**UNIT-1**

**Introduction to Micro Processors**

**Definition of microprocessor:**

We can define the microprocessor based on 3 things. They are

1. Based on the application of the device.

* The CPU of any microcomputer is called microprocessor.

1. Based on the name of the device.

* A small device which is able to do data processing is called microprocessor.

1. Based on the construction and operation of the device.

* The Microprocessor is a programmable device that takes binary numbers as input, performs on them arithmetic or logical operations according to the program stored in memory and then produces other numbers as an output.

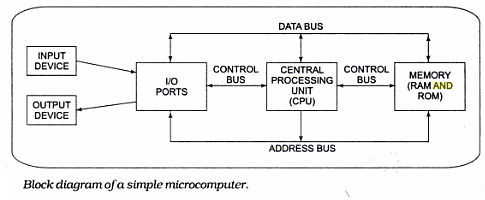
**History of microprocessors: -** Main parameter 🡪 word length

**Definitions of word length: -** The no. of bits processed by the CPU at a time are called word length.

1. **1ST Generation Microprocessors: -**For these processors Word length= 4bits.Examples for this generation processors are: 4004, 4040. These processors designed by using PMOS technology.4004 was the first processor in the world. It was introduced in the year 1971 by Intel Corporation. It is designed basically for calculator (Abacus). 4040 is the advanced version in 4-bit processors. It was introduced in the year 1972. The difference between the 4004 and 4040 is the operating clock frequency.
2. **2ND Generation Microprocessors: -** For these processors Word length=8bits.Examples for this generation processors are: 8008, 8080, 8085, M6800, Z80. The processor 8080 requires 3-power supplies for its operation, those are +12V, -12V and +5V.The 8085 Introduced in 1975 this is most popular processor in the 2nd generation processors. NMOS technology was used for the designing of this processor. The Speed of these processors is high when compared to 1st generation microprocessors.
3. **3RD Generation Microprocessors: -** For these processors Word length=16bits. Examples for this generation processors are: 8086, 8088, 80186, 80286, M68000, Z8080. The first 16-bit processor is 8086 it was introduced in the year 1978.The HMOS technology was used for 8086 processor. The 8088 is the combination of 8085&8086 features. Its operation like 8086 when it performs internal operations and its operation is like 8085 when it performs external devices related operations. The speed of these processors is high as compared to previous generation (2nd generation) processors.
4. **4TH Generation Microprocessors: -** For these processors Word length=32bits.Examples for this generation processors are: 80386, 80486, M68020, M68030.Virtual memory and cache memory concepts were introduced from 80386 onwards. Instruction pipe-lining concept was introduced from 80486 onwards. In this 80486 processor there is a 4-stage Instruction pipe lining. The speed of these processors is high when compared to 3rd generation microprocessors.
5. **5TH Generation Microprocessors: -** For these processors Word length=64bits.Examples for this generation processors are: Pentium [80586], Pentium pro, Pentium -2 etc. The Speed of these processors is high when compared to 4th generation microprocessors.

**OVERVIEW OF MICROCOMPUTER STRUCTURE AND OPERATION**

Figure below shows a block diagram for a simple microcomputer. The major parts are the central processing unit or CPU, memory, and the input and output circuitry or I/O. Connecting these parts are three sets of parallel lines called buses. The three buses are the address bus, the data bus, and the control bus.



**Memory**: The memory section usually consists of a mixture of RAM and ROM. Memory has two purposes. The purpose is to store the binary codes for the sequences of instructions with which the computer is going to be working.

**Input/output**: The input/output or I/O section allows the computer to take in data from the outside world or send data to the outside world. Peripherals such as keyboards, video display terminals, printers, and modems are connected to the I/O section. These allow the user and the computer to communicate with each other. The actual physical devices used to interface the computer buses to external, systems are often called ports. An input port allows data from a keyboard, an A/D converter, or some other source to be read into the computer under control of the CPU. An output port is used to send data from the computer to some peripheral, such as a video display terminal, a printer, or a D/A converter.

**Central Processing Unit**: The central processing unit or CPU controls the operation of the computer. In a microcomputer the CPU is a microprocessor. The CPU fetches binary-coded instructions from memory, decodes the instructions into a series of simple actions, and carries out these actions in a sequence of steps. The CPU also contains an address counter or instruction pointer-register which holds the address of the next instruction or data item to be fetched from memory; general-purpose registers, which are used for temporary storage of binary data; and circuitry, which generates the control bus signals.

**Address Bus**: The address bus consists of 16, 20, 24, or 32 parallel signal lines. On these lines the CPU sends out the address of the memory location that is to be written to or read from. The number of memory locations that the CPU can address is determined by the number of address lines. If the CPU has N address lines, then it can directly address 2N memory locations. For example, a CPU with 16 address lines can address 216 or 65,536 memory locations, a CPU with 20 address lines can address 220 or 1,048,576 locations, and a CPU with 24 address lines can address 224 or 16,777,216 locations. When the CPU reads data from or writes data to a port, it sends the port address out on the address bus.

**Data Bus** : The data bus consists of 8, 16, or 32 parallel signal lines. As indicated by the double-ended arrows, the data bus lines are bidirectional. This means that the CPU can read data in from memory or from a port on these lines, or it can send data out to memory or to a port on these lines. Many devices in a system will have their outputs connected to the data bus, but only one device at a time will have its outputs enabled. Any device connected on the data bus must have three-state outputs so that its outputs can be disabled when it is not being used to put data on the bus.

**Control Bus**: The control bus consists of 4 to 10 parallel signal lines. The CPU sends out signals on the control bus to enable the outputs of addressed memory devices or port devices. Typical control bus signals are Memory Read, Memory Write, I/O Read, and I/O Write. To read a byte of data from a memory location, for example, the CPU sends out the memory address of the desired byte on the address bus and then sends out a Memory Read signal on the control bus. The Memory Read signal enables the addressed memory device to output a data word onto the data bus. The data word from memory travels along the data bus to the CPU.

**Features of 8086: -**

1. The Intel 8086 is a 16-bit microprocessor. The term 16-bit means that its arithmetic logic unit,   
 its internal registers, and most of its instructions are designed to work with 16-bit binary   
 words.

2. The 8086 has a 16-bit data bus, so it can read data from or write data to memory and ports   
 either 16 bits or 8 bits at a time. The 8086 has a 20-bit address bus, so it can address any one   
 of 220, or 1,048,576, memory locations.

3. 8086 processor has 20 address lines A19-A0 and 16 data lines D15-D0. The data lines are   
 multiplexed with lower order 16 address lines, and then the multiplexed address and data lines   
 are AD15-AD0. The remaining higher order 4 address lines A16-A19 are multiplexed with the   
 status lines S3-S6.

4. 8086 processor is available in 40-pin DIP (Dual in line Package).

5. The operating clock frequencies of 8086 processors are 5MHz, 8MHz, and 10MHz.

6. 8086 processor supports 256 interrupts.

8. 8086 processor provides fourteen 16 -bit registers.

9 It requires single phase clock with 33% duty cycle to provide internal timing.

10. 8086 operates in two different modes. 1. Minimum mode or Single-Processor mode

2. Maximum mode or Multi-Processor mode.

11.8086 includes few features, which enhance multiprocessing capability (it can be used with

math coprocessors like 8087, I/O processor 8089 etc.).

12. 8086 available in three different versions 8086, 8086\_1 and 8086\_2.

13. It can pre-fetches up to 6 instruction bytes from memory and queues them in order to speed up

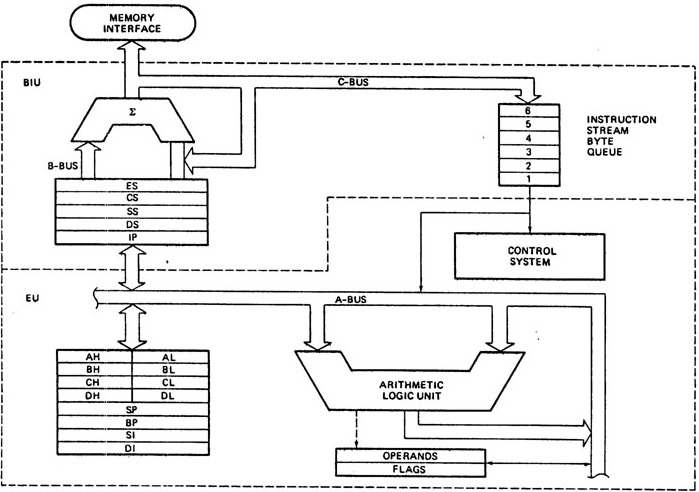
instruction execution.

14. It was implemented in the year 1978 by Intel corp. by using HMOS (hybrid metal oxide semi- conductor or high speed MOS or high density MOS) technology.

**Architecture of 8086: -** The entire architecture of 8086 is divided into two independent functional parts. They are

1) BIU: Bus Interface Unit 2) EU: Execution Unit

These two functional units can work simultaneously to increase system speed and hence the throughput. Throughput is a measure of number of instructions executed per unit time.



**BIU (Bus Interface Unit): -**

The functions performed by the BIU are

1. It sends address of the memory or I/O.

2. It fetches instruction from memory.

3. It reads data from port/memory.

4. It writes data into port/memory.

5. It supports instruction queuing.

6. It provides the address relocation facility.

**BIU has the following functional parts**

a) Summer circuit (or) adder circuit.

b) Segment registers & IP.

c) 6 Byte instruction pre-fetch queue.

1. **Summer Circuit: -** It produces a 20-bit physical address.

**b) Segment Registers: -** The BIU contains four 16-bit segment registers. They are: the Extra segment (ES) register, the Code segment (CS) register, the Data segment (DS) register, and the Stack segment (SS) register. These segment registers are used to hold the upper 16 bits of 20-bit Physical Address. It is called the Segment Base and it is also called the Base Address for each of the segment. The part of a segment starting address stored in a segment register is often called the Segment Base.

**Instruction pointer:** - Its functionality is similar to that of PC (Program Counter). It holds the address of next instruction byte to be fetched from the code-segment. It means it gives offset address of the instruction code bytes in the Code Segment.

**INSTRUCTION QUEUE (IQ) AND PIPELINING:**

* Microprocessor 8086 has six registers of 8-bit each in the instruction queue (IQ).
* Initially, all the instruction codes of a program will be present in memory location; to execute the program microprocessor has to perform two operations.

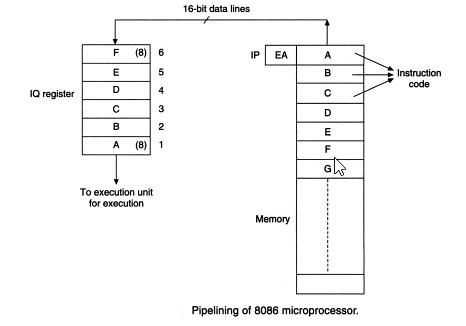
(a) Microprocessor has to read the instruction code from memory. i.e. opcode fetch operation.

(b) Microprocessor has to execute this fetched instruction.

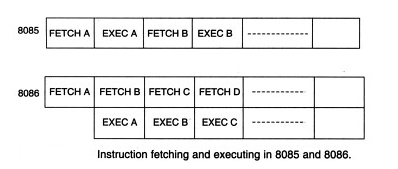
* Initially, BIU of 8086 will read six byte of instruction codes from memory location and these codes are stored in the six registers of IQ. To execute any instruction, the EU of 8086 reads the instruction code from register-1 of IQ. So 8086 performs two operations together:

1. EU executes the instruction present in the IQ register.

2. At the same time BIU prefetches the next instruction codes from memory and stores them   
 in vacant register of IQ.



* These two parallel operations. i.e. execution and prefetching are called pipelining. Fetching the next instruction while the current instruction executes is called pipelining. It increases the speed of operation of microprocessor 8086. If any branching instruction like JMP, CALL comes in the program, then BIU will flush or clear IQ register and the instruction codes are again fetched from the new branching memory address.



* If the Queue size is greater than 6B then wait state period of processor is increased because if any instruction in the queue is related to **Branching,** then instructions in the queue must be cleared as the next instructions must be taken from the branch specified location. In this period processor must be in wait state.

**EU (Execution Unit): -**

The functions performed by EU are

a) It tells the BIU where to fetch instructions or data from,

b) Decodes instructions, and

c) Executes instructions.

* The EU contains the control circuitry to perform various internal operations. A decoder in EU

decodes the instruction which is fetched from memory and 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.

**EU has the following functional parts these are**

a) ALU

b) Register Set.

c) Operand & Flag Register.

d) Control System.

**a) ALU (Arithmetic and Logical Unit):-** In 8086, ALU is of 16 bits hence 8086 is called 16-bit microprocessor. It performs Arithmetic and Logical operations on 8 bits/16bits.

ALU can do the following arithmetic operations

i) Addition ii) Subtraction iii) Multiplication

iv) Division v) Increment vi) Decrement

##### The ALU can also perform logical operations such as

i) NOT ii) AND iii) OR

iv) EX-OR v) TEST vi) Logical Shift

vii) Arithmetic Shift viii) Circular Shift (or) Rotate

**b) Register Set:**- It is used to hold the 16-bit information. The information is address, data or result of some operation.

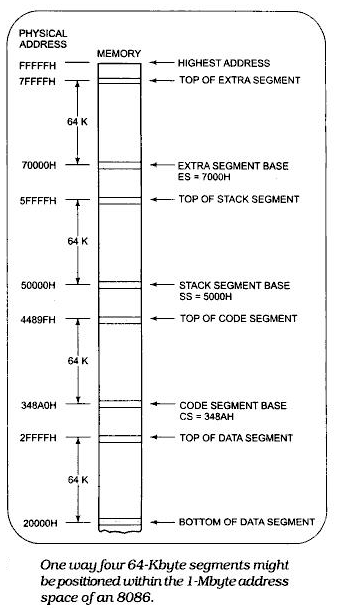
**c) Operand &Flag Register: -** The 16-bit register having 16 F/Fs can store maximum 16-bit data. Operand register cannot be used by the programmer/user. It is used only by the microprocessor to store any intermediate data or result; therefore, it is called the temporary register. Flags are also called as PSW(program status word) of 8086. Each single bit is called flag.

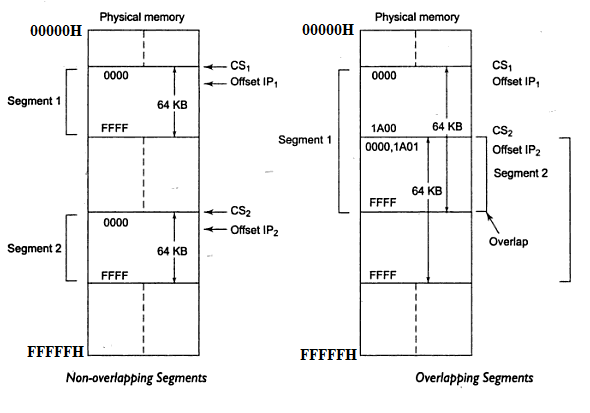
**d) Control System: -** It is divided into 2 parts. They are

1. Decoding circuit----it decodes the instruction.
2. Timing circuit---- it generates control signals at appropriate times.

**Memory segmentation:**

* The 8086 BIU sends out 20-bit addresses, so it can address any of 220 or 1,048,576 bytes in memory. However, at any given time the 8086 works with only four 65,536 byte (64-Kbyte) segments within this 1,048,576 byte (1-Mbyte) range.
* Four segment registers in the BIU are used to hold the upper 16 bits of the starting addresses of four memory segments that the 8086 is working with at a particular time. The four segment registers are the code segment (CS) register, the stack segment (SS) register, the extra segment (ES) register, and the data segment (DS) register.
* Figure shows how these four segments might be positioned in memory at a given time. The four segments can be separated as shown, or, for small programs which do not need all 64 Kbytes in each segment, they can overlap.





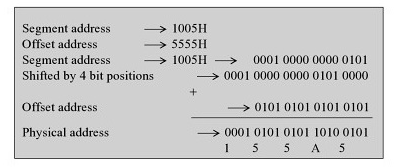
* As mentioned a segment register is used to hold the upper 16 bits of the starting address for each of the segments. The code segment register, for example, holds the upper 16 bits of the starting address for the segment from which the BIU is currently fetching instruction code bytes.
* The BIU always inserts zeros for the lowest 4 bits (nibble) of the 20-bit starting address for a segment. If the code segment register contains 348AH, for example, then the code segment will start at address 348A0H. In other words, a 64-Kbyte segment can be located anywhere within the 1-Mbyte address space, but the segment will always start at an address with zeros in the lowest 4 bits.
* The part of a segment starting address stored in a segment register is often called the segment base.
* An alternate way of representing a 20-bit physical address is the

Segmentbase : offsetform

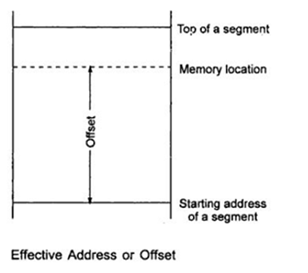
* Ex:- CS:IP

DS:SI

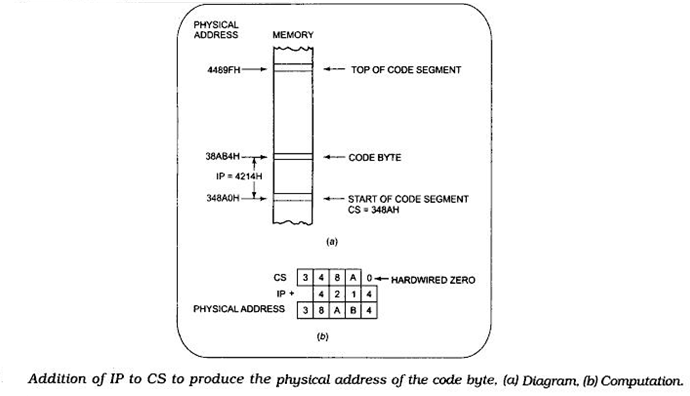
* **Physical Address = Base Address \* 10H + Offset Address**
* For generating a physical address from contents of these two registers, the content of a segment register also called as segment address is shifted left bit-wise four times and to this result, content of an offset register also called as offset address is added, to produce a 20-bit physical address.

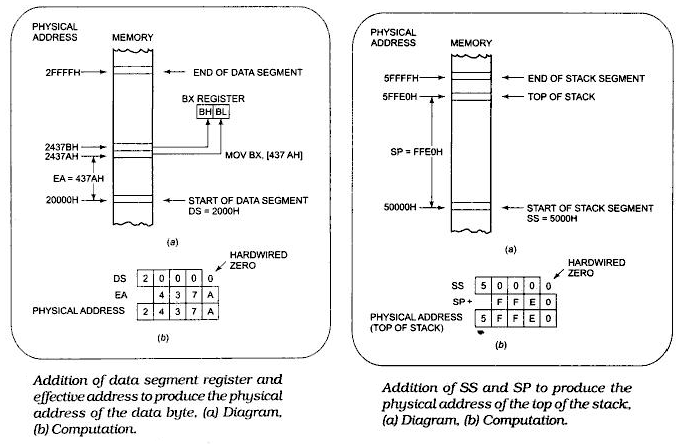


* The segment register indicates the base address of a particular segment while the offset indicates the distance of the required memory location in the segment from the base address. Since the offset is a 16-bit number each segment can have a maximum of 64K locations. Thus, the segment addressed by the segment value 1005H can have offset values from 0000H to FFFFH.



* The bus interface unit has a separate adder to perform this procedure for obtaining a physical address while addressing memory. The segment address value is to be taken from an appropriate segment register depending upon whether code, data or stack are to be accessed, while the offset may be the content of IP, BX, SI, DI, SP, BP or an immediate 16-bit value, depending upon the addressing mode.



****

* The segment:offset scheme requires only a 16-bit number to represent the base address for a segment, and only a 16-bit offset to access any location in a segment. This means that the 8086 has to manipulate and store only 16-bit quantities instead of 20-bit quantities. This makes for an easier interface with 8- and 16-bit-wide memory boards and with the 16-bit registers in the 8086.
* In a timesharing system, several users share a CPU. The CPU works on one user's program for perhaps 20 ms then works on the next user's program for 20 ms. After working 20 ms for each of the other users, the CPU comes back to the first user's program again. Each time the CPU switches from one user's program to the next, it must access a new section of code and new sections of data. Segmentation makes this switching quite easy. Each user's program can be assigned a separate set of logical segments for its code and data. The user's program will contain offsets or displacements from these segment bases. To change from one user's program to a second user's program, all that the CPU has to do is to reload the four segment registers with the segment base addresses assigned to the second user's program. In other words, segmentation makes it easy to keep users' programs and data separate from one another, and segmentation makes it easy to switch from one user's program to another user's program.

**Advantages due to Memory Segmentation: -**

* It allows the memory addressing capacity to be 1 Mbyte even though the address associated with individual instruction is only 16-bit.
* It allows instruction code, data, stack, and portion of program to be more than 64 KB long by using more than one code, data, stack segment, and extra segment.
* It facilitates use of separate memory areas for program, data and stack.
* It permits a program or its data to be put in different areas of memory, each time the program is executed i.e. program can be relocated which is very useful in multiprogramming.
* Program can work on several data sets by reloading the DS register.

**Register Organization of 8086: -**

In general there are 2 types of registers in any CPU. They are

**1. Machine accessible registers** – These Registers are only used by the Machines. User was unable to use these registers by programming instructions.

**2. User accessible registers: -**These Registers are used by the programmer in programming. These are called by the user in program by using programming instructions.

The 8086 processor also contains both types of registers.

In 8086 there are 14 registers. These are divides into 5 groups and these are User accessible registers, each register size is 16-bit.

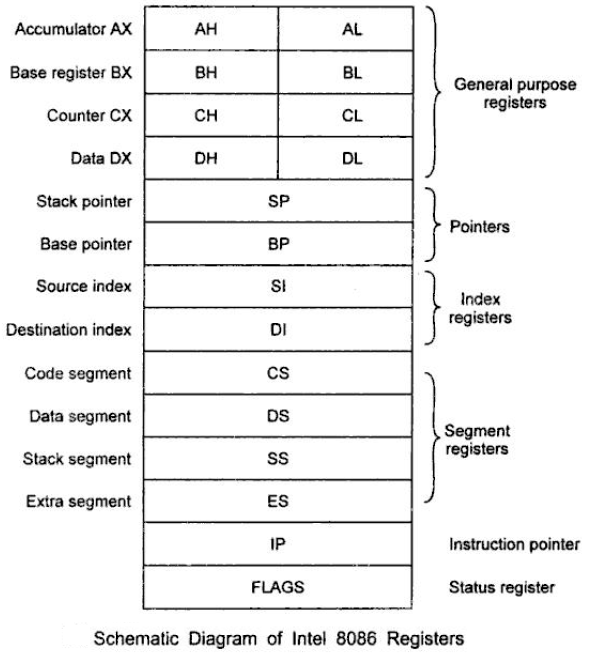
I. General Purpose Registers[GPRs] ---4

II. Segment Registers---4

III. Pointer Registers---3

IV. Index Registers---2

V. Flag register---1



**I). General Purpose Registers [GPRs]: -**There are 4 GPRs. They are

i). AX---Accumulator Reg ii). BX---Base Reg

iii). CX---Count Reg iv). DX---Data Reg

* These registers are able to hold data, address or results on temporary basis.
* After completion of any operation by CPU, the result is loaded on to GPRs in some cases.
* For 8-bit operations, each these 16-bit registers can be divided into two 8-bit registers.

AX🡪 AH+AL BX🡪 BH+BL

CX🡪 CH+CL DX🡪 DH+DL

**Special Functions of GPRs: -**

* **AX (Accumulator): -**

1. AX acts as source/destination for some arithmetic/logical operations.
2. If the processor is communicating with any input or output device, then entire communication is done through accumulator register.
3. It acts as interface between CPU and I/O device.

* **BX (Base Register): -**

This register is used to hold the offset value or the part of offset value of an operand in some addressing modes.

* **CX (Counter Register): -**

1. This register is used as a counter to perform repeated operations by using **loop** instruction.
2. If the processor performs any **string** related operations, count value for these operations must be the content of CX register.

* **DX (Data Register): -**

1. DX register is used to hold the address of an I/O device in I/O indirect addressing.
2. Whenever the processor executes 16bit\*16bit multiplication, the result size is 32-bit, in that higher order 16-bits are loaded onto DX and lower order 16-bits are loaded on AX.
3. If the processor performs 32-bit/16-bit division, in that dividend 32-bit, higher order 16-bit are content of DX. After completion of division remainder is loaded onto DX.

**II). Segment Registers: -**There are 4 segment registers; each register size is 16-bit.

1. CS[code segment registers]
2. DS[data segment registers]
3. SS[stack segment registers]
4. ES[extra segment registers]

* These segment registers are used to hold the upper 16 bits of the **Starting Address** or **Base Address** for each of the segments.  The part of a segment starting address stored in a segment register is often called the **Segment Base**.

**III). Pointer Registers: -**There are 3 pointer registers, with 16-bit length, these are

1. Instruction pointer(IP)
2. Stack pointer(SP)
3. Base pointer(BP)

**IP (Instruction Pointer): -** Holds the address of next instruction byte in code segment. It provides offset address in code segment.

**SP (Stack Pointer): -** Gives the offset value of stack segment (SS). It holds the address of the Top of the Stack. (Or) it holds the address of the stack where the last stack related operation is performed.

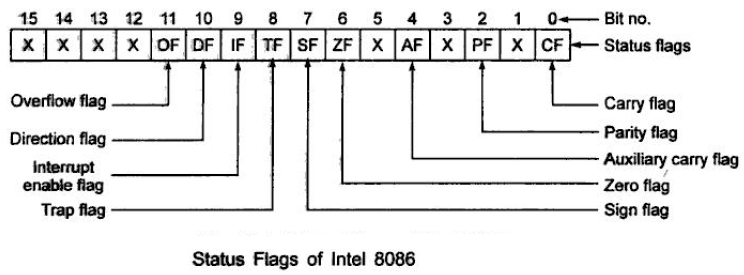
**BP (Base Pointer): -** Gives the offset value of DS or SS. It is used primarily to access parameters passed via the stack.

**IV) Index Registers: -** There are 2 registers, each 16-bit size.

1. SI (Source index register)
2. DI (Destination index register)

* These are used for string related operations
* If the processor executes any string related operation, the source address of the string is provide by the combination of DS and SI registers. SI provides Offset address relative to DS.
* Similarly, destination address is provided by the combination of ES and DI registers. DI provides Offset address relative to ES

**V) Flag Register: -**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. It is also called PSW of 8086.



* X--- Unused bit positions(total 7)
* Remaining 9 are active bit positions. Active flags of 8086.
* OV=overflow flag DF=direction flag TF=trap flag IF=interrupt flag SF=sign flag
* ZF=zero flag AC=auxiliary carry flag PF=parity flag CY=carry flag
* 9 active flags can be divided into 2 groups.

1. Conditional flags----6 b) Control flags----3

**a) Conditional Flags: -**

* Also called status flags.
* Gives the information of conditions of the result produced by the ALU.
* Gives the status of ALU after completion of arithmetic/logical operations.
* All the flags in LSB side are status flags along with OV in MSB side.

**CF (Carry flag):** When the microprocessor performs the addition of two 8/16-bit numbers, then the result obtained will be of 9/17 bits. The last carry generated i.e. 9/17th bit is stored in CF.

Similarly, when the microprocessor performs the subtraction of two 8/16-bit numbers X - Y, then:

* If X> = Y, then additional borrow required to perform the subtraction X- Y is zero, so CF = 0.
* If X <Y, Men additional borrow required to perform the subtraction X -Y is one, so CF = 1.

**PF (Parity flag):** When the microprocessor performs any arithmetic or logical operations, then the status of only 8 Least Significant Bits of the result is stored in PF. The parity flag is set to 1, if the result contains even no. of 1’s in its LSByte. i.e.

* If the count of numbers of 1s bit is 0/2/4/6/8 in the 8 LSBs of the result, then the result is of even parity, so PF = 1.
* If the count of numbers of 1s bit is 1/3/5/7 in the 8 LSBs of the result, then the result is of odd [parity, so](http://parity.so) PF = 0.

**AC (Auxiliary carry flag):** When the microprocessor performs the addition of two 8/16-bit numbers, then the carry bit generated after adding 4 Least Significant Bits (nibble) is stored in AC

flags. Similarly, when the microprocessor performs the subtraction of two 8/16 bit-numbers then the borrow required to perform subtraction of 4 LSBs is stored in AC flags.

This is not a general-purpose flag; it is used internally by the processor to perform Binary to BCD conversion

**ZF (Zero flag):** When the microprocessor performs any arithmetic or logical operations of the 8-bit number, then all the 8 LSBs of the result are zero, i.e. 00h, then ZF = 1 otherwise ZF = 0. Similarly, when the microprocessor performs any arithmetic or logical operations of the 16-bit number, then all the 16 LSBs of one result are zero 0000h, then ZF = 1 otherwise ZF = 0.

**SF (Sign flag):** When the microprocessor performs any arithmetic or logical operations of the 8/16-bit number, then Most Significant Bits of the 8–bit result/16-bit result are directly stored into sign flags. If the result obtained is correct binary number then the sign flags bit will give correct sign of the result. If SF = 0/1, then the result is +ve/-ve respectively.

**OF (Overflow flag):** When the microprocessor performs any arithmetic or logical operations of the 8/16-bit sign binary numbers and when the sign binary result of the 8/16-bit is out of the range, i.e. incorrect, then OF = 1, so sign flag bit will also be incorrect. If the 8/16-bit sign binary result is within the range, i.e. the result is correct sign binary number, then OF = 0, so sign flag bit will give correct sign of the result.

When the microprocessor performs the addition of two 8/16-bit numbers, then:

1. If carry into the MSBs is equal to the carry out of the MSBs, then the sign result is correct and OF = 0.
2. If carry into the MSBs is not equal to the carry out of the MSBs, then the sign result is incorrect and OF = 1.

Similarly, when the microprocessor performs the subtraction of 8/16-bit numbers, then:

1. If borrow into the MSBs is equal to the borrow out of the MSBs, then the sign result is correct and OF = 0.
2. If borrow into the MSBs is not equal to the borrow out of the MSBs, then sign result is incorrect and OF = 1.

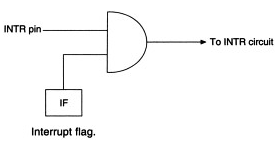
**b) Control Flags: -** Control flags are set or reset deliberately to control the operations of the execution unit. Control flags are as follows: These flags are set or reset by user.

**TF (Trap flag):** Trap flag is used to detect any error in a program (debugging) by executing the program in single stepping mode.

* If logic 0 is stored in TF, then microprocessor will execute all the instructions of a program in one operation (free run operation).
* If logic 1 is stored in TF, then microprocessor will execute one instruction of the program at a time, after executing each instruction microprocessor will execute INT1 (software interrupt instruction), so microprocessor will branch from main program to subprogram. This subprogram is used to display the result obtained in different registers of microprocessor on CRT screen, so after each instruction programmer can verify the result. Hence, the error in the program can be detected.

**IF (Interrupt flag):** Interrupt flag is used to control maskable interrupt of microprocessor 8086

* If logic 0 is stored in IF, then INTR (interrupt) pin is disabled, i.e. even if INTR signal is applied on INTR pin, then microprocessor is not interrupted.
* If logic 1 is stored in IF, then INTR pin is enabled, i.e. when logic 1 is applied on INTR pin, then output of the AND gate is 1, so microprocessor gets interrupted.



* It can be set by executing instruction STI and can be cleared by executing CLI instruction.

**DF (Direction flag):** When the microprocessor executes string instructions, then the SI register is used to store effective address (EA) of the source memory location and the DI register is used to store effective address of the destination memory locations.

* To start data transfer from first numbers, initially starting source and destination address are stored in SI and DI. After each byte/word transfer, we have to increment the effective address of SI and DI by 1/2 respectively. This can be done by storing logic 0 in DF. Processor operates in auto-increment mode of operation.
* Similarly, to start the data transfer from last numbers, initially last memory address of source and destination are stored in SI and DI. After each byte/word transfer, we have to decrease the effective address of SI and DI by 1/2 respectively. This can be done by storing logic 1 in DF. Processor operates in auto-decrement mode of operation.

**Addressing Modes of 8086**

Operands may be of three types:

* + Implicit
  + Explicit
  + Both Implicit and Explicit.

**Implicit operands** mean that the instruction by definition has some specific operands. The programmers do NOT select these operands.

|  |
| --- |
| **Example: Implicit operands** |
| XLAT ; automatically takes AL and BX as operands  DAA ; it operates on the contents of AL. |

**Explicit operands** mean the instruction operates on the operands specified by the programmer.

|  |
| --- |
| **Example: Explicit operands** |
| MOV AX, BX; it takes AX and BX as operands  XCHG SI, DI; it takes SI and DI as operands |

**Implicit and explicit operands**

|  |
| --- |
| **Example: Implicit/Explicit operands** |
| MUL BX; automatically multiply BX explicitly times AX |

* The location of an operand value in memory space is called the **Effective Address (EA).**
* The method by which the address of source of data or the address of destination of result are given in the instruction is called addressing mode.

We can classify the addressing modes of 8086 into two groups:

* Addressing modes of data-related instructions.
* Addressing modes of branching instructions.

Addressing modes of data-related instructions are

### Register operand Addressing Mode

1. Immediate operand Addressing Mode
2. Implicit Addressing Mode

### Memory operand Addressing Modes

* + 1. Displacement-only (or Direct) Addressing Mode
    2. The Register Indirect Addressing Mode

### Indexed Addressing Mode

* + 1. Based Addressing Mode
    2. Relative Indexed Addressing Mode
    3. Relative Based Addressing Mode
    4. Based Indexed Addressing Mode
    5. Relative Based Indexed Addressing Mode

1. I/O Addressing
   * 1. Fixed port Addressing mode

b. Variable port Addressing mode

Addressing modes of branching instructions are

1. Intersegment Addressing mode
2. Direct mode

b. Indirect mode

1. Intrasegment Addressing mode
   1. Direct mode

b. Indirect mode

The different data related Addressing modes are described below: -

**Implicit / Implied Addressing Mode**

* If the address of source of data as well as the address of destination of result is fixed, then there is no need to give any operand along with the instruction.
* Such instruction is called Implicit addressing mode instruction.

EX: - STD

STC

DAA

XLAT

**Immediate Addressing Mode**

* In this Addressing mode the 8-bit/ 16-bit data required for executing the instruction is given directly along with the instruction.
* The immediate operand, which is stored along with the instruction, resides in the code segment -- not in the data segment.
* This addressing mode is also faster to execute an instruction because the operand is read with the instruction from memory.
* Also note that immediate data are constant data, whereas the data transferred from a register or memory location are variable data.
* Here are some examples:

|  |
| --- |
| **Example: Immediate Operands** |
| MOV AL, 20H ; move the constant 20 into register AL  ADD AX, 0005H ; add constant 5 to register AX  MOV DX, offset msg ; move the address of message to register DX |

**Register addressing mode**

* In this Addressing mode the 8-bit/ 16-bit data required for executing the instruction is present in the 8-bit/ 16-bit register and the name of this register is given along with the instruction.
* For register addressing modes, there is no need to compute the effective address. The operand is in a register and to get the operand there is no memory access involved.
* In this addressing mode, the registers used are:
* reg16: 16-bit general registers: AX, BX, CX, DX, SI, DI, SP or BP.
* reg8 : 8-bit general registers: AH, BH, CH, DH, AL, BL, CL, or DL.
* Sreg : segment registers: CS, DS, ES, or SS. There is an exception: CS cannot be a destination.

|  |
| --- |
| **Example: Register Operands** |
| MOV AX, BX ; mov reg16, reg16  ADD AX, SI ; add reg16, reg16  MOV DS, AX ; mov Sreg, reg16 |

* Some rules in register addressing modes:

1. You may not specify CS as the destination operand.

Example: mov CS, 02h –> wrong

2. Only one of the operands can be a segment register. You cannot move data from one segment register to another with a single mov instruction. To copy the value of CS to DS, you would have to use some sequence like:

MOV DS,CS -> wrong   
MOV AX, CS -> the way we do it  
MOV DS, AX

* You should never use the segment registers as data registers to hold arbitrary values. They should only contain segment addresses.

# [Memory Operand Addressing Modes](http://www.8085projects.info/post/Memory-Addressing-Modes.aspx)

* These addressing modes are used to specify the location of an operand in memory.
* To access data in memory, the 8086 must also produce a 20-bit physical address.
* It does this by adding a 16-bit value called the effective address to a segment base address represented by the 16-bit number in one of the four segment registers.
* The execution unit calculates the effective address for an operand using information specified in the instruction.
* We can use a number in the instruction as the effective address or to use the contents of a specified register as the effective address or to compute the effective address by adding a number in the instruction to the contents of one or two specified registers.

**Direct Addressing Mode**: -

* If the 8-bit/ 16-bit data required for executing the instruction is present in the memory location, the effective address of this memory location will be given directly along with the instruction then such instruction is called Direct Addressing mode instruction.
* The value of effective address given in the instruction will always be enclosed in brackets.

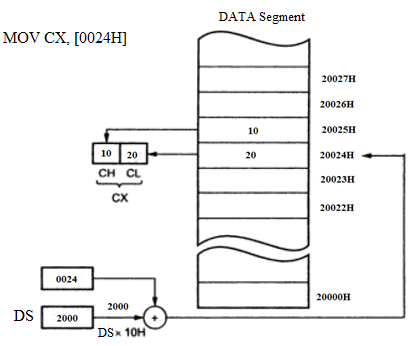
EX:-

MOV AL, [2500H]

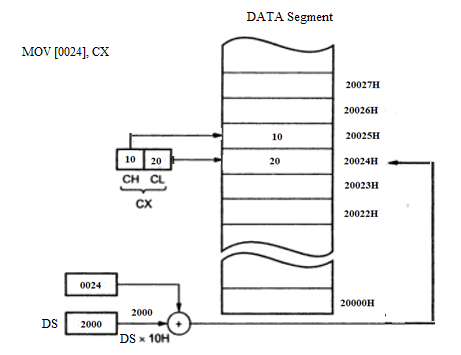
MOV RESULT, AX

MOV AL, ES:[2000H]

MOV CX, [0024H]



MOV [0024], CX



**Register indirect addressing**:-

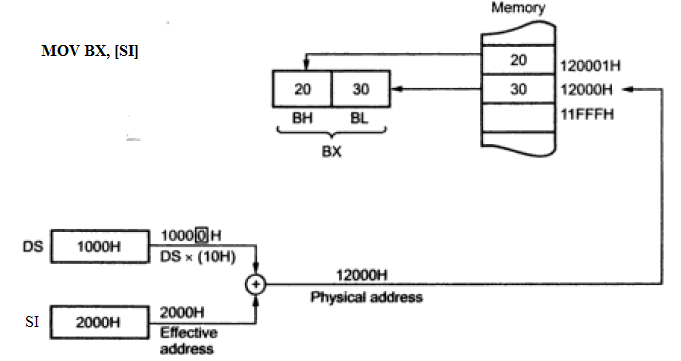
* If the 8-bit/ 16-bit data required for executing the instruction is present in the memory location, the 16- bit effective address of this memory location is present in Index (SI or DI) or base register (BX or BP) and then the name of this register is given along with the instruction enclosed in brackets then such instruction is called Register Indirect Addressing mode instruction.
* If the BP register addresses memory, the Stack segment is used by default.
* For the BX, SI and DI, the Data segment is used by default.

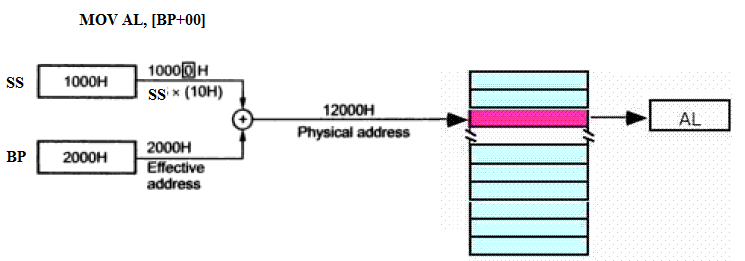
EX: - MOV AL, [BX]

MOV DX, [SI]

MOV BX, [BP+0] ; we cannot use [BP] without a displacement

MOV BX, [SI]





**BASE INDEX Addressing mode:-**

* If the 8-bit/ 16-bit data required for executing the instruction is present in the memory location, the 16- bit effective address of this memory location is obtained by adding the 16-bit number of the given base register [BX]/ [BP] along with the given index register [SI]/ [DI]
* For calculating physical address, microprocessor will always take the base address from default segment register of the given base register [BX]/ [BP].
  + - * + EA =(BX) / (BP) + (SI) / (DI)

EX: -

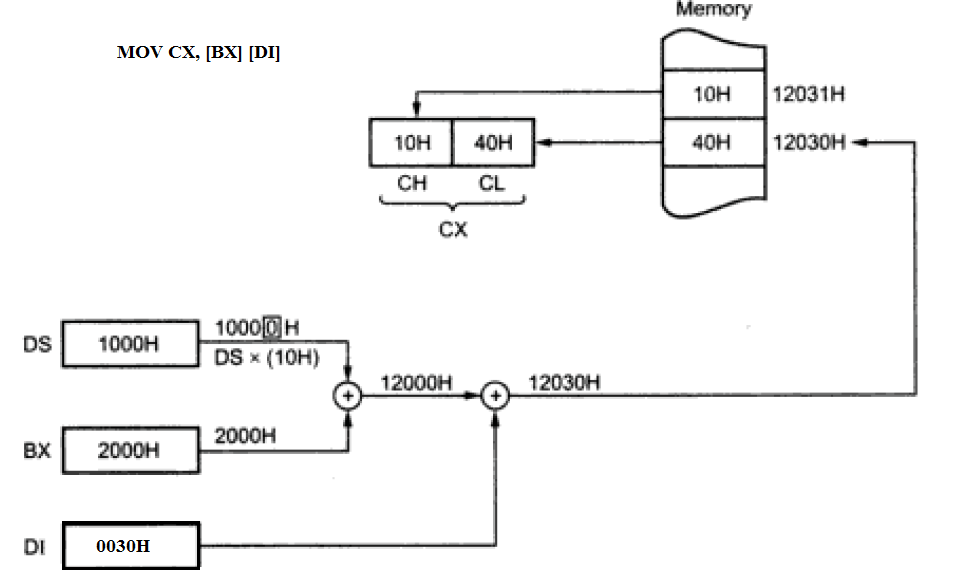
MOV AL, [BX] [SI]

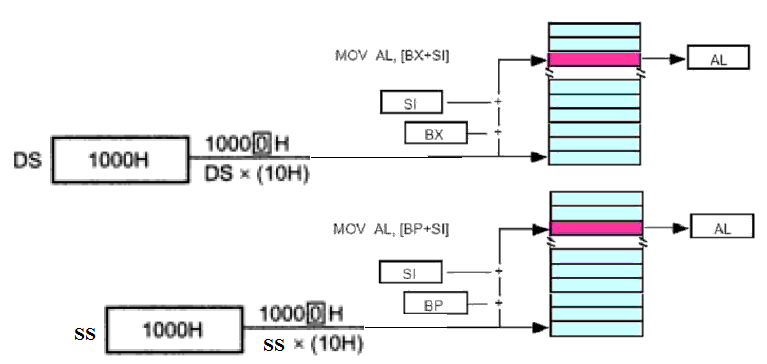
MOV CX, [BP] [DI]

MOV BH, [BP] [SI]

MOV BP, [BX] [DI]

MOV [BX] [SI], AX





**Register Relative Addressing mode:-**

* If the 8-bit/ 16-bit data required for executing the instruction is present in the memory location, the 16- bit effective address of this memory location is obtained by adding the 16- bit number of the register [BX] / [BP]/ [SI] / [DI] to the 8/16-bit displacement given along with the instruction.
* Such instruction is called Register Relative Addressing mode instruction.
* Remember that displacement should be added to the register within [ ].
  + - * + EA =(BX) / (BP) / (SI) / (DI) + 8/16-bit displacement

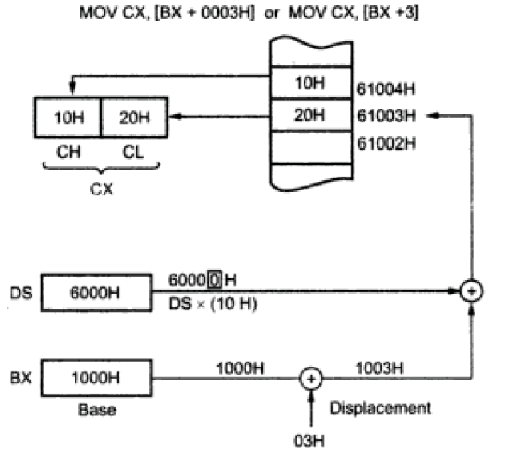
Register Relative Addressing mode can be further divided into two types.

1. **Relative Base addressing mode:**

* If EA is obtained by adding 16-bit number of base register (BP)/ (BX) along with displacement, then it is called Relative Base addressing mode
  + - * + EA =(BX) / (BP) + 8/16-bit displacement

Ex:- MOV AL,[BX+2]

MOV 06H [BX], AX



1. **Relative Index addressing mode:**

* If EA is obtained by adding 16-bit number of base register [SI] / [DI] along with displacement, then it is called Relative Base addressing mode
  + - * + EA = (SI) / (DI) + 8/16-bit displacement

Ex:- MOV AL,[SI+2]

**Relative BASE INDEX Addressing mode: -**

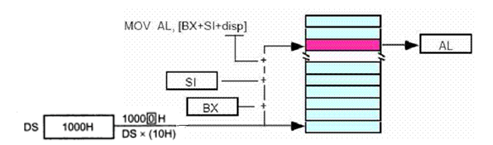
* If the 8-bit/ 16-bit data required for executing the instruction is present in the memory location, the 16- bit effective address of this memory location is obtained by adding the 16-bit number of the given base register [BX]/ [BP] , 16-bit number of the given index register [SI]/ [DI] along with the given 8/16-bit displacement.
  + - * + EA =(BX) / (BP) + (SI) / (DI) + 8/16-bit displacement

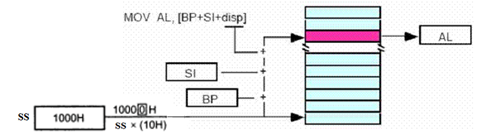
Ex: -

MOV AL, 0AH [BX] [SI]

MOV CX, 1234H [BP] [DI]

MOV BH, 05H [BP] [SI]





**I/O Port Addressing Modes: -**

* I/O devices can be interfaced to 8086 either in Memory mapped I/O or Peripheral mapped I/O.
* For Memory mapped I/O, memory addressing modes are used.
* For Peripheral mapped I/O or I/O mapped I/O, I/O port addressing modes are used.
* There are two types of I/O port addressing modes: Fixed Port type and Variable Port type.
* For the fixed port type, the 8-bit address of a port is specified directly in the instruction. This allows fixed access to ports numbered 0 to 255.

Ex: - IN AX, 50H; input a word from port 0050H to AX

IN AL, 3CH; input a byte from port 003CH to AL

OUT 50H, AX; output a word from AX to port 0050H

OUT 3CH, AL; output a byte from AL to port 003CH

* For the variable port type, the address of a port is loaded into DX register before the IN/ OUT instruction.
* Since DX is a 16-bit register, the Port address can be any number from 0000H to FFFFH. Therefore, up to 65536 ports are addressable in this mode.

Ex: - MOV DX, 0F020H; Initialize DX to point to Port

IN AL, DX; Input a byte from 8-bit port 0F020H to AL

IN AX, DX; Input a word from 16-bit port 0F020H to AX

MOV DX, 0F020H; Initialize DX to point to Port

OUT DX, AL; Output a byte from AL to 8-bit port 0F020H

OUT DX, AX; Output a word from AX to 16-bit port 0F020H

* The variable Port addressing has the advantage that the port address can be computed or dynamically determined by the program.

**Addressing modes of branching instructions**

1. **Intrasegment Branching Addressing mode**

* In Intrasegment branching, only the content in IP is changed. This addressing mode is further divided into two types:
  + 1. **Intrasegment Direct Addressing mode**: If the 16-bit offset address is given as operand along with the branching instruction and this 16-bit number is transferred into IP, then it is called Intrasegment direct addressing mode.

EX: -

CALL 2000H

JMP NEAR PTR UP

JMP 1200H

* + 1. **Intrasegment Indirect Addressing mode**: If the 16-bit register or memory is given as operand along with the branching instruction and this 16-bit number of this register or 16 bit content of the memory is transferred into IP, then it is called Intrasegment Indirect addressing mode.

EX: -

CALL BX ; Indirect within-segment

CALL WORD PTR [SI]

JMP AX ; INDIRECT NEAR JUMP

JMP WORD PTR [BX] ; INDIRECT NEAR JUMP

1. **Intersegment Branching Addressing mode**

In Intersegment branching, we have to change the EA in IP as well as Base address in CS. This addressing mode is further divided into two types:

* + 1. **Intersegment Direct Addressing mode**: If two 16- bit numbers are given as operand along with the branching instruction in which the 1st number is transferred into IP and 2nd number is transferred into CS, then it is called Intersegment Direct Addressing mode.

Ex: -

CALL FAR PTR FACTO ; Direct Call to another segment

JMP FAR PTR CONTINUE ; DIRECT FAR JUMP

JMP 2000:1200H

* + 1. **Intersegment Indirect Addressing mode:**  If 32- bit memory is given as operand along with the branching instruction, then 16 LSbits of memory data is transferred into IP and 16 MSbits of memory data is transferred into CS then such instructions are called Intersegment Indirect Addressing mode.

Ex: -

JMP DWORD PTR [BX] ; INDIRECT FAR JUMP

CALL DWORD PTR [SI] ; Indirect Call to another segment

**Classification of 8086 Instruction set**

Instructions in the instruction set of 8086 are classified into different types based on type of operation.

* Data transfer instructions
* Arithmetic instructions
* Bit manipulation instructions
* String instructions
* Program execution transfer instructions
* Process control instructions

Note :

1. Register can be any 16-bit general registers: AX, BX, CX, DX, SI, DI, SP and BP.
2. Register can be any 8-bit registers: AH, BH, CH, DH, AL, BL, CL, or DL.
3. Segment registers can be CS, DS, ES, or SS
4. Memory can be indicated by any one of the methods given in the memory addressing modes.
5. **Data Transfer Instructions**

|  |  |  |  |
| --- | --- | --- | --- |
| General – Purpose Byte or Word Transfer Instructions | Simple Input and Output Port Transfer Instructions | Special Address Transfer Instructions | Flag Transfer Instructions |
| MOV  PUSH  POP  XCHG  XLAT | IN  OUT | LEA  LDS  LES | LAHF  SAHF  PUSHF  POPF |

**MOV: - MOV Destination, Source**

* The MOV instruction Copies a byte or word from specified source to specified destination. The contents of the specified source will not be changed.
* No flags are affected by this instruction.
* Source and Destination in an instruction cannot both be memory locations.
* The type of the source and destination must be same.
* Only branching instruction can be used to store data into CS and IP registers, otherwise writing data in CS and IP is not permitted by any other instructions.
* The possible combination of source and destination is given in the following table:

|  |  |
| --- | --- |
| **Destination** | **Source** |
| Register | Register |
| Memory |
| Immediate Data |
| Segment Register |
| Memory | Register |
| Immediate Data |
| Segment Register |
| Segment. Reg except CS | Register |
| Memory |

Ex: - MOV CX, 1234H ; put the immediate number 1234H in CX

MOV AL, BL ; Copy contents of register BL to the AL

MOV DL, [0301H] ; Copy byte in DS at offset 0301H to the DL

MOV TOTAL [BP], AX; copy Ax to two memory locations AL to the first location,

; AH to the second. EA of the first memory location is the sum of ; the displacement represented by TOTAL and the contents of BP.

;Physical Address = EA + SS

MOV ES: TOTAL [BP], AX ; copy Ax to two memory locations- AL to the first   
 ; location, AH to the Second.

; EA of the first memory location is the sum of the   
 ; displacement represented by TOTAL and the contents   
 ; of BP.

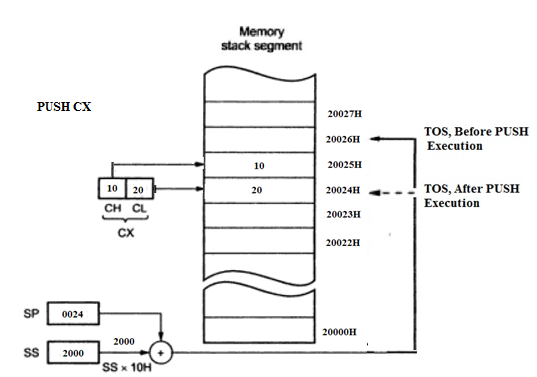
; Physical Address = EA + ES

;because of the segment override prefix ES.

**PUSH: - PUSH Source**

* The PUSH instruction decrements the stack pointer by 2 and copies a word from a specified source to the location in the stack segment where the stack pointer then points. The source of the word can be a 16-bit register, a segment register, or memory. For this instruction execution the content of SP is decrement by ‘2’.
* No flags are affected by this instruction.
* The contents of the specified source will not be changed.
* Whenever data are pushed into the stack, the first (most-significant) data byte moves to the stack segment memory location addressed by SP-1. The second (least-significant) data byte moves to the stack segment memory location addressed by SP-2. After the data are stored by a PUSH the contents of the SP register decrement by 2.

Ex: - PUSH CX ; decrement SP by 2, copy CX to stack



PUSH CS ; decrement SP by 2, copy CS to stack

PUSH AH ; ILLEGAL, must push only a word

PUSH [SI]

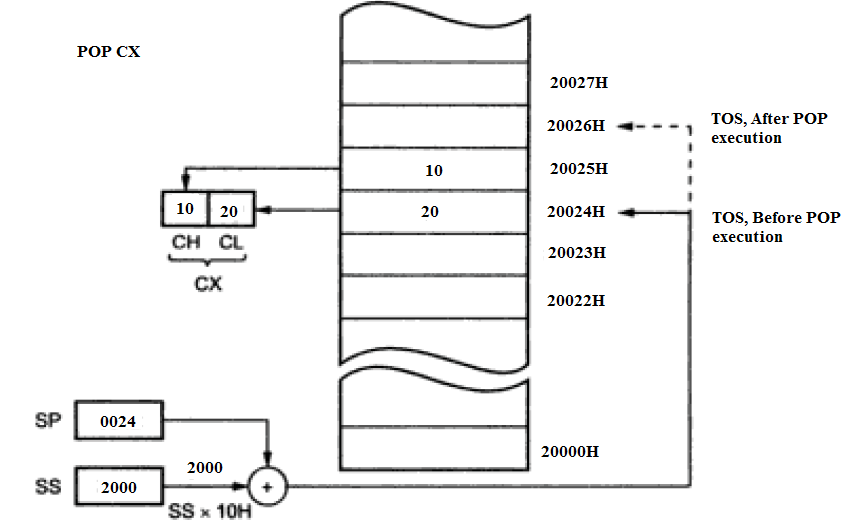
PUSH Result [BX] ; decrement SP by 2, copy word from memory in DS at

; EA = Result +[ BX] to stack

**POP**: **- POP Destination**

* The POP operation performs the inverse operation of a PUSH instruction. The POP operation Copies a word form top of stack to specified destination. Destination can be a general purpose register, segment register (except CS) or memory location. For this instruction execution the content of SP is increment by ‘2’.
* No flags are affected by this instruction.

Ex: - POP CX ; Copy a word from TOP of Stack to CX, increment SP by 2



POP DS ; Copy a word from TOP of Stack to DS, increment SP by 2.

POP [DI]

POP Result [BX] ; Copy a word from TOP of Stack to memory in DS with

; EA= Result + (BX) , increment SP by 2.

* When the POP instruction executes, the 1st byte of data removed from the stack memory location addressed by SP moves into register low byte. The 2nd byte is removed from stack segment memory location SP+1 and is placed into register higher byte. After both bytes are removed from the stack, the SP register is increment by ‘2’

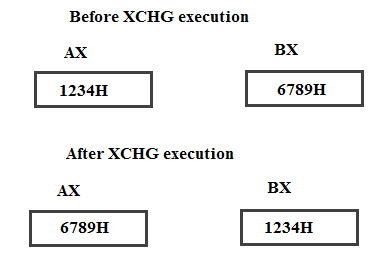
**XCHG: - XCHG Destination, Source**

This instruction exchanges the contents of Source and Destination..

* No flags are affected by this instruction.
* The type of the source and destination must be equal.
* The segment registers cannot be used in this instruction.
* The XCHG instruction cannot exchange memory to memory data.
* The possible combination of source and destination is given in the following table:

|  |  |
| --- | --- |
| **Destination** | **Source** |
| Register | Register |
| Memory | Register |
| Register | Memory |

Ex: - XCHG BX, AX; Exchange word in BX with word in AX



XCHG AL, DH ; Exchange byte in AL with byte in DH

XCHG DX, [5040H] ; Exchange word in DX with word in memory at EA = 5040H in DS.

**XLAT/XLATB:-**Translate a byte using LookUp table in the Memory

* This instruction replaces a byte in the AL register with a byte pointed to by (BX+AL) in a lookup table in Data segment memory.
* No flags are affected by this instruction.

**IN: - IN Accumulator, Port**

* Copy a byte or word from specified Input Port to Accumulator. If an 8-bit port is read, the data will go to AL. If a 16-bit port is read, the data will go to AX.
* No flags are affected by this instruction.
* The IN instruction has two possible formats, Fixed port format and variable port format.
* For the fixed port type, the 8-bit address of a port is specified directly in the instruction.

Ex: - IN AX, 50H ; input a word from port 0050H to AX

IN AL, 3CH ; input a byte from port 003CH to AL

* For the variable port type, the address of a port is loaded into DX register before the IN instruction.
* Since DX is a 16-bit register, the Port address can be any number from 0000H to FFFFH. Therefore, up to 65536 ports are addressable in this mode.

Ex:- MOV DX, 0F020H; Initialize DX to point to Port

IN AL, DX ; Input a byte from 8-bit port 0F020H to AL

IN AX, DX ; Input a word from 16-bit port 0F020H to AX

**OUT: - OUT Port, Accumulator**

* Copy a byte from AL or word from AX to specified Output Port.
* No flags are affected by this instruction.
* The OUT instruction has two possible formats, fixed port format and variable port format.
* For the fixed port type, the 8-bit address of a port is specified directly in the instruction.

Ex: - OUT 50H, AX ; output a word from AX to port 0050H

OUT 3CH, AL ; output a byte from AL to port 003CH

* For the variable port type, the address of a port is loaded into DX register before the OUT instruction.
* Since DX is a 16-bit register, the Port address can be any number from 0000H to FFFFH. Therefore, up to 65536 ports are addressable in this mode.

Ex:- MOV DX, 0F020H ; Initialize DX to point to Port

OUT DX, AL ; Output a byte from AL to 8-bit port 0F020H

OUT DX, AX ; Output a word from AX to 16-bit port 0F020H

**LEA: - Load effective address**: - **LEA Register, Source**

* This instruction copies the effective address of variable or memory location specified by the source into the given 16-bit register. (Only 16-bit register is used because EA will be of 16-bits only).
* No flags are affected by this instruction.

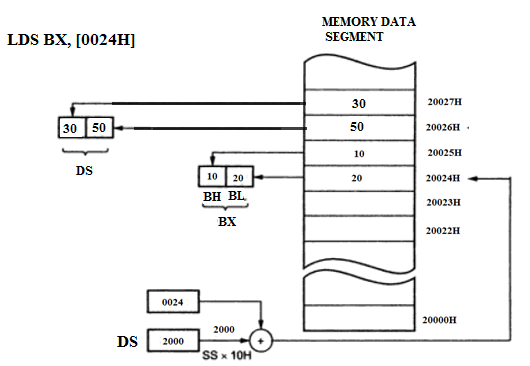
Ex: - LEA BX, LIST ; Load BX with offset of LIST in DS

LEA SI, [BX] ; Load SI with EA= (BX); this instruction executes as MOV SI, BX

**LDS**: **- LDS Register, Memory Address of first word**

* This instruction copies a word from two memory locations into the register specified in the instruction. It then copies a word from the next two memory locations into DS register.
* No flags are affected by this instruction.

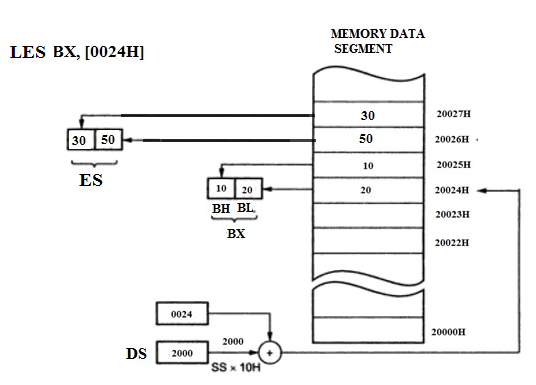
Ex: - LDS BX, [0024H];



**LES**: **- LES Register, Memory Address of first word**

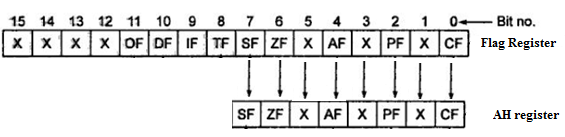
* This instruction copies a word from two memory locations into the register specified in the instruction. It then copies a word from the next two memory locations into ES register.
* No flags are affected by this instruction.

Ex: - LES BX, [0024H];



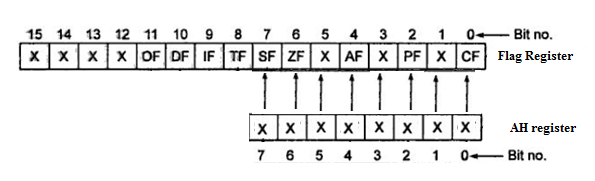
**LAHF: -**

* This instruction copies the lower byte of the 8086 flag register in to the AH register.
* No flags are affected by this instruction

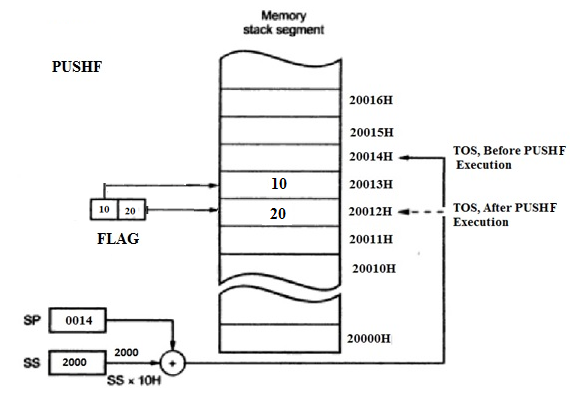


**SAHF: -**

* This instruction copies the contents of register AH into 8LSbits of the Flag register.
* SAHF changes the flags in the lower byte of the Flag register.

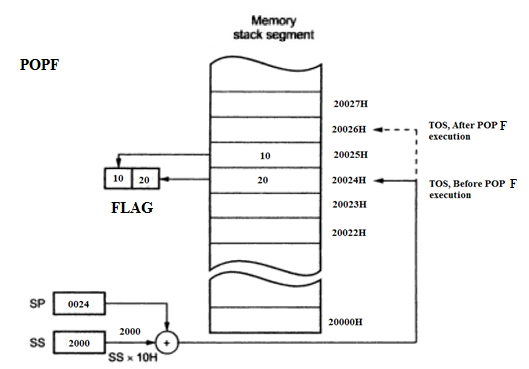
**  
  
PUSHF: -**

* The PUSHF instruction decrements the stack pointer by 2 and copies the word from Flag register to the memory location in the stack segment where the stack pointer then points.
* No flags are affected by this instruction



**POPF: -**

* This instruction copies a word from the two memory locations at the top of the stack to the Flag register and increments the Stack Pointer by 2.
* All Flags are modifies/ affected.



**2. Arithmetic Instructions**

These instructions are useful to perform Arithmetic calculations, such as addition, subtraction, multiplication, division, increment and decrement. They are again classified into four groups. They are:

|  |  |  |  |
| --- | --- | --- | --- |
| Addition Instructions | Subtraction Instructions | Multiplication Instructions | Division Instructions |
| ADD  ADC  INC  AAA  DAA | SUB  SBB  DEC  NEG  CMP  AAS  DAS | MUL  IMUL  AAM | DIV  IDIV  AAD  CBW  CWD |

**ADD: - ADD Destination, Source**

* This instruction adds the contents of source and destination and the result is stored in the same destination. The contents of the specified source will not be changed.
* In the instruction the source and destination must be of the same type.
* AF, CF, OF, PF, SF, ZF flags are affected/ modified.
* The possible combination of source and destination is given in the following table:

|  |  |
| --- | --- |
| **Destination** | **Source** |
| Register | Register |
| Immediate Data |
| Memory |
| Memory | Register |
| Immediate Data |

* The source and destination in an instruction cannot both be memory locations.

Ex: - ADD AL, 34H ; Add immediate number 34H to contents of AL, result stored in AL.

ADD DX, AX ; Add contents of DX to contents of AX, result stored in DX.

ADD AX, [BX] ; add word from memory at offset [BX] in DS to contents of AX.

**ADC: - ADD with Carry instruction: - ADC Destination, Source**

* This instruction adds the contents of source, destination and also status of the carry flag (status of CF prior to ADC instruction) and the result is stored in the same destination.
* The contents of the specified source will not be changed.
* This instruction is mainly used for multi-byte addition purpose.
* In the instruction the source and destination must be of the same type.
* AF, CF, OF, PF, SF, ZF flags are affected/ modified.
* The possible combination of source and destination is given in the following table:

|  |  |
| --- | --- |
| **Destination** | **Source** |
| Register | Register |
| Immediate Data |
| Memory |
| Memory | Register |
| Immediate Data |

* The source and destination in an instruction cannot both be memory locations.

Ex: - ADC AL, 34H ; Add immediate number 34H plus carry status to contents of AL

; Result stored in AL.( Addressing mode: Immediate AM)

ADC DX, AX ; Add contents of DX plus carry status to contents of AX, result stored ; in DX.( Addressing mode: Register AM)

ADC AX, [BX] ; add word from memory at offset [BX] in DS plus carry status to ;contents of AX.( Addressing mode: Register indirect AM)

**INC: - INC Destination**

* This instruction increments the contents of destination by one and stores the result in the same destination.
* Destination can be register or memory location.
* All flags **except CF** are modified.

Ex: - INC AX ;add 1 to contents of AX register.

INC BYTE PTR [SI] ; increment byte in DS at offset contained in SI.

;BYTE PTR Directive indicates to the assembler that the

;Byte from memory is to be incremented.

INC WORD PTR [2050H] ;increment the word at offset of [2050H] and [2051H] in the ;DS.

;WORD PTR Directive indicates to the assembler that the

;Word from memory is to be incremented.

**AAA (ASCII Adjust after Addition): -**

The data entered from the terminal is in ASCII format. In ASCII, 0 – 9 are represented by 30H – 39H.This instruction allows us to add the ASCII codes. After the addition, the AAA instruction is used to make sure the result is the correct **unpacked BCD**. This instruction does not have any operand. Similarly the following are the ASCII related instructions.

**Other ASCII Instructions:**

**AAS** (ASCII Adjust after Subtraction)

**AAM** (BCD Adjust after Multiplication)

**AAD** (BCD to Binary convert Before Division)

* The AAA instruction works only on the AL register.
* Only AF and CF are modified.

Ex:- ; assume AL=36H, ASCII 6

; assume BH= 38H, ASCII 8

ADD AL, BH ; result AL= 6EH, which is incorrect BCD

AAA ; Now AL=00000100, unpacked BCD 4

; CF= 1 indicates answer is 14 decimal

**DAA (Decimal Adjust AL after BCD Addition): -**

* It is used to make sure that the result of adding two packed BCD numbers is adjusted to correct packed BCD number.
* It only works on AL register.
* This Instruction is belongs to Implied addressing mode.
* Updates AF, CF, PF and ZF.

Ex:- ; assume AL=58 BCD = 0101 1000

; assume BH= 36 BCD = 0011 0110

ADD AL, BH ; AL=1000 1110 =8EH, not valid BCD result

DAA ; AL=1001 0100 = 94 BCD

**SUB: - SUB Destination, Source**

* This instruction subtracts the source contents from the destination contents and the result is stored in the same destination. The contents of the specified source will not be changed.
* In the instruction the source and destination must be of the same type.
* AF, CF, OF, PF, SF, ZF flags are affected/ modified.
* The possible combination of source and destination is given in the following table:

|  |  |
| --- | --- |
| **Destination** | **Source** |
| Register | Register |
| Immediate Data |
| Memory |
| Memory | Register |
| Immediate Data |

* The source and destination in an instruction cannot both be memory locations.
* For subtraction, CF functions as borrow flag.

Ex; - SUB AL, 74H ; Subtract immediate number 74H from AL

SUB DX, AX ; DX-AX, result in DX

SUB AX, [BX] ; subtract word at effective address [BX] from AX

**SBB: -Subtract with Borrow: - SBB Destination, Source**

* This instruction subtracts the source contents and the contents of CF from the destination contents and the result is stored in the same destination.
* The contents of the specified source will not be changed.
* In the instruction the source and destination must be of the same type.
* SBB instruction allows to subtract two multi-byte numbers because any borrow produced by subtracting least significant byte is included in the result when SBB instruction executes.
* AF, CF, OF, PF, SF, ZF flags are affected/ modified.
* The possible combination of source and destination is given in the following table:

|  |  |
| --- | --- |
| **Destination** | **Source** |
| Register | Register |
| Immediate Data |
| Memory |
| Memory | Register |
| Immediate Data |

* The source and destination in an instruction cannot both be memory locations.
* For subtraction, CF functions as borrow flag.

Ex: - SBB AL, 74H

SBB DX, AX ; perform DX-AX-CF operation, Result in DX

SBB AX, [BX]

**DEC: - DEC Destination**

* This instruction decrements the contents of destination by one and stores the result in the same destination.
* Destination can be register or memory location.
* All flags **except CF** are modified.

Ex: - DEC AX ; subtract 1 to contents of AX register.

DEC BYTE PTR [SI] ; decrement byte in DS at offset contained in SI.

; BYTE PTR Directive indicates to the assembler that the

; Byte from memory is to be decremented.

DEC WORD PTR [2050H] ; decrement the word at offset of [2050H] and [2051H] in the ;DS.

; WORD PTR Directive indicates to the assembler that the

; Word from memory is to be decremented.

**DAS (Decimal Adjust after BCD Subtraction): -**

* It is used to make sure that the result of subtracting two packed BCD numbers is adjusted to a correct packed BCD number.
* It only works on AL register.
* This Instruction is belongs to Implied addressing mode.
* Updates AF, CF, SF, PF and ZF. OF is undefined.

Ex:- ; assume AL=86 BCD = 1000 0110

; assume BH= 57 BCD = 0101 0111

SUB AL, BH ; AL=0010 1111 =2FH, not valid BCD result

DAS ; AL=0010 1001 = 29 BCD

; assume AL=97 BCD = 1001 0111

; assume BH= 28 BCD = 0010 1000

SUB AL, BH ; AL=0110 1111 =6FH, not valid BCD result

DAS ; AL=0110 1001 = 69 BCD

**NEG: - NEG Destination**

* This instruction replaces the number in a destination with the 2’s complement of that number.
* This instruction forms the 2’s complement by subtracting the original word or byte from zero.
* Destination can be a register or a memory location.
* NEG instruction is useful for changing the sign of a signed word or byte
* NEG instruction updates AF, CF, SF, PF, ZF and OF.

.

Ex: - NEG AX ; replace the number in AX with its 2’s complement

NEG BYTE PTR [1250H] ; replace byte at offset [1250H] in DS with its 2’s complement

NEG WORD PTR [BX] ; replace word at offset [BX] in DS with its 2’s complement

**CMP: - CMP Destination, Source**

* This instruction compares the contents of source and destination by subtracting the source contents from the destination contents.
* The contents of source and destination are not changed, but the flags are set to indicate the results of the comparison.
* AF, CF, OF, PF, SF, ZF flags are affected/ modified.
* The possible combination of source and destination is given in the following table:

|  |  |
| --- | --- |
| **Destination** | **Source** |
| Register | Register |
| Immediate Data |
| Memory |
| Memory | Register |
| Immediate Data |

* The source and destination in an instruction cannot both be memory locations.
* In the instruction the source and destination must be of the same type.
* For the instruction CMP AX, BX, we will get the following values of flags

|  |  |  |  |
| --- | --- | --- | --- |
|  | CF | ZF | SF |
| AX = BX | 0 | 1 | 0 |
| AX > BX | 0 | 0 | 0 |
| AX < BX | 1 | 0 | 1 |

Ex:- CMP AX, BX

CMP BH, AL

CMP AX, [2050H]

CMP BX, 3050H

**MUL: - MUL Source**

* It is an unsigned multiplication instruction.
* Microprocessor 8086 will multiply the equal length of data, i.e. 8\*8 or 16\*16 bit multiplication.
* It multiplies two bytes to produce a word or two words to produce a double word.
* This instruction assumes one of the operand in AL or AX.
* Source can be a register or memory location.
* For two bytes multiplication the result size is a word (16-bit) that is by default stored in the AX register.
* Similarly if two words are multiplied the result size is 32-bit that is by default stored in the combination of DX and AX registers.
* The contents of the specified source will not be changed.
* If the most significant byte of a 16-bit result or the most significant word of a 32-bit result is 0, CF and OF will both be 0’s.
* AF, PF, SF and ZF are undefined after a MUL instruction.

Ex;- MUL BL ; AL times BL, result in AX

MUL BYTE PTR [2050H]

MUL CX

**IMUL: -IMUL Source**

* It is a signed multiplication instruction. The conditions for this are similar to the MUL instruction.
* If the magnitude of product does not require all the bits of the destination, the unused bits will be filled with copies of the sign bit.
* If the upper byte of 16-bit result or the upper word of a 32-bit result contains only copies of the sign bits then CF and OF will both be 0.
* If the upper byte of 16-bit result or the upper word of a 32-bit result contains part of the product then CF and OF will both be 1.

Ex;- IMUL BL ; AL times BL, result in AX

IMUL BYTE PTR [2050H] ;

IMUL CX

**AAM** (BCD Adjust after Multiplication): -

* After the two unpacked BCD digits are multiplied, the AAM instruction is used to adjust the product to two unpacked BCD digits in AX.
* AAM works only after the multiplication of two unpacked BCD bytes
* It works only on an operand in AL.
* AAM updates PF, SF and ZF.
* AF, CF and OF are left undefined.

EX: - ; assume AL= 00000111 = unpacked BCD 7

; assume BL= 00000101 = unpacked BCD 5

MUL BL ; AL x BL; result in AX

; AX = 00000000 00100011 =0023H

AAM ; AX =00000011 00000101 = 0305H

; Which is unpacked BCD for 35

**DIV: - DIV Source**

* It is an unsigned division instruction.
* This instruction is used to divide an unsigned word by a byte or to divide an unsigned double word (32 bits) by a word.
* Source can be a register or a memory location which will be the divisor for the Division operation.
* For a word divided by a byte operation, the word must be in the AX register. After the division, AL will contain in 8- bit Quotient and AH will contain an 8-bit remainder.(result)
* If the result quotient is too large to fit in AL the 8086 will automatically do a type 0 interrupt.
* When a doubleword is divided by a word, the most significant word must be in DX and AX must contain least significant word of the Doubleword.
* After the division, AX will contain in 16-bit Quotient and DX will contain an 16-bit remainder (result).
* If the result quotient is too large to fit in AX the 8086 will automatically do a type 0 interrupt.
* The contents of the specified source will not be changed.

Ex: - DIV BH ; AX/BH, AL=quotient, AH=remainder

DIV BYTE PTR [2050H]

DIV BX ; divide doubleword in DX and AX by word in CX.

DIV WORD PTR [SI]

**IDIV: - IDIV Source**

* It is a signed division instruction.
* The conditions for this are similar to the DIV instruction.
* The sign of the remainder is same as the sign of the dividend.

**CBW (Convert Byte to Word):** This instruction converts signed byte in AL to signed word in AX. The conversion is done by extending the sign bit of AL throughout AH. This Instruction is belongs to Implied addressing mode.

**CWD (Convert Word to Double Word):** This instruction converts signed word in AX to signed double word in DX: AX. The conversion is done by extending the sign bit of AX throughout DX. This Instruction also belongs to implied addressing mode.

**AAD (BCD to Binary Convert before Division): -**

* AAD converts two unpacked BCD digits in AH and AL to the equivalent binary number in AL.
* PF, SF, and ZF are updated. AF, CF and OF are undefined after AAD.

EX: - ; AX= 0607H unpacked BCD for 67 decimal

AAD ; result: AX = 0043 =43H =67 decimal

**3. Bit manipulation Instructions**

|  |  |  |
| --- | --- | --- |
| Logical Instructions | Shift Instructions | Rotate Instructions |
| NOT  AND  OR  XOR  TEST | SHL  SAL SHR  SAR | ROL  ROR  RCL  RCR |

* Except NOT, all the Logical operations always clear the Carry and Overflow flags, while the other flags change to reflect the condition of the result. AF will be undefined.

**NOT: - NOT Destination**

* This instruction inverts each bit of the byte or word at the specified destination. i.e. it forms the 1’s complement of the destination and stores the result at the same destination
* The operand can be a register or memory location.
* No flags are affected by the NOT instruction.

Ex: - NOT AX ; complement contents of AX register

NOT BYTE PTR [SI] ; Complement memory byte at offset [SI] in Data segment

NOT BH

NOT WORD PTR [BX]

**AND: - AND Destination, Source**

* This instruction logically AND each bit in a source byte or word with the same number bit in a destination byte or word.
* The result is stored in the same destination. The contents of the specified source will not be changed.
* A bit ANDed with a 0 will be cleared, and ANDed with a 1 will not change. So, we can use the AND instruction to selectively clear or not clear bits in an operand.
* Mainly used for masking of bits operation. The task of clearing a bit in a binary number is called masking.
* CF and OF become zero after the operation. PF, SF and ZF are updated. AF is undefined.
* The possible combination of source and destination is given in the following table:

|  |  |
| --- | --- |
| **Destination** | **Source** |
| Register | Register |
| Immediate Data |
| Memory |
| Memory | Register |
| Immediate Data |

* The source and destination in an instruction cannot both be memory locations.
* In the instruction the source and destination must be of the same type.

Ex: - AND AX, BX ; AND word in AX with word in BX, result in AX

AND AX, [3050H]

AND AX, FF00H ; FF00H: Masks lower byte and leaves upper byte unchanged

AND [3040H], 1250H

**OR: - OR Destination, Source**

* This instruction logically OR each bit in a source byte or word with the same number bit in a destination byte or word.
* The result is stored in the same destination. The contents of the specified source will not be changed.
* CF and OF become zero after the operation. PF, SF and ZF are updated. AF is undefined.
* The possible combination of source and destination is given in the following table:

|  |  |
| --- | --- |
| **Destination** | **Source** |
| Register | Register |
| Immediate Data |
| Memory |
| Memory | Register |
| Immediate Data |

* The source and destination in an instruction cannot both be memory locations.
* In the instruction the source and destination must be of the same type.
* A bit ORed with a 1 will be set and ORed with a 0 will not change. So, we can use the OR instruction to selectively set or not set bits in an operand.

Ex: - OR AX, BX ; AX ORed with BX, result in AX. BX not changed

OR AX, [3050H] ;

OR AX, 1200H ; AX ORed with immediate 1200H

OR [3040H], 1250H ;

**XOR: - XOR Destination, Source**

* This instruction logically Exclusive-OR each bit in a source byte or word with the same number bit in a destination byte or word.
* The result is stored in the same destination. The contents of the specified source will not be changed.
* CF and OF become zero after the operation. PF, SF and ZF are updated. AF is undefined.
* The possible combination of source and destination is given in the following table:

|  |  |
| --- | --- |
| **Destination** | **Source** |
| Register | Register |
| Immediate Data |
| Memory |
| Memory | Register |
| Immediate Data |

* The source and destination in an instruction cannot both be memory locations.
* In the instruction the source and destination must be of the same type.
* A bit Exclusive-ORed with a 1 will be inverted, Exclusive-ORed with a 0 will not change. So, we can use the XOR instruction to selectively invert or not invert bits in an operand.
* A common use of the XOR instruction is to clear register to zero.

Ex: - XOR AX, AX ; AX register will be cleared.

XOR BX, [3050H]

XOR CX, 1200H

XOR [3040H], 1250H

**TEST: - TEST Destination, Source**

* The TEST instruction performs the AND operation. The difference is that AND instruction changes the destination operand, whereas the TEST instruction does not.
* A TEST only affects the condition of the flag register which indicate the result of the TEST.
* TEST instruction is often used to set flags before a conditional jump instruction.
* Usually the Test instruction is followed by either the JZ or JNZ instruction.
* The destination operand is normally tested against immediate data. The value of immediate data is 01H to test the Least Significant bit position, 02H to test the next bit, 04H for the next and so on.
* CF and OF become zero after the operation. PF, SF and ZF are updated. AF is undefined.
* The possible combination of source and destination is given in the following table:

|  |  |
| --- | --- |
| **Destination** | **Source** |
| Register | Register |
| Immediate Data |
| Memory |
| Memory | Register |
| Immediate Data |

* The source and destination in an instruction cannot both be memory locations.
* In the instruction the source and destination must be of the same type.

Ex: - TEST AL, CH ; AND CH with AL, no result stored. Update PF, SF, ZF

TEST AX, 0004H ; to test bit D2 of AX register

**Shift Instructions: -**

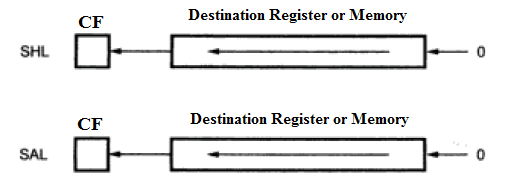
**(Shift logical Left) SHL Destination, Count**

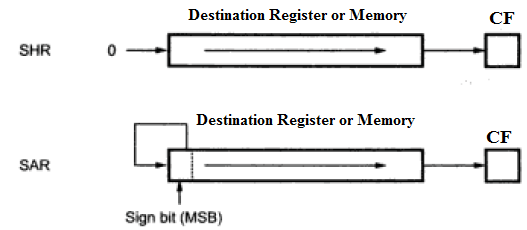
**(Shift logical Right) SHR Destination, Count**

**(Shift Arithmetic Left) SAL Destination, Count**

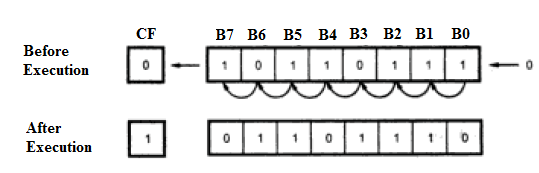
**(Shift Arithmetic Right) SAR Destination, Count**

* Shift and Rotate instructions manipulate binary numbers at the binary bit level.
* The microprocessor’s instruction set contains four different shift instructions. Two are logical shifts and two are arithmetic shifts.
* Logical shift operations function with unsigned numbers and arithmetic shifts function with signed numbers.
* The logical shifts move a ‘0’ into the right most bit position for a logical Left Shift and a ‘0’ into the left most bit position for a logical Right Shift.
* The Arithmetic shift left and Logical left shift are identical.
* The Arithmetic right shift and Logical right shift are different because the Arithmetic right shift copies the sign bit through the number, whereas the Logical right shift copies a ‘0’ through the number.
* A Shift left always multiplies by 2 for each bit position shifted and a Shift Right always divides by 2 for each position shifted. i.e. shifting a number two places multiplies or divides by 4.
* If the desired number of shifts is one, this can be specified by putting a ‘1’ in the count position of the instruction.
* For shifts of more than 1 bit position, the desired number of shifts is loaded into the CL register and CL is placed in the count position of the instruction.
* Flags are affected as follows: CF contains the bit most recently shifted out. For the count of one, OF will be a 1 if the two MSb’s (i.e. MSbit before shift and MSbit after shift) are not the same. For multiple shifts, OF is meaningless. SF and ZF will be updated to show the condition of the destination. AF is undefined. PF will have meaning only for an 8- bit destination.





Ex: - SAL AX, 1 ;Shift word in AX 1 bit position left, 0 in LSB.



MOV CL, 02 ; load desired number of shifts in CL

SAL BX, CL

SHL AX, CL

SAR BYTE PTR [BX], CL

SHR WORD PTE [SI], CL

**Rotate Instructions: -**

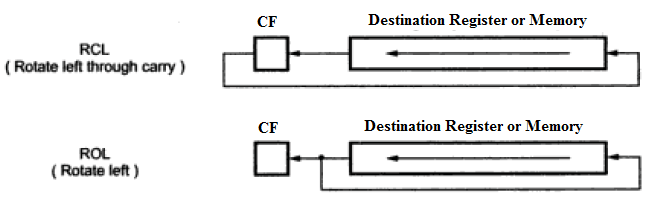
* Rotate instruction position binary data by rotating the information in a register or memory location either from one end to another or through the carry flag.
* Rotate instruction can be thought of as circular.

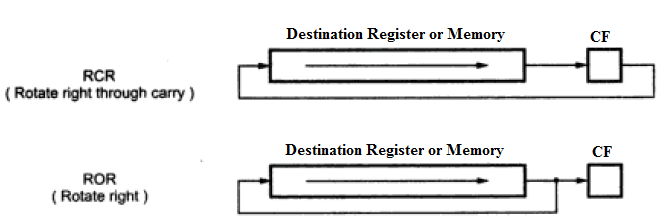
**(Rotate Operand Left through CF) RCL Destination, Count**

**(Rotate Operand Right through CF) RCR Destination, Count**

**(Rotate Operand Left) ROL Destination, Count**

**(Rotate Operand Right) ROR Destination, Count**

****

****

* The ‘C’ in the middle of the mnemonic should help to remember that CF is in the rotated loop and help to distinguish RCL instruction from the ROL instruction.
* Destination can be a Register or a memory location.
* To rotate the operand one bit position, this can be specified by putting a ‘1’ in the count position of the instruction.
* To rotate more than 1 bit position, the desired number is loaded into the CL register and CL is placed in the count position of the instruction.
* The rotate instruction affects only CF and OF. After rotate, CF will contain the bit most recently rotated out. OF will be a 1 after single bit rotation if the MSb was changed by the rotate. OF is undefined after multibit rotate.
* ROL/ ROR instruction can be used to swap the nibbles in a byte or to swap the bytes in a word.

Ex: - RCL AX, 1 ; rotate word in AX 1 bit position left, MSb to CF, CF to LSb

ROL AX, 1 ; rotate word in AX 1 bit position left, MSb to LSb and CF

RCR AX, 1 ;

MOV CL, 04 ; Load CL for rotating 4 bit positions.

ROR SI, CL ;

RCL BX, CL

**4. String Instructions**

|  |
| --- |
| String Instructions |
| MOVS/MOVSB/MOVSW  LODS/LODSB/LODSW STOS/STOSB/STOSW  CMPS/CMPSB/CMPSW  SCAS/SCASB/SCASW  REP  REPE/PEPZ  REPNE/REPNZ |

* A string is a series of bytes or words stored in successive memory locations.
* The string instructions will automatically increment or decrement the pointers after each operation, depending on the state of the direction flag DF.
* If the direction flag is cleared with a CLD instruction, then the pointers in SI and DI will automatically be incremented after each string operation.
* If the direction flag is set with a STD instruction, then the pointers in SI and DI will automatically be decremented after each string operation.

Note: only the actual registers used by the string instructions increment or decrement.

**MOVS/MOVSB/MOVSW: - Move string byte or string word**

* One of the more useful string data transfer instructions is MOVS because it transfers data from one memory location to another. This is the only memory – to – memory transfer allowed in the 8086 microprocessor.
* The MOVS instruction transfers a byte or word from the memory location in data segment addressed by SI to the extra segment location addressed by DI.
* After the byte or word is transferred, SI and DI are automatically adjusted to point to next source and next destination depending on DF.
* IF DF = 0, then SI and DI are incremented by 1 for byte transfer or by 2 for word transfer.
* IF DF = 1, then SI and DI are decremented by 1 for byte transfer or by 2 for word transfer.
* This instruction affects no flags.

EX: -

LEA SI, SOURCE

LEA DI, DEST

CLD

MOVSB ; transfers a byte from DS pointed by SI to ES location

; pointed by DI and SI =SI+1, DI= DI+1.

**LODS/LODSB/LODSW: - Load string byte into AL or load string word into AX**

* This instruction loads AL or AX with data stored at the location in data segment pointed by the SI register.
* After the copy, SI content will be automatically adjusted to point to next source depending on DF.
* IF DF = 0, then SI is incremented by 1 for byte transfer or by 2 for word transfer.
* IF DF = 1, then SI is decremented by 1 for byte transfer or by 2 for word transfer.
* This instruction affects no flags.

EX: -

LEA SI, SOURCE

CLD

LODSB ; copy byte from string to AL and SI incremented by 1.

**STOS/STOSB/STOSW:- Store byte or word in string from AL or AX**

* This instruction stores or copies a byte from AL or a word from AX to a memory location in the extra segment pointed to by DI register.
* After the copy, DI content will be automatically adjusted to point to next destination depending on DF.
* IF DF = 0, then DI is incremented by 1 for byte transfer or by 2 for word transfer.
* IF DF = 1, then DI is decremented by 1 for byte transfer or by 2 for word transfer.
* This instruction affects no flags.

EX: -

LEA SI, SOURCE

CLD

STOSW ; copy a word from AX to extra segment pointed by DI

; and DI incremented by 2.

**CMPS/CMPSB/CMPSW: - Compare string byte or string words**

* This instruction is used to compare a bye or word of a string in data segment pointed by SI with a byte or word of another string in Extra segment pointed by DI.
* The comparison is done by subtracting the byte or word pointed to by DI from the byte or a word pointed to by SI.
* After the comparison, SI and DI will automatically be incremented or decremented to point to the next elements in the two strings depending on DF.
* All the flags are modified by the instruction but neither operand is affected.
* The CMPS instruction can be used with a REPE or REPNE prefix to compare all the elements of a string.

EX: - LEA SI, SOURCE

LEA DI, DEST

CLD

CMPSW

**SCAS/SCASB/SCASW: - Scan a string byte or a string word**

* This instruction is used to compare a byte in AL or a word in AX with a byte or word pointed to by DI in Extra segment memory.
* The comparison is done by subtracting string element (byte or word) addressed by DI from the content of AL or AX.
* All the flags are modified by the instruction but neither operand is affected.

EX: - MOV AL, 41H

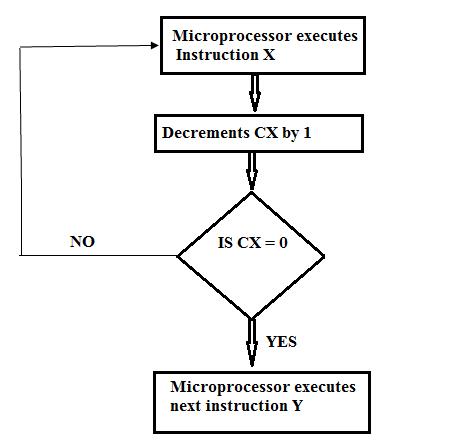
LEA DI, DEST

CLD

SCASB

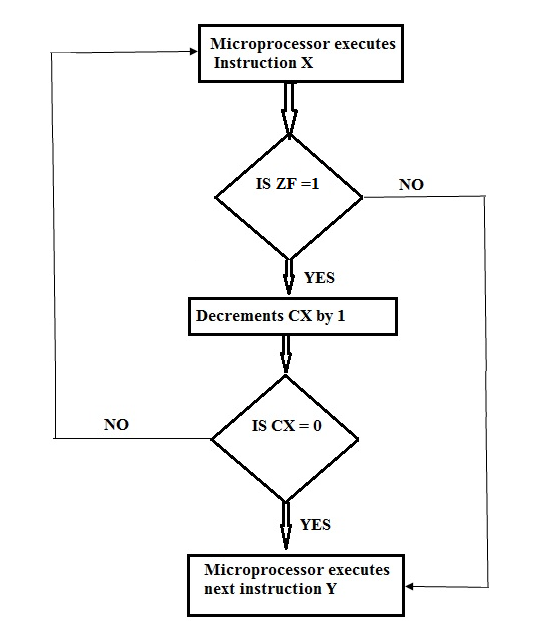
**REP: - (Unconditional Repeat)**

* REP is a prefix which is added to any string data transfer instruction except the LODS instruction.
* The REP prefix causes CX to decrement by 1 each time the string instruction executes.
* After CX decrements, the string instruction repeats if CX ≠ 0.
* If the CX reaches a value of 0, the instruction terminates and the program continues with the next sequential instruction.

****

**REPE/ REPZ: - (Repeat if equal / Repeat if result is Zero)**

* REPE/ REPZ is a conditional repeat prefix instruction, which is often used with the compare string instruction or with the scan string instruction.
* The instruction is repeatedly executed only when ZF=1 and up to counter value in CX is reset.

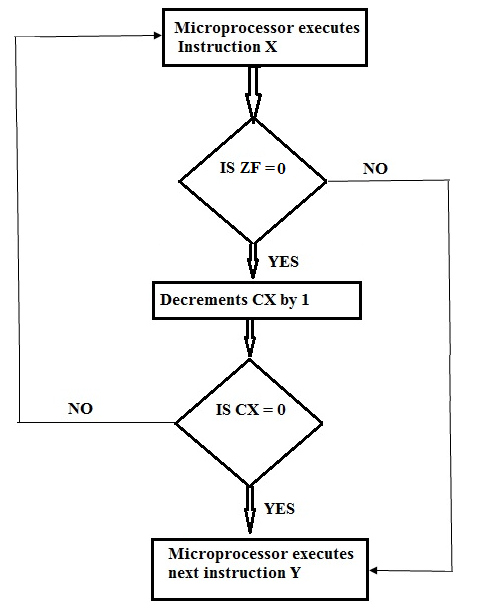


EX: - REPE CMPSB ; compare string bytes until end of string

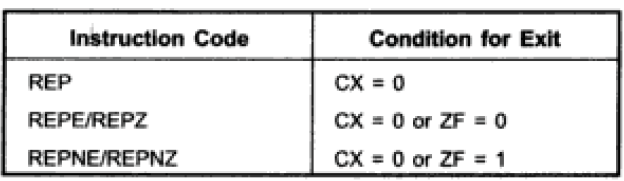
; or until string bytes not equal

**REPNE/ REPNZ: - (Repeat if not equal / Repeat if result is not Zero)**

* REPNE/ REPNZ is a conditional repeat prefix instruction, which is often used with the with the scan string instruction.
* The instruction is repeatedly executed only when ZF= 0 and up to counter value in CX is reset.



**Summary of REP instructions**



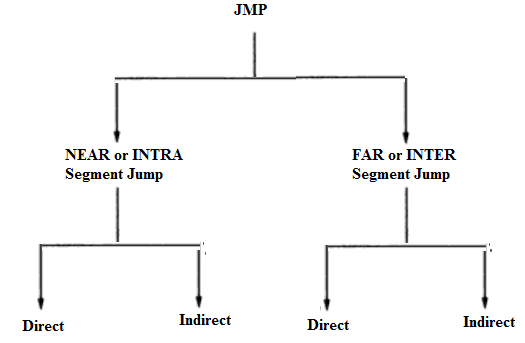
**5. Program Execution Transfer Instructions**

* These instructions are used to tell the 8086 to start fetching instructions from some new address, rather than continuing in sequence

|  |  |  |  |
| --- | --- | --- | --- |
| Unconditional Transfer Instructions | conditional Transfer Instructions | Iteration Control Instructions | Interrupt Instructions |
| CALL  RET  JMP | JA/JNBE  JAE/JNB  JB/JNAE  JBE/JNA  JC  JE/JZ  JG/JNLE  JGE/JNL  JL/JNGE  JLE/JNG  JNC  JNE/JNZ  JNO  JNP/JPO  JNS  JO  JP/JPE  JS | LOOP  LOOPE/LOOPZ  LOOPNE/LOOPNZ  JCXZ | INT  INTO  IRET |

**JMP: - (Unconditional Jump to specified destination) JMP Operand**

* This instruction will always cause the 8086 to fetch its next instruction from the location specified in the instruction rather from the next sequential location
* When the 8086 executes a JMP instruction, it loads a new number into the Instruction pointer register and in some cases it also loads a new number into the code segment register.
* If the JMP destination is in the same code segment, the 8086 only has to change the contents of the IP. This type of JUMP is referred to as a NEAR or INTRASEGMENT Jump.
* If the JMP destination is in a code segment which has a different name from the segment in which the JMP instruction is located, the 8086 has to change the contents of both CS and IP to make the JMP. This type of JMP is referred to as a FAR or INTERSEGMENT Jump.
* NEAR and FAR Jumps are further described as either Direct or Indirect.
* If the Destination address for the Jump is specified directly as part of the instruction, then the Jump is specified as Direct.
* If the Destination address for the Jump is contained in a register or memory location, the Jump is referred to as indirect, because the 8086 has to go to the specified register or memory location to get the required destination address.



* A special case of the Direct NEAR – type Jump instruction is the Direct SHORT- type Jump.
* A SHORT Jump is a 2-byte instruction that allows Jumps or branches to memory locations within -128 to +127 bytes from the address following the Jump.
* A 3- byte NEAR Jump allows a branch or jump within -32768 to +32767 bytes (i.e. to any memory location in the current code segment) from the current instruction pointer location.
* The JMP instruction affects no flags.

EX: -

JMP 2000H ; DIRECT NEAR JUMP

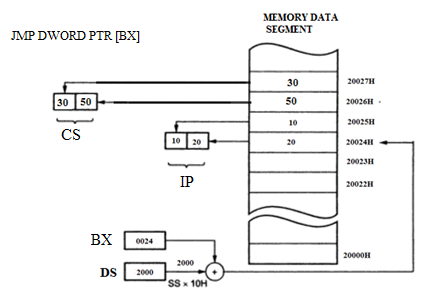
JMP AX ; INDIRECT NEAR JUMP

JMP WORD PTR [BX] ; INDIRECT NEAR JUMP

JMP NEAR PTR NEXT ; DIRECT NEAR JUMP

JMP FAR PTR CONTINUE ; DIRECT FAR JUMP

JMP DWORD PTR [BX] ; INDIRECT FAR JUMP

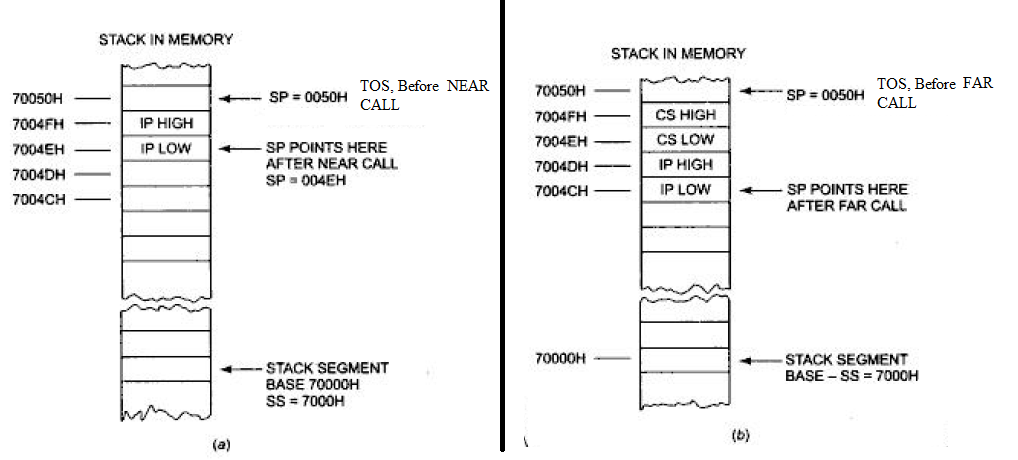


**CALL: -** (**Unconditional CALL) CALL Operand**

* The CALL instruction is used to transfer execution from the main program to a Subprogram or Procedure.
* If the main program and the subprogram are present in the same code memory segment, then it is called intrasegment branching in which only IP content is changed. This type of CALL is NEAR CALL.
* If the main program and the subprogram are present in the different code memory segments, then it is called intersegment branching in which both CS and IP contents are changed. This type of CALL is FAR CALL.
* The 8086 CALL instruction performs two operations when it executes

1. First it stores the address of the instruction after the CALL instruction on the stack. This address is called return address because it is the address that execution will return to after the procedure execution completes.
2. When the 8086 executes a NEAR CALL instruction, it decrements the stack pointer by 2 and copies the offset of the next instruction after the CALL onto the stack
3. When the 8086 executes a FAR CALL instruction, it decrements the stack pointer by 2 and copies the contents of CS register to the stack, again decrements the stack pointer by 2 and copies the offset of the next instruction after the CALL onto the stack.
4. The second operation of the CALL instruction is to change the contents of the IP and in some cases, the contents of the CS register to contain the starting address of the procedure.

* CALL instruction affects no flags.



EX: -

CALL 2000H ; Direct Near Call or Direct Intrasegment Call

CALL NEAR PTR FACTO ; Direct within-segment Call (Near or Intrasegment)

CALL BX ; Indirect within-segment Call (Near or Intrasegment)

CALL WORD PTR [BX] ; Indirect within-segment Call (Near or Intrasegment)

CALL DWORD PTR [BX] ; Indirect Call to another segment (Far or Intersegment)

CALL FAR PTR FACTO ; Direct Call to another segment (Far or Intersegment)

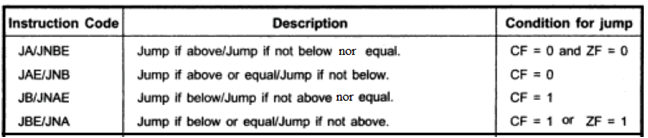
**RET: -**

* The RET instruction will return execution from a procedure to the next instruction after the CALL instruction which was used to call the procedure.
* If the CALL is NEAR, then the RET instruction at the end of the subprogram will return execution to the instruction after the CALL by copying the offset saved on the stack to IP.
* If the CALL is FAR, then the RET instruction at the end of the subprogram will return execution to the instruction after the CALL by restoring the saved values of CS and IP from the stack.
* The RET instruction affects no flags.
* A RET instruction can be followed by a number, for example RET 4. In this case the stack pointer will be incremented by an additional four addresses after the IP or the IP and CS are popped off the stack.

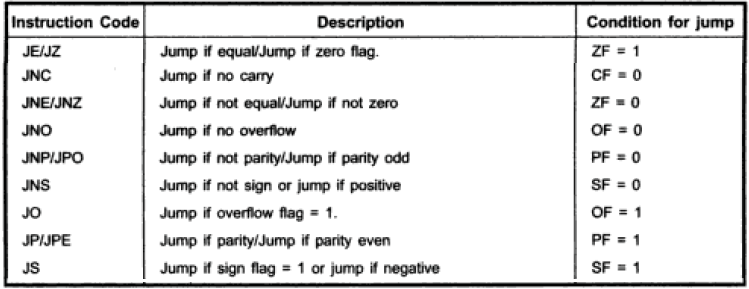
**8086 Conditional Jump Instructions**

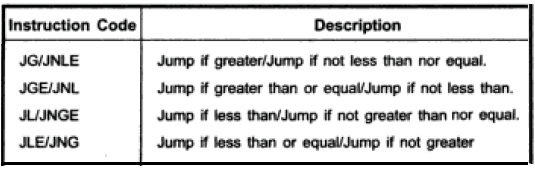
* In the 8086 the six conditional flags indicate the conditions that are present after an instruction.
* The 8086 conditional jump instructions look at the state of a specified flags to determine whether the jump should be made or not.
* All the conditional jumps are Short-type Jumps. This means that the destination label must be in the same code segment as the Jump instruction and also, the destination address must be in the range of -128 bytes to +127 bytes from the address of the instruction after the jump instruction.
* If the condition given in the instruction is satisfied, then new 16-bit numbers are transferred into IP. Hence, microprocessor will read the next instruction code from this new address.
* If the condition given in the instruction is not satisfied, then new 16-bit number is not transferred into IP. So microprocessor will not branch, instead microprocessor will read the next instruction code in sequence.

The different conditional jump instructions are given below:



* Note that the terms above and below are used when we are working with unsigned binary numbers.
* The terms greater (more positive) and less are used when we are working with signed binary numbers.





After comparison of signed numbers, we will get the following values of flags, depending upon the sign of the result.

|  |  |  |
| --- | --- | --- |
| **OF** | **SF** | **Sign of the Result** |
| 0 | 0 | + ve result or Zero Result |
| 0 | 1 | -ve result |
| 1 | 0 | -ve result |
| 1 | 1 | + ve result or Zero Result |

**JG / JNLE:** Condition is result should be positive

1. OF=SF=0 and ZF=0
2. OF=SF=1 and ZF=0 ((OF XOR SF) OR ZF ) = 0

**JGE / JNL:** Condition is result should be positive or Zero

1. OF=SF=0
2. OF=SF=1 (OF XOR SF) =0

**JL / JNGE:** Condition is result should be negative.

1. OF=0 and SF=1
2. OF=1 and SF=0 (OF XOR SF) =1

**JNG / JLE:** Condition is result should be negative or Zero.

1. OF=0 and SF=1
2. OF=1 and SF=0
3. ZF=1 ((OF XOR SF) OR ZF ) = 1

**IRET: - Interrupt Return**

* The IRET instruction is used at the end of the interrupt service procedure to return execution to the interrupted program.
* This is done by restoring the saved values of IP, CS and flags from the stack to IP, CS and Flag register.
* The RET instruction should not normally be used to return from interrupt procedure because it does not copy the flags from the stack back to the flag register.

**INT: - INT Type**

* The 8086 INT instruction can be used to cause the 8086 to do any one of the 256 possible interrupt types. The desired interrupt type is specified as part of the instruction.
* The address of the procedure is taken from the memory whose address is four times the type number.
* In other word the IP value for any interrupt type is always at an address of 4 times the interrupt type, and the CS value is at a location two addresses higher.

**INTO Instruction** :

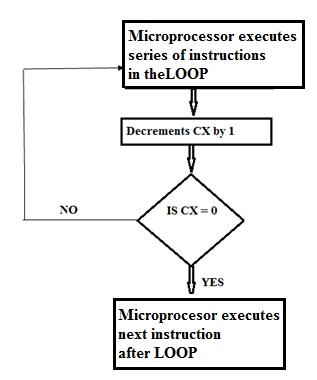
* If the overflow flag is set, this instruction will cause the 8086 to do an indirect far call to a procedure written to handle overflow condition. To do the call 8086 will read a new value for IP from address 00010H and a new value of CS from address 00012H.

**LOOP Related Instructions: -**

* If the group of instructions to be executed more than once, then it can be done by using the Loop instruction.
* The number of times the instruction sequence is to be repeated is loaded into CX register.
* The LOOP instructions are basically Conditional Jump instructions which have the format LOOP Label.
* The LOOP instructions can do only SHORT Jumps. This means that the destination address for the Jump must be in the range of -128 bytes to +127 bytes from the instruction after the LOOP instruction.
* LOOP instructions affect no flags.
* LOOP instructions combine two operations in each instruction
  + 1st operation is to decrement the CX register by ‘1’
  + 2nd operation is to check the CX register and in some cases also the Zero flag to decide whether to do a jump to the specified Label.

**LOOP: - LOOP LABEL**

* Each time the LOOP instruction executes, CX is automatically decremented by 1.
* If CX ≠0 execution will jump to a destination specified by a label in the instruction.
* If CX=0 after the auto decrement, execution will simply go on to the next instruction after LOOP.



EX: - LEA SI, SOURCE

MOV CX, 05H

MOV AL, 00H

UP: ADD AL, [SI]

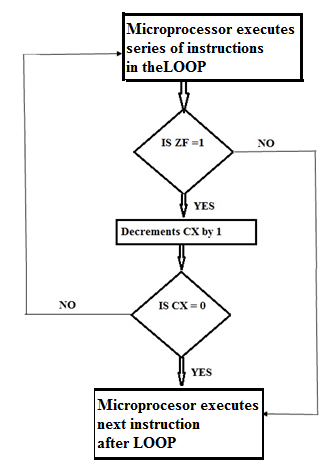
ADC AH, 00H

INC SI

LOOP UP

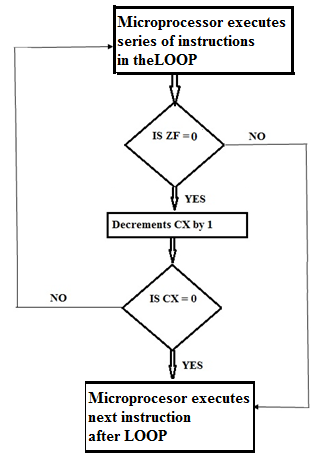
**LOOPE/LOOPZ: - LOOP if equal / Loop if result is zero**

* Each time the LOOP instruction executes, CX is automatically decremented by 1.
* If CX ≠0 ad ZF =1, execution will jump to destination specified by a label in the instruction.
* In other words, program execution will exit from the repeat loop if CX has been decremented to Zero or the Zero flag is not set.

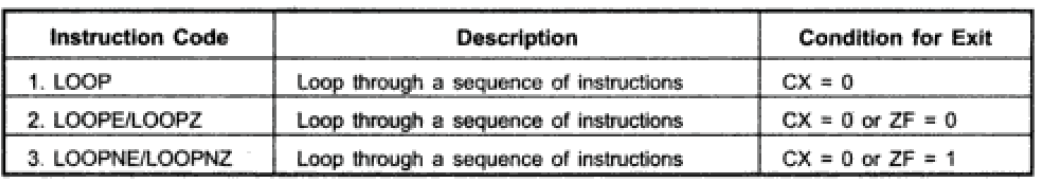


**LOOPNE/LOOPNZ: - LOOP if not equal / Loop if result is not zero**

* Each time the LOOP instruction executes, CX is automatically decremented by 1.
* If CX ≠0 ad ZF =0, execution will jump to destination specified by a label in the instruction.
* In other words, program execution will exit from the repeat loop if CX has been decremented to Zero or the Zero flag is set.



**Summary of LOOP**



**JCXZ : – JCXZ Operand ( Jump if the CX register is Zero)**

* The instruction will cause a jump to a label given in the instruction if the CX register contain all 0’s.
* Note that this instruction does not look at the Zero Flag when it decides whether to jump or not.
* The destination label for the Jump must be in the range of -128 bytes to +127 bytes from the instruction after the JCXZ instruction.
* This instruction affects no flags.

**6. Process control instructions**

|  |  |  |
| --- | --- | --- |
| Flag set/clear Instructions | External Hardware Synchronization instructions | No operation Instruction |
| STC  CLC  CMC  STD  CLD  STI  CLI | HLT  WAIT  ESC  LOCK | NOP |

**STC Instruction :**

This instruction sets the carry flag, STC does not affect any other flag.

**CLC Instruction :**

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

**CMC Instruction :**

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

**STD Instruction :**

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

**CLD Instruction :**

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

**STI Instruction :**

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

**CLI Instruction :**

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

**NOP instruction :**

At the time of execution of NOP instruction, no operation is performed except fetch and decode. It takes three clock cycles to execute the instruction. NOP instruction does not affect any flag. This instruction is used to fill in time delays or to delete and insert instructions in the program while trouble shooting.

**HLT Instruction :**

The HLT instruction will cause the 8086 to stop fetching and executing instructions. The 8086 will enter a halt state. The only ways to get the processor out of the halt state are with an interrupt signal on the INTR pin, an interrupt signal on the NMI pin, or a reset signal on the RESET input.

**WAIT Instruction :**

When this instruction executes, the 8086 enters an idle condition where it is doing no processing. The 8086 will stay in this idle state until a signal is asserted on the 8086 input pin, or until a valid interrupt signal is received on the INTR or the NMI interrupt input pins. If a valid interrupt occurs while the 8086 is in this idle state, the 8086 will return to the idle state after the execution of interrupt service procedure. WAIT affects no flags. The WAIT instruction is used to synchronize the 8086 with external hardware such as the 8087 math coprocessor.

**LOCK instruction :**

The LOCK prefix allows a microprocessor to make sure that another processor does not take control of the system bus while it is in the middle of a critical instruction which uses the system bus. The LOCK prefix is put in front of the critical instruction. When an instruction with a LOCK prefix executes, the 8086 will assert its bus lock signal output. This signal is connected to an external bus controller device which then prevents any other processor from taking over the system bus. LOCK affects no flags.

**8086 Interrupts**

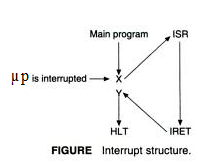
**Definition:** The meaning of ‘interrupts’ is to break the sequence of operation.

When microprocessor executes any program, then it is called main program. If in between the main program the microprocessor (µp) is interrupted then the microprocessor will branch from the main program to the subprogram. This subprogram is called interrupt service routine (ISR). After executing this ISR, the IRET instruction comes and the microprocessor returns back from the ISR to the main program from where microprocessor has left. Interrupts of the microprocessor are categorized basically into two types:

I. Software interrupt, and

2. Hardware interrupt.

When microprocessor is interrupted by giving instruction INT n then it is called software interrupt. When microprocessor is interrupted by giving signal on hardware interrupt pin, then it is called hardware interrupt



**Need for Interrupt:**

* Interrupts are particularly useful when interfacing I/O devices that provide or require data at relatively low data transfer rate.
* Interrupts are commonly used technique for computer multitasking, especially in real time computing. Such a system is said to be an interrupt driven system.

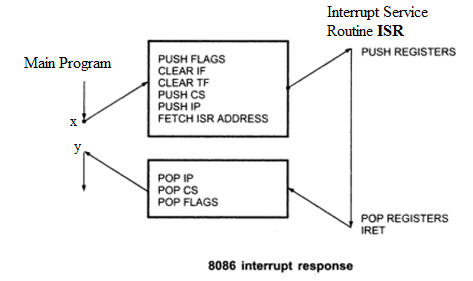
An 8086 interrupt can come from any one of the 3 sources.

* One source is an external signal applied to the non-maskable interrupt (NMI) input pin or to the interrupt (INTR) input pin. An interrupt caused by a signal applied to one of these input pins is referred to as hardware interrupt.
* A second source of an interrupt is execution of the interrupt instruction INT n. This is referred to as software interrupt.
* The third source of an interrupt is some error condition produced in the 8086 by the execution of an instruction. Example to this is divide- by- zero interrupt.

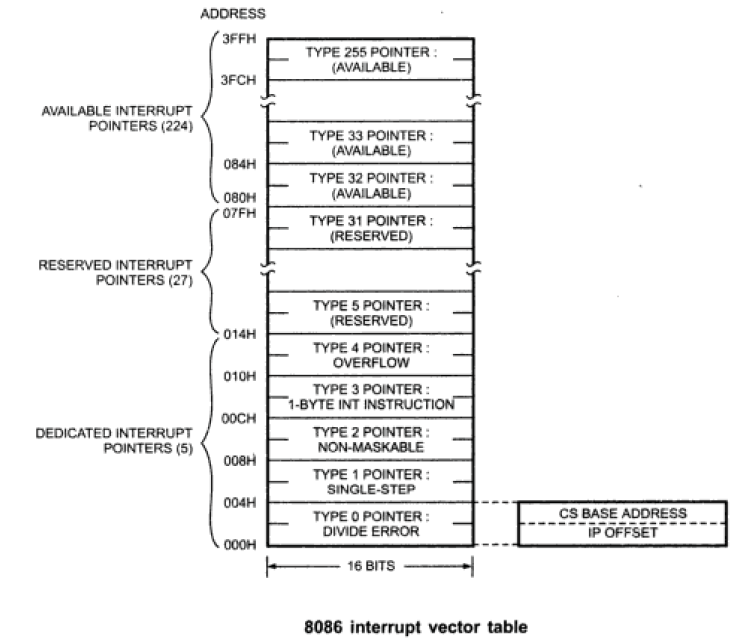
**Interrupt response**

At the end of each instruction cycle, the 8086 checks to see if any interrupts have been requested. If an interrupt has been requested, the 8086 response to the interrupt is given by the following steps

1. It decrements the stack pointer by 2 and pushes the flag register on the stack
2. It disables the 8086 INTR interrupt input by clearing the interrupt flag IF.
3. It resets the trap flag TF. So ISR is not executed in single stepping mode.
4. It decrements the stack pointer by 2 and pushes the current CS register contents on the stack.
5. It decrements the stack pointer by 2 and pushes the current IP contents on the stack.
6. It does an indirect far jump to the start of the ISR written for that interrupt.



* When the 8086 responds to an interrupt, it goes to four memory locations to get the CS and IP values for the start of the ISR. The starting address of these four memory locations is obtained using equation 4 \* n. (where n is the Interrupt type number)
* In an 8086 system, the first 1K byte of memory, from 00000H to 003FFH, is set aside as a table for storing the starting addresses of ISR.
* If this 1K memory is RAM, then the values of EA and BA can be changed. Hence, using the same INT n instruction, the microprocessor can execute different ISR stored in different memory locations, so in 8086, starting address 00000H will always be in RAM memory.
* The starting address of an interrupt-service routine ISR is often called as the interrupt vector or interrupt pointer, so the starting 1K memory locations table is referred to as the **interrupt-vector table** or the **interrupt-pointer table**.



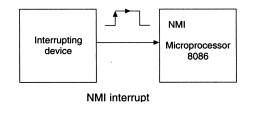
**HARDWARE INTERRUPTS OF MICROPROCESSOR 8086:**

There are two hardware interrupt pins in the microprocessor 8086.

1. NMI (Non-maskable interrupt) 2. INTR (Interrupt request)

**NMI**

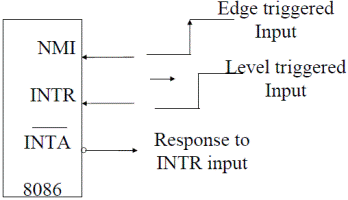
Whenever NMI is activated, INT 2 interrupt occurs, because NMI interrupt is internally decoded. The NMI is an edge-triggered input that requests an interrupt on positive edge, i.e. on 0 to 1 transition. After a positive edge, the NMI pin must remain logic 1 until it is recognized by the microprocessor. It is also important to note that before the positive edge is recognized, the NMI pin must be logic 0 for at least of two clocking periods.



* The NMI is often used for the parity errors and the other major system faults such as power failures.
* When the power supply is made OFF, then the interrupt signal is applied on the NMI, so the microprocessor branches from the main program to the ISR. This ISR is normally used to transfer data from non-battery back-up RAM to battery back-up RAM.

**INTR**

* INTR is a maskable hardware interrupt. The interrupt can be enabled/disabled using STI/CLI instructions.
* The INTR is level sensitive, which means that it must be held at logic 1 level until it is recognized.
* The signal is an acknowledge pin which is used for acknowledge, when the microprocessor receives the interrupt signal on the INTR.
* The INTR input must be externally decoded to select a call address. Any interrupt call address can be chosen for the INTR pin, but generally uses an interrupt type between 20H and FFH.
* When an interrupt occurs, the processor stores FLAGS register into stack, disables further interrupts, fetches from the bus one byte representing interrupt type, and jumps to interrupt processing routine address of which is stored in location 4 \* <interrupt type>. Interrupt processing routine should end with the IRET instruction.

**[](http://www.8085projects.info/image.asp?picture=Hardware%20Interrupts%20of%208086.gif)**

**(ii) Software Interrupts** (Internal Interrupts and Instructions)

Software interrupts can be caused by:

INT n instruction - any one interrupt from available 256 interrupts.

* When the microprocessor executes the INT n instruction, then the microprocessor branches from the main program to the ISR. The base address and effective address of the ISR are transferred into the CS and IP from four memory locations. The physical address of 1st memory location is obtained by using equation 4 \*n.

For example:

For INT 0, n = 0 hence, physical address of first memory location = 4 \*n = 4 \* 0 = 00000h.

* From the first two memory locations, new EA is transferred into IP, and from the last two memories, new BA is transferred into CS. Hence, four memory locations are required for each INT n instructions, so for 256 INT n instructions total memory locations required = 256 \* 4 = 1024 = 1K. So, starting 1K-memory location from 00000h to 003FFh is reserved for storing EA and BA of ISR for INT n instruction.
* Software interrupt processing is the same as for the hardware interrupts.

**Classification of INT n Instruction**

1. INT 0 to INT 4, are used for fixed operation in the 8086 microprocessor, hence these five interrupt instructions are called dedicated interrupts.

2. INT 5 to INT 31 are reserved for future Intel microprocessor ICs.

3. INT 32 to INT 255 can be used by the programmer.

**Dedicated Interrupts of Microprocessor 8086:**

1. **INT** **0** (Reserved for DIV/IDIV instruction): When the microprocessor performs the division of the signed number (IDIV) or unsigned numbers (DIV), then the quotient will be maximum 8/16 bits. If the quotient obtained after DIV/IDIV instruction is greater than 8/16 bits, then the microprocessor itself executes the INT 0 instruction. So, the microprocessor branches from the main program to the ISR of the INT 0. This ISR is used to display the division error message on the CRT screen. It also occurs when an attempt is made to divide by zero.
2. **INT 1** (Reserved for single stepping mode): To detect the error in a program (debugging), the program can be executed in a single stepping mode by making TF = 1. If TF is made 1, then after executing each instruction, the microprocessor itself executes INT 1 instruction. So, the microprocessor branches from the main program to the ISR of INT 1. This ISR is used to display the result of different registers of the microprocessor on the CRT screen. So, after each instruction programmer can verify the result and in this way error can be detected.
3. **INT 2** (Reserved for hardware interrupt NMI): When the positive edge and level signal are applied on the hardware interrupt pin NMI (non-maskable interrupt), then the microprocessor is interrupted. So, the microprocessor branches from the main program to the ISR. For branching from the main program to the ISR, the microprocessor will use INT 2 instruction, i.e. the values of EA and BA of the ISR are obtained from four successive memory locations starting from 4\*2 = 00008h

When the power supply is made OFF, then the interrupt signal is applied on the NMI. So, the microprocessor branches from the main program to the ISR. This ISR is normally used to transfer data from non-battery back-up RAM to battery back-up RAM.

1. **INT 3** (Reserved for break point technique): If single stepping mode is used to detect error in a program, then it will take more time, because after each instruction, the microprocessor will be interrupted and the result is displayed.

Using the break point technique, the time required to detect the error in a program is reduced. In this method by giving INT 3 instruction in between, we can break the program, i.e. the microprocessor branches from the main program to the ISR and the results of different registers are displayed on the CRT screen. So, there is no need to verify the result after each instruction, instead of this result can be verified after executing the group of instruction to be made.

1. INT 4 (Reserved for INTO instruction): The INTO (interrupt on overflow) is a conditional interrupt instruction in which the microprocessor will check the value of overflow flag (OF).

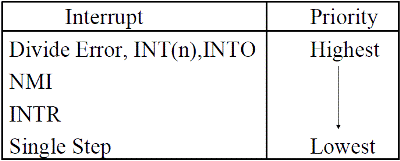
I. If OF = 0, then the microprocessor is not interrupted, instead, the microprocessor will execute the next instruction in sequence.

2. If OF= 1, then the microprocessor is interrupted, i.e. the microprocessor branches from the main program to the ISR using the INTO instruction. This ISR is used to display overflow flag.

**DIFFERENCES BETWEEN SOFTWARE AND HARDWARE INTERRUPT**

|  |  |
| --- | --- |
| Software interrupt | Hardware interrupt |
| 1. It is synchronous event  2. This interrupt is requested by executing interrupt instruction  3.It cannot be ignored or masked  4. It has highest priority among all interrupts. | 1. It is asynchronous event  2. This interrupt is requested by external device  3. It can be masked  4. The priority is lower than that of software interrupt. |

**Interrupt Priorities**

[](http://www.8085projects.info/image.asp?picture=Interrupt%20Priority%20Structure%20of%208086.gif)

1. Write an ALP to find the largest word from the 100 words present in the memory from address 76000h and store the result in register BX.

Solution:

MOV AX, 7600h

MOV DS, AX

MOV SI, 0000h

MOV CX, 0064h

MOV BX, [SI]

DEC CX

L2: ADD SI, 0002h   
CMP BX, [SI]

JA LI

MOV BX, [SI]

LI: LOOP L2

HLT

1. Write an ALP to find the largest signed word (most positive word) from the 100 signed binary numbers present in the memory from address 76000h in two's complement form. Store the result in register BX.

Solution:

MOV AX, 7600h

MOV DS, AX

MOV SI, 0000h

MOV CX, 0064h

MOV BX, [SI]

DEC CX

L2: ADD SI, 0002h   
CMP BX, [SI]

JG LI

MOV BX, [SI]

LI: LOOP L2

HLT

1. Write an ALP in the 8086 to convert the 8-bit binary number available in the memory location 2000:0100 into the BCD Store the result in the next memory location.

Solution:

MOV AX, 2000h

MOV DS, AX

MOV SI, 0100h

MOV AL, [SI]

AAM

MOV CL, 04

ROL AH, CL

OR AL, AH

INC SI

MOV [SI], AL

HLT

1. Write an ALP to convert the hexadecimal number present in the memory location HEX\_NUM to an ASCII character. Memory location HEX\_NUM contains a single digit (four MSBs are zero). Store the ASCII character in the memory location ASCII\_NUM.

Solution:

.MODEL SMALL

.DATA

ASCII\_NUM DB **04H**

.CODE

MOV AX, @DATA ; data for initialization of DS

MOV DS, AX

MOV SI, OFFSET HEX\_NUM

MOV AL, [SI]

CMP AL, 0Ah

JC LESS

ADD AL, 37 ; add 37 for letter (ASCII of 0A to 0F is 41h to 46h)

MOV ASCII\_NUM, AL

HLT

LESS: ADD AL, 30 ; add 30 for ASCII (ASCII of 00 to 09 is 30h to 39h)

MOV ASCII\_NUM, AL

HLT

END

1. Write an ALP to convert the ASCII number present in the memory location 2000: 2000h into the hexadecimal. Store the hexadecimal result in the next memory location.

Solution:

MOV AX, 2000h

MOV DS, AX

MOV SI, 2000h

MOV AL, [SI] ; transfer ASCII number from memory to AL

SUB AL, 30h

CMP AL, 09h

JG CHAR

JMP HEX

CHAR: SUB AL, 07h ; if letter from 0A to 0F, then subtract 07 to convert into HEX

HEX: INC SI ; point to the next address for storing HEX result

MOV [S1], AL

HLT