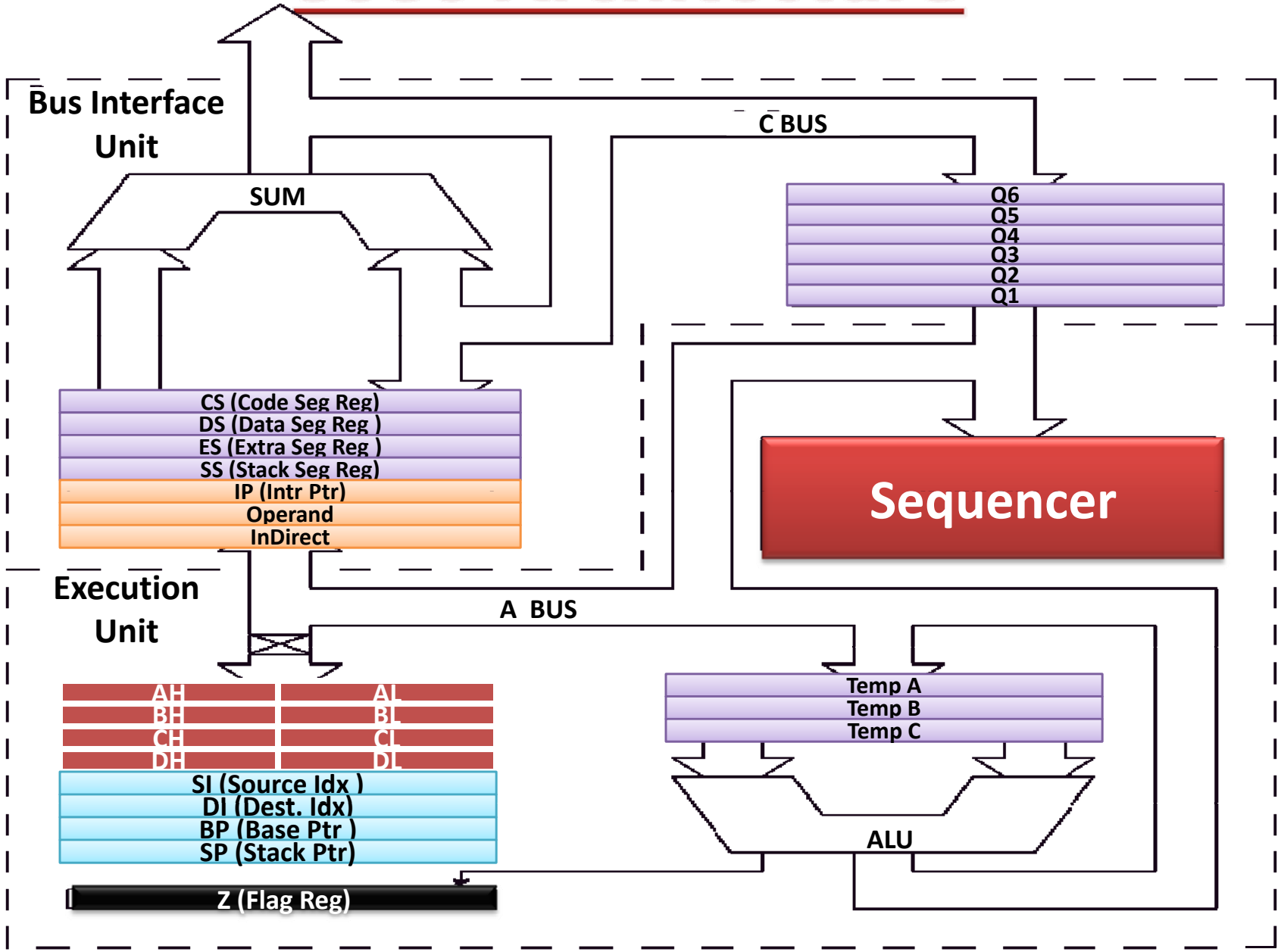
8086

- 8086
  - Block diagram (Data Path), Registers
- Memory Model
  - Stack, Data and Code Segment
- Instruction Set of x86
- Addressing mode
- Procedure and subroutine
- Peripheral device and Assembly program

8086 Architecture



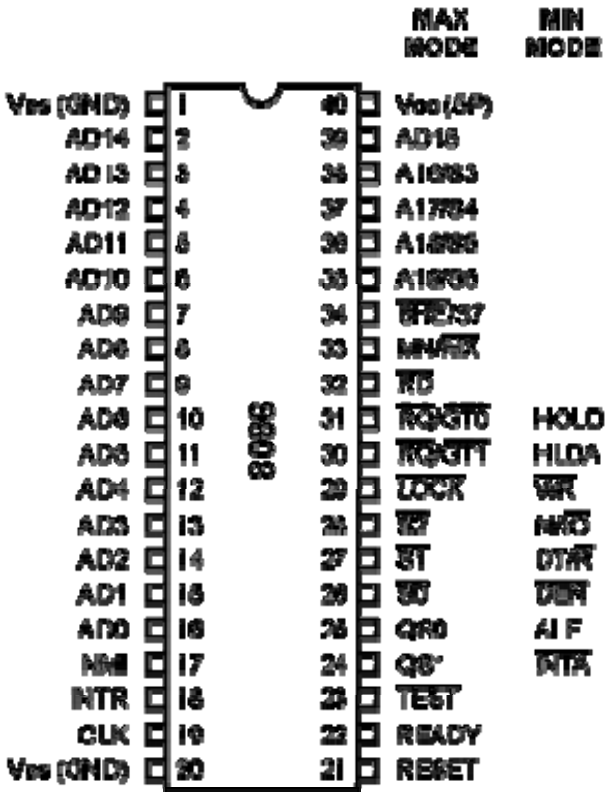
Minimum/Max Mode Configuration For 8086

The microprocessor 8086 is operated in minimum mode by strapping its MN/MX pin to logic 1.

In this mode, all the control signals are given out by the microprocessor chip itself. There is a single microprocessor in the minimum mode system.

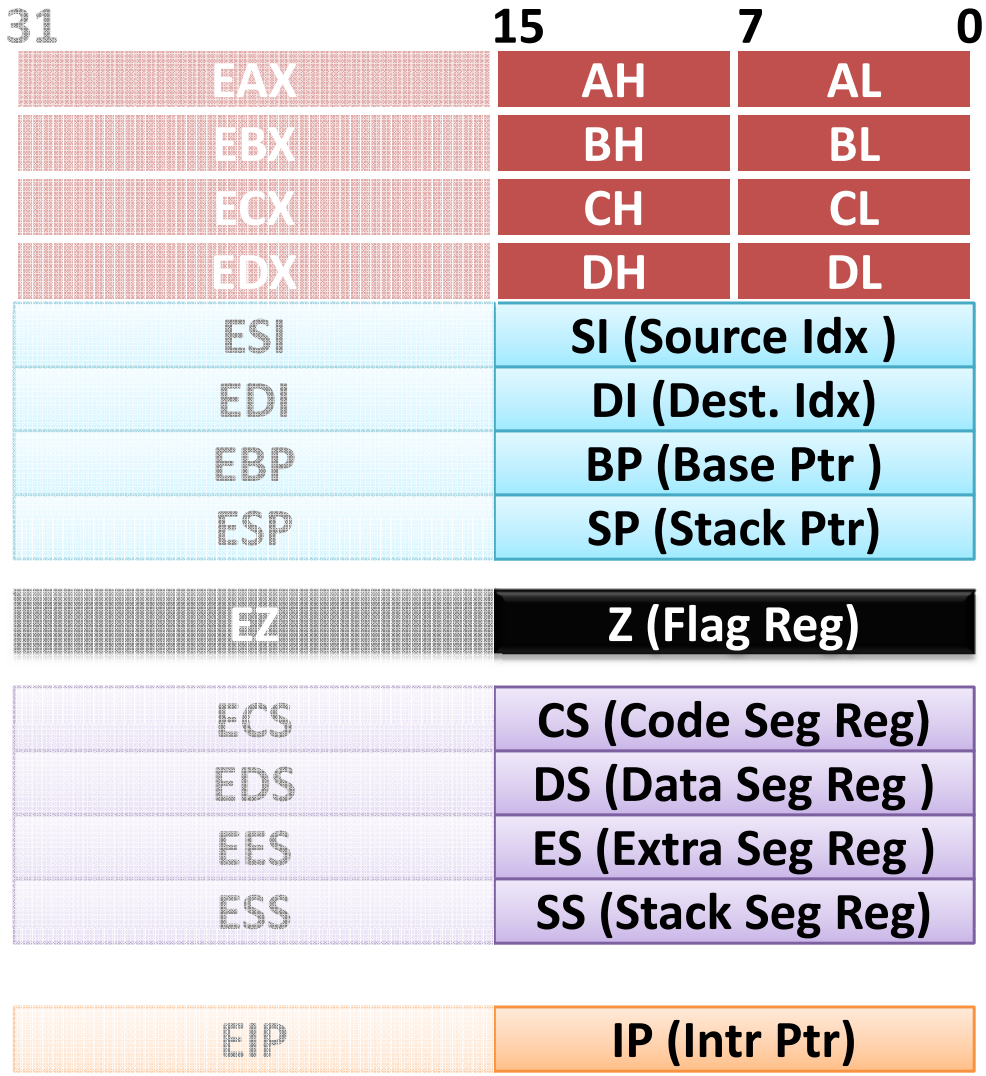
In the maximum mode, the 8086 is operated by strapping the MN/MX pin to ground.

In the maximum mode, there may be more than one microprocessor in the system configuration. The components in the system are same as in the minimum mode system.



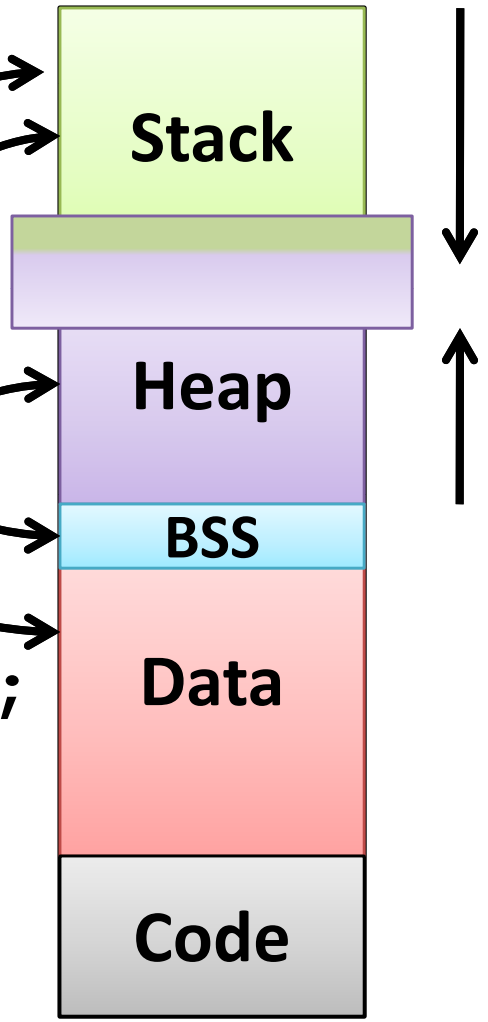
8086 & x86 Registers

- **AX** - accumulator reg
- **BX** - base address reg
- **CX** - count reg
- **DX** - data reg
- **SI** - source index reg
- **DI** - dest index reg
- **BP** - base pointer.
- **SP** - stack pointer.



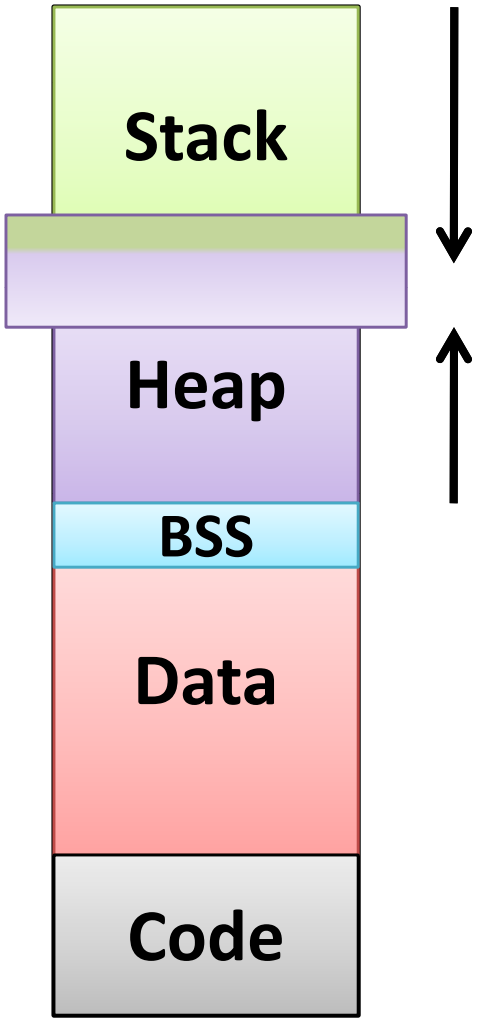
Memory layout of C program

```
int A;
int B=10;
main(){
   int Alocal;
   int *p;
   p=(int*)malloc(10);
}
```



Memory layout of C program

- Stack
  - automatic (default), local
  - Initialized/uninitialized
- Data
  - Global, static, extern
  - BSS: Block Started by Symbol
- Code
  - program instructions
- Heap
  - malloc, calloc



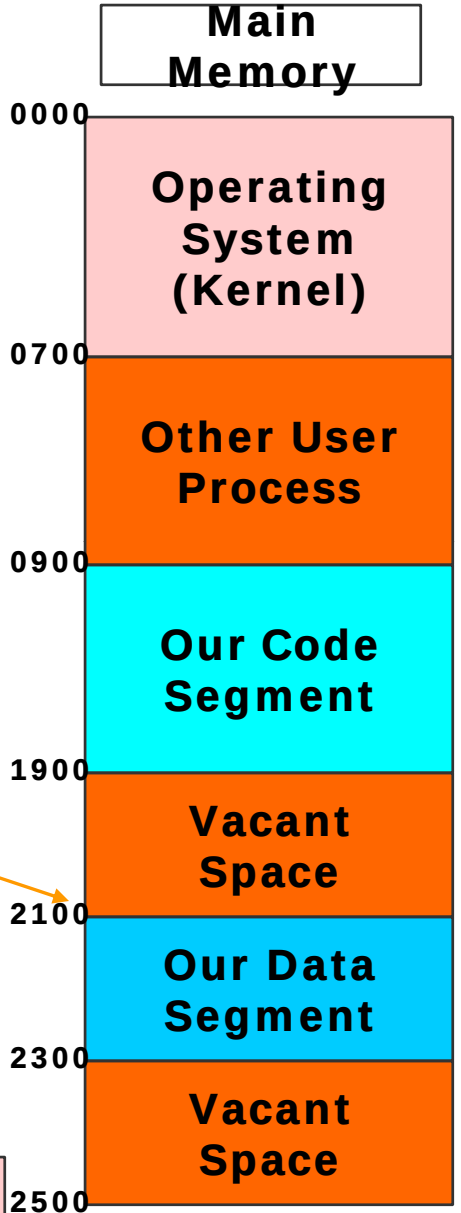
```
if (j>k)
  max = j
else
  max = k
```

```
Code_Segment:
        mov AX, [0]
        mov BX, [2]
        cmp AX,BX
        jle 0x7 //Label_1
        mov [4], AX
        jmp 0x5 //Label_2
Label_1: mov [4], BX
Label_2: ....
```

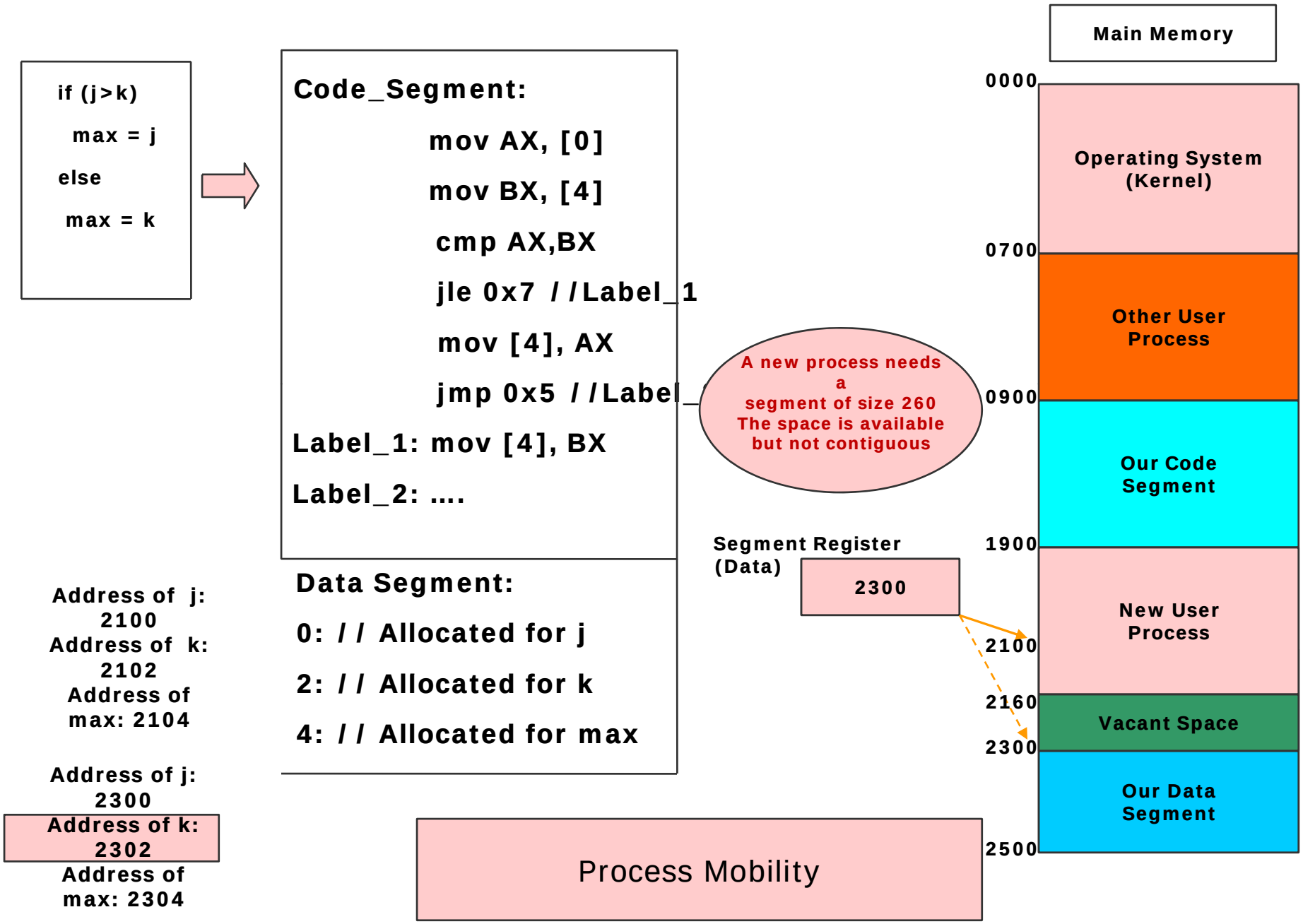
```
Data Segment:
0: // Allocated for j
2: // Allocated for k
4: // Allocated for max
```

Address of j: 2100
Address of k: 2102
Address of max: 2104

Segment Register (Data) 2100



Ease Of Programming


if (j>k)
max = j
else
max = k
Code_Segment:
mov AX, [0]
mov BX, [4]
cmp AX,BX
jle 0x7 //Label_1
mov [4], AX
jmp 0x5 //Label
Label_1: mov [4], BX
Label_2: ....
Data Segment:
0: // Allocated for j
2: // Allocated for k
4: // Allocated for max
A new process needs a segment of size 260 The space is available but not contiguous
Segment Register (Data)
2300
Main Memory
0000
Operating System (Kernel)
0700
Other User Process
0900
Our Code Segment
1900
New User Process
2100
2160
Vacant Space
2300
Our Data Segment
2500
Address of j: 2100
Address of k: 2102
Address of max: 2104
Address of j: 2300
Address of k: 2302
Address of max: 2304
Process Mobility

Multiple Segments

- The segment register can change its values to point to different segments at different times.
- X86 architecture provides additional segment registers to access multi data segments at the same time.
  - DS, ES, FS and GS
- X86 supports a separate Stack Segment Register (SS) and a Code segment Register (CS) in addition.
- By default a segment register is fixed for every instruction, for all the memory access performed by it. For eg. all data accessed by MOV instruction take DS as the default segment register.
- An **segment override prefix** is attached to an instruction to change the segment register it uses for memory data access.

mov [10], ax

- this will move the contents of ax register to memory location 0510

Opcode: 89 05 10

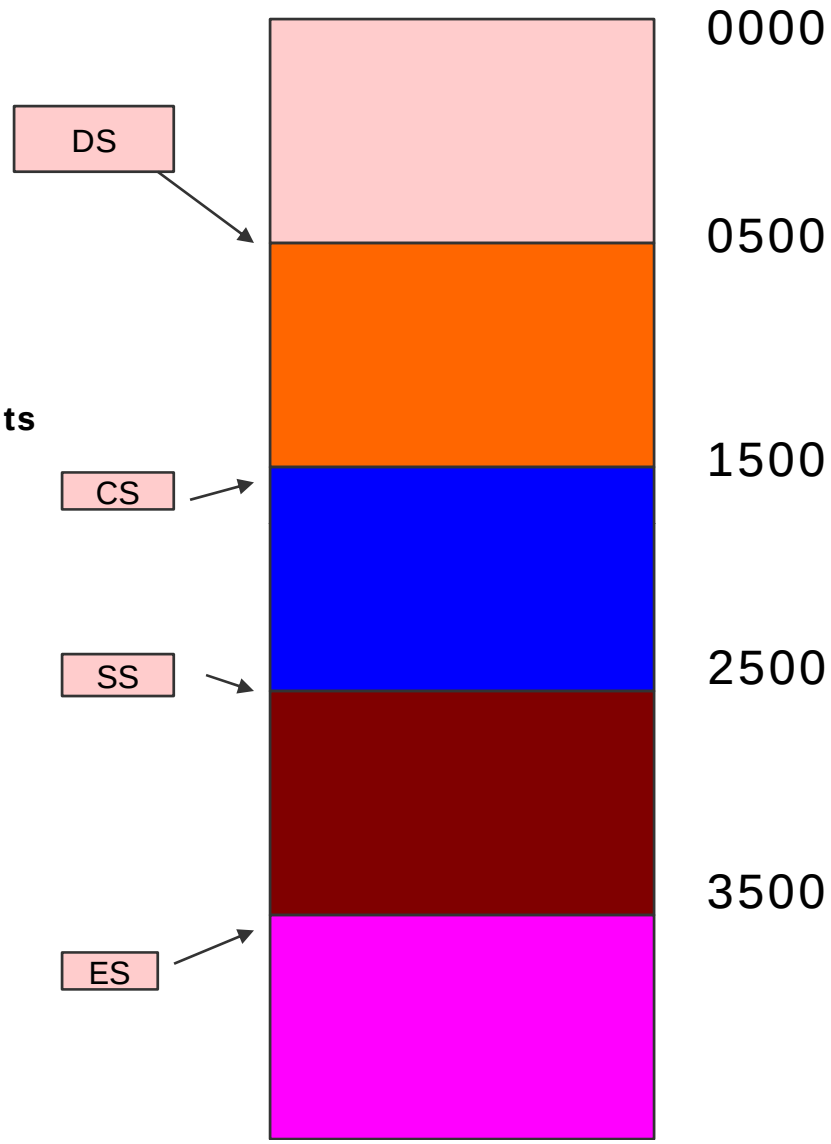
mov [ES:10], ax

-this will move the contents of ax register to memory location 3510

Opcode

26 89 05 10

“26” is the segment override prefix.



Multiple Segments

- Three salient features of using Segmentation

- Three Features
  - Code Mobility
  - Logically every segment can start with zero
  - Inter and Intra process protection ensuring data integrity.

Real Mode - Memory Addressing

- Segment << 4 + offset = 20 bit EA
- Segment size is a fixed 64K

DS = 0x1004

```
mov  [0x1000], AX
```

The mov will store the content of AX in

0x10040 + 0x1000 = 0x11040

Why this stuff? - To get 1 MB addressing using 16-bit Segment Registers

Pentium Addressing Mode Calculation

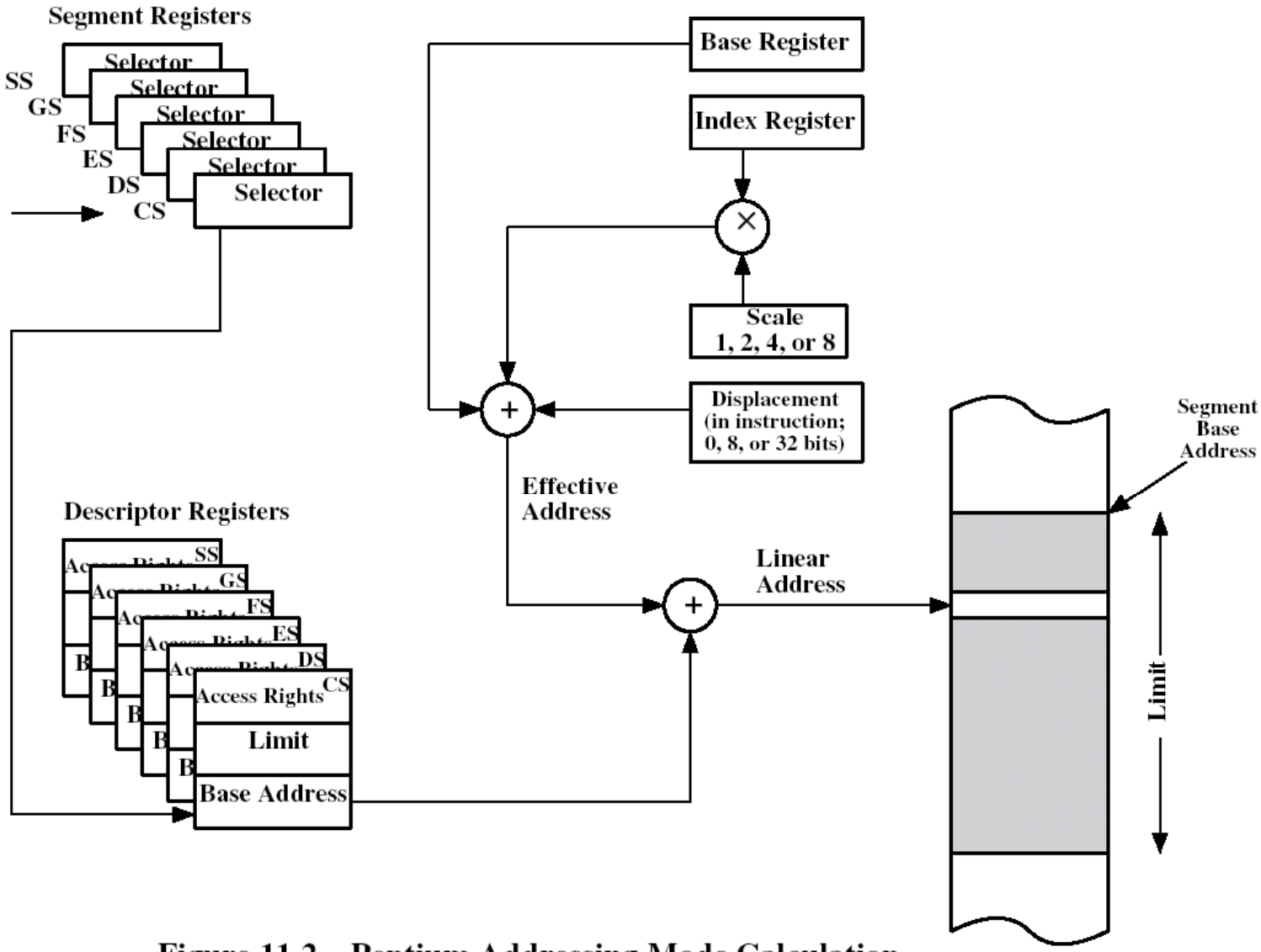


Figure 11.2 Pentium Addressing Mode Calculation

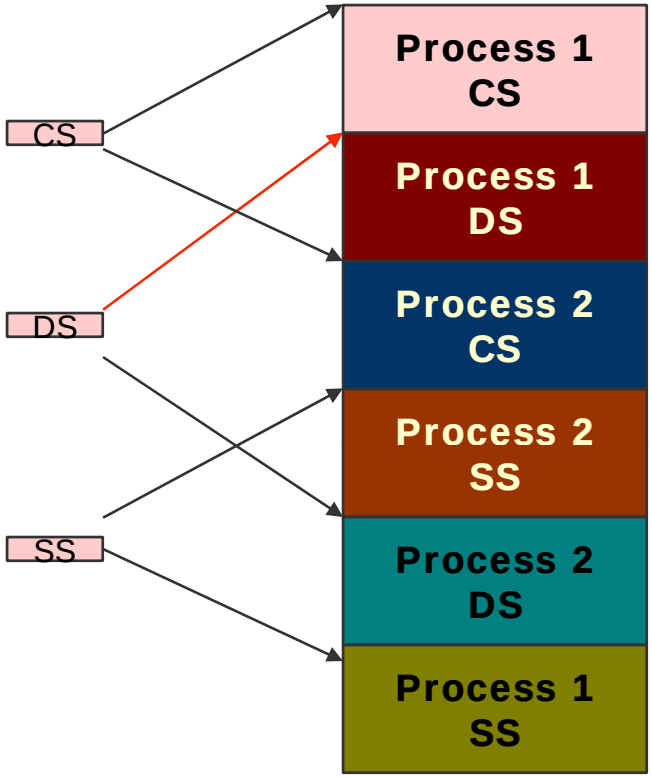
Process 1 should be prevented from loading CS, such that it can access the code of Process 2

Similarly for the DS,SS, ES, FS and GS

Privilege levels: [0-3] assigned to each segment.

0: Highest privilege

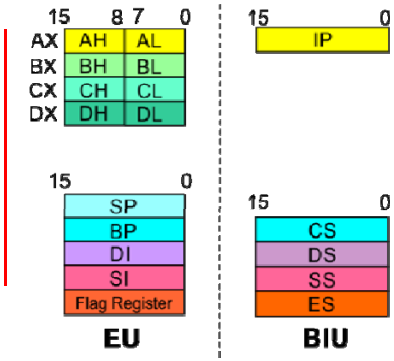
3: Lowest privilege

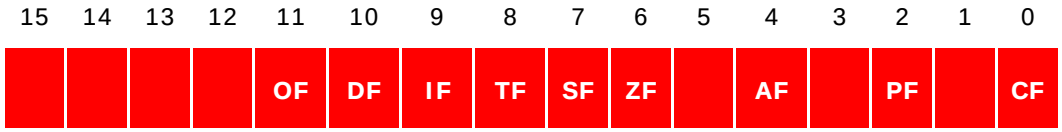


Interprocess Protection

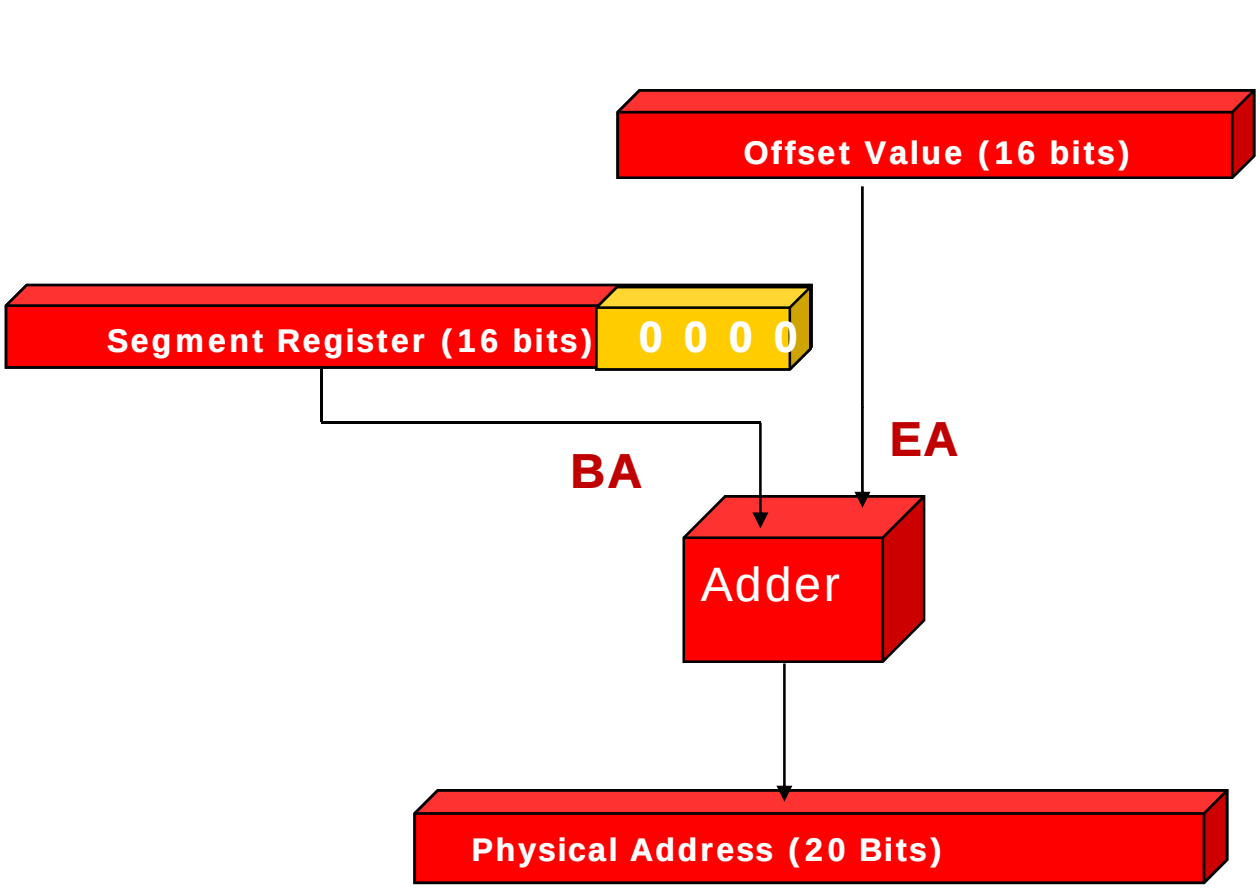
Architecture registers

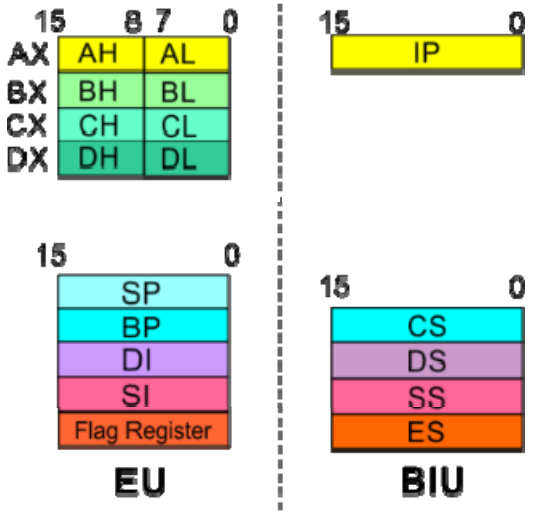
8086 registers categorized groups





Sl.No.	Type	Register width	Name of register
1	General purpose register	16 bit	AX, BX, CX, DX
		8 bit	AL, AH, BL, BH, CL, CH, DL, DH
2	Pointer register	16 bit	SP, BP
3	Index register	16 bit	SI, DI
4	Instruction Pointer	16 bit	IP
5	Segment register	16 bit	CS, DS, SS, ES
6	Flag (PSW)	16 bit	Flag register


Offset Value (16 bits)
Segment Register (16 bits)
0 0 0 0
BA
EA
Adder
Physical Address (20 Bits)


15 8 7 0
AX AH AL
BX BH BL
CX CH CL
DX DH DL
15 0
IP
15 0
SP
BP
DI
SI
Flag Register
EU
15 0
CS
DS
SS
ES
BIU

MA = BA + EA

Architecture Registers and Special Functions

Register	Name of the Register	Special Function
AX	16-bit Accumulator	Stores the 16-bit results of arithmetic and logic operations
AL	8-bit Accumulator	Stores the 8-bit results of arithmetic and logic operations
BX	Base register	Used to hold base value in base addressing mode to access memory data
CX	Count Register	Used to hold the count value in SHIFT, ROTATE and LOOP instructions
DX	Data Register	Used to hold data for multiplication and division operations
SP	Stack Pointer	Used to hold the offset address of top stack memory
BP	Base Pointer	Used to hold the base value in base addressing using SS register to access data from stack memory
SI	Source Index	Used to hold index value of source operand (data) for string instructions
DI	Data Index	Used to hold the index value of destination operand (data) for string operations

Introduction

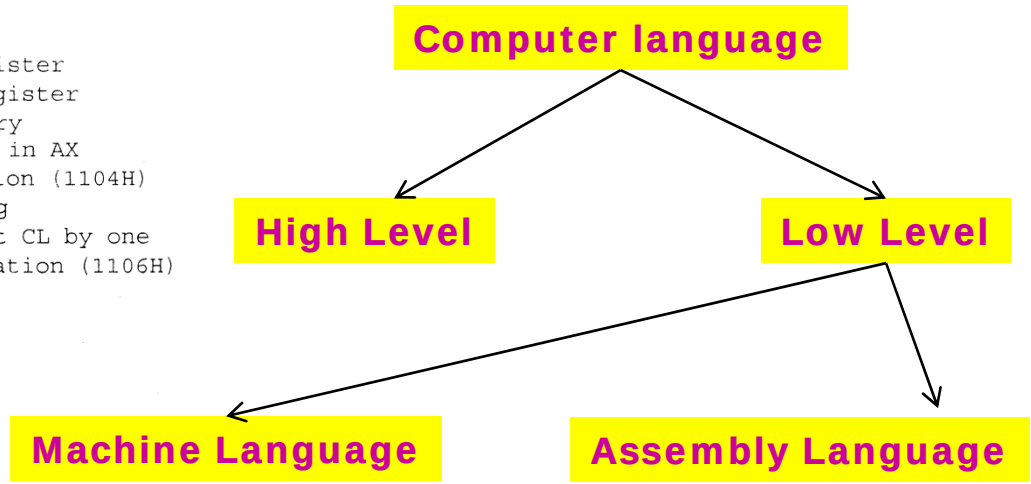
```
;PROGRAM TO ADD TWO 16-BIT DATA (METHOD-1)

DATA SEGMENT              ;Assembler directive

        ORG 1104H         ;Assembler directive
        SUM DW 0          ;Assembler directive
        CARRY DB 0        ;Assembler directive

DATA ENDS                 ;Assembler directive

CODE SEGMENT              ;Assembler directive

        ASSUME CS:CODE    ;Assembler directive
        ASSUME DS:DATA    ;Assembler directive
        ORG 1000H         ;Assembler directive

        MOV AX,205AH      ;Load the first data in AX register
        MOV BX,40EDH      ;Load the second data in BX register
        MOV CL,00H        ;Clear the CL register for carry
        ADD AX,BX         ;Add the two data, sum will be in AX
        MOV SUM,AX        ;Store the sum in memory location (1104H)
        JNC AHEAD         ;Check the status of carry flag
        INC CL            ;If carry flag is set,increment CL by one
AHEAD:  MOV CARRY,CL      ;Store the carry in memory location (1106H)
        HLT

CODE ENDS                 ;Assembler directive
END                       ;Assembler directive
```

Program

A set of instructions written to solve a problem.

Instruction

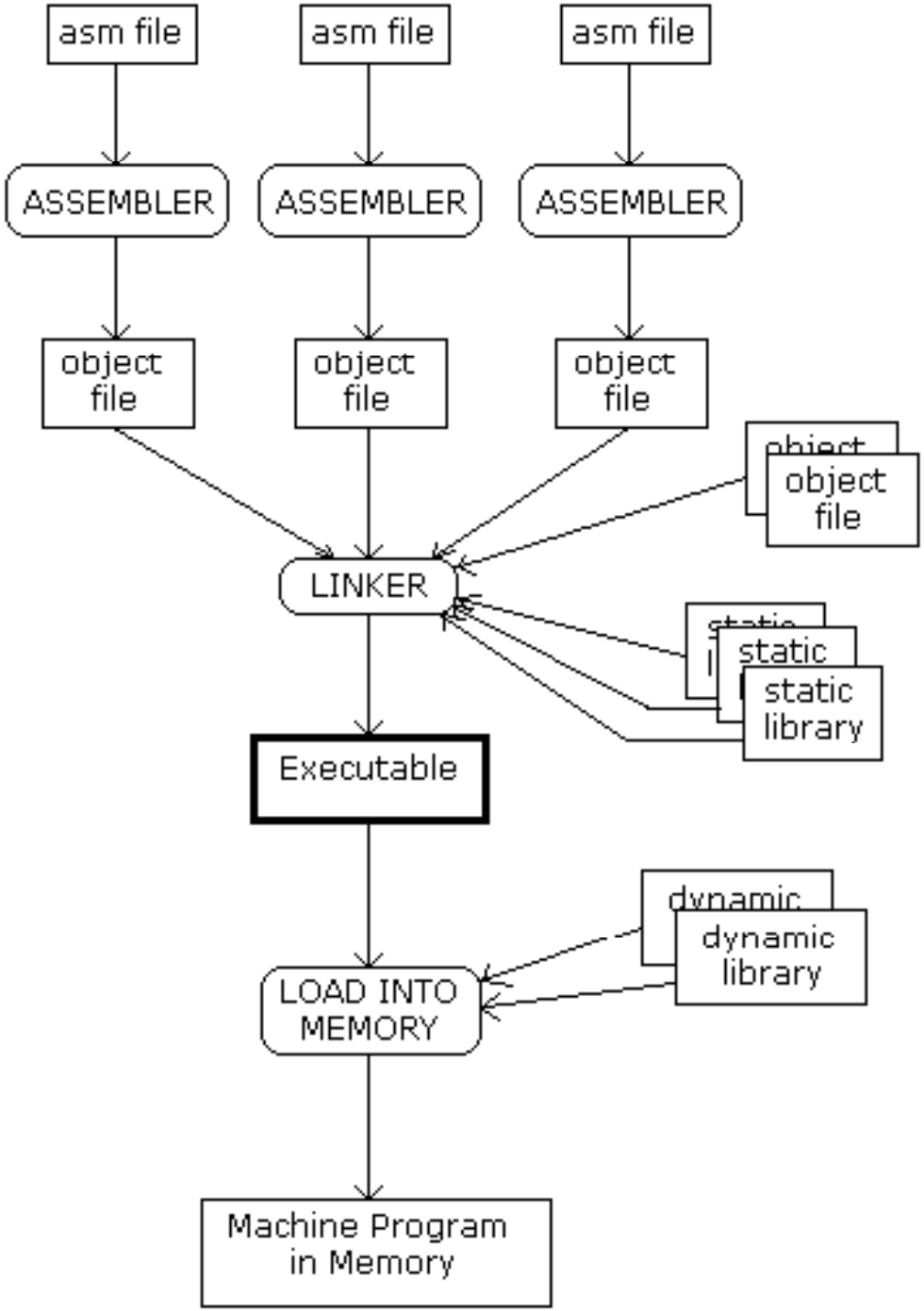
Directions which a microprocessor follows to execute a task or part of a task.



- **Binary bits**

- **English Alphabets**
- **'Mnemonics'**
- **Assembler**
  Mnemonics → Machine Language

Assemblers and Linkers

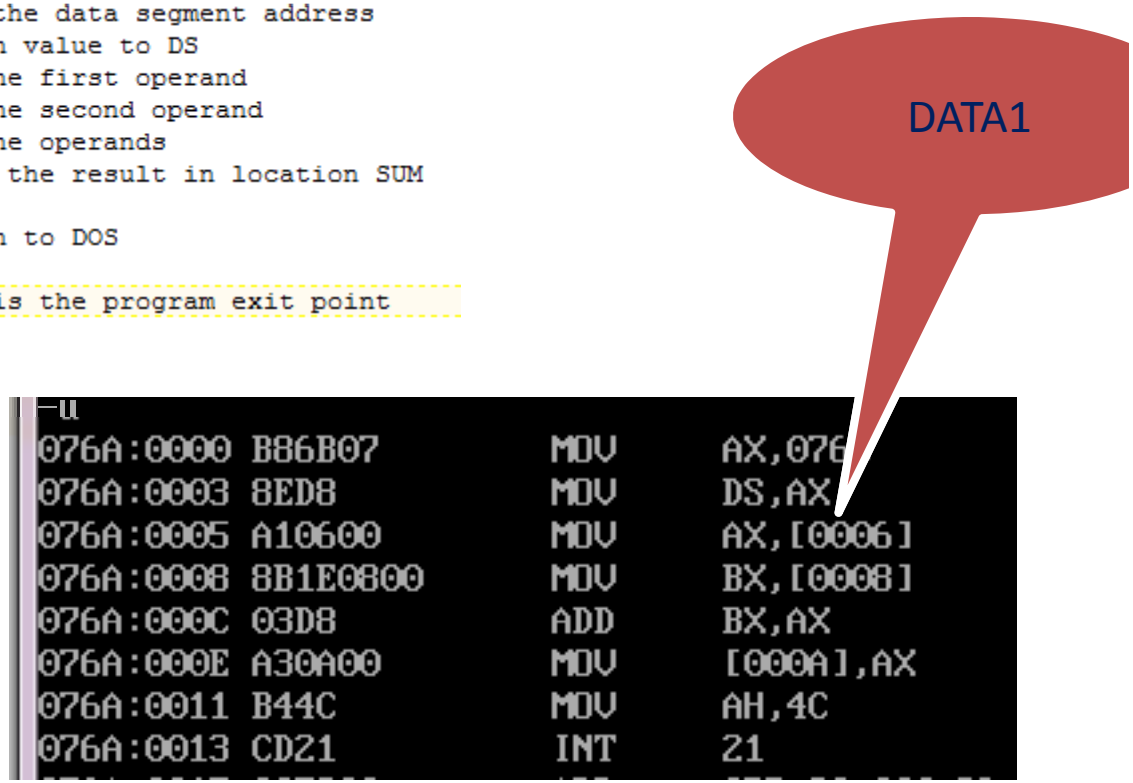


Assemble Directives

- Instructions to the Assembler regarding the program being executed.
- Control the generation of machine codes and organization of the program; but no machine codes are generated for assembler directives.
- Also called ‘pseudo instructions’
- Used to :
  - specify the start and end of a program
  - attach value to variables
  - allocate storage locations to input/ output data
  - define start and end of segments, procedures, macros etc..

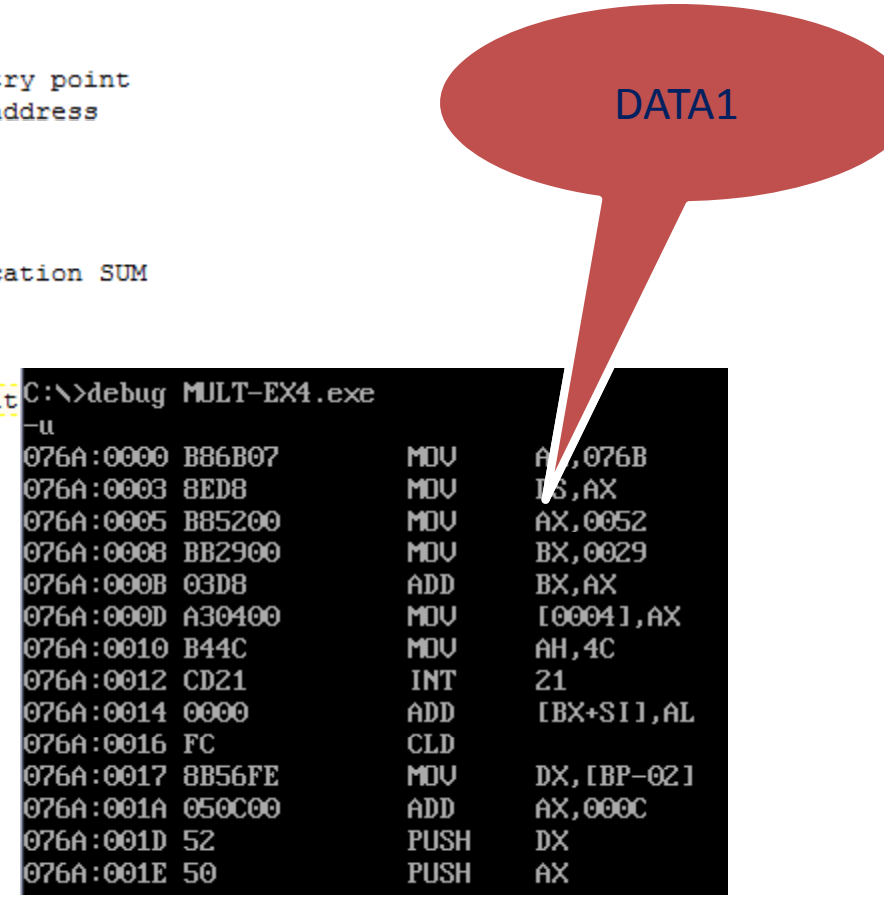
ADD example

```
                .MODEL SMALL
                .STACK 64
                .DATA
DATA1           DW      0052H
DATA2           DW      0029H
SUM             DW      ?
                .CODE
MAIN            PROC    FAR             ;this is the program entry point
                MOV     AX,@DATA        ;load the data segment address
                MOV     DS,AX           ;assign value to DS
                MOV     AX,DATA1        ;get the first operand
                MOV     BX,DATA2        ;get the second operand
                add     BX,AX           ;add the operands
                MOV     SUM,AX          ;store the result in location SUM
                MOV     AH,4CH  ;set up to
                INT     21H             ;return to DOS
MAIN            ENDP
                END     MAIN            ;this is the program exit point
```



ADD example

```
                .MODEL SMALL
                .STACK 64
                .DATA
DATA1           EQU     0052H
DATA2           EQU     0029H
SUM             DW      ?
                .CODE
MAIN            PROC    FAR             ;this is the program entry point
                MOV     AX,@DATA        ;load the data segment address
                MOV     DS,AX           ;assign value to DS
                MOV     AX,DATA1        ;get the first operand
                MOV     BX,DATA2        ;get the second operand
                add     BX,AX           ;add the operands
                MOV     SUM,AX          ;store the result in location SUM
                MOV     AH,4CH  ;set up to
                INT     21H             ;return to DOS
MAIN            ENDP
                END     MAIN            ;this is the program exit
```



Mult Example

```
                .DATA
DATA1           DB      52H
DATA2           DB      29H
SUM             DW      ?
                .CODE
MAIN            PROC    FAR             ;this is the program entry point
                MOV     AX,@DATA        ;load the data segment address
                MOV     DS,AX           ;assign value to DS
                MOV     AL,DATA1        ;get the first operand
                MOV     BL,DATA2        ;get the second operand
                IMUL    BL              ;multiply the operands
                MOV     SUM,AX          ;store the result in location SUM
                MOV     AH,4CH  ;set up to
                INT     21H             ;return to DOS
MAIN            ENDP
                END     MAIN            ;this is the program exit point
```

```
C:\>debug MULT-EX1.exe
-u
076A:0000 B86B07        MOV     AX,076B
076A:0003 8ED8          MOV     DS,AX
076A:0005 A00600        MOV     AL,[0006]
076A:0008 8A1E0700      MOV     BL,[0007]
076A:000C F6EB          IMUL    BL
076A:000E A30800        MOV     [0008],AX
076A:0011 B44C          MOV     AH,4C
076A:0013 CD21          INT     21
```

```
AX=076B  BX=0000  CX=001A  DX=0000  SP=0040  BP=0000  SI=0000  DI=0000
DS=076B  ES=075A  SS=076C  CS=076A  IP=0005   NV UP EI PL NZ NA PO NC
076A:0005 A00600        MOV     AL,[0006]                       DS:0006=52
-t

AX=0752  BX=0000  CX=001A  DX=0000  SP=0040  BP=0000  SI=0000  DI=0000
DS=076B  ES=075A  SS=076C  CS=076A  IP=0008   NV UP EI PL NZ NA PO NC
076A:0008 8A1E0700      MOV     BL,[0007]                       DS:0007=29
-t

AX=0752  BX=0029  CX=001A  DX=0000  SP=0040  BP=0000  SI=0000  DI=0000
DS=076B  ES=075A  SS=076C  CS=076A  IP=000C   NV UP EI PL NZ NA PO NC
076A:000C F6EB          IMUL    BL
-t

AX=0D22  BX=0029  CX=001A  DX=0000  SP=0040  BP=0000  SI=0000  DI=0000
DS=076B  ES=075A  SS=076C  CS=076A  IP=000E   OV UP EI PL NZ NA PO CY
076A:000E A30800        MOV     [0008],AX                       DS:0008=0000
-t

AX=0D22  BX=0029  CX=001A  DX=0000  SP=0040  BP=0000  SI=0000  DI=0000
DS=076B  ES=075A  SS=076C  CS=076A  IP=0011   OV UP EI PL NZ NA PO CY
076A:0011 B44C          MOV     AH,4C
```

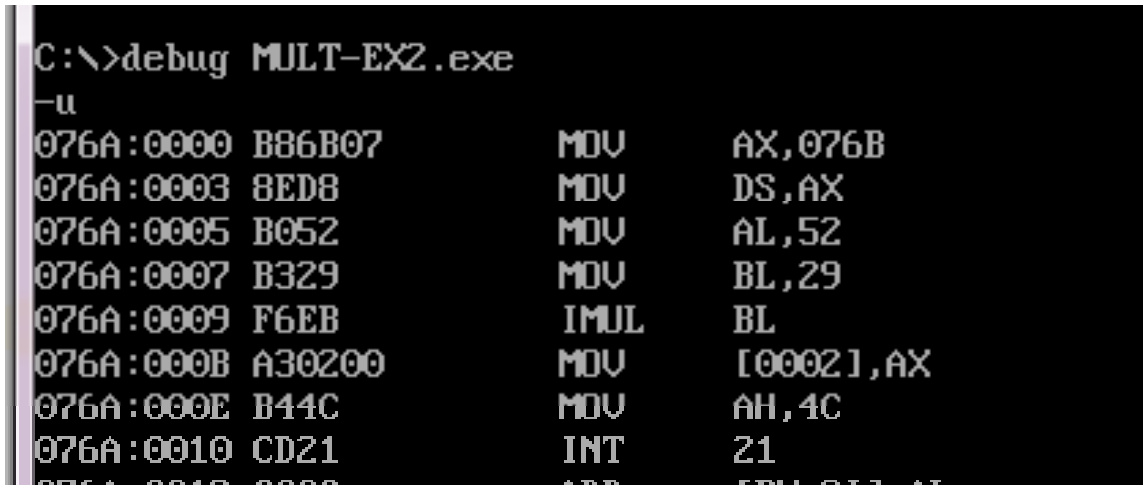
Mult Result

Mult Result

```
AX=0D22  BX=0029  CX=001A  DX=0000  SP=00    =0000  SI=0000  DI=0000
DS=076B  ES=075A  SS=076C  CS=076A  IP=0     OV UP EI PL NZ NA PO CY
076A:000E A30800        MOV     [0008],A                        DS:0008=0000
-t

AX=0D22  BX=0029  CX=001A  DX=0000  SP=     0  BP=0000  SI=0000  DI=0000
DS=076B  ES=075A  SS=076C  CS=076A  IP=     11   OV UP EI PL NZ NA PO CY
076A:0011 B44C          MOV     AH,4C
-d ds:0006
076B:0000                   52 29-22 0D 05 0C 00 52 50 E8         R)"....RP.
076B:0010  0E 49 83 C4 04 50 E8 9F-0E 83 C4 04 3D FF FF 74   .I...P......=..t
076B:0020  03 E9 11 01 B8 2F 00 50-8B 46 FC 8B 56 FE 05 0C   ...../.P.F..V...
076B:0030  00 52 50 E8 EA 48 83 C4-04 50 E8 7B 0E 83 C4 04   .RP..H...P.{....
076B:0040  3D FF FF 74 03 E9 22 0D-00 00 11 00 6A 07 A3 01   =..t.."".....j...
076B:0050  E4 40 50 8B C3 8C C2 05-0C 00 52 50 E8 C1 48 83   .@P.......RP..H.
076B:0060  C4 04 50 8D 86 FA FE 50-E8 17 73 83 C4 06 8B B6   ..P....P..s.....
076B:0070  FA FE 81 E6 FF 00 C6 82-FB FE 00 2B C0 50 8D 86   ...........+.P..
076B:0080  FB FE 50 E8 08 6A                                 ..P..j
-
```

Mult Example

```
                .MODEL SMALL
                .STACK 64
                .DATA
DATA1           EQU     52H
DATA2           EQU     29H
SUM             DW      ?
                .CODE
MAIN            PROC    FAR             ;this is the program entry point
                MOV     AX,@DATA        ;load the data segment address
                MOV     DS,AX           ;assign value to DS
                MOV     AL,DATA1        ;get the first operand
                MOV     BL,DATA2        ;get the second operand
                IMUL    BL              ;multiply the operands
                MOV     SUM,AX          ;store the result in location SUM
                MOV     AH,4CH  ;set up to
                INT     21H             ;return to DOS
MAIN            ENDP
                END     MAIN            ;this is the program exit point
```



Example : Square nos.

```
.model small
.stack 64
.data
org 7000H
nos db 05h.02h.03h.04h.05h

.code

start: mov ax,@data
       mov ds,ax
        lea si,nos        ;SI<- count
        MOV CL, [SI]     ;CL<-[SI] , firs tnumber
        MOV CH, 00       ; CH<-00
        INC SI           ;SI<-SI+1
 loop1: MOV AL, [SI]     ; AL<-[SI]
        MUL AL           ;' AX=AL*AL
        MOV [SI], AL     ; AL->[SI]
        INC SI           ; SI<-SI+1
        LOOP  loop1  ;  JUMP TO loop1 IF CX!=0 and CX=CX-1

  exit: mov ah,4ch
        int 21h
        end start
         .end
```

Count

Count

```
C:\>debug sq_1.exe
-u
076A:0000 B86B07        MOV     AX,076B
076A:0003 8ED8          MOV     DS,AX
076A:0005 8D360C70      LEA     SI,[700C]
076A:0009 8A0C          MOV     CL,[SI]
076A:000B B500          MOV     CH,00
076A:000D 46            INC     SI
076A:000E 8A04          MOV     AL,[SI]
076A:0010 F6E0          MUL     AL
076A:0012 8804          MOV     [SI],AL
076A:0014 46            INC     SI
076A:0015 E2F7          LOOP    000E
076A:0017 B44C          MOV     AH,4C
076A:0019 CD21          INT     21
076A:001B 0000          ADD     [BX+SI],AL
076A:001D 0000          ADD     [BX+SI],AL
076A:001F 0000          ADD     [BX+SI],AL
```

Example : Square nos.

```
C:\>debug sq_1.exe
-t

AX=076B  BX=0000  CX=7021  DX=0000  SP=0040  BP=0000  SI=0000  DI=0000
DS=075A  ES=075A  SS=0E6D  CS=076A  IP=0003   NV UP EI PL NZ NA PO NC
076A:0003 8ED8          MOV     DS,AX
-t

AX=076B  BX=0000  CX=7021  DX=0000  SP=0040  BP=0000  SI=0000  DI=0000
DS=076B  ES=075A  SS=0E6D  CS=076A  IP=0005   NV UP EI PL NZ NA PO NC
076A:0005 8D360C70      LEA     SI,[700C]                       DS:700C=0205
-d ds:700c
076B:7000                                       05 02 03 04                ....
076B:7010  05 19 88 45 19 5E 5F 8B-E5 5D C2 04 00 55 8B EC   ...E.^_..]...U..
076B:7020  80 3E 9E 09 00 75 07 80-3E 2C 08 00 74 20 80 3E   .>...u..>,..t .>
076B:7030  9E 09 00 74 10 80 3E 2E-08 00 74 09 8A 46 04 2A   ...t..>...t..F.*
076B:7040  E4 50 E8 19 80 8A 46 04-2A E4 50 E8 48 C6 80 3E   .P....F.*.P.H..>
076B:7050  2E 08 00 74 0A 83 6B 07-00 00 05 00 6A 07 A3 01   ...t..k.....j...
076B:7060  E5 5D C2 02 00 55 8B EC-83 7E 08 07 76 20 8B 46   .]...U...~..v .F
076B:7070  08 B1 06 D3 E0 8B 56 06-B1 03 D3 E2 03 C2 2D FC   ......V.......-.
076B:7080  01 50 E8 98 FF FE 0E 90-1D FF 76 04               .P........v.
-
```

Nos.

After execution (squares)

```
Program terminated normally
-d ds:7000
076B:7000  00 00 00 00 00 00 00 00-00 00 00 00 05 04 09 10   ................
076B:7010  19 71 88 45 19 5E 5F 8B-E5 5D C2 04 00 55 8B EC   .q.E.^_..]...U..
076B:7020  80 3E 9E 09 00 75 07 80-3E 2C 08 00 74 20 80 3E   .>...u..>,..t .>
076B:7030  9E 09 00 74 10 80 3E 2E-08 00 74 09 8A 46 04 2A   ...t..>...t..F.*
076B:7040  E4 50 E8 19 80 8A 46 04-2A E4 50 E8 48 C6 80 3E   .P....F.*.P.H..>
076B:7050  2E 08 00 74 0A 83 6B 07-00 00 1B 00 6A 07 07 72   ...t..k.....j..r
076B:7060  E5 5D C2 02 00 55 8B EC-83 7E 08 07 76 20 8B 46   .]...U...~..v .F
076B:7070  08 B1 06 D3 E0 8B 56 06-B1 03 D3 E2 03 C2 2D FC   ......V.......-.
```

Assemble Directives

DB

DW

SEGMENT
ENDS

ASSUME

ORG
END
EVEN
EQU

PROC
FAR
NEAR
ENDP

SHORT

MACRO
ENDM

- **Define Byte**
- **Define a byte type (8-bit) variable**
- **Reserves specific amount of memory locations to each variable**
- **Range : 00_H – FF_H for unsigned value;**
  00_H – $7F_H$ for positive value and
  80_H – FF_H for negative value
- **General form : variable DB value / values**

Example:

LIST DB 7FH, 42H, 35H

Three consecutive memory locations are reserved for the variable LIST and each data specified in the instruction are stored as initial value in the reserved memory location

Assemble Directives

DB

DW

SEGMENT
ENDS

ASSUME

ORG
END
EVEN
EQU

PROC
FAR
NEAR
ENDP

SHORT

MACRO
ENDM

- Define Word
- Define a word type (16-bit) variable
- Reserves two consecutive memory locations to each variable
- Range : 0000_H – $FFFF_H$ for unsigned value;
  0000_H – $7FFF_H$ for positive value and
  8000_H – $FFFF_H$ for negative value
- General form : variable DW value / values

Example:

```
ALIST DW 6512H, 0F251H, 0CDE2H
```

Six consecutive memory locations are reserved for the variable ALIST and each 16-bit data specified in the instruction is stored in two consecutive memory location.

Assemble Directives

DB

DW

SEGMENT
ENDS

ASSUME

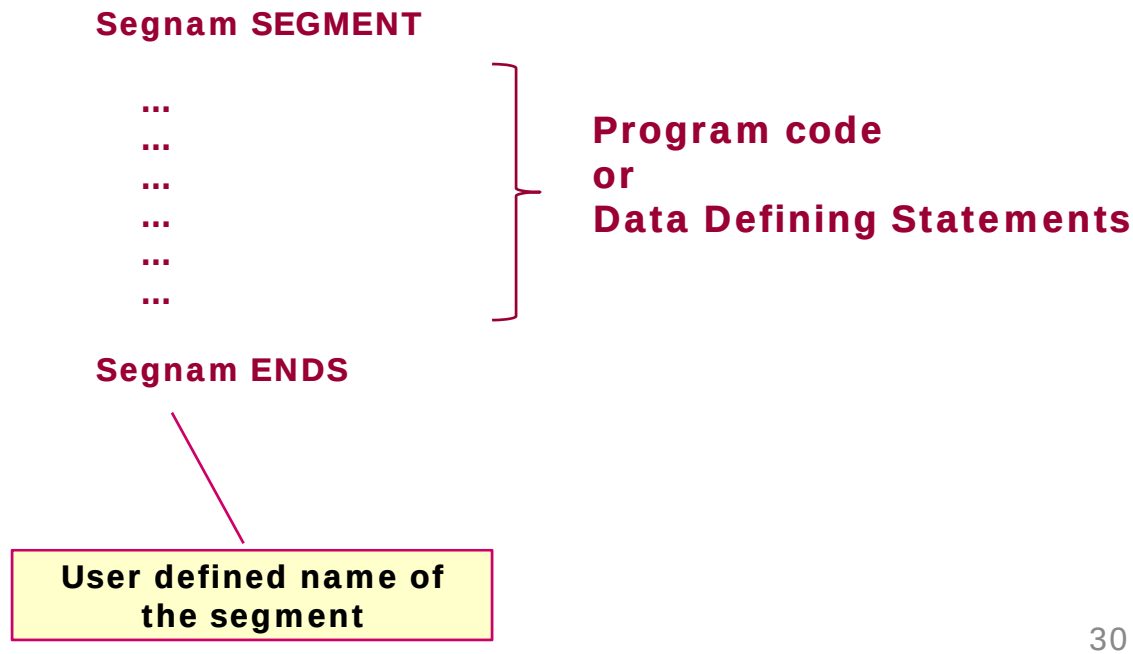
ORG
END
EVEN
EQU

PROC
FAR
NEAR
ENDP

SHORT

MACRO
ENDM

- **SEGMENT : Used to indicate the beginning of a code/ data/ stack segment**
- **ENDS : Used to indicate the end of a code/ data/ stack segment**
- **General form:**



Assemble Directives

DB

DW

SEGMENT
ENDS

ASSUME

ORG
END
EVEN
EQU

PROC
FAR
NEAR
ENDP

SHORT

MACRO
ENDM

- Informs the assembler the name of the program / data segment that should be used for a specific segment.
- General form:

ASSUME segreg : segnam, .. , segreg : segnam

Segment Register	User defined name of the segment

Example:

ASSUME CS: ACODE, DS:ADATA	Tells the compiler that the instructions of the program are stored in the segment ACODE and data are stored in the segment ADATA

Assemble Directives

DB

DW

SEGMENT
ENDS

ASSUME

ORG
END
EVEN
EQU

PROC
FAR
NEAR
ENDP

SHORT

MACRO
ENDM

- **ORG** (Origin) is used to assign the starting address (Effective address) for a program / data segment
- **END** is used to terminate a program; statements after END will be ignored
- **EVEN** : Informs the assembler to store program / data segment starting from an even address
- **EQU** (Equate) is used to attach a value to a variable

Examples:

ORG 1000H	Informs the assembler that the statements following ORG 1000H should be stored in memory starting with effective address 1000_H
LOOP EQU 10FEH	Value of variable LOOP is $10FE_H$
_SDATA SEGMENT ORG 1200H A DB 4CH EVEN B DW 1052H _SDATA ENDS	In this data segment, effective address of memory location assigned to A will be 1200_H and that of B will be 1202_H and 1203_H.

Assemble Directives

DB

DW

SEGMENT
ENDS

ASSUME

ORG
END
EVEN
EQU

PROC
ENDP
FAR
NEAR

SHORT

MACRO
ENDM

- **PROC** Indicates the beginning of a procedure
- **ENDP** End of procedure
- **FAR** Intersegment call
- **NEAR** Intrasegment call
- General form

```
procname PROC[NEAR/ FAR]
        ...
        ...
        ...
        RET
procname ENDP
```

Program statements of the procedure

Last statement of the procedure

User defined name of the procedure

Assemble Directives

DB

DW

SEGMENT
ENDS

ASSUME

ORG
END
EVEN
EQU

PROC
ENDP
FAR
NEAR

SHORT

MACRO
ENDM

Examples:

ADD64 PROC NEAR ... ... RET ADD64 ENDP	The subroutine/ procedure named ADD64 is declared as NEAR and so the assembler will code the CALL and RET instructions involved in this procedure as near call and return
CONVERT PROC FAR ... ... RET CONVERT ENDP	The subroutine/ procedure named CONVERT is declared as FAR and so the assembler will code the CALL and RET instructions involved in this procedure as far call and return

Assemble Directives

DB

DW

SEGMENT
ENDS

ASSUME

ORG
END
EVEN
EQU

PROC
ENDP
FAR
NEAR

SHORT

MACRO
ENDM

- Reserves one memory location for 8-bit signed displacement in jump instructions

Example:

JMP SHORT AHEAD	The directive will reserve one memory location for 8-bit displacement named AHEAD

Assemble Directives

DB

DW

SEGMENT
ENDS

ASSUME

ORG
END
EVEN
EQU

PROC
ENDP
FAR
NEAR

SHORT

MACRO
ENDM

- **MACRO** Indicate the beginning of a macro
- **ENDM** End of a macro
- General form:

```
macroname MACRO[Arg1, Arg2 ...]
        ...
        ...
        ...
macroname ENDM
```

Program statements in the macro

User defined name of the macro

General format

mnemonic	operand(s)	;comments

MOV	destination,source	;copy source operand to destination

Example:

```
MOV    DX,CX
```

Example 2:

```
MOV    CL,55H
MOV    DL,CL
MOV    AH,DL
MOV    AL,AH
MOV    BH,CL
MOV    CH,BH
```

AH	AL
BH	BL
CH	CL
DH	DL

What if ...

MOV AL,DX

Rule #1:

moving a value that is too large into a register will cause an error

```
MOV   BL,7F2H        ;Illegal: 7F2H is larger than 8 bits
MOV   AX,2FE456H     ;Illegal
```

Rule #2:

Data can be moved directly into nonsegment registers only

(Values cannot be loaded directly into any segment register. To load a value into a segment register, first load it to a nonsegment register and then move it to the segment register.)

```
MOV   AX,2345H          MOV   DI,1400H
MOV   DS,AX             MOV   ES,DI
```

Rule #3:

If a value less than FFH is moved into a 16-bit register, the rest of the bits are assumed to be all zeros.

MOV BX, 5

BX =0005
BH = 00, BL = 05

Macro

```
.model small

print macro q
mov AX,@data
Mov DS,AX
lea dx,q
mov ah,09h
int 21h
endm

.stack
.data
msg db "Fibonacci series is $"
.code
start:
        print msg
        mov dl,30h
        mov ah,02h
        int 21h
        mov dl,31h
        mov ah,02h
        int 21h
        mov bh,00h
        mov bl,01h
        mov cx,05
       l:mov dh,bl
        add bl,bh
        mov bh,dh
        mov dl,bl
        add dl,30h
        mov ah,02h
        int 21h
        loop l
 exit:  mov ax,4c00h ;Invoke DOS interrupt 21h, service 4ch
        int 21h ;to exit program
        end start ;tell assembler to finish
```

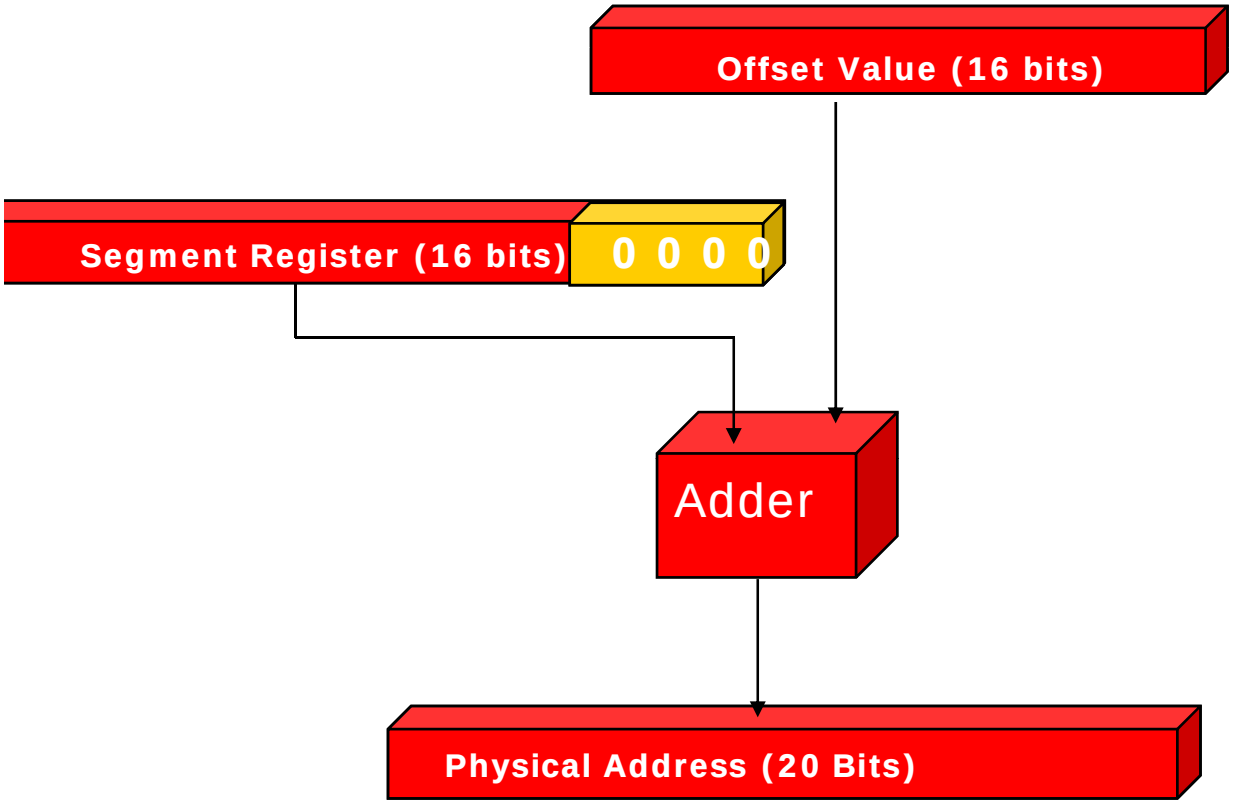
Program Memory Models

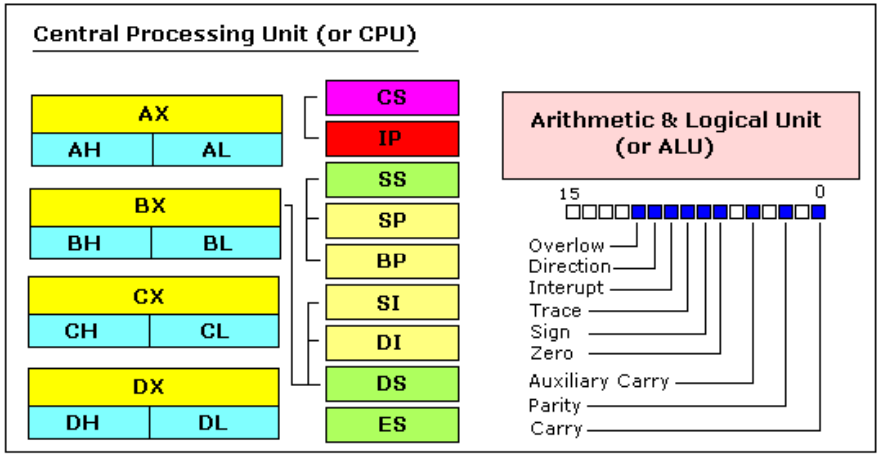
Tiny - single segment shared for code, data & stack (CS = DS = SS)

- Small - single segment for code, separate single segment shared for data & stack (CS DS = SS)
- Medium – multiple segments for code, separate single segment shared for data & stack (CS DS = SS)
- Compact - single segment for code, separate multiple segment for data & stack (CS DS ES SS)
- Large - multiple segments for Code, Data, Stack (CS DS ES SS)

ADDRESSING MODES

Addressing Modes : Memory Access





Addressing Modes

- Every instruction of a program has to operate on a data.
- The different ways in which a source operand is denoted in an instruction are known as addressing modes.

1. Register Addressing
2. Immediate Addressing
3. Direct Addressing
4. Register Indirect Addressing
5. Based Addressing
6. Indexed Addressing
7. Based Index Addressing
8. String Addressing
9. Direct I/O port Addressing
10. Indirect I/O port Addressing
11. Relative Addressing
12. Implied Addressing

Group I : Addressing modes for register and immediate data

Group II : Addressing modes for memory data

Group III : Addressing modes for I/O ports

Group IV : Relative Addressing mode

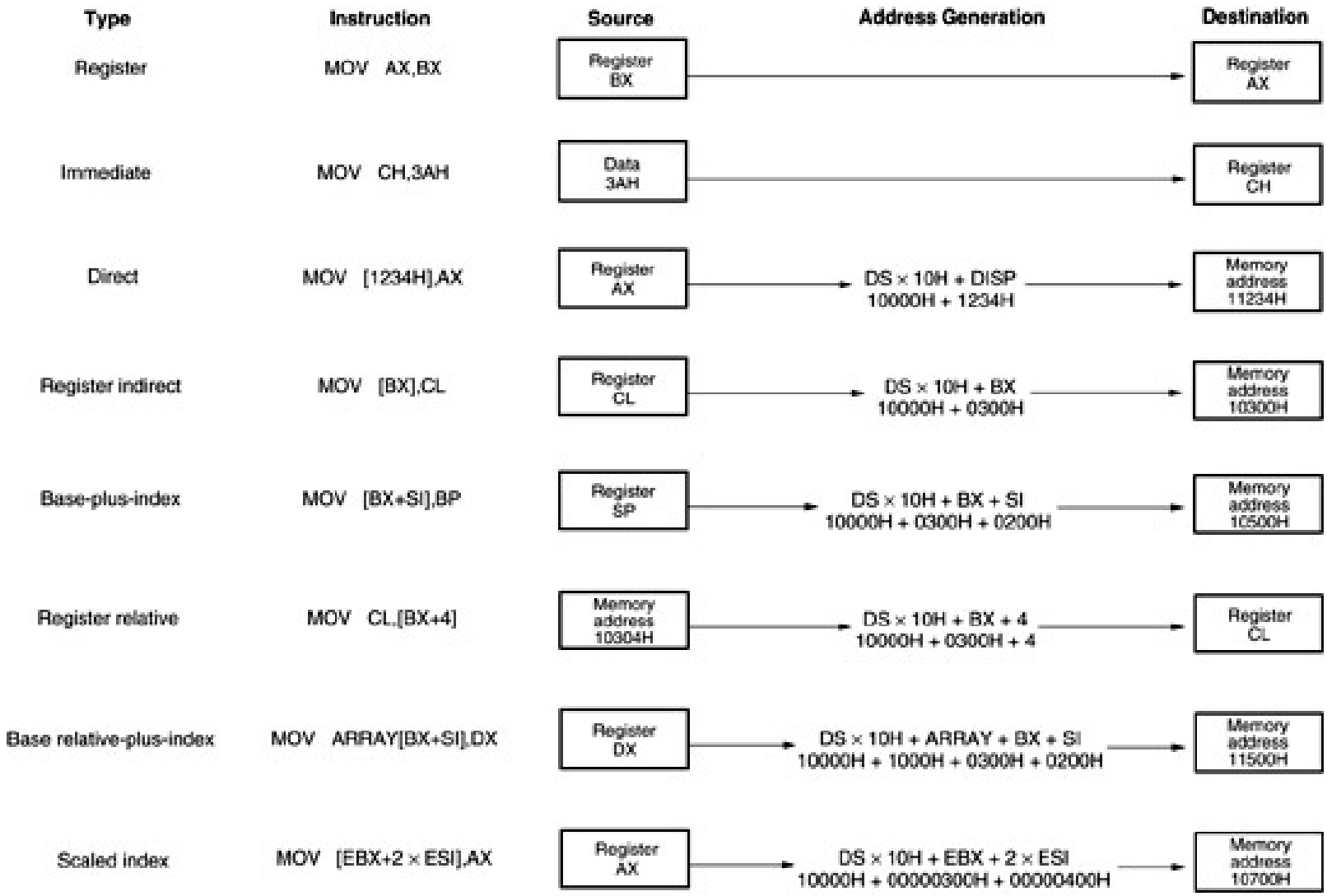
Group V : Implied Addressing mode

Addressing in x86- examples

- Register : MOV AX, BX ; **AX ← BX**
- Immediate : MOV AX, 3CH ; **AX← 3CH**
- Direct : MOV [2000], AX ; **0(DSx10h+2000)← AX**
- Reg indirect:MOV [BX], AX ; **0(DSx10h+BX)←AX**

- Base+Indx:
  MOV [BX+SI], AX ;**(DSx10h+BX+SI)←AX**
- RegRelative:
  MOV [BX+4], AX ;**(DSx10h+BX+4)←AX**
- Base Relative + Index
  MOV ARRAY[BX+SI], AX ;**(DSx10h+ARRAY+BX+SI)←AX**
- Scaled index
  MOV [BX+2 x SI], AX ; **(DSx10h+BX x 2+SI) ←AX**

addressing



Notes: EBX = 00000300H, ESI = 00000200H, ARRAY = 1000H, and DS = 1000H

Addressing Modes

Group I : Addressing modes for register and immediate data

1. Register Addressing
2. Immediate Addressing
3. Direct Addressing
4. Register Indirect Addressing
5. Based Addressing
6. Indexed Addressing
7. Based Index Addressing
8. String Addressing
9. Direct I/O port Addressing
10. Indirect I/O port Addressing
11. Relative Addressing
12. Implied Addressing

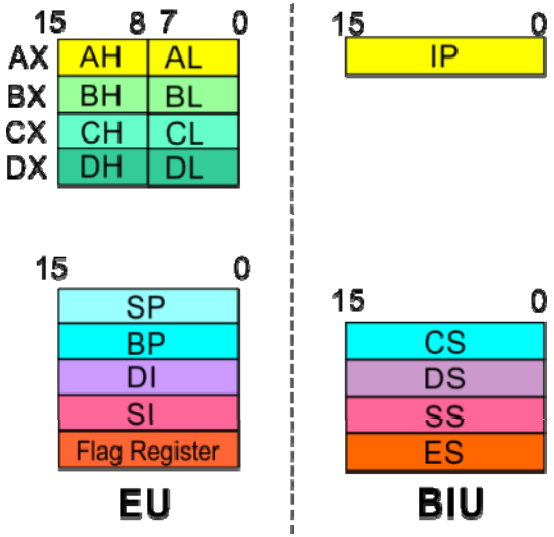
The instruction will specify the name of the register which holds the data to be operated by the instruction.

Example:

MOV CL, DH

The content of 8-bit register DH is moved to another 8-bit register CL

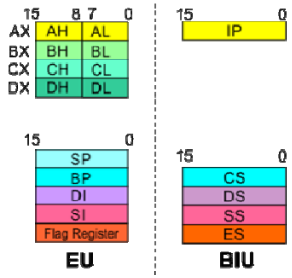
(CL) ← (DH)



Addressing Modes

Group I : Addressing modes for register and immediate data

1. Register Addressing
2. Immediate Addressing
3. Direct Addressing
4. Register Indirect Addressing
5. Based Addressing
6. Indexed Addressing
7. Based Index Addressing
8. String Addressing
9. Direct I/O port Addressing
10. Indirect I/O port Addressing
11. Relative Addressing
12. Implied Addressing



In immediate addressing mode, an 8-bit or 16-bit data is specified as part of the instruction

Example:

MOV DL, 08H

The 8-bit data (08_H) given in the instruction is moved to DL

(DL) ← 08_H

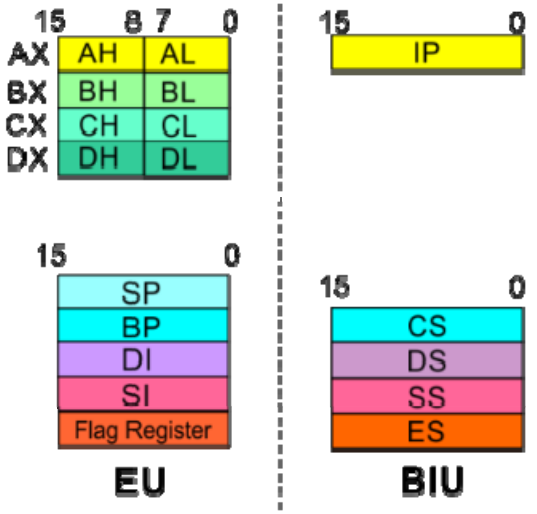
MOV AX, 0A9FH

The 16-bit data ($0A9F_H$) given in the instruction is moved to AX register

(AX) ← $0A9F_H$

Addressing Modes : Memory Access

- 20 Address lines ⇒ 8086 can address up to 2^{20} = 1M bytes of memory
- However, the largest register is only 16 bits
- Physical Address will have to be calculated
  Physical Address : Actual address of a byte in memory. i.e. the value which goes out onto the address bus.
- Memory Address represented in the form –
  Seg : Offset (Eg - 89AB:F012)
- Each time the processor wants to access memory, it takes the contents of a segment register, shifts it one hexadecimal place to the left (same as multiplying by 16_{10}), then add the required offset to form the 20- bit address

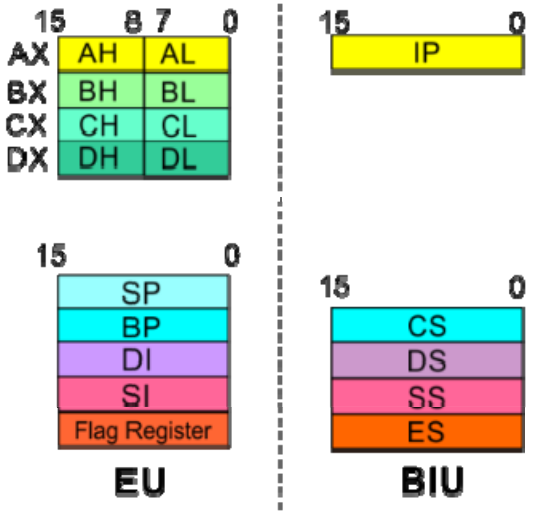


16 bytes of contiguous memory

89AB : F012 → 89AB → 89AB0 (Paragraph to byte → 89AB x 10 = 89AB0)
F012 → 0F012 (Offset is already in byte unit)
+ -------
98AC2 (The absolute address)

Addressing Modes : Memory Access

- **To access memory we use these four registers: BX, SI, DI, BP**
- **Combining these registers inside [ ] symbols, we can get different memory locations (Effective Address, EA)**
- **Supported combinations:**

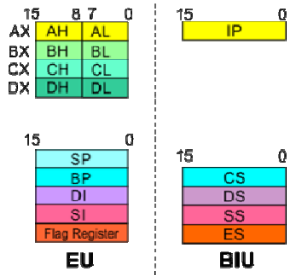


[BX + SI] [BX + DI] [BP + SI] [BP + DI]	[SI] [DI] d16 (variable offset only) [BX]	[BX + SI + d8] [BX + DI + d8] [BP + SI + d8] [BP + DI + d8]
[SI + d8] [DI + d8] [BP + d8] [BX + d8]	[BX + SI + d16] [BX + DI + d16] [BP + SI + d16] [BP + DI + d16]	[SI + d16] [DI + d16] [BP + d16] [BX + d16]

BX **BP**	**SI** **DI**	**+ disp**

Addressing Modes

1. Register Addressing
2. Immediate Addressing
3. Direct Addressing
4. Register Indirect Addressing
5. Based Addressing
6. Indexed Addressing
7. Based Index Addressing
8. String Addressing
9. Direct I/O port Addressing
10. Indirect I/O port Addressing
11. Relative Addressing
12. Implied Addressing



Here, the effective address of the memory location at which the data operand is stored is given in the instruction.

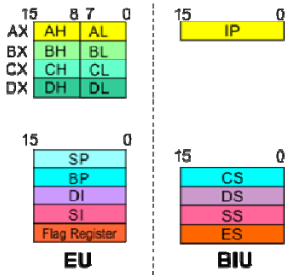
The effective address is just a 16-bit number written directly in the instruction.

Example:

```
MOV  BX, [1354H]
MOV  BL, [0400H]
```

Addressing Modes

1. Register Addressing
2. Immediate Addressing
3. Direct Addressing
4. Register Indirect Addressing
5. Based Addressing
6. Indexed Addressing
7. Based Index Addressing
8. String Addressing
9. Direct I/O port Addressing
10. Indirect I/O port Addressing
11. Relative Addressing
12. Implied Addressing



In Register indirect addressing, name of the register which holds the effective address (EA) will be specified in the instruction.

Registers used to hold EA are any of the following registers:

BX, BP, DI and SI.

Content of the DS register is used for base address calculation.

Example:

MOV CX, [BX]

Note : Register/ memory enclosed in brackets refer to content of register/ memory

Operations:

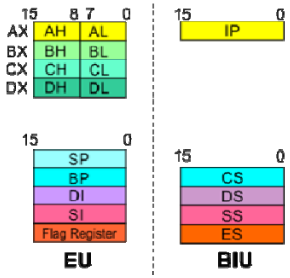
EA = (BX)
BA = (DS) x 16_{10}
MA = BA + EA

(CX) ← (MA) or,

(CL) ← (MA)
(CH) ← (MA +1)

Addressing Modes

1. Register Addressing
2. Immediate Addressing
3. Direct Addressing
4. Register Indirect Addressing
5. Based Addressing
6. Indexed Addressing
7. Based Index Addressing
8. String Addressing
9. Direct I/O port Addressing
10. Indirect I/O port Addressing
11. Relative Addressing
12. Implied Addressing



In Based Addressing, BX or BP is used to hold the base value for effective address and a signed 8-bit or unsigned 16-bit displacement will be specified in the instruction.

In case of 8-bit displacement, it is sign extended to 16-bit before adding to the base value.

When BX holds the base value of EA, 20-bit physical address is calculated from BX and DS.

When BP holds the base value of EA, BP and SS is used.

Example:

MOV AX, [BX + 08H]

Operations:

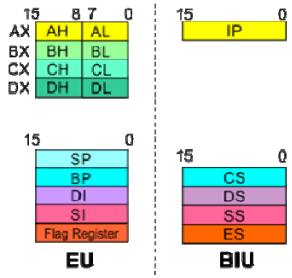
$0008_H \leftarrow 08_H$ (Sign extended)
EA = (BX) + 0008_H
BA = (DS) x 16_{10}
MA = BA + EA

(AX) ← (MA) or,

(AL) ← (MA)
(AH) ← (MA + 1)

Addressing Modes

1. Register Addressing
2. Immediate Addressing
3. Direct Addressing
4. Register Indirect Addressing
5. Based Addressing
6. Indexed Addressing
7. Based Index Addressing
8. String Addressing
9. Direct I/O port Addressing
10. Indirect I/O port Addressing
11. Relative Addressing
12. Implied Addressing



SI or DI register is used to hold an index value for memory data and a signed 8-bit or unsigned 16-bit displacement will be specified in the instruction.

Displacement is added to the index value in SI or DI register to obtain the EA.

In case of 8-bit displacement, it is sign extended to 16-bit before adding to the base value.

Example:

MOV CX, [SI + 0A2H]

Operations:

$FFA2_H \leftarrow A2_H$ (Sign extended)

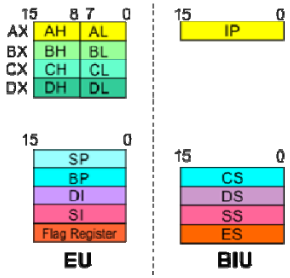
$EA = (SI) + FFA2_H$
$BA = (DS) \times 16_{10}$
$MA = BA + EA$

$(CX) \leftarrow (MA)$ or,

$(CL) \leftarrow (MA)$
$(CH) \leftarrow (MA + 1)$

Addressing Modes

1. Register Addressing
2. Immediate Addressing
3. Direct Addressing
4. Register Indirect Addressing
5. Based Addressing
6. Indexed Addressing
7. Based Index Addressing
8. String Addressing
9. Direct I/O port Addressing
10. Indirect I/O port Addressing
11. Relative Addressing
12. Implied Addressing



In Based Index Addressing, the effective address is computed from the sum of a base register (BX or BP), an index register (SI or DI) and a displacement.

Example:

MOV DX, [BX + SI + 0AH]

Operations:

$000A_H \leftarrow 0A_H$ (Sign extended)

$EA = (BX) + (SI) + 000A_H$
$BA = (DS) \times 16_{10}$
$MA = BA + EA$

$(DX) \leftarrow (MA)$ or,

$(DL) \leftarrow (MA)$
$(DH) \leftarrow (MA + 1)$

Addressing Modes

Group II : Addressing modes for memory data

1. Register Addressing
2. Immediate Addressing
3. Direct Addressing
4. Register Indirect Addressing
5. Based Addressing
6. Indexed Addressing
7. Based Index Addressing
8. String Addressing
9. Direct I/O port Addressing
10. Indirect I/O port Addressing
11. Relative Addressing
12. Implied Addressing

Note : Effective address of the Extra segment register

Employed in string operations to operate on string data.

The effective address (EA) of source data is stored in SI register and the EA of destination is stored in DI register.

Segment register for calculating base address of source data is DS and that of the destination data is ES

Example: MOVS BYTE

Operations:

Calculation of source memory location:

EA = (SI) BA = (DS) x 16_{10} MA = BA + EA

Calculation of destination memory location:

EA_E = (DI) BA_E = (ES) x 16_{10} MA_E = BA_E + EA_E

(MAE) ← (MA)

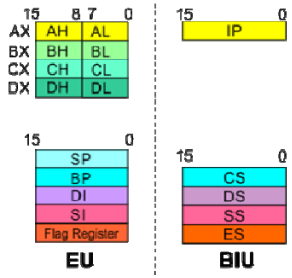
If DF = 1, then (SI) ← (SI) – 1 and (DI) = (DI) - 1

If DF = 0, then (SI) ← (SI) +1 and (DI) = (DI) + 1

Addressing Modes

Group III : Addressing modes for I/O ports

1. Register Addressing
2. Immediate Addressing
3. Direct Addressing
4. Register Indirect Addressing
5. Based Addressing
6. Indexed Addressing
7. Based Index Addressing
8. String Addressing
9. Direct I/O port Addressing
10. Indirect I/O port Addressing
11. Relative Addressing
12. Implied Addressing



These addressing modes are used to access data from standard I/O mapped devices or ports.

Example: IN AL, [0009H]

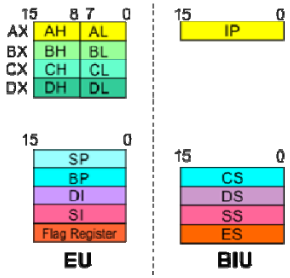
Operations: $PORT_{addr} = 0009_H$
(AL) ← (PORT)

is Content of port with address 0009_H moved to AL register

In indirect port addressing mode, the instruction will specify the name of the register which holds

Addressing Modes

1. Register Addressing
2. Immediate Addressing
3. Direct Addressing
4. Register Indirect Addressing
5. Based Addressing
6. Indexed Addressing
7. Based Index Addressing
8. String Addressing
9. Direct I/O port Addressing
10. Indirect I/O port Addressing
11. Relative Addressing
12. Implied Addressing



In this addressing mode, the effective address of a program instruction is specified relative to Instruction Pointer (IP) by an 8-bit signed displacement.

Example: JZ 0AH

Operations:

$000A_H \leftarrow 0A_H$ (sign extend)

If ZF = 1, then

$EA = (IP) + 000A_H$
$BA = (CS) \times 16_{10}$
$MA = BA + EA$

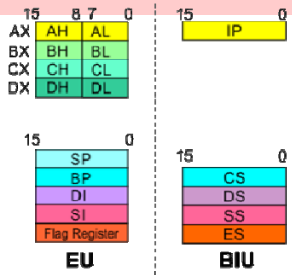
If ZF = 1, then the program control jumps to new address calculated above.

If ZF = 0, then next instruction of the program is executed.

Addressing Modes

Group IV : Implied Addressing mode

1. Register Addressing
2. Immediate Addressing
3. Direct Addressing
4. Register Indirect Addressing
5. Based Addressing
6. Indexed Addressing
7. Based Index Addressing
8. String Addressing
9. Direct I/O port Addressing
10. Indirect I/O port Addressing
11. Relative Addressing
12. Implied Addressing



Instructions using this mode have no operands. The instruction itself will specify the data to be operated by the instruction.

Example: **CLC**

This clears the carry flag to zero.

INSTRUCTION SET

Instruction Set

8086 supports 6 types of instructions.

1. **Data Transfer Instructions**
2. **Arithmetic Instructions**
3. **Logical Instructions**
4. **String manipulation Instructions**
5. **Process Control Instructions**
6. **Control Transfer Instructions**

Instruction Set

1. Data Transfer Instructions

Instructions that are used to transfer data/ address in to registers, memory locations and I/O ports.

Generally involve two operands: Source operand and Destination operand of the same size.

Source: Register or a memory location or an immediate data
Destination : Register or a memory location.

The size should be a either a byte or a word.

A 8-bit data can only be moved to 8-bit register/ memory and a 16-bit data can be moved to 16-bit register/ memory.

Instruction Set

1. Data Transfer Instructions

Mnemonics: MOV, XCHG, PUSH, POP, IN, OUT …

MOV reg2 / mem, reg1 / mem

MOV reg2, reg1	(reg2) ← (reg1)
MOV mem, reg1	(mem) ← (reg1)
MOV reg2, mem	(reg2) ← (mem)

MOV reg / mem, data

MOV reg, data	(reg) ← data
MOV mem, data	(mem) ← data

XCHG reg2 / mem, reg1

XCHG reg2, reg1	(reg2) ↔ (reg1)
XCHG mem, reg1	(mem) ↔ (reg1)

Instruction Set

1. Data Transfer Instructions

Mnemonics: **MOV, XCHG, PUSH, POP, IN, OUT ...**

PUSH reg16 / mem

PUSH reg16

$(SP) \leftarrow (SP) - 2$
$MA_S = (SS) \times 16_{10} + SP$
$(MA_S ; MA_S + 1) \leftarrow (reg16)$

PUSH mem

$(SP) \leftarrow (SP) - 2$
$MA_S = (SS) \times 16_{10} + SP$
$(MA_S ; MA_S + 1) \leftarrow (mem)$

POP reg16 / mem

POP reg16

$MA_S = (SS) \times 16_{10} + SP$
$(reg16) \leftarrow (MA_S ; MA_S + 1)$
$(SP) \leftarrow (SP) + 2$

POP mem

$MA_S = (SS) \times 16_{10} + SP$
$(mem) \leftarrow (MA_S ; MA_S + 1)$
$(SP) \leftarrow (SP) + 2$

Instruction Set

1. Data Transfer Instructions

Mnemonics: **MOV, XCHG, PUSH, POP, IN, OUT ...**

IN A, [DX]

IN AL, [DX]	$PORT_{addr}$ = (DX) (AL) ← (PORT)
IN AX, [DX]	$PORT_{addr}$ = (DX) (AX) ← (PORT)

OUT [DX], A

OUT [DX], AL	$PORT_{addr}$ = (DX) (PORT) ← (AL)
OUT [DX], AX	$PORT_{addr}$ = (DX) (PORT) ← (AX)

IN A, addr8

IN AL, addr8	(AL) ← (addr8)
IN AX, addr8	(AX) ← (addr8)

OUT addr8, A

OUT addr8, AL	(addr8) ← (AL)
OUT addr8, AX	(addr8) ← (AX)

Instruction Set

2. Arithmetic Instructions

Mnemonics: **ADD, ADC, SUB, SBB, INC, DEC, MUL, DIV, CMP…**

ADD reg2/ mem, reg1/mem

ADC reg2, reg1	(reg2) ← (reg1) + (reg2)
ADC reg2, mem	(reg2) ← (reg2) + (mem)
ADC mem, reg1	(mem) ← (mem)+(reg1)

ADD reg/mem, data

ADD reg, data	(reg) ← (reg)+ data
ADD mem, data	(mem) ← (mem)+data

ADD A, data

ADD AL, data8	(AL) ← (AL) + data8
ADD AX, data16	(AX) ← (AX) +data16

Instruction Set

2. Arithmetic Instructions

Mnemonics: **ADD, ADC, SUB, SBB, INC, DEC, MUL, DIV, CMP…**

ADC reg2/ mem, reg1/mem

ADC reg2, reg1	(reg2) ← (reg1) + (reg2)+CF
ADC reg2, mem	(reg2) ← (reg2) + (mem)+CF
ADC mem, reg1	(mem) ← (mem)+(reg1)+CF

ADC reg/mem, data

ADC reg, data	(reg) ← (reg)+ data+CF
ADC mem, data	(mem) ← (mem)+data+CF

ADC A, data

ADC AL, data8	(AL) ← (AL) + data8+CF
ADC AX, data16	(AX) ← (AX) +data16+CF

Instruction Set

2. Arithmetic Instructions

Mnemonics: **ADD, ADC, SUB, SBB, INC, DEC, MUL, DIV, CMP…**

SUB reg2/ mem, reg1/mem

SUB reg2, reg1	(reg2) ← (reg1) - (reg2)
SUB reg2, mem	(reg2) ← (reg2) - (mem)
SUB mem, reg1	(mem) ← (mem) - (reg1)

SUB reg/mem, data

SUB reg, data	(reg) ← (reg) - data
SUB mem, data	(mem) ← (mem) - data

SUB A, data

SUB AL, data8	(AL) ← (AL) - data8
SUB AX, data16	(AX) ← (AX) - data16

Instruction Set

2. Arithmetic Instructions

Mnemonics: **ADD, ADC, SUB, SBB, INC, DEC, MUL, DIV, CMP…**

SBB reg2/ mem, reg1/mem

SBB reg2, reg1	(reg2) ← (reg1) - (reg2) - CF
SBB reg2, mem	(reg2) ← (reg2) - (mem)- CF
SBB mem, reg1	(mem) ← (mem) - (reg1) –CF

SBB reg/mem, data

SBB reg, data	(reg) ← (reg) – data - CF
SBB mem, data	(mem) ← (mem) - data - CF

SBB A, data

SBB AL, data8	(AL) ← (AL) - data8 - CF
SBB AX, data16	(AX) ← (AX) - data16 - CF

Instruction Set

2. Arithmetic Instructions

Mnemonics: **ADD, ADC, SUB, SBB, INC, DEC, MUL, DIV, CMP...**

INC reg/ mem

INC reg8	(reg8) ← (reg8) + 1
INC reg16	(reg16) ← (reg16) + 1
INC mem	(mem) ← (mem) + 1

DEC reg/ mem

DEC reg8	(reg8) ← (reg8) - 1
DEC reg16	(reg16) ← (reg16) - 1
DEC mem	(mem) ← (mem) - 1

Instruction Set

2. Arithmetic Instructions

Mnemonics: **ADD, ADC, SUB, SBB, INC, DEC, MUL, DIV, CMP…**

MUL reg/ mem

MUL reg	**For byte : (AX) ← (AL) x (reg8)** **For word : (DX)(AX) ← (AX) x (reg16)**
MUL mem	**For byte : (AX) ← (AL) x (mem8)** **For word : (DX)(AX) ← (AX) x (mem16)**

IMUL reg/ mem

IMUL reg	**For byte : (AX) ← (AL) x (reg8)** **For word : (DX)(AX) ← (AX) x (reg16)**
IMUL mem	**For byte : (AX) ← (AX) x (mem8)** **For word : (DX)(AX) ← (AX) x (mem16)**

Signed Multiply (imul)

Imul

The following instructions perform 8-bit signed multiplication of **(-4 × 4)**, producing **-16** in AX:

```
mov al, -4
mov bl, 4
 imul bl ; AX = FFF0h, OF = 0
```

Imul

The following instructions perform 8-bit signed multiplication of **(-4 × 4)**, producing **-16** in AX:

```
mov al, -4
mov bl, 4
 imul bl ; AX = FFF0h, OF = 0
```

Instruction Set

2. Arithmetic Instructions

Mnemonics: ADD, ADC, SUB, SBB, INC, DEC, MUL, DIV, CMP…

DIV reg/ mem

DIV reg

For 16-bit :- 8-bit :
(AL) ← (AX) :- (reg8) Quotient
(AH) ← (AX) MOD(reg8) Remainder

For 32-bit :- 16-bit :
(AX) ← (DX)(AX) :- (reg16) Quotient
(DX) ← (DX)(AX) MOD(reg16) Remainder

DIV mem

For 16-bit :- 8-bit :
(AL) ← (AX) :- (mem8) Quotient
(AH) ← (AX) MOD(mem8) Remainder

For 32-bit :- 16-bit :
(AX) ← (DX)(AX) :- (mem16) Quotient
(DX) ← (DX)(AX) MOD(mem16) Remainder

Instruction Set

2. Arithmetic Instructions

Mnemonics: **ADD, ADC, SUB, SBB, INC, DEC, MUL, DIV, CMP…**

IDIV reg/ mem

IDIV reg

For 16-bit :- 8-bit :
(AL) ← (AX) :- (reg8) Quotient
(AH) ← (AX) MOD(reg8) Remainder

For 32-bit :- 16-bit :
(AX) ← (DX)(AX) :- (reg16) Quotient
(DX) ← (DX)(AX) MOD(reg16) Remainder

IDIV mem

For 16-bit :- 8-bit :
(AL) ← (AX) :- (mem8) Quotient
(AH) ← (AX) MOD(mem8) Remainder

For 32-bit :- 16-bit :
(AX) ← (DX)(AX) :- (mem16) Quotient
(DX) ← (DX)(AX) MOD(mem16) Remainder

Instruction Set

2. Arithmetic Instructions

Mnemonics: **ADD, ADC, SUB, SBB, INC, DEC, MUL, DIV, CMP...**

CMP reg2/mem, reg1/ mem

CMP reg2, reg1	**Modify flags ← (reg2) – (reg1)**
	If (reg2) > (reg1) then CF=0, ZF=0, SF=0 **If (reg2) < (reg1) then CF=1, ZF=0, SF=1** **If (reg2) = (reg1) then CF=0, ZF=1, SF=0**
CMP reg2, mem	**Modify flags ← (reg2) – (mem)**
	If (reg2) > (mem) then CF=0, ZF=0, SF=0 **If (reg2) < (mem) then CF=1, ZF=0, SF=1** **If (reg2) = (mem) then CF=0, ZF=1, SF=0**
CMP mem, reg1	**Modify flags ← (mem) – (reg1)**
	If (mem) > (reg1) then CF=0, ZF=0, SF=0 **If (mem) < (reg1) then CF=1, ZF=0, SF=1** **If (mem) = (reg1) then CF=0, ZF=1, SF=0**

Instruction Set

2. Arithmetic Instructions

Mnemonics: **ADD, ADC, SUB, SBB, INC, DEC, MUL, DIV, CMP...**

CMP reg/mem, data

CMP reg, data

Modify flags ← (reg) – (data)

If (reg) > data then CF=0, ZF=0, SF=0
If (reg) < data then CF=1, ZF=0, SF=1
If (reg) = data then CF=0, ZF=1, SF=0

CMP mem, data

Modify flags ← (mem) – (mem)

If (mem) > data then CF=0, ZF=0, SF=0
If (mem) < data then CF=1, ZF=0, SF=1
If (mem) = data then CF=0, ZF=1, SF=0

Instruction Set

2. Arithmetic Instructions

Mnemonics: **ADD, ADC, SUB, SBB, INC, DEC, MUL, DIV, CMP...**

CMP A, data

CMP AL, data8	**Modify flags ← (AL) – data8** **If (AL) > data8 then CF=0, ZF=0, SF=0** **If (AL) < data8 then CF=1, ZF=0, SF=1** **If (AL) = data8 then CF=0, ZF=1, SF=0**
CMP AX, data16	**Modify flags ← (AX) – data16** **If (AX) > data16 then CF=0, ZF=0, SF=0** **If (mem) < data16 then CF=1, ZF=0, SF=1** **If (mem) = data16 then CF=0, ZF=1, SF=0**

Instruction Set

3. Logical Instructions

Mnemonics: AND, OR, XOR, TEST, SHR, SHL, RCR, RCL ...

AND A, data AND AL, data8	(AL) ← (AL) & data8
AND AX, data16	(AX) ← (AX) & data16

AND reg/mem, data AND reg, data	(reg) ← (reg) & data
AND mem, data	(mem) ← (mem) & data

Instruction Set

3. Logical Instructions

Mnemonics: **AND, OR, XOR, TEST, SHR, SHL, RCR, RCL ...**

OR reg2/mem, reg1/mem OR reg2, reg1	(reg2) ← (reg2) | (reg1)
OR reg2, mem	(reg2) ← (reg2) | (mem)
OR mem, reg1	(mem) ← (mem) | (reg1)

OR reg/mem, data OR reg, data	(reg) ← (reg) | data
OR mem, data	(mem) ← (mem) | data

OR A, data OR AL, data8	(AL) ← (AL) | data8
OR AX, data16	(AX) ← (AX) | data16

Instruction Set

3. Logical Instructions

Mnemonics: AND, OR, XOR, TEST, SHR, SHL, RCR, RCL ...

XOR reg2/mem, reg1/mem	
XOR reg2, reg1	(reg2) ← (reg2) ^ (reg1)
XOR reg2, mem	(reg2) ← (reg2) ^ (mem)
XOR mem, reg1	(mem) ← (mem) ^ (reg1)

XOR reg/mem, data	
XOR reg, data	(reg) ← (reg) ^ data
XOR mem, data	(mem) ← (mem) ^ data

XOR A, data	
XOR AL, data8	(AL) ← (AL) ^ data8
XOR AX, data16	(AX) ← (AX) ^ data16

Instruction Set

3. Logical Instructions

Mnemonics: **AND, OR, XOR, TEST, SHR, SHL, RCR, RCL ...**

TEST reg2/mem, reg1/mem	
TEST reg2, reg1	Modify flags ← (reg2) & (reg1)
TEST reg2, mem	Modify flags ← (reg2) & (mem)
TEST mem, reg1	Modify flags ← (mem) & (reg1)

TEST reg/mem, data	
TEST reg, data	Modify flags ← (reg) & data
TEST mem, data	Modify flags ← (mem) & data

TEST A, data	
TEST AL, data8	Modify flags ← (AL) & data8
TEST AX, data16	Modify flags ← (AX) & data16

Instruction Set

3. Logical Instructions

Mnemonics: **AND, OR, XOR, TEST, SHR, SHL, RCR, RCL ...**

SHR reg/mem

SHR reg

i) SHR reg, 1

ii) SHR reg, CL

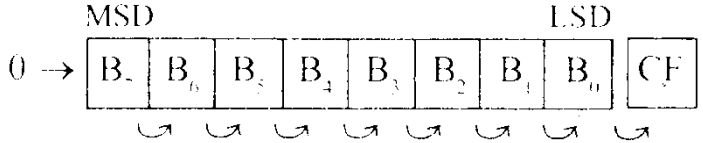
SHR mem

i) SHR mem, 1

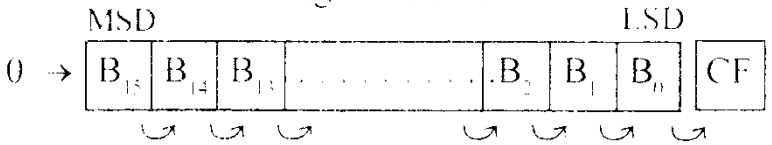
ii) SHR mem, CL

$$CF \leftarrow B_{LSD} ; B_n \leftarrow B_{n+1} ; B_{MSD} \leftarrow 0$$

reg 8 / mem 8



reg 16 / mem 16



Instruction Set

3. Logical Instructions

Mnemonics: **AND, OR, XOR, TEST, SHR, SHL, RCR, RCL ...**

SHL reg/mem or SAL reg/mem

SHL reg or SAL reg

i) SHL reg, 1 or SAL reg, 1

ii) SHL reg, CL or SAL reg, CL

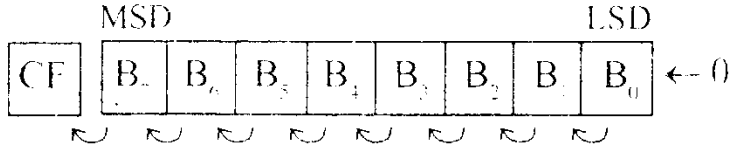
SHL mem or SAL mem

i) SHL mem, 1 or SAL mem, 1

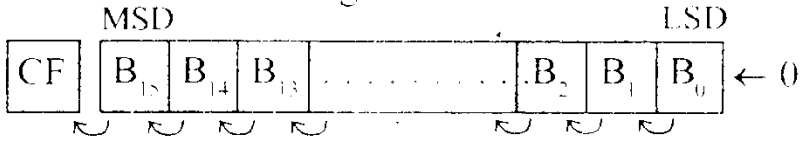
ii) SHL mem, CL or SAL mem, CL

$CF \leftarrow B_{MSD} ; B_{n+1} \leftarrow B_n ; B_{LSD} \leftarrow 0$

reg 8 / mem 8



reg 16 / mem 16



Instruction Set

3. Logical Instructions

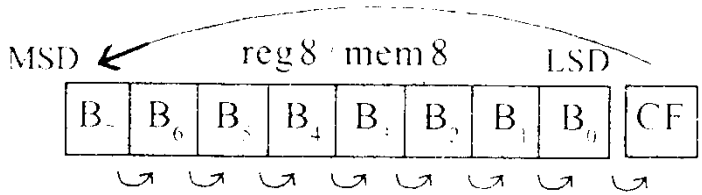
Mnemonics: **AND, OR, XOR, TEST, SHR, SHL, RCR, RCL ...**

RCR reg/mem

RCR reg

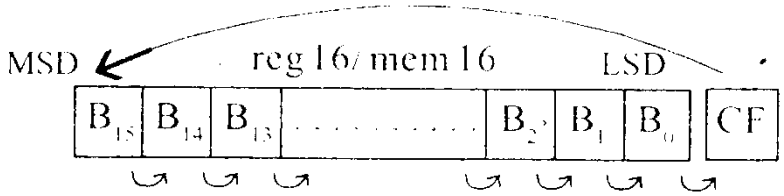
i) RCR reg, 1
ii) RCR reg, CL

$B_n \leftarrow B_{n-1}$; $B_{MSD} \leftarrow CF$; $CF \leftarrow B_{LSD}$



RCR mem

i) RCR mem, 1
ii) RCR mem, CL



Instruction Set

3. Logical Instructions

Mnemonics: **AND, OR, XOR, TEST, SHR, SHL, RCR, RCL ...**

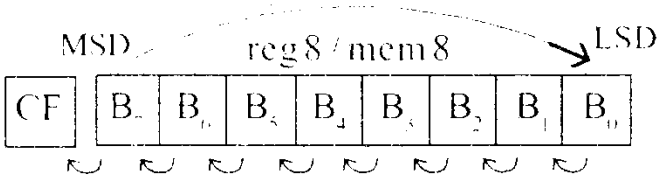
ROL reg/mem

ROL reg

i) ROL reg, 1

ii) ROL reg, CL

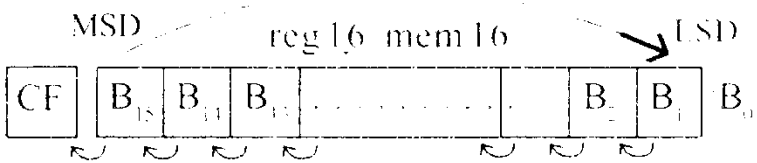
$B_{n+1} \leftarrow B_n : CF \leftarrow B_{MSD} ; B_{LSD} \leftarrow B_{MSD}$



ROL mem

i) ROL mem, 1

ii) ROL mem, CL



Instruction Set

4. String Manipulation Instructions

- String : Sequence of bytes or words
- 8086 instruction set includes instruction for string movement, comparison, scan, load and store.
- REP instruction prefix : used to repeat execution of string instructions
- String instructions end with S or SB or SW.
  S represents string, SB string byte and SW string word.
- Offset or effective address of the source operand is stored in SI register and that of the destination operand is stored in DI register.
- Depending on the status of DF, SI and DI registers are automatically updated.
- DF = 0 ⇒ SI and DI are incremented by 1 for byte and 2 for word.
- DF = 1 ⇒ SI and DI are decremented by 1 for byte and 2 for word.

Instruction Set

4. String Manipulation Instructions

Mnemonics: **REP, MOVS, CMPS, SCAS, LODS, STOS**

REP

REPZ / REPE **(Repeat CMPS or SCAS until ZF = 0)**	**While CX ≠ 0 and ZF = 1, repeat execution of string instruction and (CX) ← (CX) – 1**
REPNZ / REPNE **(Repeat CMPS or SCAS until ZF = 1)**	**While CX ≠ 0 and ZF = 0, repeat execution of string instruction and (CX) ← (CX) - 1**

Instruction Set

4. String Manipulation Instructions

Mnemonics: REP, MOVS, CMPS, SCAS, LODS, STOS

MOVS

MOVSB

$MA = (DS) \times 16_{10} + (SI)$
$MA_E = (ES) \times 16_{10} + (DI)$

$(MA_E) \leftarrow (MA)$

If DF = 0, then $(DI) \leftarrow (DI) + 1$; $(SI) \leftarrow (SI) + 1$
If DF = 1, then $(DI) \leftarrow (DI) - 1$; $(SI) \leftarrow (SI) - 1$

MOVSW

$MA = (DS) \times 16_{10} + (SI)$
$MA_E = (ES) \times 16_{10} + (DI)$

$(MA_E ; MA_E + 1) \leftarrow (MA; MA + 1)$

If DF = 0, then $(DI) \leftarrow (DI) + 2$; $(SI) \leftarrow (SI) + 2$
If DF = 1, then $(DI) \leftarrow (DI) - 2$; $(SI) \leftarrow (SI) - 2$

Instruction Set

4. String Manipulation Instructions

Mnemonics: REP, MOVS, CMPS, SCAS, LODS, STOS

Compare two string byte or string word

CMPS

CMPSB

MA = (DS) x 16_{10} + (SI)
MA_E = (ES) x 16_{10} + (DI)

Modify flags ← (MA) - (MA_E)

If (MA) > (MA_E), then CF = 0; ZF = 0; SF = 0
If (MA) < (MA_E), then CF = 1; ZF = 0; SF = 1

CMPSW

If (MA) = (MA_E), then CF = 0; ZF = 1; SF = 0

For byte operation
If DF = 0, then (DI) ← (DI) + 1; (SI) ← (SI) + 1
If DF = 1, then (DI) ← (DI) - 1; (SI) ← (SI) - 1

For word operation
If DF = 0, then (DI) ← (DI) + 2; (SI) ← (SI) + 2
If DF = 1, then (DI) ← (DI) - 2; (SI) ← (SI) - 2

Instruction Set

4. String Manipulation Instructions

Mnemonics: REP, MOVS, CMPS, SCAS, LODS, STOS

Scan (compare) a string byte or word with accumulator

SCAS

SCASB

MA_E = (ES) x 16_{10} + (DI)
Modify flags ← (AL) - (MA_E)

If (AL) > (MA_E), then CF = 0; ZF = 0; SF = 0
If (AL) < (MA_E), then CF = 1; ZF = 0; SF = 1
If (AL) = (MA_E), then CF = 0; ZF = 1; SF = 0

If DF = 0, then (DI) ← (DI) + 1
If DF = 1, then (DI) ← (DI) – 1

SCASW

MA_E = (ES) x 16_{10} + (DI)
Modify flags ← (AL) - (MA_E)

If (AX) > (MA_E ; MA_E + 1), then CF = 0; ZF = 0; SF = 0
If (AX) < (MA_E ; MA_E + 1), then CF = 1; ZF = 0; SF = 1
If (AX) = (MA_E ; MA_E + 1), then CF = 0; ZF = 1; SF = 0

If DF = 0, then (DI) ← (DI) + 2
If DF = 1, then (DI) ← (DI) – 2

Instruction Set

4. String Manipulation Instructions

Mnemonics: REP, MOVS, CMPS, SCAS, LODS, STOS

Load string byte in to AL or string word in to AX

LODS

LODSB

MA = (DS) x 16_{10} + (SI)
(AL) ← (MA)

If DF = 0, then (SI) ← (SI) + 1
If DF = 1, then (SI) ← (SI) – 1

LODSW

MA = (DS) x 16_{10} + (SI)
(AX) ← (MA ; MA + 1)

If DF = 0, then (SI) ← (SI) + 2
If DF = 1, then (SI) ← (SI) – 2

Example: Convert all string characters to upper case using load and store string instructions.

```
        MOV SI, offset Array
        MOV DI, SI MOV CX, 100 ; Array length
        MOV AX, DS
        MOV ES, AX
        CLD
Next:   LODSB
          OR AL, 20H ; convert to upper case
          STOSB
          LOOP Next
```

Instruction Set

4. String Manipulation Instructions

Mnemonics: **REP, MOVS, CMPS, SCAS, LODS, STOS**

Store byte from AL or word from AX in to string

STOS

STOSB

$MA_E = (ES) \times 16_{10} + (DI)$
$(MA_E) \leftarrow (AL)$

If DF = 0, then $(DI) \leftarrow (DI) + 1$
If DF = 1, then $(DI) \leftarrow (DI) - 1$

STOSW

$MA_E = (ES) \times 16_{10} + (DI)$
$(MA_E ; MA_E + 1) \leftarrow (AX)$

If DF = 0, then $(DI) \leftarrow (DI) + 2$
If DF = 1, then $(DI) \leftarrow (DI) - 2$

```
;Example: Copy 100 bytes from Array1 to Array2, using the MOV instruction.

        LEA DI, Array2  ; Starting address of Destination
        LEA SI, Array1          ; Starting address of Source
        MOV CX, 100             ; Number of elements = 100
Next:   MOV AL, [SI]            ; AL   <== [SI]
        MOV [DI], AL            ; [DI] <== AL
        INC SI                  ; SI + 1
        INC DI                  ; DI + 1
        LOOP Next               ; Next element
```

Example: Copy 100 bytes from Array1 to Array2, using MOVSB instruction.

```
        LEA DI, Array2  ; Starting address of Destination
        LEA SI, Array1          ; Starting address of Source
        MOV CX, 100             ; Number of elements = 100
        PUSH DS
        POP ES                  ; make ES=DS
        CLD                     ; set SI and DI to auto-increment
next:   MOVSB
        LOOP next
```

Instruction Set

5. Processor Control Instructions

Mnemonics	Explanation
STC	Set CF ← 1
CLC	Clear CF ← 0
CMC	Complement carry CF ← CF′
STD	Set direction flag DF ← 1
CLD	Clear direction flag DF ← 0
STI	Set interrupt enable flag IF ← 1
CLI	Clear interrupt enable flag IF ← 0
NOP	No operation
HLT	Halt after interrupt is set
WAIT	Wait for TEST pin active
ESC opcode mem / reg	Used to pass instruction to a coprocessor which shares the address and data bus with the 8086
LOCK	Lock bus during next instruction

Instruction Set

6. Control Transfer Instructions

- Transfer the control to a specific destination or target instruction
- Do not affect flags

❑ 8086 Unconditional transfers

Mnemonics	Explanation
CALL reg/ mem/ disp16	Call subroutine
RET	Return from subroutine
JMP reg/ mem/ disp8/ disp16	Unconditional jump

Instruction Set

6. Control Transfer Instructions

- 8086 signed conditional branch instructions
- 8086 unsigned conditional branch instructions

- Checks flags
- If conditions are true, the program control is transferred to the new memory location in the same segment by modifying the content of IP

Instruction Set

6. Control Transfer Instructions

- 8086 signed conditional branch instructions

Name	Alternate name
JE disp8 Jump if equal	JZ disp8 Jump if result is 0
JNE disp8 Jump if not equal	JNZ disp8 Jump if not zero
JG disp8 Jump if greater	JNLE disp8 Jump if not less or equal
JGE disp8 Jump if greater than or equal	JNL disp8 Jump if not less
JL disp8 Jump if less than	JNGE disp8 Jump if not greater than or equal
JLE disp8 Jump if less than or equal	JNG disp8 Jump if not greater

- 8086 unsigned conditional branch instructions

Name	Alternate name
JE disp8 Jump if equal	JZ disp8 Jump if result is 0
JNE disp8 Jump if not equal	JNZ disp8 Jump if not zero
JA disp8 Jump if above	JNBE disp8 Jump if not below or equal
JAE disp8 Jump if above or equal	JNB disp8 Jump if not below
JB disp8 Jump if below	JNAE disp8 Jump if not above or equal
JBE disp8 Jump if below or equal	JNA disp8 Jump if not above

Instruction Set

6. Control Transfer Instructions

- 8086 conditional branch instructions affecting individual flags

Mnemonics	Explanation
JC disp8	Jump if CF = 1
JNC disp8	Jump if CF = 0
JP disp8	Jump if PF = 1
JNP disp8	Jump if PF = 0
JO disp8	Jump if OF = 1
JNO disp8	Jump if OF = 0
JS disp8	Jump if SF = 1
JNS disp8	Jump if SF = 0
JZ disp8	Jump if result is zero, i.e, Z = 1
JNZ disp8	Jump if result is not zero, i.e, Z = 1