from FFFFH to 0000H. Max Count => 2^{16} = 16K = 65536 (FFFFH). Hence Max Delay →65536(12/f).

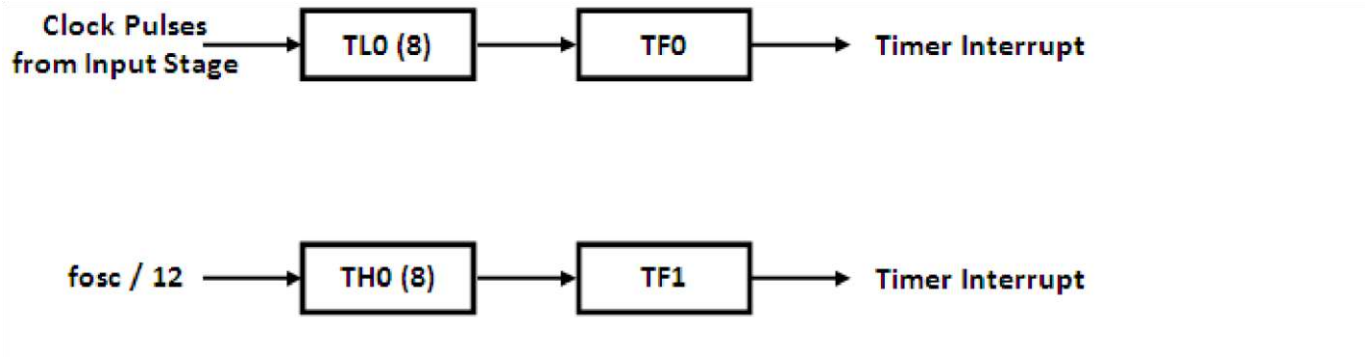
c) Timer Mode 2 (Auto reload TL from TH)



- TLX is used as an 8-bit counter.
- THX holds the count value to be reloaded.
- On each count TLX increments.
- When TLX rolls-over (i.e. from FFH to 00H), the following events take place:
  1. Timer overflow flag TFX is set, hence timer interrupt occurs.
  2. The value of THX is copied into TLX. Hence TLX is auto-reloaded form THX, and the process repeats.

- Thus the timer interrupt occurs at regular intervals "Continuously".
- This mode is used to generate a desired frequency using the Timer Flag. Max Count è 2^8 = 256 (FFH). Hence Max Delay è 256(12/f).

d) Timer Mode 3 (Two 8-bit Timers Using Timer0)



- Timer 0 is used as 2 separate 8-bit timers TH0 and TL0.
- TL0 uses the control bits (TR0 and TF0) of Timer 0.
- It can work as a Timer or a Counter.
- TH0 uses the control bits (TR1 and TF1) of Timer 1.

- It can work only as a Timer. #please refer Bharat Sir's Lecture Notes for this...
- Timer 1 can be in Mode 0, Mode 1, or Mode 2, but will not generate an interrupt.

6.11 8051 TIMER/COUNTER (HARDWARE DELAY) PROGRAMMING:

Example 1:

- **WAP to generate a delay of 20 µsec using internal timer-0 of 8051. After the delay send a "1" through Port3.1. Assume Suitable Crystal Frequency**

NOTE: In 8051, if we select a Crystal of 12 MHz, then Timer freq will be $f_{osc}/12$ è 1MHz. Hence each count will require 1/1MHz è 1 µsec. Thus for 20 µsec, the Desired Count will be 20 14H. For an Up-Counter (Mode 1):
Count = Max Count – Desired Count + 1 Count = FFFF – 14 + 1
Count = FFECH #Please refer Bharat Sir's Lecture Notes for this...

```
SOLN: MOV TMOD, #01H      ;  Program TMOD→(0000 0001)2 ...Timer0 Mode1
      MOV TL0, #0ECH      ;  Load lower byte of Count
      MOV TH0, #0FFH      ;  Load upper byte of Count
      MOV TCON, #10H      ;  Program TCON→ (0001 0000)2…start Timer0
WAIT: JNB TCON.5, WAIT    ;  Wait for overflow
      SETB P3.1           ;  Send a "1" through Port3.1
      MOV TCON, #00H      ;  Stop Timer0
HERE: SJMP HERE           ;  End of program
```

Example 2:

- **WAP to generate a Square wave of 1 KHz from the TxD pin of 8051, Q11 usingTimer1. Assume Clock Frequency of 12 MHz**

NOTE: For a Square wave of 1 KHz, the delay required is .5 msec.
We know, each count will require 1/1MHz è 1 µsec.
Thus for 500 µsec, the Desired Count will be 500_d è 01F4H. For an Up-Counter (Mode 1):
Count = Max Count – Desired Count + 1
Count = FFFF – 01F4 + 1
Count = FE0CH

```
SOLN: CLR P3.1          ;   Clear Txd Line initially
      MOV TMOD, #10H    ;   Program TMOD→(0001 0000)2...Timer1 Mode1
REPEAT: MOV TL1, #0CH   ;   Load lower byte of Count
      MOV TH1, #0FEH    ;   Load upper byte of Count
      MOV TCON, #40H    ;   Program TCON→(0100 0000)2…start Timer1
  WAIT: JNB TCON.7, WAIT ;  Wait for overflow
        CPL P3.1        ;   Toggle Txd pin after the delay
      MOV TCON, #00H    ;   Stop Timer1
        SJMP REPEAT     ;   Repeat the process
```

Example 3:

- **WAP to generate a Rectangular wave of 1 KHz, having a 25% Duty Cycle from the TxD pin of 8051, using Timer1. Assume XTAL of 12 MHz**

NOTE: For a Rectangular wave of 1 KHz, having 25% Duty Cycle: T_{ON} = 250 µsec; T_{OFF} = 750 µsec.

For T_{ON}: Desired Count = 250_d è 00FAH
Count $_{ON}$ = Max Count – Desired Count + 1

Count $_{ON}$ = FFFF – 00FA + 1
Count $_{ON}$ = FF06H

For T_{OFF}: Desired Count = 750 $_{d}$ è 02EEH
Count $_{OFF}$ = Max Count – Desired Count + 1
Count $_{OFF}$ = FFFF – 02EE + 1
Count $_{OFF}$ = FD12H

```
SOLN: MOV TMOD, #10H      ;   Program TMOD→ (0001 0000)2...Timer1 Mode1

REPEAT: MOV TL1, #06H     ;   Load lower byte of Count ON
        MOV TH1, #0FFH    ;   Load upper byte of Count ON
          SETB P3.1       ;   Display"1"at Txd
        MOV TCON, #40H    ;   Program TCON (0100 0000)2... startTimer1
     ON: JNB TCON.7, ON   ;   Maintain"1" at Txd
          CLR P3.1        ;   Clear Txd
        MOV TCON, #00H    ;   Stop Timer1

        MOV TL1, #12H     ;   Load lower byte of Count
        MOV TH1, #0FDH    ;   Load upper byte of Count OFF
        MOV TCON, #40H    ;   Program TCON (0100 0000)2...start Timer1
    OFF: JNB TCON.7, OFF  ;   Maintain "0" at Txd
        MOV TCON, #00H    ;   Stop Timer1

          SJMP REPEAT     ;   Repeat the process
```

Note: If System Freq = 12MHz, it is clear that 1 Count requires 1 msec.
In Mode 1, we have a 16bit Count.
Hence max pulses that can be desired is 2^{16} = 65536.
Count = Max Count – Desired Count + 1
= 65535 – 65536 + 1

= 0.
Thus we will get max delay if we load the count as 0000H, as it will have to "roll-over" back to 0000H to overflow.
Hence Max delay if XTAL is of 12 MHz … is 65536 µsec è **65.536 msec.**
Similarly Max delay if XTAL is of 11.0592 MHz … is 71106 µsec è **71.106 msec.**

Example 4:

- **WAP to generate a delay of <u>1 SECOND</u> using Timer1. Assume Clock Frequency of 12 MHz (Popular Question in College!)**

NOTE: Max delay if XTAL is of 12 MHz … is 65536 µsec è 65.536 msec. Hence to get a delay of 1 second, we will have to perform the counting repeatedly in a loop.
Let's keep the Desired Count 50000. (50 msec delay)
Now 50000_d = C350H
Count = Max Count – Desired Count + 1 Count = FFFF – C350 + 1
<u>Count = 3CB0H</u>
We will have to perform this counting 1sec/50msec times è **<u>20 times</u>**

```
SOLN: MOV TMOD, #10H      ;   Program TMOD→ (0001 0000)2
                              Timer1 Mode1
      MOV R0, #14H        ;   Load count 20 in R0
REPEAT: MOV TL1, #0B0H    ;   Load lower byte of Count ON
      MOV TH1, #3CH       ;   Load upper byte of Count ON
      MOV TCON, #40H      ;   Program TCON (0100 0000)2…start Timer1
  WAIT: JNB TCON.1, WAIT  ;   Wait for an overflow
      MOV TCON, #00H      ;   Stop Timer1
      DJNZ R0, REPEAT     ;   repeat the process 20 times
  HERE: SJMP HERE:        ;   End of program
```

Example 5:

- **WAP to read the data from Port1, 10 times, each after a 1 sec delay. Store the data from RAM locations 20H onwards. When the operation is complete, ring an "Alarm" connected at Port3.1. Assume CLK = 12 MHz**

NOTE: As seen from the previous program, for a delay of 1 second, we have **Count = 3CB0H.** Counting has to be performed **20 times.** Also note that all ports of 8051 are o/p ports by default. To program a port as i/p ports, **all "1"s** must be sent through it.

```
SOLN: CLR P3.1          ;   Clear Port3.1 line

MOV TMOD, #10H          ;   Program TMOD→(0001 0000)2...Timer1Mode1
MOV 90H, #0FFH          ;   Program Port1 as i/p by sending all "1"s through it

REPEAT: MOV R0, #0AH    ;   Load Data Count of 10 in R0

MOV R1, #20H            ;   Load Storage address in R1
MOV @R1, 90H            ;   Read data from Port
INC R1                  ;   Increment data storage address from next Iteration
ACALL DELAY             ;   Call delay of 1 sec before going into next Iteration
DJNZ R0, REPEAT         ;   Repeat till all 10 bytes are read
SETB P3.1               ;   Ring "Alarm" at Port3.1
HERE: SJMP HERE:        ;   End of program

DELAY: MOV R2, #14H     ;   Load count 20 in R0
REPEAT: MOV TL1, #0B0H  ;   Load lower byte of Count ON
MOV TH1, #3CH           ;   Load upper byte of Count ON
```

```
MOV TCON, #40H          ;   Program TCON
                            (0100 0000)…start Timer1

WAIT: JNB TCON.1, WAIT  ;   Wait for an overflow
   MOV TCON, #00H       ;   Stop Timer1
   DJNZ R2, REPEAT      ;   Repeat the process 20
                        times
   RET                  ;   End of delay routine
```

6.12 ADDRESSING MODES OF 8051:

Addressing Modes is the manner in which operands are given in the instruction. 8051 supports the following 5 addressing modes:

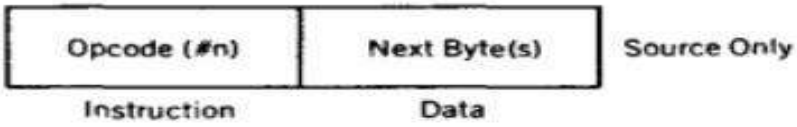
1. Immediate addressing mode
2. Register addressing mode
3. Direct addressing mode
4. Indirect addressing mode
5. Index addressing mode

1. IMMEDIATE ADDRESSING MODE

- In this addressing mode, the Data is given in the Instruction itself.
- We put a "#" symbol, before the data, to identify it as a data value and not as an address.
- **Example**

  MOV A, #35H ; A←35H

  MOV DPTR, #3000H; DPTR← 3000H



Immediate Addressing Mode

2. REGISTER ADDRESSING MODE

- In this addressing mode, Data is given by a Register in the instruction.
- The permitted registers are A, R7 … R0 of each memory bank.

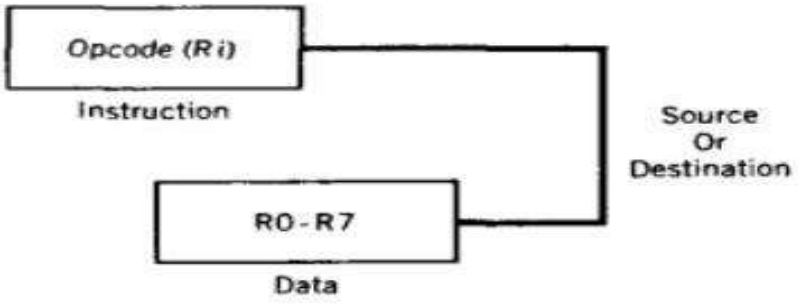
- Note: Data transfer between two RAM registers is not allowed.
- **Example**

MOV A, R0 ; A← R0 … If R0 = 25H, then A gets the Value 25H.

MOV R5, A ; R5← A

MOV Rx, Ry ; NOT ALLOWED. That's because this would allow 64 combinations of register.

; As registers invite opcodes, this would need 64 opcodes!



Register Addressing Mode

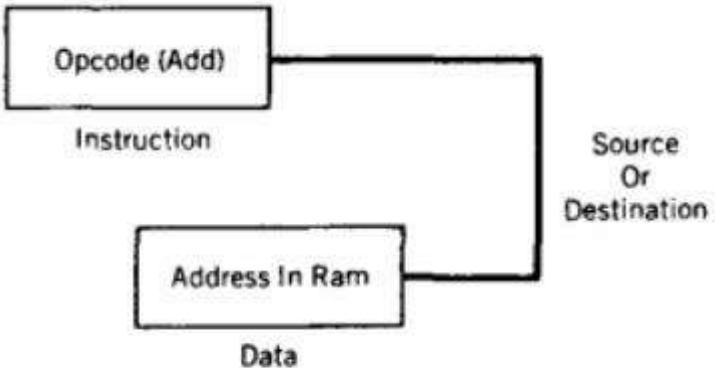
3. DIRECT ADDRESSING MODE

- Here, the address of the operand is given in the instruction.
- Only Internal RAM addresses (00H…7FH) and SFR addresses (from 80H to FFH) allowed.

- **Example**

MOV A, 35 ; A←Contents of RAM location 35H
MOV A, 80H ; A←contents of port 0 (SFR at address 80H)
MOV 20H, 30H ; [20H]←[30H]
i.e. Location 20H gets the contents of location

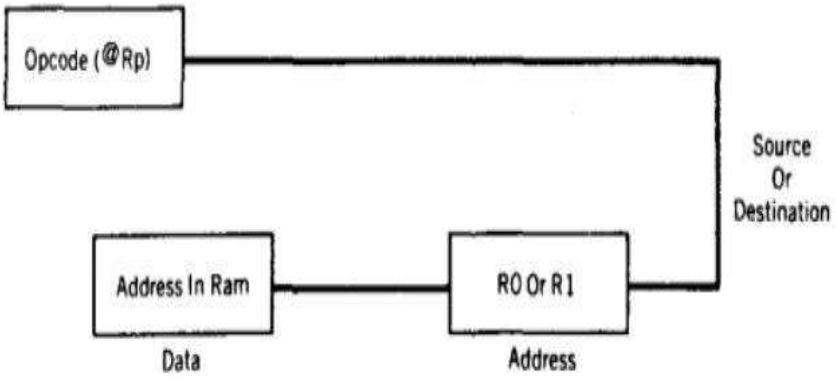
30H.



Direct Addressing Mode

4. INDIRECT ADDRESSING MODE

- Here, the address of the operand is given in a register.
- Internal RAM and External RAM can be accessed using this mode.
- The advantage of giving an address using a register is that we can increment the address in a loop, by simply incrementing the register, and hence access a series of locations.



Indirect Addressing Mode

INTERNAL RAM: (8-BIT ADDRESS GIVEN BY R0 OR R1):

- ONLY R1 or R0, called as Data Pointers, can be used to specify address (00H … 7FH).
- A "@" sign is present before the register to indicate that the register is giving an address.

- **Example:**

MOV A, @R0 ; A← [R0]

; i.e. **A**←Contents of Internal RAM Location whose address is given by R0.

; if R0 = 25H, then A gets the contents of Location 25H from Internal RAM.

MOV @R1, A ; [R1]← A

; i.e. Internal RAM Location pointed by R1 gets value of A.

EXTERNAL RAM: (16 BIT ADDRESS GIVEN BY DPTR):

- For the External RAM, address is provided by R1 or R0, or by DPTR.
- If DPTR is used to give an address, then the full 64KB range of External RAM from 0000H… FFFFH is available. This is because DPTR is 16-bit and 2^{16} = 65536.
- An "X" is present in the instruction, to indicate External RAM.

- **Example**

MOVX A, @DPTR ; A ←[DPTR] ^

; A gets the contents of External RAM location whose address is given by DPTR.

; If DPTR=2000H, then A gets contents of location 0025H from the external RAM

MOVX @DPTR, A ; [DPTR] ^←A

; i.e. A is stored at the External RAM location whose address is given by DPTR.

EXTERNAL RAM: (8 BIT ADDRESS GIVEN BY R0 OR R1):

- If R0 or R1 is used to give an address, then only the first 256 locations of External RAM is available from 0000 H to 00FF H.
- This is because R0 or R1 are 8-bit and 2^8 = only 256.
- Example

MOVX A,@R0 ; A← [R0] ^

; i.e. **A** gets the contents of External RAM Location whose address is

given by R0.

; If R0 = 25H, then A gets contents of Location 0025H from the External RAM

MOVX @R1, A ; [R1] ^←A

; i.e. **A** is stored at the External RAM Location whose address is given by R1

5. INDEXED ADDRESSING MODE

- This mode is used to access data from the Code memory (Internal ROM or External ROM).
- In this addressing mode, address is indirectly specified as a "SUM" of (A and DPTR) or (A and PC).
- This is very useful because ROM contains permanent data which is stored in the form of Look Up tables.
- To access a Look Up table, address is given as a SUM or two registers, where one acts as the base and the other acts as the index within the table.
- A "C" is present in such instructions, to indicate Code Memory.

- **Example**

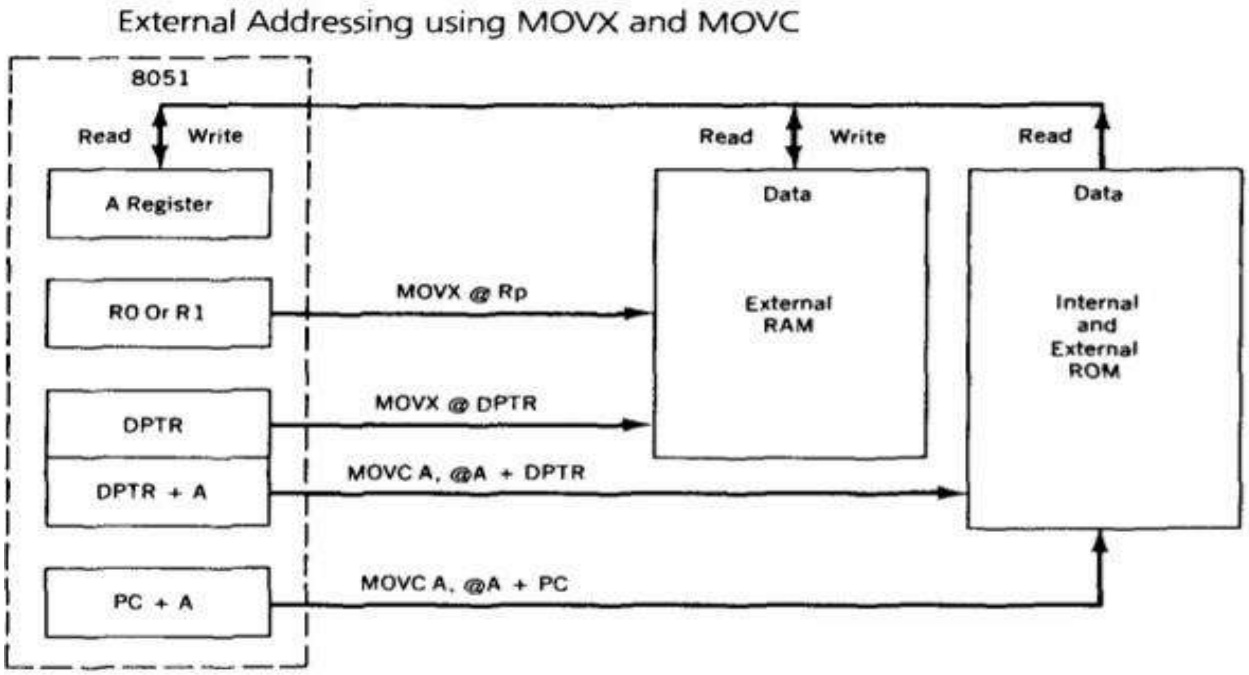
MOVC A, @A+DPTR; A← Contents of a ROM Location pointed by A+DPTR.

; If DPTR = 0400H and A = 05H,

; Then **A** gets the contents of ROM Location whose address is 0405 H.

MOVC A, @A+PC ; A← Contents of a ROM Location pointed by A+PC.

- The same instruction may operate on Internal or External ROM, depending upon the address and on the value of **EA** pin of 8051.
- If the address is in the range of 0000… 0FFFH, then **EA** pin will decide if it operates on Internal
- ROM or External ROM. IF **EA** = 0, External ROM else Internal ROM. If Address is 1000H and more, it will certainly be External ROM.



6.13 8051 ASSEMBLY LANGUAGE PROGRAMMING:

1. **Write a program to add the values of locations 50H and 51H and store the result in locations in 52h and 53H.**

```
        ORG 0000H                ; Set program counter 0000H
        MOV A,50H                ; Load the contents of Memory location 50H into A
        ADD A,51H                ; Add the contents of memory 51H with contents A
        MOV 52H,A                ; Save the LS byte of the result in 52H
        MOV A, #00               ; Load 00H into A
        ADDC A, #00              ; Add the immediate data and carry to A
        MOV 53H,A                ; Save the MS byte of the result in location 53h
        END
```

2. **Write a program to store data FFH into RAM memory locations 50H to 58H using direct addressing mode**

```
ORG 0000H                    ; Set program counter 0000H
MOV A, #0FFH                 ; Load FFH into A
MOV 50H, A    ; Store contents of A in location 50H
MOV 51H, A    ; Store contents of A in location 5IH
MOV 52H, A    ; Store contents of A in location 52H
MOV 53H, A    ; Store contents of A in location 53H
MOV 54H, A    ; Store contents of A in location 54H
MOV 55H, A    ; Store contents of A in location 55H
MOV 56H, A    ; Store contents of A in location 56H
MOV 57H, A    ; Store contents of A in location 57H
MOV 58H, A    ; Store contents of A in location 58H
END
```

3. Write a program to subtract a 16 bit number stored at locations 51H-52H from 55H-56H and store the result in locations 40H and 41H. Assume that the least significant byte of data or the result is stored in low address. If the result is positive, then store 00H, else store 01H in 42H.

```
ORG 0000H                    ; Set program counter 0000H
MOV A, 55H                   ; Load the contents of memory location 55 into
A
CLR C                        ; Clear the borrow flag
SUBB A,51H                   ; Sub the contents of memory 51H from
contents of A
MOV 40H, A                   ; Save the LS Byte of the result in location
40H
MOV A, 56H                   ; Load the contents of memory location
56H into A                   SUBB A, 52H                   ; Subtract the content
of memory 52H from the
                             Content A.
MOV 41H,                     ;Save the MS byte of the result in location
415.
MOV A, #00                   ; Load 005 into A
ADDC A, #00                  ; Add the immediate data and the carry flag
to A
MOV 42H, A                   ; If result is positive, store00H, else store
0lH in 42H
END
```

4. Write a program to add two 16 bit numbers stored at locations 51H-52H and 55H-56H and store the result in locations 40H, 41H and 42H. Assume that the least significant byte of data and the result is stored in low address and the most significant byte of data or the result is stored in high address.

```
        ORG 0000H                ; Set program counter 0000H
        MOV A, 51H               ; Load the contents of memory location 51H
into A
        ADD A, 55H               ; add the contents of 55H with contents of A
        MOV 40H, A               ; Save the LS byte of the result in location
40H
        MOV A, 52H               ; Load the contents of 52H into A
        ADDC A, 56H              ; Add the contents of 56H and CY flag with A
        MOV 41H, A               ; Save the second byte of the result in 41H
        MOV A, #00               ; Load 00H into A
        ADDC A, #00              ; Add the immediate data 00H and CY to A
        MOV 42H, A               ; Save the MS byte of the result in location
42H
        END
```

5. **Write a program to store data FFH into RAM memory locations 50H to 58H using indirect addressing mode.**

```
        ORG 0000H        ; Set program counter 0000H
        MOV A, #0FFH     ; Load FFH into A
        MOV RO, #50H     ; Load pointer, R0-50H
        MOV R5, #08H     ; Load counter, R5-08H
Start:  MOV @RO, A       ; Copy contents of A to RAM pointed by R0
        INC RO           ; Increment pointer
        DJNZ R5, start   ; Repeat until R5 is zero
        END
```

6. **Write a program to add two Binary Coded Decimal (BCD) numbers stored at locations 60H and 61H and store the result in BCD at memory locations 52H and 53H. Assume that the least significant byte of the result is stored in low address.**

```
ORG 0000H          ; Set program counter 00004
MOV A, 60H         ; Load the contents of memory location 60H into A
```

```
        ADD A, 61H          ; Add the contents of memory location 61H with
contents of A
        DA A                ; Decimal adjustment of the sum in A
        MOV 52H, A          ; Save the least significant byte of the result in location
52H
        MOV A, #00          ; Load 00H into .A
        ADDC A, #00H        ; Add the immediate data and the contents of carry
flag to A

        MOV 53H, A          ; Save the most significant byte of the result in
location 53
        END
```

7. **Write a program to clear 10 RAM locations starting at RAM address 1000H.**

```
ORG 0000H               ; Set program counter 0000H
MOV DPTR, #1000H        ; Copy address 1000H to DPTR
CLR A                   ; Clear A
MOV R6, #0AH        ; Load 0AH to R6 again:
MOVX @DPTR, A           ; Clear RAM location pointed by DPTR
INC DPTR                ; Increment DPTR
DJNZ R6, again          ; Loop until counter R6=0
END
```

8. **Write a program to compute 1 + 2 + 3 + N (say N=15) and save the sum at70H**

```
          ORG 0000H             ; Set program counter 0000H
        N EQU 15
          MOV R0, #00           ; Clear R0
        CLR A                   ; Clear A
Again:  INC R0                  ; Increment R0
          ADD A, R0             ; Add the contents of R0 with A
        CJNE R0, #N, again      ; Loop until counter, R0, N
          MOV 70H, A            ; Save the result in location 70H END
```

9. **Write a program to multiply two 8 bit numbers stored at locations 70H and 71H and store the result at memory locations 52H and 53H.**

Assume that the least significant byte of the result is stored in low address.

```
ORG 0000H            ; Set program counter 00 OH
MOV A, 70H           ; Load the contents of memory location 70h into A
MOV B, 71H           ; Load the contents of memory location 71H into B
MUL AB               ; Perform multiplication
MOV 52H, A           ; Save the least significant byte of the result in location 52H MOV 53H, B ; Save the most significant byte of the result in location 53
END
```

10. Write a program to exchange the lower nibble of data present in external memory 6000H and 6001H

```
ORG 0000H            ; Set program counter 00h
MOV DPTR, #6000H     ; Copy address 6000H to DPTR
MOVX A, @DPTR        ; Copy contents of 60008 to A
MOV R0, #45H         ; Load pointer, R0=45H
MOV @R0, A           ; load pointer, r0=45H
INC DPL              ; increment pointer
MOVX A, @DPTR        ; Copy contents of 60018 to A
                     XCHD A, @R0
MOVX @DPTR, A        ; Copy contents of A to 6001 8
DEC DPL              ; Decrement pointer

MOV A, @R0           ; Copy contents of  RAM pointed by RC
                     to A
MOVX @DPTR, A        ; Copy contents of A to RAM pointed
                     by DPTR
END
```

11. Write a program to count the number of and o's of 8 bit data stored in location 6000H.

```
ORG 00008            ; Set program counter 00008
MOV DPTR, #6000h     ; Copy address 6000H to DPTR
MOVX A, @DPTR        ; Copy number to A
MOV R0, #08          ; Copy 08 in RO
MOV R2, #00          ; Copy 00 in R2
```

```
        MOV R3, #00              ; Copy 00 in R3
        CLR C                    ; Clear carry flag
BACK:   RLC A                    ; Rotate A through carry flag
        JC NEXT                  ; If CF=1, branch to next
        INC R2                   ; If CF=0, increment R2 AJMP NEXT2
NEXT:   INC R3                   ; If CF=1, increment R3
NEXT2:  DJNZ RO, BACK            ; Repeat until R0 is zero
        END
```

12.Write a program to shift a 24 bit number stored at 57H-55H to the left Logically four Places.

Assume that the least significant byte of data is stored in lower address.

```
        ORG 0000H          ; Set program counter 0000h
        MOV R1, #04        ; Set up loop count to 4
Again:  MOV A, 55H         ; Place the least significant byte of data in A
        CLR C              ; Clear the carry flag
        RLC A              ; Rotate contents of A (55H) left through carry
        MOV 55H, A
        MOV A, 56H
        RLC A              ; Rotate contents of A (56H) left through carry
        MOV 56H, A
        MOV A, 57H
        RLC A              ; Rotate contents of A (57H) left through carry
        MOV 57H, A
        DJNZ R1, again     ; Repeat until R1 is zero
        END
```

6.14 SERIAL COMMUNICATION:

DATA COMMUNICATION:

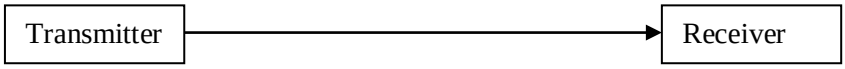
- The 8051 microcontroller is parallel device that transfers eight bits of data simultaneously over eight data lines to parallel I/O devices.
- Parallel data transfer over a long is very expensive. Hence, a serial communication is widely used in long distance communication.
- In serial data communication, 8-bit data is converted to serial bits using a parallel in serial out shift register and then it is transmitted over a single data line.

- The data byte is always transmitted with least significant bit first.

BASICS OF SERIAL DATA COMMUNICATION,
Communication Links

1. **Simplex communication link:**

- In simplex transmission, the line is dedicated for transmission. The transmitter sends and the receiver receives the data.

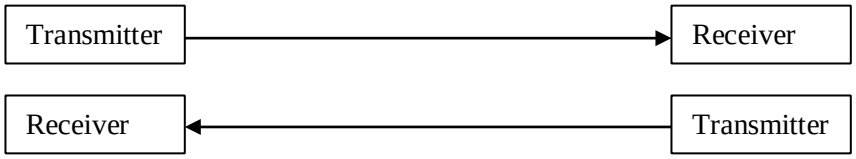


2. **Half duplex communication link:**

- In half duplex, the communication link can be used for either transmission or reception. Data is transmitted in only one direction at a time.

3. **Full duplex communication link:**

- If the data is transmitted in both ways at the same time, it is a full duplex i.e. transmission and reception can proceed simultaneously. This communication link requires two wires for data, one for transmission and one for reception.

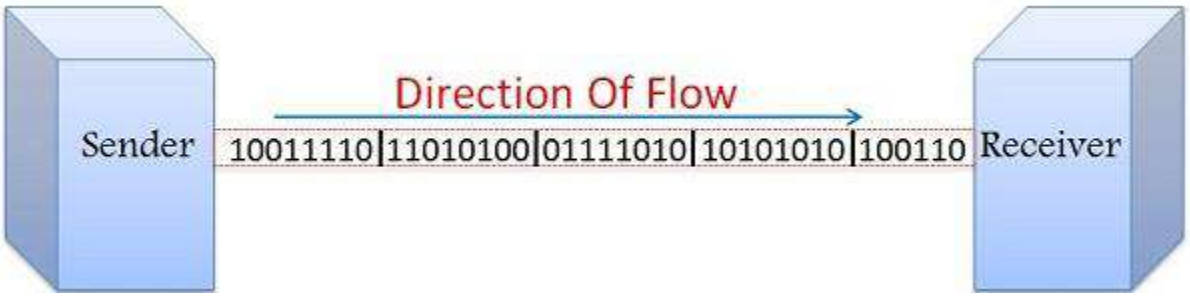


Types of Serial communication:
Serial data communication uses two types of communication.

1. Synchronous serial data communication:

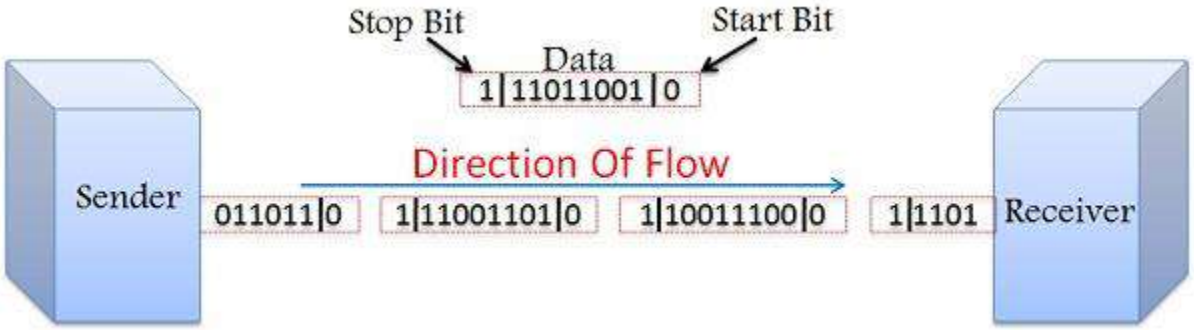
- In this transmitter and receiver are synchronized.
- It uses a common clock to synchronize the receiver and the transmitter.
- First the synch character is sent and then the data is transmitted.
- This format is generally used for high speed transmission.

- In Synchronous serial data communication a block of data is transmitted at a time.



2. Asynchronous Serial data transmission:

- In this, different clock sources are used for transmitter and receiver.
- In this mode, data is transmitted with start and stop bits.
- A transmission begins with start bit, followed by data and then stop bit.
- For error checking purpose parity bit is included just prior to stop bit.
- In Asynchronous serial data communication a single byte is transmitted at a time.



Baud rate:

- The rate at which the data is transmitted is called baud or transfer rate.
- The baud rate is the reciprocal of the time to send one bit.
- In asynchronous transmission, baud rate is not equal to number of bits per second.
- This is because; each byte is preceded by a start bit and followed by parity and stop bit.
- **For example**, in synchronous transmission, if data is transmitted with 9600 baud it means that 9600 bits are transmitted in one second.
- For bit transmission time = 1 second/ 9600 = 0.104 ms.

8051 SERIAL COMMUNICATION:

The 8051 supports a full duplex serial port.

Three special function registers support serial communication.

1. **SBUF Register:**

Serial Buffer (SBUF) register is an 8-bit register. It has separate SBUF registers for data transmission and for data reception. For a byte of data to be transferred via the TXD line, it must be placed in SBUF register. Similarly, SBUF holds the 8-bit data received by the RXD pin and read to accept the received data.

2. **SCON register:**

The contents of the Serial Control (SCON) register are shown below. This register contains mode selection bits, serial port interrupt bit (TI and RI) and also the ninth data bit for transmission and reception (TB8 and RB8).

Serial Port Control (SCON) Register							
D7	D6	D5	D4	D3	D2	D1	D0
SM0	**SM1**	**SM2**	**REN**	**TB8**	**RB8**	**TI**	**RI**

- SM0 (SCON.7) : Serial communication mode selection bit
- SM1 (SCON.6) : Serial communication mode selection bit

SM0	SM1	Mode	Description	Baud rate
0	0	Mode 0	8-bit shift register mode	Fosc / 12
0	1	Mode 1	8-bit UART	Variable (set by timer 1)
1	0	Mode 2	9-bit UART	Fosc/ 32 or Fosc/64
1	1	Mode 3	9-bit UART	Variable (set by timer 1)

- SM2 (SCON.5) : Multiprocessor communication bit. In modes 2 and 3, if set this will enable multiprocessor communication.
- REN (SCON.4) : Enable serial reception
- TB8 (SCON.3) : This is 9^{th} bit that is transmitted in mode 2 & 3.
- RB8 (SCON.2) : 9^{th} data bit is received in modes 2 & 3.
- TI (SCON.1) : Transmit interrupt flag, set by hardware must be cleared by software.
- RI (SCON.0) : Receive interrupt flag, set by hardware must be cleared by software.

3. **PCON register:**

The SMOD bit (bit 7) of PCON register controls the baud rate in asynchronous mode transmission.

Power mode Control (PCON) Register							
D7	D6	D5	D4	D3	D2	D1	D0
SMOD	**--**	**--**	**--**	**GF1**	**GF0**	**PD**	**IDL**

- SMD (PCON.7): Serial rate modify bit. Set to 1 by program to double baud rate using timer 1 for modes 1, 2, and 3. cleared by program to use timer 1 baud rate.

- GF1 (PCON.3) : General Purpose user flag bit.
- GF0 (PCON.2) : General Purpose user flag bit.
- PD (PCON.1) : Power down bit. Set to 1 by program to enter power down configuration for CHMOS processors.
- IDL (PCON.0) : Idle mode bit. Set to 1 by program to enter idle mode configuration for CHMOS processors.

SERIAL COMMUNICATION MODES:

1. Mode 0

In this mode serial port runs in synchronous mode. The data is transmitted and received through RXD pin and TXD is used for clock output. In this mode the baud rate is 1/12 of clock frequency.

2. Mode 1

In this mode SBUF becomes a 10 bit full duplex transceiver. The ten bits are 1 start bit, 8 data bit and 1 stop bit. The interrupt flag TI/RI will be set once transmission or reception is over. In this mode the baud rate is variable and is determined by the timer 1 overflow rate.

Baud rate = $[2^{smod}/32]$ x Timer 1 overflow Rate

= $[2^{smod}/32]$ x [Oscillator Clock Frequency] / [12 x [256 – [TH1]]]

3. Mode 2

This is similar to mode 1 except 11 bits are transmitted or received. The 11 bits are, 1 start bit, 8 data bit, a programmable 9^{th} data bit, 1 stop bit.

Baud rate = $[2^{smod}/64]$ x Oscillator Clock Frequency

4. Mode 3

This is similar to mode 2 except baud rate is calculated as in mode 1

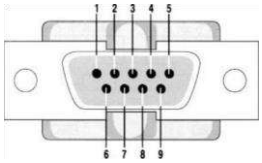
CONNECTIONS TO RS-232

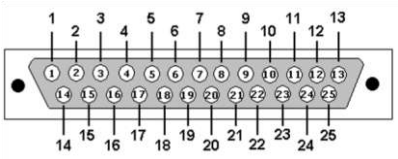
RS-232 standards:

To allow compatibility among data communication equipment made by various manufactures, and interfacing standard called RS232 was set by the Electronics Industries Association (EIA) in 1960. Since the standard was set long before the advent of logic family, its input and output voltage levels are not TTL compatible.

In RS232, a logic one (1) is represented by -3 to -25V and referred as MARK while logic zero (0) is represented by +3 to +25V and referred as SPACE. For this reason to connect any RS232 to a microcontroller system we must use voltage converters such as MAX232 to convert the TTL logic level to RS232 voltage levels and vice-versa. MAX232 IC chips are commonly referred as line drivers.

In RS232 standard we use two types of connectors. DB9 connector or DB25 connector.





DB9 Male Connector DB25 Male Connector

The pin description of DB9 and DB25 Connectors are as follows

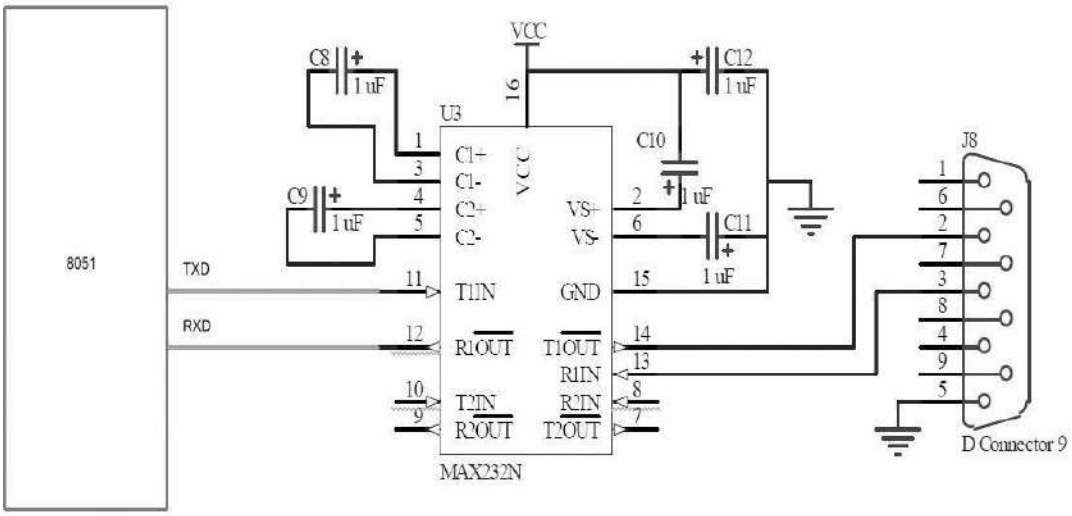
DB-25 Pin No.	DB-9 Pin No.	Abbreviation	Full Name
Pin 2	Pin 3	TD	Transmit Data
Pin 3	Pin 2	RD	Receive Data
Pin 4	Pin 7	RTS	Request To Send
Pin 5	Pin 8	CTS	Clear To Send
Pin 6	Pin 6	DSR	Data Set Ready
Pin 7	Pin 5	SG	Signal Ground
Pin 8	Pin 1	CD	Carrier Detect
Pin 20	Pin 4	DTR	Data Terminal Ready
Pin 22	Pin 9	RI	Ring Indicator

The 8051 connection to MAX232 is as follows:

- The 8051 has two pins that are used specifically for transferring and receiving data serially.
- These two pins are called TXD, RXD. Pin 11 of the 8051 (P3.1) assigned to TXD and pin 10 (P3.0) is designated as RXD. These pins TTL compatible; therefore they require line driver (MAX 232) to make them RS232

compatible.

- MAX 232 converts RS232 voltage levels to TTL voltage levels and vice versa.
- One advantage of the MAX232 is that it uses a +5V power source which is the same as the source voltage for the 8051.
- The typical connection diagram between MAX 232 and 8051 is shown below.



SERIAL COMMUNICATION PROGRAMMING IN ASSEMBLY AND C.

Steps to programming the 8051 to transfer data serially

1. The TMOD register is loaded with the value 20H, indicating the use of the Timer 1 in mode 2 (8-bit auto reload) to set the baud rate.
2. The TH1 is loaded with one of the values in table 5.1 to set the baud rate for serial data transfer.
3. The SCON register is loaded with the value 50H, indicating serial mode 1, where an 8-bit data is framed with start and stop bits.
4. TR1 is set to 1 start timer 1.
5. TI is cleared by the "CLR TI" instruction.
6. The character byte to be transferred serially is written into the SBUF register.
7. The TI flag bit is monitored with the use of the instruction JNB TI, target to see if the character has been transferred completely. 8. To transfer the next character, go to step 5.

Example 1. Write a program for the 8051 to transfer letter 'A' serially at 4800-baud rate, 8 bit data, and 1 stop bit continuously.

```
ORG 0000H
LJMP START
ORG 0030H
   START: MOV TMOD, #20H      ; select timer 1 mode 2
    MOV TH1, #0FAH            ; load count to get baud rate of 4800
    MOV SCON, #50H            ; initialize UART in mode 2
                              ; 8 bit data and 1 stop bit
     SETB TR1                 ; start timer
   AGAIN: MOV SBUF, #'A'      ; load char 'A' in SBUF
BACK: JNB TI, BACK            ; Check for transmit interrupt flag
     CLR TI                   ; Clear transmit interrupt flag
   SJMP AGAIN
   END
```

Example 2. Write a program for the 8051 to transfer the message 'EARTH' serially at 9600 baud, 8 bit data, and 1 stop bit continuously.

```
ORG 0000H
LJMP START

ORG 0030H
   START: MOV TMOD, #20H                    ; select timer 1 mode 2
MOV TH1, #0FDH              ; load count to get required baud rate of 9600 MOV
SCON, #50H                  ; initialize uart in mode 2
                                  ; 8 bit data and 1 stop bit
    SETB TR1                      ; start timer
   LOOP: MOV A, #'E'        ; load 1st letter 'E' in a
    ACALL LOAD                    ; call load subroutine
    MOV A, #'A'              ; load 2nd letter 'A' in a
    ACALL LOAD                     ; call load subroutine
    MOV A, #'R'              ; load 3rd letter 'R' in a
    ACALL LOAD                    ; call load subroutine
```

```
MOV A, #'T'                    ; load 4th letter 'T' in a
ACALL LOAD                     ; call load subroutine
MOV A, #'H'                    ; load 4th letter 'H' in a
ACALL LOAD                     ; call load subroutine
SJMP LOOP                      ; repeat steps

LOAD:  MOV SBUF, A
HERE:  JNB TI, HERE            ; Check for transmit interrupt flag
CLR TI                 ; Clear transmit interrupt flag RET

END
```

6.15 INTERFACING:

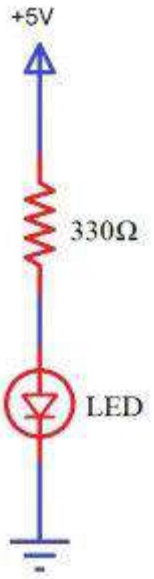
Interfacing is the process of connecting devices together so that they can exchange the information and that proves to be easier to write the programs. There are different type of input and output devices as for our requirement such as LEDs, LCDs, 7segment, keypad, motors and other devices.

Interfacing LED with 8051

- Light Emitting Diodes or LEDs are the mostly commonly used components in many applications. They are made of semiconducting material. In this project, I will describe about basics of Interfacing LED with 8051 Microcontroller.

- Light Emitting Diodes are the semiconductor light sources. Commonly used LEDs will have a cut-off voltage of 1.7V and current of 10mA. When an LED is applied with its required voltage and current it glows with full intensity.

- The Light Emitting Diode is similar to the normal PN diode but it emits energy in the form of light. The color of light depends on the band gap of the semiconductor. The following figure shows "how an LED glows?"



- Thus, LED is connected to the AT89C51 microcontroller with the help of a current limiting resistor. The value of this resistor is calculated using the following formula.

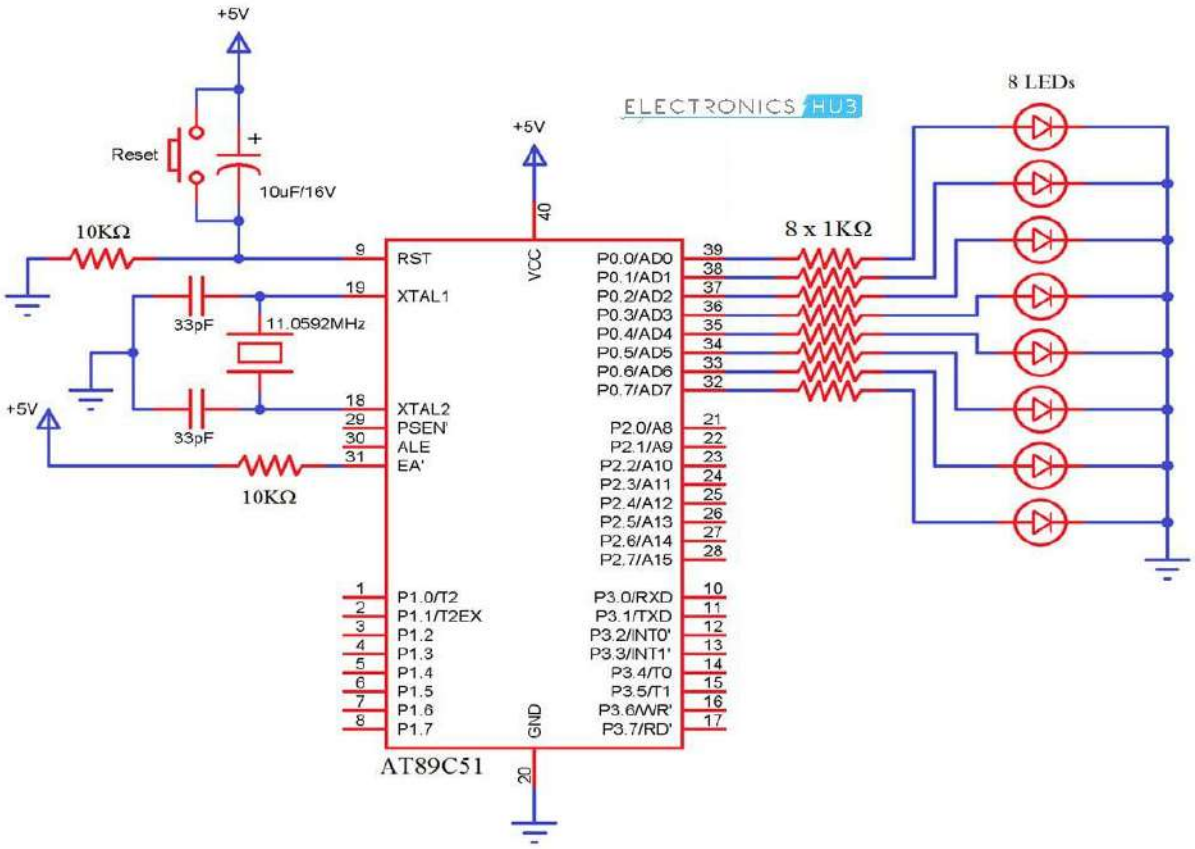
R= (V-1.7)/10mA, where V is the input voltage.

- Generally, microcontrollers output a maximum voltage of 5V. Thus, the value of resistor calculated for this is 330 Ohms. This resistor can be connected to either the cathode or the anode of the LED.

Principle behind Interfacing LED with 8051:

- The main principle of this circuit is to interface LEDs to the 8051 family micro controller. Commonly, used LEDs will have voltage drop of 1.7v and current of 10mA to glow at full intensity. This is applied through the output pin of the micro controller.

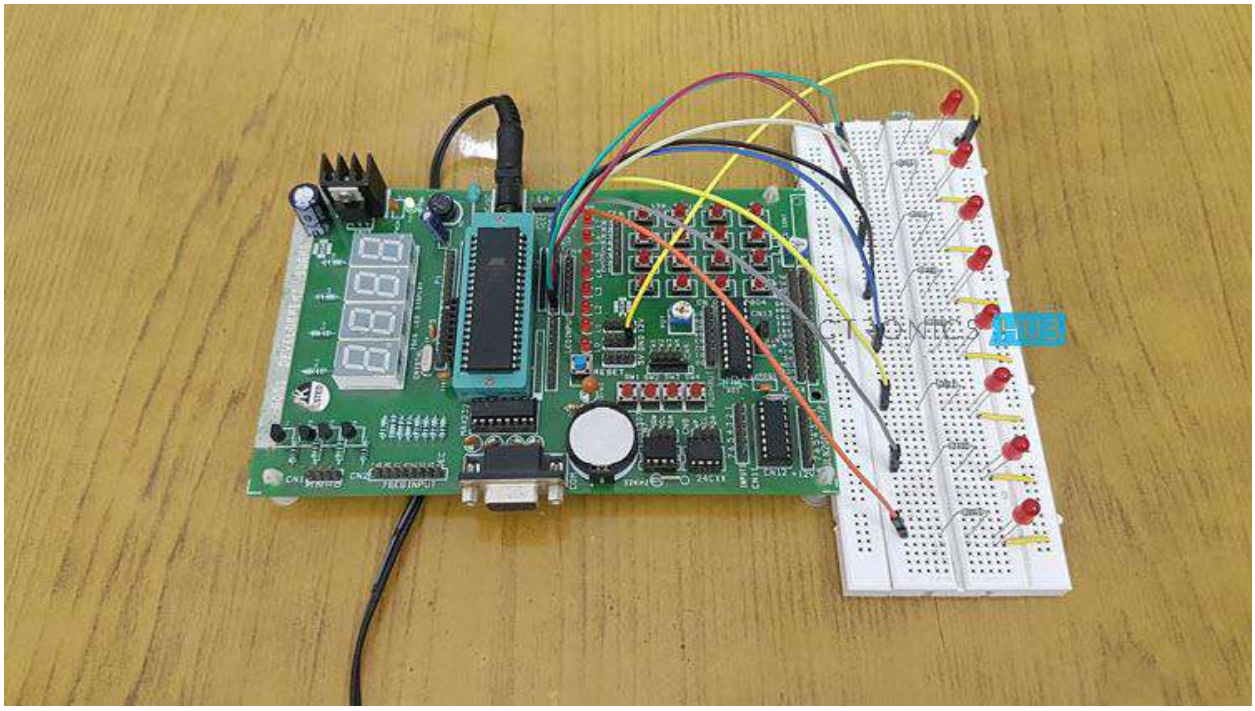
Circuit Diagram



Circuit Design

The circuit mainly consists of AT89C51 microcontroller. AT89C51 belongs to the family of 8051 microcontroller. It is an 8-bit microcontroller. This microcontroller has 4KB of Flash Programmable and Erasable Read Only Memory and 128 bytes of RAM. This can be programmed and erased a maximum of 1000 times.

It has two 16 bit timers/counters. It supports USART communication protocol. It has 40 pins. There are four ports are designated as P0, P1, P2, and P3. Port P0 will not have internal pull- ups, while the other ports have internal pull-ups.



In this circuit, LEDs are connected to the port P0. The controller is connected with external crystal oscillator to pin 18 and 19 pins. Crystal pins are connected to the ground through capacitors of 33pf.

INTERFACING STEPPER MOTOR WITH 8051:

A stepper motor is a device that translates electrical pulses into mechanical movement in steps of fixed step angle.

• The stepper motor rotates in steps in response to the applied signals.

• It is mainly used for position control.

• It is used in disk drives, dot matrix printers, plotters and robotics and process control circuits.

Structure

Stepper motors have a permanent magnet called rotor (also called the shaft) surrounded by a stator. The most common stepper motors have four stator windings that are paired with a center-tap. This type of stepper motor is commonly referred to as a four-phase or unipolar stepper motor. The center

tap allows a change of current direction in each of two coils when a winding is grounded, thereby resulting in a polarity change of the stator.

Interfacing

Even a small stepper motor require a current of 400 mA for its operation. But the ports of the microcontroller cannot source this much amount of current. If such a motor is directly connected to the microprocessor/microcontroller ports, the motor may draw large current from the ports and damage it. So a suitable driver circuit is used with the microprocessor/microcontroller to operate the motor.

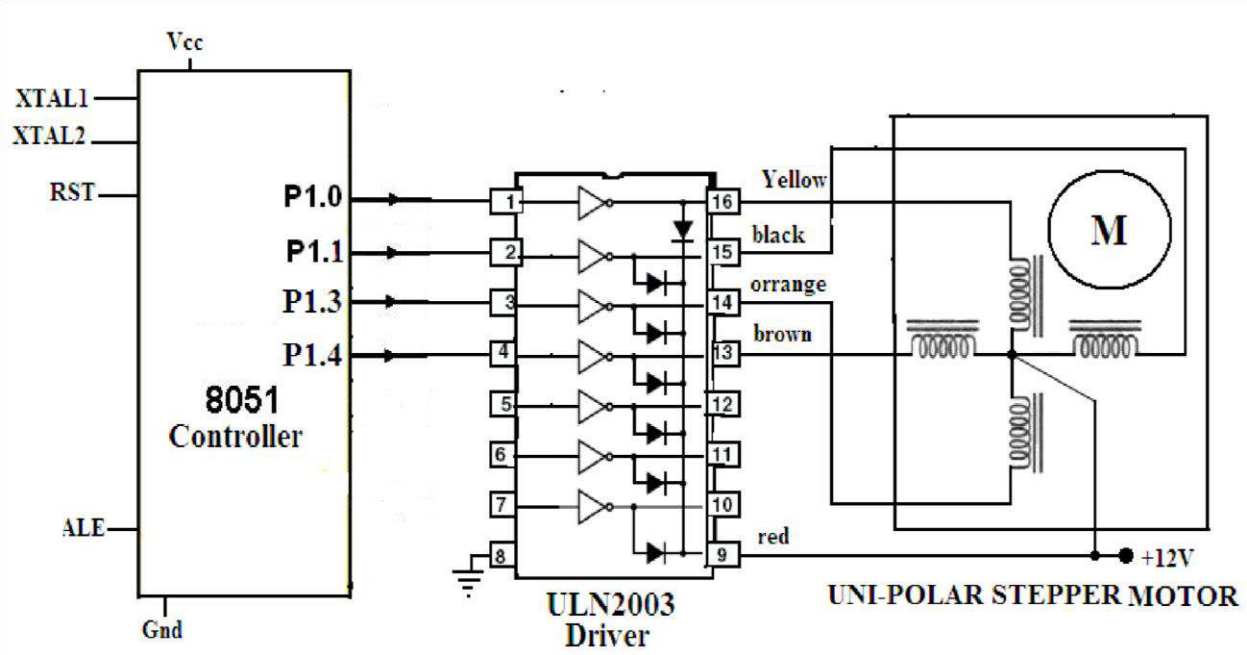
Motor Driver Circuit (ULN2003)

Stepper motor driver circuits are available readily in the form of ICs. ULN2003 is one such driver IC which is a High-Voltage High-Current Darlington transistor array and can give a current of 500mA.This current is sufficient to drive a small stepper motor. Internally, it has protection diodes used to protect the motor from damage due to back e.m.f. and large eddy currents. So, this ULN2003 is used as a driver to interface the stepper motor to the microcontroller.

Operation:

- The important parameter of a stepper motor is the **step angle**.
- It is the minimum angle through which the motor rotates in response to each **excitation pulse**.
- In a four phase motor if there are 200 steps in one complete rotation then then the step angle is $360/200 = 1.8^{o}$.
- So to rotate the stepper motor we have to apply the excitation pulse. For this the controller should send a hexa decimal code through one of its ports.
- **The hex code mainly depends on the construction of the stepper motor**. So, all the stepper motors do not have the same Hex code for their rotation.
- For example, let us consider the hex code for a stepper motor to rotate in clockwise direction is 77H, BBH, DDH and EEH.
- This hex code will be applied to the input terminals of the driver through the assembly language program.
- To rotate the stepper motor in anti-clockwise direction the same code is applied in the reverse order.

Stepper Motor interface- Schematic Diagram (for 8051)



The assembly language program for 8051 is given below.

ASSEMBLY LANGUAGE PROGRAM (8051)

```
Main  : MOV A, # 0FF H          ;  Initialization of Port 1
        MOV P1, A               ;
        MOV A, #77 H            ; Code for the Phase 1
        MOV P1, A               ;
        ACALL DELAY             ;  Delay subroutine
        MOV A, # BB H           ; Code for the Phase II
        MOV P1, A               ;
        ACALL DELAY             ;  Delay subroutine.
      MOV A, # DD H             ; Code for the Phase III
        MOV P1, A               ;
        ACALL DELAY             ; Delay subroutine
        MOV A, # EE H           ;  Code for the Phase 1
        MOV P1, A               ;
```

```
        ACALL DELAY              ;  Delay subroutine
        SJMP MAIN; Keep the motor rotating continuously.
```

DELAY Subroutine

```
        MOV R4, #0FF H           ; Load R4 with FF
        MOV R5, # 0FF            ; Load R5 with FF
LOOP1:  DJNZ R4, LOOP1           ; Decrement R4 until zero,
tLOOP2:  DJNZ R5, LOOP2          ; Decrement R5 until
                                 zero, wait
        RET                      ; Return to main
                                 program
```