Chapter 3 (programming with 8086 microprocessor)

Features of 8086 µP

→ 16 bit µP.
→ 40 pin IC
→ 16 bit data bus
→ 20 bit address bus thus it can address 2^{20} = 1 MB of memory.
→ uses segmented memory. Different memory segments for code, data, stack etc.
→ Register based µP.
→ uses pipelining feature.
→ It prefetches 6 bytes of instruction code from memory and queue them in order to speed up the program execution by overlapping instruction fetch with execution.
→ Internally divided into two units : BIU (Bus Interface Unit) and EU (Execution Unit)

Internal architecture of 8086 µP.

→ Internally divided into BIU and EU.

1. BIU

→ BIU sends out addresses, fetches instructions from the memory, reads data from ports & memory and writes data to ports / memory.
→ fetches upto 6 instruction byte and store these prefetched bytes in a FIFO register called queue.

→ BIU also contains a dedicated adder which is used to generate 20 bit physical address by using the content of segment register and 16 bit offset address.

2. EU

→ EU tells the BIU where to fetch instructions or data from, decodes and executes instructions.

→ The major units of EU are:

i. control ckt : directs internal operation

ii. Decoder : Translates the instruction fetched from queue.

iii. ALU : 16 bit ALU to perform arithmetic & logical instructions like add, substract etc

iv. 16 bit registers : AX, BX, CX, DX, SP, BP, SI, DI.

v. flag register : 16 bit register (9 flags)

Registers of 8086 μp

1. Code segment (CS) Register and Instruction pointer (IP)

→ CS register holds the starting address of memory from where code segment starts

→ IP holds the address of next code byte within the code segment.

→ The value contained in IP is known as offset and it denotes how far the code is located

from the start.
→ Both are 16 bit ~~addresses~~ registers.
→ BIU computes 20 bit physical address using the content of CS and IP.
→ content of CS is shifted left by 4 bits (or multiplied by $(16)_{10} = (10)_{16}$) and 16 bit offset from IP is added to produce 20 bit physical address.
eg: Let, [IP] = 0050 H and [CS] = 8819H
Then physical address = 8819 X 10 + 0050
= 88190 + 0050
= 881E0 H.

2. stack segment (SS) register, stack pointer (SP) and Base pointer (BP).
→ SS register contains the starting address of stack segment (current stack)
→ SP holds the 16 bit offset from the start of stack segment.
→ 20 bit physical address is computed from SS and SP (SS X 10 + SP)
→ PUSH & POP instructions use this.
→ BP is used instead of SP when based addressing mode is used. In this mode, 20 bit address is computed from SS and BP.

3. Data segment (DS) register.
→ DS register points to the starting address of data segment.
→ Operands (data) are fetched from this segment.
→ In order to access operand in data segment, 20 bit physical address is calculated from DS register combined with offset of DI (Destination Index) or SI (Source Index) or 16 bit displacement.
→ DS, SI, DI are 16 bit registers.

4. Extra segment (ES) register
→ If the data segment memory (64 KB) is not enough, then extra segment is used.
→ ES register contains starting address of ES.
→ String operations utilize ES where, 20 bit physical address is computed from ES and DI.

5. AX register.
→ 16 bit accumulator
→ can be divided into two 8 bit registers: AH & AL.
→ AL is used as 8-bit accumulator.
→ I/O operations use AX or AL register.
→ multiplication and division use AX or AL.
→ AL is similar to 8085 accumulator.

6. BX register.
→ called base register.
→ contents are used for addressing memory.
→ divided into BH and BL, each of 8 bits.
→ BH is similar to H and BL is similar to L register of 8085.

7. CX register
→ 16 bit Counter register
→ Shift, rotate, loop etc instructions use CX as counter.
eg: Loop start ; automatically decrements CX by 1 and goes to start until CX is 0.

8. DX register.
→ 16 bit data register. (DH → 8 bit, DL → 8 bit)
→ used to specify port number in I/O operation
→ 16 x 16 bit multiplication, 32/16 bit division make use of data register.

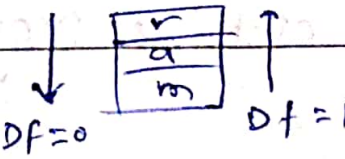
9. Flag register.
→ 16 bit register, contains 9 different flags.
→ out of the 9 flags, 6 are status and 3 are control flags.
→ control flag can be set or reset by programmer

X	X	X	X	O	D	I	T	S	Z	X	Ac	X	P	X	Cy

i. Status flags : shows the condition of result.
a. cy (carry) b. P (parity) c. Ac (Aux. carry)
d. z (zero) e. S (sign)
↳ similar to 8085
f. overflow flag (of)
→ It is set when the result of a signed number operation is large causing higher order bit to overflow into the sign bit.
→ Usually it the size of the result of the operation is greater than that of destination register, of is set.

eg: $+117 = 01110101$
$+55 = 00110111$
$+172 = 10101100$
↳ sign bit = 1 = negative

ii. Control flags : to enable or disable certain µp operations.
a. Direction flag (Df)
→ when Df = 0 : string is processed from lowest address ie. string instruction is autoincremented.
→ when Df = 1 : string is processed from highest address ie. string instruction is autodecremented



b. Interrupt flag (If)

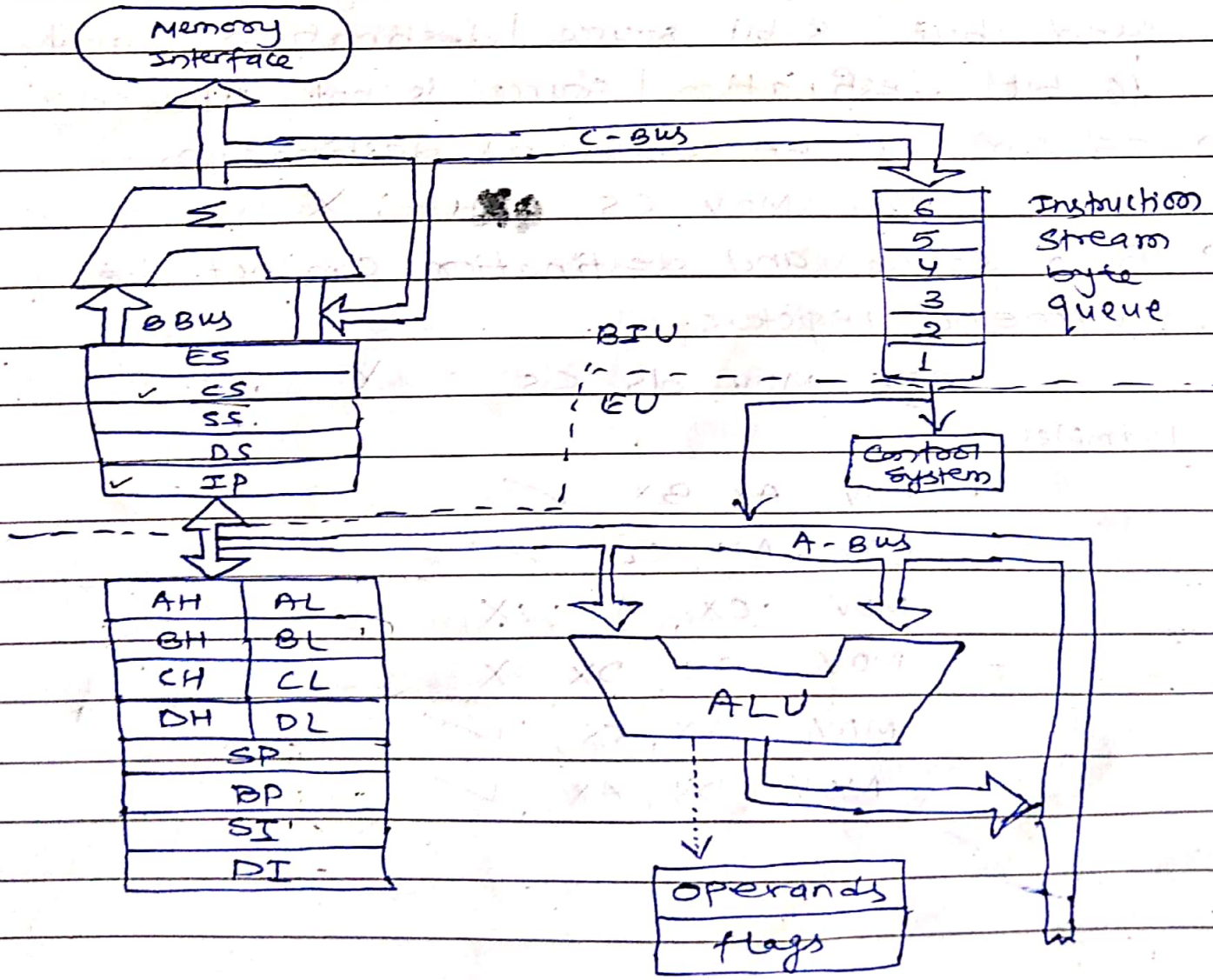
→ If = 0 : Interrupts are disabled

→ If = 1 : Interrupts are enabled.

c. Trap flag (Tf)

→ if Tf = 1, 8086 generates an internal interrupt after execution of each instruction (single step mode)

→ up stops at each instruction and waits for Command from programmer.



8086 Addressing modes

The way in which a processor can access the operand is called addressing mode.

a. Register access.

i. Register addressing mode

→ Both source and destination operands are registers

→ Both 16 bit registers or 8 bit registers can be used but 8 bit source | destination and 16 bit destination | source is not allowed.

→ CS can not be used as destination.

eg: MOV CS, [illegible]H ; X

→ Both source and destination can not be segment registers.

eg: MOV ds, cs ; X

Examples:

MOV AX, BX ✓

MOV AH, CL ✓

MOV CX, AL X

MOV BH, DX X

MOV AX, CS ✓

MOV DS, AX ✓

ii. Immediate addressing mode

→ immediate data is the part of instruction which is located in CS not DS

→ Destination is 8 (or 16) bit register and source is 8 (or 16) bit immediate data

eg:
```
MOV AX, 2437H ✓
MOV AL, 82H ✓
MOV AL, 2437H ✗
MOV DX, offset msg ✓
```

b. Memory access

→ Different addressing modes are used to specify location of operand in memory.

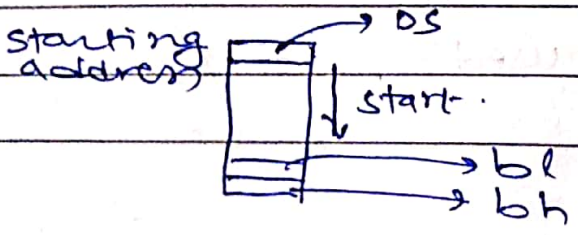
→ EU can not directly access memory : EU sends offset to BIU and BIU calculates 20 bit physical address using contents of segment register and offset sent by EU.

i. Direct addressing mode.

→ physical address is calculated using segment register and displacement value in the operand.

→ unless specified, segment register is DS.

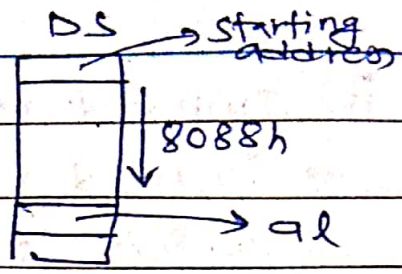
eg: mov bx, start ; moves the content of memory location at address (DS x 10 + start) to bx register



mov al, ds:[8088h] ✓

↕ same

mov al, [8088h]



mov ax, cs:[1234h] ; segment override

where DS is replaced by CS.

[AL] ← [CS x 10 + 1234]

[AH] ← [CS x 10 + 1235]

ii. Register Indirect addressing mode

→ Effective address (offset) is taken directly from one of the four base or index registers ie. BX, BP, SI and DI.

eg: mov al, [bx]

mov ax, [bp] ax ← [SS x 10 + bp]

mov cx, [si]

mov ah, [di]

→ physical address is calculated using segment register and base or index register.

→ The content at that location is moved to specified register.

→ By default, bp uses SS and others use DS.

→ segment override can be used.

iii. Based addressing mode (bx|bp)

→ offset is the sum of displacement and content of bx or bp (base register | pointer)

→ segment register may be DS or SS.

eg: mov ax, 8[bx] ;
mov ax, [bx+8] ;
al ← [ds x 10 + (bx + 8)]
ah ← [ds x 10 + (bx + 8 + 1)]

→ if stack is used, SS and bp are used otherwise DS and BX.

iv. Indexed addressing mode

→ Same as based except SI and DI are used instead of bx and bp.

→ segment involved is DS.

eg: mov ax, 8[si] ; [al] ← [ds x 10 + (si + 8)]
mov ax, [si+8]

v. Based indexed addressing mode

→ combination of based and indexed addressing modes.

→ effective address is computed from the sum of base registers (bx or bp), index registes (si or di) and a displacement.

→ DS is only involved.

eg: mov ax, 4.[bx][si] ; offset = 4+bx+si
mov ax, [bx + si] ; offset = bx + si

vi. string address

→ si is used to point first byte (word) of source string

→ di is used to point first byte (word) of destination

→ ds is default segment for source and es is the must for destination

→ si or di are increased or decreased depending upon DF.

eg: movs word.

Let, [DF] = 0, [DS] = 4000H, [Si] = 0040H, [ES] = 8000H, [Di] = 0080H, [40040H] = FDH, [40041H] = ABH, [80080H] = 54H and [80081H] = 77H

after execution,

4000 x 10 + 40 = 40040H content ie. FDH is moved to 8000 x 10 + 0080 = 80080H

∵ Df = 0, si and di are incremented by 1 (auto)

∴ Content of 40041H = ABH is moved to 80081H and si and di are again incremented by 1 to produce si = 0042H and di = 0082H

c. I/O port addressing

→ 8 bit port is directly specified in the operand.

→ 16 bit port is specified using DX register.

Examples:

IN AL, 28H ✓ ; takes 8 bit data

IN AX, 34H ✓ ; takes 16 bit data

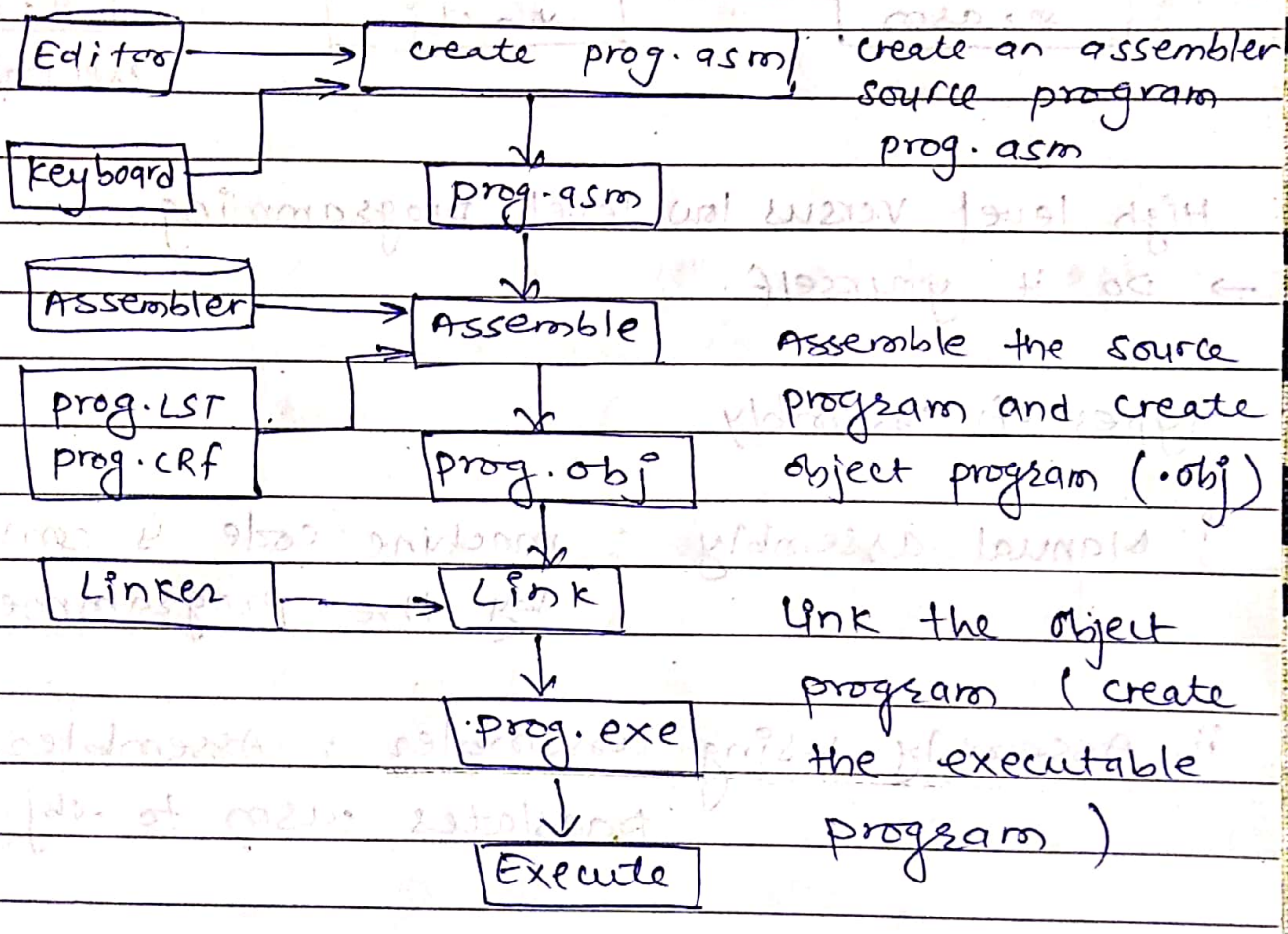
IN AL, DX ✓

IN AX, DX ✓

OUT 3BH, AL ✓ , OUT 3BH, AX ✓

OUT DX, AL ✓ , OUT DX, AX ✓

Assembly language programming.



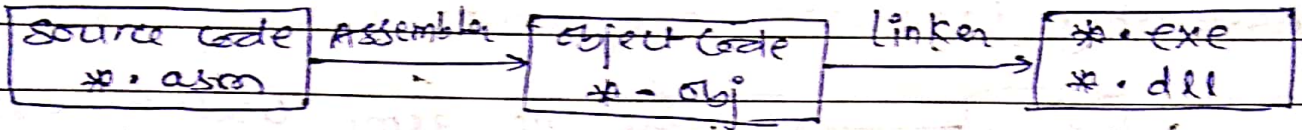
Assembler

→ program written in assembly language (using mnemonics) must be converted to machine level

language to make it understandable for the µp.

→ An assembler translates the program written in assembly language (.asm) into a machine language (.obj) so that the µp can execute the program. The process is known as assembling.

→ Linker converts the object code into executable code (.exe or .dll)



High level versus low level programming.

→ Do it yourself.

Types of assembly

i. Manual assembly : machine code is computed by the programmer.

ii. Assembly using assembler : Assembler translates .asm to .obj file.

Types of assembler

i. one pass assembler

→ This assembler goes through the assembly

language program one and translates the ALP.
→ The assembler has program defining forward references.

ii. Two pass assembler
→ This assembler scans the ALP twice.
→ In the first pass, assembler generates the table of symbols. In the second pass, the assembler translates the ALP into the machine code.

Macroassembler
→ Macro is a group of instructions that appears repeatedly in a program assigned with a specific name.
→ An assembler that can translate the ALP using macros is known as macroassemblers.

Statements
→ An ALP consists of a set of statements
→ Two types

i. Instruction
→ which the assembler translates into machine code
→ Composed of four fields:

* Label field.
* opcode field (mnemonic)
* operand field
* Comment field.

eg:

```
Label1 : MOV AL, BL ; Copy content of BL
  ↓        ↓   ↓      register into AL
Label   opcode operand  comment
```

ii. Directives

→ These are the statements that tell the assembler to perform specific actions. These statements act only during the assembly of program and generates no any machine executable data.

→ Some common directives are:

1. Title

→ It defines the title of the program so that the text typed to the right of this directive is printed at the top of each page in the listing file as comment.

2. Dosseg

→ There is standard order for placing the stack, code, data segments. One can place the

segments in any order as well. This directive tells the assembles to place the segments in standard order.

3. • model

→ This directive determines the size of each segment.

model	Description
1. tiny	→ (code + data) segments can not be greater than 64 KB.
2. small	→ Neither code nor data segment is greater than 64 KB.
3. Medium	→ The code segment may be greater than 64 KB.
4. Compact	→ Only the data segment may be greater than 64 KB.
5. Large	→ Both code and data segment may be greater than 64 KB.
6. Huge	→ All available memory may be used for data and code.

4. • stack

→ This directive sets the size of the program stack, which may be any size upto 64 KB.

5. • Code

→ It defines the part of the program that contains the code (instructions)

6. proc and endp.

→ A program consists of one or more procedures (procedure name ≤ 31 characters)

→ These procedure are defined by the proc directive along with the name of the procedure.

→ The end of any procedure is indicated by the endp directive.

→ Its name must match with the name used by the proc directive.

7. • data

→ variables that are to be used in the program must be defined in the area following the • data directive.

8. Defining data types (db, dw, dd, dq, dt).

These directives define the variables of different types.

Name	DN	Expression

Name → A program references the data item by means of name but it is an optional parameter.

On → Directives which may be:

Directives	Description	Number of bytes
DB	Define byte	1
DW	Define word	2
DD.	Define Double word	4
DQ.	Define quad word	8
DT	Define Ten bytes	10

Expression → It defines the value with which the variable is to be initialized.

Eg.

```
array1  db  5
array2  dw  10
array   db  2,3,5,6
```

; array[0] = 2

9. EQU directive

→ used to define constant.

eg: const .EQU 5000H

so, mov ax, const. ⇒ mov ax, 5000H

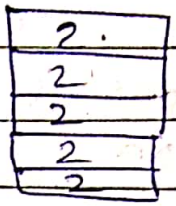
↳ name (not keyword)

10. DUP

→ whenever multiple variables are to be defined and initialized with same value, DUP directive is used.

Eg: var db 5 dup (2)

↳ var (0) = 2
var (1) = 2
var (4) = 2



array db 6 dup (?)

↳ reserves 6 bytes of uninitialized data.

11. END

→ It indicates the end of a program. Any statements after this directives are ignored.

Syntax : END proc_name.

Other directives

12. ASSUME

→ Name of logical segment that the assembler should use for the specified segment.

eg: ASSUME CS : CODE

↳ Instructions are in logical segment Code.

ASSUME DS : DATA

↳ Data are in logical segment DATA.

13. SEGMENT and ENDS

DATA SEGMENT
:
DATA ENDS

} similar to . DATA

CODE SEGMENT
⋮ } Similar to a code
CODE ENDS

14. EVEN

→ tells the assembler to increment the location counter to next even address if it is not already at even address.

15. EXTRN

→ tells the assembler that the names or labels following the directive are in some other assembly module.

Other directives (see DV Hall page 160-161)

* Global
* Group
* Include
* Label
* Name
* Org
* public

Operators in 8086

1. Length.

→ This operator tells the assembles to determine the length of the data item, such as array or string

eg: mov cx, length arr1 ; moves the number of elements in arr1, into cx register.

→ if array was declared as byte, length will produce number of bytes in the ~~string~~ array.

→ if array was declared as word, length will produce number of words in the ~~string~~ array.

2. Offset

→ It indicates the offset or displacement of a variable or procedure from the segment which contains it.

eg. mov bx, offset array1. ; moves the offset address of array1 into bx register.

3. ptr

→ used to assign a specific type to a variable or to a label.

→ It is necessary to do this in any instructions

where the type of the operand is not clear.

Eg:

```
inc [bx]    ; it is unknown whether to
              increment byte in [bx] or
              word in [bx]. Thus:
inc byte ptr [bx] ; increments the byte
                    pointed by [bx]
inc word ptr [bx] ; increments the word
                    pointed by [bx]
```

→ ptr operator assigns the type specified before ptr to the variable specified after ptr.

→ can be used to override the declared type of variable.

Eg:

```
.data
word_arr dw 437AH, 0B97H, 7C41H
:
.code
mov al, word_arr[0] ; X
mov al, byte ptr word_arr[0] ; ✓
```

*

4. short

→ It indicates that only a 1-byte displacement is needed to code a jump instruction.

→ If the jump destination is after the jump instruction in the program, the assembler will automatically reserve 2 bytes for the displacement.

Eg: jmp short label ; sets 1 byte of memory for this jump.

5. Type

→ Determines the type of specified variable.

→ Number of bytes in the type of variable is determined.

→ for byte type variable, assembler will give value of 1, for word → 2, double word → 4

→ Eg: add bx, type w_array ;
increments bx by 2 to point to next word.

some commonly used instructions in 8086:

1. MOV

→ syntax: MOV reg|mem, reg|mem|immediate

eg:
```
MOV AX, 1234
MOV BL, CL
MOV val1, AL ; val1 is defined in DS.
mov [di], 16H
mov bl, [437AH]  bl ← (ds x 10 + 437A)
mov ax, bx
mov dl, [bx]
mov ds, bx
```

2. XCHG

→ Syntax : XCHG reg|memory , reg|memory

eg: XCHG ax, dx

XCHG CL, BH

XCHG AL, Array1 [BX]

3. LEA (Load Effective Address)

→ effective address of variable is loaded into specified 16 bit registers.

→ ~~Syntax~~ Syntax : Lea reg, mem|variable

eg: Lea bx, data1.

→ Similar to mov bx, offset data1.

3. Loop

→ Syntax : Loop address

→ automatically decrements cx by 1 and goes to label unless cx is 0.

eg: Loop start

4. INC and DEC (Increment and Decrement)

→ Syntax : INC | DEC reg|mem

→ Af, of, Pf, Sf and Zf are affected

→ Cf is not affected

Eg: inc bl

inc ax

inc word ptr [si]

5. ADD and SUB / ADC and SBB

→ Syntax: ADD/SUB reg/mem, reg/mem/immediate

→ Syntax: ADC/SBB reg/mem, reg/mem/immediate

→ source and destination must be same size.

→ flags affected: AP, cf, of, Pf, Sf, zf

Eg:
```
ADD AL, 74H
ADC CL, AL
SUB AX, CX
SBB BX, [3427H]
ADD array[BX], AL
```

6. MUL / IMUL

→ IMUL: multiply signed numbers

→ MUL: multiply unsigned numbers.

→ flags affected: of, cf (other flags undefined)

→ 8 bit multiplication:

AL * reg/mem ⇒ AX

→ 16 bit multiplication:

AX * reg/mem ⇒ DX:AX

→ syntax: MUL/IMUL reg/mem

→ Eg: IMUL BL

→ let AL = 01000101 = +69 decimal

BL = 00001110 = +14 decimal

result = AX = 03C6H = +966 decimal

Eg2: IMUL BL ; -28 X 59 = Result (AX) = -1652

11100100 = E4 ⇓ 00111011 = 3B ⇓ = F98CH = 348CH

Eg3: MUL CX ; AX * CX
; result DX : Higher order
AX : Lower order

7. DIV | IDIV (DIV : unsigned, IDIV : signed)

→

Dividend	Divisor	Quotient	Remainder
AX	8 bit reg/mem	AL	AH
DX : AX	16 bit reg/mem	AX	DX

→ Eg:
Div bl ⇒ ax | bl ⇒ Q = AL, R = AH
Div CX ⇒ DX : AX | CX ⇒ Q = AX, R = DX

Eg2:
AX = 37D2H, BH = 37H
div bh ⇒ al = 5EH, ah = 65H

AX = 03ABH, BL = D3H
IDIV BL ⇒ AX | BL
⇒ BL = D3H = -2DH = -(45)10
∴ AL = E9H = -17H = (-23)10
AH = (+3)10

8. Jump
* unconditional jump (JMP)
→ JMP short | near | far address
→ near : next instruction from same cs
→ far : next instruction from another segment.
Eg: jmp continue

* conditional jump
→ jnn short address
Eg: jc 11.

i. for unsigned data (above / below)

		flags.
je / jz	jump if equal or zero	zf
jne / jnz	jump if not equal or not zero	zf
ja / jnbe	jump if above / not below or equal	zf, cf
jae / jnb	jump if above or equal / not below	cf
jb / jnae	jump if below / not above or equal	cf
jbe / jna	jump if below or equal / not above	zf, cf

→ jnc = jae / jnb, all three can be used.
→ jc = jb / jnae ; all three can be used.

ii. for signed data (g = greater, n = not, l = less)
jg / jnle ⇒ jump on greater / not less or equal (zf = 0, cf = of)
jge / jnl ⇒ (sf = of), jl / jnge ⇒ (sf ≠ of)
jle / jng ⇒ (zf = 1 or sf ≠ of)

iii. Other jumps.
* jcxz (jump if cx = 0)
* jno (jump if of = 0), jo (jump if of = 1)
* jnp / jpo (jump if pf = 0)
* jp / jpe (jump if pf = 1)

* jns (jump if sf = 0), js (jump if sf = 1)

9. CALL

Syntax : CALL procedure_name

→ near call and far call also allowed.

RET ; Return from subroutine

10. CMP

→ Syntax : CMP reg/mem, reg/mem/immediate

→ flags updated : af, of, sf, zf, pf, cf

Some logical operations :

Syntax : operation reg/mem, reg/mem/immediate

→ operation : AND, OR, XOR, TEST.

→ operation : NOT

eg: NOT BX

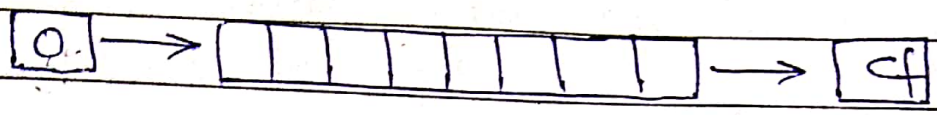
→ ~~operation~~ ;

↳ AND two operands but no result is stored. (Pf, sf, zf affected)

Shifting operation

1. SHR / SAR (Logical shift (for unsigned no.) right / Arithmetic shift (for signed no.) right)
2. SHL / SAL (Logical shift left / arithmetic shift left)

SHR. (Logical Shift right (unsigned numbers))

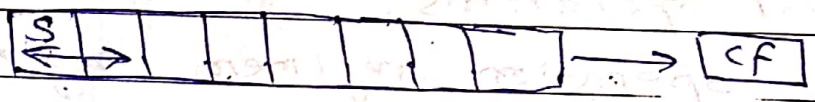


Syntax : SHR reg/mem , CL/ immediate

Eg: Let DL = 11001001

```
SHR DL, 01   ; 01100100 , cf=1
mov CL, 02
SHR DL, CL   ; 00011001 , cf=0
```

SAR: (Arithmetic shift right (signed numbers))



Syntax : SAR reg/mem , CL/immediate

Eg: Let BL = 10101001

```
SAR BL, 01   ; 11010100 , cf = 1
MOV CL, 02
SAR BL, CL   ; 11110101 , cf = 0
```

Rotating operation

* ROR (Rotate right without carry) ≡ RRC (8085)
* RCR (Rotate right with carry) ≡ RAR
* ROL (Rotate left without carry) ≡ RLC
* RCL (Rotate left with carry) ≡ RAL

Syntax: Rotate Reg/mem , CL/immediate

other instructions

1. CLC : clear carry (no operand) ⇒ Cf = 0
2. CLD : clear direction (no operand) ⇒ Df = 0
3. CLI : clear interrupt (no operand) ⇒ If = 0
4. STC : set the carry (no operand) ⇒ Cf = 1
5. STD : set the Direction (no operand) ⇒ Df = 1
6. STI : set the Interrupt (no operand) ⇒ If = 1
7. CMC : Complement the carry (no operand) ⇒ Cf = $\overline{Cf}$
8. LAHf : load AH from flag low byte (no operand)
9. SAHF : Store ~~low order flag~~ AH into lower order flag byte (no operand)
10. PUSHF : push flag register into stack (no operand)
11. POPf : pop flag register from stack (no operand)

Study yourself :

1. AAA
2. AAD
3. AAM
4. AAS

} ASCII Adjust after add, sub, mul, div

5. CBW (Signed byte into signed word)
6. CMPS | CMPSB | CMPSW (compare string, byte, word)
7. CWD (Convert signed word in AX to double word in DX : AX)
8. DAA
9. DAS

} Decimal Adjust AL after add, sub

10. ESC (Escape)
11. HLT (Halt)
12. IN / OUT
13. INT (Interrupt)
14. INTO (Interrupt on overflow)
15. IRET (Interrupt Return)
16. LES / LDS
17. LOCK
18. LODS | LODSB | LODSW

19. LOOPE / LOOPZ (loop while CX ≠ 0 and ZF = 1)
20. LOOPNE / LOOPNZ (loop while CX ≠ 0 and ZF = 0)
21. MOVS / MOVSB / MOVSW (move string, byte, word)
22. NEG (Negate) → 2's complement.
23. NOP
24. PUSH / POP
25. REP / REPE / REPZ / REPNE / REPNZ
26. SCAS / SCASB / SCASW (scan string byte, word)
27. STOS / STOSB / STOSW (store string byte, word)
28. WAIT (wait until test / interrupt signal)
29. XLAT / XLATB (Translate a byte in AL)

Dos functions and Interrupts

→ Dos functions and interrupts are used for I/O services.
→ Interrupt occurs when currently running program is interrupted
→ interrupt may be hardware or software interrupts.
→ Commonly used software interrupts are int 10h for video services and int 21h for dos services.

INT 10H.

→ Dos provides interrupt 10H for video display control.
→ This interrupt is used to control the screen

format, color, text style, scroll etc.

→ The corresponding function number is loaded in AH register with other parameters.

Eg:

function no.	Description	fun? no.	Description
00H	set video mode	01H	Set cursor line
02H	" cursor position	03H	Get cursor position
04H	Read light pen	05H	set display page
06H	scroll window up	07H	scroll window down
08H	Read character and attribute at cursor position	09H	write character and attribute
0AH	write character only	0BH	set color palette (Background / Boarder)
0CH	write dot	0DH	read dot
0EH	write character	0FH	Get-Video mode

INT 21H.

→ 87 different functions are loaded in AH register to perform different I/O operations.

Eg:

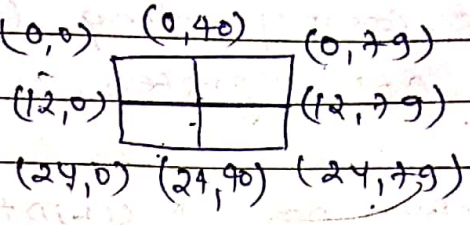
fun? no.	Description	fun? no.	Description
00H	Terminate program	01H	console I/P with echo
02H	Character o/p	03H	Auxillary I/P
04H	Auxillary o/p	05H	printer o/p
06H	Direct console I/O	07H	console I/P
08H	Console I/P without echo	09H	string o/p

Display screen and keyboard processing.

* Typical monitor

Row = 25.

Column = 80



* Tasks to do : cursor locate, clear screen, char. display

* screen location

	(Decimal) row	(Decimal) column	(Hex) row	(Hex) column
upper left corner	00	00	00	00
upper right corner	00	79	00	4f
centre of screen	12	40	0C	28
lower left corner	24	00	18	00
lower right corner	24	79	18	4f

* Cursor setting

→ interrupt 10H (INT 10H)

→ function 02H in AH register

→ required page number in BH register

→ Row in DH register

→ column in DL register

Set cursor at 08 row and 15 column

⇒

```
mov ah, 02h
mov bh, 00h ; 1st page
mov dh, 8   ; row
mov dl, 15  ; column
int 10h
```

Mint2

* clear screen
→ Interrupt 10H (INT 10H)
→ function 06H in AH register.
→ number of lines to scroll up in AL register. (00 for entire screen)
→ Attribute value (eg: color, reverse screen, blinking) in BH register.
→ starting row : column in CX register
→ Ending row : column in DX register

clear screen, set background = white and foreground = blue.

```
⇒ mov ah, 06h
  mov al, 00h
  mov bh, 71h ; 71H
  mov cx, 0000h
  mov dx, 184fh
  int 10h
```

white bg
↓
71H
↳ blue fg.

R	G	B	
0	0	0	→ Black
1	1	1	→ white

* screen display of string
→ Interrupt 21H (INT 21H)
→ function 09H in AH register
→ Requires definition of a display string in data area immediately followed by '$' sign.
→ DX : load display string offset address.

String

Display the "Test".

=>

- data

```
string db 'Test', '$'
```

- Code

```
:
mov ah, 09h
lea dx, string ; mov dx, offset string..
int 21h
```

* keyboard input of string.

→ function 0AH in AH register

→ offset of defined parameters in DX register

→ INT 21H.

* parameters for string input

- data

```
paralst    label    byte
maxlen     db       20
actlen     db       ?
kbdata     db       20 dup (' ')
```

:

- code

```
mov ah, 0ah
lea dx, paralst ; or maxlen
int 21h
```

* Hold screen
→ mov ah, 07h
→ int 21h

* character input from keyboard
→ mov ah, 01h ; with echo ie. entered char. is seen in screen
→ int 21h
→ Entered character is stored in AL register.

* screen display for single characters.
→ function : 02H in AH register → mov ah, 02h
→ Interrupt : 21H → int 21h
→ character to display in dl register

Eg:
```
mov dl, 'a'
mov ah, 02h
int 21h
```

Sample 8086 program.

```
title transfer
dosseg
.model small
.stack 100h
.code
main proc
mov ax, bx
:
main endp
.data
var1 db 20
:
end main
```

Reserved words.

* instructions — eg: MOV, ADD
* directives — eg: END, ENDP
* operators — eg: length, bytaptr
* predefined symbols — eg: @data, '$'

Identifies.

→ identifies = symbol.

* name. eg: str db 40 dup('$')
  ↓ name.
* label. eg: L3: mov ax, bx
  ↓ label.

• Exe and • com files.

→ • com file is smaller than • exe file.

→ In • exe file, the linker generates particular format with special header of 512 bytes or long.

→ program size: • com ⇒ code + data ≤ 64 KB including program segment prefix

→ segment: • com ⇒ single segment.

→ initialization: • com ⇒ auto initialized by org 100h.