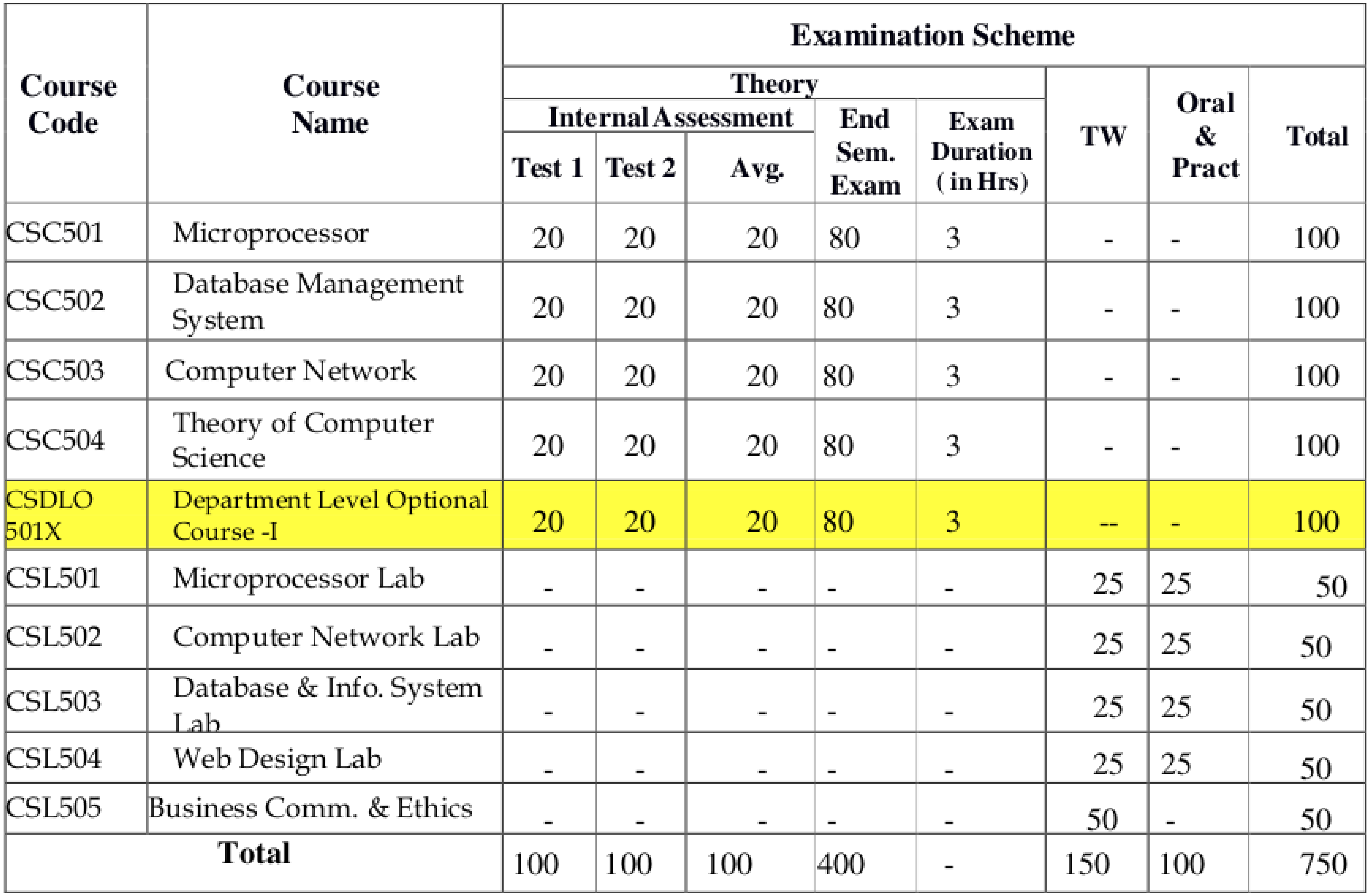
Microprocessor

Examination Scheme



Course Code	Course Name	Examination Scheme							
		Theory					TW	Oral & Pract	Total
		Internal Assessment			End Sem. Exam	Exam Duration (in Hrs)			
		Test 1	Test 2	Avg.					
CSC501	Microprocessor	20	20	20	80	3	-	-	100
CSC502	Database Management System	20	20	20	80	3	-	-	100
CSC503	Computer Network	20	20	20	80	3	-	-	100
CSC504	Theory of Computer Science	20	20	20	80	3	-	-	100
CSDLO 501X	Department Level Optional Course -I	20	20	20	80	3	--	-	100
CSL501	Microprocessor Lab	-	-	-	-	-	25	25	50
CSL502	Computer Network Lab	-	-	-	-	-	25	25	50
CSL503	Database & Info. System Lab	-	-	-	-	-	25	25	50
CSL504	Web Design Lab	-	-	-	-	-	25	25	50
CSL505	Business Comm. & Ethics	-	-	-	-	-	50	-	50
Total		100	100	100	400	-	150	100	750

Module	Detailed Contents	Hrs.
01	**The Intel Microprocessors 8086/8088 Architecture** 8086/8088 CPU Architecture, Programmer‘s Model, Functional Pin Diagram, Memory Segmentation, Banking in 8086, Demultiplexing of Address/Data bus, Study of 8284 Clock Generator, Study of 8288 Bus Controller, Functioning of 8086 in Minimum mode and Maximum mode, Timing diagrams for Read and Write operations in minimum and maximum mode	10
02	**Instruction Set and Programming** Addressing Modes Instruction set – Data Transfer Instructions, String Instructions, Logical Instructions, Arithmetic Instructions, Transfer of Control Instructions, Processor Control Instructions Assembler Directives and Assembly Language Programming, Macros, Procedures Mixed Language Programming with C Language and Assembly Language. Programming based on DOS and BIOS Interrupts (INT 21H, INT 10H)	12
03	**8086 Interrupts** Types of interrupts, Interrupt Service Routine, Interrupt Vector Table, Servicing of Interrupts by 8086 microprocessor, Programmable Interrupt Controller 8259 – Block Diagram, Interfacing the 8259 in single and cascaded mode, Operating modes, programs for 8259 using ICWs and OCWs	08

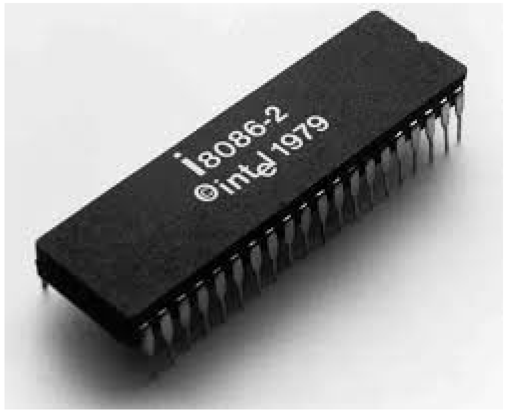
Module	Detailed Contents	Hrs
04	**Peripherals and their interfacing with 8086** 4.1 Memory Interfacing - RAM and ROM Decoding Techniques – Partial and Absolute 4.2 8255-PPI – Block diagram, Functional PIN Diagram, CWR, operating modes, interfacing with 8086. 4.3 8253 PIT - Block diagram, Functional PIN Diagram, CWR, operating modes, interfacing with 8086. 4.4 **8257-DMAC** – Block diagram, Functional PIN Diagram, Register organization, DMA operations and transfer modes	12
05	**Intel 80386DX Processor** Architecture of 80386 microprocessor 80386 registers – General purpose Registers, EFLAGS and Control registers Real mode, Protected mode, virtual 8086 mode 80386 memory management in Protected Mode – Descriptors and selectors, descriptor tables, the memory paging mechanism	06
06	Pentium Processor Pentium Architecture Superscalar Operation, Integer & Floating Point Pipeline Stages, Branch Prediction Logic, Cache Organisation and MESI Model	06

What is Microprocessor

- An integrated circuit that contains all the functions of CPU

- It is a
    - multipurpose
    - programmable
    - clock driven
    - register based

device that takes input and provides output



Microprocessor Applications

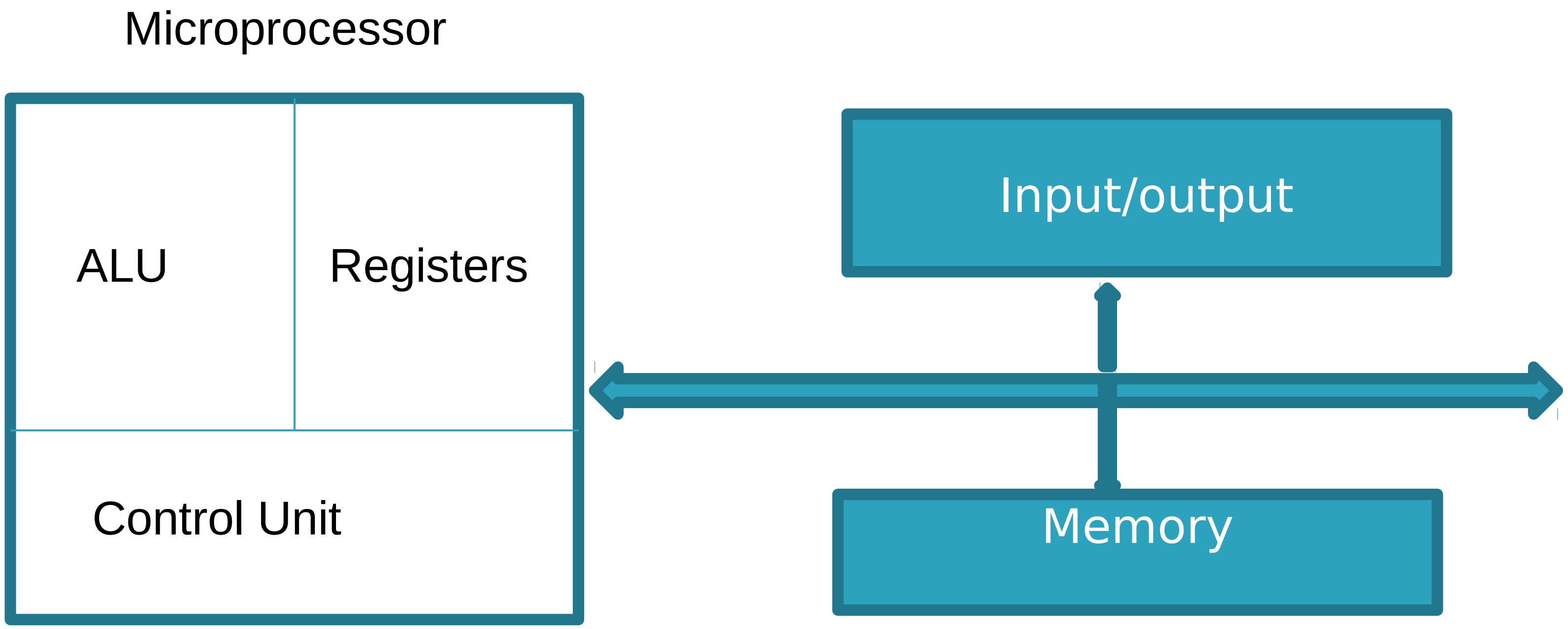
- It is used in everything from the smallest embedded systems to the largest mainframes.

  Example:
- car keys
- toys
- smoke alarms
- DVD video system
- cellular phones

Microprocessor based System

Basic microcomputer includes 3 components:

1) Microprocessor
2) Input/output
3) Memory



Basic terminologies

- Opcode
  - ➢ A binary code, that indicates the operation to be performed
- Operands
  - ➢ The data on which the operation is to be performed as well as the result of an operation is called as operands
- Instruction
  - ➢ The combination of opcode and operand that can be used to instruct a system is called an instruction.

Basic terminologies

- Instruction Set
  - A list of all the instructions that can be issued to a system is called as instruction set of that system.
- Program/subroutine/routine:
  - A set of instructions written in particular sequence to implement a task.
- Bus
  - Data bus (16 bit)
  - Address bus (20 bit)
  - Control bus

8086 Architecture

- The architecture of 8086 is divided into
  - Bus Interface Unit [BIU]
  - Execution Unit [EU]

Registers

Data registers

- There are 4 data registers (AX, BX, CX, DX)

Address register

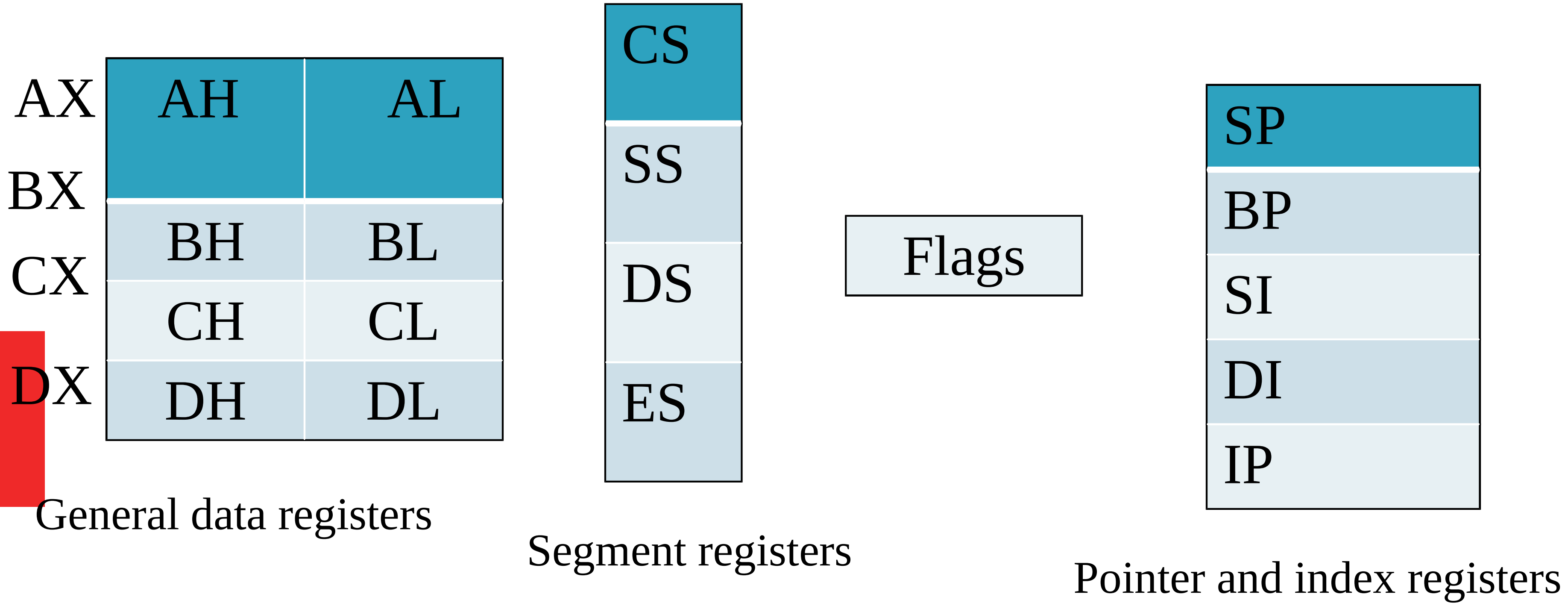
- Segment registers (CS, DS, ES, SS)
- Pointer registers (SP, BP, IP)
- Index registers (SI, DI)

Status register

- Keeps the current status of the processor
- On an IBM PC the status register is called the FLAGS register

Programmer's model of 8086

The register set of 8086 can be categorized into 4 different groups:



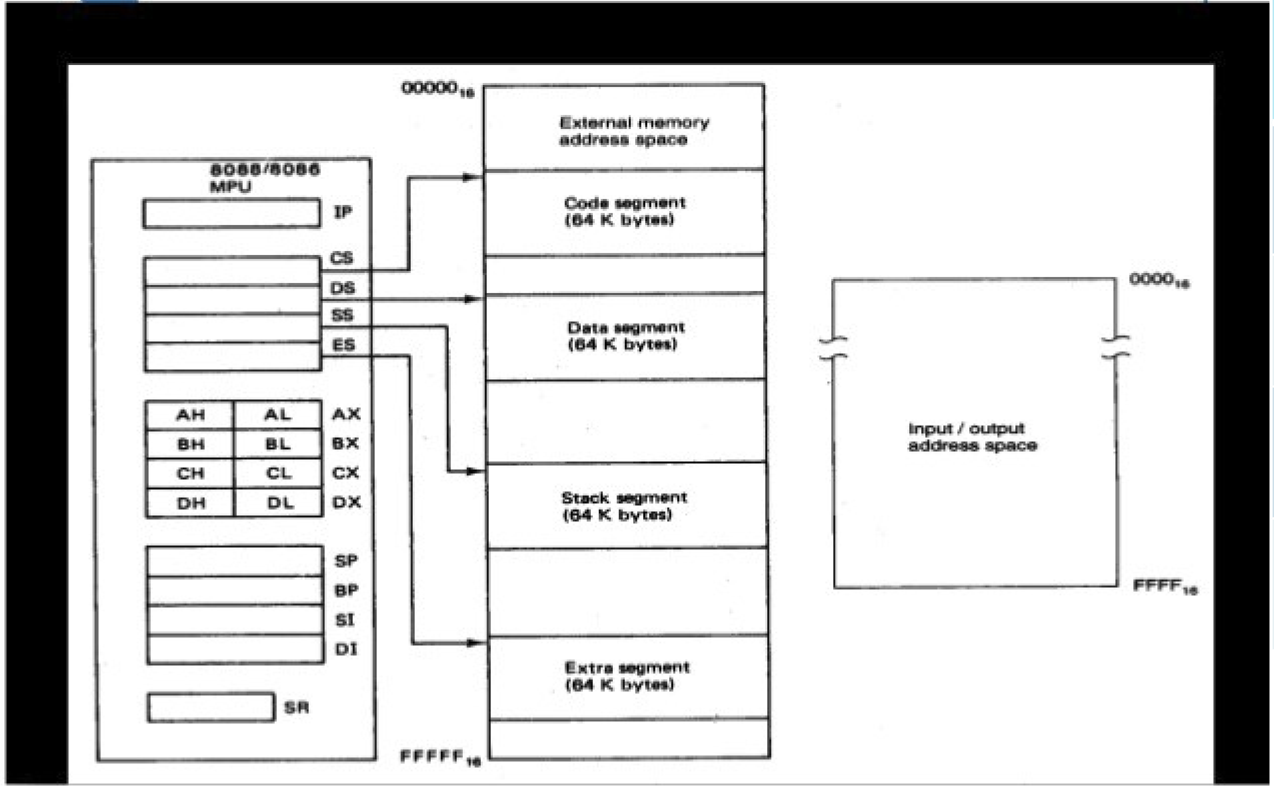
General Purpose registers

- There are four general purpose registers each of 16-bit data (AX, BX, CX, DX)
- 8-Bit Registers and 16 – Bit Registers

8-Bit Registers	16 – Bit Registers
AH	AX
AL	BX
BH	CX
BL	DX
CH	SI
CL	DI
DH	SP
DL	BP

Address Registers

Segmented Memory Representation



00000₁₆
External memory address space
8088/8086 MPU
IP
CS
DS
SS
ES
Code segment (64 K bytes)
Data segment (64 K bytes)
AH AL AX
BH BL BX
CH CL CX
DH DL DX
Stack segment (64 K bytes)
SP
BP
SI
DI
Extra segment (64 K bytes)
SR
FFFFF₁₆
0000₁₆
Input / output address space
FFFF₁₆

- **SP(Stack Pointer)**
- **BP(Base Pointer)**
- **SI(Source Index)**
- **DI(Destination Index)**
- **IP(Instruction Pointer)**

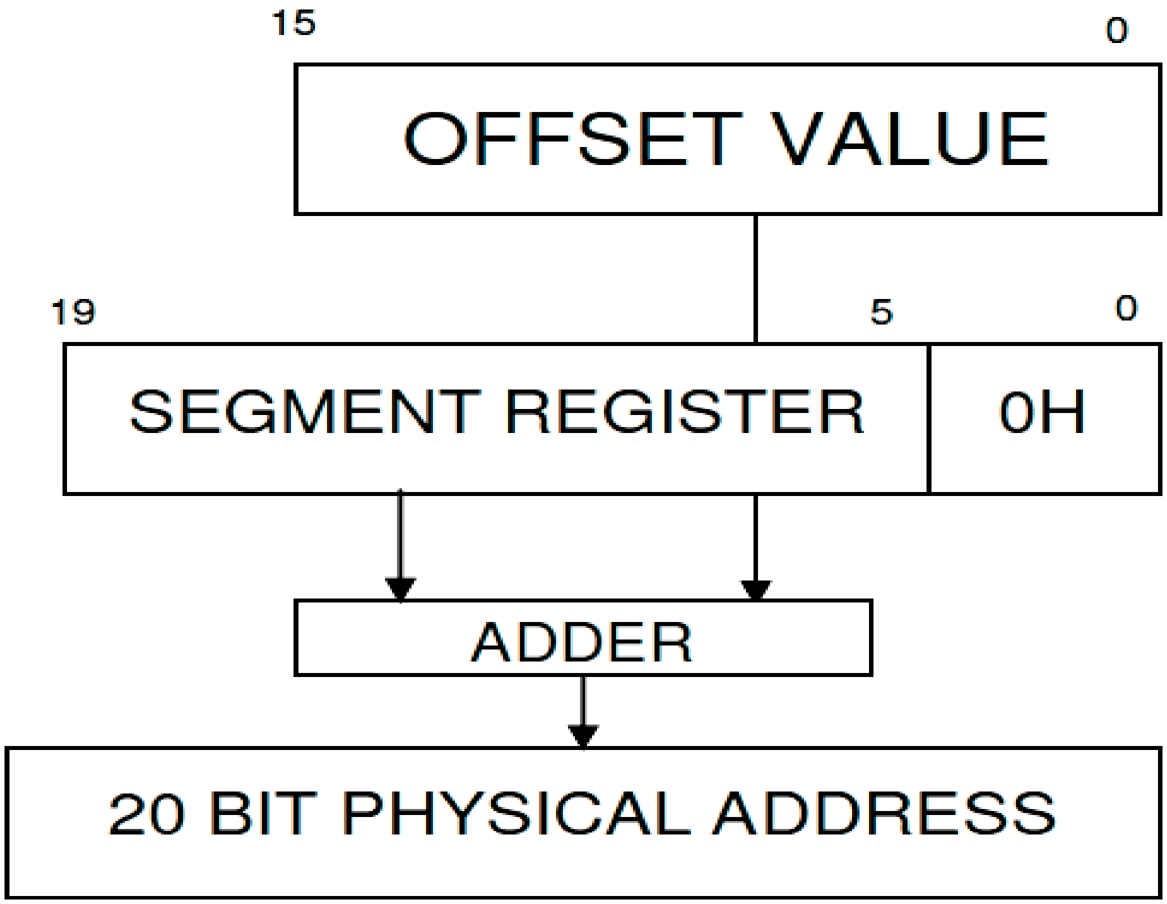
- CS:IP - address of the next instruction to be executed in the program sequence.
- SS:SP - address of the top of the stack, a temporary storage area often automatically used by the computer.

  SP is used for sequential access of the stack.
- SS:BP - a pointer into the stack.

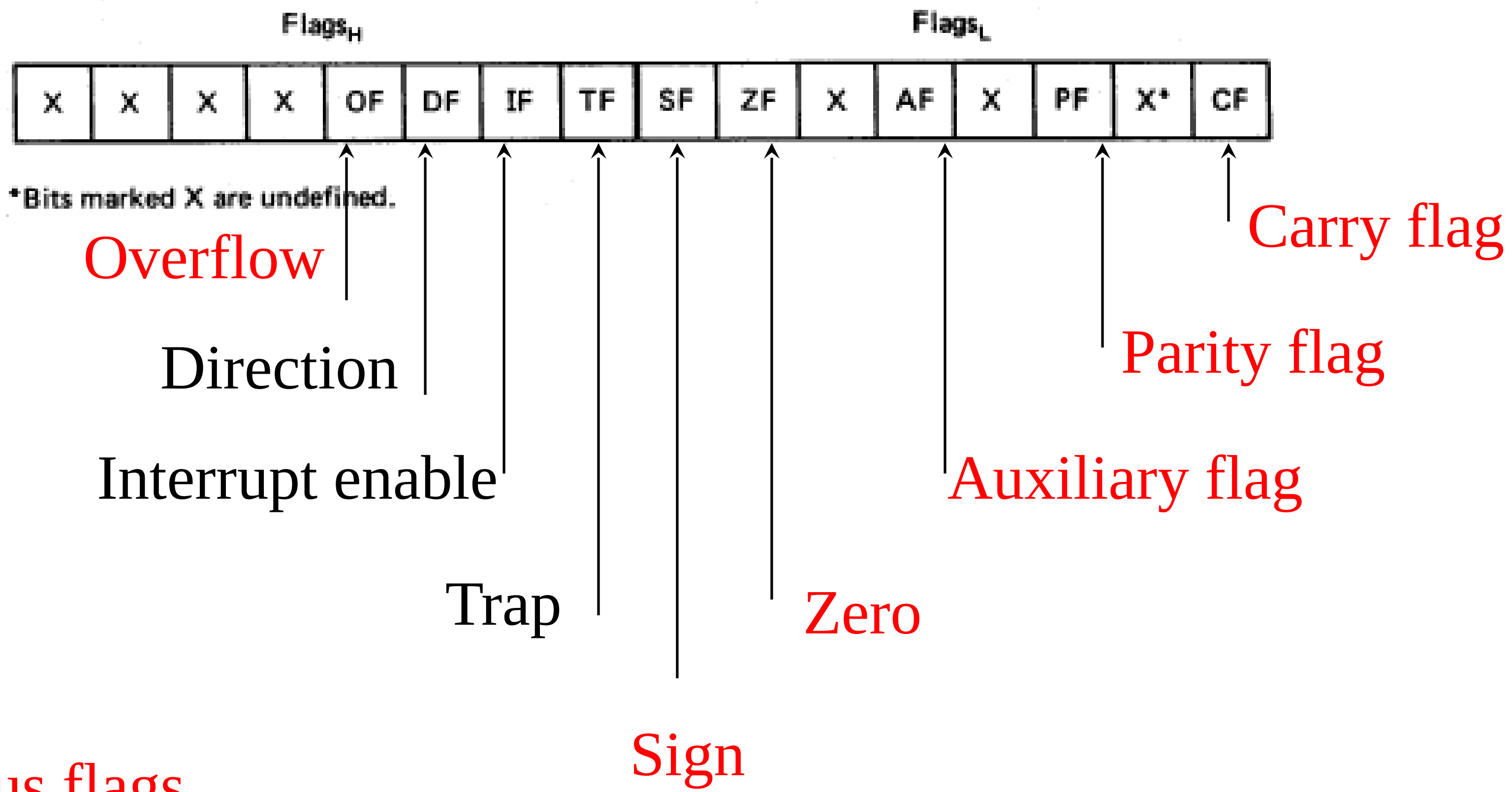
  BP is used for random access of the stack.

- DS:SI - source pointer
- ES:DI - destination pointer for string instructions
- For all other instructions DI register is used with DS segment register.
- DS:SI and ES:DI - general purpose pointers for copying and data processing.

Physical Address



Flags



6 are status flags
3 are control flag

Contents

- Addressing Modes
- Instruction Set of 8086
- Assembler Directives and Assembly Language Programming, Macros, Procedures
- Mixed Language Programming with C Language and Assembly Language
- Programming based on DOS and BIOS Interrupts

Addressing Mode

- Addressing mode indicates a way of locating data or operands

OR

- The different ways in which a processor can access data are called addressing modes

- Depending upon the data types used in the instruction & the memory addressing modes, any instruction may belongs to one or more addressing modes, or some instruction may not belong to any of the addressing mode

- To perform any operation using microprocessor, we have to give corresponding instruction to microprocessor

Addressing Mode

- Instruction is made up of:
  - Operation to be performed
  - Address of source data
  - Address of destination data

- The method by which address of source of data or address of destination of result is given in the instruction is called as Addressing Mode

Addressing Mode

- Immediate Addressing Mode
- Implicit or Implied Addressing Mode
- Register Addressing Mode
- Direct Addressing Mode

Addressing Mode

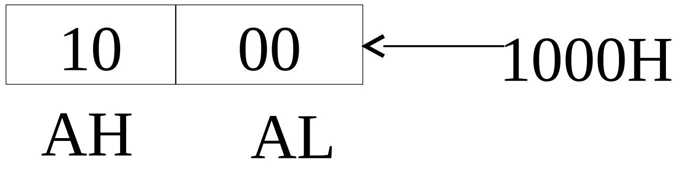
Indirect Addressing Mode:

- Register Indirect Addressing Mode
- Register Relative or Relative Register
  - Base Addressing
  - Index Addressing
- Base and Index Addressing Mode
- Base Relative plus Index Addressing Mode

Immediate Addressing Mode

- If 8/16 bit data required for executing the instruction is given directly along with the instruction then such instruction are called immediate addressing mode instruction
- Example:

  Mov Ax,1000H



- Note:

If any 8/16 bit number given in the instruction is not enclosed in brackets then this number is data

Implicit or Implied Addressing Mode

- If the address of source data as well as address of the destination of result both are fixed then there is no need to give any operand along with the instruction such instructions are called as Implicit addressing mode
- Example:
  - STC
  - CLD

Register Addressing Mode

- If 8/16 bit data required for executing the instruction is present in 8/16 bit register & the name of the this register containing data is given directly along with the instruction then such instruction are called register direct addressing mode instruction

- Example:

  Mov CL,DH

$$\begin{tikzcd}
\boxed{75H} & \boxed{75H} \arrow[l] \\
CL & DH
\end{tikzcd}$$

- Note: If the name of the register given in the instruction is not enclosed in brackets, then this register contains data

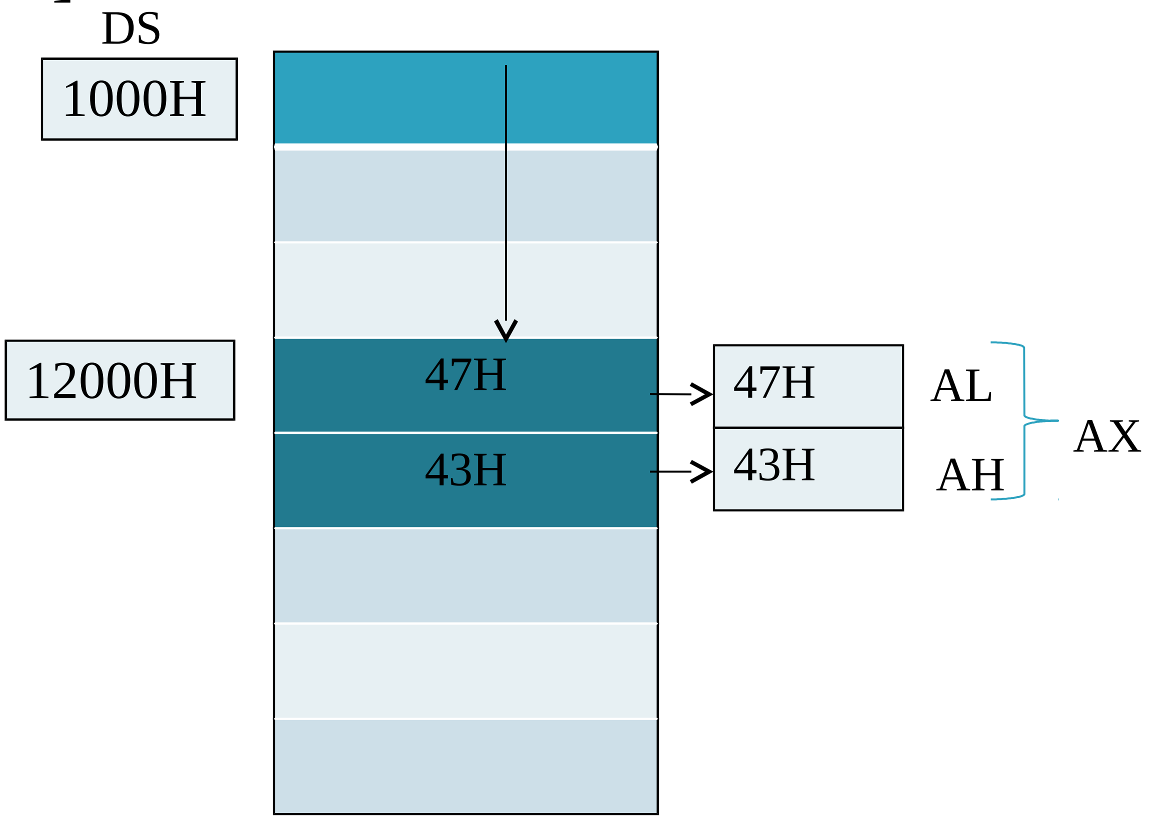
Direct Addressing Mode

- If 8/16 bit data required for executing the instruction is present in memory location and effective address of this memory location is given directly along with the instruction, then such instructions are called Direct addressing mode
- Note: If any16 bit number given in the instruction is enclosed in bracket then this number is effective address and not data

Direct Addressing Mode

- Example:

  Mov AX,[2000H]



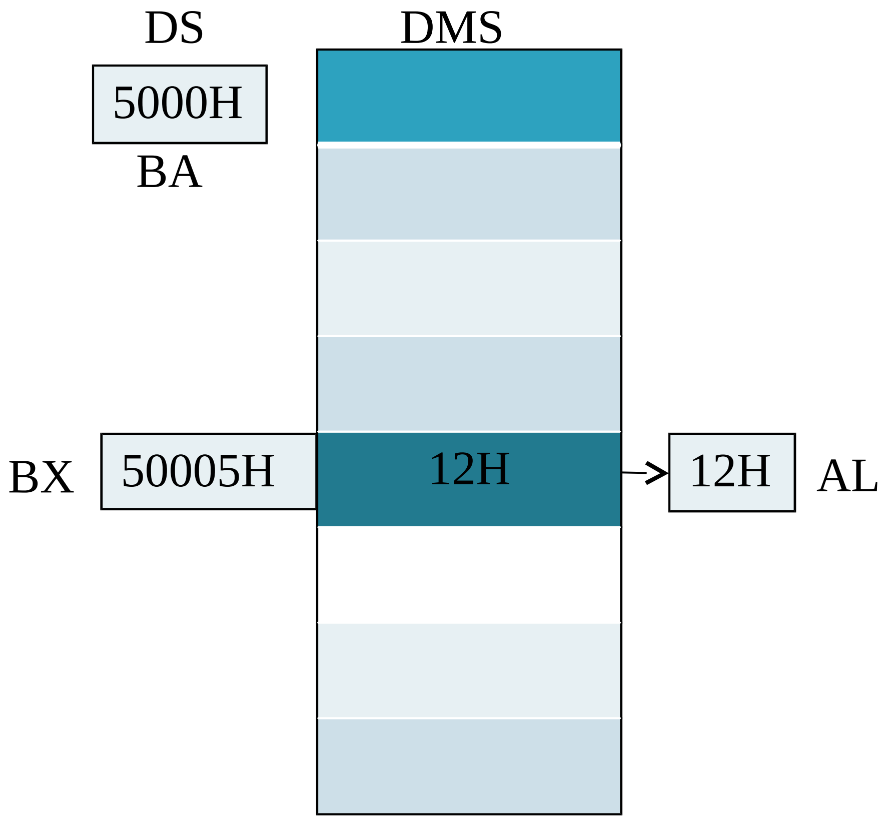
Register Indirect Addressing Mode

- If 8/16 bit data required for executing the instruction is present in memory location and 16-bit effective address of this memory location is present in register BX or SI OR DI and the name of this register is given along with the instruction. Such instructions are called Register Indirect addressing mode
- EA=[BX]/[SI]/[DI]
- Note : As BX/SI/DI will contain effective address, so the name of corresponding register is enclosed in brackets

Register Indirect Addressing Mode

- Example:

  Mov AL,[BX]
- Assume BX=5005



Register Relative Or Relative Register

Register Relative Addressing mode:

- EA=[(Base reg)/(Index reg) + 8/16 bit disp]
- Register relative addressing mode can be further divided into 2 types:
  a) Base Addressing Mode
  b) Index Addressing Mode

Base Addressing Mode

- If EA is obtained by adding 16 bit no. of given base reg BX/BP with 8/16 bit displacement then it is called base addressing mode

  EA =[(Base reg)+ 8/16bit disp]

  EA=[(BX/BP)+8/16 bit disp]

- The effective address is computed as

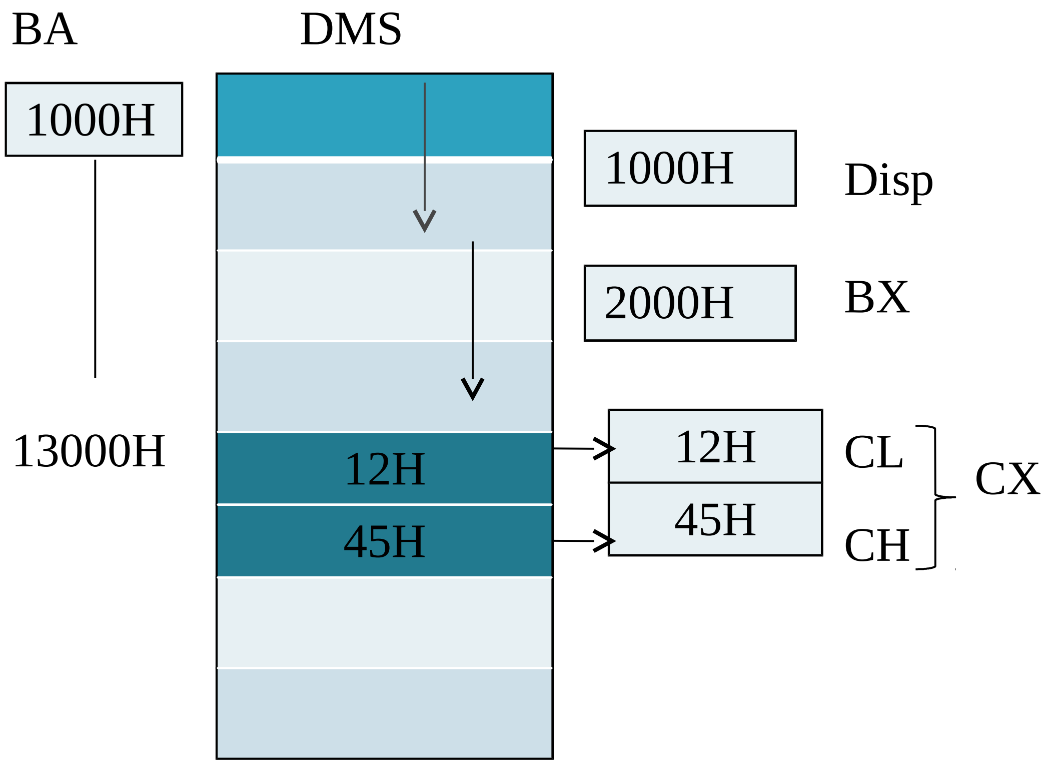
  10*DS+BX/BP+ disp

Base Addressing Mode

- Mov CX,1000H[BX]
- If BX=2000H

+disp= 1000H

EA= 3000H



- NOTE:
  - If BP is used instead of BX, then SS register is used instead of DS register.

Index Addressing Mode

- If EA of memory location is obtained by adding 16 bit no. given index register SI/DI with displacement then it is called Index addressing mode

  EA=[(Index reg)+8/16 bit displacement]
  EA=[(SI/DI)+ 8/16 bit displacement]

- The effective address is computed as
  10*DS+SI/DI+disp

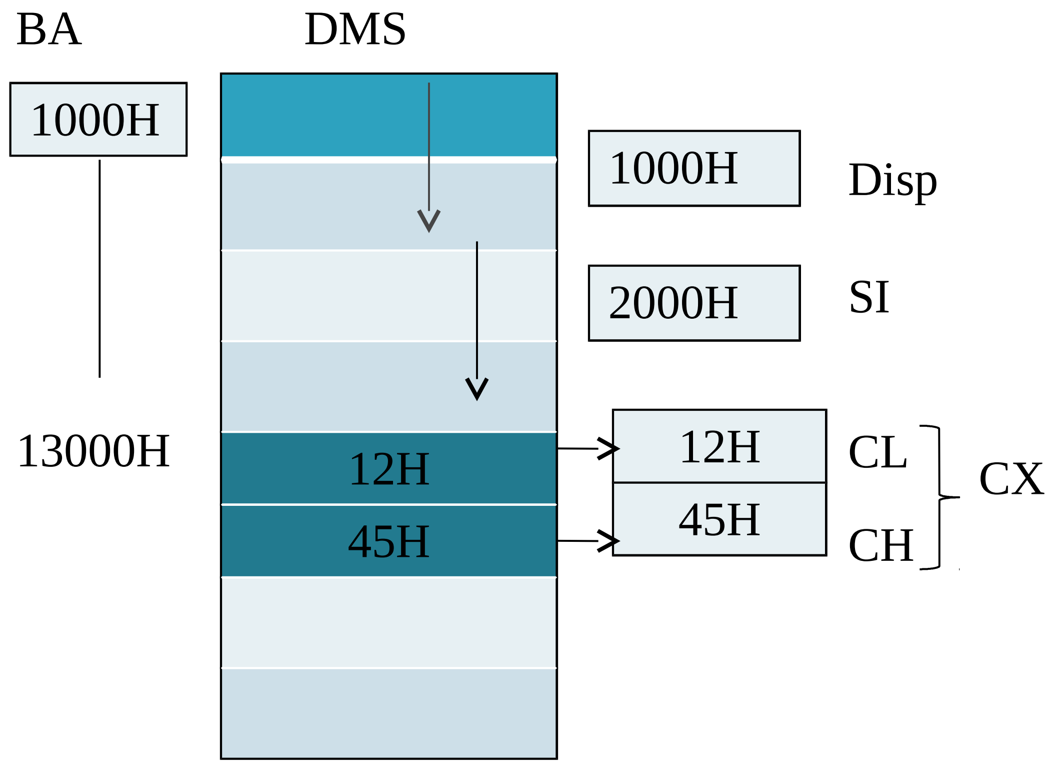
Example:
Mov AH, 7000H[SI]

Index Addressing Mode

- Mov CX,1000H[SI]
- If SI=2000H

+disp= 1000H

EA=3000H



Base-Index Addressing Mode

- If 8/16 bit data required to execute the instruction is present in memory location and EA of this memory location is obtained by adding 2 components:
  - Base Reg
  - Index Reg

EA=[(Base reg + Index reg]

EA=[(BX/BP) + (SI/DI)]

such instruction are called Base-Index Addressing mode instructions

- The effective address is computed as

10H*DS+[BX]+[SI]

Base-Index Addressing Mode

- Based and Indexed-

  Effective Address=BX +SI

  Effective Address=BX +DI

  Segment register used is DS

  or

  Effective Address=BP +SI

  Effective Address=BP +DI

  Segment register used is SS

  Ex:

  MOV AX,[BX+SI]

  i.e.

  AL <- {Shifted DS+[BX+SI]}

  AH <-{Shifted DS+[BX+SI+1]}

Base-Index Addressing Mode

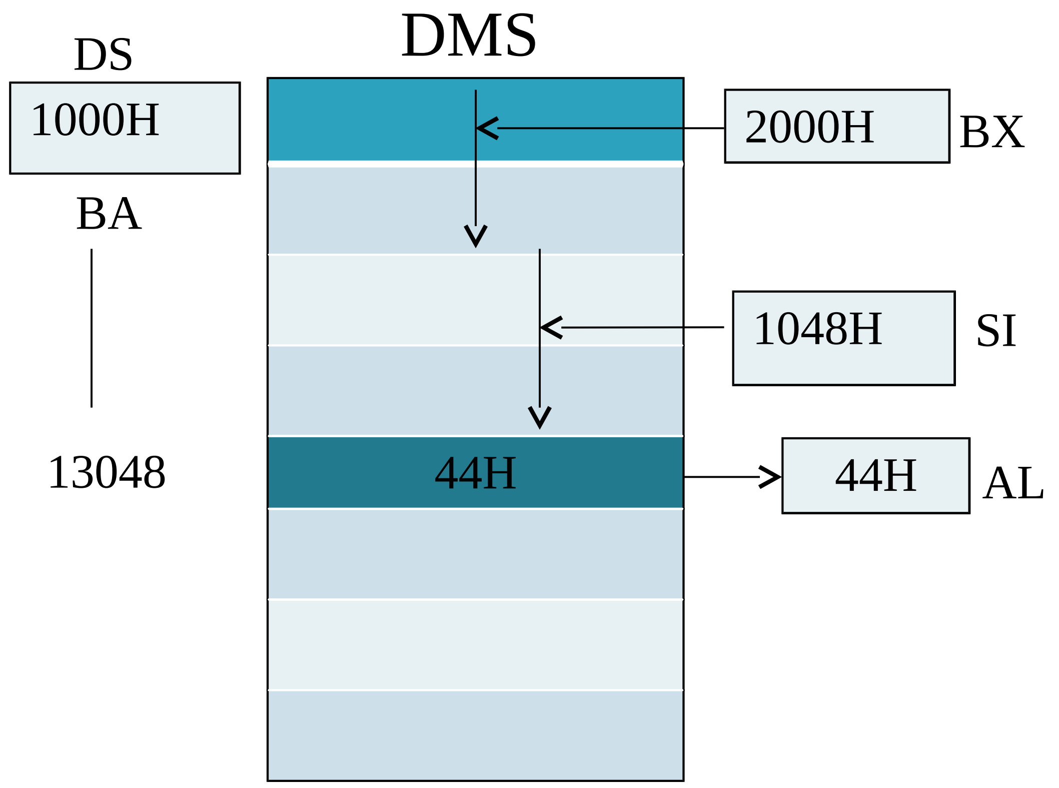
- Example:

Mov AL,[BX][SI]

If BX=2000H

+SI=1048H

EA=3048H



Base Relative plus Index Addressing Mode

- If 8/16 bit data required to execute the instruction is present in memory location and EA of this memory location is obtained by adding three components:
  - Base reg
  - Index reg
  - 8/16 bit displacement
- It is also called as Relative Base-Index Addressing Mode
- The effective address of data is computed as

  10H*DS+[BX]+[SI]+disp

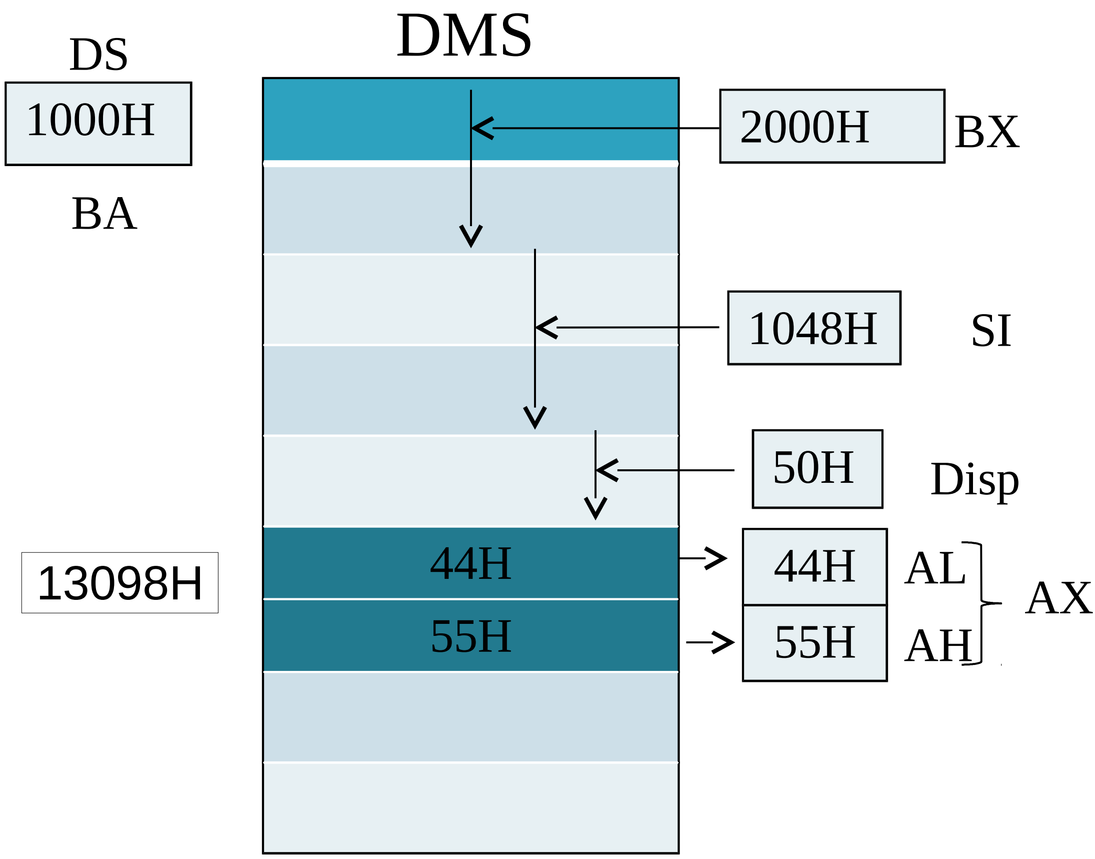
Base Relative plus Index Addressing Mode

- Example:

Mov AX,[BX+SI]50H

If BX=2000H
 +SI= 1048H
 Disp= 50H

 EA=3098H



Instruction Set of 8086

The instruction set are categorized into the following types

- Data copy or transfer instructions
- Arithmetic and logical instructions
- Shift and rotate instructions
- Machine control instructions
- Program transfer instruction
- Iteation control instruction
- Interrupt instructions
- String instructions

Data copy or transfer instructions

- It is used to transfer data from source to destination
- All the store, move, load, exchange, input and output instructions come in this category

Data copy or transfer instructions

Mov destination, source

- Copies a word or a byte of data from some source to a destination
- Destination can be a register or a memory location.
- Source can be a register, a memory location or an immediate number

Operation:

Destination $\longleftarrow$ Source

Data copy or transfer instructions

Example:

Instruction	Description of instruction
Mov AX,BX	Content of BX register is copied to AX register
Mov Bx,3456H	3456H is immediate data copied to BX register
Mov AL,[3000H]	The data from memory location 3000H is copied to AL register
Mov AL, [SI]	Content of memory location which is stored in SI is copied into AL register

Data copy or transfer instructions

Push Source

- The PUSH instruction decrements the stack pointer by two after each execution of this instruction
- This instruction pushes/copies a word from source to the location in the stack segment
- The source of the word can be a general purpose register, a segment register, or memory
- The source must be a WORD (16 bits)

Data copy or transfer instructions

- Operation:

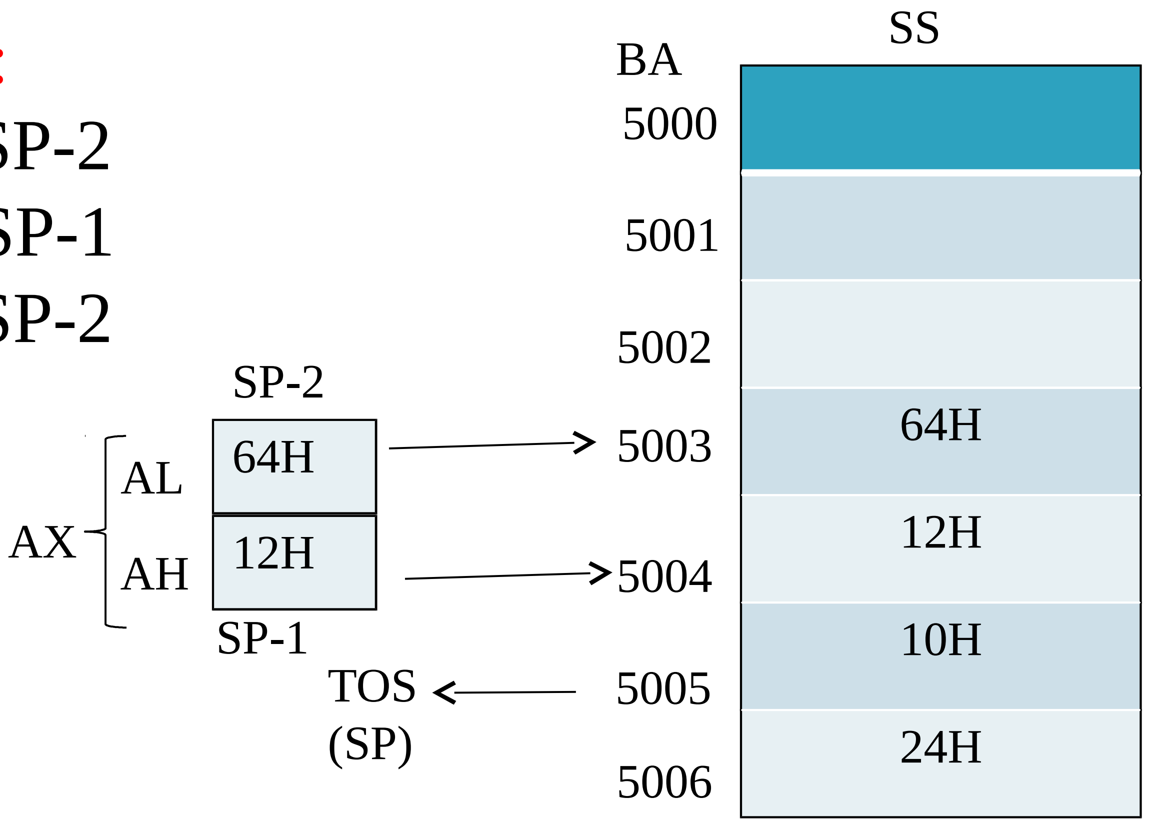
  SP <- SP-2

  MSB <- SP-1

  LSB <- SP-2

Example:

PUSH AX



TOS -> The memory location where a word was most recently stored in stack segment is called as " Top of the stack"

Data copy or transfer instructions

Instruction	Description of instructions
PUSH BX	Copy the content of BH register to higher address of stack(SP-1) and BL register to lower address of stack (SP-2). Decrement SP by 2
PUSH DS	Copy the higher byte of DS to higher address of stack(SP-1) and lower byte of DS to lower address of stack (SP-2). Decrement SP by 2
PUSH [5000H]	Copy the content of 5000H to higher address of stack(SP-1) and 5001H to lower address of stack (SP-2). Decrement SP by 2
PUSH AL	Not allowed must push a word

Data copy or transfer instructions

POP Destination

- POP a word from the stack into the given destination and increment the stack pointer by 2
- The destination can be a general purpose register, a segment register (except CS) or a memory location
- The destination must be a WORD (16 bits)
- Note: MOV, PUSH,POP are the ONLY instructions that use the segment register as operand (except cs)

Data copy or transfer instructions

Operation:

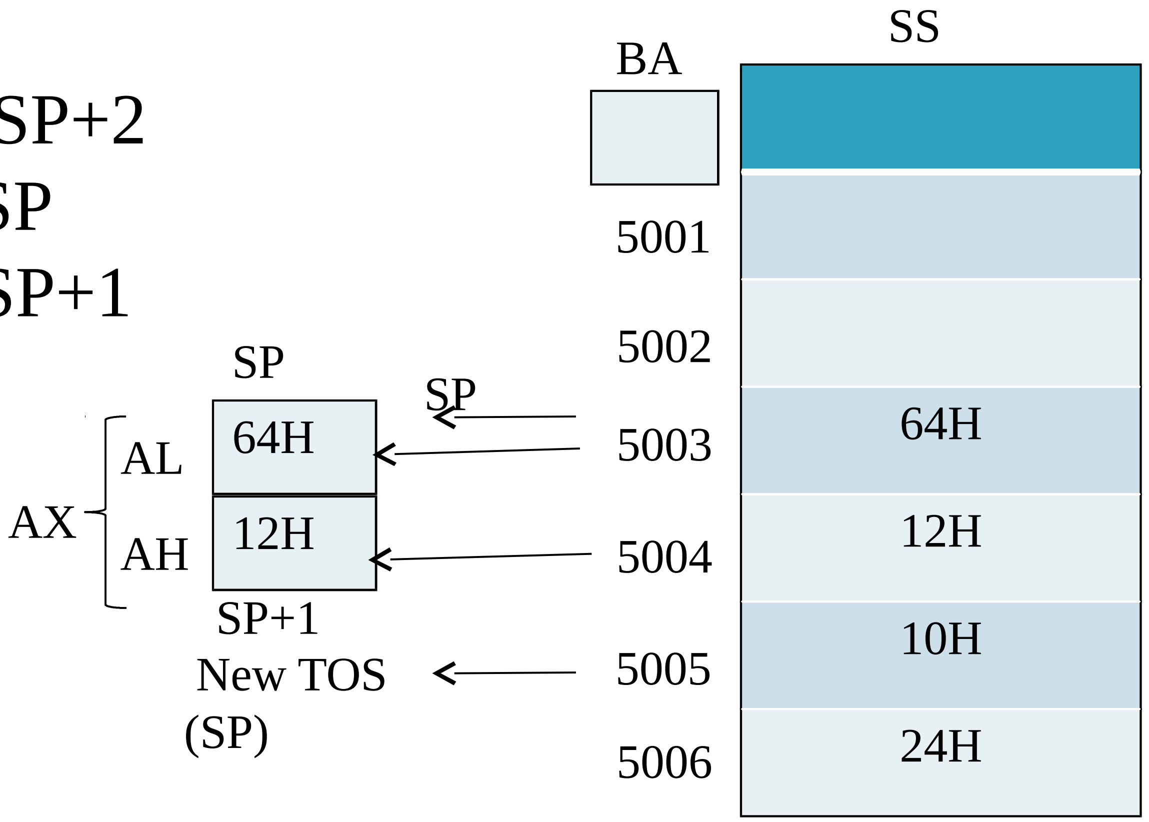
SP <- SP+2

LSB <-SP

MSB<-SP+1

Example:

POP AX



Data copy or transfer instructions

XCHG Destination, Source

- Exchange a byte / word between the source and the destination specified in the instruction
- This instruction cannot directly exchange the content of two memory locations
- The source and operand must both be word or they must both be byte

Data copy or transfer instructions

Operation

Destination $\Longleftrightarrow$ Source

Example:

- XCHG AX, BX
- XCHG AX,[7000H]

Data copy or transfer instructions

XLAT

- It replaces a byte in the AL register with a byte from a data segment/lookup table in memory
- The offset of the starting address of look up table must be loaded in BX
- To point desired byte in lookup table the XLAT instruction adds the byte in AL to offset of start of table in BX

Data copy or transfer instructions

Operation:

- AL <- DS:[BX+AL]

- If DS=2314H
- BX =2134H
- AL =02H then

```
23140H
 2134H
   02H
25276H
______________
              AL<- [25276H]
```

Data copy or transfer instructions

Simple input output port transfer instructions:

- There two addressing modes of input and output port
  - Direct Addressing Mode Or
  - Fixed Port Addressing Mode

  - Register Indirect Addressing Mode Or
  - Variable Addressing Mode

Data copy or transfer instructions

IN accumulator, port address

- Copy a byte or word from a specified port to accumulator
- The address of the port can be specified in the instruction directly or indirectly
- For the fixed port type the 8 bit address of a port is specified directly in the instruction
- For variable port type the 16 bit address of a port is specified in DX register only

Data copy or transfer instructions

Operation:

AL<- [Port] for byte

AL<- [port] and AH<- [port+1] for a word

Example:

- In fixed port type

```
IN AL, 80H
IN AX, 80H
```

- In variable port type

```
Mov DX, 8000H
    IN AL, DX
```

Data copy or transfer instructions

OUT port, accumulator

- Copy a byte or a word from accumulator specified port
- The address of the port can be specified in the instruction directly or indirectly
- For the fixed port type the 8 bit address of a port is specified directly in the instruction
- For variable port type the 16 bit address of a port is specified in DX register only

Data copy or transfer instructions

Operation:

[port]<- AL for byte

[port]<- AL and [port+1] <- AH for word

Example:

In fixed port type

```
OUT 80H, AL
OUT 80H, AX
```

In variable port type

```
MOV DX,6000H
OUT DX, AL
```

Data copy or transfer instructions

SPECIAL ADDRESS TRANSFER INSTRUCTIONS:

LEA Destination, Source

- Load effective address of the operands in specified register

LEA Register, Source

Load EA of Source into 16 bit register Register

Operation:

16 bit register $\leftarrow$ Effective Address

Example:

```
        LEA  BX,Sum
```

Data copy or transfer instructions

LDS/LES R16, M32

- Copies the content from two memory location into the register specified.
- It then copies a word from the next two memory locations into the DS/ES register

Operation:

For LDS instruction

16 bit register ← [memory address]

DS ← [memory address+2]

For LES instruction

16 bit register ← [memory address]

ES ← [memory address+2]

Data copy or transfer instructions

Example:

- LDS BX,[1234H]

Copy the content of memory location 1234H in BL, the content of 1235H in BH and the content of 1236H and 1237H in DS register

- LES BX, [1234H]

Copy the content of memory location 1234H in BL, the content of 1235H in BH and the content of 1236H and 1237H in ES register

Data copy or transfer instructions

Flag Transfer instructions

LAHF

- Load AH with the 8 LSB’s of the flag register

Operation:

AH ← 8 LSB’S of the flag resister

Data copy or transfer instructions

SAHF

- Store AH resister to 8 LSB’S of flag register

Operation:

8 LSB’S of the flag resister ← AH

Data copy or transfer instructions

PUSHF

- Push 16 bit number of flag register into two memory location TOS i.e. TOS-1,TOS-2

POPF

- POP or remove/pull 16 bit number from TOS &TOS+1 & store into flag register

Arithmetic And Logical Instructions

- These type of instruction are used to perform arithmetic and logical , increment, decrement, compare and scan operation

Arithmetic Instruction:

- These instruction performs the arithmetic operation, like addition, subtraction, multiplication and division along with the respective ASCII and Decimal adjust instructions

Arithmetic Instructions

ADD destination, source

- The add instruction adds the contents of the source operand to the destination operand
- The source may be an immediate number, a register, or a memory location
- The destination may be a register or memory location
- The source and destination must be of the same type and cannot both be memory locations
- Example: ADD DX, BX

Arithmetic Instructions

Operation:

- Destination ← destination + Source

Example:

Instructions	Description of instructions
ADD AL, 74H	Add the immediate number 74H to AL and store the result in AL
ADD AX, BX	Add the contents of BX with AX and store the result in AX
ADD AL, [6000H]	Add the content of memory location 6000H to AL and store the result in AL
ADD AL, [SI]	Add the content of memory location pointed by SI with AL and store result in AL

Arithmetic Instructions

ADC destination, source

- This instruction adds the contents of the source operand to the destination operand with carry
- The result is stored in a destination operand
- The source may be an immediate number, a register, or a memory location
- The destination may be a register or memory location

- The source and destination must be of the same type and cannot both be memory locations
- The addition of two memory locations along with carry is not possible
- The segment registers cannot be used. The memory uses DS as segment register.

Arithmetic Instructions

Operation:

- Destination ← destination + Source + CF

Example:

Instruction	Description of instruction
ADC AX,1234H	Add the immediate number 1234H to AX with carry and store result in AX
ADC AX,BX	Add the content of BX to AX with carry and store result in AX

Arithmetic Instructions

SUB Destination, Source

- This instruction subtract the source operand from the destination operand and the result is stored in destination operands
- Source operand can be a register, memory location or immediate data
- Destination operands may be a register or a memory location
- The source and destination must be of the same type and cannot both be memory locations

Arithmetic Instructions

Operation:

- Destination ← destination – Source

Example:

Instruction	Description of instruction
SUB AL,74H	Subtract the immediate number 74H from AL and stores the result in AL
SUB AX, BX	Subtract the contents of BX from AX and stores the result in AL
SUB AL,[6000H]	Subtract the content of memory location 6000H from AL and stores the result in AL
SUB AL,[SI]	Subtract the content of memory location pointed by SI from AL and stores result in AL

Arithmetic Instructions

SBB Destination, Source

- This instruction subtract the source operand and the borrow from the destination operand and the result is stored in destination operands
- Source operand can be a register, memory location or immediate data
- Destination operands may be a register or a memory location
- The source and destination must be of the same type and cannot both be memory locations

Arithmetic Instructions

Operation:

- Destination ← destination – Source – CF

Example:

Instruction	Description of instruction
SBB AX,1234H	Subtract the immediate number 1234H and borrow from AX and stores result in AL
SBB AX,BX	Subtract the content of BX and borrow from AX and stores result in AX

Arithmetic Instructions

INC destination

- This instruction adds 1 to the content of the specified register or memory location
- The destination can be a register or memory location
- Immediate data cannot be a operand of this instruction

Operation:

- Destination ← destination +1

Instructions	Description of instruction
INC AX	Increment the content of AX by 1
INC [2000H]	Increment the content of memory location 2000H by 1

Arithmetic Instructions

DEC destination

- This instruction subtracts 1 from the content of the specified register or memory location
- The destination can be a register or memory location
- Immediate data cannot be a operand of this instruction

Operation:

- Destination ← destination -1

Example:

Instructions	Description of instruction
DEC AX	Decrement the content of AX by 1
DEC [2000H]	Decrement the content of memory location 2000H by 1

Arithmetic Instructions

CMP Destination, Source

- This instruction compares the source operand with a destination operand
- The source and destination can be an immediate data, a register or a memory location
- This instruction subtracts the source operand form the destination operand
- The source and destination are not changed, but flags are set to indicate the result of the comparison
- AF,OF, SF,ZF, PF and CF are updated by the CMP instruction

Arithmetic Instructions

Operation:

Destination - Source

	CF	ZF	SF	
If D> S then	0	0	0	No borrow required, so CF=0
If D< S then	1	0	1	Sub required borrow, so CF=1
If D= S then	0	1	0	result of subtraction is 0

Example:

Instruction	Description of instruction
CMP AL,01H	Compare immediate number 01H with byte in AL
CMP BH,CL	Compare byte in CL with byte in BH

Arithmetic Instructions

MUL Source8/16 (Unsigned number multiplication)

- This instruction multiplies an unsigned byte/word in source with an unsigned byte in **AL** OR word in **AX**
- The source can be a register or a memory location
- When a **byte** is multiplied by the content of **AL**, the result (product)is put in **AX**
- When a **word** is multiplied by the content of **AX**, the result is put in **DX** and **AX** registers
- AF,PF,SF and ZF are undefined after MUL instruction

Arithmetic Instructions

Operation:

- If source ‘s’ is of 8 bits   MUL S8/16 then

$$\underset{\text{AL}}{\boxed{\text{8 bit}}} \times \underset{\text{S8}}{\boxed{\text{8 bit}}} = \underset{\text{AX}}{\boxed{\text{16 bit}}}$$

- If source is ‘s’ is of 16 bit MUL S16 then

$$\underset{\text{AX}}{\boxed{\text{16 bit}}} \times \underset{\text{S16}}{\boxed{\text{16 bit}}} = \underset{\text{DX}}{\boxed{\text{16 bit}}}\underset{\text{AX}}{\boxed{\text{16bit}}}$$

Arithmetic Instructions

Example:

Instruction	Description of instruction
MUL BH	Multiply AL with BH; Result in AX
MUL CX	Multiply Ax with CX; result high word in DX, low word in AX

Arithmetic Instructions

DIV Source8/16 (unsigned number division of accumulator data by source data)

- This instruction is used to divide an unsigned word by a byte or to divide an unsigned double word (32 bits) by a word
- When a word is divided by a byte, the word must be in the AX register
- The divisor can be in a register or a memory location

Arithmetic Instructions

Continue…

- After the division, AL will contain the 8-bit quotient, and AH will contain the 8-bit remainder

- When a double word is divided by a word, the most significant word of the double word must be in DX, and the least significant word of the double word must be in AX

- After the division, AX will contain the 16-bit quotient and DX will contain the 16-bit remainder

Arithmetic Instructions

Continue…

- For division, the dividend must always be in DX:AX or AX, but source of the divisor can be a register or a memory location

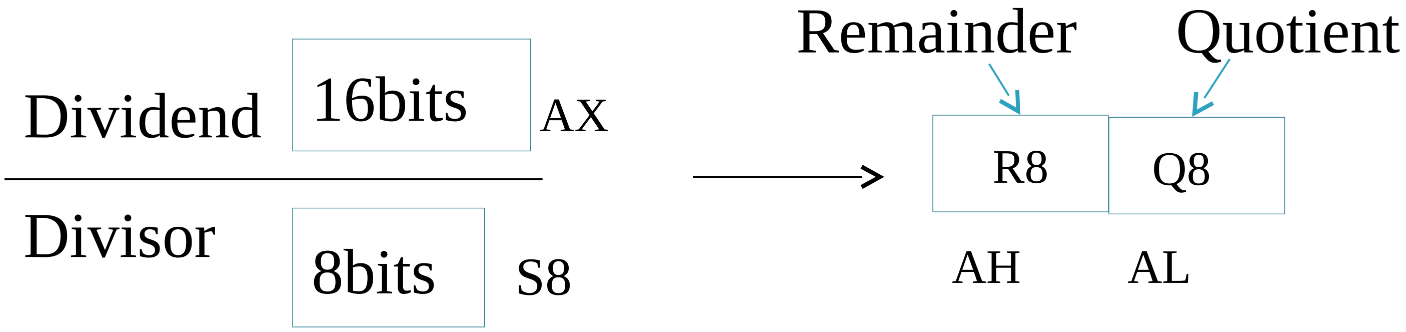
- If we want to divide a **byte by a byte**, we must first store dividend byte in AL and fill AH with all 0’s

- If we want to divide a **word by a word**, we must first store dividend word in AX and fill DX with all 0’s

Arithmetic Instructions

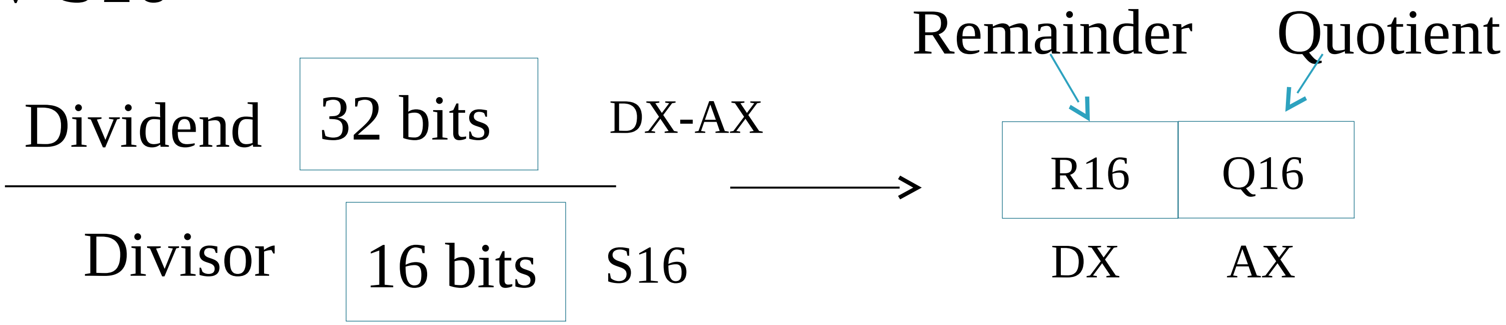
- If source 's' is of 8 bits

DIV S8



- If source 's' is of 16 bits

DIV S16



Arithmetic Instructions

Operation:

If source is byte then

AL ← AX/unsigned 8 bit source

AH ← AX MOD unsigned 8 bit source

If source is word then

AX ← DX:AX/unsigned 16 bit source

DX ← DX:AX MOD unsigned 16 bit source

Examples

DIV BL

Arithmetic Instructions

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
- CBW does not affect any flag

Operation

AH ← Filled with 8th bit of AL i.e. D7

Example

Let AX=00000000 10011011

After the execution of CBW

AX= 11111111 10011011

Arithmetic Instructions

CWD (Convert signed word to double word)

- This instruction copies the sign bit of a word in AX to all the bits in DX register
- It extends the sign of AX into all of DX
- The CWD operation must be done before performing division of signed word in AX by another word with the IDIV instruction

Arithmetic Instructions

Operation

DX← Filled with 16th bit of AX i.e.D15

Example

DX= 00000000 00000000 AX=11110000 11000111

CWD

After the execution of CWD

DX=11111111 11111111 AX=11110000 11000111

Arithmetic Instructions

DAA (Decimal adjust Accumulator)

- This instruction is used to convert the result of the addition of two BCD number to a valid BCD number
- DAA only works on AL register
- DAA instruction must be used after the ADD/ADC instruction
- Updates AF, CF, PF and OF

Arithmetic Instructions

Operation:

- If LSB’s of AL>9 or AF=1 then AL=AL+06
- If MSB’s of AL>9 or CF=1 then AL=AL+60
- If both above condition are satisfied then AL=AL+66

Example:

Let AL=14 BCD   0001 0100

CL=28 BCD then   + $\underline{0010\ 1000}$

ADD AL, CL   0011 1100 $\longrightarrow$ 3CH here C>9

AL=3CH   +   0110   Add 06 to LSB

$\underline{0100\ 0010}$ $\longrightarrow$ AL= 42 in BCD

(06 is added to AL as C>9)

Arithmetic Instruction

DAS (Decimal adjust After BCD Subtraction)

- This instruction is used to convert the result of the subtraction of two BCD number to a valid BCD number
- DAS only works on AL register
- DAS instruction must be used after the SUB/SBB instruction

Arithmetic Instructions

Operation:

- If LSB’s of AL>9 or AF=1 then AL=AL- 06
- If MSB’s of AL>9 or CF=1 then AL=AL-60
- If both above condition are satisfied then AL=AL-66

Example:

```
Let AL=55 BCD          0101 0101
CL=49H then            0100 1001
                      ___________
SUB AL, CL    –        0000 1100          0CH  here C>9,
                            0110  ——>  Sub 06 frm LSB
                      ___________
                       0000 0110  ——>AL= 06 in BCD
              _
```

(06 is subtracted from AL as C>9 & AF=1)

Arithmetic Instructions

NEG instruction

- Used to take 2’s Complement
- Syntax-

  NEG destination
- Ex:

  1) NEG AL

     Suppose AL=A3h

     After execution, AL= 5Dh

AAA (ASCII ADJUST FOR ADDITION)

- Numerical data coming into a computer from a terminal is usually in ASCII code.
- In this code, the numbers 0 to 9 are represented by the ASCII codes 30H to 39H.
- The 8086 allows you to add the ASCII codes for two decimal digits without masking off the “3” in the upper nibble of each.
- After the addition, the AAA instruction is used to make sure the result is the correct unpacked BCD.

- Let AL = 0011 0101 (ASCII 5), and
  BL = 0011 1001 (ASCII 9)

- ADD AL, BL
  AL = 0110 1110 (6EH, which is incorrect BCD)
  AAA
  AL = 0000 0100 (unpacked BCD 4)
  (unpacked BCD : One digit per byte)
  CF = 1 indicates answer is 14 decimal. This instruction clears the higher nibble.

- It works only on the AL register.
- It updates AF and CF; but OF, PF, SF and ZF are left undefined.

AAS (ASCII ADJUST FOR SUBTRACTION)

- Numerical data coming into a computer from a terminal is usually in an ASCII code.
- The 8086 allows you to subtract the ASCII codes for two decimal digits without masking the “3” in the upper nibble of each.
- The AAS instruction is then used to make sure the result is the correct BCD.

Let AL = 0011 1001 (39H or ASCII 9),

BL = 0011 0101 (35H or ASCII 5)

SUB AL, BL

AAS
AL = 0000 0100 (BCD 04), and CF = 0 (no borrow required)

AAM (BCD ADJUST AFTER MULTIPLY)

- Before you can multiply two ASCII digits, you must first mask the upper 4 bit of each.
- This leaves unpacked BCD (one BCD digit per byte) in each byte.
- After the two unpacked BCD digits are multiplied, the AAM instruction is used to adjust the product to two unpacked BCD digits in AX.

- AAM works only after the multiplication of two unpacked BCD bytes

- AAM updates PF, SF and ZF but AF; CF and OF are left undefined.

Let AL = 0000 0101 (unpacked BCD 5),
and BH =0000 1001 (unpacked BCD 9)

MUL BH
AL x BH: AX = 00000000 00101101 = 002DH

AAM
AX = 00000100 00000101 = 0405H (unpacked BCD for 45)

AAD (BCD-TO-BINARY CONVERT **BEFORE** DIVISION)

- AAD converts two unpacked BCD digits in AH and AL to the equivalent binary number in AX.
- This adjustment must be made before dividing the two unpacked BCD digits in AX by an unpacked BCD byte.
- After the BCD division,
  - AL will contain the unpacked BCD quotient
  - AH will contain the unpacked BCD remainder.
- AAD updates PF, SF and ZF; AF, CF and OF are left undefined.

Let AX = 0607 (unpacked BCD for 67 decimal), and CH = 09H

AAD
AX = 0043 (43H = 67 decimal)
DIV CH
AL = 07; AH = 04

Logical / Bit Manipulation Instructions

NOT Source:

- It complements each bit of Source to produce 1's complement of the specified operand.
- The operand can be a register or memory location.
- Example:

```
NOT BX
```

AND Destination, Source:

- It performs AND operation of Destination and Source
- Source can be immediate number, register or memory location.
- Destination can be register or memory location.
- Both operands cannot be memory locations at the same time.
- CF and OF become zero after the operation.
- PF, SF and ZF are updated.
- Example:

  AND BL,CL   BL <- BL AND CL

OR Destination, Source:

- It performs OR operation of Destination and Source.
- Source can be immediate number, register or memory location.
- Destination can be register or memory location.
- Both operands cannot be memory locations at the same time.
- CF and OF become zero after the operation.
- PF, SF and ZF are updated.

- Example:

  OR BL,CL   BL <- BL OR CL

XOR Destination, Source:

- It performs XOR operation of Destination and Source
- Source can be immediate number, register or memory location.
- Destination can be register or memory location.
- Both operands cannot be memory locations at the same time.
- CF and OF become zero after the operation.
- PF, SF and ZF are updated.

▸ Example:

XOR BL,CL   BL <- BL XOR CL

TEST Destination, Source:

- It is similar to logical AND
- Result is not stored
- Flag bis are affected (PF, SF, ZF) (CF, OF $\leftarrow$ 0)
- Source can be immediate number, register or memory location.
- Destination can be register or memory location.

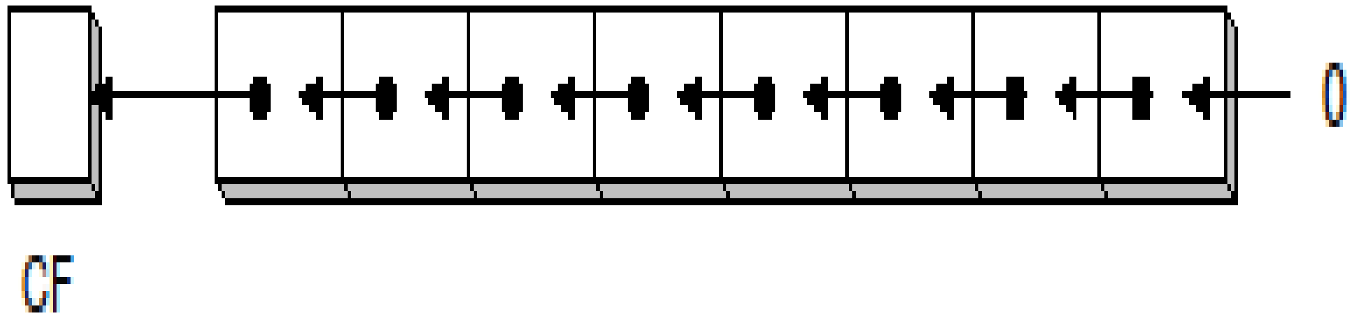
▸ Example:

TEST BL,CL

Shift Instructions

SHL/SAL Destination, Count:

- It shift bits of byte or word left, by count.
- It puts zero(s) in LSBs.
- MSB is shifted into carry flag.
- If the number of bits desired to be shifted is 1, then the immediate number 1 can be written in Count.
- However, if the number of bits to be shifted is more than 1, then the count is put in CL register.



Example:

1. SAL BL,1

   Before operation: BL=0011 0011 and CF=1

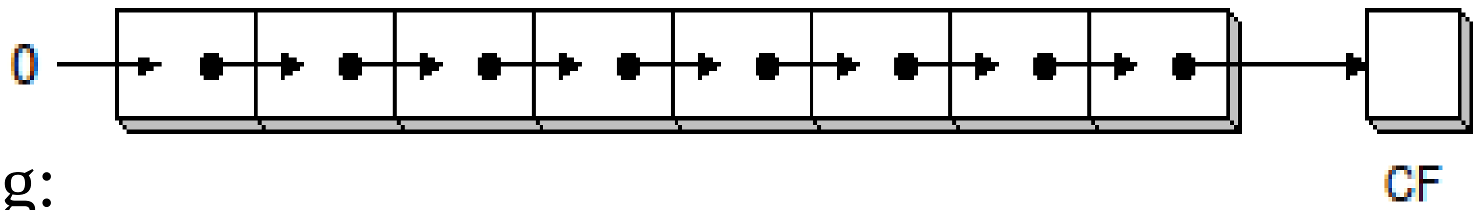
   After operation: BL=0110 0110 and CF=0

2. MOV CL,05H

   SAL BL,CL

SHR Destination, Count:

- It shift bits of byte or word right, by count.
- It puts zero(s) in MSBs.
- LSB is shifted into carry flag.
- If the number of bits desired to be shifted is 1, then the immediate number 1 can be written in Count.
- However, if the number of bits to be shifted is more than 1, then the count is put in CL register.



Eg:

1. SHR BL,1

   Before operation: BL=0011 0011 and CF=1

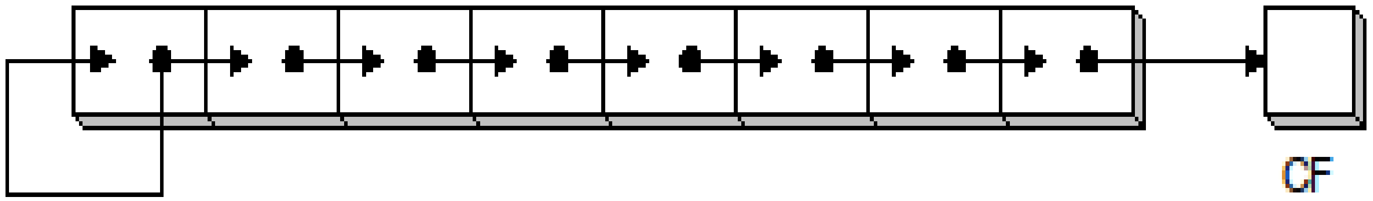
   After operation: BL=0001 1001 and CF=1

2. MOV CL,05H

   SHR BL,CL

SAL and SAR instructions

- SAL (shift arithmetic left) is identical to SHL.
- SAR (shift arithmetic right) performs a right arithmetic shift on the destination operand.



- An arithmetic shift preserves the number's sign.

Rotate Instructions

ROL Destination, Count:

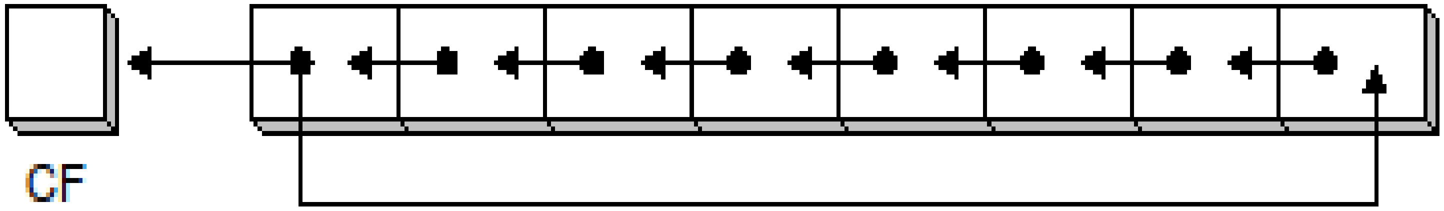
- It rotates bits of byte or word left, by count.
- MSB is transferred to LSB and also to CF.
- If the number of bits desired to be shifted is 1, then the immediate number 1 can be written in Count.
- However, if the number of bits to be shifted is more than 1, then the count is put in CL register.
- Destination can be register or memory location
- Example:

```
1.ROL AX, 1          2.MOV CL, 04H
                       ROL BL, CL
```

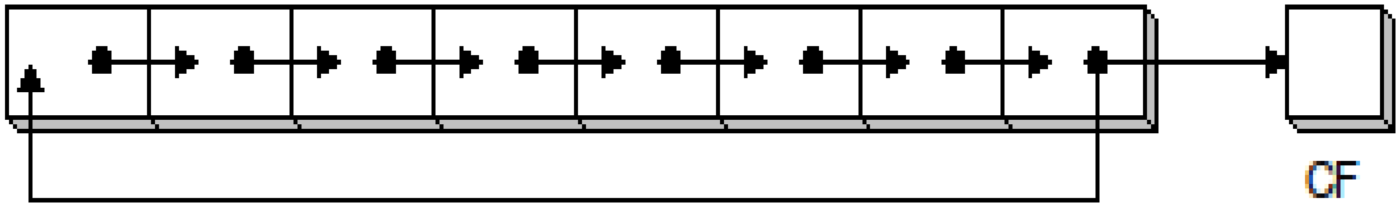
ROR Destination, Count:

- It rotates bits of byte or word right, by count.
- LSB is transferred to MSB and also to CF.
- If the number of bits desired to be shifted is 1, then the immediate number 1 can be written in Count.
- However, if the number of bits to be shifted is more than 1, then the count is put in CL register.
- Example:

```
1.ROR AX, 1
2.MOV CL, 04H
  ROR BL, CL
```



ROL



ROR

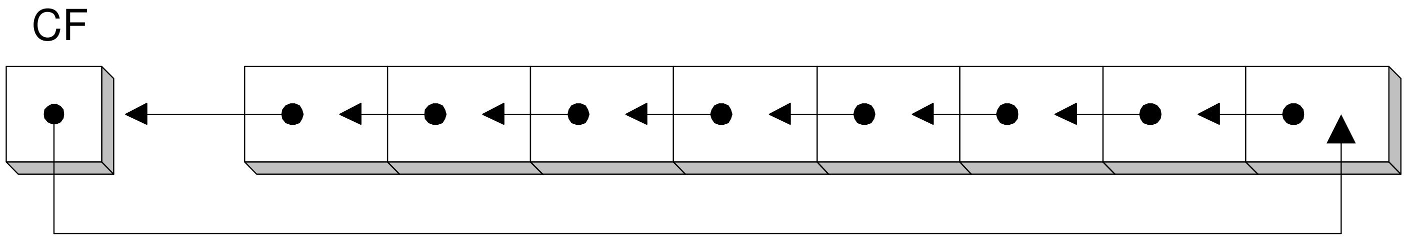
RCL Destination, Count:

- RCL (rotate carry left) shifts each bit to the left
- Copies the Carry flag to the least significant bit
- Copies the most significant bit to the Carry flag
- Example:

  1.RCL AX, 1

  2.MOV CL, 04H

  RCL BL, CL



RCR Destination, Count:

- RCR (rotate carry right) shifts each bit to the right
- Copies the Carry flag to the most significant bit
- Copies the least significant bit to the Carry flag
- Example:

  1.RCR AX, 1

  2.MOV CL, 04H

  RCR BL, CL

Processor Control Instructions

- These instructions control the processor itself.
- 8086 allows
  - to control certain control flags that causes the processing in a certain direction
  - processor synchronization if more than one microprocessor attached.

STI

- Sets the interrupt enable flag

CLI

- Clears the interupt enable flag

STC

- It sets the carry flag to 1.

CLC

- It clears the carry flag to 0.

CMC

- It complements the carry flag.

STD

- It sets the direction flag to 1.
- If it is set, string bytes are accessed from higher memory address to lower memory address.

CLD

- It clears the direction flag to 0.
- If it is reset, the string bytes are accessed from lower memory address to higher memory address.

External synchronization

HLT

- It causes the 8086 to stop fetching and executing instructions
- Different ways to get the processor out of the halt state are
  - Interrupt signal on the INTR pin
  - Interrupt signal on the NMI pin
  - Reset signal on the RESET input

NOP

- It simply uses up three clock cycles and increments the instruction pointer to point to the next instruction.
- It used to increase the delay of a loop.
- NOP does not affect any flag.

ESC

- This instruction is used to pass instructions to a coprocessor, such as the 8087 Math coprocessor, which shares the address and data bus with 8086.
- When 8086 fetches an ESC instruction, the coprocessor decodes the instruction and carries out the action specified by the 6-bit code specified in the instruction.
- Coprocessor treats all the normal 8086 instructions as NOPs.
- In some cases, the 8086 will access a data item in memory for the coprocessor

WAIT

- 8086 enters an idle condition.
- Come out of this idle state in 2 ways:
  - TEST input pin is made low
  - Interrupt signal is received on the INTR or the NMI interrupt input pins.
- If valid interrupt occurs, 8086 will return from the idle state after the interrupt service procedure executes.
- WAIT instruction is used to synchronize the 8086 with external hardware such as the 8087 Math coprocessor.

LOCK

- Many microcomputer systems contain several microprocessors.
- Each microprocessor has its own local buses and memory.
- The individual microprocessors are connected together by a system bus so that each can access system resources such as disk drive or memory.
- The LOCK prefix allows a microprocessor to make sure that another processor does not take control of the system bus

Program transfer instruction

CALL

- This instruction is used to call a subroutine or function or procedure.
  - INTRA Segment (Near) CALL
    - eg: call findMax
    - {SS:[SP-1], SS:[SP-2]} ← IP, SP ← SP-2
    - IP ← New offset address of findMax
  - INTER Segment (Far) CALL
    - {SS:[SP-1], SS:[SP-2]} ← CS, SP ← SP-2
    - {SS:[SP-1], SS:[SP-2]} ← IP, SP ← SP-2
    - CS ← New segment address of findMax
    - IP ← New offset address of findMax

RET

- It returns the control from procedure to calling program.
- Every CALL instruction should have a RET

JMP Des

- Used for unconditional jump from one place to another
- Example:

  JMP Label
- Types:
  - Intra-segment (Near Jump)
  - Inter-segment (Far Jump)

Conditional Jump Table

Mnemonic	Meaning	Jump Condition
JA	Jump if Above	CF = 0 and ZF = 0
JAE	Jump if Above or Equal	CF = 0 or ZF=1
JB	Jump if Below	CF = 1
JBE	Jump if Below or Equal	CF = 1 or ZF = 1
JC	Jump if Carry	CF = 1
JE	Jump if Equal	ZF = 1
JNC	Jump if Not Carry	CF = 0
JNE	Jump if Not Equal	ZF = 0
JNZ	Jump if Not Zero	ZF = 0
JPE	Jump if Parity Even	PF = 1
JPO	Jump if Parity Odd	PF = 0
JZ	Jump if Zero	ZF = 1

Conditional Jump Table

- Example:

```
CMP AX, 4371H
JA NEXT
```

- Compare by subtracting 4371H from AX
- Jump to label NEXT if AX above 4371H

Iteration Control Instructions

Loop Des

- This is a looping instruction.
- The number of times looping is required is placed in the CX register.
- With each iteration, the contents of CX are decremented.

Increment the value of array named prices by 1

```
MOV BX, OFFSET PRICES
MOV CX, 40
NEXT: MOV AL, [BX]
INC AL
MOV [BX], AL
INC BX
LOOP NEXT
```

LOOPE/LOOPZ

Looping occus only if zero flag is set

LOOPNE/LOOPNZ

- Looping occurs only if zero flag is reset
- Example: Searching

```
MOV BX, OFFSET ARRAY
MOV CX, 100
NEXT: CMP [BX], ODH
INC BX
LOOPNZ NEXT
```

Interrupt control instructions

- INT- Interrupt instruction with type number
- It is 2-byte instruction.
- First byte provides the op-code and the second byte provides the interrupt type number.
- There are 256 interrupt types under this group.
- The starting address for type0 interrupt is 00000H, for type1 interrupt is 00004H similarly for type2 is 00008H and ……so on.

TYPE 0 - division by zero situation.

TYPE 1 - single-step execution during the debugging of a program.

TYPE 2 - non-maskable NMI interrupt.

TYPE 3 - break-point interrupt.

TYPE 4 - overflow interrupt.

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

Interrupt control instructions

Its execution includes the following steps −

- Flag register value is pushed on to the stack.
- CS value of the return address and IP value of the return address are pushed on to the stack.
- IP is loaded from the contents of the word location ‘type number’ × 4
- CS is loaded from the contents of the next word location.
- Interrupt Flag and Trap Flag are reset to 0

INTO

- Causes an interrupt of TYPE 4
- Condition: OF=1
- Syntax: INTO

IRET

- Return to the main program from an ISR (Interrupt Service Routine)
- Return address of from stack into IP and CS registers & restore flag register from stack

String Instructions

- Source string – poined by SI in Data segment
- Destination string – pointed by DI in Extra segement
- Count for operations – CX
- DF=0 – auto increment (CLD instruction)
- DF=1 – auto decrement (STD instruction)

MOVS : MOVSB / MOVSW:

- It causes moving of byte or word from one string to another.
- In this instruction, the source string is in Data Segment and destination string is in Extra Segment.
- SI and DI store the offset values for source and destination index.
- Example:

```
MOV SI, OFFSET SOURCE
MOV DI, OFFSET DESTINATION
CLD
MOV CX, 04H
REP MOVSB
```

LODS : Load String Byte(LODSB) or String Word(LODSW)

- LODS instruction loads the AL / AX register by the content of a string pointed to by DS : SI register pair.

  Example:
  CLD
  MOV SI, OFFSET SOURCE
  LODS SOURCE

STOS : Store String Byte(STOSB) or String Word(STOSW)

- Stores the AL / AX register contents to a location in the string pointer by ES : DI register pair.

  Example:

  MOV DI, OFFSET TARGET
  STOS TARGET

CMPS : Compare String Byte(CMPSB) or String Word(CMPSW)

- The CMPS instruction can be used to compare two strings of byte or words.
- If both the byte or word strings are equal, zero Flag is set.

Example:

- Repeat the comparison of string bytes until end of string or until compared bytes are not equal

```
MOV SI, OFFSET FIRST
MOV DI, OFFSET SECOND
CLD
MOV CX, 100
REPE CMPSB
```

SCAS: Scan Byte (SCASB) or Word (SCASW)

- Compare the contents of AL (or AX) with ES:DI pair
- Comparison is done by subtarcation by byte (or word) from extra segment from AL (or AX).
- Example:

Length of string is 80

```
MOV DI, OFFSET STRING
MOV AL, 0DH
MOV CX, 80
CLD
REPNE SCAS STRING
```

REP : Repeat Instruction Prefix

- Used as an instruction prefix
- Used with string instruction only
- Causes the instruction to be repeated CX number of times.
- i. REPE / REPZ
  - repeat operation while equal / zero (ZF=1).
- ii. REPNE / REPNZ
  - repeat operation while not equal / not zero (ZF=0).

Assembly Language Programming

- Assembly language is a low-level programming language for a computer or other programmable device specific to a particular computer architecture

- Assembly language is converted into executable machine code by a utility program referred to as an assembler like NASM, MASM, TASM etc.

Assembly Language Instructions

- Mnemonics represent Machine Instructions
  - Each mnemonic used represents a single machine instruction
  - The assembler performs the translation
- Some mnemonics require operands
  - Operands provide additional information
    - register, constant, address, or variable

8086 Instruction - Basic Structure

Label *Operator* *Operand[s]* *;Comment*

Label - optional alphanumeric string
1st character must be **`a-z`**,**`A-Z`**,**`?`**,**`@`**,**`_`**,**`$`**
Last character must be **`:`**

Operator - assembly language instruction
mnemonic: an instruction format for humans
Assembler translates mnemonic into hexadecimal *opcode*

Operand[s] - 0 to 2 pieces of data required by instruction
Can be several different forms
Delineated by commas
immediate, register name, memory data, memory address

Comment - Extremely useful in assembler language

8086 Instruction - Example

Label *Operator* *Operand[s]* *;Comment*

```
INIT: mov   ax, bx      ; Copy contents of bx into ax
```

Label	-	**`INIT:`**
Operator	-	**`mov`**
Operands	-	**`ax`** and **`bx`**
Comment	-	alphanumeric string between **`;`** and **`\n`**

- Not case sensitive
- Unlike other assemblers, destination operand is first
- **`mov`** is the *mnemonic* that the assembler translates into an *opcode*

Assembler Directives

end *label*	end of program, label is entry point
proc far|near	begin a procedure; far, near keywords specify if procedure in different code segment (far), or same code segment (near)
endp	end of procedure
db	define byte
dw	define word (2 bytes)
dd	define double word (4 bytes)
dq	define quadword (8 bytes)
dt	define tenbytes
equ	equate, assign numeric expression to a name

Programs

- WAP to add two 16 bit numbers
- WAP to add two 16 bit BCD numbers
- WAP to add a word located at offset address 0800H in the segment address 3000H to another word located at offset address 0700H in the same segment. Store the result at offset address 0900H and carry at 0902H in the same segment.
- WAP to move a word string 200 bytes (i.e. 100 words) long from the offset address 1000H to 3000H in the segment 5000H.

Programs

- WAP to find the smallest word in an array of 100 words stored sequentially starting at the offset address 2000H in the segment address 5000H and store the result at the offset address 2000H.
- WAP to reverse a string consisting of 10 bytes stored in consequtive memory location starting from 3000H:4000H

Procedure

- While writing programs, it may be the case that a particular sequence of instructions is used several times.
- Same sequence can be written as a separate subprogram called a procedure
- Each time the sequence of instructions needs to be executed, CALL instruction can be used
- CALL instruction transfers the execution control to procedure.
- In the procedure, a RET instruction is used at the end.

Procedure

- Often large programs are split into number of independent tasks which can be easily designed and implemented.
- This is known as modular programming.

Advantages:

1. Programming becomes simple.
2. Reduced development time.
3. Debugging of smaller programs and procedures is easy.
4. Reuse of procedures is possible.

Procedure

Disadvantages:

1. Extra code may be required to integrate procedures.

2. Liking of procedures may be required.

3. Processor needs to do extra work to save status of current procedure and load status of called procedure.

Procedure

Defining procedures:

```
Procedure_name PROC [NEAR|FAR]
                .
                .
Procedure_name ENDP
```

- NEAR | FAR is optional and gives the types of procedure. If not used, assembler assumes the procedure as near procedure.

Example:

```
Factorial PROC NEAR
        . . .
Factorial ENDP
```

Procedure

WAP TO ADD AND SUBTRACT 2 NUMBERS. WRITE PROCEDURES FOR ADDITION AND SUBTRACTION.

Macros

- Disadvantage of using a procedure:
  - stack is needed
  - time is required to call procedures and return to calling program.
- When the repeated group of instruction is too short or not suitable to be implemented as a procedure, we use a MACRO.

Macros

- A macro is a group of instructions to which a name is given.
- Each time a macro is called in a program, the assembler will replace the macro name with the group of instructions.

  Advantages:

  1. Macro reduces the amount of repetitive coding.

  2. Program becomes more readable and simple.

  3. Execution time is less as compared to calling procedures.

  Disadvantage:

  1. Memory requirement of a program becomes more.

Macros

Defining macros:

Macros are defined before the definition of segments

```
Macro_name MACRO [argument1, argument2, …]
...
ENDM
```

Example:

Addition of two 16-bit numbers using macro

Mixed Language Programming using Assembly and C

- Programmers are more comfortable writing in C and spares the programmers some of the details of the actual implementation.
- However, there are some low-level tasks that either can be better implemented in assembly, or can only be implemented in assembly language.
- Ways of writing:
  - Inline Assembly
  - Linked Assembly

Inline Assembly

```
#include<stdio.h>
void main() {
    int a = 3, b = 3, c;
    asm {
      mov ax,a
      mov bx,b
      add ax,bx
      mov c,ax
   }
   printf("%d",c);
}
```

```
asm  mov  ax,a
asm mov bx,b
asm add ax,bx
asm mov c,ax
```

Linked Assembly

- When an assembly source file is assembled by an assembler, and a C source file is compiled by a C compiler, those two object files can be linked together by a linker to form the final executable
- **Advantage:**
  - Assembly files can be written using any syntax and assembler that the programmer is comfortable with.
  - If a change needs to be made in the assembly code, all of that code exists in a separate file, that the programmer can easily access.
- **Disadvantage:**
  - Both the assembler and the compiler need to be run
  - Those files need to be manually linked together by the programmer

DOS Interrupt(INT 21H)

User can invoke this interrupt to perform several useful functions, like inputting data from keyboard, and outputting data to monitor.

User will have to identify which function is being used by setting the AH register to a specific value.

Other registers may also need to be modified

INT 21H

INT 21H Function 01H: Inputting a single character from the keyboard with echo.

AH = 01H

AL = inputted ASCII character code

Code example to input a single character from keyboard and to echo it to the display.

```
MOV AH,01H
INT 21H
```

INT 21H

INT 21H Function 02H: Outputting a single character to the monitor.

AH = 02H

DL = ASCII character code to be displayed

Code example to output a single character to the monitor.

```
MOV AH,02H
INT 21H
```

INT 21H

INT 21H Function 09H: Outputting a string terminated with $ to the monitor.

AH = 09H

DX = String address can be calculated using OFFSET assembler directive.

msg DB 0DH, 0AH,"Enter a number1 $"

0DH and 0AH together will create a new line on the screen after displaying current msg for the next line to be displayed.

Code example to output a string terminated with $ to the monitor

```
MOV DX, OFFSET  msg
MOV AH, 09H
INT 21H
```

INT 21H

INT 21H Function 4CH: Terminate a process (EXIT).
AH = 4CH

This interrupt terminates a process.
Code example to terminate a program :

```
MOV AH, 4CH
INT 21H
```