**EC8691 MICROPROCESSORANDMICROCONTROLLER L T PC 3 0 0 3**

**OBJECTIVES:**

**The student should be made to:**

- Study the Architecture of 8086microprocessor.
- Learn the design aspects of I/O and Memory Interfacingcircuits.
- Study about communication and businterfacing.
- Study the Architecture of 8051microcontroller.

**UNIT I THE 8086 MICROPROCESSOR 9**

Introduction to 8086 – Microprocessor architecture – Addressing modes - Instruction set and assembler directives – Assembly language programming – Modular Programming - Linking and Relocation – Stacks - Procedures – Macros – Interrupts and interrupt service routines – Byte and String Manipulation.

**UNIT II 8086 SYSTEM BUS STRUCTURE 9**

8086 signals – Basic configurations – System bus timing –System design using 8086 – IO programming–Introduction to Multiprogramming – System Bus Structure – Multiprocessor configurations – Coprocessor, Closely coupled and loosely Coupled configurations
– Introduction to advancedprocessors.

**UNIT III I/O INTERFACING**

Memory Interfacing and I/O interfacing - Parallel communication interface – Serial communication interface – D/A and A/D Interface - Timer – Keyboard /display controller – Interrupt controller – DMA controller – Programming and applications Case studies: Traffic Light control, LED display , LCD display, Keyboard display interface and Alarm Controller.

**UNIT IV MICROCONTROLLER 9**

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

**UNIT V INTERFACING MICROCONTROLLER 9**

Programming 8051 Timers - Serial Port Programming - Interrupts Programming – LCD & Keyboard Interfacing - ADC, DAC & Sensor Interfacing - External Memory Interface- Stepper Motor and Waveform generation, generation- Comparison of Microprocessor, Microcontroller, PIC and ARM processors

**OUTCOMES:**

**At the end of the course, the student should be able to:**

- Design and implement programs on 8086microprocessor.
- Design I/Ocircuits.
- Design Memory Interfacingcircuits.
- Design and implement 8051 microcontroller based systems.

**TEXT BOOKS:**

1.Yu-Cheng Liu, Glenn A.Gibson, "Microcomputer Systems: The 8086 / 8088 Family - Architecture, Programming and Design", Second Edition, Prentice Hall of India,2007

2. Mohamed Ali Mazidi, Janice Gillispie Mazidi, Rolin McKinlay, "The 8051Microcontroller and Embedded Systems: Using Assembly and C", Second Edition, Pearson Education,2011

**REFERENCE:**

1. Doughlas V.Hall, "Microprocessors and Interfacing, Programming and Hardware:,TMH,2012

## UNIT – I

## THE 8086 MICROPROCESSOR

### 1.1 INTRODUCTION

A microprocessor is a computer processor which incorporates the functions of a computer's central processing unit (CPU) on a single integrated circuit (IC), or at most a few integrated circuits. The microprocessor is a multipurpose, clock driven, register based, programmable electronic device which accepts digital or binary data as input, processes it according to instructions stored in its memory, and provides results as output. Microprocessors contain both combinational logic and sequential digital logic.

- Microprocessors operate on numbers and symbols represented in the binary numeral system Microprocessor is the controlling unit or CPU of a micro-computer, fabricated on a very small chip capable or performing ALU operations and communicating with the external word connected to it. It forms a micro-computer when combined with memory and Input/outputdevices.
- Microprocessors of different word size with varying decrease of capabilities are available. Microprocessor comprises of all the functional components of the central processing unit of a general purpose computer. In other words , functionally it is equivalent to aCPU.
- ***Cost :*** The most important characteristics of a microcomputer is its low cost. Because of the widespread use of microprocessors, the volume of production is very high. That is why, microprocessor chips are available at fairly lowprices.
- ***Size*** : The second important features of a microprocessor is its small size. As a result of improvement in fabrication technology, VLSI, electronic circuitry has become so dense that a minute silicon chip can contain hundred and thousands of transistors constituting the microprocessor. Its size does not exceed a few inches on any side , even in the packaged form.
- ***Power Consumption:*** The another important characteristics is its low power consumption microprocessors are normally manufactured by Metal-Oxide semiconductortechnology.
- ***Versatility:*** The versatility of a microprocessor results from its stored program mode of operation. Keeping the same basic hardware , a microprocessor-based system can be configured for a number of applications simplify altering the software program. This also makes it veryflexible.
- ***Reliability:***AnotherimportantpropertyofVLSIdeviceswhichhasalsobeeninherniated by microprocessors is extreme reliability . It has been established that the failure rate of an IC is fairly uniform at the package level, regardless of its complexity.

**History of Microprocessor**

Evolution of microprocessor can be viewed as five kinds of generation.

## First Generation: 4-bitMicroprocessors

- The first microprocessor was introduced in 1971 by Intel Corp. It was named Intel 4004 as it was a 4 bitprocessor.
- It was a processor on a singlechip.
- It could perform simple arithmetic and logic operations such as addition, subtraction, boolean AND and boolean OR.
- It had a control unit capable of performing control functions like fetching an instruction from memory, decoding it, and generating control pulses to executeit.
- It was able to operate on 4 bits of data at a time.
- Intel introduced the enhanced version of 4004, the 4040. Some other 4 bit processors are International's PPS4 and Thoshiba'sT3472.

## Second Generation :8-bitMicroprocessors

- The first 8 bitmicroprocessorwhich could perform arithmetic and logic operations on 8 bit words was introduced in 1973 again byIntel.
- This was Intel 8008 and was later followed by an improved version, Intel 8088 ids more powerful and faster 8 bit NMOSmicroprocessor.
- Some other 8 bit processors are Zilog-80 and MotorolaM6800.

## Third Generation:16-bitMicroprocessors

- The 8-bit processors were followed by 16 bit processors. They are Intel 8086 and80286.
- Some other 16 bit processors are Zilog-z8000 and MotorolaM68000,68010.

## Fourth Generation: 32-bitMicroprocessors

- The 32 bit microprocessors were introduced by several companies but the most popular one is Intel 80386,Intel386,
- Some other 32 bit processors arePentium pro,Pentium with MMX,Pentium II
- Motorola,IBM and Apple jointly developed 32-bit RISC processors Power pc 601,603,604 and 620.

## Fifth Generation: 64-bitMicroprocessors

- The 64 bit microprocessors were introduced by several companies but the most popular is Intel i860 -64 bit RISC processor.
- Some other 32 bit processors are SUN's SPARC and ULTRASPARC, Power PC 620,Alpha 21064

| microprocessor | Year | Number oftransistors |
|---|---|---|
| 4004 | 1971 | 2250 |
| 8008 | 1972 | 2500 |
| 8080 | 1974 | 5000 |
| 8086 | 1978 | 29000 |
| I286 | 1982 | 12000 |
| I386 | 1985 | 275000 |
| I486 | 1989 | 1,18000 |
| Pentium | 1993 | 3,100,000 |
| Pentium2 | 1997 | 7,500,000 |
| Pentium3 | 1999 | 24,000,000 |
| Pentium4 | 2000 | 42,000,000 |

**Table 1.1 History of processor**

**1.2 MicroprocessorArchitecture:**

The 8086 CPU is divided into two independent functional parts,

the Bus interface unit (BIU)

and execution unit (EU).

**The Bus Interface Unit** contains

- Bus InterfaceLogic
- Segmentregisters
- Memory addressinglogic
- Six-byte queue.

1. The BIU Sends OutAddress,
2. Fetches The Instructions from Memory,
3. Read Data from Ports and Memory
4. Writes The Data to Ports andMemory.

**The execution unit:** contains

- The Data and AddressRegisters,
- The Arithmetic and LogicUnit,
- The Control Unit andFlags.

1. The control circuitry which directs internaloperations.
2. A decoder in the EU translates instructions fetched from memory into a series of actions which the EU carriesout.
3. The EU is has a 16-bit ALU which can add, subtract, AND, OR, XOR, increment, decrement, complement or shift binarynumbers.
4. The EU is decoding an instruction or executing an instruction which does not require use of thebuses.

In other words, the BIU handles all transfers of data and addresses on the buses for the execution unit.

**The Queue**: The BIU fetches up to 6 instruction bytes for the following instructions. The BIU stores these prefetched bytes in first-in-first-out register set called a queue. When the EU is ready for its next instruction it simply reads the instruction byte(s) for the instruction from the queue in the BIU. This is much faster than sending out an address to the system memory and waiting for memory to send back the next instruction byte or bytes.

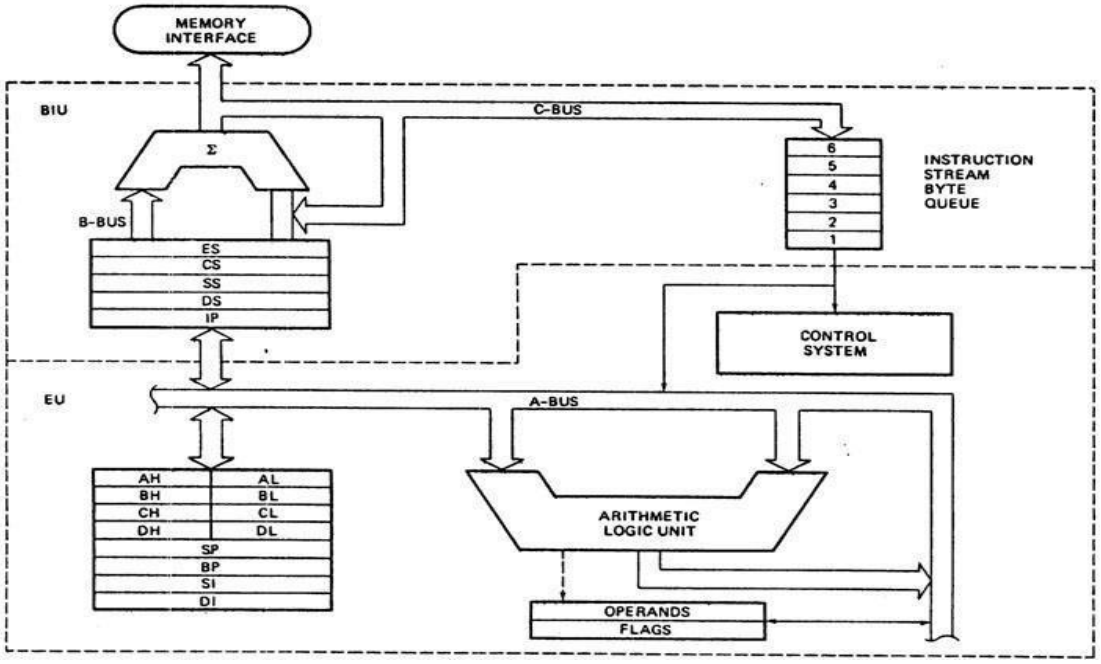


**Fig.1.2 8086 Architecture [Source: Advanced Microprocessors and Microcontrollers by A.K Ray & K.M. Bhurchandi]**

**Word Read:** Each of 1 MB memory address of 8086 represents a byte wide location.16-bit words will be stored in two consecutive memory locations. If first byte of the data is stored at an even address**,** 8086 can read the entire word in one operation.

For example, if the 16-bit data is stored at even address 00520H is 9634H

**MOV BX, [00520H]**

8086 reads the first byte and stores the data in BL and reads the 2nd byte and stores the data in BH

**BL= (00520H) i.e. BL=34H BH= (00521H) BH=96H**

If the first byte of the data is stored at an odd address, 8086 needs two operations to read the 16-bit data.

For example, if the 16-bit data is stored at even address 00521H is 3897H MOV BX, [00521H].In first operation, 8086 reads the 16-bit data from the 00520H location and stores the data of 00521H location in register BL and discards the data of 00520H location.In 2$^{nd}$operation, 8086 reads the 16-bit data from the 00522H location and stores the data of 00522H location in register BH and discards the data of 00523H location. BL= (00521H) i.e. BL=97H BH= (00522H) BH=38H

**ByteRead**: MOV BH, [Addr]

**For Even Address:**

Ex: MOV BH, [00520H]

8086 reads the first byte from 00520 location and stores the data in BH and reads the 2$^{nd}$ byte from the 00521H location and ignores it BH [ 00520H]

**For Odd Address**

Ex: MOV BH, [00521H]

8086 reads the first byte from 00520H location and ignores it and reads the 2nd byte from the 00521 location and stores the data in BH

BH = [00521H]

**Physical address formation**: Generation of 20-bitAddress

The 8086 addresses a segmented memory. The complete physical address which is 20- bits long is generated using segment and offset registers each of the size 16-bit. The content of a segment register also called as segment address, and content of an offset register also called as offset address. To get total physical address, put the lower nibble 0H to segment address and add offset address.

The content of segment register are multiplied by 10H.i.e shifted by 4 positions to the left by inserting 4 zero bits and then the offset.i.e the contents of IP register are added to the shifted contents of CS to generate physical address.

The fig 1.3 shows formation of 20-bit physical address.

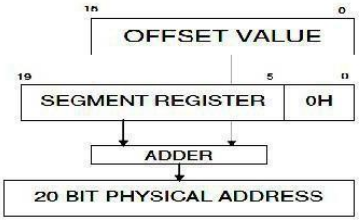
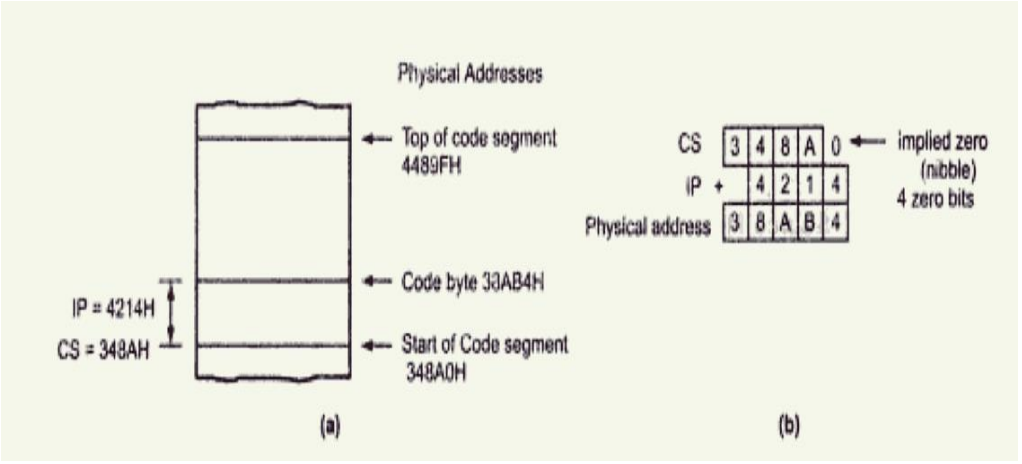



**Fig 1.3 Physical Address formation**

The contents of CS register are 348AH, therefore the shifted contents of CS register are 348A0H. When the BIU adds the offset of 4214H in IP to this starting address, we get 38AB4H as a 20-bit physical address of memory.




**[Source: Advanced Microprocessors and Microcontrollers by A.K Ray & K.M. Bhurchandi]**

**Register organization of 8086:**

All the registers of 8086 are 16-bit registers. The general purpose registers, can be used either 8-bit registers or 16-bit registers used for holding the data, variables and intermediate results temporarily or for other purpose like counter or for storing offset address for some particular addressing modes etc. The special purpose registers are used as segment registers, pointers, index registers or as offset storage registers for particular addressing modes. Fig 1.3

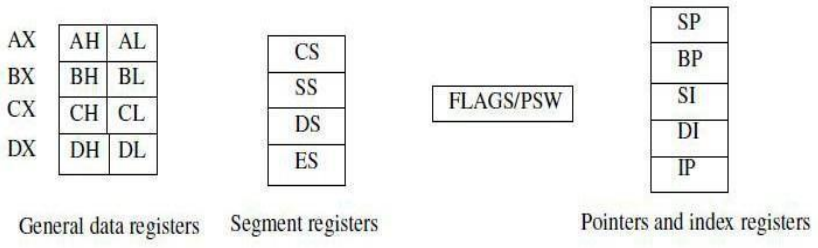



**Fig 1.4 Register organization of 8086**

**AX Register: Accumulator** register consists of two 8-bit registers AL and AH, which can be combined together and used as a 16- bit register AX. AL in this case contains the low-order byte of the word, and AH contains the high-order byte. Accumulator can be used for I/O operations, rotate and string manipulation.

**BX Register:** This register is mainly used as a **base register**. It holds the starting base location of a memory region within a data segment. It is used as offset storage for forming physical address in case of certain addressing mode.

**CX Register:** It is used as default counter - **count register** in case of string and loop instructions.

**DX Register: Data register** can be used as a port number in I/O operations and implicit operand or destination in case of few instructions. In integer 32-bit multiply and divide instruction the DX register contains high-order word of the initial or resulting number.

**Segment registers:**

1Mbyte memory is divided into 16 logical segments. The complete 1Mbyte memory segmentation is as shown in fig 1.4. Each segment contains 64Kbyte of memory. There are four segment registers.

**Code segment:** (CS) is a 16-bit register containing address of 64 KB segment with processor instructions. The processor uses CS segment for all accesses to instructions referenced by

instruction pointer (IP) register. CS register cannot be changeddirectly.

The CS register is automatically updated during far jump, far call and far return instructions. It is used for addressing a memory location in the code segment of the memory, where the executable program is stored.

**Stack segment** (SS) is a 16-bit register containing address of 64KB segment with program stack. By default, the processor assumes that all data referenced by the stack pointer (SP) and base pointer (BP) registers is located in the stack segment. SS register can be changed directly using POP instruction. It is used for addressing stack segment of memory. The stack segment is that segment of memory, which is used to store stack data.

**Data segment** (DS) is a 16-bit register containing address of 64KB segment with program data. By default, the processor assumes that all data referenced by general registers (AX, BX, CX, DX) and index register (SI, DI) is located in the data segment. DS register can be changed directly using POP and LDS instructions. It points to the data segment memory where the data is resided.

**Extra segment** (ES) is a 16-bit register containing address of 64KB segment, usually with program data. By default, the processor assumes that the DI register references the ES segment in string manipulation instructions. ES register can be changed directly using POP and LES instructions. It also refers to segment which essentially is another data segment of the memory. It also containsdata.

**Pointers and index registers.**

The pointers contain within the particular segments. The pointers IP, BP, SP usually contain offsets within the code, data and stack segments respectively

**Stack Pointer** (SP) is a 16-bit register pointing to program stack in stack segment.

**Base Pointer** (BP) is a 16-bit register pointing to data in stack segment. BP register is usually used for based, based indexed or register indirect addressing.

**Source Index** (SI) is a 16-bit register. SI is used for indexed, based indexed and register indirect addressing, as well as a source data addresses in string manipulation instructions. **Destination Index** (DI) is a 16-bit register. DI is used for indexed, based indexed and register indirect addressing, as well as a destination data address in string manipulation instructions.

**Flag Register:**

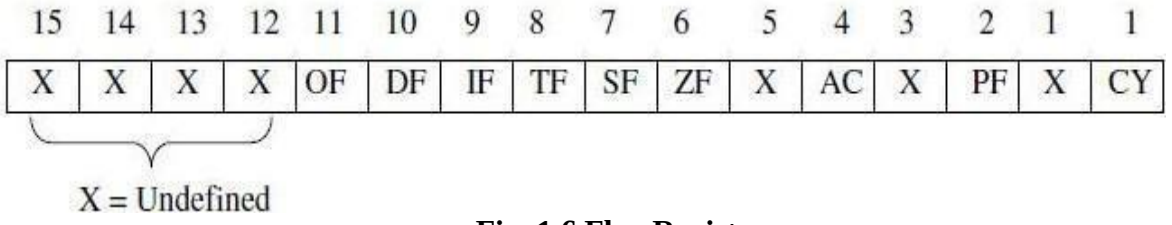



**Fig. 1.6 Flag Register**

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. The 8086 flag register as shown in the fig 1.5. 8086 has 9 active flags and they are divided into two categories:

**1.** ConditionalFlags

**2.** ControlFlags

**Conditional Flags**

**Carry Flag (CY):** This flag indicates an overflow condition for unsigned integer **a**rithmetic. It is also used in multiple-precision arithmetic.

**Auxiliary Flag (AC):** If an operation performed in ALU generates a carry/barrow from lower nibble (i.e. D0 – D3) to upper nibble (i.e. D4 – D7), the AC flag is set i.e. carry given by D3

bit to D4 is AC flag. This is not a general-purpose flag, it is used internally by the Processor to perform Binary to BCD conversion.

**Parity Flag (PF):** This flag is used to indicate the parity of result. If lower order 8-bits of the result contains even number of 1's, the Parity Flag is set and for odd number of 1's, the Parity flag is reset.

**Zero Flag (ZF):** It is set; if the result of arithmetic or logical operation is zero else it is reset.

**Sign Flag (SF):** In sign magnitude format the sign of number is indicated by MSB bit. If the result of operation is negative, sign flag is set.

**Control Flags**

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

**Trap Flag (TF):** It is used for single step control. It allows user to execute one instruction of a program at a time for debugging. When trap flag is set, program can be run in single step mode.

**Interrupt Flag (IF):** It is an interrupt enable/disable flag. If it is set, the maskable interrupt of 8086 is enabled and if it is reset, the interrupt is disabled. It can be set by executing instruction sit and can be cleared by executing CLIinstruction.

**Direction Flag (DF):** It is used in string operation. If it is set, string bytes are accessed from higher memory address to lower memory address. When it is reset, the string bytes are accessed from lower memory address to higher memory address.

***Flag Register and ADD instruction***

The flag bits affected by the ADD instructions are: CF, PF, AF, ZF, SF and OF. The OF will be studied in Chapter 6.

**Ex:** Show how the flag register is affected by the addition of 38H and 2FH.

```
Solution:   MOV  BH,38H        ;BH=38H
            ADD  BH,2FH        ;BH = BH + 2F =  38 + 2F= 67H
```

```
      38          0011  1000
+   2F          0010  1111
      67          0110  0111
```

CF = 0 since there is no carry beyond d7
PF = 0 since there is odd number of 1`s in the result
AF = 1 since there is a carry from d3 to d4
ZF = 0 since the result is not zero
SF = 0 since d7 of the result is zero

**Ex:** Show how the flag register is affected by the following addition

```
Solution:   MOV  AX,34F5H      ;AX =34F5H
            ADD  AX,95EBH      ;AX = CAE0H
```

```
      34F5        0011  0100  1111  0101
+   95EB        1001  0101  1110  1011
      CAE0        1100  1010  1110  0000
```

CF = 0 since there is no carry beyond d15
PF = 0 since there is odd number of 1s in the lower byte
AF = 1 since there is a carry from d3 to d4
ZF = 0 since the result is not zero
SF = 1 since d15 of the result is 1

## 1.3 INSTRUCTION SET OF8086

The 8086 instructions are categorized into the following main types.

1. Data Copy / TransferInstructions
2. Arithmetic and LogicalInstructions
3. Shift and Rotate Instructions
4. LoopInstructions
5. BranchInstructions
6. StringInstructions
7. Flag ManipulationInstructions
8. Machine Control Instructions

### 1.3.1 Data Copy / TransferInstructions:

The data transfer instructions move data between memory and the general-purpose and segment registers, and perform operations such as conditional moves, stack access, and data conversion.

There are four basic 8086 instructions for transferring quantities to and/or from the registers and memory such as,

- General purpose data transferinstructions
- I/O transfer instruction
- Special address transferinstruction
- Flag transfer instruction

General purpose data transfer instructions

- MOV
- PUSH
- POP
- XCHG
- XLAT

**MOV**:

This instruction copies a word or a byte of data from some source to a destination.

The destination can be a register or a memory location. The source can be a register, a memory location, or an immediate number.

Syntax:

**MOV destination,source**

Depending on the addressing modes it can transfer information from

| S.No | Type of Transfer | Instruction | Description |
|---|---|---|---|
| 1. | Register to register | MOV AX,BX | MOV r1, r2<br>(Move Data; Move the content of the one register to another).<br>[r1] <-- [r2] |
| 2. | Immediate operand to a register | MOV AX,5000H | MOV r, data.<br>(Move immediate data to register).<br>[r] <-- data. |
| 3. | Immediate operand to a memory location | MOV [8010H],5000H | MOV M, data.<br>(Move immediate data to memory).<br>M <-- data. |
| 4. | Memory location to register | MOV AX,[8010H] | MOV r, m<br>(Move the content of memory register).<br>r <-- [M] |
| 5. | Register to Memory location | MOV [8010H],AX | MOV M, r.<br>(Move the content of register to memory).<br>M <-- [r] |

| | | | |
|---|---|---|---|
| 6. | Register to segment register(except CS) | MOV [BX],AX | MOV sr, r.<br>(Move the content of register to segemnt register).<br>[sr] <-- [r] |
| 7. | segment register to Register | MOV AX,[BX] | MOV sr, m.<br>(Move the content of segemnt register to register).<br>[r] <-- [sr] |
| 8. | Memory to segment register(except CS) | MOV [BX],[8010H] | MOV sr, m.<br>(Move the content of memory to segemnt register ).<br>[sr] <-- [M] |
| 9. | segment register to Memory | MOV [8010H],[BX] | MOV sr, m.<br>(Move the content of segemnt register to memory).<br>[M] <-- [sr] |

Direct loading of the segment registers with immediate data is not permitted.

**PUSH: Push to Stack**

This instruction pushes the contents of the specified register/memory location on to the stack. The stack pointer is decremented by 2, after each execution of the instruction

E.g. **PUSH AX**
**PUSH DS**
**PUSH [5000H]**

**POP: Pop from Stack**

This instruction when executed, loads the specified register/memory location with the contents of the memory location of which the address is formed using the current stack segment and stack pointer.

The stack pointer is incremented by 2

Eg. **POP AX POP DS**
**POP [5000H]**

**XCHG: Exchange byte or word**

This instruction exchange the contents of the specified source and destination operands

Eg. **XCHG [5000H], AX**
**XCHG BX, AX**

**XLAT:Translate**

Translate byte using look-up table

Eg. **LEA BX, TABLE1**
**MOV AL, 04H**
**XLAT**

**1.3.2 Input and output port transfer instructions**:

These instruction are used to move data between accumulator and I/O port using fixed port addressing and variable port addressing.

**IN:Input the port**

Copy a byte or word from specified port to accumulator.

Eg. INAL,03H
IN AX,DX

**OUT:Output to the port**

Copy a byte or word from accumulator specified port.

Eg. OUT 03H

### 1.3.3 Special Address Transfer instructions

**LEA:Load EffectiveAddress**

Load effective address of the operand into specified register

Eg: LEA BX,ADR :effective address of label ADR

**LDS:** Load DS register and other specified register frommemory.

Eg. LDS BX,5000H

**LES:** Load ES register and other specified register frommemory.

Eg. LES BX,5000H

### 1.3.4 Flag transfer instructions:

**LAHF**:

Load (copy to) AH with the low byte the flag register. [AH] [ Flags low byte]

**SAHF:**

Store (copy) AH registers to low byte of flag register. [Flags low byte] [AH]

**PUSHF**:

Copy flag register to top of stack.

**POPF:**

Copy word at top of stack to flagregister.

### 1.3.5 ArithmeticInstructions:

The 8086 provides many arithmetic operations: addition, subtraction, negation, increment, decrement multiplication and comparing two values.

**Addition Instruction**:

- Add contents of two registers with or withoutcarry
- Add contents of a registers and a memory with or withoutcarry
- Add immediate data to a registers or a memory with or withoutcarry
- Increment the content of a register or a memorylocation
- To perform ASCII adjustment afteraddition
- To perform decimal adjustment afteraddition

**ADD:**

The add instruction adds the contents of the source operand to the destinationoperand.

**Syntax: ADD oper1, oper2**

| | |
|---|---|
| ADD AX, 0100H | Add immediate value to the content of AX |
| ADD AX, BX | Add contents of AX and BX and result in AX |
| ADD AX, [SI] | Add word from memory at offset [SI] in DS to the content of DX |
| ADD AX, [5000H] | Add content of data whose address is 5000H with AX and result in AX |
| ADD [5000H], 0100H | Add immediate value to the content of data whose address is 5000H and result in 5000H |

**ADC: Add with Carry**

This instruction performs the same operation as ADD instruction, but adds the carry flag to the result.

| | |
|---|---|
| ADC AX, 0100H | Add immediate value plus carry status to the content of AX |
| ADC AX, BX | Add contents of AX and BX plus carry status and result in AX |
| ADC AX, [SI] | Add word from memory at offset [SI] in DS plus carry status to the content of DX |
| ADC AX, [5000H] | Add content of data whose address is 5000H plus carry status with AX and result in AX |
| ADC [5000H], 0100H | Add immediate value to the content of data whose address is 5000H plus carry status and result in 5000H |

**INC: Increment**

- This instruction increases the contents of the specified Register or memory locationby
- Immediate data cannot be operand of this instruction.

Eg. INCAX
INC [BX]
INC [5000H]

**AAA: ASCII Adjust After Addition**

- The AAA instruction is executed after an ADD instruction that add two ASCII coded operand to give a byte of result inAL.
- The AAA instruction converts the resulting contents of AL to a unpacked decimal digits.
- After the addition it will check the lower 4 bits of AL is a valid BCD number in the range of 0 to9
- If it is between 0 to 9 the AF is zero and AAA setsAH=0
- If lower digit of AL is between 0 to 9 AF is set,06 is added to AL.The upper 4 bits of AL are cleared and AH is incremented byone
- If lower digit of AL greater than 9, then 06 is added to AL. The upper 4 bits of AL are cleared and AH is incremented byone.

**DAA: Decimal Adjust After Addition**

- The DAA instruction is executed after an ADD instruction that add two ASCII coded operand to give a byte of result inAL.
- The DAA instruction converts the resulting contents of AL to a unpacked decimal digits.
- If lower nibble is greater than 9,after addition it will add 06 to the lower nibble inAL.
- After adding 06 to lower nibble of AL, if upper nibble of AL is greater than 9,then adds 60H toAL.

**Subtraction Instruction**:

- Subtract contents of two registers with or without carry
- Subtract contents of a registers and a memory with or withoutcarry
- Subtract immediate data to a registers or a memory with or withoutcarry
- Decrement the content of a register or a memorylocation
- To perform ASCII adjustment afterSubtract
- To perform decimal adjustment afterSubtract

**SUB: Subtract**

The subtract instruction subtracts the source operand from the destination operand and the result is left in the destination operand.

| | |
|---|---|
| SUB AX, 0100H | Subtract immediate value to the content of AX |
| SUB AX, BX | Subtract contents of AX and BX and result in AX |
| SUB AX, [SI] | Subtract word from memory at offset [SI] in DS to the content of DX |
| SUB AX, [5000H] | Subtract content of data whose address is 5000H with AX and result in AX |
| SUB [5000H], 0100H | Subtract immediate value to the content of data whose address is 5000H and result in 5000H |

**SBB: Subtract with Borrow**

The subtract with borrow instruction subtracts the source operand and the borrow flag (CF) which may reflect the result of the previous calculations, from the destination operand

| | |
|---|---|
| SBB AX, 0100H | Subtract immediate value plus carry status to the content of AX |
| SBB AX, BX | Subtract contents of AX and BX plus carry statusand result in AX |
| SBB AX, [SI] | Subtract word from memory at offset [SI] in DS plus carry status to the content of DX |
| SBB AX, [5000H] | Subtract content of data whose address is 5000H plus carry status with AX and result in AX |
| SBB [5000H], 0100H | Subtract immediate value to the content of datawhose address is 5000H plus carry status and result in5000H |

**DEC: Decrement**

The decrement instruction subtracts 1 from the contents of the specified register or memory location.

Eg. DEC AX

DEC [5000H]

**AAS: ASCII Adjust After Subtraction**

- The AAA instruction is executed after an SUB instruction that subtracts two ASCII coded operand to give a byte of result in AL.
- The AAA instruction converts the resulting contents of AL to a unpacked decimal digits.
- After the addition it will check the lower 4 bits of AL is a valid BCD number in the range of 0 to9
- If it is between 0 to 9 the AF is zero and AAA setsAH=0
- If lower digit of AL is between 0 to 9 AF is set,06 is subtracted to AL. The upper 4 bits of AL are cleared and AH is incremented byone
- If lower digit of AL greater than 9, then 06 is subtracted to AL. The upper 4 bits of AL are cleared and AH is incremented byone

**DAS: Decimal Adjust After Subtraction**

- The DAA instruction is executed after an SUB instruction that subtract two ASCII coded operand to give a byte of result in AL.
- The DAA instruction converts the resulting contents of AL to a unpacked decimal digits.
- If lower nibble is greater than 9, after subtraction it will subtract 06 to the lower nibble inAL.
- After subtracting 06 to lower nibble of AL, if upper nibble of AL is greater than 9,then subtract 60H to AL.

**NEG: Negate**

The negate instruction forms 2's complement of the specified destination in the instruction. The destination can be a register or a memory location. This instruction can be implemented by inverting each bit and adding 1 to it.

Eg. NEG AL

AL = 0011 0101 35H

Replace number in AL with its 2's complement

AL = 1100 1011 = CBH

**CMP: Compare**

This instruction compares the source operand, which may be a register or an immediate data or a memory location, with a destination operand that may be a register or a memory location

Eg. CMP BX, 0100H CMP AX, 0100H

CMP [5000H], 0100H CMP BX, [SI]

CMP BX, CX

**Multiplication Instruction:**

**MUL:Unsigned Multiplication Byte or Word**

This instruction multiplies an unsigned byte or word by the contents of AL.

Eg. MUL BH; (AX) (AL) x (BH)

MUL CX; (DX)(AX) (AX) x (CX)

MUL WORD PTR [SI]; (DX)(AX) (AX) x ([SI])

**IMUL:Signed Multiplication**

This instruction multiplies a signed byte in source operand by a signed byte in AL or a signed word in source operand by a signed word in AX.

Eg. IMUL BH
IMUL CX
IMUL [SI]

**AAM: ASCII Adjust after Multiplication**

This instruction, after execution, converts the product available In AL into unpacked BCD format.

Eg. MOV AL, 04; AL = 04

MOV BL ,09; BL = 09

MUL BL; AX = AL*BL; AX=24H AAM;

AH = 03, AL=06

**Division Instruction:**

**DIV: Unsigneddivision**

This instruction is used to divide an unsigned word by a byte or to divide an unsigned double word by a word.

Eg. DIV CL; Word in AX / byte in CL; Quotient in AL, remainder in AH
DIV CX; Double word in DX and AX / word; in CX, and Quotient in AX; remainder in DX

**IDIV: Unsigned division**

This instruction is used to divide an signed word by a byte or to divide an unsigned double word by a word.

Eg. IDIV CL; Word in AX / byte in CL; Quotient in AL, remainder in AH
IDIV CX; Double word in DX and AX / word; in CX, and Quotient inAX; remainder in DX

**AAD: ASCII Adjust before Division**

This instruction converts two unpacked BCD digits in AH and AL to the equivalent binary number in AL. This adjustment must be made before dividing the two unpacked BCD digits in AX by an unpacked BCD byte. In the instruction sequence, this instruction appears before DIV instruction.

**Eg. AX 05 08**

AAD result in AL 00 3A 58D = 3A H in AL

The result of AAD execution will give the hexadecimal number 3A in AL and 00 in AH where 3A is the hexadecimal Equivalent of 58(decimal).

**CBW: Convert Signed Byte to Word**

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

**Eg. CBW**

AX= 0000 0000 1001 1000 Convert signed byte in AL signed word in AX. Result in AX = 1111 1111 1001 1000

**CWD: Convert Signed Word to Double Word**

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

**Eg.CWD**

Convert signed word in AX to signed double word in DX: AX DX= 1111 1111 1111 1111 Result in AX = 1111 0000 1100 0001

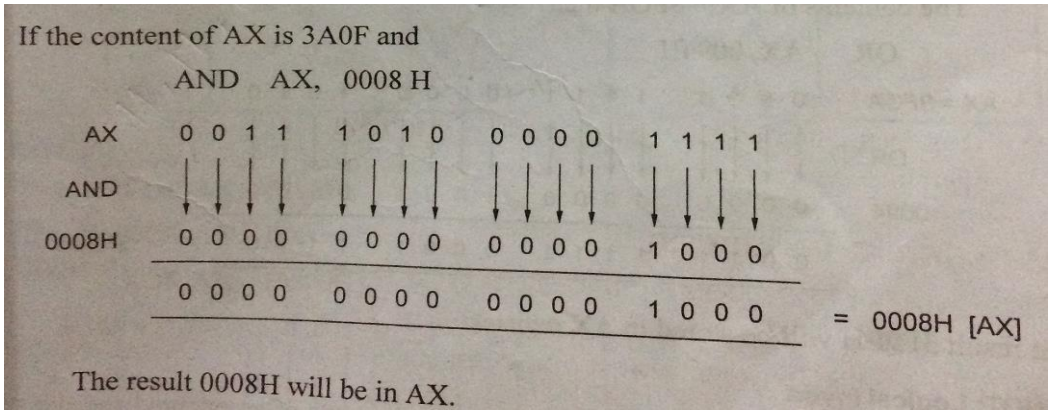
### 1.3.5. Logical instructions

**AND: LogicalAND**

This instruction bit by bit ANDs the source operand that may be an immediate register or a memory location to the destination operand that may a register or a memory location. The result is stored in the destination operand.

Syntax: | AND destination , source |

Eg. AND AX, 0008H
AND AX, BX

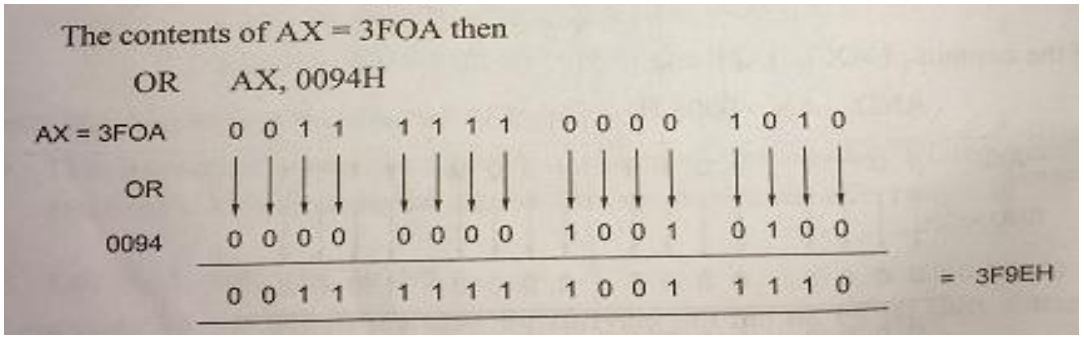



**[Source: Advanced Microprocessors and Microcontrollers by A.K Ray & K.M. Bhurchandi]**

**OR: Logical OR**

This instruction bit by bit ORs the source operand that may be an immediate, register or a memory location to the destination operand that may a register or a memory location. The result is stored in the destination operand.

Syntax: | OR destination , source |



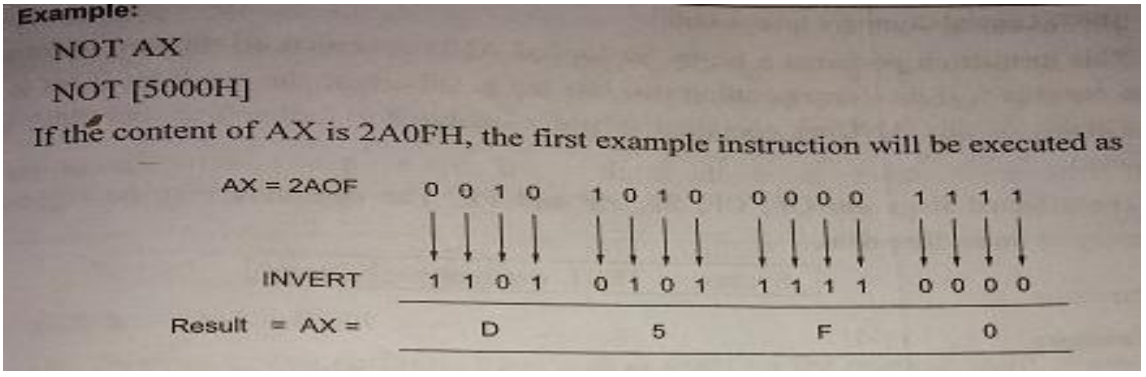

**[Source: Advanced Microprocessors and Microcontrollers by A.K Ray & K.M. Bhurchandi]**

**NOT: Logical Invert**

This instruction complements the contents of an operand register or a memory location, bit by bit.

Syntax:

| NOTdestination |
|---|




**[Source: Advanced Microprocessors and Microcontrollers by A.K Ray & K.M. Bhurchandi]**

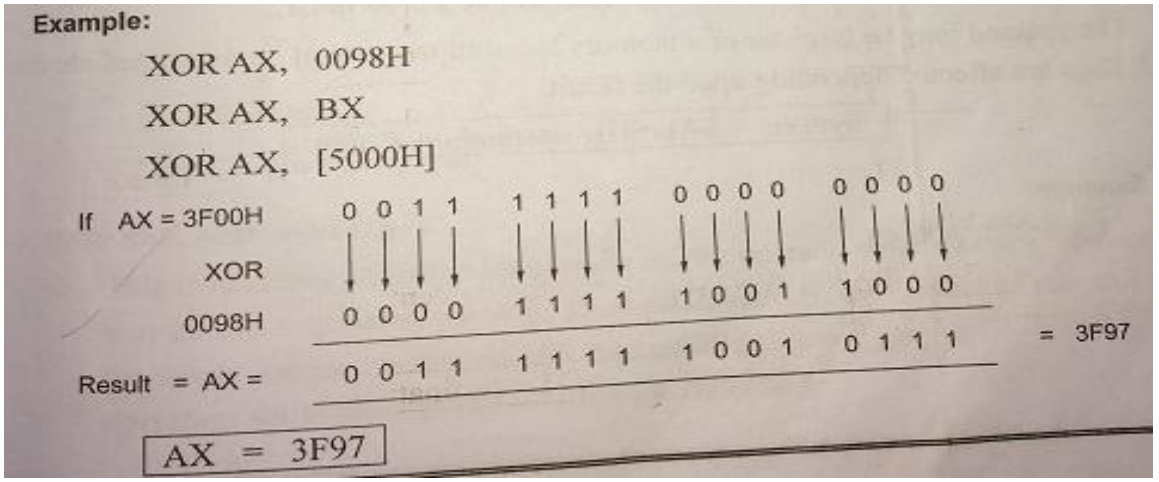
**OR: Logical Exclusive OR**

This instruction bit by bit XORs the source operand that may be an immediate, register or a memory location to the destination operand that may a register or a memory location. The result is stored in the destination operand.

Syntax:

| XOR destination ,source |
|---|

Eg. XOR AX, 0098H
XOR AX,BX




**[Source: Advanced Microprocessors and Microcontrollers by A.K Ray & K.M. Bhurchandi]**

**TEST: Logical Compare Instruction**

The TEST instruction performs a bit by bit logical AND operation on the two operands. The result of this ANDing operation is not available for further use, but flags are affected.

Syntax: TEST destination ,source

Eg. TEST [0500], 06H

**Shift and Rotate Instructions**

**SAL/SHL:**

SAL and SHL are two mnemonics for the same instruction.

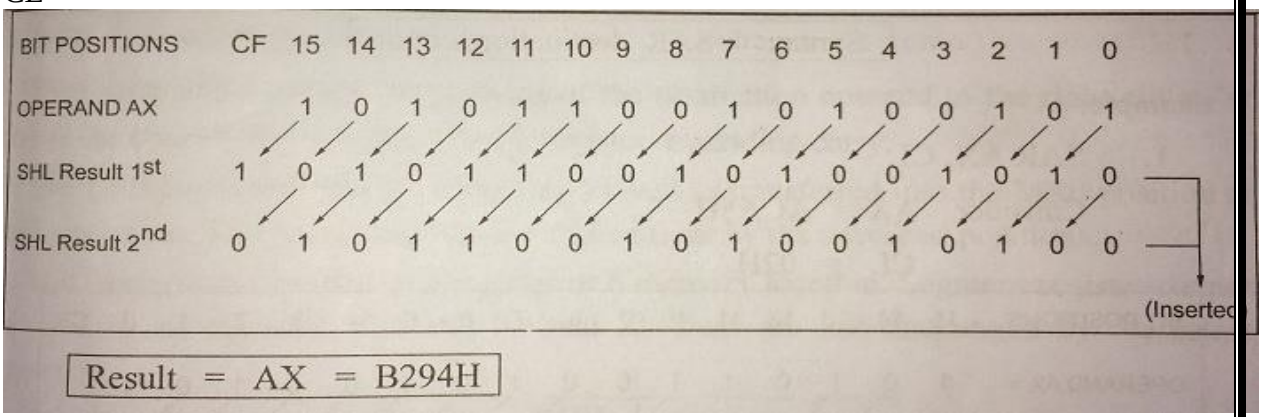
- This instruction shifts each bit in the specified destination to the left and 0 is stored at LSBposition.
- The MSB is shifted into the carryflag.
- The destination can be a byte or aword.
- It can be in a register or in a memorylocation.
- The number of shifts is indicated by count.

Syntax:

**SAL / SHL destination, count.**

Eg. SAL CX, 1

SHL AX,
CL




**[Source: Advanced Microprocessors and Microcontrollers by A.K Ray & K.M.**

**Bhurchandi]**

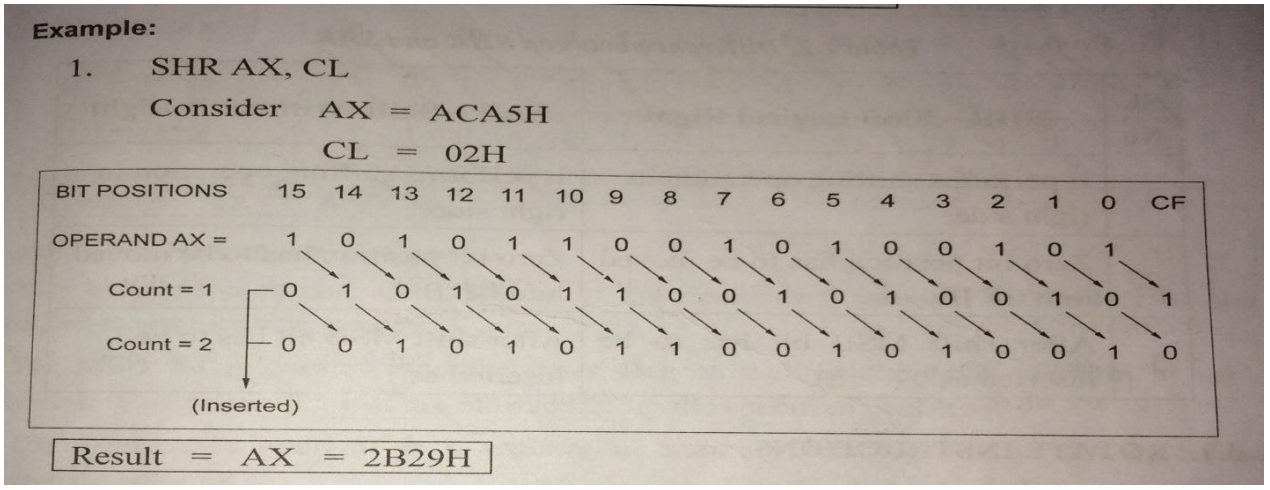
**SHR: SHR destination, count**

This instruction shifts each bit in the specified destination to the right and 0 is stored at MSB position.

- The LSB is shifted into the carryflag.
- The destination can be a byte or aword.
- It can be a register or in a memorylocation.
- The number of shifts is indicated by count.

Syntax:

| **SHR destination, count.** |
|---|



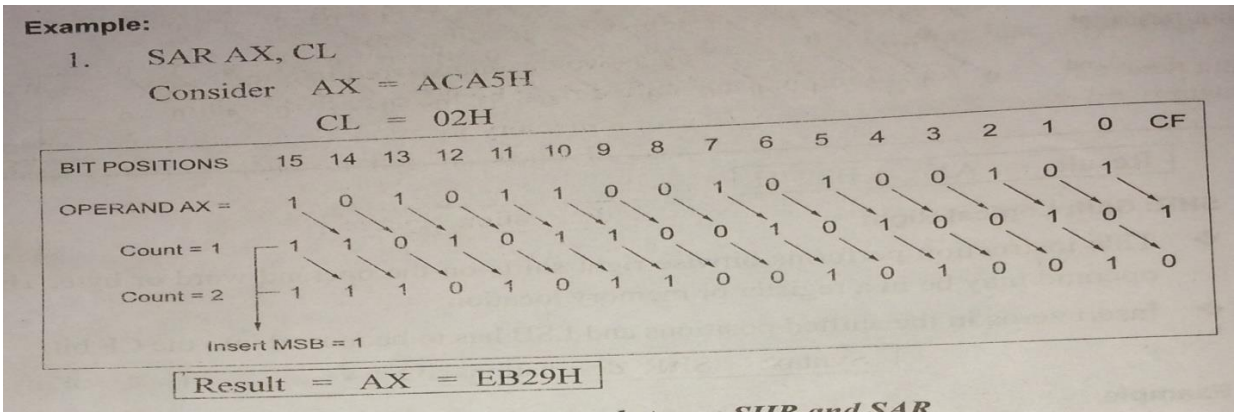

**[Source: Advanced Microprocessors and Microcontrollers by A.K Ray & K.M. Bhurchandi]**

**SAR: SAR destination, count**

This instruction shifts each bit in the specified destination some number of bit positions to the right. As a bit is shifted out of the MSB position, a copy of the old MSB is put in the MSB position. The LSB will be shifted into CF.

Syntax:

| **SAR destination, count** |
|---|




**[Source: Advanced Microprocessors and Microcontrollers by A.K Ray & K.M. Bhurchandi]**

*Table 1.2. Difference between SHR and SAR*

| Sl. No | SHR – Shift Logical Right | SAR – Shift Arithmetic Right |
|---|---|---|
| 1. | It performs shifting operation in right side. | It performs shifting operation in right side. |
| 2. | Zero bit position has to be moved into CF Bit | Zero bit position has to be moved into CF Bit |
| 3. | After shift MSB bit has to be inserted as 0 (zero) | After shift MSB bit has to be inserted as 1. |

**[Source: Advanced Microprocessors and Microcontrollers by A.K Ray & K.M. Bhurchandi]**

**ROL Instruction: Rotate left without carry**

This instruction rotates all bits in a specified byte or word to the left some number of bit positions. MSB is placed as a new LSB and a new CF.

**Syntax: ROL destination, count.**

**ROR Instruction: Rotate right without carry**

This instruction rotates all bits in a specified byte or word to the right some number of bit positions. LSB is placed as a new MSB and a new CF.

**Syntax: ROR destination, count**

**RCL Instruction: Rotate left with carry**

This instruction rotates all bits in a specified byte or word some number of bit positions to the left along with the carry flag. MSB is placed as a new carry and previous carry is place as new LSB.

**Syntax: RCL destination, count.**

**RCR Instruction: Rotate right with carry**

This instruction rotates all bits in a specified byte or word some number of bit positions to the right *along with the carry flag*. LSB is placed as a new carry and previous carry is place as new MSB.

**Syntax: RCR destination, count.**

### 1.3.6. LoopInstructions:

**Unconditional LOOP Instructions**

- **LOOP: LOOPUnconditionally**

This instruction executes the part of the program from the Label orAddressspecified in the instruction upto the LOOP instruction CX number of times. At each iteration, CX is decremented automatically and JUMP IF NOT ZERO structure.

Example: MOV CX, 0004H

**Conditional LOOP Instructions**

- **LOOPZ / LOOPELabel**

  Loop through a sequence of instructions from label while ZF=1 and CX=0.

- **LOOPNZ / LOOPENELabel**

  Loop through a sequence of instructions from label while ZF=1 and CX=0.

### 1.3.7. BranchInstructions:

Branch Instructions transfers the flow of execution of the program to a new Address specified in the instruction directly or indirectly. When this type of instruction is executed, the CS and IP registers get loaded with new values of CS and IP corresponding to the location to be transferred. The Branch Instructions are classified into two types

1. Unconditional BranchInstructions.
2. Conditional BranchInstructions.

**Unconditional Branch Instructions:**

In Unconditional control transfer instructions, the execution control is transferred to the specified location independent of any status or condition. The CS and IP are unconditionally modified to the new CS and IP.

- **CALL: UnconditionalCall**

This instruction is used to call a Subroutine (Procedure) from a main program. Address of procedure may be specified directly or indirectly. There are two types of procedure depending upon whether it is available in the same segment or in another segment.

i. Near CALL i.e., ±32Kdisplacement.

ii. For CALL i.e., anywhere outside thesegment.

On execution this instruction stores the incremented IP & CS onto the stack and loads the CS & IP registers with segment and offset Addresses of the procedure to be called. **RET:**

- **Return from theProcedure.**

At the end of the procedure, the RET instruction must be executed. When it is executed, the previously stored content of IP and CS along with Flags are retrieved into the CS, IP and Flag registers from the stack and execution of the main program continues further.

- **INT N: Interrupt TypeN.**

In the interrupt structure of 8086, 256 interrupts are defined corresponding to the types from 00H to FFH. When INT N instruction is executed, the type byte N is multiplied by 4 and the contents of IP and CS of the interrupt service routine will be taken from memory block in 0000 segment.

- **INTO: Interrupt onOverflow**

This instruction is executed, when the overflow flag OF is set. This is equivalent to a Type 4 Interruptinstruction.

- **JMP: UnconditionalJump**

This instruction unconditionally transfers the control of execution to the specified Address using an 8-bit or 16-bit displacement. No Flags are affected by thisinstruction.

- **IRET: Return fromISR**

When it is executed, the values of IP, CS and Flags are retrieved from the stack to continue the execution of the main program.

**Conditional Branch Instructions**

When this instruction is executed, execution control is transferred to theAddressspecified relatively in the instruction, provided the condition implicit in the Opcode is satisfied. Otherwise execution continues sequentially.

- **JZ/JELabel**

Transfer execution control to Address‘Label’, if ZF=1.

- **JNZ/JNELabel**

Transfer execution control to Address‘Label’, if ZF=0

- **JS Label**

Transfer execution control to Address‘Label’, if SF=1.

- **JNS Label**

Transfer execution control to Address‘Label’, if SF=0.

- **JO Label**

Transfer execution control to Address'Label', if OF=1.

- **JNO Label**

Transfer execution control to Address'Label', if OF=0.

- **JNPLabel**

Transfer execution control to Address'Label', if PF=0.

- **JPLabel**

Transfer execution control to Address'Label', if PF=1.

- **JB Label**

Transfer execution control to Address'Label', if CF=1.

- **JNB Label**

Transfer execution control to Address'Label', if CF=0.

- **JCXZLabel**

Transfer execution control to Address'Label', if CX=0

### 1.3.8. String ManipulationInstructions

A series of data byte or word available in memory at consecutive locations, to be referred as Byte String or Word String. A String of characters may be located in consecutive memory locations, where each character may be represented by its ASCII equivalent. The 8086 supports a set of more powerful instructions for string manipulations for referring to a string, two parameters are required.

I. Starting and EndAddressof theString.
II. II. Length of theString.

The length of the string is usually stored as count in the CX register. The incrementing or decrementing of the pointer, in string instructions, depends upon the Direction Flag (DF) Status. If it is a Byte string operation, the index registers are updated by one. On the other hand, if it is a word string operation, the index registers are updated bytwo.

**REP: Repeat Instruction Prefix**

This instruction is used as a prefix to other instructions, the instruction to which the REP prefix is provided, is executed repeatedly until the CX register becomes zero (at each iteration CX is automatically decremented by one).

i. REPE / REPZ - repeat operation while equal / zero.
ii. REPNE / REPNZ - repeat operation while not equal / notzero.

These are used for CMPS, SCAS instructions only, as instruction prefixes.

**MOVSB / MOVSW: Move String Byte or String Word**

Suppose a string of bytes stored in a set of consecutive memory locations is to be moved to another set of destination locations. The starting byte of source string is located in the memory location whose Address may be computed using SI (Source Index) and DS (Data Segment) contents. The starting Address of the destination locations where this string has to be relocated is given by DI (Destination Index) and ES (Extra Segment) contents.

**CMPS: Compare String Byte or String Word**

The CMPS instruction can be used to compare two strings of byte or words. The length of the string must be stored in the register CX. If both the byte or word strings are equal, zero Flag is set.

The REP instruction Prefix is used to repeat the operation till CX (counter) becomes zero or the condition specified by the REP Prefix is False.

**SCAN: Scan String Byte or String Word**

This instruction scans a string of bytes or words for an operand byte or word specified in the register AL or AX. The String is pointed to by ES: DI register pair. The length of the string

stored in CX. The DF controls the mode for scanning of the string. Whenever a match to the specified operand is found in the string, execution stops and the zero Flag is set. If no match is found, the zero flag is reset.

**LODS: Load String Byte or String Word**

The LODS instruction loads the AL / AX register by the content of a string pointed to by DS: SI register pair. The SI is modified automatically depending upon DF, If it is a byte transfer (LODSB), the SI is modified by one and if it is a word transfer (LODSW), the SI is modified by two. No other Flags are affected by this instruction.

**STOS: Store String Byte or String Word**

The STOS instruction Stores the AL / AX register contents to a location in the string pointer by ES: DI register pair. The DI is modified accordingly, No Flags are affected by this instruction.

The direction Flag controls the String instruction execution, The source index SI and Destination Index DI are modified after each iteration automatically. If DF=1, then the execution follows auto decrement mode, SI and DI are decremented automatically after each iteration. If DF=0, then the execution follows auto increment mode. In this mode, SI and DI are incremented automatically after eachiteration.

**Flag Manipulation and Processor Control Instructions**

These instructions control the functioning of the available hardware inside the processor chip. These instructions are categorized into two types:

1. Flag Manipulationinstructions.
2. Machine Controlinstructions.

### 1.3.9. Flag Manipulationinstructions

The Flag manipulation instructions directly modify some of the Flags of 8086.

i. CLC – Clear CarryFlag.

ii. CMC – Complement Carry Flag. iii. STC – Set CarryFlag.

iv. CLD – Clear Direction Flag. v. STD – Set Direction Flag.

vi. CLI – Clear Interrupt Flag. vii. STI – Set Interrupt Flag.

### 1.3.10. Machine Controlinstructions

The Machine control instructions control the bus usage and execution i.

WAIT – Wait for Test input pin to go low. ii. HLT – Halt the process.

iii. NOP – Nooperation.

iv. ESC – Escape to external device likeNDP

v. LOCK – Bus lock instruction prefix.

## 1.4 ADDRESSINGMODES

The set of mechanisms by which an instruction can specify how to obtain its operands is known as Addressing modes. The Addressing modes of 8086 can be broken into two categories such as,

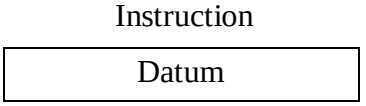
1. Data related Addressingmodes
2. Branch Addressingmodes

The CPU can access the operands (data) in a number of different modes The 8086 has 12 Addressing modes can be classified into five groups.

- Addressing modes for accessing immediate and register data (register and immediate modes).
- Addressing modes for accessing data in memory (memorymodes)
- Addressing modes for accessing I/O ports (I/Omodes)
- Relative Addressingmode
- Implied Addressingmode

**1. Immediate Addressingmode:**

In this mode, 8 or 16 bit data can be specified as part of the instruction - OP Code Immediate Operand




**Example 1: MOV CL, 03 H: Moves the 8 bit data 03 H into CL**
**Example 2: MOV DX, 0525 H:**

Moves the 16 bit data 0525 H into DX

In the above two examples, the source operand is in immediate mode and the destination operand is in register mode.

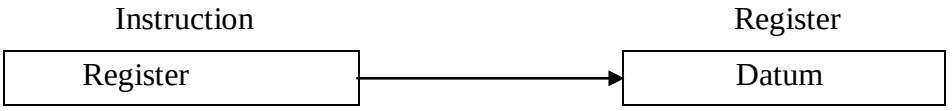
A constant such as "VALUE" can be defined by the assembler EQUATE directive such as VALUE EQU 35H

**Example: MOV BH, VALUE**

Used to load 35 H into BH

**2. Register Addressingmode:**

The operand to be accessed is specified as residing in an internal register of 8086. Anyone internal registers can be used as a source or destination operand, however only the data registers can be accessed as either a byte or word.




**Example 1: MOV DX,CX**

MOV DX (Destination Register) , CX (Source Register) Which moves 16 bit content of CS into DX.

**Example 2: MOV CL, DL**

Moves 8 bit contents of DL into CL

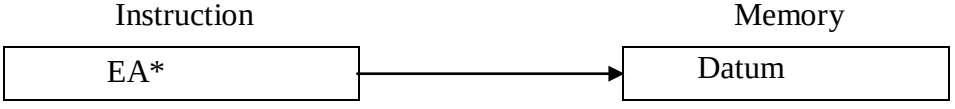
**Example 3: MOV BX, CH is an illegal instruction.**

* The register sizes must be the same.

**3. Direct Addressing mode:**

The instruction Opcode is followed by an affective address, this effective Address is directly used as the 16 bit offset of the storage location of the operand from the location specified by the current value in the selected segment register. The default segment is always DS.The 20 bit physicalAddressof the operand in memory is normally obtained as

PA = DS:EA




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

**Example 1: MOV AX, [5000H]**

If DS = 1010H, OFFSET=5000, AX = 1000H then EA=15100H.

DS:BX ➔1010H:5000H
10*HDS➔ 10100
[BX] ➔ +5000

EA ➔ 12100H

Example 2:

MOV CH, START

If [DS] = 3050 and START = 0040

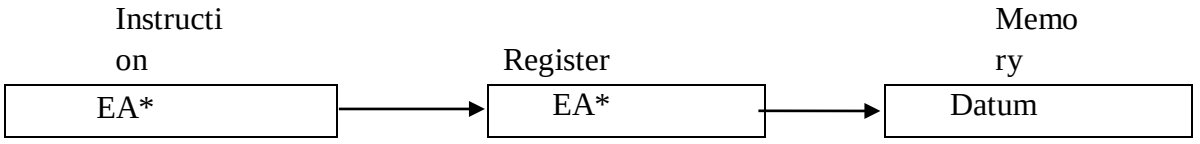
8 bit content of memory location 30540 is moved to CH.

**4.<u>Register indirect Addressingmode</u>:**

The EA is specified in either pointer (BX) register or an index (SI or DI) register.

**Example: MOV AX, [BX]**

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




**MOV AX, [BX]**

If DS = 1010H, BX = 2000H, AX = 1000H then EA=12100H.

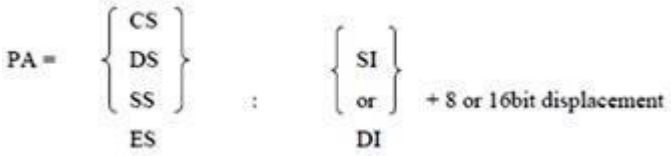
DS:BX ➔1010H:2000H
10*HDS➔ 10100
[BX] ➔ +2000

EA ➔ 12100H

**5.<u>IndexedAddressing</u>:**

The offset of the operand is stored in one of the index registers. DS and ES are the default segments for index registers SI and DI respectively.

**Example: MOV AX, [SI]**




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

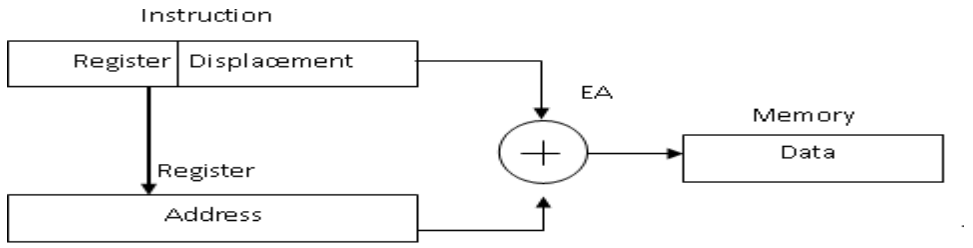
If DS = 1010H, SI = 3010H, then EA=13110H.

DS: SI ➔1010H: 3010H
10*HDS➔ 10100
[BX] ➔ +3010

EA ➔ 13110H

**6.<u>Register Relative addressing</u>:**

In this Addressing mode, the data is available at an effective Address formed by adding an 8-bit or 16-bit displacement with the content of any one of the registers BX, BP, SI and DI in the default (either DS or ES) segment. The example given before explains thismode.




**(a) Register Relative Addressing Mode**

**[Source: Advanced Microprocessors and Microcontrollers by A.K Ray & K.M. Bhurchandi]**

**Example: MOV AX, 5000H [BX]**

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

If DS = 1010H, BX = 2000H, offset=5000 then EA=13110H.

DS:[5000+BX] ➔1010H: 5000+2000H

10*H DS ➔ 10100

Offset ➔ 5000

[BX] ➔ + 2000

EA ➔ 17100H

**7.<u>BasedIndexed</u>:**

The effective Address of data is formed, in this Addressing mode, by adding content of a base register (any one of BX or BP) to the content of an index register (any one of SI or DI). The default segment register may be ES or DS.

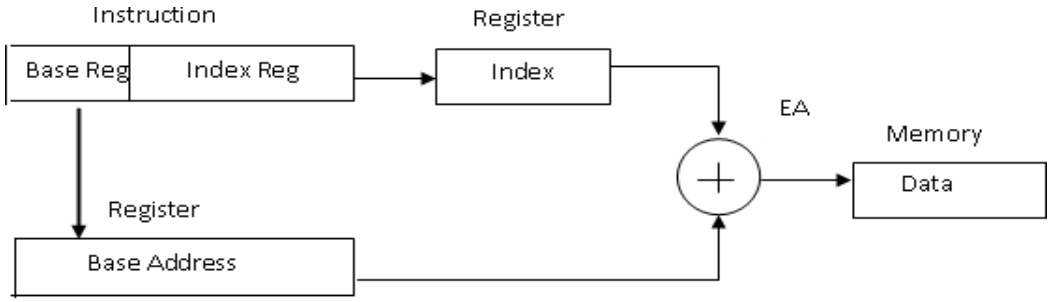



**Figure (a) Based Indexed Addressing Mode**

**[Source: Advanced Microprocessors and Microcontrollers by A.K Ray & K.M. Bhurchandi]**

**Example: MOV AX, [BX] [SI]**

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

If DS = 1010H, BX = 2000H, SI=3010 then EA=15110H.

DS:[SI+BX] ➔1010H:[3010H: 2000H]

10*HDS ➔ 10100

[SI] ➔ 3010

[BX] ➔ +2000

EA ➔ 15110H

**8.<u>Relative BasedIndexed</u>:**

The effective Address is formed by adding an 8-bit or 16-bit displacement with the sum of contents of any one of the bases registers (BX or BP) and any one of the index registers, in a default segment.

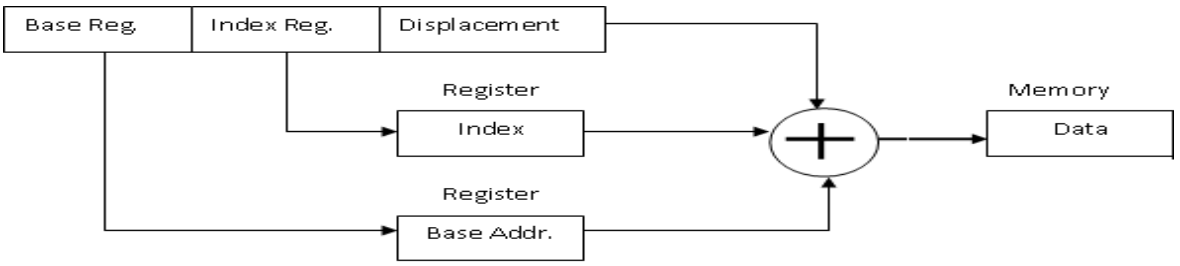



**Figure (a)** **Relative Based Indexed Addressing Mode**

**[Source: Advanced Microprocessors and Microcontrollers by A.K Ray & K.M. Bhurchandi]**

**Example: MOV AX, 5000H [BX] [SI]**

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

If DS = 1010H, BX = 2000H, SI=3010 then EA=1A110H.

DS:[5000+BX+SI] ➔1010H: [5000+2000H+3010H]

10*H DS ➔ 10100

[SI] ➔ 3010

[BX] ➔ 2000

Offset ➔ +5000

EA ➔ 1A110H

**8086 ADDRESS MODES**

| TYPE | INSTRUCTION | SOURCE | ADDRESS GENERATION | DESTINATION |
|---|---|---|---|---|
| REGISTER | MOV AX, BX | REGISTER BX | | REGISTER AX |
| IMMEDIATE | MOV CH,3AH | DATA 3AH | | REGISTER CH |
| DIRECT | MOV [1234], AX | REGISTER AX | (DS x 10H) + DISPLACEMENT<br>10000H + 1234 | MEMORY 11234H |
| REGISTER INDIRECT | MOV [BX], CL | REGISTER CL | (DS x10H) + BX<br>10000H + 0300H | MEMORY 10300H |
| BASE PLUS INDEX | MOV [BX + SI], BP | REGISTER BP | (DS x 10H) + BX + SI<br>10000H + 0300H + 0200H | MEMORY 10500H |
| REGISTER RELATIVE | MOV CL, [BX +4] | MEMORY 10304H | (DS x 10H) + BX + 4<br>10000H + 0300H + 4 | REGISTER CL |
| BASE RELATIVE PLUS INDEX | MOV ARRAY [BX + SI], DX | REGISTER DX | (DS x 10H) + ARRAY + BX + SI<br>10000H+1000H+0300H+0200H | MEMORY 11500H |

ASSUME: BX = 0300H; SI = 0200H; ARRAY = 1000H; DS = 1000H

**[Source: Advanced Microprocessors and Microcontrollers by A.K Ray & K.M. Bhurchandi]**

**Branch Related AddressingModes:**

These type of Addressing are related to whether the Addressing is within the same segment or to a different segment. Accordingly the Addressing modes in this category are known as intrasegment and intersegment with direct or indirect addressing. These are explained below:

- **Intrasegment Direct Addressing mode:** The effective Address is the sum of the IP and 8 / 16 bit displacement. It leads to a short jump if displacement is 8 bit, and this Addressing may be used conditional or unconditional in aprogram.

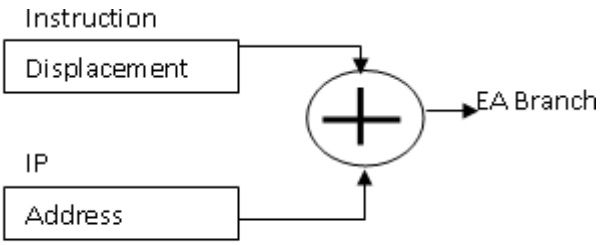



**Figure: Intrasegment Addressing Mode**

- **Intrasegment Indirect Addressing:** In this Addressing mode the effective Address may be in a register or at a memory location as accessed by any data related addressing mode except the immediate and implied mode. This Addressing mode is called only unconditionally.

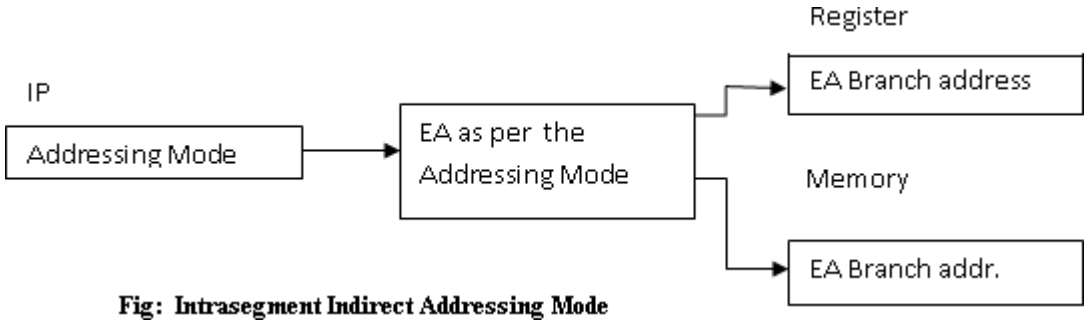



**Fig: Intrasegment Indirect Addressing Mode**

- **Intersegment direct Addressing mode:** This Addressing mode when used replaces the content of the CS and IP with the offset and segment part of the instruction. Used to branch from one segment to anothersegment.

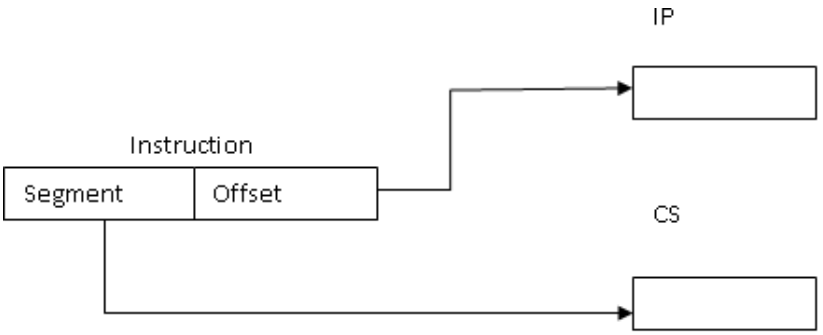



**Figure: Intersegment Direct Addressing Mode**

- **Intersegment Indirect Addressing mode:** The Addressing mode replaces the content of the CS and IP with theAddress given in a register or in memory using any of the data related Addressingmodes.

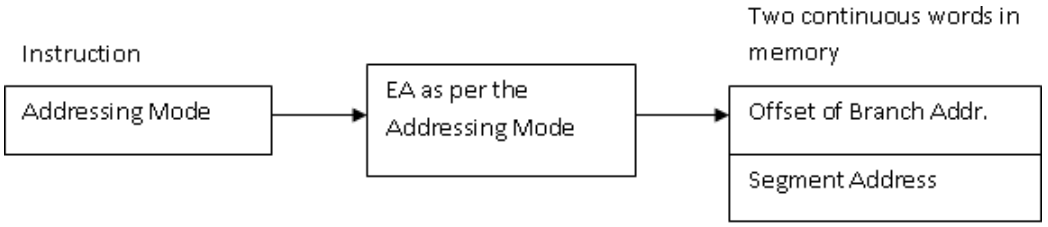



**Figure: Intersegment Indirect Addressing Mode**

### 1.5 Assemblerdirectives:

Assembler directives help the assembler to correctly understand the assembly language programs to prepare the codes. Another type of hint which helps the assembler to assign a particular constant with a label or initialize particular memory locations or labels with constants is called an operator. Rather, the operators perform the arithmetic and logical tasks unlike directives that just direct the assembler to correctly interpret the program to code it appropriately. The following directives are commonly used in the assembly language programming practice using Microsoft Macro Assembler (MASM) or Turbo Assembler (TASM).

- **DB: Define Byte** The DB directive is used to reserve byte or bytes of memory locations in the availablememory.

LIST DB 0lH, 02H, 03H, 04H

This statement directs the assembler to reserve four memory locations for a list named LIST and initialize them with the above specified four values.

- **DW: DefineWord**.

It makes the assembler reserve the number of memory words (16-bit) instead of bytes. Some examples are given to explain this directive.

Examples WORDS DW 1234H, 4567H, 78ABH, 045CH

- **DQ: Define Quad word** This directive is used to direct the assembler to reset 4words (8 bytes) of memory for the specified variable and may initialize it with the specifiedvalues.
- **DT: Define Ten Bytes.** The DT directive directs the assembler to define the specified variable requiring l -bytes for its storage and initialize the 10bytes with the specified values.
- **ASSUME**: **Assume Logical Segment Name** The ASSUME directive is used to inform the assembler, the names of the logical segments to be assumed for different segments used in the program. In the assembly language program, each segment is given aname.

For example, the code segment may be given the name CODE, data segment may be given the name DATA etc.

- **ASSUME CS: CODE** directs the assembler that the machine codes are available in a segment named CODE, and hence the CS register is to be loaded with the Address(segment) allotted by the operating system for the label CODE, whileloading.
- **ASSUME DS: DATA** indicates to the assembler that the data items related to the program, are available in a logical segment named DATA, and the DS registeris to be initialized by the segment Address value decided by the operating system for the data segment, while loading.
- **END: END of Program** The END directive marks the end of an assembly language program. When the assembler comes across this END directive, it ignores the source lines available later on. Hence, it should be ensured that the END statement should be the last statement in the file and should not appear inbetween.
- **ENDP: END of Procedure.** In assembly language programming, the subroutines are called procedures. Thus, procedures may be independent program moduleswhich return particular results or values to the calling programs. The ENDP directive is used to indicate the end of a procedure.

```
PROCEDURE STAR
.
.
.

STAR ENDP
```

- **ENDS: END of Segment** This directive marks the end of a logicalsegment.

```
DATA SEGMENT
.
.
.
DATA ENDS


ASSUME CS: CODE, DS:DATA CODE SEGMENT.
```

.
.
.
CODE ENDS
END

The above structure represents a simple program containing two segments named DATA and CODE. The data related to the program must lie between the DATA SEGMENT and DATA ENDS statements. Similarly, all the executable instructions must lie between CODE SEGMENT and CODE ENDS statements.

- **EVEN: Align on Even Memory Address** The EVEN directive updates the location counter to the next even Address if the current location counter contents are not even, and assigns the following routine or variable or constant to thatAddress.
- **EQU: Equate** The directive EQU is used to assign a label with a value or a symbol. The use of this directive is just to reduce the recurrence of the numerical values or constants in a programcode.

Example

LABEL EQU 0500H
ADDITION EQU ADD

The first statement assigns the constant 500H with the label LABEL, while the second statement assigns another label ADDITION with mnemonic ADD

- **EXTRN: External and PUBLIC: Public** The directive EXTRN informs the assembler that the names, procedures and labels declared after this directive have already been defined in some other assembly languagemodules.
- **GROUP: Group the Related segment** The directive is used to form logical groups of segments with similar purpose or type. This directive is used to inform the assembler to form a logical group of the following segmentnames.

PROGRAM GROUP CODE, DATA, STACK

The above statement directs the loader/linker to prepare an EXE file such that CODE, DATA and STACK segment must lie within a 64kbyte memory segment that is named as PROGRAM. Now, for the ASSUME statement, one can use the label PROGRAM rather than CODE, DATA and STACK asshown.

ASSUME CS: PROGRAM, DS: PROGRAM, SS: PROGRAM.

- **LABEL: Label** The Label directive is used to assign a name to the current content of the location counter. At the start of the assembly process, the assembler initializes a location counter to keep track of memory locations assigned to theprogram.
- **LENGTH: Byte Length of a Label** This directive is not available in MASM. This is used to refer to the length of a data array or astring.

MOV CX, LENGTH ARRAY

- **LOCAL** The labels, variables, constants or procedures declared LOCAL in a module are to be used only by thatmodule.
- **NAME: Logical Name of a Module** the NAME directive is used to assign a name to an assembly language program module. The module may now be referred to by its declared name.
- **OFFSET: Offset of a Label** When the assembler comes across the OFFSET operator along with a label, it first computes the 16-bit displacement (also called asoffset

interchangeably) of the particular label, and replaces the string 'OFFSET LABEL' by the computed displacement.

- **ORG: Origin** The ORG directive directs the assembler to start the memory allotment for the particular segment, block or code from the declared Addressing the ORG statement while starting the assembly process for a module, the assembler initializes a location counter to keep track of the allotted addresses for the module. If the ORG statement is not written in the program, the location counter is initialized to 0000. If an ORG 200H statement is present at the starting of the code segment of that module, then the code will start from 200H Addressing codesegment)
- **PROC: Procedure** The PROC directive marks the start of a named procedure in the statement.
- **PTR: Pointer** The pointer operator is used to declare the type of a label, variable or memory operand. The operator PTR is prefixed by either BYTE or WORD. If theprefix is BYTE, then the particular label, variable or memory operand is treated as an 8-bit quantity, while if WORD is the prefix, then it is treated as a 16- bit quantity.

Example:

MOV AL, BYTE PTR [SI]; Moves content of memory location addressed by SI (8-bit) to AL

- **SEG: Segment of a Label** The SEG operator is used to decide the segment Addressof the label, variable, or procedure and substitutes the segment base Address in place of 'SEG label'. The example given below explain the use of SEGoperator.

  Example MOV AX, SEG ARRAY; This statement moves the segment address
- **SEGMENT: Logical Segment** The SEGMENT directive marks the starting of a logical segment. The started segment is also assigned a name, i.e. label, bythis statement. The SEGMENT and ENDS directive must bracket each logical segment of a program.
- **TYPE** The TYPE operator directs the assembler to decide the data type of the specified label and replaces the 'TYPE label' by the decided data type. For the word type variable, the data type is 2, for double word type, it is 4, and for byte type, it is 1. Suppose, the STRING is a word array. Theinstruction
- **GLOBAL** The labels, variables, constants or procedures declared GLOBAL may be used by other modules of the program. Once a variable is declared GLOBAL, it can be used by any module in the program. The following statement declares the procedure ROUTINE as a global label.

ROUTINE PROC GLOBAL

## 1.6 ASSEMBLY LANGUAGE PROGRAMMING

**ALP for SUBition of two 16-bitnumbers**

```
ASSUME CS:CODE,DS:          CODE, SS:
CODE
CODE SEGMENT ORG 1000H MOV SI, 1200H MOV DI, 1300H MOV CL,
        00H MOV AX, [SI] MOV BX, [SI+02]
        SUB AX, BX
        JNC         L1
        MOV CL, 01H L1: MOV [DI], AX
        MOV [DI+02], CL
        INT 3H

CODE ENDS
END
```

**ALP for Subtraction of two 16-bit numbers**

```
ASSUME CS: CODE, DS: DATA,SS:

CODE
CODE SEGMENT
        ORG 1000H
        MOV SI, 1200H
        MOV DI, 1300H
        MOV CL, 00H
        MOV AX, [SI]
        MOV BX, [SI+2]
        SUB AX, BX
        JNC          L1
        MOV CL, 01H L1: MOV [DI], AX
        MOV [DI+2], CL
        INT 3H
CODE ENDS
END
```

**ALP for Multiplication of two 16-bit numbers**

```
ASSUME CS:  CODE,  DS:  DATA,
SS: CODE
CODE SEGMENT
        MOV SI, 1200H
        MOV DI, 1300H
        MOV AX,[SI]
        MOV BX,[SI+02]
        MUL BX
        MOV [DI], AX
        MOV [DI+02], DX
        INT 3H
CODE ENDS
END
```

**ALP for 2's Complement 16-bit number**

```
ASSUME CS: CODE, DS:
CODE, SS: CODE
CODE SEGMENT
ORG 1000H
        MOV SI, 1200H
        MOV AX, [SI]
        NOT AX
        INC AX
        MOV [1300], AX
        INT 3H
CODE ENDS
END
```

**ALP for division of 16-bit number with 8-bit number**

```
ASSUME CS: CODE, DS: DATA, SS:
CODE
CODE SEGMENT
        ORG 1000H
        MOV SI, 1200H
        MOV DI, 1300H
        MOV AX, [SI]
        MOV BX, [SI+02]
        DIVBX
```

```
        MOV [DI], AX
        MOV [DI+02], DX

        INT 3H
    CODE ENDS
    END
```

**ALP for Searching Smallest number**

```
    ASSUME CS: CODE, DS: CODE CODE SEGMENT
            ORG 1000H
        START: MOV SI, 1200H
            MOV DI, 1300H
            MOV CL, [SI]
            INC SI
            MOV AL, [SI]
            DECCL,
        AGAIN: INCSI
            CMP AL, [SI]
            JNC AHEAD / JC AHEAD MOV AL, [SI]
        AHEAD: DEC CL
            JNZ AGAIN
            MOV [DI], AL
            INT 3H
        CODE ENDS
    END
```

**ALP for Find and replace**

```
ASSUME CS: CODE, DS: CODE

CODE SEGMENT
    ORG 1000H
        MOV CX, 0005H
         MOV DI, 1200H
    MOV AL, 45H
    MOV BL, 57H
    CLD
    REPNE SCASB
    DEC DI
    MOV [DI], BL
    INT 3
CODE ENDS

END
```

### 1.7 MODULAR PROGRAMMING:

Complex programs are divided into many parts and each sub-part are known as modules. All the modules perform a well-defined task. Formulation of computer code using a module is known as **modular programming**.

The reason for breaking a program into small parts are

- Modules are easier tounderstand.
- Different modules can be assigned to differentprogrammers.
- The debugging and testing can be done in a more orderlyfashion.
- Documentation can be more easilyunderstood.
- Modifications may belocalized.

Most assembler languages are used in modularization process in three ways such as,

1. Allow data to be structured so that they can be accessed by several modules
2. Provide for procedures or subroutines
3. Permit sections of code,known asmacros.

To perform modular programming,the following tasks must be performed.

- Linking andrelocation
- StackOperation
- Procedures
- Interruptprocess

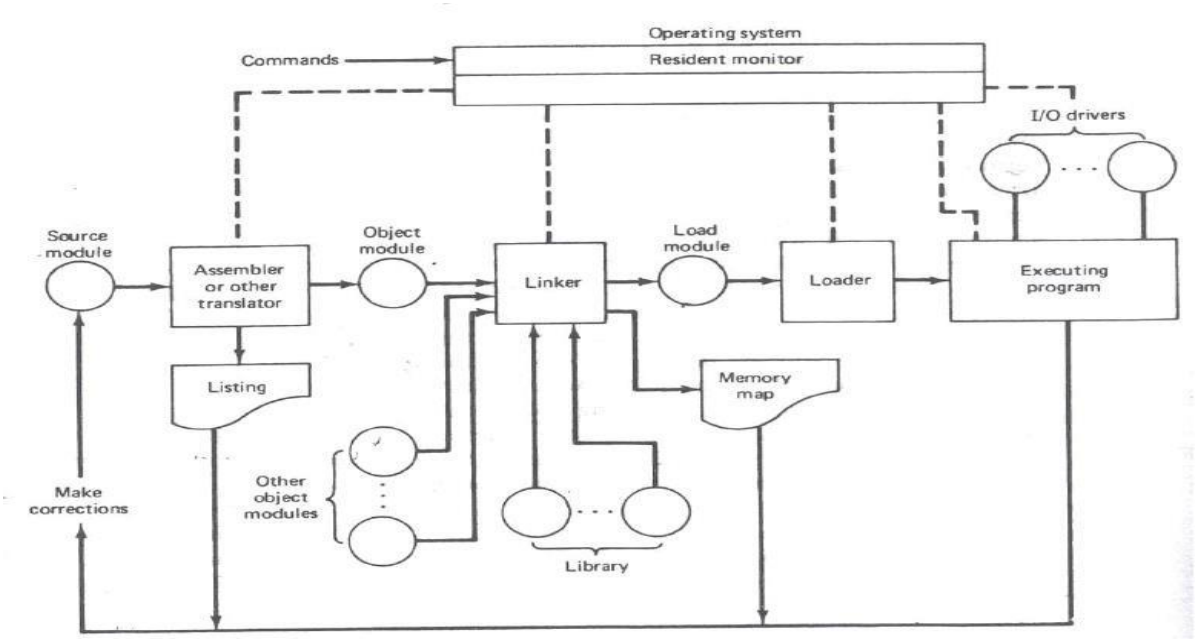
**Linking And Relocation**

The assembly language program can be written with an ordinary text editors such as word star,editor etc.The assembly language program text is an input to the assembler.The assembler translates assembly language statements to their binary equivalent known as object code. During assembling process assembler checks for syntax errors and displays them before giving object code module.The object code module contains the information about whwre the program or module to be loaded in memory.If the object module is to be linked with other separate modules then it containg additional linkage information.

At link time,separately assembled modules are combined into one single load module by the linker.The linker also SUBs any required initialization or finalization code to allow the OS to start the program running and to return control to OS after the program is completed.

At load time,the program loader copies the program into computer main memory and at execution time ,the program execution begins.

If the modules in the program they are assembled separately,then there is one main module and other modules.This main module has the first instruction to be executed and it is terminated by an END statement with entry point satisfied.Other modules are terminated by a END statement with nooperand.




Creation and Execution of Assembly language program

**[Source:Yu-Cheng Liu, Glenn A.Gibson, "Microcomputer Systems: The 8086 / 8088 Family - Architecture, Programming and Design", Second Edition, Prentice Hall of India,2007]**

**Segment combination**

In addition to the linker commands, the assembler provides a means of regulating the way segments in different object modules are organized by the linker.

Segments with same name are joined together by using the modifiers attached to the SEGMENT directives. SEGMENT directive may have the form

**Segment name SEGMENT Combination-type**

where the combine-type indicates how the segment is to be located within the load module. Segments that have different names cannot be combined and segments with the same name but no combine-type will cause a linker error.

The possible combine-types are:

**PUBLIC –** If the segments in different modules have the same name and combine- type PUBLIC, then they are concatenated into a single element in the load module. The ordering in the concatenation is specified by the linker command.

**COMMON –** If the segments in different object modules have the same name and the combine-type is COMMON, then they are overlaid so that they have the same starting address. The length of the common segment is that of the longest segment being overlaid.

**STACK –** If segments in different object modules have the same name and the combine type

STACK, then they become one segment whose length is the sum of the lengths of the individually specified segments. In effect, they are combined to form one large stack

**AT –** The AT combine-type is followed by an expression that evaluates to a constant which is to be the segment address. It allows the user to specify the exact location of the segment in memory.

**MEMORY** – This combine-type causes the segment to be placed at the last of the load module. If more than one segment with the MEMORY combine-type is being linked, only the first one will be treated as having the MEMORY combine type; the others will be overlaid as if they had COMMONcombine-type.

```
Source module 1
 DATA      SEGMENT   COMMON
 DATA          ENDS
 CODE          SEGMENT   PUBLIC
 CODE          ENDS

Source module 2
DATA       SEGMENT   COMMON
           .
           .
DATA       ENDS
CODE       SEGMENT   PUBLIC
```

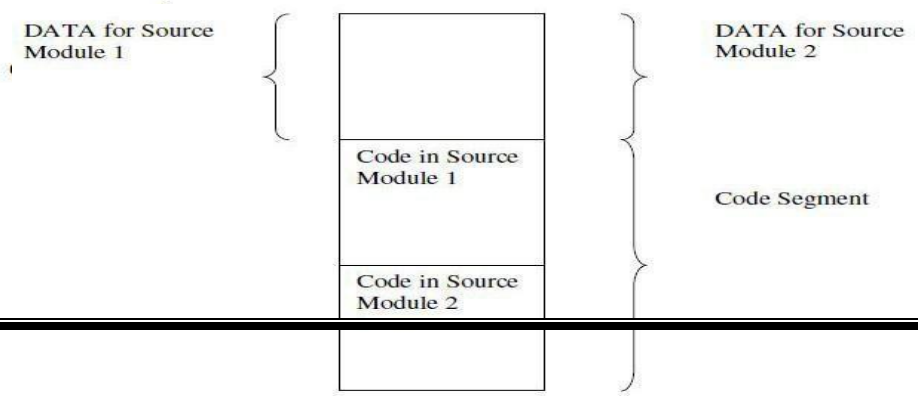
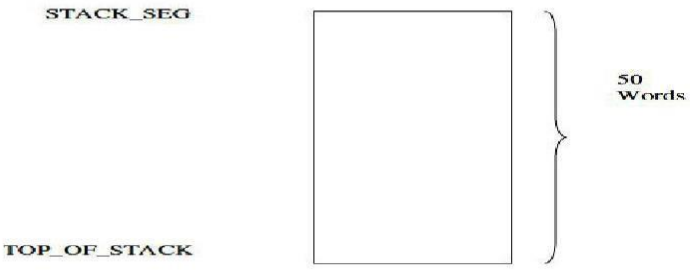


**Fig.** Segment combinations resulting from the PUBLIC and Common Combination types

**[Source:Yu-Cheng Liu, Glenn A.Gibson, "Microcomputer Systems: The 8086 / 8088 Family - Architecture, Programming and Design", Second Edition, Prentice Hall of India,2007]**

```
Source module 1
STACK_SEG  SEGMENT STACK
       DW    20 DUP (?)
       TOP-OF_STACK LABEL WORD
STACK_SEG ENDS
       END

Source module 2
       STACK_SEG          SEGMENT    STACK
        DW    30 DUP (?)
STACK_SEG          ENDS
                .
                .
END
```




**[Source:Yu-Cheng Liu, Glenn A.Gibson, "Microcomputer Systems: The 8086 / 8088 Family - Architecture, Programming and Design", Second Edition, Prentice Hall of India,2007]**

3.**Fig. Formation of a stack from two segments [Source:** Yu-Cheng Liu, Glenn A.Gibson, "Microcomputer Systems: The 8086 / 8088 Family - Architecture, Programming and Design", Second Edition, Prentice Hall of India,2007]

**Access to External Identifiers**

If an identifier is defined in an object module, then it is said to be a *local* (or *internal) identifier* relative to the module. If it is not defined in the module but is defined in one of the other modules being linked, then it is referred to as an *external* (or *global*) *identifier* relative to the module. Two lists are implemented by the EXTRN and PUBLIC directives, which have the forms:

```
EXTRN Identifier: Type..., Identifier: Type
  and
PUBLIC       Identifier,..., Identifier
```

where the identifiers are the variables and labels being declared or as being available to other modules.

The assembler must know the type of all external identifiers before it can generate the proper machine code, a type specifier must be associated with each identifier in an EXTRN statement. For a variable the type may be BYTE, WORD, or DWORD and for a label it may be NEAR or FAR.

One of the primary tasks of the linker is to verify that every identifier appearing in an EXTRN statement is matched by one in a PUBLIC statement. If this is not the case, then there will be an undefined reference and a linker error will occur. The offsets for the local identifier will be inserted by the assembler, but the offsets for the external identifiers and all segment addresses must be inserted by the linking process. The offsets associated with all external references can be

assigned once all of the object modules have been found and their external symbol tables have been examined. The assignment of the segment addresses is called *relocation* and is done after the linking process has determined exactly where each segment is to be put inmemory.

## STACKS

The stack is a block of memory that may be used for temporarily storing the contents of the registers inside the CPU. It is a top-down data structure whose elements are accessed using the stack pointer (SP) which gets decremented by two as we store a data word into the stack and gets incremented by two as we retrieve a data word from the stack back to the CPU register.

The process of storing the data in the stack is called **'pushing into'** the stack and the reverse process of transferring the data back from the stack to the CPU register is known as **'popping off'** the stack. The stack is essentially *Last-In-First -Out* (LIFO) data segment. This means that the data which is pushed into the stack last will be on top of stack and will be popped off the stack first.

The stack pointer is a 16-bit register that contains the offsetAddressof the memory location in the stack segment. The stack segment, like any other segment, may have a memory block of a maximum of 64 Kbytes locations, and thus may overlap with any other segments. Stack Segment register (SS) contains the baseAddressof the stack segment in the memory.

The Stack Segment register (SS) and Stack pointer register (SP) togetherAddressthe stack-top as explained below:

SS ⟹ 5000H
SP ⟹ 2050H

If the stack top points to a memory location 52050H, it means that the location 52050H is already occupied with the previously pushed data. The next 16 bit push operation will decrement the stack pointer by two, so that it will point to the new stack-top 5204EH and the decremented contents of SP will be 204EH. This location will now be occupied by the recently pushed data.

Thus for a selected value of SS, the maximum value of SP=FFFFH and the segment can have maximum of 64K locations. If the SP starts with an initial value of FFFFH, it will be decremented by two whenever a 16-bit data is pushed onto the stack. After successive push operations, when the stack pointer contains 0000H, any attempt to further push the data to the stack will result in stackoverflow.

After a procedure is called using the CALL instruction, the IP is incremented to the next instruction. Then the contents of IP, CS and flag register are pushed automatically to the stack. The control is then transferred to the specified Addressing the CALL instruction i.e. starting Address of the procedure. Then the procedure isexecuted.

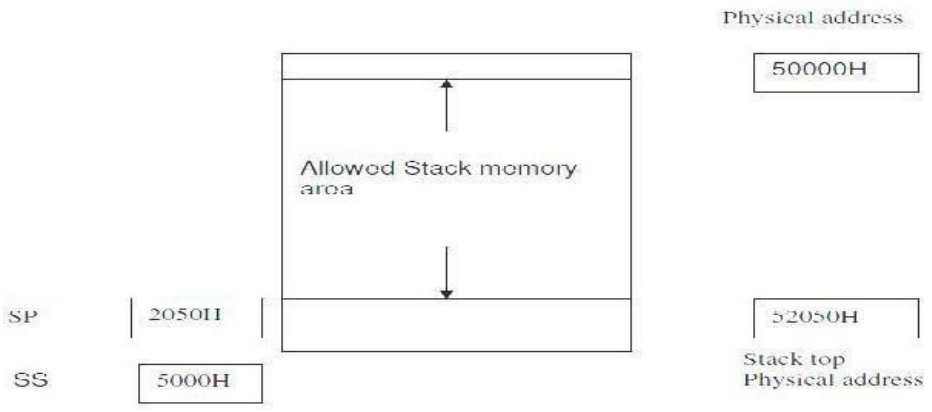



**Fig. Stack –top Address calculation**

[Source:Yu-Cheng Liu, Glenn A.Gibson, "Microcomputer Systems: The 8086 / 8088 Family - Architecture, Programming and Design", Second Edition, Prentice Hall of India,2007]

## PROCEDURES

A procedure is a set of code that can be branched to and returned from in such a way that the code is as if it were inserted at the point from which it is branched to. The branch to procedure is referred to as the ***call,*** and the corresponding branch back is known as the ***return*****.** The return is always made to the instruction immediately following the call regardless of where the call is located.

### Calls, Returns, and Procedure Definitions

The CALL instruction not only branches to the indicated address, but also pushes the Return Address onto the stack. The RET instruction simply pops the return Address from the stack. The registers used by the procedure need to be stored before their contents are changed, and then restored just before their contents are changed, and then restored just before the procedure is excited.

A CALL may be direct or indirect and intrasegment or intersegment. If the CALL is intersegment, the return must be intersegment. Intersegment call must push both (IP) and(CS) onto the stack. The return must correspondingly pop two words from the stack. In the case of intrasegment call, only the contents of IP will be saved and retrieved when call and return instructions areused.

Procedures are used in the source code by placing a statement of the form at the beginning of the procedure

**Procedure name PROC Attribute**
**--------**
**----------**
**Procedure name ENDP**

The attribute that can be used will be either NEAR or FAR. If the attribute is NEAR, the RET instruction will only pop a word into the IP register, but if it is FAR, it will also pop a word into the CS register.

A procedure may be in:

1. The same code segment as the statement that calls it.

2. A code segment that is different from the one containing the statement that calls it, but in the same source module as the callingstatement.

3. A different source module and segment from the callingstatement.

In the first case, the attribute could be NEAR provided that all calls are in the same code segment as the procedure. For the latter two cases the attribute must be FAR. If the procedure is given a FAR attribute, then all calls to it must be intersegment calls even if the call is from the same code segment. For the third case, the procedure name must be declared in EXTRN and PUBLIC statements.

**Saving and Restoring Registers**

When both the calling program and procedure share the same set of registers, it is necessary to save the registers when entering a procedure, and restore them before returning to the calling program.

```
MSK PROC NEAR
    PUSH AX
    PUSH BX
    PUSHCX
    POP CX
    POP BX
    POP AX
    RET
MSK ENDP
```

**Procedure Communication**

There are two general types of procedures, those operate on the same set of data and those that may process a different set of data each time they are called.

If a procedure is in the same source module as the calling program, then the procedure can refer to the variables directly.

When the procedure is in a separate source module it can still refer to the source module directly provided that the calling program contains the directive PUBLIC ARY, COUNT, SUM

EXTRN ARY: WORD, COUNT: WORD, SUM: WORD

**Recursive Procedures**

When a procedure is called within another procedure it called recursive procedure. To make sure that the procedure does not modify itself, each call must store its set of parameters, registers, and all temporary results in a different place in memory
*Eg. Recursive procedure to compute the factorial*

**MACROS**

**Disadvantages of Procedure**

1. Linkage associated withthem.

2. It sometimes requires more code to program the linkage than is needed to perform the task. If this is the case, a procedure may not save memory and execution time is considerablyincreased.

**Macros** is needed for providing the programming ease of a procedure while avoiding the linkage. Macro is a segment of code that needs to be written only once but whose basic structure can be caused to be repeated several times within a source module by placing a single statement at the point of each reference.

A macro is unlike a procedure in that the machine instructions are repeated each time the macro is referenced. Therefore, no memory is saved, but programming time is conserved (no linkage is required) and some degree of modularity is achieved. The code that is to be repeated is called the prototype code. The prototype code along with the statements for referencing and terminating is called the macro definition.

Once a macro is defined, it can be inserted at various points in the program by using macro calls. When a macro call is encountered by the assembler, the assembler replaces the call with the macro code. Insertion of the macro code by the assembler for a macro call is referred to as a macro expansion.. During a macro expansion, the first actual parameter replaces the first dummy parameter in the prototype code, the second actual parameter replaces the second dummy parameter, and soon.

A macro call has the form

%Macro name (Actual parameter list) with the actual parameters being separated by commas.
%MULTIPLY (CX, VAR, XYZ[BX]

Above macro call results in following set of codes.

```
PUSH DX
PUSH AX
MOV AX,CX
IMUL VAR
MOV XYZ[BX],AX
POP AX
POPDX
```

It is possible to define a macro with no dummy parameters, but in this case the call must not include any parameters. Consider a macro for pushing the contents at beginning of a procedure.

Macro definition consists of

```
%*DEFINE(SAVEREG)
( PUSH AX
PUSH BX
PUSH CX
PUSH DX
)
```

This macro is called using the statement

```
%SAVEREG
```

The above macro can be called at the beginning of the each procedure to save the register contents. A similar macro could be used to restore the register contents at the end of each procedure.

```
%*DEFINE (RESTORE)
( POP DX
POP CX
POP BX
POP AX
)
```

**Local Labels**

Consider a macro called ABSOL which makes use of labels. This macro is used to replace the operand by its absolute value.

```
%*DEFINE (ABSOL(OPER))
 ( CMP %OPER, 0
JGE NEXT
NEG %OPER
 %NEXT: NOP)
```

When the macro ABSOL is called for the first time, the label NEXT will appear in the program and, therefore it becomes defined. Any subsequent call will cause NEXT to be redefined. This will result in an error during assembly process because NEXT has been associated with more than one location. One solution to this problem would be to have NEXT replaced by a dummy parameter for the label. This would require the programmer to keep track of dummy parametersused.

One solution to this problem is the use of **Local Labels.** Local labels are special labels that will have suffixes that get incremented each time the macros are called. These suffixes are two digit numbers that gets incremented by one starting from zero. Labels can be declared as local label by attaching a prefix **Local. Local List of Local labels** at the end of first statement in the macrodefinition.

**Nested Macros**

It is possible for a macro call to appear within a macro definition. This is referred to as **Macro nesting.** The limitation of nested macros is that all macros included in the definition of a given macro must be defined before the given macro is called.

**INTERRUPTS AND INTERRUPT ROUTINES**

**Interrupt and its Need**:

The microprocessors allow normal program execution to be interrupted in order to carry out a specific task/work. The processor can be interrupted in the following ways

- by an external signal generated by aperipheral,
- by an internal signal generated by a special instruction in theprogram,
- by an internal signal generated due to an exceptional condition which occurs while executing an instruction. (For example, in 8086 processor, divide by zero is an exceptional condition which initiates type 0 interrupt and such an interrupt is also calledexecution).

The process of interrupting the normal program execution to carry out a specific task/work is referred to as **interrupt.** The interrupt is initiated by a signal generated by an external device or by a signal generated internal to the processor.

When a microprocessor receives an interrupt signal it stops executing current normal program, save the status (or content) of various registers (IP, CS and flag registers in case of 8086) in stack and then the processor executes a subroutine/procedure in order to perform the specific task/work requested by the interrupt. The subroutine/procedure that is executed in response to an interrupt is also called **Interrupt Service Subroutine (ISR).** At the end of ISR,the

stored status of registers in stack is restored to respective registers, and the processor resumes the normal program execution from the point {instruction) where it was interrupted.

The external interrupts are used to implement interrupt driven data transfer scheme. The interrupts generated by special instructions are called software interrupts and they are used to implement system services/calls (or monitor services/calls). The system/monitor services are procedures developed by system designer for various operations and stored in memory. The user can call these services through software interrupts. The interrupts generated by exceptional conditions are used to implement error conditions in the system.

**Interrupt Driven Data Transfer Scheme**

The interrupts are useful for efficient data transfer between processor and peripheral.

When a peripheral is ready for data transfer, it interrupts the processor by sending an appropriate signal. Upon receiving an interrupt signal, the processor suspends the current program execution, save the status in stack and executes an ISR to perform the data transfer between the peripheral andprocessor.

At the end of ISR the processor status is restored from stack and processor resume its normal program execution. This type of data transfer scheme is called **interrupt driven data transfer scheme.**

The data transfer between the processor and peripheral devices can be implemented either by polling technique or by interrupt method. In polling technique, the processor has to periodically poll or check the status/readiness of the device and can perform data transfer only when the device 'is ready. In polling technique the processor time is wasted, because the processor has to suspend its work and check the status of the device in predefined intervals.

If the device interrupts the processor to initiate a data transfer whenever it is ready then the processor time is effectively utilized because the processor need not suspend its work and check the status of the device in predefined intervals.

For an example, consider the data transfer from a keyboard to the processor. Normally a keyboard has to be checked by the processor once in every 10 milliseconds for a key press. Therefore once in every 10 milliseconds the processor has to suspend its work and then check the keyboard for a valid key code. Alternatively, the keyboard can interrupt the processor, whenever a key is pressed and a valid key code is generated. In this way the processor need not waste its time to check the keyboard once in every 10 milliseconds.

**Classification of Interrupts**

In general the interrupts can be classified in the following three ways:

1. Hardware and softwareinterrupts
2. Vectored and Non Vectoredinterrupt:
3. Maskable and Non Maskableinterrupts.

The interrupts initiated by external hardware by sending an appropriate signal to the interrupt pin of the processor is called *hardware interrupt*. The 8086 processor has two interrupt pins **INTR and NMI**. The interrupts initiated by applying appropriate signal to these pins are called hardware interrupts of 8086.

The *software interrupts* are program instructions. These instructions are inserted at desired locations in a program. While running a program, if software interrupt instruction is encountered then the processor initiates an interrupt. The 8086 processor has 256 types of software interrupts. The software interrupt instruction is INT n, where n is the type number in the range 0 to255.

When an interrupt signal is accepted by the processor, if the program control automatically branches to a specific Address(called vector Address) then the interrupt is called *vectored interrupt*. The automatic branching to vector Address is predefined by the manufacturer of processors. (In these vector Addresses the interrupt service subroutines (ISR) are stored). In *non-vectored interrupts* the interrupting device should supply the Address of the ISR to be executed in response to the interrupt. All the 8086 interrupts are vectored interrupts. The vector Address for an 8086 interrupt is obtained from a vector table implemented in the first 1kb memory space (00000h to 03FFFh).

The processor has the facility for accepting or rejecting hardware interrupts. Programming the processor to reject an interrupt is referred to as *masking or disabling* and programming the processor to accept an interrupt is referred to as *unmasking or enabling*. In 8086 the interrupt flag (IF) can be set to one to unmask or enable all hardware interrupts and IF is cleared to zero to mask or disable a hardware interrupts except NMI.

The interrupts whose request can be either accepted or rejected by the processor are called *maskable interrupts*. The interrupts whose request has to be definitely accepted (or cannot be rejected) by the processor are called *non-maskable interrupts*. Whenever a request is made by non-maskable interrupt, the processor has to definitely accept that request and service that interrupt by suspending its current program and executing an ISR. In 8086 processor all the hardware interrupts initiated through INTR pin are maskable by clearing interrupt flag (IF). The interrupt initiated through NMI pin and all software interrupts arenon-maskable.

**Sources of Interrupts in 8086**

An interrupt in 8086 can come from one of the following three sources.

1. One source is from an external signal applied to NMI or INTR input pin of the processor. The interrupts initiated by applying appropriate signals to these input pins are called hardware interrupts.
2. A second source of an interrupt is execution of the interrupt instruction "INT n", where n is the type number. The interrupts initiated by "INT n" instructions are called softwareinterrupts.

3. The third source of an interrupt is from some condition produced in the 8086 by the execution of an instruction. An example of this type of interrupt is divide by zero interrupt. Program execution will be automatically interrupted if you attempt to divide an operand by zero. Such conditional interrupts are also known as exceptions.

**Interrupts of 8086**

The 8086 microprocessor has 256 types of interrupts. INTEL has assigned a type number to each interrupt. The type numbers are in the range of 0 to 255. The 8086 processor has dual facility of initiating these 256 interrupts. The interrupts can be initiated either by executing "INT n" instruction where n is the type number or the interrupt can be initiated by sending an appropriate signal to INTR input pin of the processor.

In general, the interrupts can be classified as following: 1. External Hardware Interrupts
2.Non Maskable Interrupts
3. Software Interrupts 4.
Internal Interrupts 5.Reset

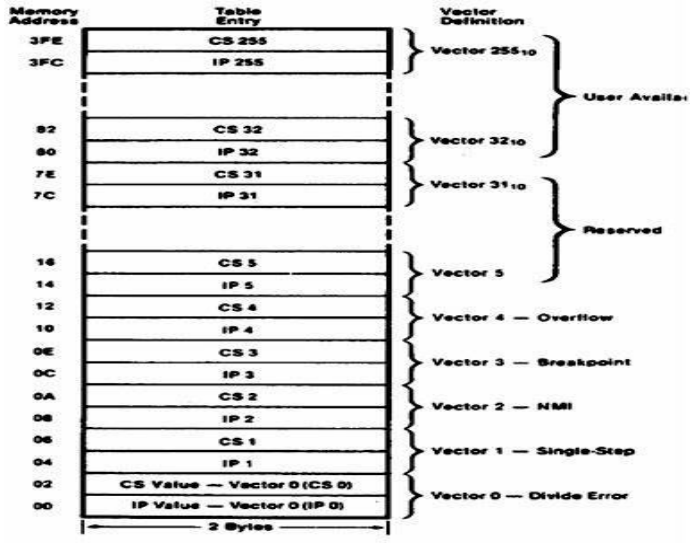
**Interrupts Priority**

Hardware, software and internal interrupts are serviced on a priority basis.
Each interrupt is given a different priority level by assigning a type number. Type 0 identiifes highest priority and type 255 identifies the lowest priority interrupt.
The interrupt priority follows.

- Reset
- Internal interrupts
- Software interrupts
- Non maskable interrupts
- External hardware interrupts

**Interrupt Vector Table**

The first 1Kbyte of memory of 8086 (00000 to003FF) is set aside as a table for storing the starting addresses of Interrupt Service Procedures(ISP).Since 4-bytes are required for storing starting addresses of ISPs, the table can hold 256 Interrupt procedures.
The starting Address of an ISP is often called the Interrupt Vector or Interrupt Pointer. Therefore the table is referred as Interrupt Vector Table. In this table, IP value is put in as low word of the vector & CS is put in highvector.




**[Source: Advanced Microprocessors and Microcontrollers by A.K Ray & K.M. Bhurchandi]**

The operation of an interrupt on the 8086 microprocessor

1. External interface sends an interrupt signal, to the Interrupt Request (INTR) pin, or an internal interruptoccurs.
2. The CPU finishes the present instruction and sends Interrupt Acknowledge (INTA) to hardware interface.
3. The interrupt type N is sent to the Central Processing Unit (CPU) via the data bus from hardware interface.
4. The contents of the flag registers are pushed onto thestack.
5. Both interrupts (IF) and (TF) flags are cleared. This disables the INTR pin and the trap or single stepfeature.
6. The contents of the code segment register (CS) are pushed onto thestack.
7. The contents of the instruction pointer (IP) are pushed onto thestack.
8. The interrupt vector contents are fetched ,from(4*N) and then placed into the IP and from(4*N+2) into the CS so that the next instruction executes at the interrupt service procedure addressed by the interruptvector.
9. While returning from the interrupt service routine by the Interrupt return (IRET) instruction, The IP,CS and Flag registers are popped from the stack and return their state prior to theinterrupt.

| Mnemonic | Meaning | Format | Operation | Flags Affected |
|---|---|---|---|---|
| CLI | Clear interrupt flag | CLI | 0 → (IF) | IF |
| STI | Set interrupt flag | STI | 1 → (IF) | IF |
| INT n | Type n software interrupt | INT n | (Flags) → ((SP) − 2)<br>0 → TF,IF<br>(CS) → ((SP) − 4)<br>(2 + 4 · n) → (CS)<br>(IP) → ((SP) − 6)<br>(4 · n) → (IP) | TF, IF |
| IRET | Interrupt return | IRET | ((SP)) → (IP)<br>((SP) + 2) → (CS)<br>((SP) + 4) → (Flags)<br>(SP) + 6 → (SP) | All |
| INTO | Interrupt on overflow | INTO | INT 4 steps | TF, IF |
| HLT | Halt | HLT | Wait for an external interrupt or reset to occur | None |
| WAIT | Wait | WAIT | Wait for $\overline{TEST}$ input to go active | None |

8086 Interrupt Instructions. **[Source: Advanced Microprocessors and Microcontrollers by A.K Ray & K.M. Bhurchandi]**

## BYTE AND STRING HANDLING INSTRUCTIONS

The 8086 microprocessor is equipped with special instructions to handle string operations. By string we mean a series of data words or bytes that reside in consecutive memory locations. The string instructions of the 8086 permit a programmer to implement operations such as to move data from one block of memory to a block elsewhere in memory. A second type of operation that is easily performed is to scan a string and data elements stored in memory looking for a specific value.

Other examples are to compare the elements and two strings together in order to determine whether they are the same ordifferent.

**Move String:** MOV SB, MOV SW:

An element of the string specified by the source index (SI) register with respect to the current data segment (DS) register is moved to the location specified by the destination index (DI) register with respect to the current extra segment (ES) register. The move can be performed on a byte (MOV SB) or a word (MOV SW) of data. After the move is complete, the contents of both SI & DI are automatically incremented or decremented by 1 for a byte move and by 2 for a word move. Address pointers SI and DI increment or decrement depends on how the direction flag DF isset.

**Load and store strings:** (LOD SB/LOD SW and STO SB/STO SW)
**LOD SB:** Loads a byte from a string in memory into AL. The Addressing SI is used relative to DS to determine the Address of the memory location of the string element.
(AL) ¬ [(DS) + (SI)] (SI) ¬ (SI) + 1
**LOD SW**: The word string element at the physical Address derived from DS and SI is to be loaded into AX. SI is automatically incremented by 2.
(AX) ¬ [(DS) + (SI)] (SI) ¬ (SI) + 2
**STO SB**:
Stores a byte from AL into a string location in memory. This time the contents of ES and DI are used to form the Address of the storage location in memory.
[(ES) + (DI)] ¬ (AL) (DI) ¬ (DI) + 1
**STO SW:** [(ES) + (DI)] ¬ (AX) (DI) ¬ (DI) + 2
**Repeat String: REP**
The basic string operations must be repeated to process arrays of data. This is done by inserting a repeat prefix before the instruction that is to be repeated. Prefix REP causes the basic string operation to be repeated until the contents of register CX become equal to zero. Each time the instruction is executed, it causes CX to be tested for zero, if CX is found to be nonzero it is decremented by 1 and the basic string operation is repeated.
Example: Clearing a block of memory by repeating STOSB MOV AX, 0

```
MOV ES, AX
MOV DI, A000
MOV CX,
OFCDF
REP STOSBNEXT:
```

The prefixes REPE and REPZ stand for same function. They are meant for use with the CMPS and SCAS instructions. With REPE/REPZ the basic compare or scan operation can be repeated as long as both the contents of CX are not equal to zero and zero flag is1.

REPNE and REPNZ works similarly to REPE/REPZ except that now the operation is repeated as long as CX¹0 and ZF=0. Comparison or scanning is to be performed as long as the string elements are unequal (ZF=0) and the end of the string is not yet found (CX¹0).

**Auto Indexing for String Instructions:**

SI & DI addresses are either automatically incremented or decremented based on the setting of the direction flag DF.
When CLD (Clear Direction Flag) is executed DF=0 permits auto increment by 1. When STD (Set Direction Flag) is executed DF=1 permits auto decrement by 1.