**Class Notes**

**UNIT- I**

**Introduction to Microprocessor Systems:** Architecture and PIN diagram of 8085, Timing Diagram, memory organization, Addressing modes, Interrupts. Assembly Language Programming.

**Introduction to Microprocessor Systems**

Microprocessor is a electronic chip, that functions as the central processing unit of a  computer. All processors are use the basic concept of stored program execution. program or instructions are stored sequentially in the memory. Every microprocessor has its own associated set of instructions. [Instruction set](http://mvn.edu.in/mvnlms/mod/assign/view.php?id=904) for microprocessor is in two forms one in mnemonic, which is comparatively easy to understand and the other is binary machine code.

[**8085 microprocessor**](http://mvn.edu.in/mvnlms/mod/quiz/view.php?id=909):

* The **Intel 8085** is an [8-bit](http://en.wikipedia.org/wiki/8-bit) [microprocessor](http://en.wikipedia.org/wiki/Microprocessor) introduced by [Intel](http://en.wikipedia.org/wiki/Intel) in 1977. The 8085 is a conventional [von Neumann](http://en.wikipedia.org/wiki/Von_Neumann_architecture)design based on the Intel 8080.
* It is designed by using nmos technology. The "5" in the model number came from the fact that the 8085 requires only a +5-[Volt](http://en.wikipedia.org/wiki/Volt) (V) power supply. rather than requiring the +5 V, −5 V and +12 V supplies the 8080 needed.
* It has 8 bit data bus and 16 bit address bus. it can work upto 5 MHz frequency. It has 40 pins in its chip. Lower order address bus is multiplexed with data bus to minimize the chip size.
* The 8085 has extensions to support new interrupts, with three maskable interrupts (RST 7.5, RST 6.5 and RST 5.5), one [non-maskable interrupt](http://en.wikipedia.org/wiki/Non-maskable_interrupt) (TRAP), and one externally serviced interrupt (INTR). The RST n.5 interrupts refer to actual pins on the processor, a feature which permitted simple systems to avoid the cost of a separate interrupt controller.

### Architecture of 8085

### This is the functional Block Diagram of 8085 Microprocessor.

### http://www.8085projects.info/images/Arhitecture-Pic1.png

8085 architecture consists of five functional units:

1. **Arithmatic and logic unit** :

The ALU performs the actual numerical and logic operation such as add’, ‘subtract’, ‘AND’, ‘OR’, etc. Uses data from memory and from Accumulator to perform arithmetic. Always stores result of operation in Accumulator.

**2. Genaral purpose registers**:

8-bit B and 8-bit C registers can be used as one 16-bit BC register pair. When used as a pair the C register contains low-order byte. Some instructions may use BC register as a data pointer.

8-bit D and 8-bit E registers can be used as one 16-bit DE register pair. When used as a pair the E register contains low-order byte. Some instructions may use DE register as a data pointer.

8-bit H and 8-bit L registers can be used as one 16-bit HL register pair. When used as a pair the L register contains low-order byte. HL register usually contains a data pointer used to reference memory addresses.

3.   **special purpose registers:**

**a) Accumulator** or A register is an 8-bit register used for arithmetic, logic, I/O and load/store operations.

**b) Flag** is an 8-bit register containing 5 1-bit flags:

* Sign - set if the most significant bit of the result is set.
* Zero - set if the result is zero.
* Auxiliary carry - set if there was a carry out from bit 3 to bit 4 of the result.
* Parity - set if the parity (the number of set bits in the result) is even.
* Carry - set if there was a carry during addition, or borrow during subtraction/comparison.

**c)Stack pointer** is a 16 bit register. This register is always incremented/decremented by 2

**d)Program counter** is a 16-bit register.

**4. instruction register and decoder**:

Temporary store for the current instruction ofa program. Latest instruction sent here from memory prior to execution. Decoder then takes instruction and ‘decodes’ or interprets the instruction. Decoded instruction then passed to next stage.

**5. Timing and control unit**: Generates signals within uP to carry out the instruction, which has been decoded. In reality causes certain connections between blocks of the uP to be opened or closed, so that data goes where it is required, and so that ALU operations occur.

**STACK AND STACK RELATED INSTRUCTIONS:**

The stack is a group of memory locations in the R/W memory that is used for the temporary storage of binary information during the execution of the program.

The stack related instructions are PUSH & POP

**PURPOSE OF SID AND SOD LINES:**

**SID** (Serial input data line):       It is an input line through which the microprocessor accepts serial data

**SOD** (Serial output data line):   It is an output line through which the microprocessor sends output serial data.

**OPCODE:**The part of the instruction that specifies the operation to be performed is called the operation code or Opcode.

**FUNCTION OF IO/M SIGNAL IN THE 8085:**

It is a status signal. It is used to differentiate between memory locations and I/O operations.

**\***When this signal is low (IO/M = 0) it denotes the memory related operations.

**\***When this signal is high (IO/M = 1) it denotes an I/O operation.

**FLAGS:**

The flags are used to reflect the data conditions in the accumulator. The 8085 flags are S-Sign flag, Z-Zero flag, AC-Auxiliary carry flag, P-Parity flag, CY-Carry flag.

### Pin Diagram of 8085

### http://mvn.edu.in/mvnlms/pluginfile.php/1665/mod_book/chapter/341/Q4.PNG

                                     Fig: [pin diagram of 8085](http://mvn.edu.in/mvnlms/mod/assign/view.php?id=902)

* 8085 up is an 8-bit general purpose microprocessor capable of addressing 64Kb of memory.

The device has 40 pins,+5 V power supply, operate on 3 to 5 MHZ frequency single phase clock.

All the signals can be classified in 8085 up pin diagram into six groups –

1)  Address Bus:  in this 16 signals lines. These lines are splits into two segments -

a) A15-A8– unidirectional and used for the higher order address (MSB).

b)AD7-AD0 – Dual purpose such as data bus as well as lower address data bus(LSB).

**2)**Control and status signals: these signals are used to identify the nature of operation.

**Three control signals that are-**

**RD** – it is a active low signal. Which indicate that the selected IO or Memory device is to be read and data is available on the data bus.

**WR**-it is a active low signal which indicate that the data on the data bus are to be written into a selected memory or IO location.

**ALE**- it is a +ve going pulse generated everytime the 8085 begins an operation (machine cycle): which indicate that the bits on AD7-AD0are address bits.

**Three status signals that are** –

**IO/M-** this is a status signal used to differentiate between IO and Memory operations.when it is hign then IO operation and When it is low then Memory operation.

**S1 and S0-** status signals,similar to IO/M,can identify various operations.that are rarely used in the systems.

**3) Power supply:**

VCC : +5 V

VSS : Ground

**4) Clock Frequency:**

X1, X2**:** A crystal (RC,LC N/W) is connected at these two pins. this frequency is internally divided by 2.

CLK OUT: clock output this signal can be used as the system clock for the other devices.

**5)** **Externally initiated signals: In this**

**Five interrupt signals:** TRAP, RST 7.5, RST 6.5, RST 5.5 ,INTR.

INTA: interrupt acknowledge

RESET IN: It is a active low signal. When active program counter is set to zero.

RESET OUT: This signal indicates that the MPU is being reset, the signal can be used to reset other devices.

READY: If ready is high during a read or write cycle, it indicates that the memory or peripheral is ready to send or receive data. If ready is low, the CPU will wait for ready to go high before completing the read or write cycle.

HOLD: this signal indicate that another master is requesting the use of the address and data buses.

HLDA: HOLD Acknowledge indicates that the CPU has received the Hold request and that it will relinquish the buses I the next clock cycle. HLDA goes low after the HOLD request is removed. The CPU   takes the buses one half clock cycle HLDA goes low.

**6) Serial I/O ports:**

 SOD: serial output data line. The output SOD is set or reset as specified by the SIM instruction.

 SID: Serial input data line, the data on this line is loaded into accumulator whenever a RIM instruction is executed.

In this data bits are sent over a single line one bit at a time.

For ex: Transmission over phone lines.

**Three control signals that are-**

**RD** – it is a active low signal. Which indicate that the selected IO or Memory device is to be read and data is available on the data bus.

**WR**-it is a active low signal which indicate that the data on the data bus are to be written into a selected memory or IO location.

**ALE**- it is a +ve going pulse generated everytime the 8085 begins an operation (machine cycle): which indicate that the bits on AD7-AD0 are address bits.

**Three status signals that are** –

**IO/M-** this is a status signal used to differentiate between IO and Memory operations.when it is hign then IO operation and When it is low then Memory operation.

**S1 and S0-** status signals,similar to IO/M,can identify various operations.that are rarely used in the systems.

### Instruction Set

An instruction is a binary pattern designed inside a microprocessor to perform a specific function. Each instruction is represented by 8 bit binary value.

Types of [instruction set](http://mvn.edu.in/mvnlms/mod/assign/view.php?id=904):

**1)** **Data transfer instructions:**

Instructions, which are used to transfer data from one register to another register, from memory to register or register to memory, come under this group. Examples are: MOV, MVI, LXI, LDA, STA etc. When an instruction of data transfer group is executed, data is transferred from the source to the destination without altering the contents of the source. For example, when MOV A, B is executed the content of the register B is copied into the register A, and the content of register B remains unaltered. Similarly, when LDA 2500 is executed the content of the memory location 2500 is loaded into the accumulator. But the content of the memory location 2500 remains unaltered.

#### EXAMPLES:

1. MOV r1, r2 (Move Data; Move the content of the one register to another).  [r1] <-- [r2]

2. MOV r, m (Move the content of memory register). r <-- [M]

3. MOV M, r. (Move the content of register to memory). M <-- [r]

4. MVI r, data. (Move immediate data to register). [r] <-- data.

5. MVI M, data. (Move immediate data to memory). M <-- data.

6. LXI rp, data 16. (Load register pair immediate). [rp] <-- data 16 bits, [rh] <-- 8 LSBs of data.

7. LDA addr. (Load Accumulator direct). [A] <-- [addr].

8. STA addr. (Store accumulator direct). [addr] <-- [A].

9. LHLD addr. (Load H-L pair direct). [L] <-- [addr], [H] <-- [addr+1].

10.SHLD addr. (Store H-L pair direct) [addr] <-- [L], [addr+1] <-- [H].

11.LDAX rp. (LOAD accumulator indirect) [A] <-- [[rp]]

12.STAX rp. (Store accumulator indirect) [[rp]] <-- [A].

13.XCHG. (Exchange the contents of H-L with D-E pair) [H-L] <-->  [D-E].

**2)** **Arithmatic instructions**:

The instructions of this group perform arithmetic operations such as addition, subtraction; increment or decrement of the content of a register or memory.

**Examples:**

1). ADD r. (Add register to accumulator) [A] <-- [A] + [r].

2) .ADD M. (Add memory to accumulator) [A] <-- [A] + [[H-L]].

3).ADC r. (Add register with carry to accumulator). [A] <-- [A] + [r] + [CS].

4). ADC M. (Add memory with carry to accumulator) [A] <-- [A] + [[H-L]] [CS].

5) .ADI data (Add immediate data to accumulator) [A] <-- [A] + data.

6) .ACI data (Add with carry immediate data to accumulator). [A] <-- [A] + data + [CS].

7).DAD rp. (Add register paid to H-L pair). [H-L] <-- [H-L] + [rp].

8).SUB r. (Subtract register from accumulator). [A] <-- [A] – [r].

9).SUB M. (Subtract memory from accumulator). [A] <-- [A] – [[H-L]].

10).SBB r. (Subtract register from accumulator with borrow). [A] <-- [A] – [r] – [CS].

11).SBB M. (Subtract memory from accumulator with borrow). [A] <-- [A] – [[H-L]] – [CS].

12).SUI data. (Subtract immediate data from accumulator) [A] <-- [A] – data.

13).SBI data. (Subtract immediate data from accumulator with borrow). [A] <-- [A] – data – [CS].

14).INR r (Increment register content) [r] <-- [r] +1.

15). INR M. (Increment memory content) [[H-L]] <-- [[H-L]] + 1.

16).DCR r. (Decrement register content). [r] <-- [r] – 1.

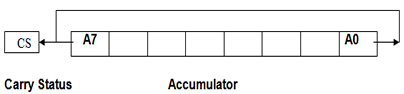
17).DCR M. (Decrement memory content) [[H-L]] <-- [[H-L]] – 1.

18).INX rp. (Increment register pair) [rp] <-- [rp] – 1.

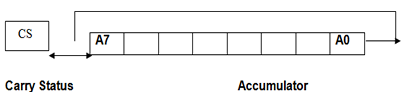
19).DCX rp (Decrement register pair) [rp] <-- [rp] -1.

20).DAA (Decimal adjust accumulator) .

**3) Logical instructions cont..:**

16).RLC (Rotate accumulator left) [An+1] <-- [An], [A0] <-- [A7],[CS] <-- [A7]**.**

The content of the accumulator is rotated left by one bit. The seventh bit of the accumulator is moved to carry bit as well as to the zero bit of the accumulator. Only CS flag is affected.

17).RRC. (Rotate accumulator right) [A7] <-- [A0], [CS] <-- [A0], [An] <-- [An+1].

The content of the accumulator is rotated right by one bit. The zero bit of the accumulator is moved to the seventh bit as well as to carry bit. Only CS flag is affected.

18). RAL. (Rotate accumulator left through carry) [An+1] <-- [An], [CS] <-- [A7], [A0] <-- [CS].

19).RAR. (Rotate accumulator right through carry) [An] <-- [An+1], [CS] <-- [A0], [A7] <-- [CS].

**4) Branching Instructions:**

This group includes the instructions for conditional and unconditional jump, subroutine call and return, and restart.

**Examples:**

1. MP addr (label). (Unconditional jump: jump to the instruction specified by the address). [PC] <-- Label.
2. Conditional Jump addr (label): After the execution of the conditional jump instruction the program jumps to the instruction specified by the address (label) if the specified condition is fulfilled. The program proceeds further in the normal sequence if the specified condition is not fulfilled. If the condition is true and program jumps to the specified label, the execution of a conditional jump takes 3 machine cycles: 10 states. If condition is not true, only 2 machine cycles; 7 states are required for the execution of the instruction.
   1. **JZ** addr (label). (Jump if the result is zero)
   2. **JNZ** addr (label) (Jump if the result is not zero)
   3. **JC** addr (label). (Jump if there is a carry)
   4. **JNC** addr (label). (Jump if there is no carry)
   5. **JP** addr (label). (Jump if the result is plus)
   6. **JM** addr (label). (Jump if the result is minus)
   7. **JPE** addr (label) (Jump if even parity)
   8. **JPO** addr (label) (Jump if odd parity)
3. CALL addr (label) (Unconditional CALL: call the subroutine identified by the operand)

CALL instruction is used to call a subroutine. Before the control is transferred to the subroutine, the address of the next instruction of the main program is saved in the stack. The content of the stack pointer is decremented by two to indicate the new stack top. Then the program jumps to subroutine starting at address specified by the label.

1. RET (Return from subroutine).
2. RST n (Restart) Restart is a one-word CALL instruction. The content of the program counter is saved in the stack. The program jumps to the instruction starting at restart location.

**5). Stack,I/O and Machine control instructions:**

1. N port-address. (Input to accumulator from I/O port) [A] <-- [Port]
2. OUT port-address (Output from accumulator to I/O port) [Port] <-- [A]
3. PUSH rp (Push the content of register pair to stack)
4. PUSH PSW (PUSH Processor Status Word)
5. POP rp (Pop the content of register pair, which was saved, from the stack)
6. POP PSW (Pop Processor Status Word)
7. HLT (Halt)
8. XTHL (Exchange stack-top with H-L)
9. SPHL (Move the contents of H-L pair to stack pointer)
10. EI (Enable Interrupts)
11. DI (Disable Interrupts)
12. SIM (Set Interrupt Masks)
13. RIM (Read Interrupt Masks)
14. NOP (No Operation).

# Interrupts In 8085

Interrupt is a mechanism by which an I/O or an instruction can suspend the normal execution of processor and get itself serviced. Generally, a particular task is assigned to that interrupt signal. In the microprocessor based system the interrupts are used for data transfer between the peripheral devices and the microprocessor.

### Interrupt Service Routine(ISR):A small program or a routine that when executed services the corresponding interrupting source is called as an ISR.

### Maskable/Non-Maskable Interrupt:An interrupt that can be disabled by writing some instruction is known as Maskable Interrupt otherwise it is called Non-Maskable Interrupt.

**There are 6 pins available in 8085 for interrupt:**

1. TRAP
2. RST 7.5
3. RST6.5
4. RST5.5
5. INTR
6. INTA

### Execution of Interrupts

When there is an interrupt requests to the Microprocessor then after accepting the interrupts Microprocessor send the INTA (active low) signal to the peripheral. The vectored address of particular interrupt is stored in program counter. The processor executes an interrupt service routine (ISR) addressed in program counter. 

There are two types of interrupts used in 8085 Microprocessor:

1. Hardware Interrupts
2. Software Interrupts

### Software Interrupts

A software interrupts is a particular instructions that can be inserted into the desired location in the program. There are eight Software interrupts in 8085 Microprocessor. From RST0 to RST7.

1. RST0
2. RST1
3. RST2
4. RST3
5. RST4
6. RST5
7. RST6
8. RST7

They allow the microprocessor to transfer program control from the main program to the subroutine program. After completing the subroutine program, the program control returns back to the main program.   
We can calculate the vector address of these interrupts using the formula given below:

Vector Address = Interrupt Number \* 8

For Example:

RST2: vector address=2\*8 = 16

RST1: vector address=1\*8 = 08

RST3: vector address=3\*8 = 24

Vector address table for the software interrupts:

|  |  |
| --- | --- |
| **Interrupt** | **Vector Address** |
| RST0 RST1 | 0000H 0008H |
| RST2 RST3 | 0010H 0018H |
| RST4 RST5 | 0020H 0028H |
| RST6 RST7 | 0030H 0038H |

**Hardware Interrupt**

I have already discussed that there are 6 interrupt pins in the microprocessor used as Hardware Interrrupts given below:

1. TRAP
2. RST7.5
3. RST6.5
4. RST5.5
5. INTR

INTA is not an interrupt. INTA is used by the Microprocessor for sending the acknowledgement.TRAP has highest priority and RST7.5 has second highest priority and so on.   
  
The Vector address of these interrupts are given below:

|  |  |
| --- | --- |
| **Interrupt** | **Vector Address** |
| RST7.5 | 003CH |
| RST6.5 | 0034H |
| RST5.5 | 002CH |
| TRAP | 0024H |

### TRAP

It is non maskable edge and level triggered interrupt. TRAP has the highest priority and vectores interrupt. Edge and level triggered means that the TRAP must go high and remain high until it is acknowledged. In case of sudden power failure, it executes a ISR and send the data from main memory to backup memory.  TRAP can not be masked but it can be delayed using HOLD signal. This interrupt transfers the microprocessor's control to location 0024H. TRAP interrupts can only be masked by reseting the microprocessor. There is no other way to mask it.   
  
**RST7.5**

It has the second highest priority. It is maskable and edge level triggered interrupt. The vector address of this interrupt is 003CH. Edge sensitive means input goes high and no need to maintain high state until it is recognized. It can also be reset or masked by reseting microprocessor. It can also be resetted by DI instruction. 

### RST6.5 and RST5.5

These are level triggered and maskable interrupts. When RST6.5 pin is at logic 1, INTE flip-flop is set. RST 6.5 has third highest priority and RST 5.5 has fourth highest priority.  
It can be masked by giving DI and SIM instructions or by reseting microprocessor.   
**INTR**

It is level triggered and maskable interrupt. The following sequence of events occurs when INTR signal goes high:

1. The 8085 checks the status of INTR signal during execution of each instruction.
2. If INTR signal is high, then 8085 complete its current instruction and sends active low interrupt acknowledge signal, if the interrupt is enabled.
3. On receiving the instruction, the 8085 save the address of next instruction on stack and execute received instruction.

It has the lowest priority. It can be disabled by reseting the microprocessor or by DI and SIM instruction.

## [Addressing Modes in 8085](http://microprocessorforyou.blogspot.in/2012/06/addressing-modes-in-8085.html)

There are five addressing modes in 8085.

**1. Immediate Addressing Mode: -** An immediate is transferred directly to the register.

**Eg:-** MVI A, 30H (30H is copied into the register A)  
          MVI B,40H(40H is copied into the register B).

**2. Register Addressing Mode: -** Data is copied from one register to another register.

**Eg: -** MOV B, A (the content of A is copied into the register B)

        MOV A, C (the content of C is copied into the register A).

**3. Direct Addressing Mode: -** Data is directly copied from the given address to the register.

**Eg: -** LDA 3000H (The content at the location 3000H is copied to the register A).

**4. Indirect Addressing Mode: -** The data is transferred from the address pointed by the data in a register to other register.

**Eg: -** MOV A, M (data is transferred from the memory location pointed by the regiser to the accumulator).

**5.Implied Addressing Mode: -** This mode doesn't require any operand. The data is specified by opcode itself.

**Eg: -** RAL,CMP

# Timing Diagrams of 8085

It is one of the best way to understand to process of micro-processor/controller. With the help of timing diagram we can understand the working of any system, step by step working of each instruction and its execution, etc.

It is the graphical representation of process in steps with respect to time. The timing diagram represents the clock cycle and duration, delay, content of address bus and data bus, type of operation ie. Read/write/status signals.

**Important terms related to timing diagrams:**

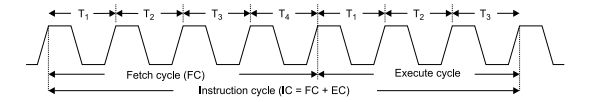
**1. Instruction cycle:** this term is defined as the number of steps required by the cpu to complete the entire process ie. Fetching and execution of one instruction. The fetch and execute cycles are carried out in synchronization with the clock.

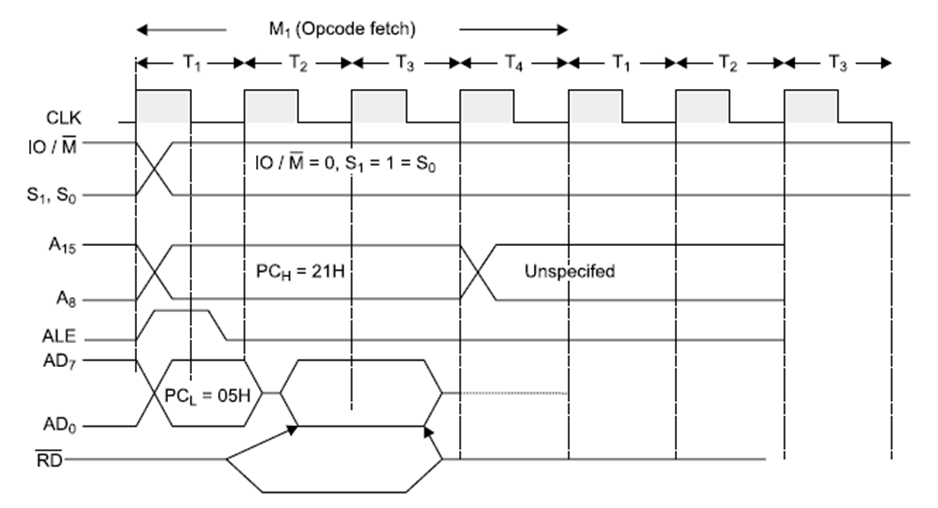
**2. Machine cycle:** It is the time required by the microprocessor to complete the operation of accessing the memory devices or I/O devices. In machine cycle various operations like opcode fetch, memory read, memory write, I/O read, I/O write are performed.

**3. T-state:** Each clock cycle is called as T-states.

**Rules to identify number of machine cycles in an instruction:**

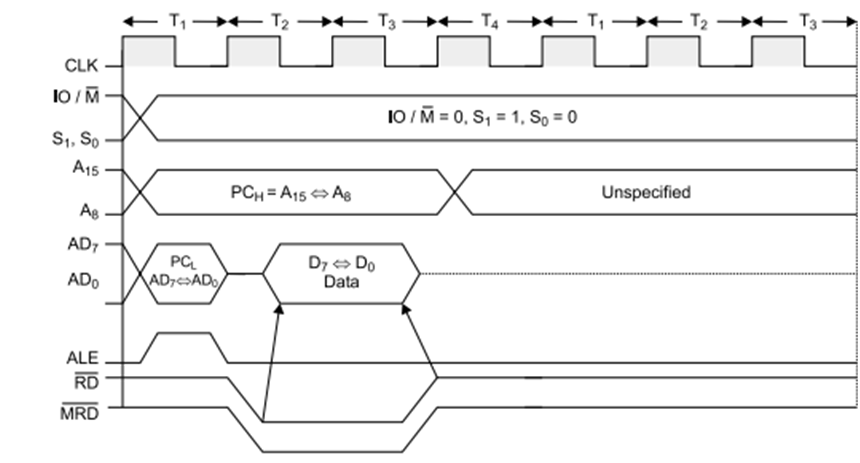
1. If an addressing mode is direct, immediate or implicit then No. of machine cycles = No. of bytes.  
  
2. If the addressing mode is indirect then No. of machine cycles = No. of bytes + 1. Add +1 to the No. of machine cycles if it is memory read/write operation.  
  
3. If the operand is 8-bit or 16-bit address then, No. of machine cycles = No. of bytes +1.  
  
4. These rules are applicable to 80% of the instructions of 8085.

**Timing Diagram:**  
Where, Instruction cycle= Fetch Cycle(FC) + Executecycle(EC).  
  
  
**Opcode fetch:**

 The microprocessor requires instructions to perform any particular action. In order to perform these actions microprocessor utilizes Opcode which is a part of an instruction which provides detail(ie. Which operation Âµp needs to perform) to microprocessor.*Fig: Opcode fetch timing diagram*  
**Operation:**

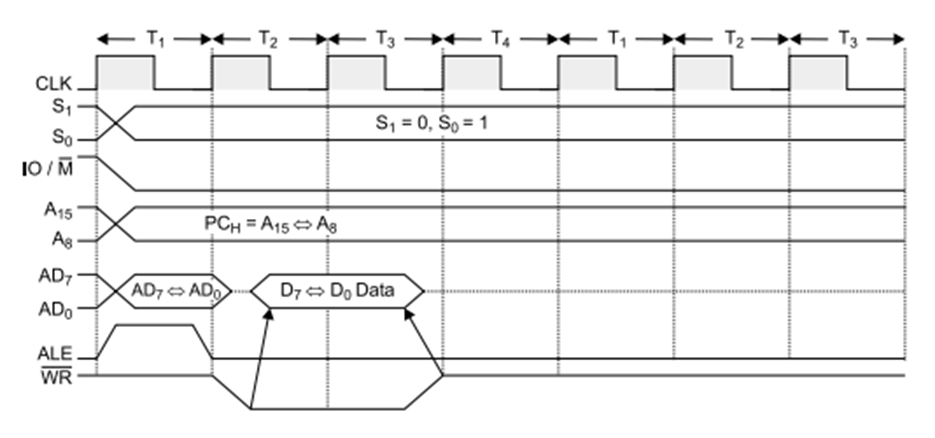
 During T1 state, microprocessor uses IO/M(bar), S0, S1 signals are used to instruct microprocessor to fetch opcode.  
 Thus when IO/M(bar)=0, S0=S1= 1, it indicates opcode fetch operation.  
 During this operation 8085 transmits 16-bit address and also uses ALE signal for address latching.  
 At T2 state microprocessor uses read signal and make data ready from that memory location to read opcode from memory and at the same time program counter increments by 1 and points next instruction to be fetched.  
 In this state microprocessor also checks READY input signal, if this pin is at low logic level ie. '0' then microprocessor adds wait state immediately between T2 and T3.  
 At T3, microprocessor reads opcode and store it into instruction register to decode it further.  
 During T4 microprocessor performs internal operation like decoding opcode and providing necessary actions.  
 The opcode is decoded to know whether T5 or T6 states are required, if they are not required then Âµp performs next operation.

**Read and write timing diagram for memory and I/O Operation**

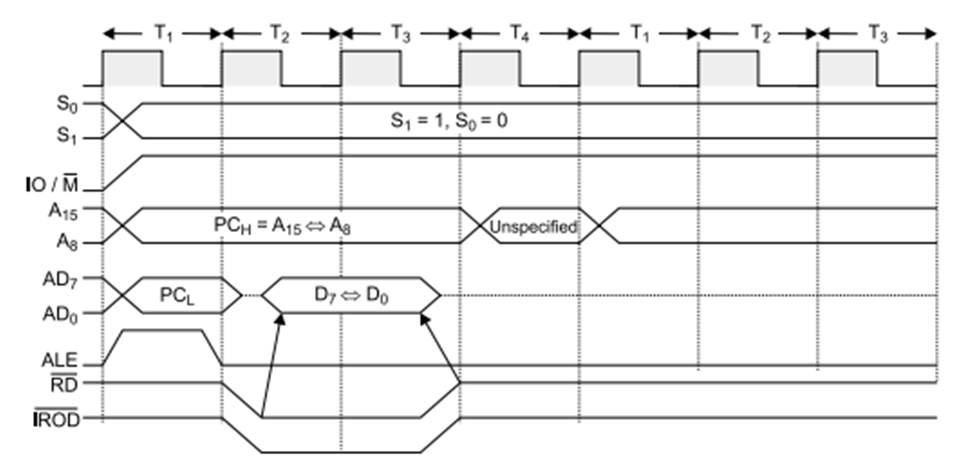
**Memory Read:**<>Figure: Memory read timing diagram  
  
**Operation:**

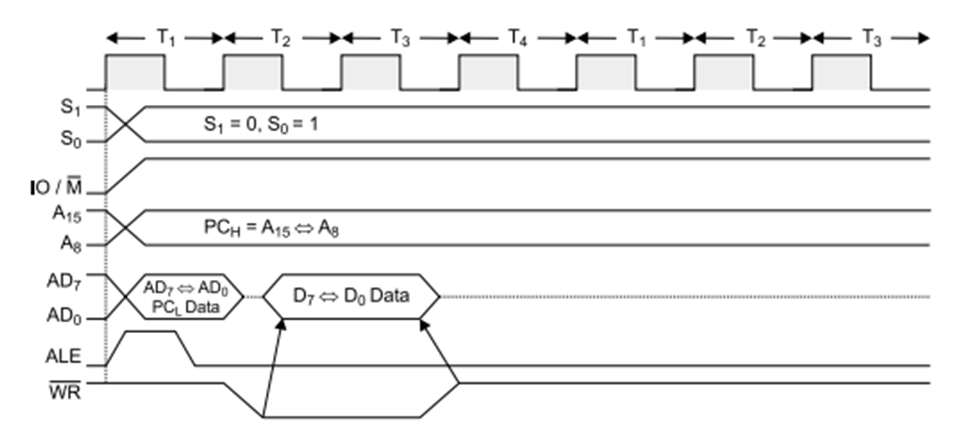
 It is used to fetch one byte from the memory.  
 It requires 3 T-States.  
 It can be used to fetch operand or data from the memory.  
 During T1, A8-A15 contains higher byte of address. At the same time ALE is high. Therefore Lower byte of address A0-A7 is selected from AD0-AD7.  
 Since it is memory ready operation, IO/M(bar) goes low.  
 During T2 ALE goes low, RD(bar) goes low. Address is removed from AD0-AD7 and data D0-D7 appears on AD0-AD7.

 During T3, Data remains on AD0-AD7 till RD(bar) is at low signal.

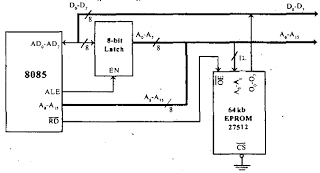
**Memory Write:***Figure: Memory write timing diagram*  
  
**Operation:**

It is used to send one byte into memory.  
 It requires 3 T-States.  
 During T1, ALE is high and contains lower address A0-A7 from AD0-AD7.  
 A8-A15 contains higher byte of address.  
 As it is memory operation, IO/M(bar) goes low.  
 During T2, ALE goes low, WR(bar) goes low and Address is removed from AD0-AD7 and then data appears on AD0-AD7.  
 Data remains on AD0-AD7 till WR(bar) is low.

**IO Read:***Figure: I/O read timing diagram*  
  
**Operation:**   
  
1.It is used to fetch one byte from an IO port.  
2.It requires 3 T-States.  
3.During T1, The Lower Byte of IO address is duplicated into higher order address bus A8-A15.  
4.ALE is high and AD0-AD7 contains address of IO device.  
5.IO/M (bar) goes high as it is an IO operation.  
6.During T2, ALE goes low, RD (bar) goes low and data appears on AD0-AD7 as input from IO device.  
7.During T3 Data remains on AD0-AD7 till RD(bar) is low.

**IO Write:**Figure:I/O write timing diagram  
  
**Operation:**  
 It is used to writ one byte into IO device.  
 It requires 3 T-States.  
 During T1, the lower byte of address is duplicated into higher order address bus A8-A15.  
 ALE is high and A0-A7 address is selected from AD0-AD7.  
 As it is an IO operation IO/M (bar) goes low.  
 During T2, ALE goes low, WR (bar) goes low and data appears on AD0-AD7 to write data into IO device.  
 During T3, Data remains on AD0-AD7 till WR(bar) is low.

## Memory Interfacing in 8085 Microprocessor.

* **Memory Interfacing:-**As we know that any system which process digital data needs the facility for storing the data. Interfacing is a technique to be used for connecting the Microprocessor to Memory.Now a day’s Semiconductor memories are used for storing purpose. There are some of the advantages of the  semiconductor memory.  
    
  Small size  
  High speed  
  Better reliability  
  Low cost  
  Generally, RAM or ROM is used for memory interfacing.  
    
    
  **Memory:-**A memory is a digital IC which stores the data in binary form.  
   **Memory Size:-**The number of location and number of bits per word will vary from memory to memory. For example, If a particular memory chip is capable of storing M words with each word having N-bits. Then the size of the memory will be M× N.  
  **Interfacing a ROM memory of 4096\*8 with 8085 Microprocessor:-**  
  Given memory size = 4096 \* 8  
  4096 =2^12.  
  So 12 lines will be used for interfacing. A0 to A11  
  In this system A0 to A11 lines of Microprocessor will be connected to the address lines of the memory. and D0 to D7 of the 8085 microprocessor will be connected to the data bus of the memory.As we know that the it is EPROM, so only RD pin is connected to the microprocessor. There is not the facility for writing data.  
    
  In case if you are using RAM then you have to connect one more pin for writing operation.  
  Memory Interfacing in 8085 Microprocessor.
* Memory Interfacing:-As we know that any system which process digital data needs the facility for storing the data. Interfacing is a technique to be used for connecting the Microprocessor to Memory.Now a days Semiconductor memories are used for storing purpose. There are some of the advantages of the semiconductor memory.  
  Small size  
  High speed  
  Better reliability  
  Low cost  
  Generally, RAM or ROM is used for memory interfacing.  
    
  Memory:-A memory is a digital IC which stores the data in binary form.  
  Memory Size:-The number of location and number of bits per word will vary from memory to memory. For example, If a particular memory chip is capable of storing M words with each word having N-bits. Then the size of the memory will be M× N.  
  Interfacing a ROM memory of 4096\*8 with 8085 Microprocessor:-  
    
  Given memory size = 4096 \* 8  
  4096 =2^12.  
  So 12 lines will be used for interfacing. A0 to A11  
    
  In this system A0 to A11 lines of Microprocessor will be connected to the address lines of the memory. and D0 to D7 of the 8085 microprocessor will be connected to the data bus of the memory.As we know that the it is EPROM, so only RD pin is connected to the microprocessor. There is not the facility for writing data.In case if you are using RAM then you have to connect one more pin for writing operation.  
  [](http://2.bp.blogspot.com/-WwTY8S7oa1s/TudxEKJPn8I/AAAAAAAAAA8/8ACNoFPQWTY/s1600/Mem-Interfacing-Pic8-pic17.png)
* ASSEMBLY LANGUAGE PROGRAMMING

Count number of one’s in a number

Statement: Write a program to count number of l’s in the contents of D register and store the count in the B register.

Sample problem

(2200H) = 04

(2201H) = 34H

(2202H) = A9H

(2203H) = 78H

(2204H) = 56H

Result = (2202H) = A9H

1. MVI B, 00H
2. MVI C, 08H
3. MOV A, D
4. BACK: RAR
5. JNC SKIP
6. INR B
7. SKIP: DCR C
8. JNZ BACK
9. HLT

## Arrange in ascending order

Statement: Write a program to sort given 10 numbers from memory location 2200H in the ascending order.

1. MVI B, 09 :"Initialize counter"
2. START :"LXI H, 2200H: Initialize memory pointer"
3. MVI C, 09H :"Initialize counter 2"
4. BACK: MOV A, M :"Get the number"
5. INX H :"Increment memory pointer"
6. CMP M :"Compare number with next number"
7. JC SKIP :"If less, don’t interchange"
8. JZ SKIP :"If equal, don’t interchange"
9. MOV D, M
10. MOV M, A
11. DCX H
12. MOV M, D
13. INX H :"Interchange two numbers"
14. SKIP:DCR C :"Decrement counter 2"
15. JNZ BACK :"If not zero, repeat"
16. DCR B :"Decrement counter 1"
17. JNZ START
18. HLT :"Terminate program execution"

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**UNIT- II**

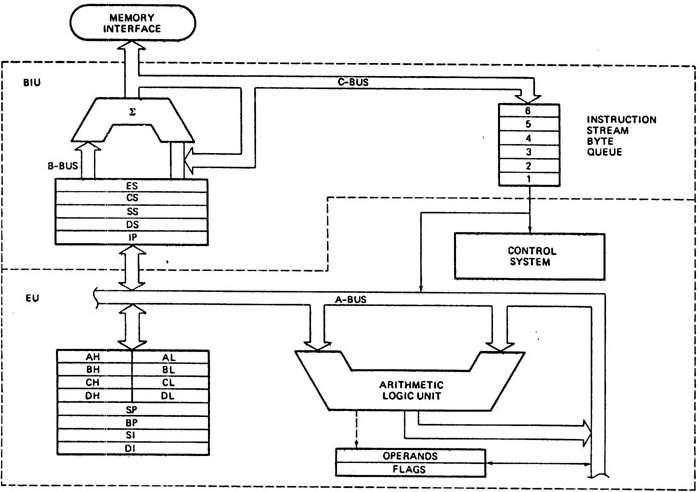
**UNIT-2:8086 Microprocessor:** 8086 Architecture, difference between 8085 and 8086 architecture, generation of physical address, PIN diagram of 8086, Minimum Mode and Maximum mode, Bus cycle, Memory Organization, Memory Interfacing, Addressing Modes, Assembler Directives, Instruction set of 8086, Assembly Language Programming, Hardware and Software Interrupts.

**[T2][No. of hrs. :12]**

**Block Diagram of Intel 8086**

The 8086 CPU is divided into two independent functional units:

1. Bus Interface Unit (BIU)
2. Execution Unit (EU)



**Fig. 1: Block Diagram of Intel 8086**

**Features of 8086 Microprocessor:**

1. Intel 8086 was launched in 1978.
2. It was the first 16-bit microprocessor.
3. This microprocessor had major improvement over the execution speed of 8085.
4. It is available as 40-pin Dual-Inline-Package (DIP).
5. It is available in three versions:
   1. 8086 (5 MHz)
   2. 8086-2 (8 MHz)
   3. 8086-1 (10 MHz)
6. It consists of 29,000 transistors.

**Bus Interface Unit (BIU)**

The function of BIU is to:

* Fetch the instruction or data from memory.
* Write the data to memory.
* Write the data to the port.
* Read data from the port.

**Instruction Queue**

1. To increase the execution speed, BIU fetches as many as six instruction bytes ahead to time from memory.
2. All six bytes are then held in first in first out 6 byte register called instruction queue.
3. Then all bytes have to be given to EU one by one.
4. This pre fetching operation of BIU may be in parallel with execution operation of EU, which improves the speed execution of the instruction.

**Execution Unit (EU)**

The functions of execution unit are:

* To tell BIU where to fetch the instructions or data from.
* To decode the instructions.
* To execute the instructions.

The EU contains the control circuitry to perform various internal operations. A decoder in EU decodes the instruction fetched memory to generate different internal or external control signals required to perform the operation. EU has 16-bit ALU, which can perform arithmetic and logical operations on 8-bit as well as 16-bit.

**General Purpose Registers of 8086**

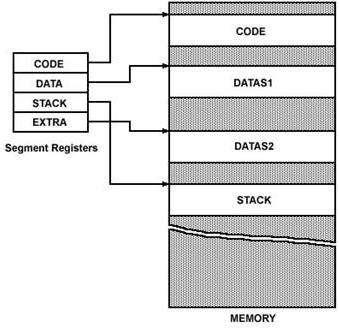
These registers can be used as 8-bit registers individually or can be used as 16-bit in pair to have AX,

BX, CX, and DX.

1. **AX Register:** AX register is also known as accumulator register that stores operands forarithmetic operation like divided, rotate.
2. **BX Register:** This register is mainly used as a base register. It holds the starting baselocation of a memory region within a data segment.
3. **CX Register:** It is defined as a counter. It is primarily used in loop instruction to store loopcounter.
4. **DX Register:** DX register is used to contain I/O port address for I/O instruction.

**Segment Registers**

Additional registers called segment registers generate memory address when combined with other in the microprocessor. In 8086 microprocessor, memory is divided into 4 segments as follow:



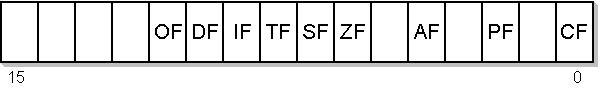
**Fig. 2: Memory Segments of 8086**

1. **Code Segment (CS):** The CS register is used for addressing a memory location in the CodeSegment of the memory, where the executable program is stored.
2. **Data Segment (DS):** The DS contains most data used by program. Data are accessed in theData Segment by an offset address or the content of other register that holds the offset address.
3. **Stack Segment (SS):** SS defined the area of memory used for the stack.

1. **Extra Segment (ES):** ES is additional data segment that is used by some of the string to holdthe destination data.

**Flag Registers of 8086**

Flag register in EU is of 16-bit and is shown in fig. 3:



**Fig. 3: Flag Register of 8086**

Flags Register determines the current state of the processor. They are modified automatically by CPU after mathematical operations, this allows to determine the type of the result, and to determine conditions to transfer control to other parts of the program. 8086 has 9 flags and they are divided into two categories:

1. Conditional Flags
2. Control Flags

**Conditional Flags**

Conditional flags represent result of last arithmetic or logical instruction executed. Conditional flags are as follows:

* **Carry Flag (CF):** This flag indicates an overflow condition for unsigned integer arithmetic.It is also used in multiple-precision arithmetic.
* **Auxiliary Flag (AF):** If an operation performed in ALU generates a carry/barrow fromlower nibble (i.e. D0 – D3) to upper nibble (i.e. D4 – D7), the AF flag is set i.e. carry given by D3 bit to D4 is AF flag. This is not a general-purpose flag, it is used internally by the

processor to perform Binary to BCD conversion.

* **Parity Flag (PF):** This flag is used to indicate the parity of result. If lower order 8-bits of theresult contains even number of 1’s, the Parity Flag is set and for odd number of 1’s, the Parity Flag is reset.
* **Zero Flag (ZF):** It is set; if the result of arithmetic or logical operation is zero else it is reset.
* **Sign Flag (SF):** In sign magnitude format the sign of number is indicated by MSB bit. If theresult of operation is negative, sign flag is set.

* **Overflow Flag (OF):** It occurs when signed numbers are added or subtracted. An OFindicates that the result has exceeded the capacity of machine.

**Control Flags**

Control flags are set or reset deliberately to control the operations of the execution unit. Control flags

are as follows:

1. **Trap Flag (TP):** 
   1. It is used for single step control.
   2. It allows user to execute one instruction of a program at a time for debugging.
   3. When trap flag is set, program can be run in single step mode.
2. **Interrupt Flag (IF):** 
   1. It is an interrupt enable/disable flag.
   2. If it is set, the maskable interrupt of 8086 is enabled and if it is reset, the interrupt is disabled.
   3. It can be set by executing instruction sit and can be cleared by executing CLI instruction.
3. **Direction Flag (DF):** 
   1. It is used in string operation.
   2. If it is set, string bytes are accessed from higher memory address to lower memory address.
   3. When it is reset, the string bytes are accessed from lower memory address to higher memory address.

**Difference Between 8085 and 8086**

**8085 Microprocessor:**

* is 8-bit Microprocessor.
* has 16-bit address line.
* has 8-bit data bus.
* The speed is 3 MHz.
* has 5 flags.
* does not support pipe-lining.
* does not support memory segmentation.
* has 6500 transistors.
* has no minimum or maximum mode.
* only 64 KB memory is used together.

**8086 Microprocessor:**

* is 16-bit Microprocessor.
* has 20-bit address line.
* has 16-bit data bus.
* The speed can vary between 5,8, and 10MHz for three different microprocessor.
* has 9 flags.
* support pipe-lining.
* support memory segmentation.
* has 29000 transistors.

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

****

**Bus Interface Unit (BIU):**

– The BIU performs all bus operations such as instruction fetching, reading and writing operands for memory and calculating the addresses of the memory operands.

– The instruction bytes are transferred to the instruction queue.

– It provides a full 16 bit bidirectional data bus and 20 bit address bus.

– The bus interface unit is responsible for performing all external bus operations.

***Specifically it has the following functions***:

– Instruction fetch , Instruction queuing, Operand fetch and storage, Address calculation relocation and Bus control.

– The BIU uses a mechanism known as an instruction queue to implement a ***pipeline architecture*.**

– This queue permits prefetch of up to six bytes of instruction code. Whenever the queue of the BIU is not full and it has room for at least two more bytes and at the same time EU is not requesting it to read or write operands from memory, the BIU is free to look ahead in the program by prefetching the next sequential instruction.

– These prefetching instructions are held in its FIFO queue. With its 16 bit data bus, the BIU fetches two instruction bytes in a single memory cycle.

– After a byte is loaded at the input end of the queue, it automatically shifts up through the FIFO to the empty location nearest the output.

– The EU accesses the queue from the output end. It reads one instruction byte after the other from the output of the queue. If the queue is full and the EU is not requesting access to operand in memory.

– These intervals of no bus activity, which may occur between bus cycles, are known as ***Idle state****.*

– If the BIU is already in the process of fetching an instruction when the EU request it to read or write operands from memory or I/O, the BIU first completes the instruction fetch bus cycle before initiating the operand read / write cycle.

– The BIU also contains a dedicated **adder** which is used to generate the 20bit physical address that is output on the address bus. This address is formed by adding an appended 16 bit segment address and a 16 bit offset address.

– For example: The physical address of the next instruction to be fetched is formed by combining the current contents of the code segment CS register and the current contents of the instruction pointer IP register.

**EXECUTION UNIT (EU)**

– The Execution unit is responsible for decoding and executing all instructions.

– The EU extracts instructions from the top of the queue in the BIU, decodes them, generates operands if necessary, passes them to the BIU and requests it to perform the read or write bys cycles to memory or I/O and perform the operation specified by the instruction on the operands.

– During the execution of the instruction, the EU tests the status and control flags and updates them based on the results of executing the instruction.

– If the queue is empty, the EU waits for the next instruction byte to be fetched and shifted to top of the queue.

– When the EU executes a branch or jump instruction, it transfers control to a location corresponding to another set of sequential instructions.

– Whenever this happens, the BIU automatically resets the queue and then begins to fetch instructions from this new location to refill the queue

**The BIU contains the following registers**:

IP - the Instruction Pointer

CS - the Code Segment Register

DS - the Data Segment Register

SS - the Stack Segment Register

ES - the Extra Segment Register

The BIU fetches instructions using the CS and IP, written CS:IP, to contract the 20-bit address. Data is fetched using a segment register (usually the DS) and an effective address (EA) computed by the EU depending on the addressing mode.

**Internal Registers of 8086**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **AX** | | **AH** |  | **AL** | **Accumulator** |
|  | **BX** | | **BH** |  | **BL** | **Base Register** |
| **EU** | **CX** | | **CH** |  | **CL** | **Count Register** |
| **Registers** | **DX** | | **DH** |  | **DL** | **Data Register** |
|  |  |  |  | **SP** | | **Stack Pointer** |
|  |  |  |  | **BP** | | **Base Pointer** |
|  |  |  |  | **SI** | | **Source Index Register** |
|  |  |  |  | **DI** | | **Destination Index Register** |
|  |  |  |  |  | | **Flag Register** |
|  |  |  |  | **FR** | |
|  |  |  |  |  |  | **Code Segment Register** |
|  |  |  |  | **CS** | |
| **BIU** |  |  |  | **DS** | | **Data Segment Register** |
|  |  |  | **SS** | | **Stack Segment Register** |
| **Registers** |  |  |  |
|  |  |  | **ES** | | **Extra Segment Register** |
|  |  |  |  |
|  |  |  |  |  | | **Instruction Pointer** |
|  |  |  |  | **IP** | |
|  |  |  |  |  | |  |

* The 8086 has four groups of the user accessible internal registers.
* These are:

Instruction pointer(IP)

Four General purpose registers(AX,BX,CX,DX) Four pointer (SP,BP,SI,DI)

Four segment registers (CS,DS,SS,ES) Flag Register(FR)

The 8086 has a total of fourteen 16-bit registers including a 16 bit register called the ***status register***

***(flag register)***, with 9 of bits implemented for status and control flags.Most of the registers contain data/instruction offsets within 64 KB memory segment.There are four different 64 KB segments for instructions, stack, data and extra data. To specify where in 1 MB of processor addressable memory these 4 segments are located the processor uses four segment registers:

**Segment Registers**

1. **Code segment** (CS) is a 16-bit register containing address of 64 KB segment withprocessor instructions. The processor uses CS segment for all accesses to instructions referenced by instruction pointer (IP) register.
2. **Stack segment** (SS) is a 16-bit register containing address of 64KB segment withprogram stack. By default, the processor assumes that all data referenced by the stack pointer (SP) and base pointer (BP) registers is located in the stack segment. SS register can be changed directly using POP instruction.
3. **Data and Extra segment** (DS and ES) is a 16-bit register containing address of 64KBsegment with program data. By default, the processor assumes that all data referenced by general registers (AX, BX, CX, and DX) and index register (SI, DI) is located in the data and Extra segment.

**Data Registers**

1. **AX (Accumulator)**
   * It is consists of two 8-bit registers AL and AH, which can be combined together and used as a 16-bit register AX. AL in this case contains the low-order byte of the word, and AH contains the high-order byte. Accumulator can be used for I/O operations and string manipulation.
2. **BX (Base register)**
   * + It is consists of two 8-bit registers BL and BH, which can be combined together and used as a 16-bit register BX. BL in this case contains the low-order byte of the word, and BH contains the high-order byte.
     + BX register usually contains a offset for data segment.
3. **CX (Count register)**
   * It is consists of two 8-bit registers CL and CH, which can be combined together and used as a 16-bit register CX. When combined, CL register contains the low-order byte of the word, and CH contains the high-order byte.
   * Count register can be used in Loop, shift/rotate instructions and as a counter in string manipulation.
   * 8086 has the LOOP instruction which is used for conuter purpose when it is executed CX/CL is automatically decremented by 1.

*EX*

MOV CL, 05H

START NOP

LOOP START (here CL is automatically decremented by 1without DCR instruction.

1. **DX (Data register)**
   * It is consists of two 8-bit registers DL and DH, which can be combined together and used as a 16-bit register DX. When combined, DL register contains the low-order byte of the word, and DH contains the high-order byte.
   * DX can be used as a port number in I/O operations.
   * In integer 32-bit multiply and divide instruction the DX register contains high-order word of the initial or resulting number.

**Pointer register**

1. **Stack Pointer** (SP) is a 16-bit register is used to hold the offset address for stacksegment.
2. **Base Pointer** (BP) is a 16-bit register is used to hold the offset address for stacksegment.
   * 1. BP register is usually used for based, based indexed or register indirect addressing.
     2. The difference between SP and BP is that the SP is used internally to store the address in case of interrupt and the CALL instrn.
3. **Source Index** (SI) and **Destination Index** (DI )

These two 16-bit register is used to hold the offset address for DS and ES in case of string manipulation instrn.

* 1. SI is used for indexed, based indexed and register indirect addressing, as well as a source data addresses in string manipulation instructions.
     1. DI is used for indexed, based indexed and register indirect addressing, as well as a destination data addresses in string manipulation

instructions.

**Instruction Pointer** (IP)

It is a 16-bit register. It acts as a program counter and is used to hold the offset address for CS.

**A flag is** a 16-bit register containing 9 one bit flags.

1. **Overflow Flag** (OF)

This flag is set if an overflow occurs. i.e. if the result of a signed operation is large enough to be accommodated in a destination register.

1. **Direction Flag** (DF) –

This is used by string manipulation instructions. If this flag bit is ‘0’, the string is processed beginning from the low est 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***.***

1. **Interrupt-enable Flag** (IF) –

If this flag is set, the maskable interrupts are recognized by the CPU. Otherwise they are ignored. Setting this bit enables maskable interrupts.

1. **Single-step Flag** (TF) –

If this flag is set, the processor enters the single step execution mode. In other words, a trap interrupt is generated after execution of each instruction. The processor executes the current instruction and the control is transferred to the Trap interrupt service routine.

1. **Sign Flag** (SF) –

This flag is set when the result of any computation is negative. For signed computations, the sign flag equals the MSB of the result.

1. **Zero Flag** (ZF) - set if the result is zero.
2. **Auxiliary carry Flag** (AF) –

set if there was a carry from or borrow to bits 0-3 in the AL register.

1. **Parity Flag** (PF) –

set if parity (the number of "1" bits) in the low-order byte of the result is even.

1. **Carry Flag** (CF) –

This flag is set when there is a carry out of MSB in case of addition or a borrow in case of subtraction.be generated out of the most significant bit position. The carry flag, in this case, will be set to 1’. In case, no carry is generated, it will be ‘0.

**Segmented Memory**

**Reason for Segmented Memory:**

8086 has a 20-bit address bus. So it can address a maximum of 1MB of memory and each memory location is addressed by a 20 bit address.

To hold a 20-bit address there must be a 20-bit address register available within processor but 8086 only has 16-bit registers. So 20-bit address can’t be stored inside the 16-bit register. To avoid this problem segmented memory is used in 8086.

Total 1MB memory can be divided into some equal size segments each of having capacity 64 KB.

So max no of segments is 16. (1mb/64 kb=16)

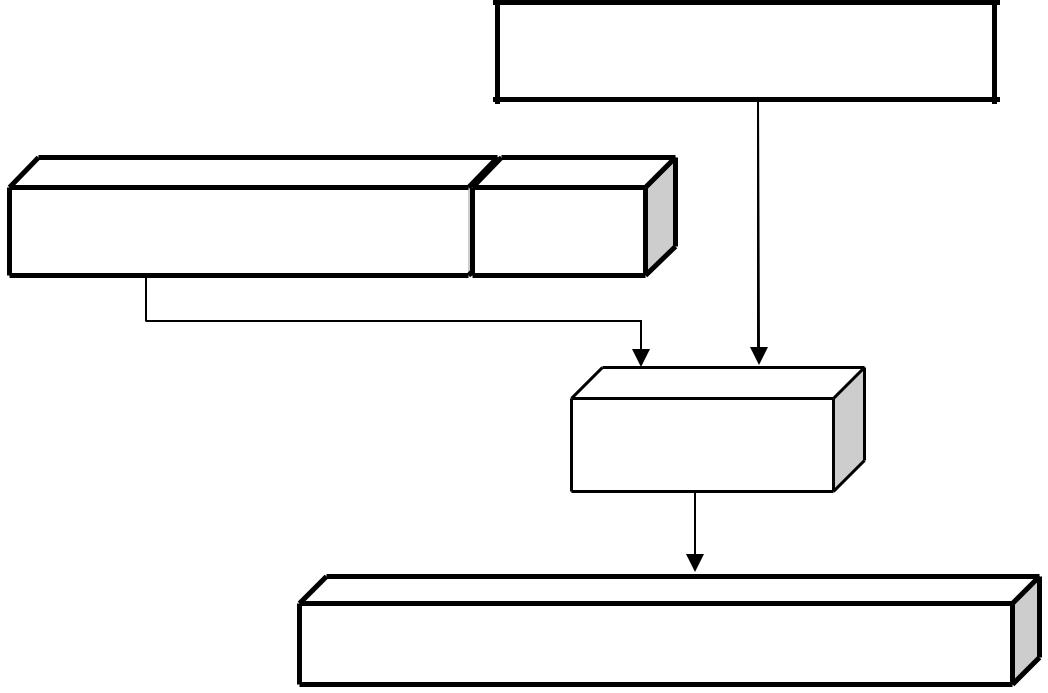
8086 can work with only four 64KB segments at a time within this 1MB range. Each location in a particular segment can be expressed by two addresses.

1. ***Segment Address (16 bit):*** It refers the starting address of a segment and it isfixed for whole of the segment.
2. ***Offset or Displacement Address (16 bit):*** It refers the individual location inthat segment and it is varied location wise.

By using these two addresses the 20 bit physical address can be calculated as below:

Physical address (20 bit) = [Segment Address (16 bit) \* 10]H + Offset Address(16 bit)

According to this formula segment address is multiplied by 10 and is added to offset. This is equivalent to shifting of segment register content towards left 4 times so that four zero are added to right side (MSB) of the segment address and added with the offset address to get the physical address which is 20 bit.



**Offset Address (16 bit)**

|  |  |
| --- | --- |
| **Segment Register (16** | **0000** |
| **bit)** |  |

**ADDER**

**Physical address (20 bit)**

EX:-

Given Segment Address=3578H , Offset Address =6676H

So Physical address = [Segment Address \* 10]H + Offset Address

* [357 8 \* 10]H + 6676H
* 357 80+6676
* 3BD F6H

**Types of Segments**

There are four types of memory segments defined in 8086:

* Code segment(CS)
* Data segment (DS)
* Stack segment(SS)
* Extra segment(ES)

**Code segment (CS):** This segment is used to store code/program instructions.

**Data and Extra segment (DS& ES):** This segment is used to store data used in the program. **Stack segment (SS):** This segment is used to store the stack contents.

**Types of Segments R egisters:**

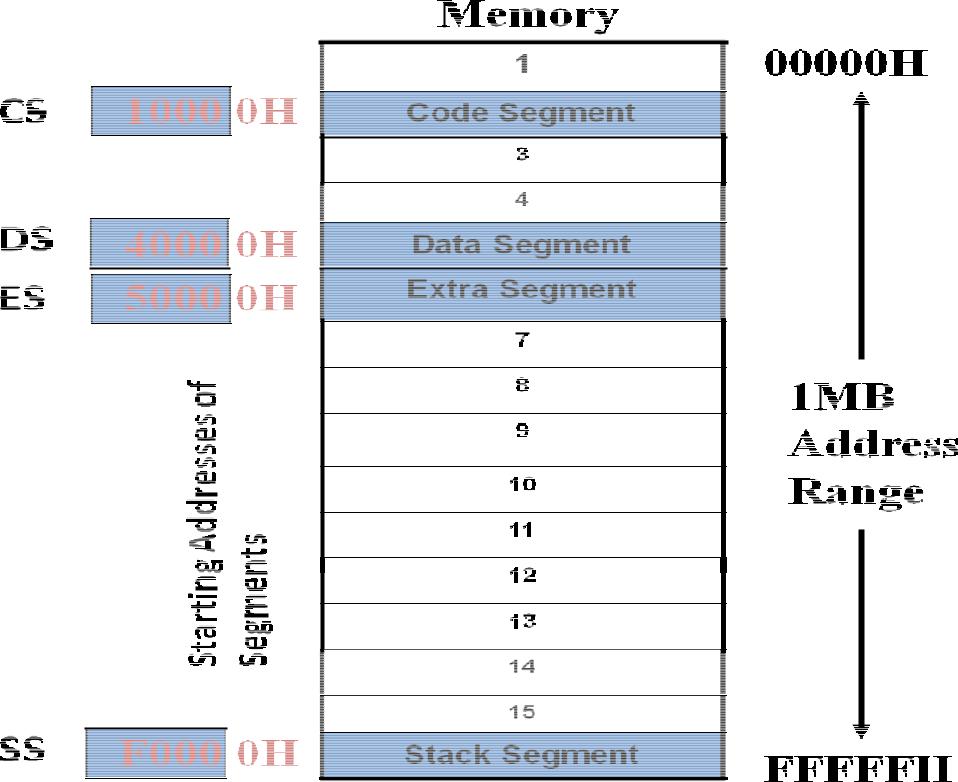
To hold the upper 16-bits of the starting address for each of the segme nts, there are four segment registers:

– CS **(Code Segment register)**

– DS **(Data Segmen t register)**

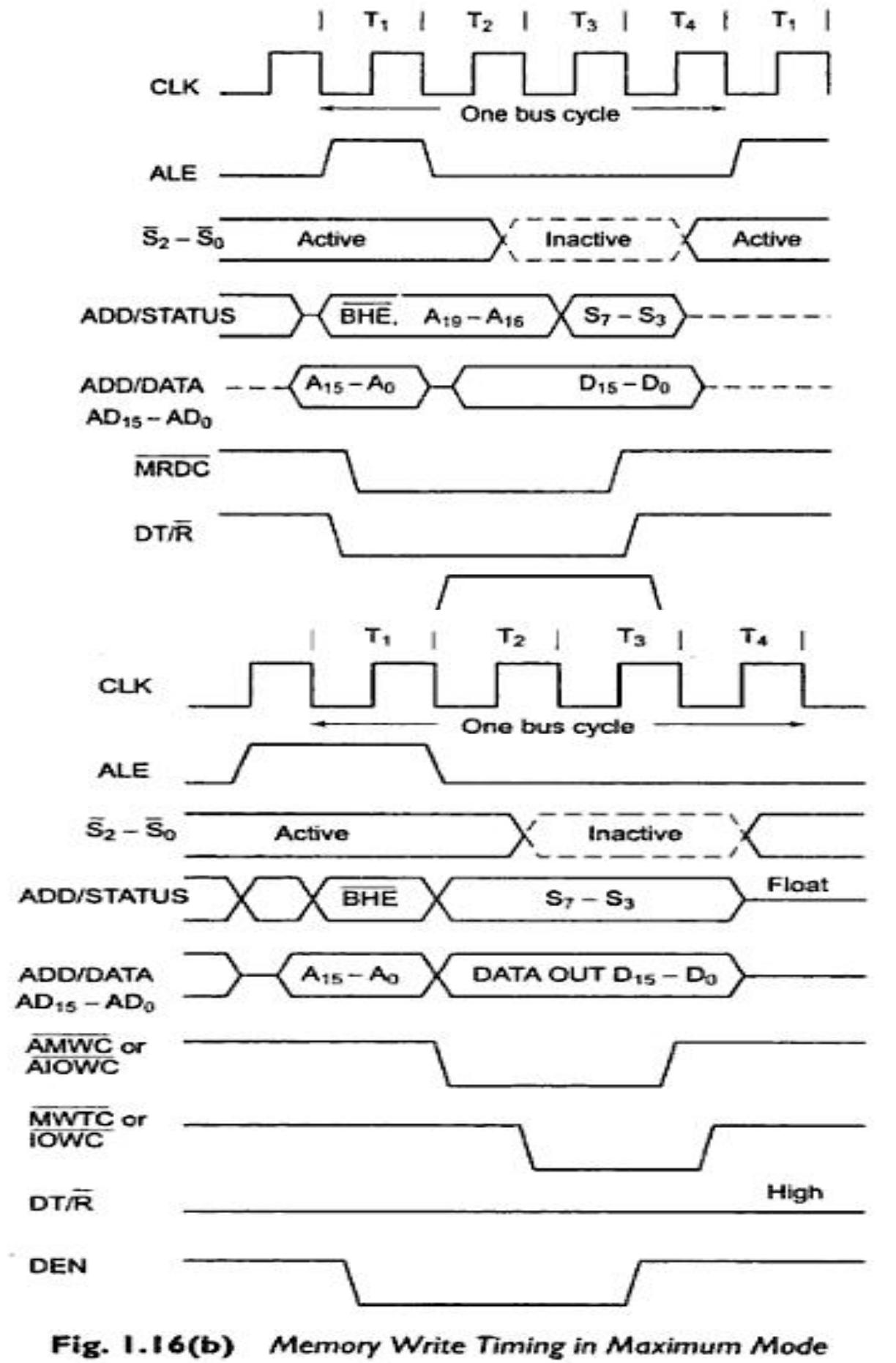
– SS **(Stack Segment register)**

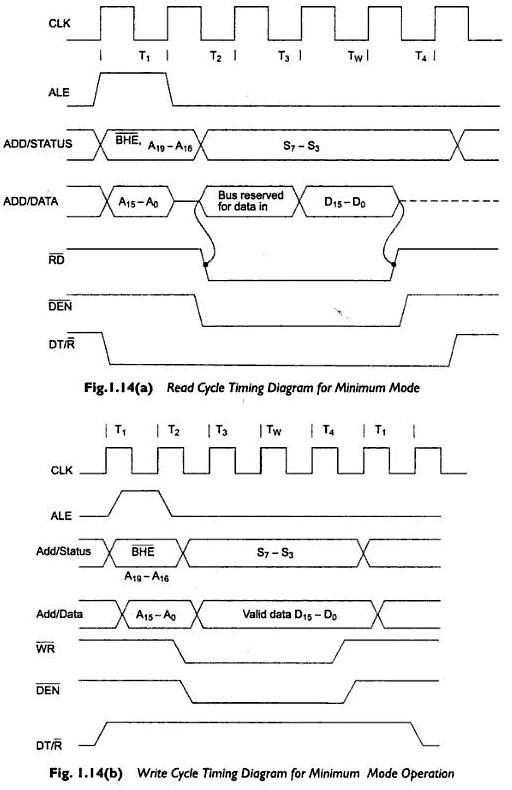
– ES **(Extra Segmen t register)**

****

**Advantage of memory Segmentation:**

* Allows the memory capa city to be 1 Mbytes although the actual addresses to be handled are of 16-bit size.
* Allows the placing of cod e, data and stack portions of the same program in different parts (segments) of memo ry, 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.

****

****

**ADDRESSING MODES OF 8086**

Addressing mode indicates a way of locating data or operands. Depending upon the data types used in the instruction and the memory addressing modes, any instruction may belong to one or more addressing modes, or some instruction may not belong to any of the addressing modes. Thus the addressing modes describe the types of operands and the way they are accessed for executing an instruction. Here, we will present the addressing modes of the instructions depending upon their types. According to the flow of instruction execution, the instructions may be categorized as

1. Sequential control flow instructions and
2. Control transfer instructions.

Sequential control flow instructions are the instructions, which after execution, transfer control to the next instruction appearing immediately after it (in the sequence) in the program. For example, the arithmetic, logical, data transfer and processor control instructions are sequential control flow instructions. The control transfer instructions, on the other hand, transfer control to some predefined address somehow specified in the instruction after their execution. For example, INT, CALL, RET and JUMP instructions fall under this category.

The addressing modes for sequential control transfer instructions are explained as follows:

**1. Immediate:** In this type of addressing, immediate data is a part ofinstruction, and appears in the form of successive byte or bytes.

Example: MOV AX, 0005H

In the above example, 0005H is the immediate data. The immediate data may be 8-bit or 16-bit in size.

**2. Direct:** In the direct addressing mode, a 16-bit memory address (offset)is directly specified in the instruction as a part of it.

Example: MOV AX, [5000H]

.Here, data resides in a memory location in the data segment, whose effective address may be computed using 5000H as the offset address and content of DS as segment address. The effective address, here, is 10H\*DS+5000H.

**3. Register**: In register addressing mode, the data is stored in a registerand it is referred using the particular register. All the registers, except IP, may be used in this mode.

Example: MOV BX, AX.

**4. Register Indirect:** Sometimes, the address of the memory location,which contains data or operand, is determined in an indirect way, using the offset registers. This mode of addressing is known as register indirect mode. In this addressing mode, the offset address of data is in either BX or SI or DI registers. The default segment is either DS or ES. The data is supposed to be available at the address pointed to by the content of any of the above registers in the default data segment.

Example: MOV AX, [BX]

Here, data is present in a memory location in DS whose offset address is in BX. The effective address of the data is given as 10H\*DS+ [BX].

**5. Indexed:** In this addressing mode, offset of the operand is stored in oneof the index registers. DS and ES are the default segments for index registers SI and DI respectively. This mode is a special case of the above discussed register indirect addressing mode.

Example: MOV AX, [SI]

Here, data is available at an offset address stored in SI in DS. The effective address, in this case, is computed as 10H\*DS+ [SI].

**6. Register Relative:** In this addressing mode, the data is available at aneffective address formed by adding an 8-bit or 16-bit displacement with the content of any one of the registers BX, BP, SI and DI in the default (either DS or ES) segment. The example given before explains this mode.

Example: MOV Ax, 50H [BX]

Here, effective address is given as 10H\*DS+50H+ [BX].

**7. Based Indexed:** The effective address of data is formed, in thisaddressing mode, by adding content of a base register (any one of BX or BP) to the content of an index register (any one of SI or DI). The default segment register may be ES or DS.

Example: MOV AX, [BX] [SI]

Here, BX is the base register and SI is the index register. The effective address is computed as 10H\*DS+ [BX] + [SI].

**8. Relative Based Indexed:** The effective address is formed by adding an8-bit or 16-bit displacement with the sum of contents of any one of the bases registers (BX or BP) and any one of the index registers, in a default segment.

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

Here, 50H is an immediate displacement, BX is a base register and SI is an index register. The effective address of data is computed as 160H\*DS+ [BX] + [SI] + 50H.

For the control transfer instructions, the addressing modes depend upon whether the destination location is within the same segment or a different one. It also depends upon the method of passing the destination address to the processor. Basically, there are two addressing modes for the control transfer instructions, viz. inter-segment and intra-segment addressing modes.

**ADDRESSING MODES FOR CONTROL TRANSFER INSTRUCTION**

1. **Intra-segment direct mode:** In this mode, the address to which thecontrol is to be transferred lies in the same segment in which the control transfer instruction lies and appears directly in the instruction as an immediate displacement value. In this addressing mode, the displacement is computed relative to the content of the instruction pointer IP.

The effective address to which the control will be transferred is given by the sum of 8 or 16 bit displacement and current content of IP. In case of jump instruction, if the signed displacement (d) is of 8 bits (i.e. –128<d<+128), we term it as short jump and if it is of

* 1. its (i.e. –32768<+32768), it is termed as long jump.

**1.Intra-segment Indirect Mode:** In this mode, the displacement to whichthe control is to be transferred, is in the same segment in which the control transfer instruction lies, but it is passed to the instruction indirectly. Here, the branch address is found as the content of a register or a memory location. This addressing mode may be used in unconditional branch instructions.

**2.Inter-segment Direct Mode:** In this mode, the address to which thecontrol is to be transferred is in a different segment. This addressing mode provides a means of branching from one code segment to another code segment. Here, the CS and IP of the destination address are specified directly in the instruction.

**3.Inter-segment Indirect Mode:** In this mode, the address to which thecontrol is to be transferred lies in a different segment and it is passed to the instruction indirectly, i.e. contents of a memory block containing four bytes, i.e. IP (LSB), IP (MSB), CS (LSB) and CS (MSB) sequentially. The starting address of the memory block may be referred using any of the addressing modes, except immediate mode.

**8086 Instruction Set and Assembler Directives**

The 8086 microprocessor supports 6 types of Instructions. They are

1. Data transfer instructions
2. Arithmetic instructions

1. Bit manipulation instructions
2. String instructions
3. Program Execution Transfer instructions (Branch & loop Instructions)
4. Processor control instructions

**1. Data Transfer instructions: These** instructions are used to transfer thedata from source operand to destination operand. All the store, move, load, exchange, input and output instructions belong to this group.

**General purpose byte or word transfer instructions**:

MOV **:** Copy byte or word from specified source to specified destination

PUSH **:** Push the specified word to top of the stack

POP **:** Pop the word from top of the stack to the specified location

PUSHA **:** Push all registers to the stack

POPA **:** Pop the words from stack to all registers

XCHG **:** Exchange the contents of the specified source and destination operands one of which may be a register or memory location.

XLAT : Translate a byte in AL using a table in memory

**Simple input and output port transfer instructions**

1. IN : Reads a byte or word from specified port to the accumulator
2. OUT : Sends out a byte or word from accumulator to a specified port

**Special address transfer instructions**

1. LEA : Load effective address of operand into specified register

|  |  |  |
| --- | --- | --- |
| 2. | LDS | : Load DS register and other specified register from |
|  | memory | |
| 3. | LES | : Load ES register and other specified register from |
|  | memory. | |

**Flag transfer registers**

|  |  |  |
| --- | --- | --- |
| 1. | LAHF : Load AH with the low byte of the flag register | |
| 2. | SAHF | : Store AH register to low byte of flag register |
| 3. | PUSHF : Copy flag register to top of the stack | |
| 4. | POPF | : Copy word at top of the stack to flag register |

**2. Arithmetic instructions :** These instructions are used to perform variousmathematical operations like addition, subtraction, multiplication and division etc….

**Addition instructions**

1.ADD **:** Add specified byte to byte or word to word 2.ADC **:** Add with carry

3.INC **:** Increment specified byte or specified word by 1

4.AAA **:** ASCII adjust after addition

5.DAA **:** Decimal (BCD) adjust after addition

**Subtraction instructions**

1. SUB : Subtract byte from byte or word from word
2. SBB : Subtract with borrow

|  |  |  |
| --- | --- | --- |
| 3. | DEC | : Decrement specified byte or word by 1 |
| 4. | NEG | : Negate or invert each bit of a specified byte or word |
|  | and add 1(2’s complement) | |
| 5. | CMP | : Compare two specified byte or two specified words |
| 6. | AAS | : ASCII adjust after subtraction |
| 7. | DAS | : Decimal adjust after subtraction |

**Multiplication instructions**

1. MUL **:** Multiply unsigned byte by byte or unsigned word or word.
2. IMUL **:** Multiply signed bye by byte or signed word by word
3. AAM **:** ASCII adjust after multiplication

**Division instructions**

1. DIV : Divide unsigned word by byte or unsigned double word by word

1. IDIV : Divide signed word by byte or signed double word by word
2. AAD : ASCII adjust after division

|  |  |  |
| --- | --- | --- |
| 4. | CBW | : Fill upper byte of word with copies of sign bit of |
|  | lower byte | |
| 5. | CWD | : Fill upper word of double word with sign bit of |
|  | lower word. | |

**3. Bit Manipulation instructions :** These instructions include logical , shiftand rotate instructions in which a bit of the data is involved.

**Logical instructions**

1. NOT :Invert each bit of a byte or word.
2. AND : ANDing each bit in a byte or word with the corresponding bit in another byte or word.
3. OR : ORing each bit in a byte or word with the corresponding bit in another byte or word.
4. XOR : Exclusive OR each bit in a byte or word with the corresponding bit in another byte or word.
5. TEST :AND operands to update flags, but don’t change operands.

**Shift instructions**

|  |  |  |
| --- | --- | --- |
| 1. | SHL/SAL : Shift bits of a word or byte left, put zero(S) in | |
|  | LSBs. |  |
| 2. | SHR | : Shift bits of a word or byte right, put zero(S) in |
|  | MSBs. |  |
| 3. | SAR | : Shift bits of a word or byte right, copy old MSB |

into new MSB.

**Rotate instructions**

* 1. ROL : Rotate bits of byte or word left, MSB to LSB and to Carry Flag [CF]
  2. ROR : Rotate bits of byte or word right, LSB to MSB and to Carry Flag [CF]
  3. RCR :Rotate bits of byte or word right, LSB TO CF and CF to

MSB

* 1. RCL :Rotate bits of byte or word left, MSB TO CF and CF to

LSB

1. **String instructions**

A string is a series of bytes or a series of words in sequential memory locations. A string often consists of ASCII character codes.

1. REP : An instruction prefix. Repeat following instruction until CX=0
2. REPE/REPZ : Repeat following instruction until CX=0 or zero flag ZF=1
3. REPNE/REPNZ : Repeat following instruction until CX=0 or zero flag ZF=1
4. MOVS/MOVSB/MOVSW: Move byte or word from one string to another
5. COMS/COMPSB/COMPSW: Compare two string bytes or two string words
6. INS/INSB/INSW: Input string byte or word from port
7. OUTS/OUTSB/OUTSW : Output string byte or word to port
8. SCAS/SCASB/SCASW: Scan a string. Compare a string byte with a byte in AL or a string word with a word in AX
9. LODS/LODSB/LODSW: Load string byte in to AL or string word into AX

**5.Program Execution Transfer instructions**

These instructions are similar to branching or looping instructions. These instructions include conditional & unconditional jump or loop instructions.

**Unconditional transfer instructions**

|  |  |  |
| --- | --- | --- |
| 1. | CALL | : Call a procedure, save return address on stack |
| 2. | RET | : Return from procedure to the main program. |
| 3. | JMP | : Goto specified address to get next instruction |

**Conditional transfer instructions**

|  |  |  |
| --- | --- | --- |
| 1. | JA/JNBE | : Jump if above / jump if not below or equal |
| 2. | JAE/JNB | : Jump if above /jump if not below |
| 3. | JBE/JNA | : Jump if below or equal/ Jump if not above |
| 4. | JC | : jump if carry flag CF=1 |
| 5. | JE/JZ | : jump if equal/jump if zero flag ZF=1 |
| 6. | JG/JNLE | : Jump if greater/ jump if not less than or equal |
| 7. | JGE/JNL | : jump if greater than or equal/ jump if not less than |
| 8. | JL/JNGE | : jump if less than/ jump if not greater than or equal |
| 9. | JLE/JNG | : jump if less than or equal/ jump if not greater than |
| 10. JNC | | : jump if no carry (CF=0) |

11. JNE/JNZ : jump if not equal/ jump if not zero(ZF=0)

12. JNO : jump if no overflow(OF=0)

13. JNP/JPO : jump if not parity/ jump if parity odd(PF=0)

14. JNS : jump if not sign(SF=0)

15. JO : jump if overflow flag(OF=1)

16. JP/JPE : jump if parity/jump if parity even(PF=1)

17. JS : jump if sign(SF=1)

**6. Iteration control instructions**

These instructions are used to execute a series of instructions for certain number of times.

|  |  |
| --- | --- |
| 1. | LOOP :Loop through a sequence of instructions until CX=0 |
| 2. | LOOPE/LOOPZ : Loop through a sequence of instructions while |
|  | ZF=1 andCX = 0 |

* 1. LOOPNE/LOOPNZ : Loop through a sequence of instructions while ZF=0 and CX =0
  2. JCXZ : jump to specified address if CX=0

1. **Interrupt instructions**

|  |  |  |
| --- | --- | --- |
| 1. | INT | : Interrupt program execution, call service procedure |
| 2. | INTO : Interrupt program execution if OF=1 | |
| 3. | IRET | : Return from interrupt service procedure to main program |

**8.High level language interface instructions**

1. ENTER : enter procedure
2. LEAVE :Leave procedure
3. BOUND : Check if effective address within specified array bounds

**9.Processor control instructions**

Flag set/clear instructions

|  |  |  |
| --- | --- | --- |
| 1. | STC | : Set carry flag CF to 1 |
| 2. | CLC | : Clear carry flag CF to 0 |
| 3. | CMC : Complement the state of the carry flag CF | |
| 4. | STD | : Set direction flag DF to 1 (decrement string pointers) |
| 5. | CLD | : Clear direction flag DF to 0 |
| 6. | STI | : Set interrupt enable flag to 1(enable INTR input) |
| 7. | CLI | : Clear interrupt enable Flag to 0 (disable INTR input) |

**10. External Hardware synchronization instructions**

|  |  |  |
| --- | --- | --- |
| 1. | HLT | : Halt (do nothing) until interrupt or reset |
| 2. | WAIT | : Wait (Do nothing) until signal on the test pin is low |
| 3. | ESC | : Escape to external coprocessor such as 8087 or 8089 |
| 4. | LOCK | : An instruction prefix. Prevents another processor from |

taking the bus while the adjacent instruction executes.

**11. No operation instruction**

1. NOP : No action except fetch and decode

Instruction Description

**AAA** Instruction - ASCII Adjust after Addition **AAD** Instruction - ASCII adjust before Division

**AAM** Instruction - ASCII adjust after Multiplication **AAS** Instruction - ASCII Adjust for Subtraction

**ADC** Instruction - Add with carry.

**ADD** Instruction - ADD destination, source

**AND** Instruction - AND corresponding bits of two operands

*Example*

**AAA** Instruction:

AAA converts the result of the addition of two valid unpacked BCD digits to a valid 2-digit BCD number and takes the AL register as its implicit operand.

Two operands of the addition must have its lower 4 bits contain a number in the range from 0-9.The AAA instruction then adjust AL so that it contains a correct BCD digit. If the addition produce carry (AF=1), the AH register is incremented and the carry CF and auxiliary carry AF flags are set to 1. If the addition did not produce a decimal carry, CF and AF are cleared to 0 and AH is not altered. In both cases the higher 4 bits of AL are cleared to 0.

AAA will adjust the result of the two ASCII characters that were in the range from 30h (“0”) to 39h(“9”).This is because the lowe r 4 bits of those character fall in the range of 0-9.The result of addition is not a ASCII character but it is a BCD digit.

**Example:**

**MOV AH, 0 ; Clear AH for MSD**

**MOV AL, 6 ; BCD 6 in AL**

**ADD AL, 5 ; Add BCD 5 to digit in AL**

**AAA ; AH=1, AL=1 representing BCD 11.**

**AAD Instruction:** ADD converts unpacked BCD digits in the AH andAL register into a single binary number in the AX register in preparation for a division operation.

Before executing AAD, place the Most significant BCD digit in the AH register and Last significant in the AL register. When AAD is executed, the two BCD digits are combined into a single binary number by setting AL=(AH\*10)+AL and clearing AH to 0.

**Example:**

**MOV AX, 0205h ; The unpacked BCD number 25**

**AAD ; After AAD, AH=0 and**

**; AL=19h (25)**

After the division AL will then contain the unpacked BCD quotient and AH will contain the unpacked BCD remainder.

**Example:**

* **AX=0607 unpacked BCD for 67 decimal**
* **CH=09H**

**AAD ; Adjust to binary before division ; AX=0043 = 43H =67 decimal**

**DIV CH ; Divide AX by unpacked BCD in CH**

* **AL = quotient = 07 unpacked BCD**
* **AH = remainder = 04 unpacked BCD**

**AAM** Instruction - AAM converts the result of the multiplication of twovalid unpacked BCD digits into a valid 2-digit unpacked BCD number and takes AX as an implicit operand.

To give a valid result the digits that have been multiplied must be in the range of 0 – 9 and the result should have been placed in the AX register. Because both operands of multiply are required to be 9 or less, the result must be less than 81 and thus is completely contained in AL.

AAM unpacks the result by dividing AX by 10, placing the quotient (MSD) in AH and the remainder (LSD) in AL.

**Example:**

**MOV AL, 5**

**MOV BL, 7**

**MUL BL ; Multiply AL by BL, result in AX**

**AAM ; After AAM, AX =0305h (BCD 35)**

**AAS** Instruction: AAS converts the result of the subtraction of twovalid unpacked BCD digits to a single valid BCD number and takes the AL register as an implicit operand.

The two operands of the subtraction must have its lower 4 bit contain number in the range from 0 to 9.The AAS instruction then adjust AL so that it contain a correct BCD digit.

**MOV AX, 0901H ; BCD 91**

**SUB AL, 9 ; Minus 9**

**AAS ; Give AX =0802 h (BCD 82) ( a )**

* **AL =0011 1001 =ASCII 9**
* **BL=0011 0101 =ASCII 5 SUB AL, BL ; (9 - 5) Result:**
* **AL = 00000100 = BCD 04, CF = 0 AAS ; Result:**

* **AL=00000100 =BCD 04**
* **CF = 0 NO Borrow required ( b )**
* **AL = 0011 0101 =ASCII 5**
* **BL = 0011 1001 = ASCII 9 SUB AL, BL ; ( 5 - 9 ) Result:**
* **AL = 1111 1100 = - 4**
* **in 2’s complement CF = 1 AAS ; Results:**
* **AL = 0000 0100 =BCD 04**
* **CF = 1 borrow needed.**

**ADD** Instruction:

These instructions add a number from source to a number from some destination and put the result in the specified destination. The add with carry instruction ADC, also add the status of the carry flag into the result.

The source and destination must be of same type, means they must be a byte location or a word location. If you want to add a byte to a word, you must copy the byte to a word location and fill the upper byte of the word with zeroes before adding.

**EXAMPLE:**

**ADD AL, 74H ; Add immediate number 74H to content of AL ADC CL, BL ; Add contents of BL plus**

* **carry status to contents of CL.**
* **Results in CL**

**ADD DX, BX ; Add contents of BX to contents ; of DX**

**ADD DX, [SI] ; Add word from memory at ; offset [SI] in DS to contents of DX**

**; Addition of Un Signed numbers**

**ADD CL, BL ; CL = 01110011 =115 decimal**

* **+ BL = 01001111 = 79 decimal**
* **Result in CL = 11000010 = 194 decimal**
* **Addition of Signed numbers**

**ADD CL, BL ; CL = 01110011 = + 115 decimal**

* **+ BL = 01001111 = +79 decimal**
* **Result in CL = 11000010 = - 62 decimal**
* **Incorrect because result is too large to fit in 7 bits.**

**AND** Instruction:

This Performs a bitwise Logical AND of two operands. The result of the operation is stored in the op1 and used to set the flags.

**AND op1, op2**

To perform a bitwise AND of the two operands, each bit of the result is set to 1 if and only if the corresponding bit in both of the operands is 1, otherwise the bit in the result I cleared to 0.

**AND BH, CL ; AND byte in CL with byte in BH ; result in BH**

**AND BX, 00FFh ; AND word in BX with immediate ; 00FFH. Mask upper byte, leave ; lower unchanged**

**AND CX, [SI] ; AND word at offset [SI] in data ; segment with word in CX ; register. Result in CX register.**

**; BX = 10110011 01011110**

**AND BX, 00FFh ; Mask out upper 8 bits of BX**

* **Result BX = 00000000 01011110**
* **CF =0, OF = 0, PF = 0, SF = 0,**
* **ZF = 0**

**CALL** Instruction

•Direct within-segment (near or intrasegment) •Indirect within-segment (near or intrasegment) •Direct to another segment (far or intersegment) •Indirect to another segment (far or intersegment)

**CBW** Instruction - Convert signed Byte to signed word **CLC** Instruction - Clear the carry flag

**CLD** Instruction - Clear direction flag **CLI** Instruction - Clear interrupt flag

**CMC** Instruction - Complement the carry flag

**CMP** Instruction - Compare byte or word - CMP destination, source.

**CMPS/CMPSB/**

**CMPSW** Instruction - Compare string bytes or string words

**CWD** Instruction - Convert Signed Word to - Signed Double wordExample

**CALL** Instruction:

This Instruction is used to transfer execution to a subprogram or procedure. There are two basic types of CALL’s: Near and Far.

A Near CALL is a call to a procedure which is in the same code segment as the CALL instruction.

When 8086 executes the near CALL instruction it decrements the stack pointer by two and copies the offset of the next instruction after the CALL on the stack. This offset saved on the stack is referred as the return address, because this is the address that execution will returns to after the procedure executes. A near CALL instruction will also load the instruction pointer with the offset of the first instruction in the procedure.

A RET instruction at the end of the procedure will return execution to the instruction after the CALL by coping the offset saved on the stack back to IP.

A Far CALL is a call to a procedure which is in a different from that which contains the CALL instruction. When 8086 executes the Far CALL instruction it decrements the stack pointer by two again and copies the content of CS

register to the stack. It then decrements the stack pointer by two again and copies the offset contents offset of the instruction after the CALL to the stack.

Finally it loads CS with segment base of the segment which contains the procedure and IP with the offset of the first instruction of the procedure in segment. A RET instruction at end of procedure will return to the next instruction after the CALL by restoring the saved CS and IP from the stack.

**; Direct within-segment (near or intrasegment )**

**CALL MULTO ; MULTO is the name of the procedure. The assembler determines displacement of MULTO from the instruction after the CALL and codes this displacement in as part of the instruction.**

**; Indirect within-segment ( near or intrasegment )**

**CALL BX ; BX contains the offset of the first instruction of the procedure. Replaces contents of word of IP with contents o register BX.**

**CALL WORD PTR [BX] ; Offset of first instruction of procedure is in two memory addresses in DS. Replaces contents of IP with contents of word memory location in DS pointed to by BX.**

**; Direct to another segment- far or intersegment.**

**CALL SMART ; SMART is the name of the Procedure**

**SMART PROC FAR; Procedure must be declare as an far**

**CBW** Instruction - CBW converts the signed value in the AL registerinto an equivalent 16 bit signed value in the AX register by duplicating the sign bit to the left.

This instruction copies the sign of a byte in AL to all the bits in AH. AH is then said to be the sign extension of AL.

**Example:**

**; AX = 00000000 10011011 = - 155 decimal**

**CBW ; Convert signed byte in AL to signed word in AX.**

* **Result in AX = 11111111 10011011**
* **= - 155 decimal**

**CLC** Instruction:

CLC clear the carry flag (CF) to 0 This instruction has no affect on the processor, registers, or other flags. It is often used to clear the CF before returning from a procedure to indicate a successful termination. It is also use to clear the CF during rotate operation involving the CF such as ADC, RCL, RCR.

**Example:**

**CLC ; Clear carry flag.**

**CLD** Instruction:

This instruction reset the designation flag to zero. This instruction has no effect on the registers or other flags. When the direction flag is cleared / reset SI and DI will

automatically be incremented when one of the string instruction such as MOVS, CMPS, SCAS, MOVSB and STOSB executes.

**Example:**

**CLD ; Clear direction flag so that string pointers auto increment**

**CLI** Instruction:

This instruction resets the interrupt flag to zero. No other flags are affected. If the interrupt flag is reset, the 8086 will not respond to an interrupt signal on its INTR input. This CLI instruction has no effect on the nonmaskable interrupt input, NMI

**CMC** Instruction:

If the carry flag CF is a zero before this instruction, it will be set to a one after the instruction. If the carry flag is one before this instruction, it will be reset to a zero after the instruction executes. CMC has no effect on other flags.

**Example:**

**CMC; Invert the carry flag.**

**CWD** Instruction:

CWD converts the 16 bit signed value in the AX register into an equivalent 32 bit signed value in DX: AX register pair by duplicating the sign bit to the left.

The CWD instruction sets all the bits in the DX register to the same sign bit of the AX register. The effect is to create a 32- bit signed result that has same integer value as the original 16 bit operand.

**Example:**

**Assume AX contains C435h. If the CWD instruction is executed, DX will contain FFFFh since bit 15 (MSB) of AX was 1. Both the original value of AX (C435h) and resulting value of DX: AX (FFFFC435h) represents the same signed number.**

**Example:**

* **DX = 00000000 00000000**
* **AX = 11110000 11000111 = - 3897 decimal**

**CWD ; Convert signed word in AX to signed double**

* **word in DX:AX**
* **Result DX = 11111111 11111111**
* **AX = 11110000 11000111 = -3897 decimal.**

**DAA** Instruction - Decimal Adjust Accumulator

**DAS** Instruction - Decimal Adjust after Subtraction

**DEC** Instruction - Decrement destination register or memory DECdestination.

**DIV** Instruction - Unsigned divide-Div source **ESC** Instruction

When a double word is divided by a word, the most significant word of the double word must be in DX and the least significant word of the double word must be in AX. After the division AX will contain the 16 –bit result (quotient) and DX will contain a 16 bit remainder. Again, if an attempt is made to divide by zero or quotient is too large to fit in AX (greater than FFFFH) the 8086 will do a type of 0 interrupt.

**Example:**

**DIV CX ; (Quotient) AX= (DX: AX)/CX**

**: (Reminder) DX= (DX: AX)%CX**

For DIV the dividend must always be in AX or DX and AX, but the source of the divisor can be a register or a memory location specified by one of the 24 addressing modes.

If you want to divide a byte by a byte, you must first put the dividend byte in AL and fill AH with all 0’s. The SUB AH, AH instruction is a quick way to do.

If you want to divide a word by a word, put the dividend word in AX and fill DX with all 0’s. The SUB DX, DX instruction does this quickly.

**Example: ; AX = 37D7H = 14, 295 decimal**

* **BH = 97H = 151 decimal**

**DIV BH ; AX / BH**

* **AX = Quotient = 5EH = 94 decimal**
* **AH = Remainder = 65H = 101 decimal**

**ESC** Instruction - Escape instruction is used to pass instruction to acoprocessor such as the 8087 math coprocessor which shares the address and data bus with an 8086. Instruction for the coprocessor is represented by a 6 bit code embedded in the escape instruction. As the 8086 fetches instruction byte, the coprocessor also catches these bytes from data bus and puts them in its queue. The coprocessor treats all of the 8086 instruction as an NOP. When 8086 fetches an ESC instruction, the coprocessor decodes the instruction and carries out the action specified by the 6 bit code. In most of the case 8086 treats ESC instruction as an NOP.

**HLT** Instruction - HALT processing

**IDIV** Instruction - Divide by signed byte or word IDIV source **IMUL** Instruction - Multiply signed number-IMUL source

**IN** Instruction - Copy data from a port IN accumulator, port **INC** Instruction - Increment - INC destination

**HALT** Instruction - The HLT instruction will cause the 8086 to stopfetching and executing instructions. The 8086 will enter a halt state. The only way to get the processor out of the halt state are with an interrupt signal on the INTR pin or an interrupt signal on NMI pin or a reset signal on the RESET input.

**IDIV** Instruction - This instruction is used to divide a signed word by asigned byte or to divide a signed double word by a signed word.

**Example:**

**IDIV BL ; Signed word in AX is divided by signed byte in BL**

**IMUL** Instruction - This instruction performs a signed multiplication.

**IMUL op** ; In this form the accumulator is the multiplicand and op is themultiplier. op may be a register or a memory operand.

**IMUL op1, op2** ; In this form op1 is always be a register operand and op2may be a register or a memory operand.

**Example:**

**IMUL BH ; Signed byte in AL times multiplied by ; signed byte in BH and result in AX.**

**Example:**

* **69 \* 14**
* **AL = 01000101 = 69 decimal**
* **BL = 00001110 = 14 decimal**

**IMUL BL ; AX = 03C6H = + 966 decimal**

* **MSB = 0 because positive result**
* **- 28 \* 59**
* **AL = 11100100 = - 28 decimal**
* **BL = 00001110 = 14 decimal**

**IMUL BL ; AX = F98Ch = - 1652 decimal**

* **MSB = 1 because negative result**

**IN** Instruction: This IN instruction will copy data from a port to the ALor AX register.

For the Fixed port IN instruction type the 8 – bit port address of a port is specified directly in the instruction.

**Example:**

**IN AL, 0C8H ; Input a byte from port 0C8H to AL**

**IN AX, 34H ; Input a word from port 34H to AX**

**A\_TO\_D EQU 4AH**

**IN AX, A\_TO\_D ; Input a word from port 4AH to AX**

For a variable port IN instruction, the port address is loaded in DX register before IN instruction. DX is 16 bit. Port address range from 0000H – FFFFH.

**Example:**

**MOV DX, 0FF78H ; Initialize DX point to port**

**IN AL, DX ; Input a byte from a 8 bit port ; 0FF78H to AL**

**IN AX, DX ; Input a word from 16 bit port to ; 0FF78H to AX.**

**INC** Instruction:

INC instruction adds one to the operand and sets the flag according to the result. INC instruction is treated as an unsigned binary number.

**Example:**

**; AX = 7FFFh**

**INC AX ; After this instruction AX = 8000h**

**INC BL ; Add 1 to the contents of BL register**

**INC CL ; Add 1 to the contents of CX register.**

**INT** Instruction - Interrupt program

**INTO** Instruction - Interrupt on overflow. **IRET** Instruction - Interrupt return

**JA/JNBE** Instruction - Jump if above/Jump if not below nor equal. **JAE/JNB/JNC** Instructions- Jump if above or equal/ Jump if not

below/

Jump if no carry.

**JA / JNBE** - This instruction performs the Jump if above (or) Jump ifnot below or equal operations according to the condition, if CF and ZF = 0.

**Example:**

**( 1 ) CMP AX, 4371H ; Compare by subtracting 4371H ; from AX**

**JA RUN\_PRESS ; Jump to label RUN\_PRESS if ; AX above 4371H ( 2 ) CMP AX, 4371H ; Compare ( AX – 4371H)**

**JNBE RUN\_PRESS ; Jump to label RUN\_PRESS if ; AX not below or equal to 4371H**

**JAE / JNB / JNC** - This instructions performs the Jump if above orequal, Jump if not below, Jump if no carry operations according to the condition, if CF = 0.

**Examples**:

**1**. **CMP AX, 4371H ; Compare ( AX – 4371H)**

**JAE RUN ; Jump to the label RUN if AX is ; above or equal to 4371H. 2. CMP AX, 4371H ; Compare ( AX – 4371H)**

**JNB RUN\_1 ; Jump to the label RUN\_1 if AX ; is not below than 4371H**

**3. ADD AL, BL ; Add AL, BL. If result is with in JNC OK ; acceptable range, continue**

**JB/JC/JNAE** Instruction - Jump if below/Jump if carry/ Jump if notabove nor equal

**JBE/JNA** Instructions- Jump if below or equal / Jump if not above **JCXZ** Instruction - Jump if the CX register is zero

**JE/JZ** Instruction - Jump if equal/Jump if zero

**JG/JNLE** Instruction- Jump if greater/Jump if not less than nor equal **JB/JC/JNAE** Instruction - This instruction performs the Jump if below

(or) Jump if carry (or) Jump if not below/ equal operations according to the condition, if CF = 1

**Example:**

**1. CMP AX, 4371H ; Compare (AX – 4371H)**

**JB RUN\_P ; Jump to label RUN\_P if AX is ; below 4371H 2. ADD BX, CX ; Add two words and Jump to**

**JC ERROR ; label ERROR if CF = 1**

**JBE/JNA** Instruction - This instruction performs the Jump if below orequal (or) Jump if not above operations according to the condition, if CF and ZF = 1

**Example:**

**CMP AX, 4371H ; Compare (AX – 4371H )**

**JBA RUN ; Jump to label RUN if AX is ; below or equal to 4371H CMP AX, 4371H ; Compare ( AX – 4371H )**

**JNA RUN\_R ; Jump to label RUN\_R if AX is ; not above than 4371H**

**JCXZ** Instruction:

This instruction performs the Jump if CX register is zero. If CX does not contain all zeros, execution will simply proceed to the next instruction.

**Example:**

**JCXZ SKIP\_LOOP ; If CX = 0, skip the process**

**NXT: SUB [BX], 07H ; Subtract 7 from data value**

**INC BX ; BX point to next value**

**LOOP NXT ; Loop until CX = 0**

**SKIP\_LOOP ; Next instruction**

**JE/JZ** Instruction:

This instruction performs the Jump if equal (or) Jump if zero operations according to the condition if ZF = 1

**Example:**

**NXT: CMP BX, DX ; Compare ( BX – DX )**

**JE DONE ; Jump to DONE if BX = DX,**

**SUB BX, AX ; Else subtract Ax**

**INC CX ; Increment counter**

**JUMP NXT ; Check again**

**DONE: MOV AX, CX; Copy count to AX Example:**

**IN AL, 8FH ; read data from port 8FH SUB AL, 30H ; Subtract minimum value**

**JZ STATR ; Jump to label if result of ; subtraction was 0**

**JG/JNLE** Instruction:

This instruction performs the Jump if greater (or) Jump if not less than or equal operations according to the condition if ZF =0 and SF = OF

**Example:**

**CMP BL, 39H ; Compare by subtracting ; 39H from BL JG NEXT1 ; Jump to label if BL is ; more positive than 39H CMP BL, 39H ; Compare by subtracting ; 39H from BL**

**JNLE NEXT2 ; Jump to label if BL is not ; less than or equal 39H**

**JGE/JNL** Instruction - Jump if greater than or equal/ Jump if not lessthan

**JL/JNGE** Instruction - Jump if less than/Jump if not greater than orequal

**JLE/JNG** Instruction - Jump if less than or equal/ Jump if not greater **JMP** Instruction - Unconditional jump to - specified destination

**JGE/JNL** Instruction - This instruction performs the Jump if greaterthan or equal / Jump if not less than operation according to the condition if SF = OF

**Example:**

**CMP BL, 39H ; Compare by the ; subtracting 39H from BL**

**JGE NEXT11 ; Jump to label if BL is ; more positive than 39H ; or equal to 39H**

**CMP BL, 39H ; Compare by subtracting ; 39H from BL JNL NEXT22 ; Jump to label if BL is not ; less than 39H**

**JL/JNGE** Instruction - This instruction performs the Jump if less than /Jump if not greater than or equal operation according to the condition, if SF ≠ OF

**Example:**

**CMP BL, 39H ; Compare by subtracting 39H ; from BL**

**JL AGAIN ; Jump to the label if BL is more ; negative than 39H CMP BL, 39H ; Compare by subtracting 39H ; from BL**

**JNGE AGAIN1 ; Jump to the label if BL is not ; more positive than 39H or ; not equal to 39H**

**JLE/JNG** Instruction - This instruction performs the Jump if less thanor equal / Jump if not greater operation according to the condition, if ZF=1 and SF ≠ OF

**Example:**

**CMP BL, 39h** ; **Compare by subtracting 39h ; from BL**

**JLE NXT1 ; Jump to the label if BL is more ; negative than 39h or equal to 39h**

**CMP BL, 39h ; Compare by subtracting 39h ; from BL**

**JNG AGAIN2 ; Jump to the label if BL is not ; more positive than 39h**

**JNA/JBE** Instruction - Jump if not above/Jump if below or equal **JNAE/JB** Instruction - Jump if not above or equal/ Jump if below

**JNB/JNC/JAE** Instruction - Jump if not below/Jump if no carry/Jumpif above or equal

**JNE/JNZ** Instruction - Jump if not equal/Jump if not zero

**JNE/JNZ** Instruction - This instruction performs the Jump if not equal /Jump if not zero operation according to the condition, if ZF=0

**Example:**

**NXT: IN AL, 0F8H ; Read data value from port**

**CMP AL, 72 ; Compare ( AL – 72 )**

**JNE NXT ; Jump to NXT if AL ≠ 72**

**IN AL, 0F9H ; Read next port when AL = 72**

**MOV BX, 2734H ; Load BX as counter**

**NXT\_1: ADD AX, 0002H ; Add count factor to AX**

**DEC BX ; Decrement BX**

**JNZ NXT\_1 ; Repeat until BX = 0**

**JNG/JLE** Instruction - Jump if not greater/ Jump if less than or equal **JNGE/JL** Instruction - Jump if not greater than nor equal/Jump if less

than

**JNL/JGE** Instruction - Jump if not less than/ Jump if greater than orequal

**JNLE/JG** Instruction - Jump if not less than nor equal to /Jump ifgreater than

**JNO** Instruction – Jump if no overflow

**JNP/JPO** Instruction – Jump if no parity/ Jump if parity odd **JNS** Instruction - Jump if not signed (Jump if positive)

**JNZ/JNE** Instruction - Jump if not zero / jump if not equal **JO** Instruction - Jump if overflow

**JNO** Instruction – This instruction performs the Jump if no overflowoperation according to the condition, if OF=0

**Example:**

**ADD AL, BL ; Add signed bytes in AL and BL**

**JNO DONE ; Process done if no overflow -**

**MOV AL, 00H ; Else load error code in AL**

**DONE: OUT 24H, AL ; Send result to display**

**JNP/JPO** Instruction – This instruction performs the Jump if not parity/ Jump if parity odd operation according to the condition, if PF=0

**Example:**

**IN AL, 0F8H ; Read ASCII char from UART**

**OR AL, AL ; Set flags**

**JPO ERROR1 ; If even parity executed, if not ; send error message**

**JNS** Instruction - This instruction performs the Jump if not signed(Jump if positive) operation according to the condition, if SF=0

**Example:**

**DEC AL ; Decrement counter**

**JNS REDO ; Jump to label REDO if counter has not ; decremented to**

**FFH**

**JO** Instruction - This instruction performs Jump if overflow operationaccording to the condition OF = 0

**Example:**

**ADD AL, BL ; Add signed bits in AL and BL**

**JO ERROR ; Jump to label if overflow occur ; in addition**

**MOV SUM, AL ; else put the result in memory ; location named SUM**

**JPE/JP** Instruction - Jump if parity even/ Jump if parity

**JPO/JNP** Instruction - Jump if parity odd/ Jump if no parity **JS** Instruction - Jump if signed (Jump if negative)

**JZ/JE** Instruction - Jump if zero/Jump if equal

**JPE/JP** Instruction - This instruction performs the Jump if parity even /Jump if parity operation according to the condition, if PF=1

**Example:**

**IN AL, 0F8H ; Read ASCII char from UART**

**OR AL, AL ; Set flags**

**JPE ERROR2 ; odd parity is expected, if not ; send error message**

**JS** Instruction **-** This instruction performs the Jump if sign operationaccording to the condition, if SF=1

**Example:**

**ADD BL, DH ; Add signed bytes DH to BL**

**JS JJS\_S1 ; Jump to label if result is ; negative**

**LAHF** Instruction - Copy low byte of flag register to AH

**LDS** Instruction - Load register and Ds with words from memory –LDS register, memory address of first word

**LEA** Instruction - Load effective address-LEA register, source

**LES** Instruction Load register and ES with words from memory –LESregister, memory address of first word.

**LAHF** Instruction: LAHF instruction copies the value of SF, ZF, AF,PF, CF, into bits of 7, 6, 4, 2, 0 respectively of AH register. This LAHF instruction was provided to make conversion of assembly language programs written for 8080 and 8085 to 8086 easier.

**LDS** Instruction: This instruction loads a far pointer from the memoryaddress specified by op2 into the DS segment register and the op1 to the register.

**LDS op1, op2**

**Example:**

**LDS BX, [4326] ; copy the contents of the memory at displacement 4326H in DS to BL, contents of the 4327H to BH. Copy contents of 4328H and 4329H in DS to DS register.**

**LEA** Instruction - This instruction indicates the offset of the variable ormemory location named as the source and put this offset in the indicated 16 – bit register.

**Example:**

**LEA BX, PRICE ; Load BX with offset of PRICE ; in DS LEA BP, SS:STAK ; Load BP with offset of STACK ; in SS LEA CX, [BX][DI] ; Load CX with EA=BX + DI**

**LOCK** Instruction - Assert bus lock signal

**LODS/LODSB/ LODSW** Instruction - Load string byte into AL orLoad string word into AX.

**LOOP** Instruction - Loop to specified label until CX = 0

**LOOPE / LOOPZ** Instruction - loop while CX≠0 and ZF = 1

**LODS/LODSB/LODSW** Instruction - This instruction copies a bytefrom a string location pointed to by SI to AL or a word from a string location pointed to by SI to AX. If DF is cleared to 0, SI will automatically incremented to point to the next element of string.

**Example:**

**CLD ; Clear direction flag so SI is auto incremented**

**MOV SI, OFFSET SOURCE\_STRING ; point SI at start of the string LODS SOUCE\_STRING ; Copy byte or word from ; string to AL or AX**

**LOOP** Instruction - This instruction is used to repeat a series ofinstruction some number of times

**Example:**

**MOV BX, OFFSET PRICE**

**; Point BX at first element in array**

**MOV CX, 40 ; Load CX with number of ; elements in array NEXT: MOV AL, [BX] ; Get elements from array**

**ADD AL, 07H ; Ad correction factor DAA ; decimal adjust result**

**MOV [BX], AL ; Put result back in array**

**LOOP NEXT ; Repeat until all elements ; adjusted.**

**LOOPE / LOOPZ** Instruction - This instruction is used to repeat agroup of instruction some number of times until CX = 0 and ZF = 0

**Example:**

**MOV BX, OFFSET ARRAY**

**; point BX at start of the array**

**DEC BX**

**MOV CX, 100 ; put number of array elements in ; CX**

**NEXT:INC BX ; point to next element in array**

**CMP [BX], 0FFH ; Compare array elements FFH**

**LOOP NEXT**

**LOOPNE/LOOPNZ** Instruction - This instruction is used to repeat agroup of instruction some number of times until CX = 0 and ZF = 1

**Example:**

**MOV BX, OFFSET ARRAY1 ; point BX at start of the array**

**DEC BX**

**MOV CX, 100 ; put number of array elements in ; CX**

**NEXT:INC BX ; point to next elements in array**

**CMP [BX], 0FFH ; Compare array elements 0DH**

**LOOPNE NEXT**

**MOV** Instruction - MOV destination, source

**MOVS/MOVSB/ MOVSW** Instruction - Move string byte or stringword-MOVS destination, source

**MUL** Instruction - Multiply unsigned bytes or words-MUL source **NEG** Instruction - From 2’s complement – NEG destination

**NOP** Instruction - Performs no operation.

**MOV** Instruction - The MOV instruction copies a word or a byte ofdata from a specified source to a specified destination.

**MOV op1, op2 Example:**

**MOV CX, 037AH ; MOV 037AH into the CX.**

**MOV AX, BX ; Copy the contents of register BX ; to AX**

**MOV DL, [BX] ; Copy byte from memory at BX ; to DL, BX contains the offset of byte in DS.**

**MUL** Instruction:

This instruction multiplies an unsigned multiplication of the accumulator by the operand specified by op. The size of op may be a register or memory operand.

**MUL op**

**Example: ; AL = 21h (33 decimal) ; BL = A1h(161 decimal )**

**MUL BL ; AX =14C1h (5313 decimal) since AH≠0, ; CF AND OF WILL SET to 1.**

**MUL BH ; AL times BH, result in AX**

**MUL CX ; AX times CX, result high word in DX, ; low word in AX.**

**NEG** Instruction - NEG performs the two’s complement subtraction ofthe operand from zero and sets the flags according to the result. **; AX =**

**2CBh**

**NEG AX ; after executing NEG result AX =FD35h. Example:**

**NEG AL ; Replace number in AL with its 2’s complement NEG BX ; Replace word in BX with its 2’s complement NEG BYTE PTR[BX]; Replace byte at offset BX in**

* **DS with its 2’s complement**

**NOP** Instruction:

This instruction simply uses up the three clock cycles and increments the instruction pointer to point to the next instruction. NOP does not change the status of any flag. The NOP instruction is used to increase the delay of a delay loop.

**NOT** Instruction - Invert each bit of operand –NOT desti nation.

**OR** Instruction - Logically OR corresponding of two operands- ORdestination, source.

**OUT** Instruction - Output a byte or word to a port – OUT port,accumulator AL or AX.

**POP** Instruction - POP destination

**NOT** Instruction - NOT perform the bitwise complement of op andstores the result back into op.

**NOT op**

**Example:**

**NOT BX ; Complement contents of BX register.**

**; DX =F038h**

**NOT DX ; after the instruction DX = 0FC7h**

**OR** Instruction - OR instruction perform the bit wise logical OR of twooperands.Each bit of the result is cleared to 0 if and only if both corresponding bits in each operand are 0, other wise the bit in the result is set to 1.

**OR op1, op2**

**Examples:**

**OR AH, CL ; CL ORed with AH, result in AH.**

**; CX = 00111110 10100101**

**OR CX, FF00h ; OR CX with immediate FF00h**

* **result in CX = 11111111 10100101**
* **Upper byte are all 1’s lower bytes ; are unchanged.**

**OUT** Instruction - The OUT instruction copies a byte from AL or aword from AX or a double from the accumulator to I/O port specified by op. Two forms of OUT instruction are available: **(1)** Port number is specified by an immediate byte constant, ( 0 - 255 ).It is also called as fixed port form. **(2)** Port number is provided in the DX register ( 0 – 65535 )

**Example**: **(1)**

**OUT 3BH, AL ; Copy the contents of the AL to port 3Bh OUT 2CH, AX ; Copy the contents of the AX to port 2Ch**

**(2) MOV DX, 0FFF8H ; Load desired port address in DX OUT DX, AL ; Copy the contents of AL to ; FFF8h**

**OUT DX, AX ; Copy content of AX to port ; FFF8H**

**POP** Instruction:

POP instruction copies the word at the current top of the stack to the operand specified by op then increments the stack pointer to point to the next stack.

**Example**:

**POP DX ; Copy a word from top of the stack to ; DX and increments SP by 2.**

**POP DS ; Copy a word from top of the stack to**

* **DS and increments SP by 2.**

**POP TABLE [BX]**

* **Copy a word from top of stack to memory in DS with**
* **EA = TABLE + [BX].**

**POPF** Instruction - Pop word from top of stack to flag - register. **PUSH** Instruction - PUSH source

**PUSHF** Instruction - Push flag register on the stack

**RCL** Instruction - Rotate operand around to the left through CF – RCLdestination, source.

**RCR** Instruction - Rotate operand around to the right through CF- RCRdestination, count

**POPF** Instruction - This instruction copies a word from the twomemory location at the top of the stack to flag register and increments the stack pointer by 2.

**PUSH** Instruction: PUSH instruction decrements the stack pointer by 2and copies a word from a specified source to the location in the stack segment where the stack pointer pointes.

**Example:**

**PUSH BX ; Decrement SP by 2 and copy BX to stack PUSH DS ; Decrement SP by 2 and copy DS to stack**

**PUSH TABLE[BX] ; Decrement SP by 2 and copy word ; from memory in DS at**

**; EA = TABLE + [BX] to stack.**

**PUSHF** Instruction:

This instruction decrements the SP by 2 and copies the word in flag register to the memory location pointed to by SP.

**RCL** Instruction:

RCL instruction rotates the bits in the operand specified by op1 towards left by the count specified in op2.The operation is circular, the MSB of operand is rotated into a carry flag and the bit in the CF is rotated around into the LSB of operand.

**RCR op1, op2 Example:**

**CLC ; put 0 in CF**

**RCL AX, 1 ; save higher-order bit of AX in CF RCL DX, 1 ; save higher-order bit of DX in CF ADC AX, 0 ; set lower order bit if needed.**

**Example:**

**RCL DX, 1 ; Word in DX of 1 bit is moved to left, and ; MSB of word is given to CF and**

* **CF to LSB.**
* **CF=0, BH = 10110011**

**RCL BH, 1 ; Result: BH =01100110**

* **CF = 1, OF = 1 because MSB changed**
* **CF =1, AX =00011111 10101001**

**MOV CL, 2 ; Load CL for rotating 2 bit position**

**RCL AX, CL ; Result: CF =0, OF undefined**

* **AX = 01111110 10100110**

**RCR** Instruction - RCR instruction rotates the bits in the operandspecified by op1 towards right by the count specified in op2. **RCR op1,** **op2**

**Example: ( 1) RCR BX, 1 ; Word in BX is rotated by 1 bit towards**

* **right and CF will contain MSB bit and**
* **LSB contain CF bit.**

**( 2) ; CF = 1, BL = 00111000**

**RCR BL, 1 ; Result: BL = 10011100, CF =0 ; OF = 1 because MSB is changed to 1.**

**REP/REPE/REPZ/**

**REPNE/REPNZ -** (Prefix) Repeat String instruction until specified conditionexist

**RET** Instruction – Return execution from procedure to ca lling program. **ROL** Instruction - Rotate all bits of operand left, MSB to LSB ROL

destination, count.

**ROL** Instruction - ROL instruction rotates the bits in the operandspecified by op1 towards left by the count specified in op2. ROL moves each bit in the operand to next higher bit position. The higher order bit is moved to lower order position. Last bit rotated is copied into carry flag.

**ROL op1, op2**

**Example: ( 1 )**

**ROL AX, 1 ; Word in AX is moved to left by 1 bit**

* **and MSB bit is to LSB, and CF**
* **CF =0, BH =10101110**

**ROL BH, 1 ; Result: CF, Of =1, BH = 01011101**

**Example: ( 2 )**

* **BX = 01011100 11010011**
* **CL = 8 bits to rotate**

**ROL BH, CL ; Rotate BX 8 bits towards left ; CF =0, BX =11010011 01011100**

**ROR** Instruction - Rotate all bits of operand right, LSB to MSB – RORdestination, count

**SAHF** Instruction – Copy AH register to low byte of flag register

**ROR** Instruction - ROR instruction rotates the bits in the operand op1to wards right by count specified in op2. The last bit rotated is copied into CF.

**ROR op1, op2 Example:**

**( 1 ) ROR BL, 1 ; Rotate all bits in BL towards right by 1 ; bit position, LSB bit is moved to MSB**

**; and CF has last rotated bit.**

**( 2 ); CF =0, BX = 00111011 01110101**

**ROR BX, 1 ; Rotate all bits of BX of 1 bit position ; towards right and CF =1,**

**BX = 10011101 10111010**

**Example ( 3 )**

**; CF = 0, AL = 10110011,**

**MOVE CL, 04H ; Load CL**

**ROR AL, CL ; Rotate all bits of AL towards right ; by 4 bits, CF = 0, AL**

* **00111011**

**SAHF** Instruction: SAHF copies the value of bits 7, 6, 4, 2, 0 of theAH register into the SF, ZF, AF, PF, and CF respectively. This instruction was provided to make easier conversion of assembly language program written for 8080 and 8085 to 8086.

**SAL/SHL** Instruction - Shift operand bits left, put zero in LSB(s)SAL/AHL destination, count

**SAR** Instruction - Shift operand bits right, new MAB = old MSBSAR destination, count.

**SBB** Instruction - Subtract with borrow SBB destination, source **SAL / SHL** Instruction - SAL instruction shifts the bits in the

operand specified by op1 to its left by the count specified in op2. As a bit is shifted out of LSB position a 0 is kept in LSB position. CF will contain MSB bit.

**SAL op1, op2 Example:**

**; CF = 0, BX = 11100101 11010011**

**SAL BX, 1 ; Shift BX register contents by 1 bit ; position towards left ; CF = 1, BX = 11001011 1010011**

**SAR** Instruction - SAR instruction shifts the bits in the operandspecified by op1 towards right by count specified in op2.As bit is shifted out a copy of old MSB is taken in MSB

MSB position and LSB is shifted to CF.

**SAR op1, op2**

**Example: ( 1 )**

**; AL = 00011101 = +29 decimal, CF = 0**

**SAR AL, 1 ; Shift signed byte in AL towards right**

* **( divide by 2 )**
* **AL = 00001110 = + 14 decimal, CF = 1**

**( 2 ) ; BH = 11110011 = - 13 decimal, CF = 1 SAR BH, 1 ; Shifted signed byte in BH to right**

* **BH = 11111001 = - 7 decimal, CF = 1**

**SBB** Instruction - SUBB instruction subtracts op2 from op1, thensubtracts 1 from op1 is CF flag is set and result is stored in op1 and it is used to set the flag.

**Example:**

**SUB CX, BX ; CX – BX. Result in CX**

**SUBB CH, AL ; Subtract contents of AL and ; contents CF from contents of CH. ; Result in CH**

**SUBB AX, 3427H ; Subtract immediate number ; from AX**

**Example:**

* **Subtracting unsigned number ; CL = 10011100 = 156 decimal ; BH = 00110111 = 55 decimal**

**SUB CL, BH ; CL = 01100101 = 101 decimal ; CF, AF, SF, ZF = 0, OF, PF = 1**

* **Subtracting signed number**
* **CL = 00101110 = + 46 decimal**
* **BH = 01001010= + 74 decimal**

**SUB CL, BH ; CL = 11100100 = - 28 decimal**

* **CF = 1, AF, ZF =0**,
* **SF = 1 result negative**

**STD** Instruction - Set the direction flag to 1 **STI** Instruction - Set interrupt flag ( IF)

**STOS/STOSB/ STOSW** Instruction - Store byte or word in string. **SCAS/SCASB/** - Scan string byte or a

**SCASW** Instruction string word.

**SHR** Instruction - Shift operand bits right, put zero in MSB **STC** Instruction - Set the carry flag to 1

**SHR** Instruction - SHR instruction shifts the bits in op1 to right by thenumber of times specified by op2.

**Example:**

**( 1 )SHR BP, 1 ; Shift word in BP by 1 bit position to right ; and 0 is kept to MSB**

**( 2 ) MOV CL, 03H ; Load desired number of shifts into CL SHR BYTE PYR[BX] ; Shift bytes in DS at offset BX and**

**; rotate 3 bits to right and keep 3 0’s in MSB ( 3 )**

**; SI = 10010011 10101101, CF = 0**

**SHR SI, 1 ; Result: SI = 01001001 11010110**

* **CF = 1, OF = 1, SF = 0, ZF = 0**
  + **TEST** Instruction – AND operand to update flags
  + **WAIT** Instruction - Wait for test signal or interrupt signal
  + **XCHG** Instruction - Exchange XCHG destination, source
  + **XLAT/ XLATB** Instruction - Translate a byte in AL
  + **XOR** Instruction - Exclusive OR corresponding bits of two operands –XOR destination, source
  + **TEST** Instruction - This instruction ANDs the contents of a source byteor word with the contents of specified destination word. Flags are updated but neither operand is changed. TEST instruction is often used to set flags before a condition jump instruction

**Examples:**

**TEST AL, BH ; AND BH with AL. no result is ; stored. Update PF, SF,**

**ZF**

**TEST CX, 0001H ; AND CX with immediate ; number**

* **no result is stored, Update PF, ; SF Example:**
* **AL = 01010001**

**TEST Al, 80H ; AND immediate 80H with AL to ; test f MSB of AL is 1 or 0**

* **ZF = 1 if MSB of AL = 0**
* **AL = 01010001 (unchanged)**
* **PF = 0, SF = 0**
* **ZF = 1 because ANDing produced is 00**
  + **WAIT** Instruction - When this WAIT instruction executes, the 8086enters an idle condition. This will stay in this state until a signal is asserted on TEST input pin or a valid interrupt signal is received on the INTR or NMI pin.

**FSTSW STATUS** ; copy 8087 status word to memory

**FWAIT** ; wait for 8087 to finish before- ; doing next 8086 instruction

**MOV AX, STATUS** ; copy status word to AX to ; check bits

In this code we are adding up of FWAIT instruction so that it will stop the execution of the command until the above instruction is finishes it’s work.so that you are not loosing data and after that you will allow to continue the execution of instructions.

* **XCHG** Instruction - The Exchange instruction exchanges the contentsof the register with the contents of another register (or) the contents of the register with the contents of the memory location. Direct memory to memory exchange are not supported.

**XCHG op1, op2**

The both operands must be the same size and one of the operand must always be a register.

**Example:**

**XCHG AX, DX** ; Exchange word in AX with word in DX

**XCHG BL, CH** ; Exchange byte in BL with byte in CH

**XCHG AL, Money [BX]** ; Exchange byte in AL with byte ; in memory atEA.

* **XOR** Instruction - XOR performs a bit wise logical XOR of theoperands specified by op1 and op2. The result of the operand is stored in op1 and is used to set the flag.

**XOR op1, op2**

**Example: ( Numerical )**

* **BX = 00111101 01101001**
* **CX = 00000000 11111111**

**XOR BX, CX ; Exclusive OR CX with BX ; Result BX = 00111101 10010110**

**ASSEMBLER DIRECTIVES :**

Assembler directives are the directions to the assembler which indicate how an operand or section of the program is to be processed. These are also called pseudo operations which are not executable by the microprocessor. The various directives are explained below.

**1. ASSUME** : The ASSUME directive is used to inform the assembler thename of the logical segment it should use for a specified segment.

Ex: ASSUME DS: DATA tells the assembler that for any program instruction which refers to the data segment ,it should use the logical segment called DATA.

**2.DB -**Define byte. It is used to declare a byte variable or set aside one ormore storage locations of type byte in memory.

For example, CURRENT\_VALUE DB 36H tells the assembler to reserve 1 byte of memory for a variable named CURRENT\_ VALUE and to put the value 36 H in that memory location when the program is loaded into RAM .

**3. DW -Define word.** It tells the assembler to define a variable of type wordor to reserve storage locations of type word in memory.

**4**. **DD(define double word**) :This directive is used to declare a variable oftype double word or restore memory locations which can be accessed as type double word.

**5.DQ (define quadword) :**This directive is used to tell the assembler todeclare a variable 4 words in length or to reserve 4 words of storage in memory .

**6.DT (define ten bytes):**It is used to inform the assembler to define a variablewhich is **10** bytes in length or to reserve 10 bytes of storage in memory.

**7. EQU –Equate** It is used to give a name to some value or symbol**.** Everytime the assembler finds the given name in the program, it will replace the name with the value or symbol we have equated with that name

**8.ORG** -**Originate** : The ORG statement changes the starting offset addressof the data.

It allows to set the location counter to a desired value at any point in the program.For example the statement ORG 3000H tells the assembler to set the location counter to 3000H.

**9 .PROC**- Procedure: It is used to identify the start of a procedure orsubroutine.

**10. END**- End program .This directive indicates the assembler that this is theend of the program module.The assembler ignores any statements after an END directive.

**11**. **ENDP**- End procedure: It indicates the end of the procedure (subroutine)to the assembler.

**12.ENDS**-End Segment: This directive is used with the name of the segmentto indicate the end of that logical segment.

Ex: CODE SEGMENT : Start of logical segment containing

code

CODE ENDS : End of the segment named CODE.

|  |  |
| --- | --- |
|  |  |
|  |
|  |  |
|  |  |
|  |
|  |  |
|  |  |

**UNIT- III**

**Interfacing of 8086 with 8255, 8254/ 8253, 8251, 8259:** Introduction, Generation of I/O Ports, Programmable Peripheral Interface (PPI)-Intel 8255, Sample-and-Hold Circuit and Multiplexer, Keyboard and Display Interface, Keyboard and Display Controller (8279), Programmable Interval timers (Intel 8253/8254), USART (8251), PIC (8259), DAC, ADC, , Stepper Motor.

**INTERFACING**

We know that keyboard and Displays are used as communication channel with outside world. So it is necessary that we interface keyboard and displays with the microprocessor. This is called I/O interfacing. In this type of interfacing we use latches and buffers for interfacing the keyboards and displays with the microprocessor.

But the main disadvantage with this interfacing is that the microprocessor can perform only one function. It functions as an input device if it is connected to buffer and as an output device if it is connected to latch. Thus the capability is very limited in this type of interfacing.

**Memory-mapped I/O** (**MMIO**) and **port-mapped I/O** (**PMIO**):

(which is also called *isolated I/O*) are two complementary methods of performing [input/output](https://en.wikipedia.org/wiki/Input/output) (I/O) between the [CPU](https://en.wikipedia.org/wiki/Central_processing_unit) and [peripheral devices](https://en.wikipedia.org/wiki/Peripheral_device) in a [computer](https://en.wikipedia.org/wiki/Computer). An alternative approach is using dedicated I/O processors, commonly known as [channels](https://en.wikipedia.org/wiki/Channel_I/O) on [mainframe computers](https://en.wikipedia.org/wiki/Mainframe_computer), which execute their own [instructions](https://en.wikipedia.org/wiki/Instruction_(computer_science)).

Memory-mapped I/O (not to be confused with [memory-mapped file](https://en.wikipedia.org/wiki/Memory-mapped_file) I/O) uses the same [address bus](https://en.wikipedia.org/wiki/Address_bus) to address both memory and I/O devices – the memory and registers of the I/O devices are mapped to (associated with) address values. So when an address is accessed by the CPU, it may refer to a portion of physical RAM, but it can also refer to memory of the I/O device. Thus, the CPU instructions used to access the memory can also be used for accessing devices. Each I/O device monitors the CPU's address bus and responds to any CPU access of an address assigned to that device, connecting the [data bus](https://en.wikipedia.org/wiki/Bus_(computing)) to the desired device's [hardware register](https://en.wikipedia.org/wiki/Hardware_register). To accommodate the I/O devices, areas of the addresses used by the CPU must be reserved for I/O and must not be available for normal physical memory. The reservation might be temporary, such as with the [Commodore 64](https://en.wikipedia.org/wiki/Commodore_64) that does [bank switching](https://en.wikipedia.org/wiki/Bank_switching) between its I/O devices and regular memory, or permanent.

Port-mapped I/O often uses a special class of CPU instructions designed specifically for performing I/O, such as the in and out instructions found on microprocessors based on the [x86](https://en.wikipedia.org/wiki/X86) and [x86-64](https://en.wikipedia.org/wiki/X86-64) architectures. Different forms of these two instructions can copy one, two or four bytes (outb, outw and outl, respectively) between the EAX register or one of that register's subdivisions on the CPU and a specified I/O port which is assigned to an I/O device. I/O devices have a separate address space from general memory, either accomplished by an extra "I/O" pin on the CPU's physical interface, or an entire [bus](https://en.wikipedia.org/wiki/Computer_bus) dedicated to I/O. Because the address space for I/O is isolated from that for main memory, this is sometimes referred to as isolated I/O.

**8255 –Programmable Pheripheral Interface**

The 8255 is a widely used, programmable, parallel I/O device.  
  
• It can be programmed to transfer data under various conditions, from simple I/O to interrupt I/O.   
• It is flexible, versatile and economical and complex. 

The 8255 has 24 I/O pins that can be grouped primarily into two 8 bit parallel ports: A and B, with the remaining 8 bits a port C. The 8 bits of port C can be used as individual bits or be grouped in two 4-bit ports: CUPPER (CU) and CLOWER (CL), as shown in the figure 1.1. The functions of these ports are defined by writing a control word in the control register.

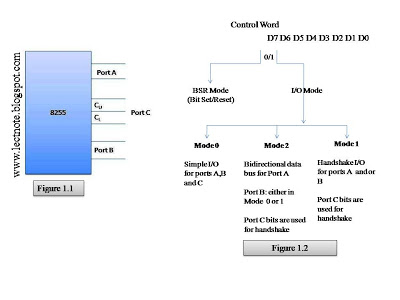
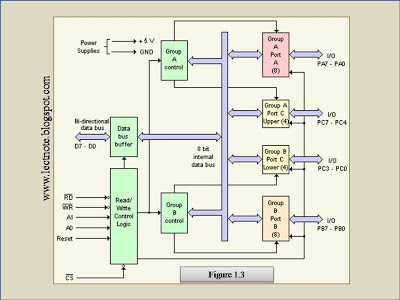
[](http://2.bp.blogspot.com/_dLQ27-zGUIM/Sxk1LwjIGOI/AAAAAAAAADo/2EZAeVAvk1c/s1600-h/IO%20Ports.jpg)

Figure 1.2 shows all the functions of 8255; classified according to two modes: the Bit Set/Reset(BSR) mode and I/O mode. The BSR mode is used to set or reset the bits in port C. The I/O mode is further divided into three modes: Mode 0, Mode 1 and Mode 2. In Mode 0, all ports function as simple I/O ports. Mode 1 is a hand shake mode whereby Ports A and/or B use bits from port C as handshake signals. In Mode 2 Port A can be set up for bidirectional data transfer using handshaking signals from Port C, and Port B can be set up either in Mode 0 or Mode 1.

**Block Diagram of the 8255**

[](http://1.bp.blogspot.com/_dLQ27-zGUIM/SxnYqK8b82I/AAAAAAAAADs/8I_GMijyZUA/s1600-h/8255_Block_Diagram.jpg)

**CONTROL LOGIC**

[http://2.bp.blogspot.com/_dLQ27-zGUIM/SxnZM7c2bNI/AAAAAAAAADw/48odtBklNZI/s200/RD.JPG](http://2.bp.blogspot.com/_dLQ27-zGUIM/SxnZM7c2bNI/AAAAAAAAADw/48odtBklNZI/s1600-h/RD.JPG)

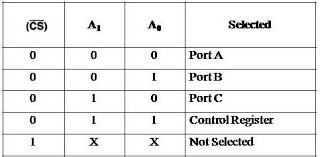
(Read): This control signal enables the Read operation. When the signal is low, the MPU reads data fro a selected I/O Port of the 8255.

[http://4.bp.blogspot.com/_dLQ27-zGUIM/SxnZdMZ6-tI/AAAAAAAAAD0/XHmATtGaNtc/s1600/WR.JPG](http://4.bp.blogspot.com/_dLQ27-zGUIM/SxnZdMZ6-tI/AAAAAAAAAD0/XHmATtGaNtc/s1600-h/WR.JPG)

(Write): This control signal enables the write operation. When the signal goes low, MPU writes into a selected I/O Port or control register.  
  
**RESET**(Reset): This is an active high signal; it clears the control register and sets all ports in the input      mode.

[http://3.bp.blogspot.com/_dLQ27-zGUIM/SxnZ2_lxiiI/AAAAAAAAAD4/_G8mRiR_Agk/s1600/cs.JPG](http://3.bp.blogspot.com/_dLQ27-zGUIM/SxnZ2_lxiiI/AAAAAAAAAD4/_G8mRiR_Agk/s1600-h/cs.JPG)

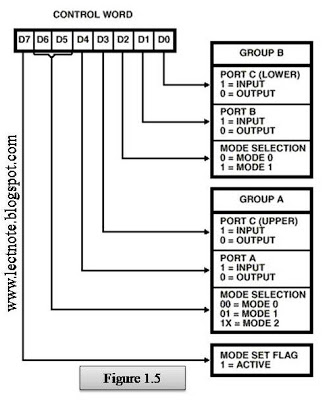
**, A0 and A1:** Theses are device select signals. Chip Select is connected to a decoded address, and A0 and A1 are generally connected to MPU address lines A0 and A1 respectively

[](http://2.bp.blogspot.com/_dLQ27-zGUIM/SxnaTtWTKOI/AAAAAAAAAD8/FNxNXk8Qz20/s1600-h/CS,A0,A1.jpg)

**CONTROL WORD**

Figure 1.5 shows a register called the control register. The contents of this register called control word. This register can be accessed to write a control word when A0 and A1 are at logic 1. This control register is not accessible for a read operation.

**Bit D7** of the control register specifies either **I/O** function or the **Bit Set/Reset** function. If bit D7=1, bits D6-D0 determines I/O functions in various modes. If bit D7=0, Port C operates in the Bit Set/Reset (BSR) mode. The BSR control word does not affect the functions of Port A and Port B.

[](http://3.bp.blogspot.com/_dLQ27-zGUIM/SxnbPTvPdYI/AAAAAAAAAEA/pwFVPfbsJW4/s1600-h/ControlWord.jpg)

To communicate with peripherals through the 8255, three steps are necessary:  
  
  
1. Determine the address of ports A, B and C and of the control register according to the chip select logic and address lines A0 andA1.  
  
2. Write the control word in the control register.  
  
3. Write I/O instructions to communicate with peripherals through Ports A, B and C.  
  
**Operating Modes**   
    
**Mode 0: Simple Input or Output**

In this mode, ports A, B are used as two simple 8-bit I/O ports and port C as two 4-bit ports. Each port can be programmed to function as simply an input port or an output port. The input/output features in Mode 0 are as follows.

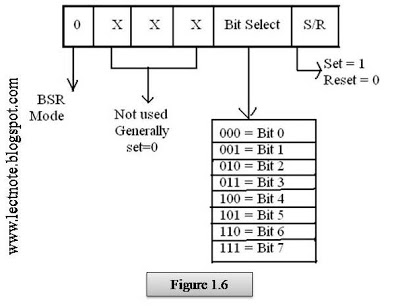
1. Outputs are latched.  
  
2. Inputs are not latched.  
  
3. Ports don’t have handshake or interrupt capability.  
  
**Mode 1: Input or Output with Handshake**   
    
In this mode, handshake signals are exchanged between the MPU and peripherals prior to data transfer. The features of the mode include the following:   
  
  
1. Two ports (A and B) function as 8-bit I/O ports. They can be configured as either as input or output ports.  
  
2. Each port uses three lines from ort C as handshake signals. The remaining two lines of Port C can be used for simple I/O operations.  
  
3. Input and Output data are latched.  
  
4. Interrupt logic is supported.  
  
**Mode 2: Bidirectional Data Transfer** 

This mode is used primarily in applications such as data transfer between two computers. In this mode, Port A can be configured as the bidirectional port and Port B either in Mode 0 or Mode 1. Port A uses five signals from Port C as handshake signals for data transfer. The remaining three signals from port C can be used either as simple I/O or as handshake for port B.

**BSR (Bit Set/Reset) Mode**

The BSR mode is concerned only with the eight bits of port C, which can be set or reset by writing an appropriate control word in the control register. A control word with bit D7 =0 is recognized as a BSR control word, and it does not alter any previously transmitted control word with bit D7=1; thus the I/O operations of ports A and B are not affected by a BSR control word. In BSR mode, individual bits of port C can be used for applications such as an on/off switch. Ports A and B are not affected by the BSR Mode.

**BSR CONTROL WORD**  
  
  
This control word, when written in the control register, sets or resets one bit at a time,

[](http://3.bp.blogspot.com/_dLQ27-zGUIM/SxncunoPzmI/AAAAAAAAAEE/xUWDDG2WRD0/s1600-h/BSR_Control_Word.jpg)

***SAMPLE AND HOLD CIRCUIT***

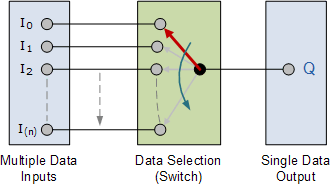
***MULTIPLEXER***

**Multiplexing is the generic term used to describe the operation of sending one or more analogue or digital signals over a common transmission line at different times or speeds and as such, the device we use to do just that is called a Multiplexer.**

The *multiplexer*, shortened to “MUX” or “MPX”, is a combinational logic circuit designed to switch one of several input lines through to a single common output line by the application of a control signal. Multiplexers operate like very fast acting multiple position rotary switches connecting or controlling multiple input lines called “channels” one at a time to the output.

The most basic type of multiplexer device is that of a one-way rotary switch as shown.

Basic Multiplexing Switch



The rotary switch, also called a wafer switch as each layer of the switch is known as a wafer, is a mechanical device whose input is selected by rotating a shaft. In other words, the rotary switch is a manual switch that you can use to select individual data or signal lines simply by turning its inputs “ON” or “OFF”. So how can we select each data input automatically using a digital device.

In digital electronics, multiplexers are also known as data selectors because they can “select” each input line, are constructed from individual [**Analogue Switches**](http://www.electronics-tutorials.ws/combination/comb_1.html) encased in a single IC package as opposed to the “mechanical” type selectors such as normal conventional switches and relays.

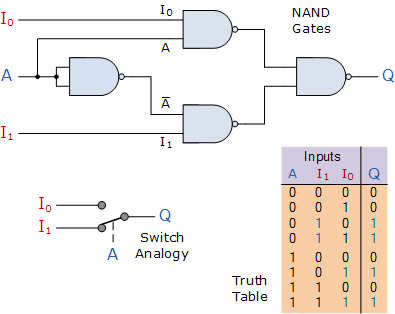
They are used as one method of reducing the number of logic gates required in a circuit design or when a single data line or data bus is required to carry two or more different digital signals. For example, a single 8-channel multiplexer.

Generally, the selection of each input line in a multiplexer is controlled by an additional set of inputs called *control lines* and according to the binary condition of these control inputs, either “HIGH” or “LOW” the appropriate data input is connected directly to the output. Normally, a multiplexer has an even number of 2N data input lines and a number of “control” inputs that correspond with the number of data inputs.

Note that multiplexers are different in operation to *Encoders*. Encoders are able to switch an n-bit input pattern to multiple output lines that represent the binary coded (BCD) output equivalent of the active input.

We can build a simple 2-line to 1-line (2-to-1) multiplexer from basic logic NAND gates as shown.

2-input Multiplexer Design



The input A of this simple 2-1 line multiplexer circuit constructed from standard NAND gates acts to control which input ( I0 or I1 ) gets passed to the output at Q.

From the truth table above, we can see that when the data select input, A is LOW at logic 0, input I1passes its data through the NAND gate multiplexer circuit to the output, while input I0 is blocked. When the data select A is HIGH at logic 1, the reverse happens and now input I0 passes data to the output Q while input I1 is blocked.

So by the application of either a logic “0” or a logic “1” at A we can select the appropriate input, I0 orI1 with the circuit acting a bit like a single pole double throw (SPDT) switch. Then in this simple example, the 2-input multiplexer connects one of two 1-bit sources to a common output, producing a 2-to-1-line multiplexer and we can confirm this in the following Boolean expression.

Q = A.I0.I1 + A.I0.I1 + A.I0.I1 + A.I0.I1

and for our 2-input multiplexer circuit above, this can be simplified too:

Q = A.I1 + A.I0

We can increase the number of data inputs to be selected further simply by following the same procedure and larger multiplexer circuits can be implemented using smaller 2-to-1 multiplexers as their basic building blocks. So for a 4-input multiplexer we would therefore require two data select lines as 4-inputs represents 22 data control lines give a circuit with four inputs, I0, I1, I2, I3 and two data select lines A and B as shown.

**8279 Programmable Keyboard/Display Controller and Interfacing: The Keyboard/Display Controller 8279**

Intel’s 8279 is a general purpose Keyboard Display controller that simultaneously drives the display of a system and interfaces a Keyboard with the CPU. The Keyboard Display interface scans the Keyboard to identify if any key has been pressed and sends the code of the pressed key to the CPU. It also transmits the data received from the CPU,to the display device.

Both of these functions are performed by the controller in repetitive fashion without involving the CPU. The Keyboard is interfaced either in the interrupt or the polled mode.In the interrupt mode, the processor is requested service only if any key is pressed, otherwise the CPU can proceed with its main task.

In the polled mode, the CPU periodically reads an internal flag of 8279 to check for a key pressure. The Keyboard section can interface an array of a maximum of 64 keys with the CPU. The Keyboard entries (key codes) are debounced and stored in an 8-byte FIFO RAM that is further accessed by the CPU to read the key codes. If more than eight characters are entered in the FIFO (i.e. more that eight keys are pressed), before any FIFO read operation, the overrun status is set. If a FIFO contains a valid key entry, the CPU is interrupted (in interrupt mode) or the CPU checks the status (in polling) to read the entry.

Once the CPU reads a key entry, the FIFO is updated, i.e. the key entry is pushed out of the FIFO to generate space for new entries. The 8279 normally provides a maximum of sixteen 7-seg display interface with CPU It contains a 16-byte display RAM that can be used either as an integrated block of 16x8-bits or two 16x4-bit block of RAM. The data entry to RAM block is controlled by CPU using the command words of the 8279.

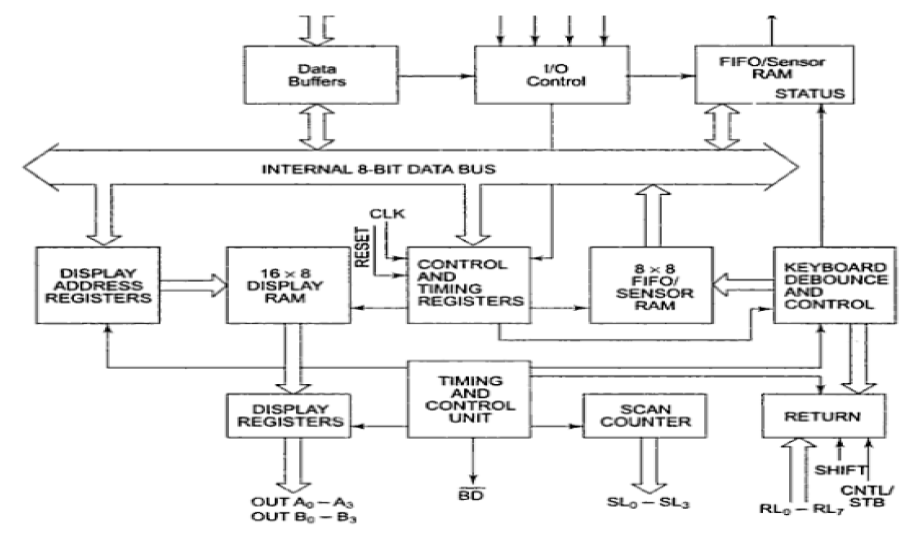
**Architecture and Signal Descriptions of 8279**

The Keyboard display controller chip 8279 provides1. A set of four scan lines and eight return lines for interfacing keyboards. A set of eight output lines for interfacing display.

**I/O Control and Data Buffer**

The I/O control section controls the flow of data to/from the 8279. The data buffer interfaces the external bus of the system with internal bus of 8279. the I/O section is enabled only if D is low.

**8279 Internal Architecture**

****

The pin Ao, RD and WR select the command, status or data read/write operations carried out by the CPU with 8279.

**Control and Timing Register and Timing Control**

These registers store the keyboard and display modes and other operating conditions programmed by CPU. The registers are written with Ao=1 and WR =0. The timing and control unit controls the basic timings for the operation of the circuit. Scan Counter divide down the operating frequency of 8279 to derive scan keyboard and scan display frequencies.

**Scan Counter**

The Scan Counter has two modes to scan the key matrix and refresh the display.In the Encoded mode, the counter provides a binary count that is to be externally decoded to provide the scan lines for keyboard and display (four externally decoded scan lines may drive up to 16 displays).

In the decoded scan mode, the counter internally decodes the least significant 2 bits and provides a decoded 1 out of 4 scan on SL0-SL3 (four internally decoded scan lines may drive up to 4 Displays). The

**Return Buffers and Keyboard Debounce and Control**

This section scans for a Key closure row-wise. If it is detected, the Keyboard debounce unit debounces the key entry (i.e. wait for 10 ms). After the debounce period, if the key continues to be detected. The code of the Key is directly transferred to the sensor RAM along with SHIFT and CONTROL key status.

**FIFO/Sensor RAM and Status Logic**

In Keyboard or strobed input mode, this block acts as 8-byte first-in-first-out (FIFO) RAM. Each key code of the pressed key is entered in the order of the entry, and in the meantime, read by the CPU, till the RAM becomes empty. The status logic generates an interrupt request after each FIFO read operation till the FIFO is empty.

In scanned sensor matrix mode, this unit acts as sensor RAM. Each row of the sensor RAM is loaded with the status of the corresponding row of sensors in the matrix. If a sensor changes its state, the IRQ line goes high to interrupt the CPU.

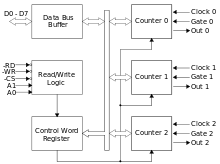
**Display Address Registers and Display RAM.**

The Display address registers hold the addresses of the word currently being written or read by the CPU to or from the display RAM. The contents of the registers are automatically updated by 8279 to accept the next data entry by CPU. The 16-byte display RAM contains the 16-byte of data to be displayed on the sixteen 7-seg displays in theencoded scan mode.

The 8253 was used in IBM PC compatibles since their introduction in 1981.[[1]](https://en.wikipedia.org/wiki/Intel_8253#cite_note-1) In modern times, this PIT is not included as a separate chip in an x86 PC. Rather, its functionality is included as part of the motherboard's [southbridge](https://en.wikipedia.org/wiki/Southbridge_(computing)) chipset. In some modern chipsets, this change may show up as measurable timing differences in accessing a PIT using the[x86](https://en.wikipedia.org/wiki/X86) [I/O address](https://en.wikipedia.org/wiki/I/O_address) space. Reads and writes to such a PIT's registers in the I/O address space may complete much faster.

Newer motherboards also include a counter through the [Advanced Configuration and Power Interface](https://en.wikipedia.org/wiki/Advanced_Configuration_and_Power_Interface) (ACPI), a counter on the Local Advanced Programmable Interrupt Controller ([Local APIC](https://en.wikipedia.org/wiki/Local_APIC)), and a [High Precision Event Timer](https://en.wikipedia.org/wiki/High_Precision_Event_Timer). The CPU itself also provides the [Time Stamp Counter](https://en.wikipedia.org/wiki/Time_Stamp_Counter) (TSC) facility.

## Features[[edit](https://en.wikipedia.org/w/index.php?title=Intel_8253&action=edit&section=2)]

[](https://en.wikipedia.org/wiki/File:Intel_8253_block_diagram.svg)

Block diagram of Intel 8253

The timer has three counters, called channels. Each channel can be programmed to operate in one of six modes. Once programmed, the channels can perform their tasks independently. The timer is usually assigned to [IRQ](https://en.wikipedia.org/wiki/Interrupt_request)-0 (highest priority hardware interrupt) because of the critical function it performs and because so many devices depend on it.[[2]](https://en.wikipedia.org/wiki/Intel_8253#cite_note-2)

### PROGRAMMABLE INTERVAL TIMERS (INTEL 8253/8254)

There are 3 [counters](https://en.wikipedia.org/wiki/Counter_(digital)) (or [timers](https://en.wikipedia.org/wiki/Timer)), which are labeled as "Counter 0", "Counter 1" and "Counter 2".[[3]](https://en.wikipedia.org/wiki/Intel_8253#cite_note-Intel_8254-3) Each counter has 2 input pins – "CLK" ([clock](https://en.wikipedia.org/wiki/Clock) input) and "GATE" – and 1-pin, "OUT", for data output. The 3 counters are 16-bit down counters independent of each other, and can be easily read by the [CPU](https://en.wikipedia.org/wiki/Central_processing_unit).

In the original IBM PCs, the first counter (selected by setting A1=A0=0, see [Control Word Register](https://en.wikipedia.org/wiki/Intel_8253#Control_Word_Register) below) is used to generate a[timekeeping](https://en.wikipedia.org/wiki/Clock_signal) interrupt. The second counter (A1=0, A0=1) is used to trigger the refresh of [DRAM](https://en.wikipedia.org/wiki/DRAM) memory. The last counter (A1=1, A0=0) is used to generate tones via the [PC speaker](https://en.wikipedia.org/wiki/PC_speaker).

Besides the counters, a typical Intel 8253 microchip also contains the following components:

### Data/Bus Buffer

This block contains the logic to buffer the data bus to / from the microprocessor, and to the internal registers. It has 8 input pins, usually labelled as D7..D0, where D7 is the [LSB](https://en.wikipedia.org/wiki/Least_significant_bit).

### Read/Write Logic

The Read/Write Logic block has 5 pins, which are listed below. Notice that "/X" denotes an active low signal.

* /RD: read signal
* /WR: write signal
* /CS: chip select signal
* A0, A1: address lines

Operation mode of the PIT is changed by setting the above hardware signals. For example, to write to the Control Word Register, one needs to set /CS=0, /RD=1, /WR=0, A1=A0=1.

### Control Word Register

Port 43h R/W  
Port 53h R/W – second chip…  
  
This register contains the programmed information which will be sent (by the [microprocessor](https://en.wikipedia.org/wiki/Microprocessor)) to the device. It defines how the PIT logically works. Each access to these ports takes about 1 µs.

To initialize the counters, the microprocessor must write a control word (CW) in this register. This can be done by setting proper values for the pins of the Read/Write Logic block and then by sending the control word to the Data/Bus Buffer block.

The control word register contains 8 bits, labeled D7..D0 (D7 is the [MSB](https://en.wikipedia.org/wiki/Most_significant_bit)).

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Bit#** | **D7** | **D6** | **D5** | **D4** | **D3** | **D2** | **D1** | **D0** | **Short Description** |
| **Name** | **SC1** | **SC0** | **RW1** | **RW0** | **M2** | **M1** | **M0** | **BCD** |  |
|  | 0 | 0 | x | X | x | x | x | x | Counter 0 at port 40h R/W |
|  | 0 | 1 | x | X | x | x | x | x | Counter 1 at port 41h R/W |
|  | 1 | 0 | x | X | x | x | x | x | Counter 2 at port 42h R/W |
|  |  |  |  |  |  |  |  |  |  |
|  | x | x | 0 | 0 | x | x | x | x | Counter Latch, value can be read out in the way RW1, RW0 was set before. The value is held until it is read out or overwritten. |
|  | x | x | 0 | 1 | x | x | x | x | Read/Write bits 0..7 of counter value |
|  | x | x | 1 | 0 | x | x | x | x | Read/Write bits 8..15 of counter value |
|  | x | x | 1 | 1 | x | x | x | x | 2xRead/2xWrite bits 0..7 then 8..15 of counter value |
|  |  |  |  |  |  |  |  |  |  |
|  | x | x | x | x | 0 | 0 | 0 | x | Mode 0: Interrupt on Terminal Count |
|  | x | x | x | x | 0 | 0 | 1 | x | Mode 1: Hardware Retriggerable One-Shot |
|  | x | x | x | x | 0 | 1 | 0 | x | Mode 2: Rate Generator |
|  | x | x | x | x | 0 | 1 | 1 | x | Mode 3: Square Wave |
|  | x | x | x | x | 1 | 0 | 0 | x | Mode 4: Software Triggered Strobe |
|  | x | x | x | x | 1 | 0 | 1 | x | Mode 5: Hardware Triggered Strobe (Retriggerable) |
|  |  |  |  |  |  |  |  |  |  |
|  | x | x | x | x | x | x | x | 0 | Counter is a 16 bit binary counter(0..65535,FFFFh) |
|  | x | x | x | x | x | x | x | 1 | Counter is a 16 bit decimal counter 4 x 4bit decades(0..9999) |
|  |  |  |  |  |  |  |  |  |  |
| **Name** | **1** | **1** | **\_\_\_\_\_**  **count** | **\_\_\_\_\_**  **status** | **C2** | **C1** | **C0** | **0** |  |
|  | 1 | 1 | 0 | 1 | x | x | x | 0 | Counter(C0..C2) value(s) can be read out. |
|  | 1 | 1 | 1 | 0 | x | x |  |  |  |

When setting the PIT, the microprocessor first sends a control message, then a count message to the PIT. The counting process will start after the PIT has received these messages, and, in some cases, if it detects the rising [edge](https://en.wikipedia.org/wiki/Signal_edge) from the GATE input signal.

On PCs the address for timer0 (chip) is at port 40h..43h like described and the second timer1 (chip) is at 50h..53h.

### Status Byte

8 bit  
The Status Byte is read like an 8 bit counter value (port 40h..42h R).

Bit# D7 D6 D5 D4 D3 D2 D1 D0

Name output null RW1 RW0 M2 M1 M0 BCD

count

-------------------------------------------

0 x x x x x x x Out Pin is 0

1 x x x x x x x Out Pin is 1

-------------------------------------------

x 0 x x x x x x The value of the latch is loaded into the counter.

A new value can be written to the latch.

x 1 x x x x x x Counter value is 0.

-------------------------------------------

x x = = = = = = like defined in the Control Word Register

## Operation Modes

The D3, D2, and D1 bits of the Control Word set the operating mode of the timer. There are 6 modes in total; for modes 2 and 3, the D3 bit is ignored, so the missing modes 6 and 7 are aliases for modes 2 and 3. Notice that, for modes 0, 2, 3 and 4, GATE must be set to HIGH to enable counting. For mode 5, the rising edge of GATE starts the count. For details on each mode, see the reference links.

### Mode 0 (000): Interrupt on Terminal Count

Mode 0 is used for the generation of accurate time delay under software control. In this mode, the counter will start counting from the initial COUNT value loaded into it, down to 0. Counting rate is equal to the input clock frequency.

The OUT pin is set low after the Control Word is written, and counting starts one clock cycle after the COUNT programmed. OUT remains low until the counter reaches 0, at which point OUT will be set high until the counter is reloaded or the Control Word is written. The Gate signal should remain active high for normal counting. If Gate goes low counting gets terminated and current count is latched till Gate pulse goes high again. the first byte of the new count when loaded in the count register,stop the previous count.

### Mode 1 (001): Programmable One Shot

In this mode 8253 can be used as [Monostable Multivibrator](https://en.wikipedia.org/wiki/Monostable_Multivibrator). GATE input is used as trigger input.

OUT will be initially high. OUT will go low on the Clock pulse following a trigger to begin the one-shot pulse, and will remain low until the Counter reaches zero. OUT will then go high and remain high until the CLK pulse after the next trigger.

After writing the Control Word and initial count, the Counter is armed. A trigger results in loading the Counter and setting OUT low on the next CLK pulse, thus starting the one-shot pulse. An initial count of N will result in a one-shot pulse N CLK cycles in duration.

The one-shot is retriggerable, hence OUT will remain low for N CLK pulses after any trigger. The one-shot pulse can be repeated without rewriting the same count into the counter. GATE has no effect on OUT. If a new count is written to the Counter during a oneshot pulse, the current one-shot is not affected unless the counter is retriggered. In that case, the Counter is loaded with the new count and the oneshot pulse continues until the new count expires.

### Mode 2 (X10): Rate Generator

In this mode, the device acts as a divide-by-n counter, which is commonly used to generate a real-time clock interrupt.

Like other modes, counting process will start the next clock cycle after COUNT is sent. OUT will then remain high until the counter reaches 1, and will go low for one clock pulse. OUT will then go high again, and the whole process repeats itself.

The time between the high pulses depends on the preset count in the counter's register, and is calculated using the following formula:

Value to be loaded into counter = {{\it {f_{input}}} \over {\it {f_{output}}}}

Note that the values in the COUNT register range from n to 1; the register never reaches zero.

### Mode 3 (X11): Square Wave Generator

This mode is similar to mode 2. However, the duration of the high and low clock pulses of the output will be different from mode 2.

Suppose n is the number loaded into the counter (the COUNT message), the output will be

* high for {n \over 2} counts, and low for {n \over 2} counts, if n is even.
* high for {n+1 \over 2} counts, and low for {n-1 \over 2} counts, if n is odd.

### Mode 4 (100): Software Triggered Strobe

After Control Word and COUNT is loaded, the output will remain high until the counter reaches zero. The counter will then generate a low pulse for 1 clock cycle (a strobe) – after that the output will become high again.

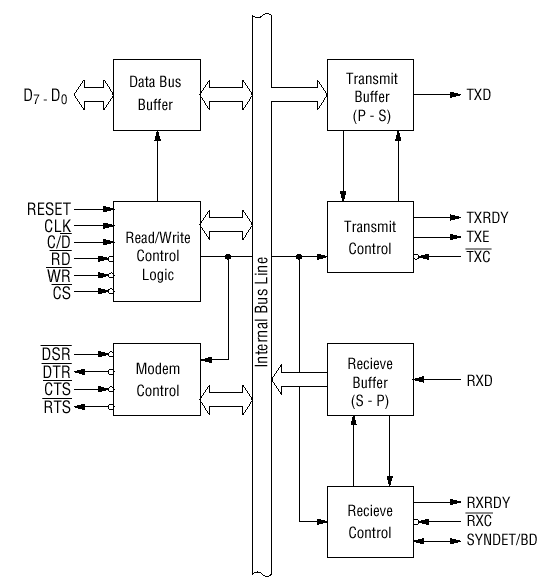
### Mode 5 (101): Hardware Triggered Strobe

This mode is similar to mode 4. However, the counting process is triggered by the GATE input.

After receiving the Control Word and COUNT, the output will be set high. Once the device detects a rising edge on the GATE input, it will start counting. When the counter reaches 0, the output will go low for one clock cycle – after that it will become high again, to repeat the cycle on the next rising edge of GATE.

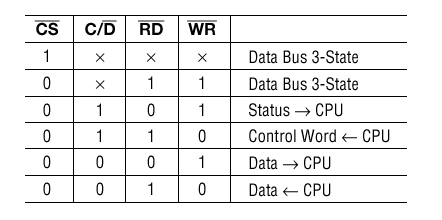
## 8251 UNIVERSAL SYNCHRONOUS ASYNCHRONOUS RECEIVER TRANSMITTER (USART)

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



**Block diagram of the 8251 USART (Universal Synchronous Asynchronous Receiver Transmitter)**

The 8251 functional configuration is programed by software. Operation between the 8251 and a CPU is executed by program control. Table 1 shows the operation between a CPU and the device.



**Table 1 Operation between a CPU and 8251**

### Control Words

There are two types of control word.

1. Mode instruction (setting of function)

2. Command (setting of operation)

#### 1) Mode Instruction

Mode instruction is used for setting the function of the 8251. Mode instruction will be in "wait for write" at either internal reset or external reset. That is, the writing of a control word after resetting will be recognized as a "mode instruction."

Items set by mode instruction are as follows:

• Synchronous/asynchronous mode

• Stop bit length (asynchronous mode)

• Character length

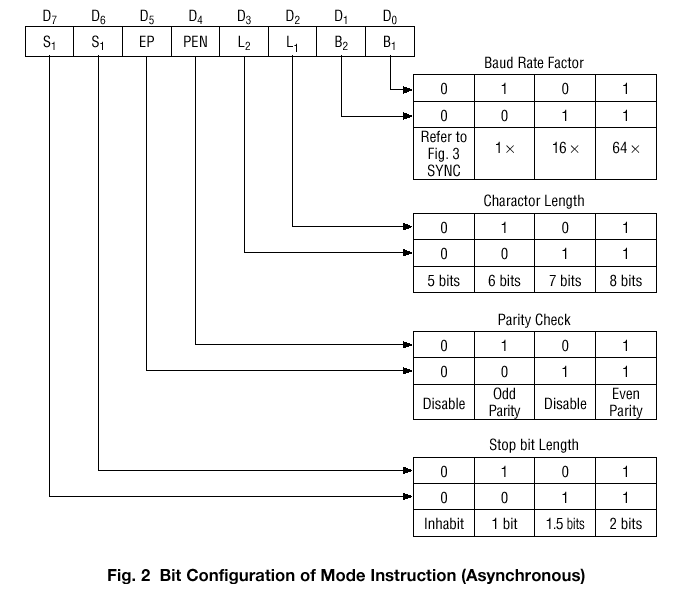
• Parity bit

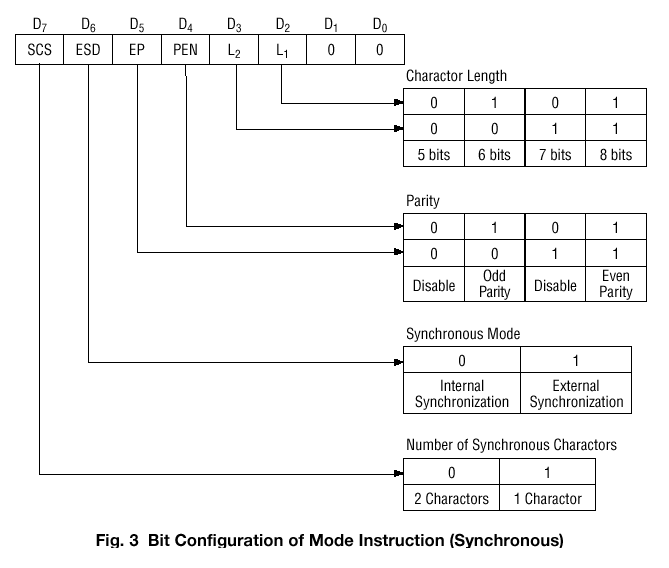
• Baud rate factor (asynchronous mode)

• Internal/external synchronization (synchronous mode)

• Number of synchronous characters (Synchronous mode)

The bit configuration of mode instruction is shown in Figures 2 and 3. In the case of synchronous mode, it is necessary to write one-or two byte sync characters. If sync characters were written, a function will be set because the writing of sync characters constitutes part of mode instruction.





#### 2) Command

Command is used for setting the operation of the 8251. It is possible to write a command whenever necessary after writing a mode instruction and sync characters.

Items to be set by command are as follows:

• Transmit Enable/Disable

• Receive Enable/Disable

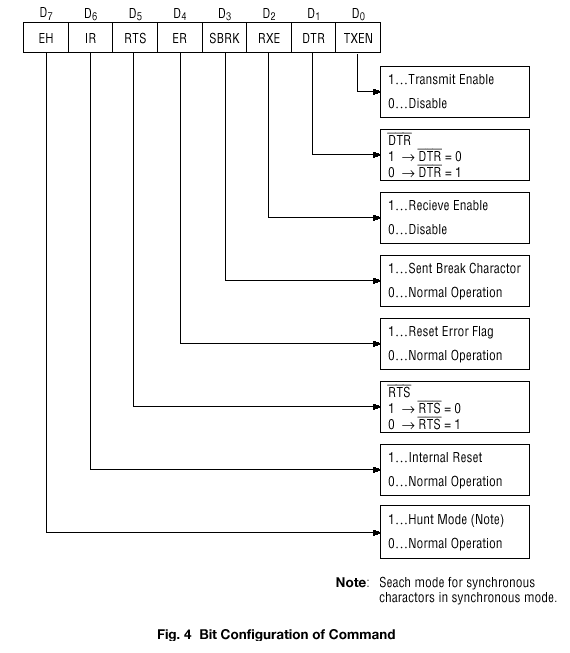
• DTR, RTS Output of data.

• Resetting of error flag.

• Sending to break characters

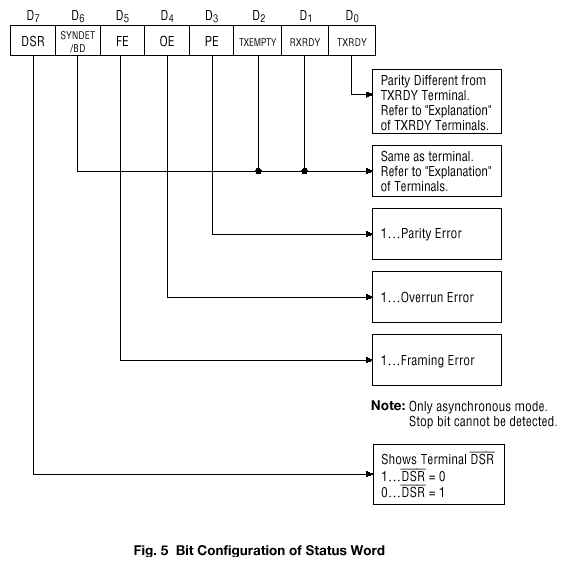
• Internal resetting

• Hunt mode (synchronous mode)



#### Status Word

It is possible to see the internal status of the 8251 by reading a status word. The bit configuration of status word is shown in Fig. 5.



#### Pin Description

**D 0 to D 7 (l/O terminal)**

This is bidirectional data bus which receive control words and transmits data from the CPU and sends status words and received data to CPU.

**RESET (Input terminal)**

A "High" on this input forces the 8251 into "reset status." The device waits for the writing of "mode instruction." The min. reset width is six clock inputs during the operating status of CLK.

**CLK (Input terminal)**

CLK signal is used to generate internal device timing. CLK signal is independent of RXC or TXC. However, the frequency of CLK must be greater than 30 times the RXC and TXC at Synchronous mode and Asynchronous "x1" mode, and must be greater than 5 times at Asynchronous "x16" and "x64" mode.

**WR (Input terminal)**

This is the "active low" input terminal which receives a signal for writing transmit data and control words from the CPU into the 8251.

**RD (Input terminal)**

This is the "active low" input terminal which receives a signal for reading receive data and status words from the 8251.

**C/D (Input terminal)**

This is an input terminal which receives a signal for selecting data or command words and status words when the 8251 is accessed by the CPU. If C/D = low, data will be accessed. If C/D = high, command word or status word will be accessed.

**CS (Input terminal)**

This is the "active low" input terminal which selects the 8251 at low level when the CPU accesses. Note: The device won’t be in "standby status"; only setting CS = High.

**TXD (output terminal)**

This is an output terminal for transmitting data from which serial-converted data is sent out. The device is in "mark status" (high level) after resetting or during a status when transmit is disabled. It is also possible to set the device in "break status" (low level) by a command.

**TXRDY (output terminal)**

This is an output terminal which indicates that the 8251is ready to accept a transmitted data character. But the terminal is always at low level if CTS = high or the device was set in "TX disable status" by a command. Note: TXRDY status word indicates that transmit data character is receivable, regardless of CTS or command. If the CPU writes a data character, TXRDY will be reset by the leading edge or WR signal.

**TXEMPTY (Output terminal)**

This is an output terminal which indicates that the 8251 has transmitted all the characters and had no data character. In "synchronous mode," the terminal is at high level, if transmit data characters are no longer remaining and sync characters are automatically transmitted. If the CPU writes a data character, TXEMPTY will be reset by the leading edge of WR signal. Note : As the transmitter is disabled by setting CTS "High" or command, data written before disable will be sent out. Then TXD and TXEMPTY will be "High". Even if a data is written after disable, that data is not sent out and TXE will be "High".After the transmitter is enabled, it sent out. (Refer to Timing Chart of Transmitter Control and Flag Timing)

**TXC (Input terminal)**

This is a clock input signal which determines the transfer speed of transmitted data. In "synchronous mode," the baud rate will be the same as the frequency of TXC. In "asynchronous mode", it is possible to select the baud rate factor by mode instruction. It can be 1, 1/16 or 1/64 the TXC. The falling edge of TXC sifts the serial data out of the 8251.

**RXD (input terminal)**

This is a terminal which receives serial data.

**RXRDY (Output terminal)**

This is a terminal which indicates that the 8251 contains a character that is ready to READ. If the CPU reads a data character, RXRDY will be reset by the leading edge of RD signal. Unless the CPU reads a data character before the next one is received completely, the preceding data will be lost. In such a case, an overrun error flag status word will be set.

**RXC (Input terminal)**

This is a clock input signal which determines the transfer speed of received data. In "synchronous mode," the baud rate is the same as the frequency of RXC. In "asynchronous mode," it is possible to select the baud rate factor by mode instruction. It can be 1, 1/16, 1/64 the RXC.

**SYNDET/BD (Input or output terminal)**

This is a terminal whose function changes according to mode. In "internal synchronous mode." this terminal is at high level, if sync characters are received and synchronized. If a status word is read, the terminal will be reset. In "external synchronous mode, "this is an input terminal. A "High" on this input forces the 8251 to start receiving data characters.

In "asynchronous mode," this is an output terminal which generates "high level"output upon the detection of a "break" character if receiver data contains a "low-level" space between the stop bits of two continuous characters. The terminal will be reset, if RXD is at high level. After Reset is active, the terminal will be output at low level.

**DSR (Input terminal)**

This is an input port for MODEM interface. The input status of the terminal can be recognized by the CPU reading status words.

**DTR (Output terminal)**

This is an output port for MODEM interface. It is possible to set the status of DTR by a command.

**CTS (Input terminal)**

This is an input terminal for MODEM interface which is used for controlling a transmit circuit. The terminal controls data transmission if the device is set in "TX Enable" status by a command. Data is transmitable if the terminal is at low level.

**RTS (Output terminal)**

This is an output port for MODEM interface. It is possible to set the status RTS by a command.

* **INTRODUCTION:8259:-PIC**

**1. It is programmed to work with either 8085 or 8086 processor.**

**2. It manage 8-interrupts according to the instructions written into its control registers.**

**3. In 8086 processor, it supplies the type number of the interrupt and the type number is  programmable. In 8085 processor, the interrupt vector address is programmable. The priorities of the interrupts are programmable.**

**4. The interrupts can be masked or unmasked individually.**

**5. The 8259s can be cascaded to accept a maximum of 64 interrupts.**

**FUNCTIONAL BLOCK DIAGRAM OF 8259:**

**It has eight functional blocks. They are,**

1. **Control logic**
2. **Read Write logic**
3. **Data bus buffer**
4. **Interrupt Request Register (IRR)**
5. **In-Service Register (ISR)**
6. **Interrupt Mask Register (IMR)**
7. **Priority Resolver (PR)**
8. **Cascade buffer.**

**The data bus and its buffer are used for the following activities.**

1. **The processor sends control word to data bus buffer through D0-D7.**
2. **The processor read status word from data bus buffer through D0-D7**
3. **From the data bus buffer the 8259 send type number (in case of 8086) or the call opcode and     address (in case of 8085) through D0-D7 to the processor.**

* **The processor uses the RD (low), WR (low) and A0 to read or write 8259.**
* **The 8259 is selected by CS (low).**
* **The IRR has eight input lines (IR0-IR7) for interrupts. When these lines go high, the request is stored in IRR. It registers a request only if the interrupt is unmasked.**
* **Normally IR0 has highest priority and IR7 has the lowest priority. The priorities of the interrupt request input are also programmable.**
* **First the 8259 should be programmed by sending Initialization Command Word (ICW) and Operational Command Word (OCW). These command words will inform 8259 about the following,**

1. **Type of interrupt signal (Level triggered / Edge triggered).**
2. **Type of processor (8085/8086).**
3. **Call address and its interval (4 or 8)**
4. **Masking of interrupts.**
5. **Priority of interrupts.**
6. **Type of end of interrupts.**

* **The interrupt mask register (IMR) stores the masking bits of the interrupt lines to be masked. The relevant information is send by the processor through OCW.**
* **The in-service register keeps track of which interrupt is currently being serviced.**
* **The priority resolver examines the interrupt request, mask and in-service registers and determines whether INT signal should be sent to the processor or not.**
* **The cascade buffer/comparator is used to expand the interrupts of 8259.**
* **In cascade connection one 8259 will be directly interrupting 8086 and it is called master 8259.**
* **To each interrupt request input of master 8259 (IR0-IR7), one slave 8259 can be connected. The 8259s interrupting the master 8259 are called slave 8259s.**
* **Each 8259 has its own addresses so that each 8259 can be programmed independently by sending command words and independently the status bytes can be read from it.**

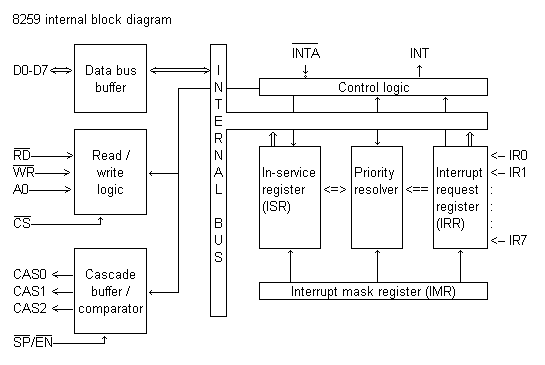
## Features:

* 8 levels of interrupts.
* Can be cascaded in master-slave configuration to handle 64 levels of interrupts.
* Internal priority resolver.
* Fixed priority mode and rotating priority mode.
* Individually maskable interrupts.
* Modes and masks can be changed dynamically.
* Accepts IRQ, determines priority, checks whether incoming priority > current level being serviced, issues interrupt signal.
* In 8085 mode, provides 3 byte CALL instruction. In 8086 mode, provides 8 bit vector number.
* Polled and vectored mode.
* Starting address of ISR or vector number is programmable.
* No clock required.

## Pinout

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| [[8259 pinout]](http://www.thesatya.com/images/pin8259.png) | |  |  | | --- | --- | | D0-D7 | Bi-directional, tristated, buffered data lines. Connected to data bus directly or through buffers | | RD-bar | Active low read control | | WR-bar | Active low write control | | A0 | Address input line, used to select control register | | CS-bar | Active low chip select | | CAS0-2 | Bi-directional, 3 bit cascade lines. In master mode, PIC places slave ID no. on these lines. In slave mode, the PIC reads slave ID no. from master on these lines. It may be regarded as slave-select. | | SP-bar / EN-bar | Slave program / enable. In non-buffered mode, it is SP-bar input, used to distinguish master/slave PIC. In buffered mode, it is output line used to enable buffers | | INT | Interrupt line, connected to INTR of microprocessor | | INTA-bar | Interrupt ack, received active low from microprocessor | | IR0-7 | Asynchronous IRQ input lines, generated by peripherals. | |

## Block diagram

[](http://www.thesatya.com/images/blk8259.png)

## ICW1 (Initialisation Command Word One)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | | A0 | | 0 | | |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 | | A7 | A6 | A5 | 1 | LTIM | ADI | SNGL | IC4 | |

D0: IC4: 0=no ICW4, 1=ICW4 required   
D1: SNGL: 1=Single PIC, 0=Cascaded PIC   
D2: ADI: Address interval. Used only in 8085, not 8086. 1=ISR's are 4 bytes apart (0200, 0204, etc) 0=ISR's are 8 byte apart (0200, 0208, etc)   
D3: LTIM: level triggered interrupt mode: 1=All IR lines level triggered. 0=edge triggered   
D4-D7: A5-A7: 8085 only. ISR address lower byte segment. The lower byte is

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| A7 | A6 | A5 | A4 | A3 | A2 | A1 | A0 |

of which A7, A6, A5 are provided by D7-D5 of ICW1 (if ADI=1), or A7, A6 are provided if ADI=0. A4-A0 (or A5-A0) are set by 8259 itself:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| ADI=1 (spacing 4 bytes)   |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | | IRQ | A7 | A6 | A5 | A4 | A3 | A2 | A1 | A0 | | IR0 | A7 | A6 | A5 | 0 | 0 | 0 | 0 | 0 | | IR1 | A7 | A6 | A5 | 0 | 0 | 1 | 0 | 0 | | IR2 | A7 | A6 | A5 | 0 | 1 | 0 | 0 | 0 | | IR3 | A7 | A6 | A5 | 0 | 1 | 1 | 0 | 0 | | IR4 | A7 | A6 | A5 | 1 | 0 | 0 | 0 | 0 | | IR5 | A7 | A6 | A5 | 1 | 0 | 1 | 0 | 0 | | IR6 | A7 | A6 | A5 | 1 | 1 | 1 | 0 | 0 | | IR7 | A7 | A6 | A5 | 1 | 1 | 1 | 0 | 0 | | ADI=0 (spacing 8 bytes)   |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | | IRQ | A7 | A6 | A5 | A4 | A3 | A2 | A1 | A0 | | IR0 | A7 | A6 | 0 | 0 | 0 | 0 | 0 | 0 | | IR1 | A7 | A6 | 0 | 0 | 1 | 0 | 0 | 0 | | IR2 | A7 | A6 | 0 | 1 | 0 | 0 | 0 | 0 | | IR3 | A7 | A6 | 0 | 1 | 1 | 0 | 0 | 0 | | IR4 | A7 | A6 | 1 | 0 | 0 | 0 | 0 | 0 | | IR5 | A7 | A6 | 1 | 0 | 1 | 0 | 0 | 0 | | IR6 | A7 | A6 | 1 | 1 | 0 | 0 | 0 | 0 | | IR7 | A7 | A6 | 1 | 1 | 1 | 0 | 0 | 0 | |

## ICW2 (Initialisation Command Word Two)

Higher byte of ISR address (8085), or 8 bit vector address (8086).

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | | A0 | | 1 | | |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 | | A15 | A14 | A13 | A12 | A11 | A10 | A9 | A8 | |

## ICW3 (Initialisation Command Word Three)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | | A0 | | 1 | | |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | |  | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 | | Master | S7 | S6 | S5 | S4 | S3 | S2 | S1 | S0 | | Slave | 0 | 0 | 0 | 0 | 0 | ID3 | ID2 | ID1 | |

* Master mode: 1 indicates slave is present on that interrupt, 0 indicates direct interrupt
* Slave mode: ID3-ID2-ID1 is the slave ID number. Slave 4 on IR4 has ICW3=04h (0000 0100)

## ICW4 (Initialisation Command Word Four)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | | A0 | | 1 | | |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 | | 0 | 0 | 0 | SFNM | BUF | M/S | AEOI | Mode | |

* SFNM: 1=Special Fully Nested Mode, 0=FNM
* M/S: 1=Master, 0=Slave
* AEOI: 1=Auto End of Interrupt, 0=Normal
* Mode: 0=8085, 1=8086

## OCW1 (Operational Command Word One)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | | A0 | | 1 | | |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 | | M7 | M6 | M5 | M4 | M3 | M2 | M1 | M0 | |

IRn is masked by setting Mn to 1; mask cleared by setting Mn to 0 (n=0..7)

## OCW2 (Operational Command Word Two)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | | A0 | | 1 | | |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 | | R | SL | EOI | 0 | 0 | L3 | L2 | L1 | |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | R | SL | EOI | Action |
| EOI | 0 | 0 | 1 | Non specific EOI (L3L2L1=000) |
| 0 | 1 | 1 | Specific EOI command (Interrupt to clear given by L3L2L1) |
| Auto rotation of priorities (L3L2L1=000) | 1 | 0 | 1 | Rotate priorities on non-specific EOI |
| 1 | 0 | 0 | Rotate priorities in auto EOI mode set |
| 0 | 0 | 0 | Rotate priorities in auto EOI mode clear |
| Specific rotation of priorities (Lowest priority ISR=L3L2L1) | 1 | 1 | 1 | Rotate priority on specific EOI command (resets current ISR bit) |
| 1 | 1 | 0 | Set priority (does not reset current ISR bit) |
| 0 | 1 | 0 | No operation |

## OCW3 (Operational Command Word Three)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | | A0 | | 1 | | |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 | | D7 | ESMM | SMM | 0 | 1 | MODE | RIR | RIS | |

|  |  |  |
| --- | --- | --- |
| ESMM | SMM | Effect |
| 0 | X | No effect |
| 1 | 0 | Reset special mask |
| 1 | 1 | Set special mask |

## Interrupt sequence (single PIC)

1. One or more of the IR lines goes high.
2. Corresponding IRR bit is set.
3. 8259 evaluates the request and sends INT to CPU.
4. CPU sends INTA-bar.
5. Highest priority ISR is set. IRR is reset.
6. 8259 releases CALL instruction on data bus.
7. CALL causes CPU to initiate two more INTA-bar's.
8. 8259 releases the subroutine address, first lowbyte then highbyte.
9. ISR bit is reset depending on mode.

DIGITAL TO ANALOG CONVERTER and its interfacing with 8086

A **DAC** is a **Digital-to-Analog Converter**

A DAC [Digital-to-Analog Converter] is an electronic processor inside digital media players that converts digital audio information (comprised of series of 0s and 1s) into an analog audio signal that can be sent to headphones, or better yet, a [headphone amp](http://www.headphone.com/headphone-amps/amplifiers.php). If you want all those digits inside the player to sound as good as when the music was originally recorded, you need to use a [high-quality DAC](http://www.headphone.com/headphone-amps/digital-to-analog-converters.php). The simplest way to get a digital audio signal to an external DAC is via the USB port on your [computer](http://www.headphone.com/learning-center/tips-for-computer-listening.php), but one can also get a digital feed from the optical and coaxial outputs -- also known as SPDiF connections -- found on CD or DVD players, MAC computers, home theatre/stereo receivers, and other streaming audio gadgets of all kinds. Remember that lossy mp3 files will always lack detail and dynamics so the biggest thing to improve your headphones is ripping your [digital music](http://www.headphone.com/learning-center/art-ii-network-architecture.php) in uncompressed formats or at the highest possible bitrate for best sound quality.

###### DAC 0800

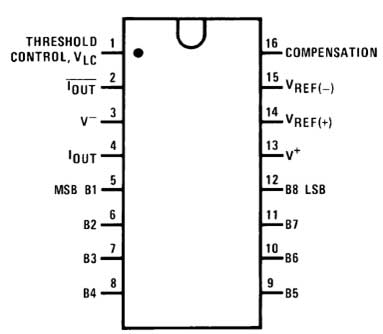
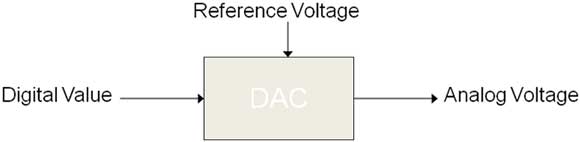


Fig 1: **pin diagram of DAC 0800**



How a DAC works

**Ramp Wave Form Generation**

.OUTPUT 2500AD

CONTROL EQU FFC6H ;control port address for 8255

PORTA EQU FFC0H ;porta address for 8255

PORTB EQU FFC2H ;portb address for 8255

PORTC EQU FFC4H ;portc address for 8255

DSEG SEGMENT

ORG 0000: 4000H

MSG DB' Dac mode ',0h

MSG1 DB 'Ramp wave output',0h

DSEG ENDS

CSEG SEGMENT

ORG 0000 : 5000H

ASSUME CS : CSEG, DS : DSEG

START:

MOV DX, CONTROL

MOV AL, 80H ;initialise all ports as output

OUT DX, AL ;ports

;displaying message on LCD

Call far f800 : 4bb1h ;clear display

Mov di, 80h ;display in upper line

MOV SI, offset MSG

CALL FAR f800 : 4FC0h ;display output routine

MOV DI, C0H ;display in lower line

MOV SI, OFFSET MSG1

CALL FAR F800 : 4FC0H ;display output routine

;ramp wave form generation

MOV BL, 00H

RAMP: MOV DX, PORTB

MOV AL, BL ;increment the digital code

OUT DX, AL ;and output it on to PORTB

INC BL

;check to see if a NMI-INTR

JNZ RAMP

MOV BL, 00H

JMP RAMP

CSEGENDS

END

SQUARE WAVEFORM:

DA00: MOV DX, PORTB

MOV AL, 00H ;outport 00 for 0V level

CALL OUTPUT

MOV AL, 0FFH

CALL OUTPUT

JMP DA00

;routine to output digital value

OUTPUT: OUT DX, AL

CALL DELAY

RET

DELAY: MOV CX,0FFH ;to vary the frequency alter the delay count

LUP1: LOOP LUP1

RET

STEP WAVEFORM:

DA00: MOV DX, PORTB

MOV AL, 00H ;outport 00 for 0V level

CALL OUTPUT

MOV AL, 7FH ;outport 7F for 2.5V level

CALL OUTPUT

MOV AL, FFH ;outport FF for 5V level

CALL OUTPUT

;look for NMI-INTR if user

JMP DA00 ;switch to ramp w/f generation

;routine to output digital value

OUTPUT: OUT DX, AL

MOV CX, FFH

DELAY: LOOP DELAY

RET

TRIANGULAR WAVEFORM:

DA00: MOV DX, PORTB

MOV AL, 00H ;outport 00 for 0V level

UP: CALL OUTPUT

INC AL

CMP AL, 00H

JNZ UP

MOV AL, 0FFH ;to change amplitude change count

UP1: CALL OUTPUT

DEC AL

CMP AL, 0FFH

JNZ UP1

JMP DA00

;routine to output digital value

OUTPUT: OUT DX, AL

CALL DELAY

RET

DELAY: MOV CX, 07H ;to vary the frequency alter the delay count

LUP1: LOOP LUP1

RET

**Step waveform:**

DA00: MOV DX, PORTB

MOV AL, 00H ;outport 00 for 0V level

CALL OUTPUT

MOV AL, 7FH ;outport 7F for 2.5V level

CALL OUTPUT

MOV AL, FFH ;outport FF for 5V level

CALL OUTPUT

;look for NMI-INTR if user

JMP DA00 ;switch to ramp w/f generation

;routine to output digital value

OUTPUT: OUT DX, AL

MOV CX, FFH

DELAY: LOOP DELAY

RET

**Triangular waveform:**

DA00: MOV DX, PORTB

MOV AL, 00H ;outport 00 for 0V level

UP: CALL OUTPUT

INC AL

CMP AL, 00H

JNZ UP

MOV AL, 0FFH ;to change amplitude change count

UP1: CALL OUTPUT

DEC AL

CMP AL, 0FFH

JNZ UP1

JMP DA00

;routine to output digital value

OUTPUT: OUT DX, AL

CALL DELAY

RET

DELAY: MOV CX, 07H ;to vary the frequency alter the delay count

LUP1: LOOP LUP1

RET

#### **Analog to Digital converter and its interfacing with 8086**

###### ADC 0809

The **ADC0808**, **ADC0809** data acquisition component is a Monolithic CMOS device with an 8-bit analog-to-digital converter,8-channel multiplexer and microprocessor compatible control logic. The 8-bit A/D converter uses successive approximations the conversion technique. The converter features a high impedance chopper stabilized comparator, a 256R voltage divider with analog switch tree and a successive approximation register.

###### INTERFACING ADC 0809 WITH 8086

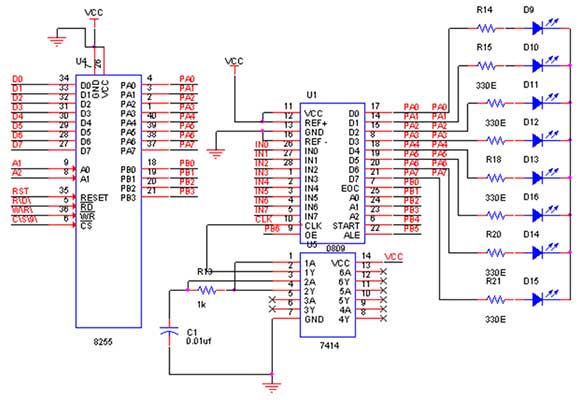
**8086 microprocessor** it doesn’t have an On chip ADC to accept the digital input, it will not accept analog input, so we need a ADC to process the analog signal. For **interfacing ADC 0809** we require 8 data lines. So ADC **0809** is an 8 bit ADC has 8 channels works on successive approximation conversion technique

###### PIN ASSIGNMENT WITH 8086

|  |  |  |  |
| --- | --- | --- | --- |
|  | **8255** | **ADC 0809** | **ADC 0809** |
| **Data lines** | PA0 | D0 | adc-0809 |
| PA1 | D1 |
| PA2 | D2 |
| PA3 | D3 |
| PA4 | D4 |
| PA5 | D5 |
| PA6 | D6 |
| PA7 | D7 |
| **Control Lines** | PB0 | E0C |
| PB1 | A0 |
| PB2 | A1 |
| PB3 | A2 |
| PB4 | Start |
| PB5 | ALE |
| PB6 | OE |
| PB7 | NC |
| **PWR** | 17,19 | Vcc | Supply form 8085/8086/80805 trainer Kit |
| 18,20 | Gnd |

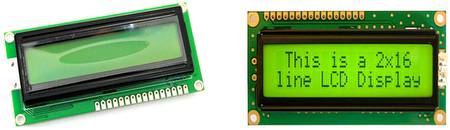
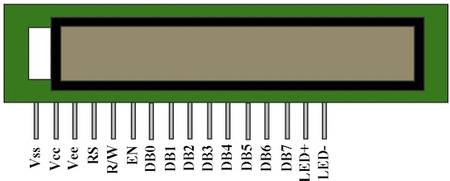
###### 

###### CIRCUIT DIAGRAM TO INTERFACE ADC 0809 WITH 8086

- See more at: https://www.pantechsolutions.net/microcontroller-boards/adc-0809-interfacing-with-8086-ps2-lab-kit#sthash.jEq53FPx.dpuf

**Program:**

1. .OUTPUT 2500AD
2. CONTROL EQU FFC6H ;control port address for 8255
3. PORTA EQU FFC0H ;port a address for 8255
4. PORTB EQU FFC2H ;port b address for 8255
5. PORTC EQU FFC4H ;port c address for 8255
6. KWAD EQU F800 : 4EEDH
7. DBDTA EQUF800 : 4F1FH
8. DSEG SEGMENT
9. ORG 0000 : 4000H
10. MSG DB 'Enter channel No',0h
11. DSEG ENDS
12. CSEG SEGMENT
13. ASSUME CS : CSEG, DS : DSEG
14. ORG 0000 : 5000H
15. ;displaying message on LCD
16. Callfar f800 : 4bb1h ;clear display
17. movdi, 80h ;display in upper line
18. MOV SI, offset MSG ;
19. CALLFAR f800 : 4FC0h ;display output routine
20. MOV AX, 0000H
21. MOV DS, AX
22. ;AD00: CALL FAR KWAD ;get key for channel selection
23. MOV CX, SI
24. MOV AL, 90H ;control word for PPI
25. MOV DX, CONTROL
26. OUT DX, AL ;portA->i/p port,portB->o/p port
27. ;portC->o/p port.
28. AD00: MOV AL, CL ;output channel number
29. MOV DX, PORTC
30. OUT DX, AL ;start conversion
31. MOV AL, 0FH ;PC7 (START/ALE) set
32. MOV DX, CONTROL
33. OUT DX, AL
34. PUSH CX
35. MOV CX, 3FFFH
36. DEL1: LOOP DEL1
37. POP CX
38. MOV AL, 0EH ;PC7 reset
39. MOV DX, CONTROL
40. OUT DX, AL ;look for EOC
41. MOV AL, 0CH ;reset PC6 to read EOC
42. OUT DX, AL
43. AD01: MOV DX, PORTA
44. IN AL, DX ;poll the EOC line which
45. AND AL, 80H ;is connected to PA7 line
46. CMP AL, 80H
47. JNZ AD01 ;if EOC (PA7) is high read the digital value otherwise again check for EOC (PA7) line
48. MOV AL, 0DH ;set OE (PC6) to read value
49. MOV DX, CONTROL
50. OUT DX, AL ;before reading data from ADC set PC6 line
51. MOV DX, PORTA
52. IN AL, DX ;read digital value
53. MOV AH, 00H
54. MOV SI, AX
55. PUSH CX
56. CALL FAR DBDTA ;display digital value
57. POP CX
58. JMP AD00
59. CSEG ENDS
60. END

**LCD(Liquid Crystal Display):**   
LCD (Liquid Crystal Display) is very popular for displaying in Embedded Applications. LCDs are very cheap and easy to interface with microprocessors; LCDs are widely used in devices like telephones, vending machines, washing machines, toys etc.   
LCD comes in several varieties i.e. 16\*2, 20\*2, 20\*4 etc. These different LCD varieties can display different number of characters i.e. 16\*2 can display 32 characters at a time.   
   
**Figure:** LCD module  
Liquid Crystal displays are created by sandwiching a thin 10-12 µm layer of aliquid-crystal fluid between two glass plates. A transparent, electricallyconductive film or backplane is put on the rear glass sheet. Transparent sectionsof conductive film in the shape of the desired characters are coated on the frontglass plate.   
  
When a voltage is applied between a segment and the backplane, an electric fieldis created in the region under the segment. This electric field changes thetransmission of light through the region under the segment film.   
  
**Characteristics:**   
Each module contains a CMOS controller and all necessary drivers which have low power consumption. The controller is equipped with an internal character generator ROM, RAM and RAM for display data. All display functions are controllable by instructions making interfacing practical.   
  
**PIN CONFIGURATION:**   
There are total 16 pins in this LCD module.   
   
**Figure:** LCD with its pin configuration  
  
The following table describes the pin configuration of the LCD Module

|  |  |  |  |
| --- | --- | --- | --- |
| **Pin No** | **Symbol** | **Description** | **Function** |
| 1 | VSS | Ground | 0V (GND) |
| 2 | VCC | Power supply for Logic Circuit | +5V |
| 3 | VEE | LCD Contrast Adjustment |  |
| 4 | RS | INSTRUCTION/DATA RegisterSelection | RS = 0 : INSTR Register  RS = 1 : DATA Register |
| 5 | R/W | READ/WRITE Selection | R/W = 0 : Register WRITE  R/W = 1 : Register READ |
| 6 | E | ENABLE Signal | Sends data to data pins when a high to low pulse applied |
| 7 | DB0 | DATA INPUT/OUTPUT LINES | 8 BIT: DB0-DB7 |
| 8 | DB1 |
| 9 | DB2 |
| 10 | DB3 |
| 11 | DB4 |
| 12 | DB5 |
| 13 | DB6 |
| 14 | DB7 |
| 15 | LED+ | Supply Voltage For LED+ | +5V |
| 16 | LED- | Supply Voltage For LED- | 0V |

**Table:** The Pin Configuration of the LCD Module  
  
**REGISTERS:**   
There are mainly 2 registers in our LCD module.

* **Instruction register**
* **Data register**

The IR is a write only register to store instruction codes like “Display Clear” or “Cursor Shift” as well as addresses of the Display Data RAM (DD RAM) or the Character Generator RAM (CG RAM).   
  
The DR is a read/write register used for temporarily storing data to be read/written to/from the DD RAM or CG RAM. Data written into the DR is automatically written into DD RAM or CG RAM by an internal operation of the display controller. The DR is also used to store data when reading out data from DD RAM or CG RAM.   
  
When address information is written into IR, data is read out from DD RAM or CG RAM to DR by an internal operation. Data transfer is then completed by reading the DR. After performing a read from the DR, data in the DD RAM or CG RAM at the next address is sent to the DR for the next read cycle. The register select (RS) signal determines which of these two registers is selected. 

|  |  |  |
| --- | --- | --- |
| **RS** | **R/W** | **Operation** |
| 0 | 0 | IR write, internal operation (Display Clear etc.) |
| 0 | 1 | Busy flag (DB7) and Address Counter (DB0 ~ DB6) read |
| 1 | 0 | DR Write, Internal Operation (DR ~ DD RAM or CG RAM) |
| 1 | 1 | DR Read, Internal Operation (DD RAM or CG RAM) |

If we want display any character we should send the ASCII value of that character.   
  
**Busy Flag:**   
When the busy flag is high or “1” the module is performing an internal operation and the next instruction will not be accepted. As shown in Table 1.4, the busy flag outputs to DB7 when RS=0 and a read operation is performed. The next instruction must not be written until ensuring that the busy flag is low or “0”.   
  
**Address Counter (AC):**   
The address counter (AC) assigns addresses to the DD RAM and the CG RAM. When the address of an instruction is written into the IR, the address information is sent from the IR to the AC. The selection of either DD RAM or CG RAM is also determined concurrently by the same instruction. After writing into or reading from the DD RAM or CG RAM the addresscounter (AC) is automatically incremented by 1 or decremented by 1 (determined by the I/D bit in the “Entry Mode Set” command.) AC contents are output to DB0 ~ DB7 when RS = 0 and a read operation is performed, as shown in Table.   
  
**DDRAM: Display Data RAM**  
Display data RAM (DDRAM) stores display data in 8-bit character codes. Its capacity is 80x8 bits, or 80 characters. Whatever you send to the DDRAM will display on the LCD. For LCDs like 1x16, only 16 characters are visible, so whatever you write after 16 characters in DDRAM is not visible to the user.   
  
Engineering Study Material   
**Figure:** DDRAM Address for 2 Lines LCD  
  
**CGROM: Character Generator ROM**  
whenever we send an ASCII value to DDRAM, how the character is displayed on LCD? The answer is CGROM.   
  
The character generator ROM generates 5 x 8 dot or 5 x 10 dot character patterns from 8-bit character codes (see below figure for more details). It can generate 208 5 x 8 dot character patterns and 32 5 x 10 dot character patterns.   
  
   
**CGRAM:** Character Generator RAM  
CGRAM is used to create custom characters in LCD. In the character generator RAM, the user can program character patterns. For 5 x 8 dots, eight character patterns can be written.   
  
Later in this tutorial we will discuss how to use CGRAM area to make custom character and also making animations.   
  
As we can see in above figure, the character code from 0x00 to 0x07 is occupied by the user defined characters. So we can create 8 characters only, if we want to create more than 8 then we have to delete previous one.   
  
If user wants to display the fourth custom character then the code to display 0x03 from CGROM i.e. when user sends 0x03 to the DDRAM then the fourth user created pattern will be displayed on the LCD.   
  
**INSTRUCTION SET:**

|  |  |  |  |
| --- | --- | --- | --- |
| **No.** | **Instruction** | **Hex** | **Dec** |
| 1 | Function Set: 8-bit, 1 Line, 5x7 Dots | 0x30 | 48 |
| 2 | Function Set: 8-bit, 2 Line, 5x7 Dots | 0x38 | 56 |
| 3 | Function Set: 4-bit, 1 Line, 5x7 Dots | 0x20 | 32 |
| 4 | Function Set: 4-bit, 2 Line, 5x7 Dots | 0x28 | 40 |
| 5 | Entry Mode | 0x06 | 6 |
| 6 | Display off Cursor off(clearing display without clearing DDRAM content) | 0x08 | 8 |
| 7 | Display on Cursor on | 0x0E | 14 |
| 8 | Display on Cursor off | 0x0C | 12 |
| 9 | Display on Cursor blinking | 0x0F | 15 |
| 10 | Shift entire display left | 0x18 | 24 |
| 11 | Shift entire display right | 0x1C | 30 |
| 12 | Move cursor left by one character | 0x10 | 16 |
| 13 | Move cursor right by one character | 0x14 | 20 |
| 14 | Clear Display (also clear DDRAM content) | 0x01 | 1 |
| 15 | Set DDRAM address or cursor position on display | 0x80+add | 128+add |
| 16 | Set CGRAM address or set pointer to CGRAM location | 0x40+add | 64+add |

**Sample Program:**   
CODESEGMENT  
ASSUME CS:CODE,DS:CODE,ES:CODE,SS:CODE  
;   
;   
START:ORG 0H; Use 1000H for MDA series  
MOVAX,CS  
MOVDS,AX; Making the DS (Data Segment) and CS (Code Segment) value same  
;   
MOVSS,AX; Making the SS (Stack Segment) also same with CS, DS  
MOVSP,STACK  
;   
CALL ALLCLR  
;   
CALL LN11  
MOV SI,OFFSET LINE1  
CALL STRING  
;   
CALL LN21  
MOV SI,OFFSET LINE2  
CALL STRING  
;Blinks the whole display   
BLINK:CALL DISPOFF  
CALL TIMER  
CALL DISPON  
CALL TIMER  
JMP BLINK  
;   
LINE1 DB'Hi SAKSHIStudents!’,00H,00H  
LINE2 DB'Make me friend.',00H,00H  
;   
; LCD instruction  
ALLCLR:MOV AH, 00000001B; Clears entire display  
JMP OUT1  
;   
DISPOFF:MOVAH, 00001000B; Display off, cursor off, not blink  
JMP OUT1   
;   
DISPON: MOVAH, 00001111B; Display on, cursor on, cursor blink  
JMP OUT1  
;   
LN11:MOV AH, 00000010B; Returns to home position  
JMP OUT1  
;   
LN21:MOVAH, 11000000B; Sets RAM address so that the cursor is positioned ;at the head of the 2nd line.   
JMP OUT1  
;   
;To write to instruction register  
OUT1:PUSH AX  
PUSH DX  
CALL BUSY  
MOV AL,AH  
MOV DX,IR\_WR  
OUT DX,AL  
POP DX  
POP AX  
RET  
; busy flag check, must be done before any write operation  
BUSY: PUSH DX  
PUSH AX  
MOV DX,ST\_RD  
BUSY1: IN AL,DX  
AND AL,10000000B  
JNZ BUSY1  
POP AX  
POP DX  
RET  
;   
;To send a single character  
CHAROUT: PUSH DX  
PUSH AX  
CALL BUSY  
MOV AL,AH  
MOV DX,DR\_WR  
OUT DX,AL  
POP AX  
POP DX  
RET  
;To out a string line from address CS:[SI]   
STRING:MOV AH,BYTE PTR CS:[SI]   
CMP AH,00H  
JE STRING1  
;   
CALL BUSY  
CALL CHAROUT  
INC SI  
JMP STRING  
STRING1:   
RET  
; Timer Makes delay  
TIMER: PUSH CX  
MOV CX,0FFFFH  
TIMER1:DEC CX  
JNZ TIMER1  
POP CX  
RET  
;   
CODE ENDS  
END START

The most common application of liquid crystal technology is in liquid crystal displays (LCDs). From the ubiquitous wrist watch and pocket calculator to an advanced VGA computer screen, this type of display has evolved into an important and versatile interface.

A liquid crystal display consists of an array of tiny segments (called pixels) that can be manipulated to present information. This basic idea is common to all displays, ranging from simple calculators to a full color LCD television.

Why are liquid crystal displays important? The first factor is size. As will be shown in the following sections, an LCD consists primarily of two glass plates with some liquid crystal material between them. There is no bulky picture tube. This makes LCDs practical for applications where size (as well as weight) are important.

In general, LCDs use much less power than their cathode-ray tube (CRT) counterparts. Many LCDs are reflective, meaning that they use only ambient light to illuminate the display. Even displays that do require an external light source (i.e. computer displays) consume much less power than CRT devices.

Liquid crystal displays do have drawbacks, and these are the subject of intense research. Problems with viewing angle, contrast ratio, and response time still need to be solved before the LCD replaces the cathode-ray tube. However with the rate of technological innovation, this day may not be too far into the future.

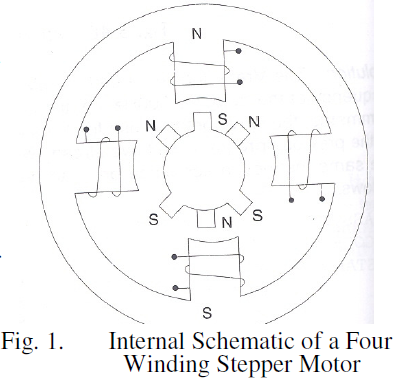
We will restrict this discussion to traditional LCDs since the major technological advances have been developed for this group of devices. Other LC applications, such as that employing polymer stabilization of LC structure, are discussed in the appropriate section covering those materials.

# STEPPER MOTOR INTERFACING

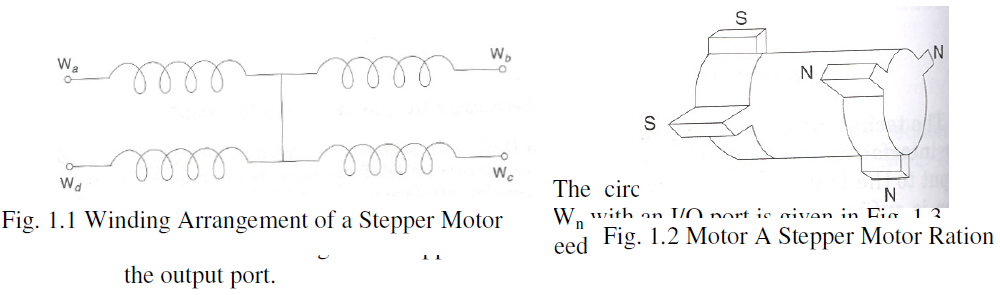
A stepper motor is stepped from one position to the next by changing the currents through the fields in the motor. The two common field connections are referred to as two phase or four phase. There are three main areas of applications for stepper motor. i. Instrumentation ii. Computer peripherals iii. Machine drives.They are used in floppy drives, dot-matrix printers, X-Y plotters, digital watches  
etc to rotate things in steps of small angles. The step size in typical stepper motor varies from 0.9 to30.

A stepper motor is a device used to obtain an accurate position control of rotating shafts. A stepper motor employs rotation of its shaft in terms of steps, rather than continuous rotation as in case of AC or DC motors. To rotate the shaft of the stepper motor, a sequence of pulses is needed to be applied to the windings ofthe stepper motor, in proper sequence. The number of pulses required for one complete rotation of the shaft of the stepper motor are equal to its number of internal teeth on its rotor. The stator teeth and the rotor teeth lock with each other to fix a position of the shaft. With a pulse applied to the winding input, the rotor rotates by one teeth position or an angle x.The angle x may be calculated as. x =360° /no. of rotor teeth

After the rotation of the shaft through angle x the rotor locks itself with the next tooth in the sequence on the internal surface of stator. The internal schematic of a typical stepper motor with four windings is shown in Fig. 1.

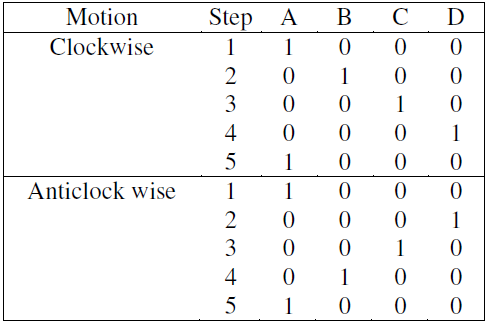


The stepper motors have been designed to work with digital circuits. Binary level pulses of 0-5V are required at its winding inputs to obtain the rotation of shafts.The sequence of the pulses can be decided, depending upon the required motion of the shaft. Figure 1.1 shows a typical winding arrangement of the stepper motor. Figure 1.2 shows conceptual positioning of the rotor teeth on the surface of rotor, for a six teeth rotor.

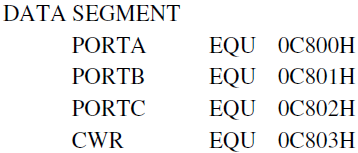


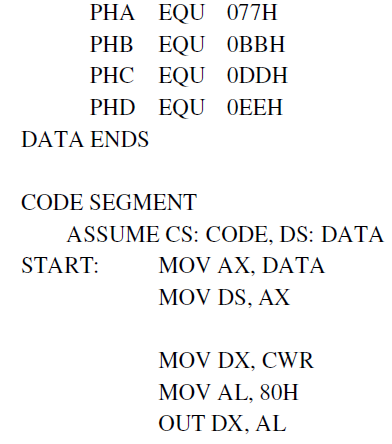
A typical stepper motor may have parameters like torque 3 kg-em, operating voltage 12V, current rating 1.2A and a step angle 1.80, i.e. 200 steps/revolution (number of rotor teeth).

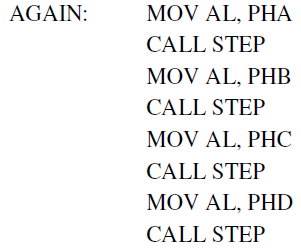
A simple scheme for rotating the shaft of a stepper motor is called as wave scheme. In this scheme, the windings Wa, Wb, We and Wd are applied with the required voltage pulses, in a cyclic fashion. By reversing the sequence of  
excitation, the direction of rotation of the stepper motor shaft may be reversed. Table 1 shows the excitation sequences for clockwise and anticlockwise rotations. Another popular scheme for rotation of a stepper motor shaft applies pulses to two successive windings at a time but these are shifted only by one position at a time.This scheme for rotation of stepper motor shaft is shown in Table 1. Table 1 excitation Sequences of a Stepper Motor Using Wave Switching Scheme

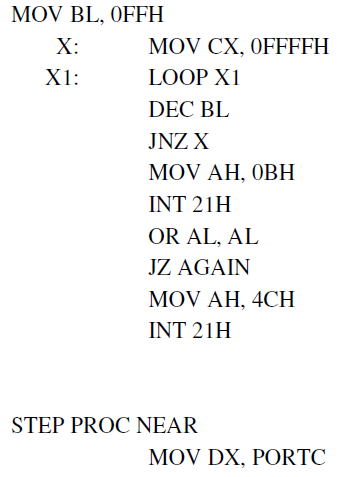


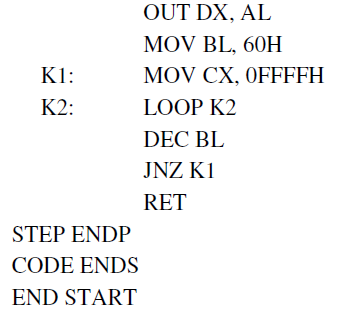
**PROGRAM FOR STEPPER MOTOR TO ROTATE CLOCKWISE/ ANTICLOCKWISE  
DIRECTION FOR N ROTATIONS.**

****

****

****

****

****

**UNIT-IV**

**Overview of Microcontroller 8051:** Introduction to 8051 Micro-controller, Architecture, Memory organization,Special function registers, Port Operation, Memory Interfacing, I/O Interfacing, Programming 8051 resources,interrupts, Programmer’s model of 8051, Operand types, Operand addressing, Data transfer instructions,Arithmetic instructions, Logic instructions, Control transfer instructions, Timer & Counter Programming,Interrupt Programming.

**[T3][No. of hrs. 11]**

**Introduction :**

A decade back the process and control operations were totally implemented by the Microprocessors only. But now a days the situation is totally changed and it is occupied by the new devices called Microcontroller. The development is so drastic that we can’t find any electronic gadget without the use of a microcontroller. This microcontroller changed the embedded system design so simple and advanced that the embedded market has become one of the most sought after for not only entrepreneurs but for design engineers also.

**What is a Microcontroller?**

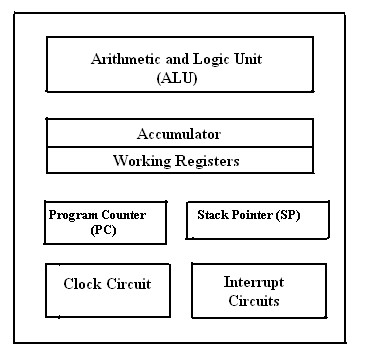
A single chip computer or A CPU with all the peripherals like RAM, ROM, I/O Ports, Timers , ADCs etc... on the same chip. For ex: Motorola’s 6811, Intel’s 8051, Zilog’s Z8 and PIC 16X etc…

**MICROPROCESSORS & MICROCONTROLLERS:**

**Microprocessor:**

A CPU built into a single VLSI chip is called a microprocessor. It is a general-purpose device and additional external circuitry are added to make it a microcomputer. The microprocessor contains arithmetic and logic unit (ALU), Instruction decoder and control unit, Instruction register, Program counter (PC), clock circuit (internal or external), reset circuit (internal or external) and registers. But the microprocessor has no on chip I/O Ports, Timers , Memory etc.

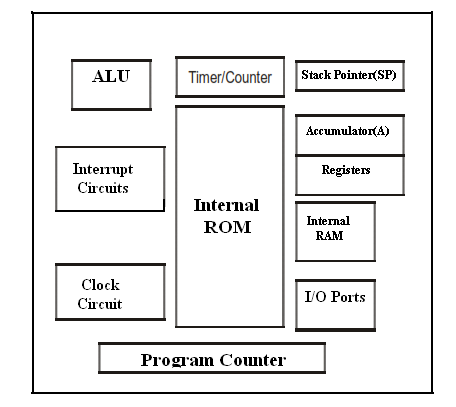
For example, Intel 8085 is an 8-bit microprocessor and Intel 8086/8088 a 16-bit microprocessor. The block diagram of the Microprocessor is shown in Fig.1



**Fig.1 Block diagram of a Microprocessor.**

**MICROCONTROLLER :**

A microcontroller is a highly integrated single chip, which consists of on chip CPU (Central Processing Unit), RAM (Random Access Memory), EPROM/PROM/ROM (Erasable Programmable Read Only Memory), I/O (input/output) – serial and parallel, timers, interrupt controller. For example, Intel 8051 is 8-bit microcontroller and Intel 8096 is 16-bit microcontroller. The block diagram of Microcontroller is shown in Fig.2.



**Fig.2.Block Diagram of a Microcontroller**

**Distinguish between Microprocessor and Microcontroller**

|  |  |  |
| --- | --- | --- |
| **S.No** | **Microprocessor** | **Microcontroller** |
| 1 | A microprocessor is a general purpose device which is called a CPU | A microcontroller is a dedicated chip which is also called single chip computer. |
| 2 | A microprocessor do not contain onchip I/OPorts, Timers, Memories etc.. | A microcontroller includes RAM, ROM, serial and parallel interface, timers, interrupt  circuitry (in addition to CPU) in a single chip. |
| **3** | Microprocessors are most commonly used as the CPU in microcomputer systems | Microcontrollers are used in small, minimum component designs performing control-oriented applications. |
| 4 | Microprocessor instructions are mainly nibble or byte addressable | Microcontroller instructions are both bit addressable as well as byte addressable. |
| 5 | Microprocessor instruction sets are mainly intended for catering to large volumes of data. | Microcontrollers have instruction sets catering to the control of inputs and outputs. |
| 6 | Microprocessor based system design is complex and expensive | Microcontroller based system design is rather simple and cost effective |
| **7** | The Instruction set of microprocessor is complex with large number of instructions. | The instruction set of a Microcontroller is very simple with less number of instructions. For, ex: PIC microcontrollers have only 35 instructions**.** |
| 8 | A microprocessor has zero status flag | A microcontroller has no zero flag. |

**EVOLUTION OF MICROCONTROLLERS :**

The first microcontroller TMS1000 was introduced by Texas Instrumentsin the year 1974. In the year 1976, Motorola designed a Microprocessor chip called 6801 which replaced its earlier chip 6800 with certain add-on chips to make a computer. This paved the way for the new revolution in the history of chip design and gave birth to a new entity called “**Microcontroller”.** Later **t**he Intel company produced its first Microcontroller 8048 with a CPU and 1K bytes of EPROM, 64 Bytes of RAM an 8-Bit Timer and 27 I/O pins in 1976. Then followed the most popular controller 8051 in the year 1980 with 4K bytes of ROM,128 Bytes of RAM , a serial port, two 16-bit Timers , and 32 I/O pins. The 8051 family has many additions and improvements over the years and remains a most acclaimed tool for today’s circuit designers. INTEL introduced a 16 bit microcontroller 8096 in the year 1982 . Later INTEL introduced 80c196 series of 16-bit Microcontrollers for mainly industrial applications. Microchip, another company has introduced an 8-bit Microcontroller PIC 16C64 in the year 1985.The 32-bit microcontrollers have been developed by IBM and Motorola. MPC 505 is a 32-bit RISC controller of Motorola. The 403 GA is a 32 -bit RISC embedded controller of IBM.

In recent times ARM company (Advanced RISC machines) has developed and introduced 32 bit controllers for high-end application devices like mobiles , Ipods etc...

**TYPES OF MICROCONTROLLERS :**

Microcontrollers can be classified on the basis of internal bus width, architecture, memory and instruction set as 4-bit,8-bit,16-bit and 32-bit micrcontrollers.

**4-bit Microcontrollers**: These 4-bit microcontrollers are small size, minimum pin count and low cost controllers which are widely used for low end applications like LED & LCD display drivers ,portable battery chargers etc.. Their power consumption is also low. The popular 4-bit controllers are Renasa M34501 which is a 20 pin DIP chip with 4kB of ROM,256 Bytes of RAM,2-Counters and 14 I/O Pins. Similarly ATAM862 series from ATMEL.

**8-bit Microcontrollers :** These are the most popular and widely used microcontrollers .About 55% of all [CPUs](http://en.wikipedia.org/wiki/Central_processing_unit) sold in the world are 8-bit microcontrollers only.The 8-bit microcontroller has 8-bitinternal bus and the ALU performs all the arithmetic and logical operations on a byte instruction. The well known 8-bit microcontroller is 8051 which was designed by Intel in the year 1980 for the use in embedded systems. Other 8-bit microcontrollers are Intel 8031/8052 and Motorola MC68HC11 and AVR Microcontrollers, Microchip’s PIC Microcontrollers 12C5XX ,16C5X and 16C505 etc...

**16-bit Microcontrollers :** When the microcontroller performs 16-bit arithmetic and logical operations at an instruction, the microcontroller is said to be a 16-bit microcontroller. The internal bus width of 16-bit microcontroller is of 16-bit. These microcontrollers are having increased memory size and speed of operation when compared to 8-bit microcontrollers.These are most suitable for programming in Highlevel languages like C or C++ .They find applications in disk drivers,modems,printers,scanners and servomotor control. Examples of 16-bit microcontrollers are Intel 8096 family and Motorola MC68HC12 and MC68332 families, The performance and computing capability of 16 bit microcontrollers are enhanced with greater precision as compared to the 8-bit microcontrollers.

**32-Bit Microcontrollers :**These microcontrollers used in highend applications like Automative control, Communication networks,Robotics,Cell phones ,GPRS & PDAs etc..For EX:PIC32,ARM 7,ARM9 ,SHARP LH79520 ,ATMEL 32 (AVR) ,Texas Instrument’s –. TMS320F2802x/2803x etc..are some of the popular 32-bit microcontrollers.

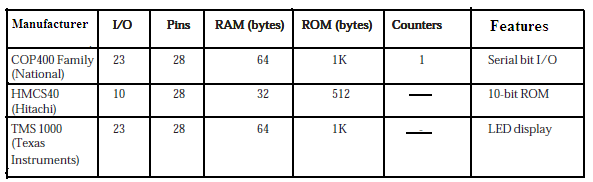
**COMMERCIAL MICROCONTROLLERS**

There are various manufacturers who are supplying various types of microcontrollers suitable for different applications depending on the power consumption and the available features..They are given below in tables . First the various members of INTEL 51 family are given in below table.

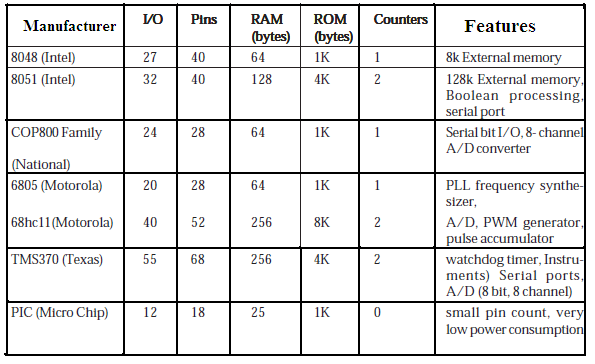
**INTEL MCS 51 Family**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Microcontroller** | **On chip RAM (Bytes)** | **On chip program memory** | **Timers/Counters** | **Interrupts** | **Serial ports** |
| 8031 | 128 | None | 2 | 5 | 1 |
| 8032 | 256 | None | 3 | 6 | 1 |
| 8051 | 128 | 4K ROM | 2 | 5 | 1 |
| 8052 | 256 | 8K ROM | 3 | 6 | 1 |
| 8751 | 128 | 4K EPROM | 2 | 5 | 1 |
| 8752 | 256 | 8K EPROM | 3 | 6 | 1 |

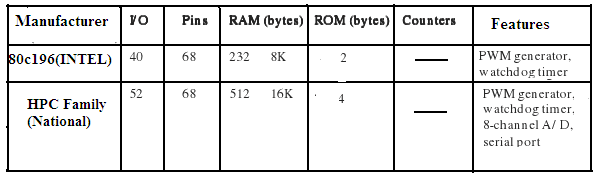
The following table gives the 4-bit microcontrollers from different manufacturers.



**4-Bit Microcontrollers**.



**8-Bit Microcontrollers.**

****

**16-Bit Microcontrollers**

The following table gives the list of PIC microcontrollers from Micro chip Inc

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Microcontroller | Pins | I/O Lines | On chip ADCs | EPROM  X 12 words | On chip RAM (Bytes) |
| 16C54 | 18 | 12 | None | 512 | 25 |
| 16C55 | 28 | 20 | None | 512 | 24 |
| 16C56 | 18 | 12 | None | 1k | 25 |
| 16C57 | 28 | 20 | None | 2k | 72 |
| 17C42A | 40 | 33 | None | 2k | 232 |
| 17C43 | 40 | 33 | None | 4k | 454 |
| 17C44 | 40 | 33 | None | 8k | 454 |
| 17C71 | 18 | 13 | 8bit ADCs | 1kx14 | 36 |
| 17C752 | 40 | 33 | 10Bit ADC | 8kx16 | 678 |

**MICROCONTROLLER DEVELOPMENT TOOLS:**

To develop an assembly language program we need certain program development tools. An assembly language program consists of Mnemonics which are nothing but short abbreviated English instructions given to the controller.The various development tools required for Microcontroller programming are explained below.

**1. Editor :** An Editor is a program which allows us to create a file containing the assembly language statements for the program. Examples of some editors are PC write Wordstar. As we type the program the editor stores the ACSII codes for the letters and numbers in successive RAM locations. If any typing mistake is done editor will alert us to correct it. If we leave out a program statement an editor will let you move everything down and insert a line. After typing all the program we have to save the program . This we call it as source file. The next step is to process the source file with an assembler.

Ex: Sample. asm

**2.Assembler :** An Assembler is used to translate the assembly language mnemonics into machine language( i.e binary codes). When you run the assembler it reads the source file of your program from where you have saved it. The assembler generates a filee with the extension **.hex**. This file consists of hexadecimal values encoding a sequence of data and their starting offset or absolute address.

**3.Compiler :** A compiler is a program which converts the high level language program like “C” into binary or machine code. Using high level languages it is easy to manage complex data structures which are often required for data manipulation. Because of its ease , flexibility and debug options now a days the compilers have become very popular in the market. Compilers like Keil ,Ride and IAR workbench are very popular.

**3. Debugger/Simulator :** A debugger is a program which allows execute the program, and troubleshoot or debug it. The debugger allows to look into the contents of registers and memory locations after the program runs. We can also change the contents of registers and memory locations and rerun the program. Some debuggers allows to stop the program after each instruction so that you can check or alter memory and register contents. This is called single step debug. A debugger also allows to set a breakpoint at any point in the program. If we insert a break point , the debugger will run the program up to the instruction where the breakpoint is put and then stop the execution.

A simulator is a software program which virtually executes the instructions similar to a microcontroller and shows the results. This will help in evaluating the results without committing any errors. By doing so we can detect the possible logic errors

**INTEL 8051 MICRCONTROLLER :**

The 8051 microcontroller is a very popular 8-bit microcontroller introduced by Intel in the year 1981 and it has become almost the academic standard now a days. The 8051 is based on an 8-bit CISC core with Harvard architecture. Its 8-bit architecture is optimized for control applications with extensive Boolean processing. It is available as a 40-pin DIP chip and works at +5 Volts DC. The salient features of 8051 controller are given below.

**SALIENT FEATURES :** The salient features of 8051 Microcontroller are

i. 4 KB on chip program memory (ROM or EPROM)).

ii. 128 bytes on chip data memory(RAM).

iii. 8-bit data bus

iv. 16-bit address bus

v. 32 general purpose registers each of 8 bits

vi. Two -16 bit timers T0 and T1

vii. Five Interrupts (3 internal and 2 external).

ix. Four Parallel ports each of 8-bits (PORT0, PORT1,PORT2,PORT3) with a total of 32 I/O

lines.

x. One 16-bit program counter and One 16-bit DPTR ( data pointer)

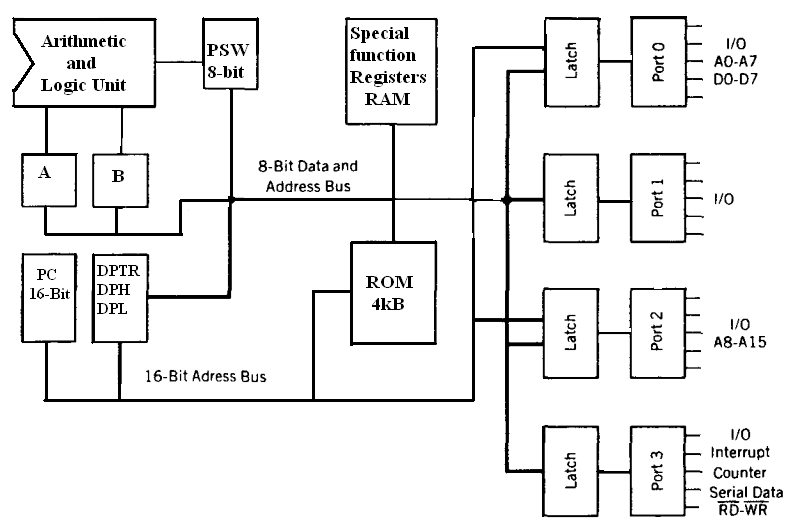
xi. One 8-bit stack pointer

xii. One Microsecond instruction cycle with 12 MHz Crystal.

xiii. One full duplex serial communication port.

**ARCHITECTURE & BLOCK DIAGRAM OF 8051 MICROCONTROLLER:**

The architecture of the 8051 microcontroller can be understood from the block diagram. It has Harward architecture with RISC (Reduced Instruction Set Computer) concept. The block diagram of 8051 microcontroller is shown in Fig 3. below1.It consists of an 8-bit ALU, one 8-bit PSW(Program Status Register), A and B registers , one 16-bit Program counter , one 16-bit Data pointer register(DPTR),128 bytes of RAM and 4kB of ROM and four parallel I/O ports each of 8-bit width.

****

**Fig.3. Block Diagram of 8051 Microcontroller**

8051 has 8-bit ALU which can perform all the 8-bit arithmetic and logical operations in one machine cycle. The ALU is associated with two registers A & B

**A and B Registers** : The A and B registers are special function registers which hold the results of many arithmetic and logical operations of 8051.The A register is also called the **Accumulator** and as it’s name suggests, is used as a general register to accumulate the results of a large number of instructions. By default it is used for all mathematical operations and also data transfer operations between CPU and any external memory.

The B register is mainly used for multiplication and division operations along with A register.

MUL AB : DIV AB.

It has no other function other than as a location where data may be stored.

**The R registers**: The "R" registers are a set of eight registers that are named R0, R1, etc. up to

and including R7. These registers are used as auxillary registers in many operations. The "R" registers are also used to temporarily store values.

**Program Counter(PC) :** 8051 has a 16-bit program counter .The program counter always points to the address of the next instruction to be executed. After execution of one instruction the program counter is incremented to point to the address of the next instruction to be executed.It is the contents of the PC that are placed on the address bus to find and fetch the desired instruction.Since the PC is 16-bit width ,8051 can access program addresses from 0000H to FFFFH ,a total of 6kB of code.

**Stack Pointer Register (SP) :** It is an 8-bit register which stores the address of the stack top. i.e the Stack Pointer is used to indicate where the next value to be removed from the stack should be taken from. When a value is pushed onto the stack, the 8051 first increments the value of SP and then stores the value at the resulting memory location. Similarly when a value is popped off the stack, the 8051 returns the value from the memory location indicated by SP, and then decrements the value of SP. Since the SP is only 8-bit wide it is incremented or decremented by two . SP is modified directly by the 8051 by six instructions: PUSH, POP, ACALL, LCALL, RET, and RETI. It is also used intrinsically whenever an interrupt is triggered.

**STACK in 8051 Microcontroller :** The stack is apart of RAM used by the CPUto store information temporarily. This information may be either data or an address .The CPU needs this storage area as there are only limited number of registers. The register used to access the stack is called the Stack pointer which is an 8-bit register..So,it can take values of 00 to FF H.When the 8051 is powered up ,the SP register contains the value 07.i.e the RAM location value 08 is the first location being used for the stack by the 8051 controller

There are two important instructions to handle this stack.One is the PUSH and the Other is the POP. The loading of data from CPU registers to the stack is done by PUSH and the loading of the contents of the stack back into aCPU register is done by POP.

EX : MOV R6 ,#35 H

MOV R1 ,#21 H

PUSH 6

PUSH 1

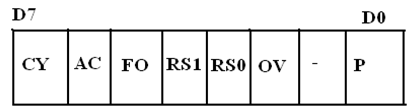
In the above instructions the contents of the Registers R6 and R1 are moved to stack and they occupy the 08 and 09 locations of the stack.Now the contents of the SP are incremented by two and it is 0A

Similarly POP 3 instruction pops the contents of stack into R3 register.Now the contents of the SP is decremented by 1

In 8051 the RAM locations 08 to 1F (24 bytes) can be used for the Stack.In any program if we need more than 24 bytes of stack ,we can change the SP point to RAM locations 30-7F H.this can be done with the instruction MOV SP,# XX.

**Data Pointer Register(DPTR) :** It is a 16-bit register which is the only user-accessible. DPTR, as the name suggests, is used to point to data. It is used by a number of commands which allow the 8051 to access external memory. When the 8051 accesses external memory it will access external memory at the address indicated by DPTR. This DPTR can also be used as two 8-registers DPH and DPL.

**Program Status Register (PSW) :** The 8051 has a 8-bit PSW registerwhich is alsoknown as Flag register.In the 8-bit register only 6-bits are used by 8051.The two unused bits are user definable bits.In the 6-bits four of them are conditional flags .They are Carry –CY,Auxiliary Carry-AC, Parity-P,and Overflow-OV .These flag bits indicate some conditions that resulted after an instruction was executed.



The bits PSW3 and PSW4 are denoted as RS0 and RS1 and these bits are used th select the bank registers of the RAM location. The meaning of various bits of PSW register is shown below.

CY PSW.7 Carry Flag

AC PSW.6 Auxiliary Carry Flag

FO PSW.5 Flag 0 available for general purpose .

RS1 PSW.4 Register Bank select bit 1

RS0 PSW.3 Register bank select bit 0

OV PSW.2 Overflow flag

--- PSW.1 User difinable flag

P PSW.0 Parity flag .set/cleared by hardware.

The selection of the register Banks and their addresses are given below.

|  |  |  |  |
| --- | --- | --- | --- |
| **RS1** | **RS0** | **Register Bank** | **Address** |
| 0 | 0 | 0 | 00H-07H |
| 0 | 1 | 1 | 08H-0FH |
| 1 | 0 | 2 | 10H-17H |
| 1 | 1 | 3 | 18H-1FH |

**Memory organization :** The 8051 microcontroller has 128 bytes of Internal RAM and 4kB of on chip ROM .The RAM is also known as Data memory and the ROM is known as program memory. The program memory is also known as Code memory .This Code memory holds the actual 8051 program that is to be executed. In 8051 this memory is limited to 64K .Code memory may be found on-chip, as ROM or EPROM. It may also be stored completely off-chip in an external ROM or, more commonly, an external EPROM. The 8051 has only 128 bytes of Internal RAM but it supports 64kB of external RAM. As the name suggests, external RAM is any random access memory which is off-chip. Since the memory is off-chip it is not as flexible interms of accessing, and is also slower. For example, to increment an Internal RAM location by 1,it requires only 1 instruction and 1 instruction cycle but to increment a 1-byte value stored in External RAM requires 4 instructions and 7 instruction cycles. So, here the external memory is 7 times slower.

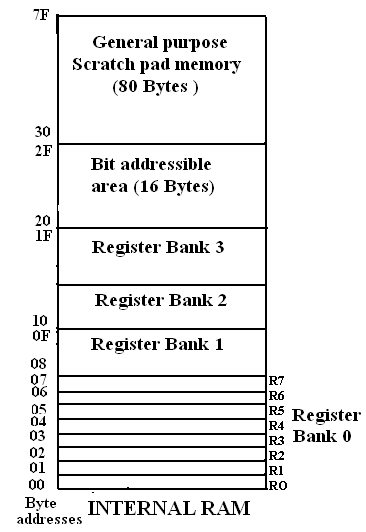
**Internal RAM OF 8051 :**

This Internal RAM is found on-chip on the 8051 .So it is the fastest RAM available, and it is also the most flexible in terms of reading, writing, and modifying it’s contents. Internal RAM is volatile, so when the 8051 is reset this memory is cleared. The 128 bytes of internal RAM is organized as below.

(i) Four register banks (Bank0,Bank1, Bank2 and Bank3) each of 8-bits (total 32 bytes). The default bank register is Bank0. The remaining Banks are selected with the help of RS0 and RS1 bits of PSW Register.

(ii) 16 bytes of bit addressable area and

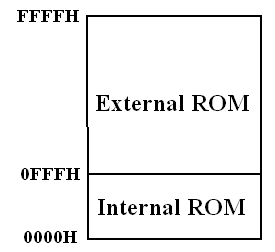
(iii) 80 bytes of general purpose area (Scratch pad memory) as shown in the diagram below. This area is also utilized by the microcontroller as a storage area for the operating stack.



The 32 bytes of RAM from address 00 H to 1FH are used as working registers organized as four banks of eight registers each.The registers are named as R0-R7 .Each register can be addressed by its name or by its RAM address.

For EX : MOV A, R7 or MOV R7,#05H

**Internal ROM (On –chip ROM):** The 8051 microcontroller has 4kB of on chip ROM but it can be extended up to 64kB.This ROM is also called program memory or code memory. The CODE segment is accessed using the program counter (PC) for opcode fetches and by DPTR for data. The external ROM is accessed when the EA(active low) pin is connected to ground or the contents of program counter exceeds 0FFFH.When the Internal ROM address is exceeded the 8051 automatically fetches the code bytes from the external program memory.



**SPECIAL FUNCTION REGISTERS (SFRs) :** In 8051 microcontroller there certainregisters which uses the RAM addresses from 80h to FFh and they are meant for certain specific operations .These registers are called Special function registers (SFRs).Some of these registers are bit addressable also.

The list of SFRs and their functional names are given below. In these SFRs some of them are related to I/O ports (P0,P1,P2 and P3) and some of them are meant for control operations (TCON,SCON, PCON..) and remaining are the auxillary SFRs, in the sense that they don't directly configure the 8051.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **S.No** | **Symbol** | | **Name of SFR** | **Address (Hex)** |
| 1 | ACC\* | | Accumulator | **0E0** |
| 2 | B\* | | B-Register | **0F0** |
| 3 | PSW\* | | Program Status word register | **0DO** |
| 4 | SP | | Stack Pointer Register | **81** |
| 5 | DPTR | DPL | Data pointer low byte | **82** |
| DPH | Data pointer high byte | **83** |
| 6 | P0\* | | Port 0 | **80** |
|  | P1\* | | Port 1 | **90** |
| 8 | P2\* | | Port 2 | **0A** |
| 9 | P3\* | | Port 3 | **0B** |
| 10 | IP\* | | Interrupt Priority control | **0B8** |
| 11 | IE\* | | Interrupt Enable control | **0A8** |
| 12 | TMOD | | Tmier mode register | **89** |
| 13 | TCON\* | | Timer control register | **88** |
| 14 | TH0 | | Timer 0 Higher byte | **8C** |
| 15 | TL0 | | Timer 0 Lower byte | **8A** |
| 16 | TH1 | | Timer 1Higher byte | **8D** |
| 17 | TL1 | | Timer 1 lower byte | **8B** |
| 18 | SCON\* | | Serial control register | **98** |
| 19 | SBUF | | Serial buffer register | **99** |
| 20 | PCON | | Power control register | **87** |

**The** \* **indicates the bit addressable SFRs**

**Table:SFRs of 8051 Microcontroller**

**PARALLEL I /O PORTS :**

The 8051 microcontroller has four parallel I/O ports , each of 8-bits .So, it provides the user 32 I/O lines for connecting the microcontroller to the peripherals. The four ports are P0 (Port 0), P1(Port1) ,P2(Port 2) and P3 (Port3). Upon reset all the ports are output ports. In order to make them input, all the ports must be set i.e a high bit must be sent to all the port pins. This is normally done by the instruction “SETB”.

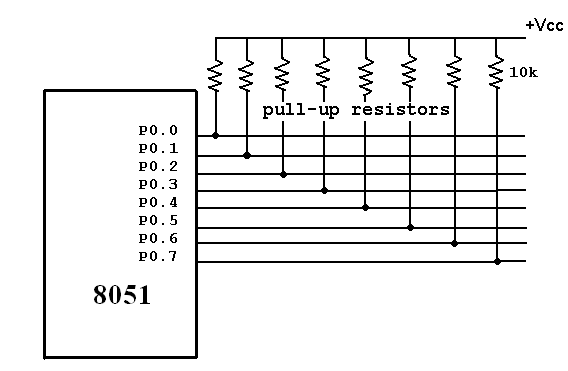
Ex: MOV A,#0FFH ; A = FF

MOV P0,A ; make P0 an input port

**PORT 0:**

Port 0 is an 8-bit I/O port with dual purpose. If external memory is used, these port pins are used for the lower address byte address/data (AD0-AD7), otherwise all bits of the port are either input or output.. Unlike other ports, Port 0 is not provided with pull-up resistors internally ,so for PORT0 pull-up resistors of nearly 10k are to be connected externally as shown in the fig.2.

**Dual role of port 0**: Port 0 can also be used as address/data bus(AD0-AD7), allowing it to be used for both address and data. When connecting the 8051 to an external memory, port 0 provides both address and data. The 8051 multiplexes address and data through port 0 to save the pins. ALE indicates whether P0 has address or data. When ALE = 0, it provides data D0-D7, and when ALE =1 it provides address and data with the help of a 74LS373 latch.



**Port 1:** Port 1 occupies a total of 8 pins (pins 1 through 8). It has no dual application and acts only as input or output port. In contrast to port 0, this port does not need any pull-up resistors since pull-up resistors connected internally. Upon reset, Port 1 is configured as an output port. To configure it as an input port , port bits must be set i.e a high bit must be sent to all the port pins. This is normally done by the instruction “SETB”. For Ex :

MOV A, #0FFH ; A=FF HEX

MOV P1,A ; make P1 an input port by writing 1’s to all of its pins

**Port 2 :** Port 2 is also an eight bit parallel port. (pins 21- 28). It can be used as input or output port. As this port is provided with internal pull-up resistors it does not need any external pull-up resistors. Upon reset, Port 2 is configured as an output port. If the port is to be used as input port, all the port bits must be made high by sending FF to the port. For ex,

MOV A, #0FFH ; A=FF hex

MOV P2, A ; make P2 an input port by writing all 1’s to it

**Dual role of port 2** : Port2 lines are also associated with the higher order address lines A8-A15. In systems based on the 8751, 8951, and DS5000, Port2 is used as simple I/O port.. But, in 8031-based systems, port 2 is used along with P0 to provide the 16-bit address for the external memory. Since an 8031 is capable of accessing 64K bytes of external memory, it needs a path for the 16 bits of the address. While P0 provides the lower 8 bits via A0-A7, it is the job of P2 to provide bits A8-A15 of the address. In other words, when 8031 is connected to external memory, Port 2 is used for the upper 8 bits of the 16 bit address, and it cannot be used for I/O operations.

**PORT 3** : Port3 is also an 8-bit parallel port with dual function.( pins 10 to 17). The port pins can be used for I/O operations as well as for control operations. The details of these additional operations are given below in the table. Port 3 also do not need any external pull-up resistors as they are provided internally similar to the case of Port2 & Port 1. Upon reset port 3 is configured as an output port . If the port is to be used as input port, all the port bits must be made high by sending FF to the port. For ex,

MOV A, #0FFH ; A= FF hex

MOV P3, A ; make P3 an input port by writing all 1’s to it

**Alternate Functions of Port 3 :** P3.0 and P3.1 are used for the RxD (Receive Data) and TxD (Transmit Data) serial communications signals. Bits P3.2 and P3.3 are meant for external interrupts. Bits P3.4 and P3.5 are used for Timers 0 and 1 and P3.6 and P3.7 are used to provide the write and read signals of external memories connected in 8031 based systems

|  |  |  |  |
| --- | --- | --- | --- |
| **S.No** | **Port 3 bit** | **Pin No** | **Function** |
| 1 | P3.0 | 10 | RxD |
| 2 | P3.1 | 11 | TxD |
| 3 | P3.2 | 12 |  |
| 4 | P3.3 | 13 |  |
| 5 | P3.4 | 14 | T0 |
| 6 | P3.5 | 15 | T1 |
| 7 | P3.6 | 16 |  |
| 8 | P3.7 | 17 |  |

**Table: PORT 3 alternate functions**

**Interrupt Structure**: An interrupt is an external or internal event that disturbs the microcontroller to inform it that a device needs its service. The program which is associated with the interrupt is called the **interrupt service routine** (ISR) or **interrupt handler**. Upon receiving the interrupt signal the Microcontroller , finish current instruction and saves the PC on stack. Jumps to a fixed location in memory depending on type of interrupt Starts to execute the interrupt service routine until RETI (return from interrupt)Upon executing the RETI the microcontroller returns to the place where it was interrupted. Get pop PC from stack

The 8051 microcontroller has **FIVE** interrupts in addition to Reset. They are

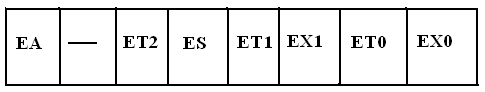
* Timer 0 overflow Interrupt
* Timer 1 overflow Interrupt
* External Interrupt 0(INT0)
* External Interrupt 1(INT1)
* Serial Port events (buffer full, buffer empty, etc) Interrupt

Each interrupt has a specific place in code memory where program execution (interrupt service routine) begins.

* External Interrupt 0: 0003 H
* Timer 0 overflow: 000B H
* External Interrupt 1: 0013 H
* Timer 1 overflow: 001B H
* Serial Interrupt : 0023 H

Upon reset all Interrupts are disabled & do not respond to the Microcontroller. These interrupts must be enabled by software in order for the Microcontroller to respond to them. This is done by an 8-bit register called Interrupt Enable Register (IE).

**Interrupt Enable Register :**

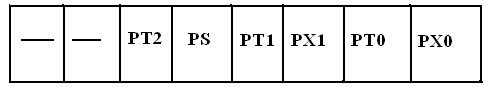


* EA : Global enable/disable. To enable the interrupts this bit must be set High.
* --- : Undefined-reserved for future use.
* ET2 : Enable /disable Timer 2 overflow interrupt.
* ES : Enable/disable Serial port interrupt.
* ET1 : Enable /disable Timer 1 overflow interrupt.
* EX1 : Enable/disable External interrupt1.
* ET0 : Enable /disable Timer 0 overflow interrupt.
* EX0 : Enable/disable External interrupt0

Upon reset the interrupts have the following priority.(Top to down). The interrupt with the highest PRIORITY gets serviced first.

* 1. External interrupt 0 (INT0)
  2. Timer interrupt0 (TF0)
  3. External interrupt 1 (INT1)
  4. Timer interrupt1 (TF1)
  5. Serial communication (RI+TI)

Priority can also be set to “high” or “low” by 8-bit IP register.- Interrupt priority register



IP.7: reserved

IP.6: reserved

IP.5: Timer 2 interrupt priority bit (8052 only)

IP.4: Serial port interrupt priority bit

IP.3: Timer 1 interrupt priority bit

IP.2: External interrupt 1 priority bit

IP.1: Timser 0 interrupt priority bit

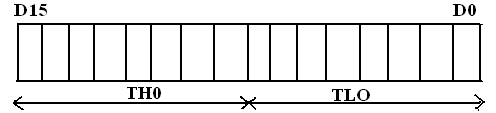
IP.0: External interrupt 0 priority bit

**TIMERS in 8051 Microcontrollers :** The 8051 microcontroller has two 16-bit timers Timer 0 (T0) and Timer 1(T1) which can be used either to generate accurate time delays or as event counters. These timers are accessed as two 8-bit registers TLO, THO & TL1 ,TH1 because the 8051 microcontroller has 8-bit architecture.

**TIMER 0 :** The Timer 0 is a 16-bit register and can be treated as two 8-bit registers (TL0 & TH0) and these registers can be accessed similar to any other registers like A,B or R1,R2,R3 etc…

Ex : The instruction Mov TL0,#07 moves the value 07 into lower byte of Timer0.

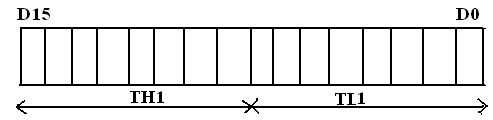
Similarly Mov R5,TH0 saves the contents of TH0 in the R5 register.



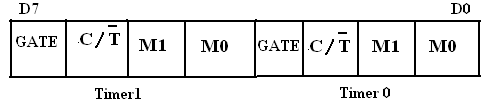
**TIMER 1 :** The Timer 1 is also a 16-bit register and can be treated as two 8-bit registers (TL1 & TH1) and these registers can be accessed similar to any other registers like A,B or R1,R2,R3 etc…

Ex : The instruction MOV TL1,#05 moves the value 05 into lower byte of Timer1.

Similarly MOV R0,TH1 saves the contents of TH1 in the R0 register



**TMOD Register :** The various operating modes of both the timers T0 and T1 are set by an 8-bit register called TMOD register**.** In this TMOD register the lower 4-bits are meant for Timer 0 and the higher 4-bits are meant for Timer1.



**GATE**: This bit is used to start or stop the timers by hardware .When GATE= 1 ,the timers can be started / stopped by the external sources. When GATE= 0, the timers can be started or stopped by software instructions like SETB TR0 or SETB TR1

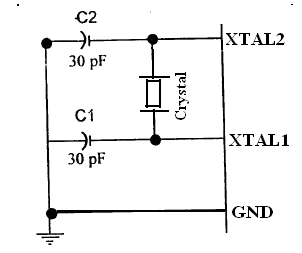
**C/T (clock/Timer) :** This bit decides whether the timer is used as delay generator or event counter. When **C/T = 0 ,**the Timer is used as delay generator and if C/T=1 the timer is used as an event counter. The clock source for the time delay is the crystal frequency of 8051.

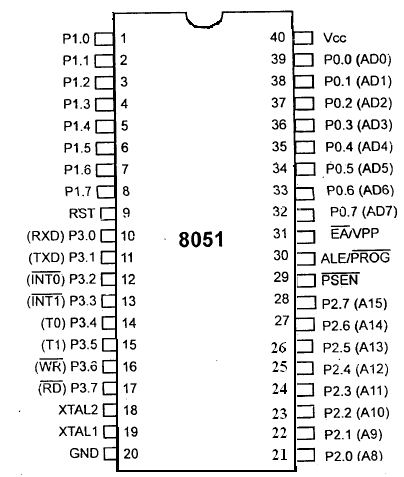
**M1,M0 (Mode) :** These two bits are thetimer mode bits. The timers of the 8051 can be configured in three modes.Mode0, Mode1 and Mode2.The selection and operation of the modes is shown below.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **S.No** | **M0** | **M1** | **Mode** | **Operation** |
| 1 | 0 | 0 | 0 | 13-bit Timer mode  8-bit Timer/counter. THx with TLx as 5-bit prescalar |
| 2 | 0 | 1 | 1 | 16-bit Timer mode.16-bit timer /counter without pre-scalar |
| 3 | 1 | 0 | 2 | 8-bit auto reload. THx contains a value that is to be loaded into TLx each time it overflows |
| 4 | 1 | 1 | 3 | Split timer mode |

**PIN Diagram of 8051 Microcontroller :** The 8051 microcontroller is available as a 40 pin DIP chip and it works at +5 volts DC. Among the 40 pins , a total of 32 pins are allotted for the four parallel ports P0,P1,P2 and P3 i.e each port occupies 8-pins .The remaining pins are VCC, GND, XTAL1, XTAL2, RST, EA ,PSEN.

**XTAL1,XTAL2**: These two pins are connected to Quartz crystal oscillator which runs the on-chip oscillator. The quartz crystal oscillator is connected to the two pins along with a capacitor of 30pF as shown in the circuit. If we use a source other than the crystal oscillator, it will be connected to XTAL1 and XTAL2 is left unconnected.





**RST**: The RESET pin is an input pin and it is an active high pin. When a high pulse is applied to this pin the microcontroller will reset and terminate all activities. Upon reset all the registers except PC will reset to 0000 Value and PC register will reset to 0007 value.

** (External Access):** This pin isan active low pin. This pin is connected to ground when microcontroller is accessing the program code stored in the external memory and connected to Vcc when it is accessing the program code in the on chip memory. This pin should not be left unconnected.

**(Program Store Enable) :** This is an output pin which is active low. When the microcontroller is accessing the program code stored in the external ROM ,this pin is connected to the OE (Output Enable) pin of the ROM.

**ALE (Address latch enable):** This is an output pin, which is active high**.** When connected to external memory , port 0 provides both address and data i.e address and data are multiplexed through port 0 .This ALE pin will demultiplex the address and data bus .When the pin is High , the AD bus will act as address bus otherwise the AD bus will act as Data bus.

**P0.0- P0.7(AD0-AD7) :** The port 0 pins multiplexed with Address/data pins .If the microcontroller is accessing external memory these pins will act as address/data pins otherwise they are used for Port 0 pins.

**P2.0- P2.7(A8-A15) :** The port2 pins are multiplexed with the higher order address pins **.**When the microcontroller is accessing external memory these pins provide the higher order address byte otherwise they act as Port 2 pins.

**P1.0- P1.7 :**These 8-pins are dedicated for Port1 to perform input or output port operations.

**P3.0- P3.7 :**These 8-pins are meant for Port3 operations and also for some control operations like Read,Write,Timer0,Timer1 ,INT0,INT1 ,RxD and TxD

**ADDRESSING MODES OF 8051 :**

The way in which the data operands are accessed by different instructions is known as the addressing modes. There are various methods of denoting the data operands in the instruction. The 8051 microcontroller supports mainly 5 addressing modes. They are

1.Immediate addressing mode

2.Direct Addressing mode

3.Register addressing mode

4. Register Indirect addressing mode

5.Indexed addressing mode

**Immediate addressing mode :** The addressing mode in which the data operand is a constant and it is a part of the instruction itself is known as Immediate addressing mode. Normally the data must be preceded by a # sign. This addressing mode can be used to transfer the data into any of the registers including DPTR.

Ex: MOV A , # 27 H : The data (constant) 27 is moved to the accumulator register

ADD R1 ,#45 H : Add the constant 45 to the contents of the accumulator

MOV DPTR ,# 8245H :Move the data 8245 into the data pointer register.

MOV P1,#21 H

**Direct addressing mode**: The addressing mode in which the data operand is in the RAM location (00 -7FH) and the address of the data operand is given in the instruction is known as Direct addressing mode. The direct addressing mode uses the lower 128 bytes of Internal RAM and the SFRs

MOV R1, 42H : Move the contents of RAM location 42 into R1 register

MOV 49H,A : Move the contents of the accumulator into the RAM location 49.

ADD A, 56H : Add the contents of the RAM location 56 to the accumulator

**Register addressing mode** :The addressing mode in which the data operand to be manipulated lies in one of the registers is known as register addressing mode.

MOV A,R0 : Move the contents of the register R0 to the accumulator

ADD A,R6 :Add the contents of R6 register to the accumulator

MOV P1, R2 : Move the contents of the R2 register into port 1

MOV R5, R2 : This is invalid .The data transfer between the registers is not allowed.

**Register Indirect addressing mode :**The addressing mode in which a register is used as a pointer to the data memory block is known as Register indirect addressing mode.

MOV A,@ R0 :Move the contents of RAM location whose address is in R0 into **A** (accumulator)

MOV @ R1 , B : Move the contents of B into RAM location whose address is held by R1

When R0 and R1 are used as pointers, they must be preceded by @ sign

**One of the advantages of register indirect addressing mode is that it makes accessing the data more dynamic than static as in the case of direct addressing mode.**

**Indexed addressing mode :** This addressing mode is usedin accessing the data elements of lookup table entries located in program ROM space of 8051.

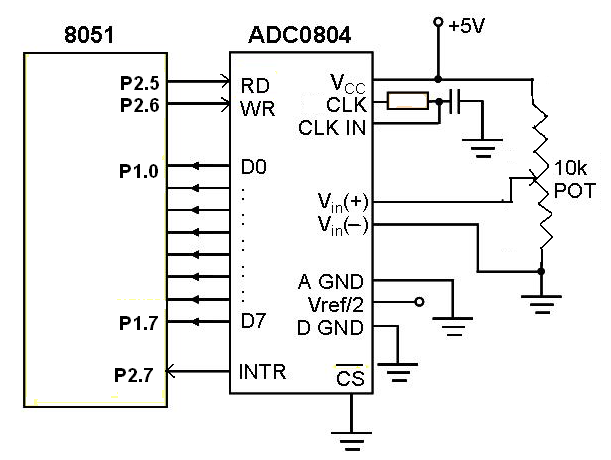
Ex : MOVC A,@ A+DPTR

The 16-bit register DPTR and register A are used to form the address of the data element stored in on-chip ROM. Here C denotes code .In this instruction the contents of A are added to the 16-bit DPTR register to form the 16-bit address of the data operand.

**Interfacing of ADC 0804 to 8051 Microcontroller :**

ADC 0804 is a single channel analog to digital converter i.e., it can take only one analog signal. ADC 0804 has 8 bit resolution. The higher resolution ADC gives smaller step size. Step size is smallest change that can be measured by an ADC. For an ADC with resolution of 8 bits, the step size is 19.53mV (5V/255). The time taken by the ADC to convert analog data into digital form depends on the frequency of clock source. The conversion time of ADC 0804 is around 110us. To use the internal clock a capacitor and resistor are used as shown in the circuit. The input to the ADC is given from a regulated power supply and a 10K potentiometer

The 8051 Microcontroller is used to provide the control signals to the ADC. CS(chip select) pin of ADC is directly connected to ground. The pin P1.1, P1.0 and P1.2 are connected to the pin WR, RD and INTR of the ADC respectively. When the input voltage from the preset is varied the output of ADC varies also varies.



From the circuit it is clear that the ADC interfaced directly to the microcontroller. The Port1 is used as an input port which receives the digital data from the ADC.Port pins P2.5 and P2.6 are used for SOC and EOC operation.When the conversion is over the ADC will send an interrupt signal to the microcontroller through the pin P2.7 .Now the Microcontroller receives digital data through the Port1.This data after conversion to decimal data is displayed on the LCD module .

The assembly language program for ADC is given below .

MOV P1 , 0FF H ; Make the port1 high and configure port1 as Input port

**BACK**: CLR P2.6 ; Generation of SOC pulse

SETB P2.5 ;

**LOOP** JB P2.7 , LOOP ; Wait for conversion, Is conversion over?

CLR P2.5 ; Enable Read the digital data

MOV A ,P1 ; Read digital data through Port1

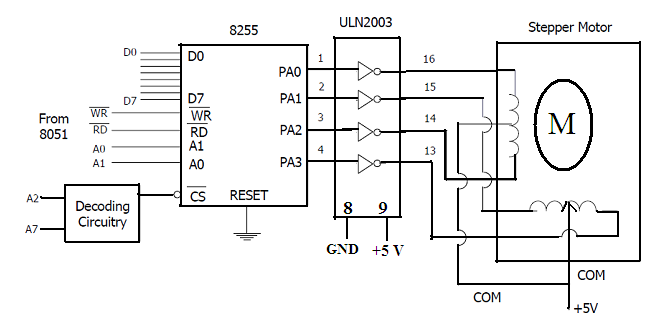
SETB P2.5 ; Disable read after read operation

CALL DISPLAY ; Display the data on LCD module

SJMP BACK ; Continue the conversion process

**Stepper motor Interfacing**

  A stepper motor is a device that translates electrical pulses into mechanical movement. The stepper motor rotates in steps in response to the applied signals. It is used in applications such as disk drives, dot matrix printers, plotters and robotics.It is mainly used for position control. Stepper motors have a permanent magnet called rotor (also called the shaft) surrounded by a stator . There are also steppers called variable reluctance stepper motors that do not have a PM rotor. The most common stepper motors have four stator windings that are paired with a center-tapped. 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.



**ASSEMBLY LANGUAGE PROGRAM**

Main mov stepper, #0CH ; move the code to phase1 into the port

acall delay

mov stepper, #06H ; phase II code

acall delay

mov stepper, #03H ;Phase III code

acall delay ;Call delay subroutine program

mov stepper, #09H ;Phase IV code

acall delay

sjmp Main

CALL DELAY PROGRAM :

mov r7,#4

wait2:

mov r6,#0FFH

wait1:

mov r5,#0FFH

wait:

djnz r5,wait

djnz r6,wait1

djnz r7,wait2

ret

end

**8051-SERIAL COMMUNICATION :**

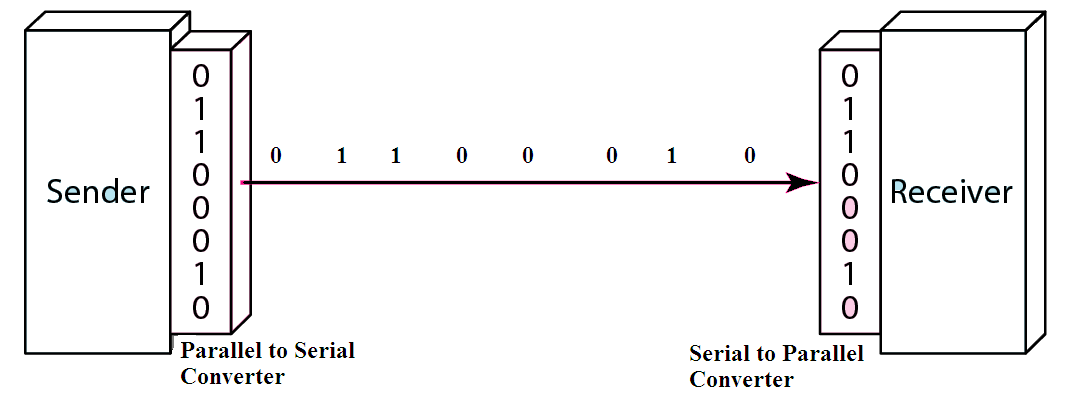
**Basics of Serial communication**

Data transfer between two electronic devices (Ex Between a computer and microcontroller or a peripheral device) is generally done in two ways

(i).Serial data Transfer and

(ii).Parallel data Transfer

Serial communication uses only one or two data lines to transfer data and is generally used for long distance communication. In serial communication the data is sent as one bit at a time in a timed sequence on a single wire. Serial Communication takes place in two methods, Asynchronous data Transfer and Synchronous data Transfer.



**Serial Data Transfer**

**Asynchronous data transfer** allows data to be transmitted without the sender having to send a clock signal to the receiver. Instead, special bits will be added to each word in order to synchronize the sending and receiving of the data. When a word is given to the UART for Asynchronous transmissions, a bit called the "Start Bit" is added to the beginning of each word that is to be transmitted. The Start Bit is used to alert the receiver that a word of data is about to be sent, and to force the clock in the receiver into synchronization with the clock in the transmitter.



**Serial data transmission**

After the Start Bit, the individual bits of the word of data are sent .Here each bit in the word is transmitted for exactly the same amount of time as all of the other bits. When the entire data word has been sent, the transmitter may add a Parity Bit that the transmitter generates. The Parity bit may be used by the receiver to perform simple error checking. Then at least one Stop Bit is sent by the transmitter. If the Stop Bit does not appear when it is supposed to, the UART considers the entire word to be corrupted and will report a Framing Error.

Baud rate is a measurement of transmission speed in asynchronous communication , it represents the number of bits/sec that are actually being sent over the serial link. The Baud count includes the overhead bits Start, Stop and Parity that are generated by the sending UART and removed by the receiving UART.

In the **Synchronous data transfer** method the receiver knows when to “read” the next bit coming from the sender. This is achieved by sharing a clock between sender and receiver. In most forms of serial Synchronous communication, if there is no data available at a given time to transmit, a fill character will be sent instead so that data is always being transmitted. Synchronous communication is usually more efficient because only data bits are transmitted between sender and receiver, however it will be more costly because extra wiring and control circuits are required to share a clock signal between the sender and receiver.

Devices that use serial cables for their communication are split into two categories.

1. DTE (Data Terminal Equipment). Examples of DTE are computers, printers & terminals.

2. DCE (Data Communication Equipment). Example of DCE is modems.

**Parallel Data Transfer :**

Parallel communication uses multiple wires (bus) running parallel to each other, and can transmit data on all the wires simultaneously. i.e all the bits of the byte are transmitted at a time. So, speed of the parallel data transfer is extremely high compared to serial data transfer. An 8-bit parallel data transfer is 8-times faster than serial data transfer. Hence with in the computer all data transfer is mainly based on Parallel data transfer. But only limitation is due to the high cost ,this method is limited to only short distance communications.

**Differences between Serial data transfer and Parallel data transfer**

|  |  |  |
| --- | --- | --- |
| **S.No** | **Serial Communication** | **Parallel Communication** |
| 1 | Data is transmitted bit after the bit in a single line | Data is transmitted simultaneously through group of lines(Bus) |
| 2 | Data congestion takes place | No, Data congestion |
| 3 | Low speed transmission | High speed transmission |
| 4 | Implementation of serial links is not an easy task. | Parallel data links are easily implemented in hardware |
| 5. | In terms of transmission channel costs such as data bus cable length, data bus buffers, interface connectors, it is less expensive | It is more expensive |
| 6 | No , crosstalk problem | Crosstalk creates interference between the parallel lines. |
| 7 | No effect of inter symbol interference and noise | Parallel ports suffer extremely from inter-symbol interference (ISI) and noise, and therefore the data can be corrupted over long distances. |
| 8 | The bandwidth of serial wires is much higher. | The bandwidth of parallel wires is much lower. |
| 9 | Serial interface is more flexible to upgrade , without changing the hardware | Parallel data transfer mechanism rely on hardware resources and hence not flexible to upgrade. |
| 10 | Serial communication work effectively even at high frequencies. | Parallel buses are hard to run at high frequencies. |

**SERIAL COMMUNICATION IN 8051 MICROCONTROLLER**

The 8051 has two pins for transferring and receiving data by serial communication. These two pins are part of the Port3(P3.0 &P3.1) .These pins are TTL compatible and hence they require a line driver to make them RS232 compatible .Max232 chip is one such line driver in use. Serial communication is controlled by an 8-bit register called SCON register,it is a bit addressable register.

**SCON (Serial control) register :**

****

|  |  |  |
| --- | --- | --- |
| SM0 | SCON.7 | Serial port mode selector |
| SM1 | SCON.6 | Serial port mode selector |
| SM2 | SCON.5 | Used for multiprocessor mode communication (not applicable for 8051) |
| REN | SCON.4 | Receive enable. Set or cleared by making this bit either 1 or 0 foe enable /disable reception. |
| TB8 | SCON.3 | 9th data bit transmitted in modes 2 and 3 |
| RB8 | SCON.2 | 9th data bit received in modes 2 and 3.it is not used in mode 0 & mode 1.If SM2 = 0 RB8 is the stop bit . |
| TI | SCON.1 | Transmit interrupt flag |
| RI | SCON.0 | Receive interrupt flag. |

* M0 , SM1 : These two bits of SCON register determine the framing of data by specifying the number of bits per character and start bit and stop bits. There are 4 serial modes.

SM0 SM1

0 0 : Serial Mode 0

0 1 : Serial Mode 1, 8 bit data, 1 stop bit, 1 start bit

1 0 : Serial Mode 2

1 1 : Serial Mode 3

* REN (Receive Enable) also referred as SCON.4. When it is high,it allows the 8051 to receive data on the RxD pin. So to receive and transfer data REN must be set to 1.When REN=0,the receiver is disabled. This is achieved as below

SETB SCON.4

& CLR SCON.4

* TI (Transmit interrupt) is the D1 bit of SCON register. When 8051 finishes the transfer of 8-bit character, it raises the TI flag to indicate that it is ready to transfer another byte. The TI bit is raised at the beginning of the stop bit.
* RI (Receive interrupt) is the D0 bit of the SCON register. When the 8051 receives data serially ,via RxD, it gets rid of the start and stop bits and places the byte in the SBUF register. Then it raises the RI flag bit to indicate that a byte has been received and should be picked up before it is lost. RI is raised halfway through the stop bit.

**Communication through RS232**

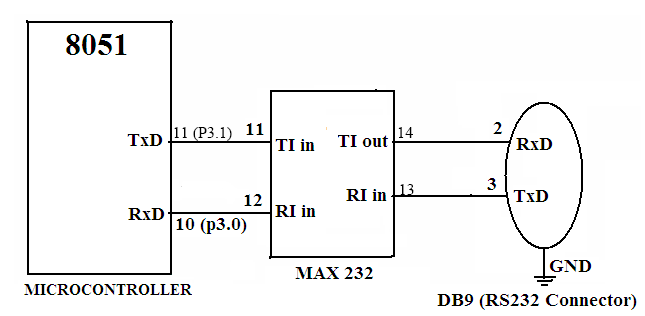
A personal computer has a serial port known as communication port or COM Port used to connect a modem for example or any other device, there could be more then one COM Port in a PC. Serial ports are controlled by a special chip called UART (Universal Asynchronous Receiver Transmitter).

RS 232 standard describes a communication method where information is sent bit by bit on a physical channel. The **RS stands for Recommended Standard**.The information must be broken up in data words. The length of a data word is variable.

It is one of the popularly known interface standard for serial communication between DTE & DCE. This RS-232-C is the commonly used standard when data are transmitted as voltage .This standard was first developed by Electronic industries association(EIA). For the RS-232C, a 25 pin D type connector is used . DB-25P male and DB-25S female. RS-232 standard was first introduced in 1960’s by Telecommunications Industry Association(TIA).

**Interfacing the 8051 Microcontroller to PC:**

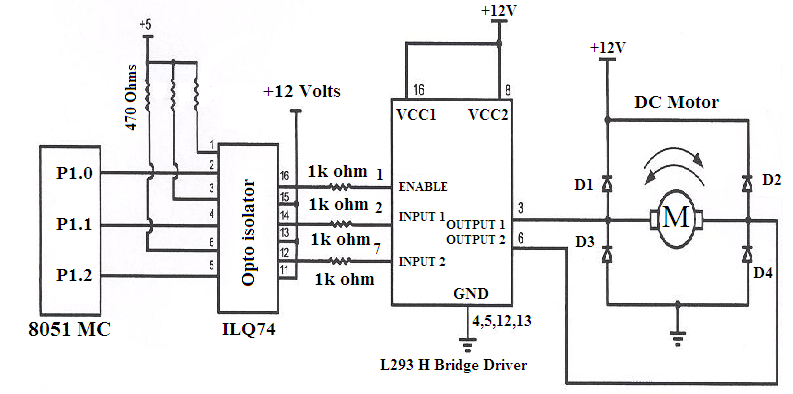
As the RS-232 standard is developed earlier to TTL devices ,a USART such as 8251 is not directly compatible with these signal levels .Because of this ,voltage transistors called line drivers and line receivers are used to interface TTL logic with RS-232 signals . The line driver MC 1488 is used to convert RS-232 to TTL.The microcontroller is connected to the PC using the DB9 connector.



The TxD and Rx D pins are connected to the TI in and RI in pins of the MAX 232 IC and the TI out and RI in pins of the MAX IC are connected to the RxD and TxD pins of the DB9 connector as shown in the interface diagram.

**INTERFACING DC MOTOR- 8051**

A DC motor runs with the help of Direct Current. It produces torque by using both electricity and magnetic fields. The DC motor has rotor, stator, field magnet, brushes, shaft, commutator. The DC motor requires more current to produce initial torque than in running state.Interfacing the DC motor directly to 8051 microcontroller is not possible. Because the DC motor uses large current (200-300mA in small DC motors) to run. When this current flow into the 8051 microcontroller, the IC will get damaged. Therefore we use a driving circuit with an opto isolator and a L298 Dual H-Bridge driver. The opto-isolator provides additional protection to the microcontroller.



Continuous, sustained operation of the motor will cause the L293 Dual H-Bridge driver to overheat. So,a suitable heat sink must be used.

**Assembly Language program**

|  |  |  |  |
| --- | --- | --- | --- |
| **ORG** | **0000H** |  | **Remarks** |
| **MAIN** | **CLR** | **P1.0** |  |
|  | **CLR** | **P1.1** |  |
|  | **CLR** | **P1.2** |  |
|  | **SETB** | **P2.7** |  |
| **MONITOR** |  |  |  |
|  | **SETB** | **P1.0** | **Enable the H-bridge driver** |
|  | **JNB** | **P2.7 CLOCKWISE** |  |
|  | **CLR** | **P1.1** | **01 is for Counter clockwise** |
|  | **SETB** | **P1.2** |  |
|  | **SJMP** | **MONITOR** |  |
| **CLOCKWISE** | **SETB** | **P1.1** | **10 is for clockwise** |
|  | **CLR P1.2** |  |  |
|  | **SJMP** | **MONITOR** |  |

**INTERFACING DAC -8051 MICROCONTROLLER**

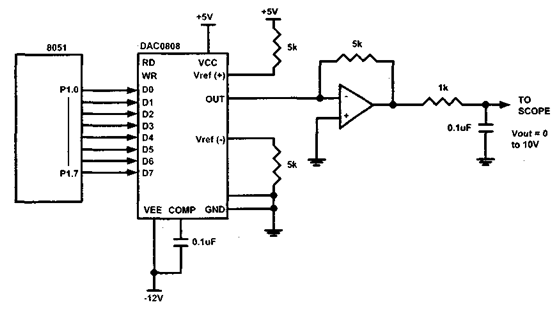
The DAC 0800 is a simple monolithic 8-bit D/A converter. It has fast settling time of 100ns. It can be directly interfaced to TTL, CMOS, PMOS and others. It operates at 4.5V to +18V supply. The number of data bit inputs decides the resolution of the DAC since the number of analog output levels is equal to 2″, where n is the number of data bit inputs. Therefore, an 8-input DAC such as the DAC0808 provides 256 discrete voltage (or current) levels of output.

The interfacing circuit is shown below. port 1(8 bits of the microcontroller is connected to the input data lines of DAC-08.The reference current is determined by the resistor R1 and the reference voltage V ref. The resistor R2 is generally equal to R1 to match the input impedance of reference source. The output (taken from pin number 4 is observed either on a digital multimeter or on a cathode ray oscilloscope.

The output current Io is calculated as follows:

Io = Vref/R1[Ao/2 + A1/4 + A2/8 + … +A7/256]

The output voltage Vo is obtained as follows: Vo =Io  \* R1



**Assembly Language Program**

MOV A, #DATA\* ; (A) = #Data

START : MOV 90H, A ; (port -1) = (A)

INC A

LJMP START ; Repeat

---------------------xxx---------------------

**REFERENCE**  : Muhammed Ali Mazidi, Janice Gillies Pie Mazidi, “The 8051 Microcontroller and Embedded Systems”— Pearson EducationAsia.