MICROPROCESSOR & INTERFACING NOTES

AKSHANSH CHAUDHARY

Microprocessor and Interfacing Notes, First Edition

Presented by: Akshansh Chaudhary
Graduate of BITS Pilani, Dubai Campus
Batch of 2011

Course content by: Ms. Susanna S. Henry
Then Faculty, BITS Pilani, Dubai Campus

Layout design by: AC Creations © 2013

The course content was prepared during Spring, 2013.
More content available at: www.Akshansh.weebly.com

Microprocessor & Interfacing

The Intel Microprocessor: Architecture, Programming & Interfacing Pearson Edu.
8th edition, 2009
– By Barry Brey.

T1	11/3/13	20%.
Q1	25/3/13	5%.
T2	29/4/13	20%. (OB) → ~~open book~~
Compre ~~Q2~~	4/6/13	40%.

Lab: 15%
- 5% before mid sem.
- 5% lab compre
- 5% performance & behaviour

Chapter - 1

Introduction to µp & Computer

1. ~~EMAG~~ ENIAC : Electronic Numerical Integrator And Calculator
2. FORTRAN : FORmula TRANslation
3. ALGOL : ALGOrithmic Language
4. COBOL : COmmon Business Oriented Language
5. RPG : Report Program Generator
6. KIPS : Kilo Instructions Per Second
7. BCD : Binary Coded Decimal
8. K = 1024 : Kilo Bytes
9. TTL : Transistor Transistor Logic
10. DOS : Disk Operating System
11. MIPS : Millions of Instructions Per Second
12. CISC : Complex Instructions Set Computer

13. RISC : Reduced Instructions Set Computer

14. PC : Personal Computer

15. GUI : Graphical User Interface

16. VGA : Variable Graphics Array

17. CAD : Computer Aided Drafting/Design

18. WYSIWYG : What You See Is What You Get

19. AMD : Advanced Micro Devices

20. AGP : Advanced Graphics Port

21. PCI : Peripheral Component Interconnect

22. ISA : Industry Standard Architecture

23. EISA : Extended Industry Standard Architecture

24. DRAM : Dynamic Random Access Memory

25. SRAM : Static Random Access Memory

26. ROM : Read Only Memory (flash memory)

27. PROM : Programmable Read Only Memory

28. EEPROM : Electrically Erasable Programmable Read Only Memory

29. SDRAM : Synchronous Dynamic Random Access Memory

30. CD : Compact Disk

31. DVD : Digital ~~Dis~~ Versatile Disk

32. USB : Universal Serial Bus.

33. TPA : Transient Program Area.

34 BIOS : Basic Input Output System

35 CD ROM : Compact Disk Read Only Memory

36. RAM : Random Access Memory (Read/Write memory)

37. EMS : Expanded Memory System

38 CPU : Central Processing Unit

39 I/O : Input / Output

40 MRDC : Memory Read Control.

41 MWTC : Memory Write Control

42. IORC : Input Output Read Control

13. IOWC : Input Output Write Control

14. ASCII : American Standard Code for Information Interchange

15. MMX : MultiMedia E MultiMedia eXtension

16. XMS : eXtended Memory System

~~17. AGP~~ :

Notes

* Bit : Binary Digit (value 0/1)
* 8 Bits = 1 Byte
* 4 bits = 1 Nibble.

Data Width

(i) byte ——— 8 bits
(ii) word ——— 16 bits
(iii) double word (2) — 32 bits
(iv) quad word (4) — 64 bits
(v) octal word (8) — 128 bits.

Binary addition :- $1+1 = (1)0$ (carry)

Puffin

Information

$2^{10} = 1024$

1. KB = Kilo Byte = 2^{10} bytes
2. MB = Mega Byte = 2^{10} KB
3. GB = Giga Byte = 2^{10} MB
4. TB = Tera Byte = 2^{10} GB

Complement (binary)

1's Complement

Q) 0101 1010

1's 1010 0101

2's (1's complement + 1)

Q) 0101 1010

1's 1010 0101

+1

2's 1010 0110

Hexadecimal to BCH → Binary Coded Hexadecimal

1) 23 : 0010 0011
2) AD4 : 1010 1101 0100
3) 34.AD : 0011 0100 . 1010 1101

BCH to Hexadecimal

1) 1100 0010 : C2
2) 1000 1011 1010 : 8BA

* Decimal to Binary

2	107	
2	53	1
2	26	1
2	13	0
2	6	1
2	3	0
	1	1

$(1101011)_2$

* Decimal to Octal

2	1238	
2	619	0
2	309	1
2	154	1
2	77	0
2	38	1
2	19	0
2	9	1
2	4	1
2	2	0
	1	0

$(1238)_{10} = (2326)_8$

8	1238	
8	154	6
8	19	2
	2	3

$(2326)_8$

* Decimal to Hexadecimal.

16	92
	5 C

92.

$= (5C)_{16}$

✱ Decimal to Binary

		Binary
0.125×2	$= 0.250$	0
0.250×2	$= 0.5$	0
0.5×2	$= 1$	1

$= (.001)_2$

✱ Decimal to Octal

$0.125 \times 8 = 1. \quad = (0.1)_8$

✱ Decimal to Hexa

$0.046875 \times 16 = 0.75 \quad 0$

$0.75 \times 16 = 12.00 \quad C$

$0.2 \times 16 = .0 \quad 0$

$= (0.0C)_{16}$

14 12 8
1) 46875
×16
.750000

875
×16
1200

✱ Decimal to Binary

$(7.0625)_{10} = (?)_2$

2 | 7	$.0625 \times 2 = .1250$	0
2 | 3 | 1	$0.125 \times 2 = 0.250$	0
1 | 1	$0.25 \times 2 = 0.5$	0
	$0.5 \times 2 = 1$	1

$= (111.0001)_2$

✱ Binary to decimal

$(110.101)_2$

$= 6\ 110 = 6$

$.101 = 1\times2^{-1} + 0\times2^{-2} + 1\times2^{-3}$

$= \frac{1}{2} + 0 + \frac{1}{8}$

$\frac{5}{8} = 0.62$

$= (6.625)_{10}$

✱ Octal to decimal

$(125.7)_8$

$(125)_8 = 5 + 16 + 64 = 85$

$0.7 \times 8^{-1} = \frac{7}{8}\ 0.875$

$= (85.875)_{10}$

✱ $(25.2)_6$ to decimal

$(25)_6 = 5\times6^0 + 2\times6 = 5 + 12 = 17.$

$.2\times6^{-1} = \frac{1}{3} = 0.33$

$= (17.33)_{10}$

* $(11011.0111)_2$ to decimal

$(11011)_2 = 27$

$.0111 = \frac{1}{4} + \frac{1}{8} + \frac{1}{16} = \frac{7}{16} = 0.4375$

$= (27.4375)_{10}$

* (6A.C) to decimal

$(6A)_{16} = A \times 16^0 + 6 \times 16^1 = 96 + 10 = 106$

$.C = 12 \times \frac{1}{16} = \frac{3}{4} = 0.75$

$= (106.75)_{10}$ dec

BCD

Packed BCD → stored as 2 digits per byte

Unpacked BCD → 1 digit per byte

2 digit in 1 byte → 8 bit

Decimal	Packed	Unpacked
12	0001 0010	0000 0001 0000 0010
623	0000 0110 0010 0011	0000 0110 0000 0010 0000 0011
910	0000 1001 0001 0000	0000 1001 0000 0001 0000 0000

1 byte for each digit

* **-8 in BCD**

-8 = 2's complement of +8

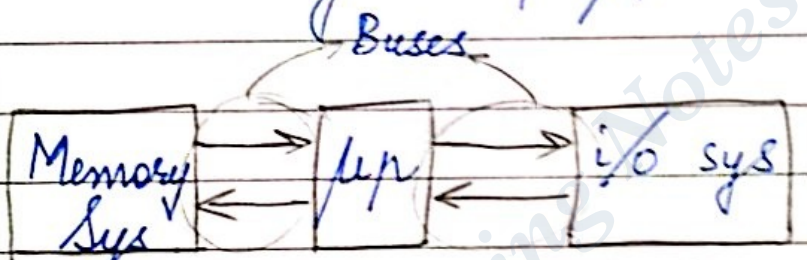
+8 = 0000 1000

1's = 1111 0111

2's = +1

-8 → 1111 1000 ✓

§ **Block Diagram of µp based Comp. Sys.**



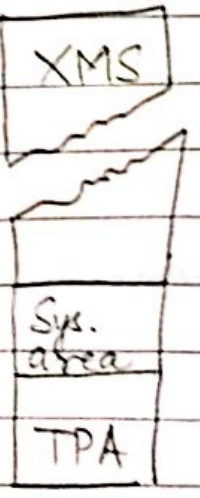
DRAM, SRAM
SDRAM, ROM, EEPROM

8086
8088
80186
80286
80386
80486
pentium

P, P, S, K, M,
CD, DVD, HDD, FDD

Hard disk drive
floppy disk drive
Printer
Plotter
Scanner
Keyboard
Monitor

§ **Memory map of a PC.**



Puffin
Date 10/2/13
Page

Simple arithmetic & logic operations.

Operation	
ADD	NOT
SUB	NEG
MUL	Shift
DIV	Rotate
AND.	
OR	

→ negate

Decisions found in 8086 µp.

Decision	
Zero	Parity
Sign	Overflow
Carry	

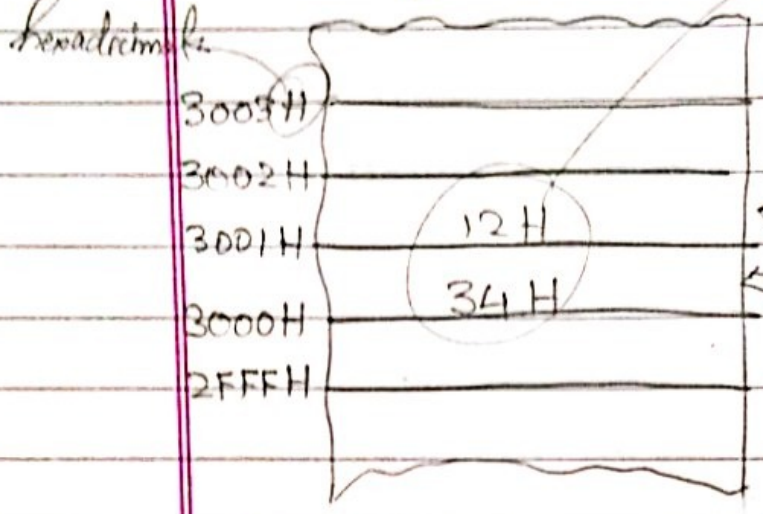
→ odd
→ even

Byte Sized Data

- data stored as unsigned integers : 0 to 255
  signed integers : -128 to +127
- -ve no. represented using 2's complement method

☆ Word sized data

storing 1234 H on memory loc^n.



- 1) data = 1234 H.
- 2) address location: 3000 H & 3001 H.

- i) word = 16 bits.
  - → 2 bytes of data (16 bits = 8+8 bits = 1+1 byte = 2 bytes)
    - MSB → highest
    - LSB → lowest memory loc^n.
- ii) Method of storing no. — little endian format.

eg: | 12 | 34 |

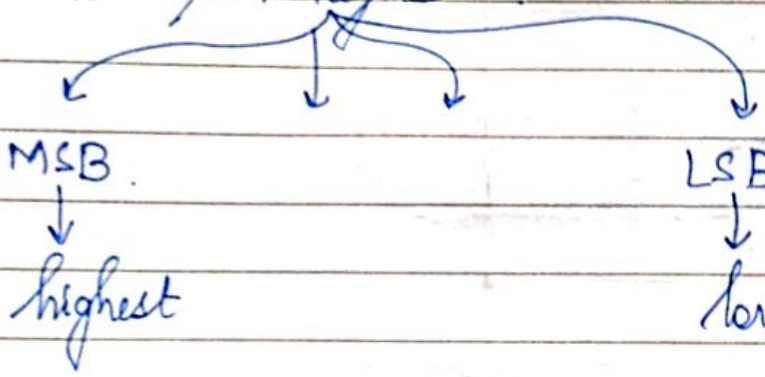
12 → MS byte; 34 → LS byte

Note: If MSB → lowest memory loc^n; LSB → highest memory loc^n.

Then, method of storing: Big Endian format.

✱ Double word sized data

1) 32 bits / 4 bytes.



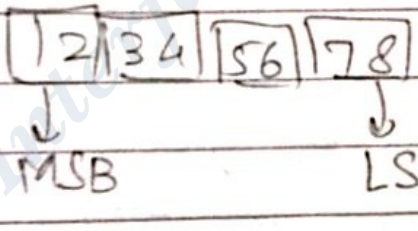
eg: given the data : 12345678 H.

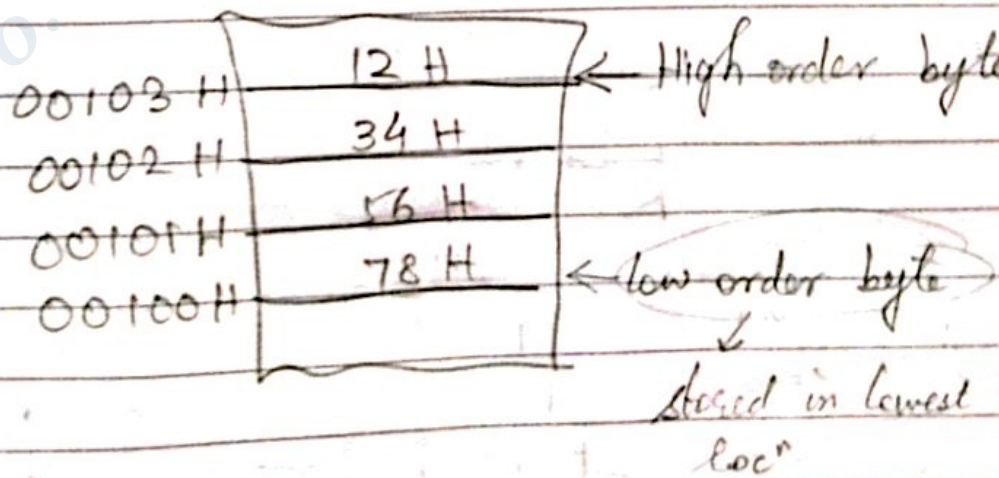
32 bit = 4 bits.

Store in memory loc^n from 00100 H

00103 H.

Ans:





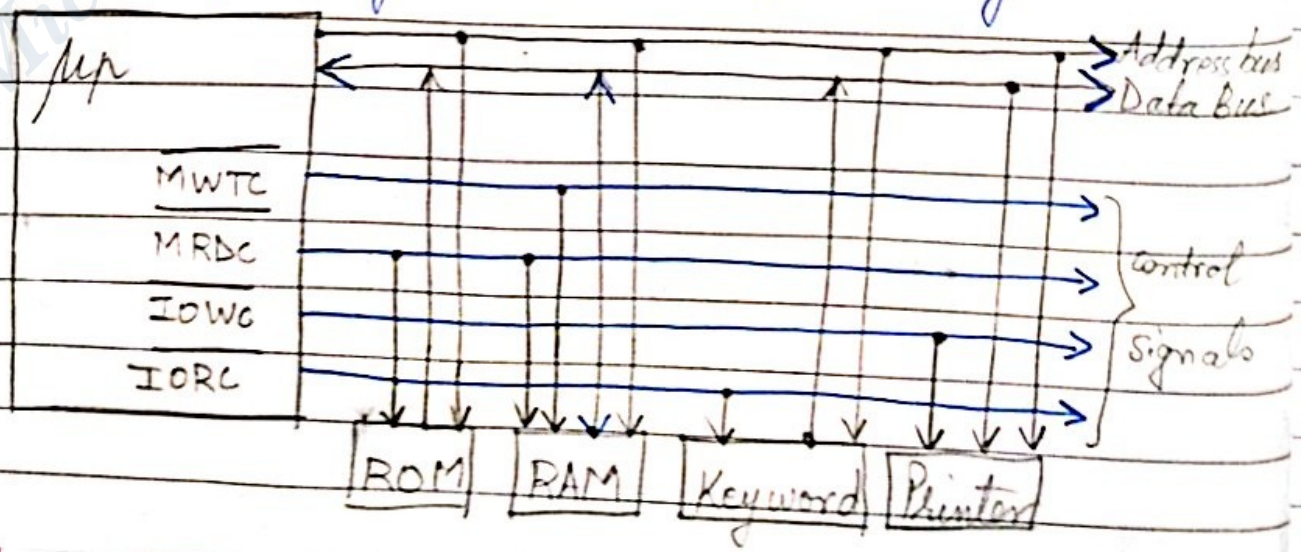
✱ Directives

1) DB : Define Byte
2) DW : Define Word
3) DD : Define Double word
4) DQ : Define Quad word

✱ Microprocessor (≡µp) :—

- µp is referred to as the CPU.
- 3 main tasks of a µp
  - → data transfer b/w
    - → µp & memory
    - → µp & i/o
  - → simple
    - → arithmetic operⁿˢ } ADD, SUB, MUL, DIV,
    - → logical operⁿˢ } AND, OR, NOT .....
  - → program flow via simple decisions

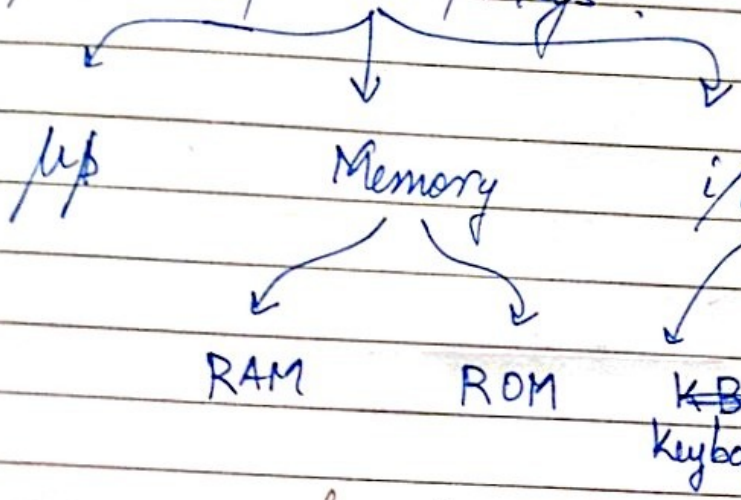
✱ Block diagram of a Computer Sys.



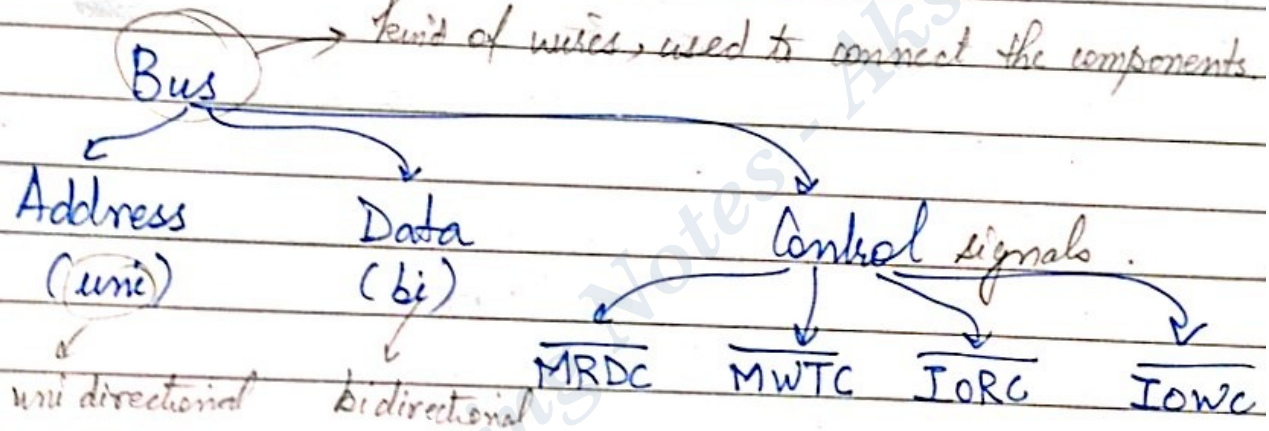
Note: See direction

Puffin
Date 11.2.18
Page

☆ Components of a Comp. Sys.



☆ Bus → kind of wires, used to connect the components.

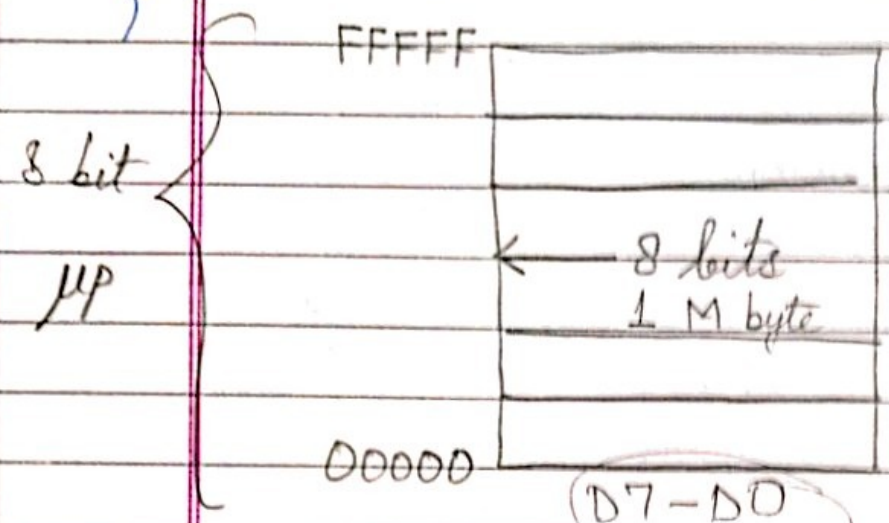


Points ☆
1. 8086 & 8088 address 1 Mega Byte of memory
2. 20 bit address.
3. Selects location 00000H-FFFFFH

§ INTEL FAMILY OF µp BUS & MEMORY SIZES.

µp	DBW Data Bus Width	ABW Address Bus Width	Memory Size
8086	16	20	1MB
8088	8	20	1MB
Pentium	64	32	4GB

PHYSICAL MEMORY SYSTEM



For 16 bit μp :- Make 2 boxes like above.
32 " 4 "
64 " 8 "

✱ Historical background of μp's :-

1. Mechanical age
2. Electrical age
3. μp age
4. Modern μp's

✱ Early 8 bit μp's :

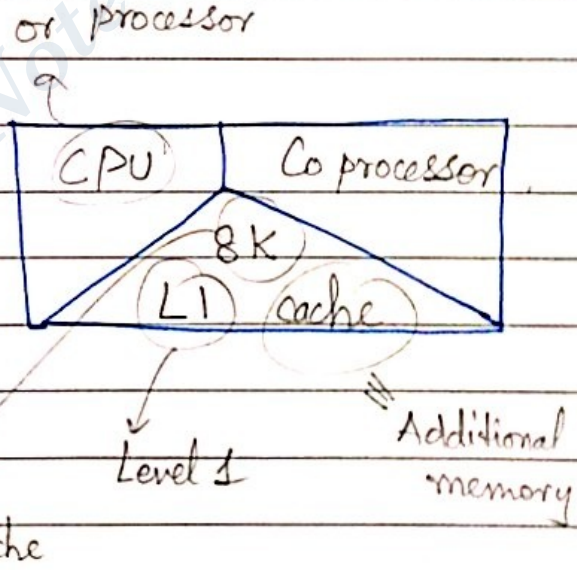
Manufactures	Part No.
Intel	8080
Motorola	MC 6800
Fairchild	F-8
Zilog	Z-8

* Core (P) version

	µp
	80186 80188
P1	8086, 8088, ~~8186~~, ~~8188~~
P2	80286
P3	80386
P4	80486
P5	Pentium
P6	Pentium Pro (PP), Pentium II (PII), Pentium III (PIII), Pentium IV (P IV), Core-2
P7	Itanium

* Conceptual view :-

1. 80486 processor



2. Pentium

2 processors

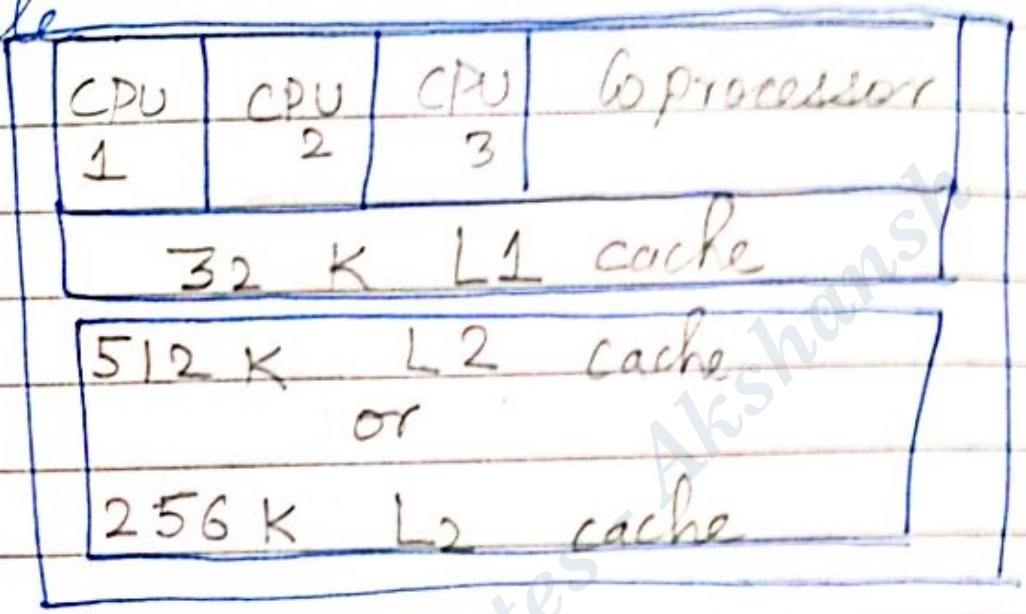
CPU 1	CPU 2	Co processor
16 K L1 cache		

3. Pentium Pro

CPU 1	CPU 2	CPU 3	Co processor
16 K L1 cache			
256 K L2 cache			

2 levels of caches

Pentium II, PIII, PIV,
Core 2 module



Puffin
Date 11.2.18
Page

Chapter-2

THE μp & its ARCHITECTURE

Segment Register	Starting Address	Ending Address
a 4 digit no. used to denote that its hexadecimal (add H)	add a 0 in the end	Add FFFF to starting Add.
2000 H	20000 H	2FFFF H (20000 + FFFF)
4561 H	45610 H + FFFF = 5560F	5560F H
2001 H	20010 H + FFFF = 3000F	3000F H
2100 H	21000 H + FFFF = 30FFF	30FFF H
AB00 H	AB000 H + FFFF = BAFFF	BAFFF H
1234 H	12340 H + FFFF = 2233F	2233F H
0A28 H	0A280 H + FFFF = 1A27F	1A27F H
0A0F H	0A0F0 H + FFFF = 1AFEF	1AFEF H

R ⇒ 64 bit register
E ⇒ 32 " "

Puffin
Date 12.2.13
Page

* PIC : Program Interrupt Controller

Segment Register	Starting Address	Ending Address
090FH	090F0H	19FEFH
	+ FFFF	
	19FEF	
4900H	49000H	58FFFH
	+ FFFF	
	58FFFH	
3400H	34000H	43FFFH
	+ FFFF	
	43FFF	
1000H.	10000H	1FFFFH
	FFFF	
	1FFFF	

REGISTERS → store data of sizes :- Byte, Word, Double word

used for what purpose.

Multi-purpose
- RAX Accumulator (It a 64 bit register)
- RBX Base Index
- RCX Count
- RDX Data
- RBP Base Pointer
- RDI Destination Index
- RSI Source Index
- R8 - R15

Special purpose
- RIP Instr. Pointer
- RSP Stack Pointer
- RFlags

* Half carry :- eg: Add^{n} of 8 bits, say. If after 4 bits, ∃ a carry. So, Auxiliary flag is 1.

Puffin
Date
Page

* **Segment Register** (Another type of register apart from those)

1. (CS) Segment Code : Gives starting address of the section of memory where info. is stored
2. (DS) Data : It holds data & the data is accessed by offset address
3. (ES) Extra : Used by string instructions to hold destin^{n} data
4. (SS) Stack : A part of memory where data is present & is addressed/accessed by Stack pointer (SP) & Base Pointer (BP)
5. FS + GS : Add^{nal} segments is used for internal oper^{n} in Windows

* **List of flag bits** → have value 0/1

1. (C) Carry → gives info: carry generated/not → becomes 1 when result of any oper^{n} (arithmetic) exceeds capacity of machine. Includes both carry & borrow
2. (P) Parity → gives info: no. of 1's is even: 1, odd: 0
3. (A) Auxiliary Carry → gives info. of half carry
4. (Z) Zero → Result zero: 1, other: 0
5. (S) Sign → -ve no: 1, +ve no: 0
6. (T) Trap → when I/p interrupts flow of program: 1
7. (I) Interrupt → disables the communic^{n} b/w PIC & µp
8. (D) Direction → They are incremented/decremented during string instructions (held by DI & SI registers)
9. (O) Overflow → to select privilege/resources available for I/p o/p devices.
10. IOPL I/p o/p Priveledge level
11. NT Nested Task → Indicates if one task is nested within another task
12. RF Resume Flag → controls the resumption of execution after next instruction
13. (VM) Virtual Mode → the selected virtual mode oper^{n} when it is 1
14. (AC) Alignment Check → gives info: if a word/double word is misaligned on a non word boundary
15. (VIF) Virtual Interrupt Flag → its a copy of (IF) Interrupt flag
16. (VIP) Virtual Interrupt Pending → gives info about virtual mode, used in multitasking envt.
17. (ID) Identification → to provide info about VIF to OS
    1: ⇒ processing is using CPU ID → details of manufacturer, etc.

* **REGISTERS** registers used by :-

Program Visible (AP)	PI (Program Invisible)
1. Used - applic^{n} programming	1. Not addressable directly during applic^{n} programming (AP)
2. Specified by instruments. ⇓ we can make changes in the register	2. Used indirectly during System Programming (SP) ⇓ only sys. can make changes, not us.

* Virtual mode oper^{n} :- Using the 1st 1MB of memory for oper^{n}. This is the default mode in a µp

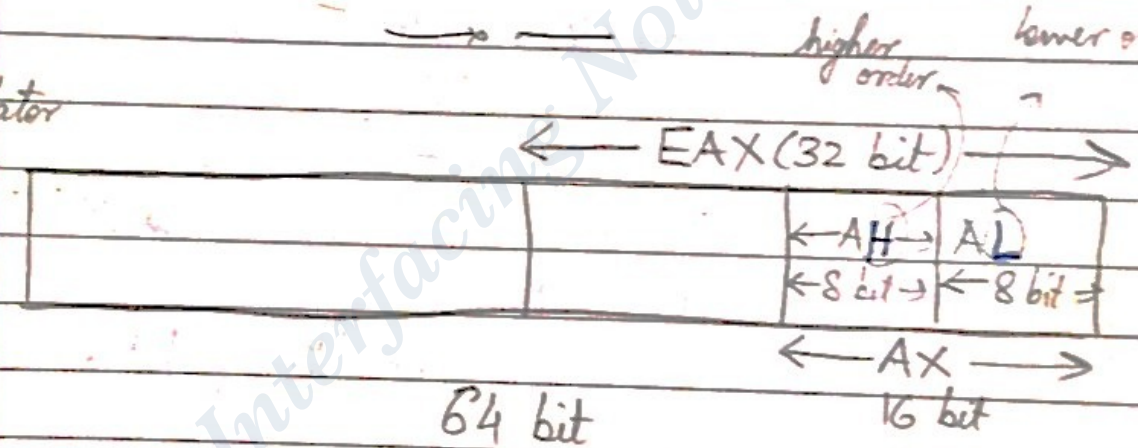
MODES OF OPERATION

Real mode memory
Also called
- DOS memory
- exists at loc^{ns} 00000H - FFFFFH (first 1 MB of memory)
- present in all versions of µp

Address memory to cover 1 Mega Byte.

Protected mode memory
Also called
- Windows memory
- any loc^{n} in PMS (Protected Memory Sys)
- not present in any earlier µp. Only there from 80286 - core 2.

Flat



- Full box → RAX register (64 bit)
- divide box into 2 halves → One half called as EAX register (32 bit)
- divide that further into 2 halves One half called as AX (16 bit).

- - - - -

AX = AH + AL (AH: higher order, AL: lower order)
BX = BH + BL
CX = CH + CL
DX = DH + DL

- Whenever ADD is performed, both registers should be of same size.
- Code ⇒ Instructions to process the data

Puffin
Date
Page

★ 8 bit registers (list)

- ✓AH ✓CH
- ✓AL ✓CL
- ✓BH ✓DH
- ✓BL ✓DL

eg: CODE:- ADD AL, AH
(AL = destination, AH = source; ADD AH to AL; result stored in AL)

After ADD, AH remains same (AL+AH) gets stored as result in AL

eg: AL = 3, AH = 5
ADD AL, AH
Now AL = 8, AH = 5

★ 16 Bit registers (list)

- ✓AX ✓IP
- ✓BX ✓FLAGS
- ✓CX ✓CS
- ✓DX ✓DS
- ✓SP ✓ES
- ✓BP ✓SS
- ✓SI ✓FS
- ✓DI ✓GS

eg: ADD DX, CX
(DX = destination, CX = source; stores result)

★ FLAGS :- (write before)

C: Carry flag S: Sign bit D: Direction flag
A: Auxiliary flag T: Trap flag O: Overflow flag
Z: Zero flag I: Interrupt flag IOPL: I/O privilege level
NT:

* 32 bit registers (list)

EAX , ESP. eg: ADD ECX, EBX
EBX
ECX → extended bit Registers.
EDX
EBP

* Section 2.2

REAL MODE MEMORY ADDRESSING

Virtual mode :- Several 1 MB sections of memory co-exist.

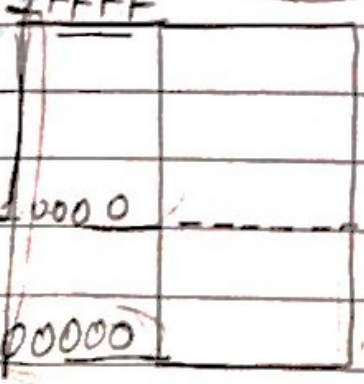
Real mode :- Only 1 MB of memory is used.

* Segments & Offsets.

↳ any memory loc^{n} is got by adding segment address & offset address.

↳ a section of Segment.

eg: Consider a segment of memory



→ Implies basically the range of memory loc^{ns}

→ Every one of those memory loc^{ns} is an offset.

64 KB segment.

* Way to get memory address :-
Append segment address with a zero & add it to offset address. That gives memory address. (done in the beginning of chapter)

* Default segment & offset registers.

CS : IP
Segment Offset.

Eg, memory address can be found as

CS (0) → zero appended
+ IP
Ans.

Why other combin^ns of registers :-

SS : SP or BP
DS : BX, DI, SI
ES : DI

} other combin^n of registers

Puffin
Date 8.2.18
Page

To Be Done in MASSEM

Chapter -3

ADDRESSING MODES

- Data Addressing mode
  - Commands :-
    - Move
      - Data is transfered b/w
        - Register & Register
        - Register & memory
        - ~~Segment & Segment~~ → not possible
    - Register
    - Immediate
    - Direct
    - Register Indirect
    - Base + Index
    - Register Relative
    - Base Relative + Index

- Program Memory Addressing mode
  - Commands :-
    - Jump & call.
    - Program Relative
    - Direct
    - Indirect.

Section 3.1 DATA ADDRESSING MODES.

- MOV INSTRUCTION

MOV (AX, BX)
AX → destin" register
BX → source

Note: Nothing happens to data in source register

* No flags are affected by MOV instruction

* 64 bit registers

RAX RBP
RBX RDI **
RCX
RDX

* Note :- The command

MOV CS, AX

→ code segment

(Selects only the codes)

So, we cannot move the contents from any register to CS

Q. Copy contents of CS & DS
directly not possible : both segment registers
So, done in 2 steps.

MOV AX, CS
MOV DS, AX ✓

Q. Copy the contents of BX into CX, no. of bits = 32

mov ECX, EBX

Q. Copy contents of BP into SP

mov SP, BP

Q. Copy contents of CL to byte portion of RB

mov RBL, CL

→ byte portion

Q Copy word portion of R10 into BP (base pointer)

mov BP, R10W.

Q Copy content of accumulator into code segment

mov ~~CS~~, AX not valid

(nothing into CS)

Q CS to DS → not possible in 1 step

mov AX, CS

mov DS, AX

---

§ ☆ Immediate addressing : data is present directly in the instruction (instead of register)

eg : mov BL, 44

* The data can be byte, word ✝ μp, double word for 8086 - Pentium

eg ② :- mov AX, #3456 H

→ This format is rare, but possible

* Moving a hexadecimal no. → begin with 0 end with H.

☆ Moving a character :- eg : mov BH, 'A'

Q Copy $(100)_2$ into AX register.

mov AX, 100 B.

Paffin

Micro Assembler

* Program writing in ~~MS~~ ~~MASM~~ MASSEM

- .MODEL TINY → Directs assembler to assemble program into single CS.
- .CODE → Indicates start of CS
- .START UP → it'll start with 1st instruction

- .MODEL SMALL
- .DATA
- .START UP.

- - - - - - ← → Start writing Assembly lang. Instructions

- .EXIT → exit to DOS
- END ; → end program file
  → # check if sup # is pal or not

* Labels : Symbolic name for memory locns.
* OPCODE , OPERANDS
  eg : mov BX, CX

* Comment line : (Begins with ;)
  eg : mov AX, 51 ; Comment.

* Direct Data Addressing

Direct Addressing
- Transfer b/w memory locn to AH, AL, AX or EAX
- 4's 3 bytes sized

Displacement addressing
- All instructions left come under this
- 4 bytes

Direct Addressing

eg: Suppose a memory loc^n named NUMBER exists.
Then, data can be copied as:-

MOV AL, NUMBER

If we know address of memory loc^n then, the data present in memory loc^n can be copied as:-

MOV AL, [1234H]

Q. Move contents of a memory loc^n of name DATA to CH

i.e. direct not possible.
Do displacement addressing

eg: MOV CH, DATA
eg: MOV ES, DATA 6

Register Indirect Addressing

eg: mov AX, [BX]
Suppose BX has data 1000 & DS has 100.
So, append 0 to DS & add to BX

BX 1000
DS 100 + 0
2000

This is effective address.
The data from this address gets stored in AX

Note: Some contents must already have been in the address [2000] that gets stored in AX.

✓ MOV AX, 'NUMBER'
✗ MOV AX, 123BH

Puffin
Date
Page

2^{15} + + + — — — 2^6 2^5 2^4 2^3 2^2 2^1 2^0

- Register indirect addressing doesn't happen b/w memory to memory.

eg Some instructions :

mov [BP], DL ✓
mov [DL], BH ✓
mov [DI], [BX] ✗

* [DI], [SI] or [BX] : If they ∃ in instruction, then final address is got by appending '0' to DS (Data Segment)
* [BP] : If it exists in instruction, then, final address is got by appending '0' to SS (Stack Seg)
* If we don't know the size of the data that is being moved
* Suppose we want to use only a specific portion of the destin^n address (say, 8 bits out of total 64 bits), then we use pointer.
  eg : mov byte ptr [DI], 10H
* Pointers → Byte Ptr
  → Word Ptr
  → Quad ptr.

ARRAY

Array name
Data word

Syntax: DATAS DW 50 DUP(?)

```
.CODE
.STARTUP
mov AX,D
mov ES,AX
mov BX, OFFSET DATAS
mov CX,50
Again:
    mov AX, ES:[046CH]
    mov [BX], AX
    INC BX        → one increment for count
    INC [BX]      → one increment for memory loc".
    LOOP AGAIN
.EXIT
END
```

for looping

- BX stores starting address of an array (its starting element.)
- DI stores loc" of every elemt. of array.

BASE + PLUS INDEX addressing

→ uses 1 Base & 1 Index register

eg: mov DX, [BX+DI]

→ value of DS appended to zero? Then, BX & DI values added } added together

eg: BX : 1000
DI : 0010 → + 1010
DS = 100+0 = 1000
2010 → effective address

Contents of 2010 are put in DX

```
.MODEL SMALL
.DATA
ARRAY   DB   16 dup(?)
        DB   29H        ← this is the value we want to refer to in the array
        DB   20 dup(?)  ← extending array
.CODE
.STARTUP
mov  BX, OFFSET ARRAY
mov  DI, 10H            ← 16th no.
mov  AL, [BX + DI]
mov  DI, 20H
mov  [BX + DI], AL
.EXIT
END
```

Register Relative Addressing

- Contents of base & index are added to get effective address.

OR

- Add contents of displacement address to index or base register.

eg: `mov AX, [BX + 1000H]`

(BX: BASE, 1000H: DISPLACEMENT)

or

`mov AX, [DI + 100H]`

(DI: INDEX, 100H: DISPLACEMENT)

Let DS : 0200 (append '0') → 02000

BX = 0100 → 0100

Displ = 1000H → +1000H

= 3100H → final address

mov AX, [SRDI]

☆ Base Relative-Plus Index Addressing

↳ Add Base + Index + Displ.

eg mov AX, [BX + SI + 100H]

let BX = 0020H
SI = 0010H
DS = 1000H

0030H
append 0 → 10000H
+ 100H (displ)
= 10130H

Note :- 10130H will have 8 bit data & AX is 16 Bit. So, value of 10130 & 10131 get stored in AX

☆ Scaled Index Addressing

eg: mov EAX, [EDI + 2* ECX]

↳ X scaling factor with index register or base register.

☆ RIP addressing

↳ only used by processors manufactured by Intel.

☆ Program Memory Addressing Modes

↳ used in JMP & CALL instruction

Direct | Relative | Indirect

☆ Direct program memory addressing

↳ Similar to 'Goto' stmt in C lang

JMP (Jump) Instruction

- Short Jump (1 Byte)
- Near Jump (2 Bytes)
- Far Jump (Anywhere in memory)

- eg: JMP [10000H]
- Consists of CS (instead of DS)

CS $\xrightarrow{\text{append } 0}$ CS×10'

\+ IP

final address → That is calculated & program jumps to that loc"

✱ Section - 3.2
PROGRAM MEMORY ADDRESSING MODES

Instructions

✱ Relative Program Memory Addressing
Instructions are relative to IP (Instruction pointer)

eg: JMP [2] ⇒ 2 Bytes
⇒ IP moves by 2 bytes
⇒ Jump to the next instruction (after 2 bytes)

eg:	1378:1000	EB	
	1378:1001	mov BX,02	JMP [2]
	1378:1002		
	1378:1003		
	1378:1004		→ Jumps to this

* Indirect Program Memory Addressing
↳ it uses any register → Direct & displacement also

eg: JMP AX (any register) jumps to loc" address by register AX

§ Section - 3.3

Stack Memory Addressing Modes
↳ portion of memory that holds data temporarily.
- Its LIFO (Last in 1st out) memory
- It has 2 instructions: PUSH & POP
- Registers used: Stack pointer & Stack segment

eg: PUSH BX → Copies
↳ Pushes contents of BX into stack (address)

Stack segment (append 0) → SS x 10
+ Stack pointer (SP)
effective address

eg: POP CX
↳ data is removed from stack & placed into CX
(Address of stack found in the same way as above)

eg: POP F
↳ data removed & placed in flag register

eg: POP FD (data is word)
↳ Same oper" as in POP F.
But data is double word

Puffin
Date 3/3/13
Page

Ch 3 Problems

Q Suppose, DS = 0200H, BX = 0300H & DI = 400H find memory address accessed by each of the following instruction

(a) mov AL, [1234H]
(b) mov EAX, [BX]
(c) mov [DI], AL

(a) [1234H]

DS = 0200H $\xrightarrow{\text{append } 0}$ 02000 H
+ 1234 H
memory add ← 3234 H Ans.

} Direct Addressing

(b) Register indirect addressing :-

BX = 0300H
DS = 0200 H $\xrightarrow{\text{append } 0}$ 02000H
+0300H
memory addres → [2300H] Ans

(c) Register Indirect addressing

DI + (DS x 10) = 0400H + 02000 H
memory add. = 2400H .

↳ contents of AL are put into this loc".

Q. Diff. b/w

mov BX, DATA & mov BX, OFFSET DATA

DATA: If has some data. BX will hold data in the memory DATA

OFFSET DATA: If has array. BX will hold # address of the 1st element of array

Q. Suppose DS = 1300H, SS = 1400H, BP = 1500H, SI = 0100H
Determine address for following instructions:

① mov EAX, [BP + 200H]
② mov AL, [BP + SI - 200H]
③ mov AL, [SI - 0100H]

① Register Relative Add.

Consider contents of SS when BP is given. In other cases, use DS.

SS x 100 = 14000 H
+ BP = 1500 H
+ 200H + 200 H
15700 H. Ans

② Base Relative Plus Index.

BP given, so, use SS instead of DS

SS x 10 = 14000 H
+ SI = 0100 H
+ BP = 1500 H
+ Displ. = 15600 H
Ans = -0200 H → Displ.
= 15400 H

(3) Register Relative

DS x 10	=	13000 H
+ SI		0100 H
+ Displacement		13100 H
Ans		− 0100 H → Displacement
		→ 13000 H

Puffin
Date 5/3/2013
Pg

Chapter - 4

☆ DATA MOVEMENT INSTRUCTIONS

Section 4.1 : Machine language

- → microprocessor understands only 0/1
- → length of instruction: 1 — 13 bytes
- → 8086 : operates on 16 bits
- → After 3rd version of 8086 till pentium 4 : operates on 32 bits.

- 16 Bit Instruction Mode

Op code	Mode Register Memory	Displacement	for Immediate data
Occupies 1-2 bytes	0-1 bytes	0-1 bytes	0-2 bytes

- 32 bit Instruction mode

Address Size	Register Size			Scaled Index		
Occupies 0-1 bytes	0-1 bytes			0-1 bytes		

extra contents as compared to 16 bit mode.

Eg: considering a 16 bit instruction

mov AX, BX → 16 bit registers

for 32 bit register :-

Address size 67

Register size 66 mov AX, BX → 16 bit registers

written if instruction is 16 bit (address/register) & mode of oper^n is 32 bit.

Eg: mov EAX, EBX

→ 66, 67 won't come, as its a 32 bit instruction.

- op code: gives info. of oper^n.

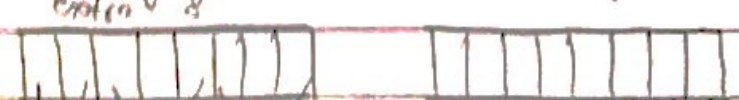
Considering 1 byte of info.

op code

→ tells about data
W = 0 : Byte
W = 1 : Word/double w.

→ dir^r of flow of data
→ when D = 1 : Dir^n of flow of data from memory to register
→ D = 0 : Vice versa

2 bytes ⇒ 16 bits



extra 8

(comes before)

Info of mode of oper^n | Info of register | Info of memory/register

* MOD

- ~~no~~ MOD for 16 bit instruction mode

MOD	f^n
00	no displacement
01	8 bits "
10	# Array 16 bits "
11	R/M is a register

Register ↘ memory

- MOD for 32 bit instruction mode

MOD	f^n
00	no displacement
01	8 bits "
10	# Array 32 bit "
11	R/M is a register

eg : mov AL, [DI] : no displacement
⇒ MOD = 00

mov AL, [DI + 2] : displacement = 2
MOD = 01 (can be represented using 8 bits)

mov AL, [DI + 1000 H]

Displ. = 1000 H

MOD = 10

0001 0000 0000 0000
16 bits represent^n

REG & R/M when MOD = 11

Code	W=0 Byte	W=1 Word	W=1 Double word
000	AL	AX	EAX
001	CL	CX	ECX
010	DL	DX	EDX
011	BL	BX	EBX
100	AH	SP	ESP
101	CH	BP	EBP
110	DH	SI	ESI
111	BH	DI	EDI

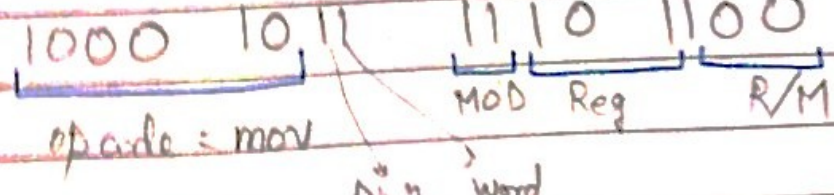
eg: when mode = 11 & code = 100, for 1 byte of data → AH

eg:

→ Register = CH

→ Register size & Address size not given. So, its a 16 bit instruction, not 32 bit

eg: 8BEC H



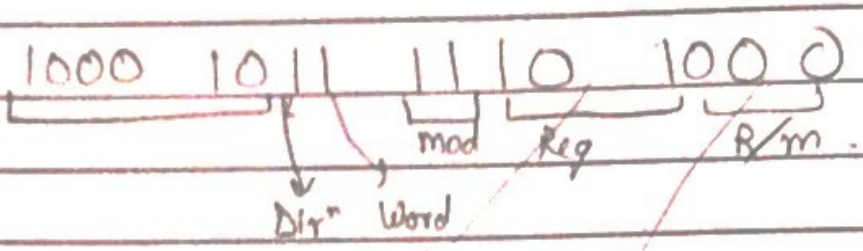
Instruction : mov BP, SP

* Note :- 100010 : Its the code for MOV instruction

Puffin Date 6/3/18 Page

→ 32 bit

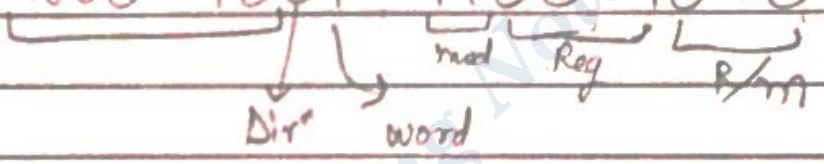
eg : (66) 8BE8 H



Instr" :- mov EBP, EAX

→ 32 bit

eg : (66) 89CA



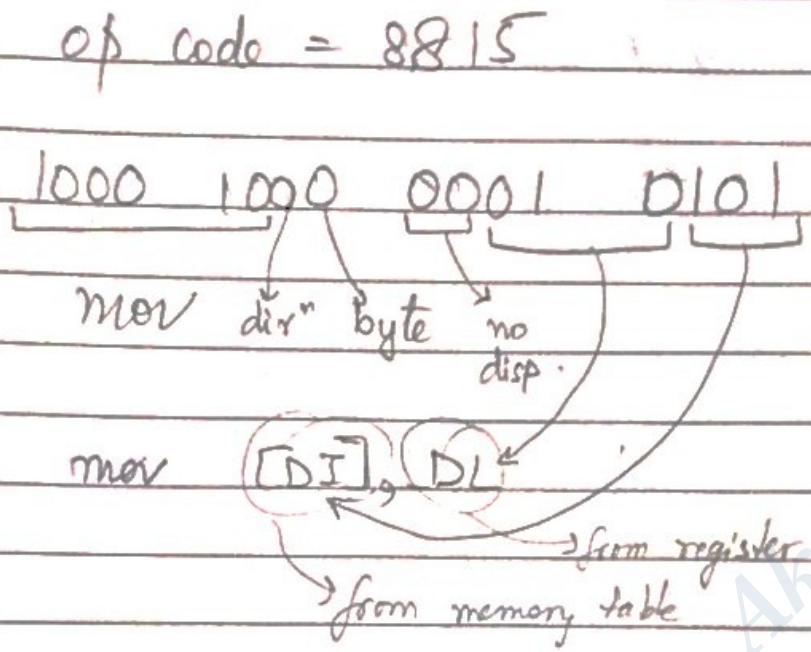
mov EDX, ECX

R/M memory addressing

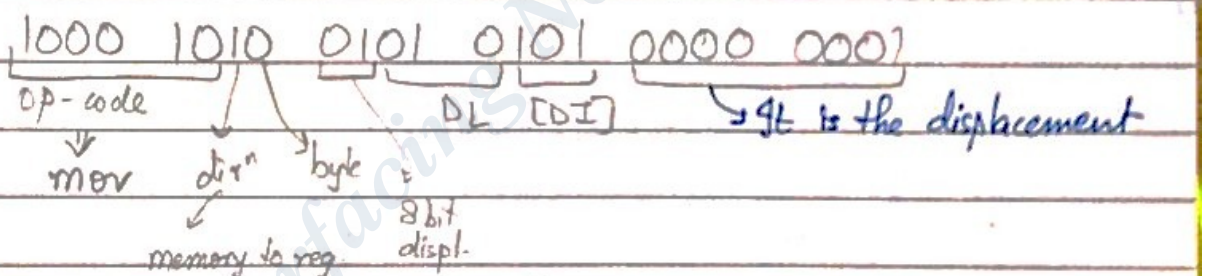
* 16 BIT R/M → use this table only when MOD ≠ 11.
(otherwise, if MOD = 11, only see Register table)

R/M code	
000	[BX+SI]
001	[BX+DI]
010	[BP+SI]
011	[BP+DI]
100	[SI]
101	[DI]
110	[BP]
111	[BX]

eg:- op code = 8815

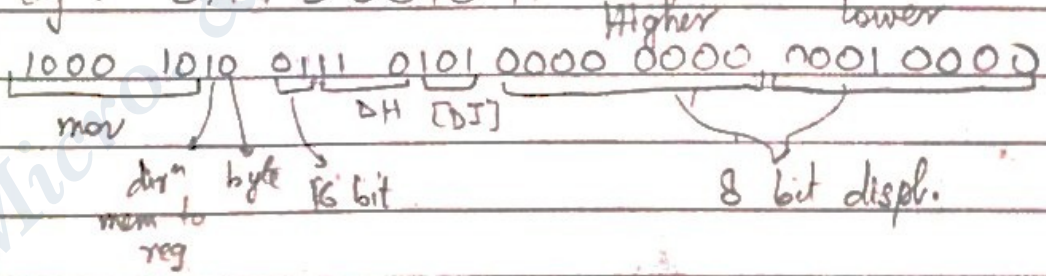


eg: 8A5501

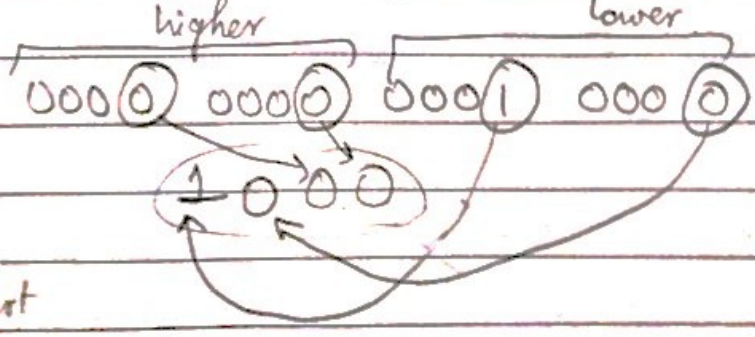


⇒ mov DL, [DI + 1]

eg: 8A750010 H



⇒ mov [DI + 1000], DH



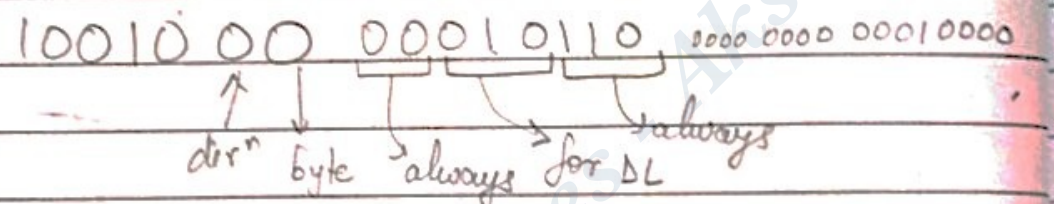
write higher part first

Then, lower part

☆ Special addressing mode

→ ∀ instructions which have displacement
→ MOD = 00 } always
→ R/M = 110 }

eg: mov [1000H], DL



☆ 32 bit Addressing mode (Selected by R/M)

R/M Code	Function
000	DS:[EAX]
001	[ECX]
010	[EDX]
011	[EBX]
100	- uses scaled index type
101	SS:[EBP]
110	[ESI]
111	[EDI]

☆ SS → multiply

00 = X 1
01 = X 2
10 = X 4
11 = X 8

☆ Scaled Index Byte

denote index register ⇒ 32 bit.

eg: 67 66 8B 04 8B H

0110 0111, 0110 0110 1000 1011 | 00 000 100 | 10 001 011

- 0110 0111, 0110 0110 → no need ∵ they denote that its 32 bit instruction
- 1000 1011 → mov; dirⁿ: mem to reg; double word
- 00 000 100 → MOD; EAX R/M
- 10 001 011 → SS; Index register reg; Base reg

mov EAX, [EBX + 4*ECX]

- EBX → 011 : Base
- 4 → SS = 01 ∴ ×4
- ECX → 001 : Index reg

☆ An Immediate Instruction

eg: mov WORD PTR [BX+1000H], 1234H

11 0001 | 1 | 1 | 10 | 000 | 11 | 0000 0000 | 0001 0000

- 11 0001 → op-code for MOV immediate
- 1 → Dirⁿ (always)
- 1 → word (as given above)
- 10 → MOD (always 10)
- 000 → Reg (always kept 000 for immediate)
- 11 → R/M
- 0000 0000 → low Displ.
- 0001 0000 → High Displ.

.... 0011 0100 | 0001 0010

- 0011 0100 (3, 4) → low data
- 0001 0010 (1, 2) → High data.

✱ Segment MOV instruction

1000 11	0	0	11	001	100
op-code for segment mov	Dir" (always 00)	Byte (always 00)	MODE (always 11)	Reg	R/m

From table with mod = 11

⇒ MOV AH, CS

✱ Note: change here comes in choosing Reg.
We have to select it from :-

Code	Segment Reg.
000	ES
001	CS
010	SS
011	DS
100	FS
101	GS
110	X
111	X

✱ MOV CS, R/M
or ~~MOV~~
POP CS

} Instructions not allowed.

Regarding choice of R/M: choose it from table when MOD = 11.

SECTION 4.2

PUSH / POP

↳ related to pushing & popping data from stack

☆ Stack behave as LIFO (Last In First Out)

☆ PUSH/POP

FORMS
- → Register : contents from 64 bit register pushed to stack/popped from stack
- → Memory : contents of memory loc^n pushed/popped
- → Immediate : Immediate data → pushed on stack → POP NOT POSSIBLE
- → Segment : contents of segment register : pushed/popped
  In ES : POP not possible
- → Flag : Push & pop possible from stack
- → All register : Both push & pop both work ☆ registers.

☆ PUSH : Transfer of 2 bytes of data (for 8086)
" 2-4 " (80386-P4)
- → copies data to stack
- → source of data → 16 bit
  → 32 bit

→ ☆ PUSHA :
Copies contents of all registers onto stack (except segment registers).
Copying order :- AX, CX, DX, BX, SP, BP, SI & DI.

→ ☆ PUSHF :
Copies contents of flag register onto stack.

→ ☆ PUSHAD :
Copies contents of 32 bit registers (80386-P4)

for 2 byte data
☆ Basically, MSB moves to loc^n, pointed by (SP − 1)
LSB moves to stack, pointed by (SP − 2)

So, for 2 bytes of data, SP is decremented by 2.
Ily, for 4 bytes of data, SP is decremented by 4.

- when data is present in register.

eg for instruction : PUSHI EAX.

say, Data :- 6AB3
SP - 0800
SS = 0300,

So, [(SS X 10) + SP] → gives loc^n of stack

So, now, 03800 - 1 = 37FF : loc^n of AH (i.e 6A).

03800 - 2 = 37FE : loc^n of AL (i.e B3)

for immediate data :- PUSH : 16 bits
PUSH(D) : 32 bits data.
↳ double.

↳ when range of value: Opcode:
00 - FF 6A H
0100 - FFFF 68 H

eg : for immediate data → 8
Range → 2 bytes (00 - FF)
So, opcode = 6A
hence, we get 6A 08

eg : PUSHI 1000 H
H L
H: of 4 bytes
opcode 68 0010 H
↳ write L first, then H.

eg PUSH 'R'
ASCII value of R gets pushed.

Note: POP CS not possible.
↳ default code of memory cannot be changed.

Puffin Date 19.3.13 Page

POP
↳ Data popped out from stack & put into register, memory, segment, flag.

→ POPF: Removes data from stack & puts in flag.

→ POPFD (Double): 32 bit data removed from stack & put into flag.

→ POPA (all): Data removed from stack & moved to registers:-
Order: DI, SI, BP, SP, BX, DX, CX, AX

eg: POP BX → 2 bytes
Last In, first out of stacks → That value (2 bytes)
1st byte goes to BL & 2nd byte goes to BH.
(The loc of that value is given by SP)
eg: SP = 0000H, SS = 1000 H. So, loc = (SS x 10) + SP
So, value stored in loc 10000 goes to BL
& that stored in loc 10001 goes to BH.
↳ Pointed by SP.

→ POP AD: 32 bit data removed from stack & moved to all registers
↳ order as in POP A.

Format for Stack Initialization in MASM

(M1)

```
STACK_SEG    SEGMENT STACK
             DW 60H DUP(?)
STACK_SEG    ENDS
```
↳ size of stack

☆ Note :- If stack is not initialized & PUSH/POP instruction is used, a default stack, called as PSP (Program Stack Prefix) will be used, which has 128 bytes of memory. ∃ no error in execution

Puffin

(M2) .MODEL SMALL
.STACK 100H → Stack Initializ^n

☆ Note :- We know
Memory address = (SS × 10) + SP.
or, actual add.
So, Illy, given memory loc^n, say, 10000,
SS = 1000
SP = 0000

☆ Section 4.3

↳ LOAD EFFECTIVE ADDRESS (LEA)

* ∃ 2 types of instructions

LEA

✓ registers have offset address

LDS, LES

✓ registers have offset address, but, LDS, LES
LDS → Data Segment
LES → Extra Segment
also have segment address.

eg : LEA AX, NUMB
AX is loaded with offset address of NUMB
Illy. LEA SI, LIST

Puffin

* A memory has diff segments → CS, SS, DS.
The address b/w any segment is called offset address.

Q. LEA BX, [DI] : address of DI goes to BX
mov BX, [DI] : contents of address of DI → BX

Q. LEA BX, LIST ≡ mov BX, offset LIST
* LEA used for copying address to register.
(although slower than mov instruction)
→ LEA BX, NUMB ≡ mov BX, offset NUMB
→ LEA BX, [DI] ; mov ~~BX~~, offset [DI]
→ not possible.

Q. Instruction :- SI, [BX+DI]
BX = 1000H → address
DI = 2000H
What is there in SI ?
SI = BX + DI = 3000H

suppose: BX = 1000
DI = FFFF
SI = 0F00
} such an addition, called as MODULO-64K SUM

Q. WAP to exchange contents of BX & CX.
⇒ 2 memory locns DATA1 & DATA 2.
Contents of SI → BX, DI → CX
Given: load SI with address of DATA 1
load DI with address of DATA 2

eg. 4.3

```
.CODE
.STARTUP
    LEA SI, DATA1
    LEA DI, DATA2
    mov BX, [SI]
    mov CX, [DI]
    mov AX, BX      } See ALITER also for textbook
    mov BX, CX
    mov CX, AX      }
.EXIT
END.
```

★ For * LDS, LES, LFS, LGS, LSS : MOD = 11 is not used.

Sol: Eg-4.4 from textbook

Section - 4.4

STRING DATA TRANSFERS

String Instructions :-

LODS
STOS
MOVS
INS
OUTS

String Instructions

While dealing with flags, D, DI, SI : req.

→ D : Dirⁿ flag
: only for string instruction.
: D = 0 : auto increement
D = 1 : auto decrement.

Puffin
Date
Page

* Further Instructions : STD : Set dir^n flag (D=1)
CLD : Clear dir^n flag (D=0)

* In String Data Transfer.
→ when data is | contents of SI / DI incremented / decremented

1 B	by 1
Word	by 2
Double	by 4

* DI : access ES
* SI : access DS.

→

* ~~LOA~~ LODS:
loads AL, AX or EAX with data stored in DS.
→ If D=0 : SI increments.
D=1 : SI decrements.
eg: , if data is 1 B, D=0 ⟳ SI → SI+1
→ LODS B : AL selected..
→ LODS W : AX selected.
→ LODS D : EAX selected.
→ LODS Q : RAX selected.

Date 21.3.2013
Page

Section - 4.4 (continued)

§ **STOS** (opposite fn is that of LODS)

↳ It will store the contents of AL, AX or EAX at extra segment.

↳ Extra segment is always addressed by DI

↳ Data segment is always addressed by SI

i.e ES:[DI]

DS:[SI].

Store String → STOS B (Byte) ES:[DI] = AL

→ STOS W (word) ES:[DI] = AX

→ STOS LIST ES:[DI] = AL

If LIST is a byte

Repeat instruction → REP STOS B

§ ✱ **MOVS**: Transfer is b/w data segment & extra segment.

✱ B/w segment register & Memory to memory

DS:[SI]

ES:[SI]

→ MOVS B ES:[DI] = DS:[SI]

Ily,

→ MOVS W

→ MOVS D → Double word

→ MOVS Byte 1, Byte 2.

§ INS → Input, String
↳ Data is transferred from i/o device to ES:[DI].
↳ DX: always holds the ADD (address) of i/p device.
→ INSB ES:[DI] = [DX]
→ INSW "
→ INS LIST "

§ OUTS (opposite of INS) addressed by SI
Data from memory (DS) is put into i/o device.
Instructions :- OUTSB [DX] = DS:[SI], OUTSW, OUTSDATA7

★ Miscellaneous data transfer Instructions :
✓ XCHG → Exchange contents b/w Registers & Memory loc^n
✓ IN & OUT
✓ MOVSX & MOVZX → Zero extension
✓ BSWAP → Sign extension
✓ CMOV → Byte → swap

eg Instructions
XCHG AL, CL
XCHG CX, BP
XCHG AL, DATA2

✓ IN & OUT
↳ Similar to INS & OUTS.
↳ In them, data is storing. Here, nothing like that
↳ IN
Fixed Port Addressing — Variable port Addressing

Sample instructions :- IN AL, P8 (P8 → Port no. 8)
IN AX, P8
OUT P8, AL
OUT P8, AX

★ example 4.12 : Imp.

✓ MOVSX & MOVZX

↳ eg: MOVSX CX, BL.

Suppose BL = 10001111

BL → 8 bits, CX = 16 bits

So, what goes to CX?

Extend BL by adding 8 more bits.

Where, the values of the 8 bits depends on value of MSB of BL

i.e. if BL = (1)1000111 —extension→ (1111 1111)(1)1000111

(0)1000111 (00000000)(0)1000111

✱ MOVZX: In zero extension, the extended bits are '0' always.

✓ BSWAP

↳ for 32 bits of data ⇒ 4 bytes of data

So, swapping done as :-

1st byte swapped with 4th byte

2nd byte " " 3rd byte

eg: BSWAP EAX

If EAX = 00 11 22 33 H (initially)

After instruction :-

EAX = 33 22 11 00 H

✓ CMOV → mov.

→ conditional

↳ ✓ depends on the initial/previous cond^n

↳ Instructions :- CMOV(B) → Below zero value.

CMOV(AE) → Above & equal to zero value

CMOV(Z) → Zero value

* µp handles ≃ 32,000 instructions of Add^n

Puffin
Date
Page

Chapter - 5

ARITHMETIC & LOGICAL INSTRUCTIONS

Section - 5.1

Add^n, Subtraction & Comparison.

1) Register Add^n
2) Immediate add^n
3) Array add^n
4) Increment add^n
5) Add^n with carry

Same types as that of Add^n

* NOT POSSIBLE
  → Add^n b/w :
  ✓ Memory loc^{ns}.
  ✓ Segment Registers

D Register Add^n.
eg: ADD CX, BX

* Only changes seen are changes in FLAGS (except Interrupt & Trap flag)

2) Immediate.
eg. Add AL, 33H

3) Array Add^n
eg. Add AL, array[SI]
Add AL, array[SI+2]
add AL, array[SI+4]
adding diff^t elemts. in an array.

Comparison: Compares contents of 2 registers.
eg: CMP BL, AL

* Another type: Comparison & Exchange
i.e, comparing & exchanging the result.
Source & destin^n

4) Increment Add^n:
- contents of register incremented by 1

eg: INC BL
INC SP

✓ BYTE PTR
WORD PTR
These are used in case we want to increment memory loc^n

eg: INC BYTE PTR [BX]

5) Add^n with Carry

eg: ADC AL, AH
→ when data in instruction is > 16 bit.
Here, in above instruction, AL + AH + C gets stored in AL

eg: XADD = Add^n is performed, but contents of destin^n goes to source register
(80486 - Pentium 4)

Puffin
Date 2/4/13
Page

Section - 5.2

✱ Multiplication & Division

→ Multiplic^n done b/w Reg & Reg. or Reg. & Memory loc^n.

→ Data → 8
16
32 bits.
Signed
Unsigned integer.

→ Instruction :- MUL : Unsigned
IMUL : Signed

✱ Multiplic^n of

Data is of	Product stored in
8 bit x 8 bit	16 bits
16 x 16	32
32 x 32	64

multiplic^n

eg : for instruction mul CL
→ (AL) x CL happens.
→ Result gets stored in (AX) always.

eg : IMUL DX
→ 16 bit data, Result is 32 bits, gets stored in DX-AX

eg : MUL BYTE PTR [BX]
→ contents in memory loc^n shown by BX gets multiplied to AL. Result → AX

eg : MUL (ECX) → 32 bit
MSB LSB
64 bit result, gets stored as (EDX)-(EAX)

Puffin
Date
Page

Q. Multiply 10 & 5. Result is 50. Store the result in DX.

```
.model tiny
.code
.startup
mov al,5
mov bl,10.
mul bl
mov dx,ax
.exit
end
```

Q Multiply the contents of DS

mul DS ×

mul WORD PTR [SI]

∵ data is word, DS will be indexed by SI.

* Division :

8 bit division : No. to be divided is converted to 16 bits & then divided.

eg : 8 ÷ 2 = 4 → Quotient

8 → Dividend, 2 → Divisor

Dividend → converted to 16 bits → stored in AX.

After division : Quotient → AL & Remainder → stored in AH

(Note : AX is over written)

16-bit division :

Just like for 16 bit multiplic", Dividend gets stored in DX-AX (for making 32 bit)

data is converted as

- Unsigned
  - extension as MOV ZX
- Signed.
  - extension done as MOVSX.
  - Converted using
    - CBW → convert Byte to word
    - CWD → convert word to double word.

Q Divide -100 by 9

H L

0100

-100 gets stored in DX-AX

9 gets stored in CL

ALWAYS

* Transfer divisor to CL / CX

* Instruction :

```
MOV CL, 9        → 8 bits.
MOV AX, -100     → (The data is converted to 32 bits & transferred to DX-AX AUTOMATICALLY)
CWD
IDIV CX
```

→ data from DX-AX is accessed, divided by CX & then stored in AL & AH.

Buffin
Date 14/12
Pg

Q Divide the contents in SI
Instruction :- IDIV SI.

Section - 5.3

BCD & ASCII ARITHMETIC

★ BCD ADJUST INSTRUCTIONS : reqd to confirm that result is in BCD form.

- DAA : Decimal Adjust after Addition.
- DAS : Decimal Adjust after Subtraction.

→ as the name says, its used after add" instruction
after carry add" "
→ it functions ONLY in AL register
→ eg: ADD BCD 1234 with BCD 3099

Code.

```
mov DX, 1234
mov BX, 3099
mov AL, BL      (∵ DAA only for with AL)
add AL, DL      (i.e, first 2 bits of BX & DX are
                 added)
DAA
mov AL, BH      (next half)
add AL, DH
DAA  ------> to confirm that result is in
             BCD form
```

• DAS : result gets stored in (AL) always.
→ works the same way as DAA
eg : SUB BCD 4321 with BCD 3077
Store result in CX

```
Code: mov DX, 4321
      mov BX, 3077
      mov AL, DL
      Sub AL, BL
      DAS
      mov CL, AL
      mov AL, DH
      Sub AL, BH
      DAS
      mov CH, AL
```

ASCII ARITHMETIC INSTRUCTIONS

→ (adding) ASCII nos., result should be in ASCII.
↳ or any oper^n
→ Decimal : 0-9 ; ASCII : 30-39
→ Instruction types :
• AAA : Ascii adjust after addition
• AAD : ASCII adjust BEFORE division
• AAM : Ascii adjust after multiplic^n
• AAS : " " " subtraction.

• AAA :
eg. adding 31 & 39.
Result is 6A (not an ASCII). So, convert it to ASCII. So, use AAA instruction
↳ ADD 3030 to result.

31
+39
6A → not an ASCII
≡10
→ to store as 0100

to get equivalent ascii,
S1) use AAA
S2) add 3030 to 0100 = 0130

Code:

```
MOV AX,31
ADD AL,39
AAA
ADD AX,3030
```

AH AL
0301
0309
+01
0310
+3030
3330

0031
0039
31
+39
6A
0110 1010

eg: If result is 23 ≡ 0203
now in ASCII = 0203
3030
3233 i.e stored like 32-33 (within range of 30-39)

• AAD

eg: Code:- MOV BL,9
MOV AX,702H
AAD
DIV BL

§ Section 5.4

BASIC LOGIC INSTRUCTIONS

→ (all) flags get affected.

→ Instructions :

→ AND : Performs logical Multiplic^n.

If A, B are i/ps & A AND B is o/p

Truth table :-

A	B	AND. (X)
0	0	0
0	1	0
1	0	0
1	1	1

→ Memory to Memory, Segment Addressing } 2 modes NOT used by AND instruction

→ GATE

(Costly)

→ Instruction
AND BX
(very cheap)

✱ MASKING :

Clearing the bits in binary no.

eg. data is 3135

To mask leftmost BITS of data

So, mov BX, 3135.

→ 0011 0001 0011 0101

Leftmost bits of BH & BL.

Instruction :- AND BX, 0F0F

Result in BX :- 0000 0001 0000 0100

Masked ✓

→ also called Inclusive OR

→ **OR** : Performs logical addition of 2 i/p's A & B, the truth table

A	B	A+B (A OR B)
0	0	0
0	1	1
1	0	1
1	1	1

→ Memory to memory, segment register { These addressing modes are not allowed.

eg: mov AL, 5
mov BL, 5
mul BL
AAM
OR AX, 3030 : to make to ASCII.

o/p : BL stored the value of 35, as: 0305
+ 3030
ASCII Value ← 3335

→ Exclusive-OR (odd parity)

→

A	B	A⊕B
0	0	0
0	1	1
1	0	1
1	1	0

Truth table

→ Segment register addressing modes not allowed.

✱ Instruction :

AND - Clears bits ⇒ making 1.
OR - Sets the bits
XOR - Complement the bits.

Q. Make contents of a register, say CH as zero. using XOR gate :-

Instruction : XOR CH, CH : 2 Bytes ✓
(MOV CH, 00H) : 3 Bytes
Preferred

✱ Test & Bit Test Instructions

→ Tests a bit being 0 or non zero
→ result of test instruction : shown in Zero flag
eg : suppose the 4th bit in CX register is 0. So, test bit shows 1 & Zero flag also shows 1
→ checking is done by AND operation, but, difference is, that the value of registers don't change. Only flag bits change
→ Test instruction is followed by JZ & JNZ instructions (JZ : Jump if zero, JNZ : Jump if not zero). (Std)
→ To test : Right ~~left~~ most bit :
TEST AL, 1
JNZ RIGHT.

Left bit :
TEST AL, 128
JNZ LEFT.

overall, they test value of AL

Test the 4th bit of AX for zero

Puffin
Date
Page

* Diff^t types of BIT TEST Instructions :
  - ✓ BT : Test the bit for zero ; eg: BT AX,4
  - ✓ BTC : Test the bit for zero & Complement it
  - ✓ BTR : Test the bit & Reset it
  - ✓ BTS : Test the bit & Set it to 1.

eg : BTS AX , 4
↳ The 4th bit will be tested & it will be set to 1.

* NOT & NEG :
  - ↳ NOT CX : 1's complement done of contents of CX
  - ↳ NEG CX : 2's complement done of the contents of CX

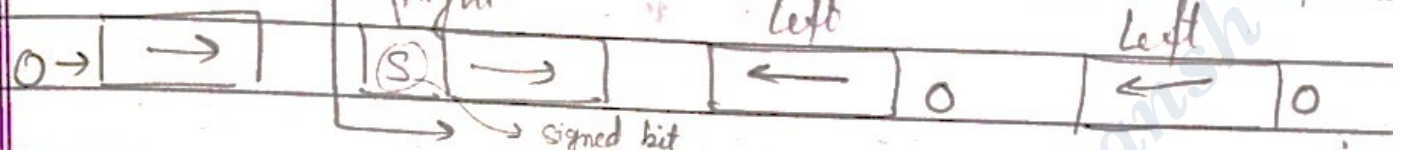
§ SECTION - 5.5

SHIFT & ROTATE

- Instructions in which contents of register/memory loc^n is moved.
- Instructions used to control the I/O devices

☆ SHIFT Instruction

SHR — Shift Right (Logical shift)
SAR — Arithmetic Shift Right
SHL — Shift Left
SAL — Arithmetic Shift Left



Just like done in digital design.

- Shift left ×2
- Shift right ÷2.

eg: consider 1010 ≡ 10

1 0 1 0 0

16 + 4

= 20 = 2 × 10.

Q Write instruction to shift contents of DX, 14 times

SHL DX, 14

- The no. of times the shift has to be done is written in 2 ways:

mov CL, 14 SHL DL, CL (check) or DX, CL.	SHL DX, 14.
(use of CL register)	(use of data directly)

☆ Double precision shift:

∃ 3 operands in the instruction (instead of 2, as above)

eg SHRD AX, BX, 12.

AX : Shift right by 12 position
Rightmost 12 bits of BX go into leftmost 12 bits of AX.

Q. Shift left 16 bits of BX & leftmost bit of CX ↷ Rightmost of BX (16)

eg :- SHLD BX, CX, 16

Check

eg, say :- BX = 1111 1111 1111 1111

Thun: SHR BX, 16.

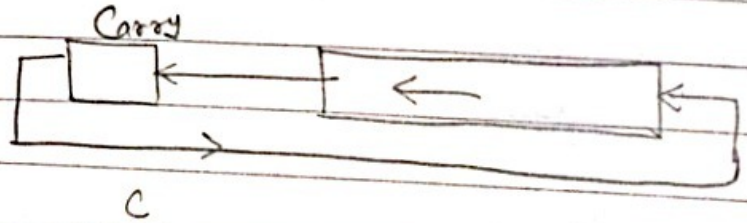
o/p :- BX 0000 0000 0000 0000

ROTATE Instruction :-

Rotates the contents of register or memory loc^n from one end to the other.

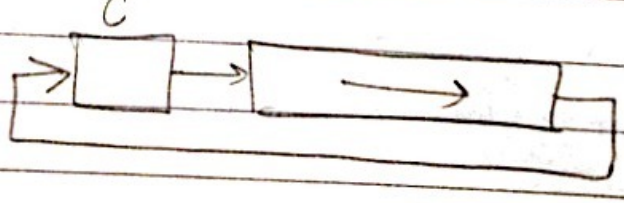
ROL - Rotate from left end
RCL - Rotate from left end & entering from carry flag
ROR - Rotate from right end
RCR - Rotate from right end & entering from carry flag.

RCL

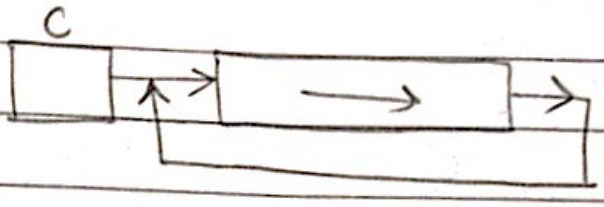


ROL

RCR



ROR



eg. ROL SI, 14
↳ SI contents rotate left by 14 places.

eg: mov AL, F0H ⟶ AL = 1111 0000
mov BL, 8FH ⟶ BL = 1000 1111
add AL, BL AL = 0000 1111
ROL AL, 4 ↳ carry

o/p for ROL AL,4 : 1 1111 0000 (1 = Carry)

o/p for ROL AL,8 : 1 0000 1111 (1 = Carry)

✡ BIT SCAN Instruction

Contents of each register is scanned for the 1st ONE BIT.

✓ Searching / scanning done from
↳ Left to right
BSF
(Bit Scan Forward)

↳ Right to left
BSB
(Bit Scan Backward)

eg: BSF EBX, EAX

Contents of EAX are searched. The 1st bit found → its posn goes to BX.

Opposite of BSF

Puffin
Date
Page

Section - 5.6

STRING INSTRUCTION

String Scan

Extra Segment (ES)

A block of memory is compared with contents of AL, AX or EAX

- Depending upon size of contents :-
  - SCASB
    AL compared with ES
  - SCASW
    AX with ES
  - SCASD
    EAX with ES

Instructions accompanied by

REPE → Repeat equal

REPNE → Repeat not equal.

eg: REPE SCASB.

Note :- Content is NOT present in ES (to compare). The content is present in (DI x 10 + ES)

String Compare

2 memory locns are compared.

✓ Accompanied by REPE or REPNE

- Depending on its size,
  - CMPSB
  - CMPSW
  - CMPSD.

eg :-
REPNE CMPSB

Chapter - 5

SUMMARY of Instructions

5.1. Way to write : Instruction Destination, Source

ADD. : Addition

INC : Incremental addition

ADC : Addition with carry

→ carry bit is added with result of addⁿ

SUB : Subtraction

DEC : Decrement

SBB : Subtraction with borrow

→ carry flag having borrow is subtracted from the result of subtraction.

CMP : compare instruction (a type of subtraction)

: change comes only in flag bits

CMPXCHG : compare & exchange

CMPXCHG Destinⁿ, Source

Contents of source = contents of AX → Contents of destinⁿ copied to AX

Contents of source ≠ contents of AX → Contents of source copied to AX

5.2. Way to write : Instruction Source

MUL : Multiplicⁿ (unsigned)

IMUL : Multiplicⁿ (signed)

→ AX multiplied by source

→ contents stored in AX

→ destination is always AL, AX or DX-AX, EDX-EAX → for 32 bit

DIV : Division (unsigned)

IDIV : Division (signed)

→ AX ÷ (source)

→ quotient : AL & remainder stored in AH.

* Use of instruction :

MOVZX : for unsigned } reqd in program

CBW : for signed nos.

5.3

DAA : decimal adjust after addition
DAS : decimal adjust after subtraction

AAA : ASCII adjust after addⁿ
ABD : ASCII adjust before divisionⁿ
AAM : ASCII adjust after multiplicⁿ
AAS : ASCII adjust after subtraction

} Add 3030 to make ASCII

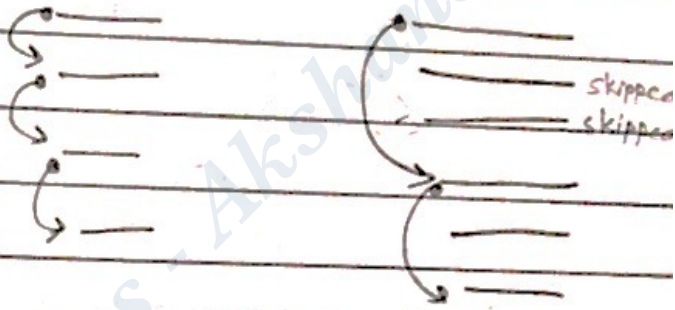
Chapter - 6

PROGRAM CONTROL INSTRUCTIONS

↳ Instructions directing flow of program:

step by step

skipping instruction



* Such skipping or flow of instructions is seen in:-
  - ✓ Compare
  - ✓ Test
  - ✓ Jump

  } Instructions

Section - 6.1 : THE JUMP GROUP

* Jumps:

Allows programmer to skip sections of a program & branch to any part of the memory for next instruction.

* UNCONDITIONAL JUMP

SHORT jump	NEAR jump	FAR jump
• 2 bytes instruction	• 3 bytes instruction	• 5 bytes
• allows jump within +127 bytes to -128 bytes of memory	• ± 32 KB is allowed range (within Code Segment)	• Jump anywhere within real memory sys.
called as Intrasegment	called as Intrasegment	called as Intersegment (not limited only to Cs)

• Short Jump | Near Jump | Far Jump

EB	Disp.

E9	Displ Low	Displ High

EA	IP_L	IP_H	CS_L	CS_H

op-code

called as Relative jumps ∵ they can be moved along with related software to any loc^n.

✱ Short Jump

✱ Jumps occurs :-

Displ. + IP = Jump address.

→ meaning, the no. of the instruction, as seen in code view.

✱ Instruction : JMP SHORT _____

eg : Consider the instruction

0009 JMP NEXT

IP

If Displ. = 0007,

the new Jump address = 0009
+ 0007
= 0010

eg :
```
MOV  AX, 1
ADD  AX, BX
JMP  SHORT  NEXT
 ;    ;    ;
 ;    ;    ;
NEXT : MOV BX, AX
```

• Self : example 6.2, 6.3.

Puffin
Date
Page

✱ Near Jump :-

Same as far jump, distance is more

Displacement :- 16 bit signed no.

✱ Instruction :- ~~JUMP~~ JMP

Jump address = Signed Displacement + IP
+ (CS x 10)

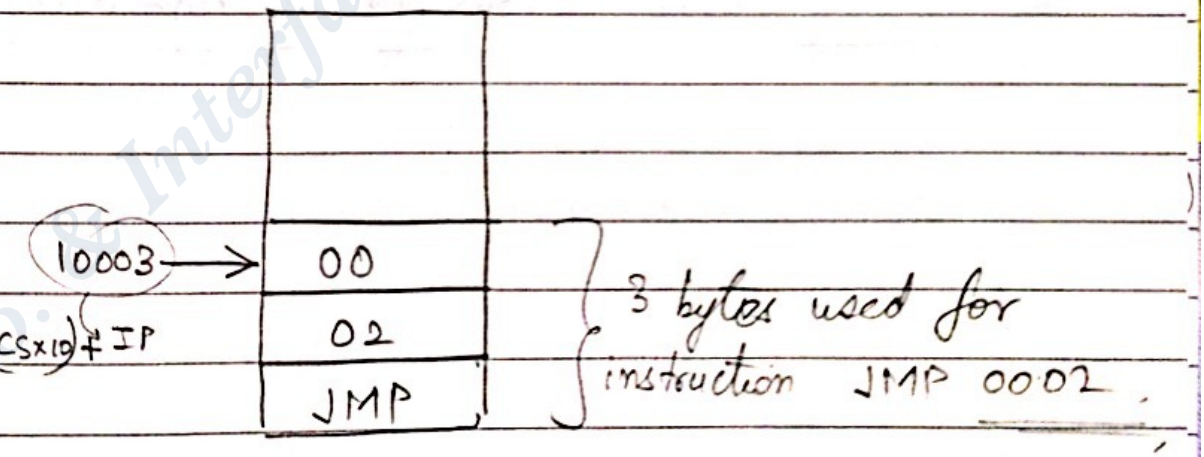
- relative Jump
- relocatable jump

IP → Instruction pointer.

✓ Actual address = (CS x 10) + IP + Displ.
→ where the instruction goes

✱ Self : example 6.2

✱ 3 bytes ⇒ it needs 3 bytes to write instruction



✱ Far Jump :-

5 bytes of instruction

2nd & 3rd byte : Offset address

4th & 5th byte : Segment address.

eg :

A3
00
01
27
JMP

5 bytes used for Far Jump.
Segment Add = A300
Offset add = 0127.

Jump Address = (Segment Add) x 10 + Offset add.

* Instruction :- JMP FAR ____

✱ JUMPS WITH REGISTER OPERANDS

- register 16/32 bit
- an indirect jump.

Self: example 6.4.

✱ Indirect Jumps using an Index

JMP TABLE [SI]

→ makes use of '[ ]'
→ '[ ]' : refers to memory loc" so that the instruction can directly access jump table.

- The address is referred to SI or DI
- Its indirect jump (∵ address is calculated)
- Its either short or near. (If far, it'll be written in instruction & as FAR PTR)

✱ Conditional Jump & Conditional Set

↳ jumping to particular loc^{n} based on cond^{n}s

like JG : Jump if > : for signed
JA : Jump if above : for unsigned nos
JB : Jump if below

Flags : S, Z, C, P, O
↳ These flags are tested in the cond^{nal} Jump.

Table 6.1

Jump.

Signed (−128 to +127)		Unsigned nos (0 to 255)
JG	Different instructions	JA
JL		JB
JGE		JAB
JLE		JBE
JE		JE
JNE		JNE

- Conditional Set
  The instruction sets a byte to 01 or, clears a byte to 00.

Self Table 6.2 :- Different (set) instructions

eg : SETNC MEM
↳ a label named MEM has the value set to 01 when ∄ no carry (i.e C=0)
00 when ∃ carry (i.e C=1)

§ LOOP Instruction
↳ combin^{n} of 2 things :
• decrement CX
• JNZ → cond^{nl} jump

S1) If CX≠0, it jumps to the address indicated in the instruction.
S2) for CX=0, the next instruction follows.

* CONDITIONAL loops.
↳ A loop instruction with a cond^{n}
LOOPE : loop if equal to
LOOPNE : loop if not equal to

Date 22.4.13
Page

Section - 6.2

CONTROLLING THE FLOW OF PROGRAMS.

Instructions : The DOT COMMANDS :

- .IF
- .ELSE
- .ELSEIF
- .ENDIF (end the if stat)
- .WHILE
- .ENDW (end of while cmd)
- .BREAK
- .CONTINUE

* .IF Command
  ↳ Relational operators used with .IF :

equal ← == , != (not equal) , > , >= , < , <= , & , ! , && ,

|| → logical OR , | → or

& → bit test ; ! → logical inversion ; && → logical AND

ex. 6.9 , 6.10, 6.13 : self

* .BREAK & .CONTINUE
  ↳ used with .IF & .WHILE instructions.

* REPEAT-UNTIL loop.
  - .REPEAT (Start of the loop)
  - .UNTIL (end of loop which holds the cond^n)

(just like :- do ... while } Instruction in C lang.)

* .UNTILCXZ
  Use of CX register as a counter to repeat a loop fixed no. of times.

Section - 6.3

PROCEDURES :

- A set of instructions to perform a task. (Just like Functions in C lang.)
- After task is over, control goes back to program.
- Its a reusable section of software that is stored in memory. It can be used & reused any no. of times.
- Disadvantage : Takes time to like & return back.
- Syntax :
  - Begins with : PROC
  - Ends with : ENDP
- Types

NEAR	FAR
(Intrasegment)	(Intersegment)
b/w the same segment.	b/w diff^t segments.

- RET : Return cmd (used for NEAR & FAR both) (To return back to program).
  - RET of ~~near~~ NEAR uses C3H opcode
    FAR uses CBH opcode.
- for NEAR, RET removes 16 bit no. from stack & places it in IP to return.
- for FAR, RET removes 32 bit no. from stack & places it in IP & CS to return.

- address of the next instruction goes to the IP → for NEAR.
- address of next instruction as well as CS is used (put in IP) → for FAR

• when the work is 'global' : FAR
  'local' : NEAR.

CALL:

Transfers the control / flow of program to procedure

↳ Diff b/w Jump & call:
CALL saves return address on the stack.

Types

NEAR call

- 3 bytes long
  ↳ 1'st byte : opcode
  2nd & 3rd byte : displacement

S1) On execution, offset address of next instruction is pushed to stack

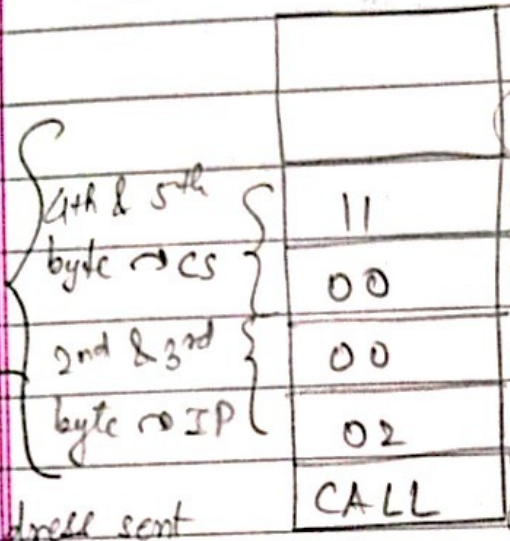
S2) Offset address of next instruction appears in IP

S3) After saving this return address, it then adds the displacement from bytes 2 & 3 to the IP.

FAR Call

- 5 bytes long
  ↳ 2nd & 3rd byte : have the new IP
  → 4th & 5th byte : have the new CS
  → 1'st byte : opcode

eg for FAR CALL



* CALLS with register operands:
  eg. CALL BX

* CALLS with indirect memory addresses
  eg - CALL TABLE[BX]

RET (Return)

NEAR Return
It removes 16 bit no. from stack & places it into IP.

FAR Return
Removes 32 bit no. from stack & puts it into IP.

Section - 6.4

Introduction to INTERRUPTS :-

↳ It can be a hardware call or software call.

- Program is interrupted by calling Interrupt Service Procedure (ISP) or Interrupt Handler.

- **Software Interrupts :-**

  3 types of CALL instructions :-

  INT | INTO | INT 3

- **Interrupt Vectors :-**
  - 4 byte no. : using them interrupts the program
  - stored in first 1024 bytes of memory
  - First 2 bytes contain IP & last 2 bytes contain CS.
  - Around 256 interrupt vectors are present, each of which contains address of ISP.

- **Interrupt vector table**

from Table 6.4

Interrupt vector no.	Address	MP	fn
0	0H - 3H	ALL	Divide Error
6	18H - 1BH	80186 - Core 2	Invalid opcode
C	30H - 33H	80386 - Core 2	Stack fault
⋮	⋮	⋮	⋮

Software Interrupt :-

↳ INT

2 bytes long

eg. INT 10H

↳ memory loc" = 10x4 = 40H

→ IRET/IRETD :
- pops stack data into IP
- pop stack data back into CS.
- IRET : 16 bits
  IRETD : 32 bits

→ INT 3 :
- designed as a break pt, for breaking flow of software.
- 1 byte instruction.

→ INT O :
- Tests an overflow flag.
- If O=0 : Interrupt occurs
  O=1 : no oper^n.

Self * Interrupt Service Procedure.

Q Develop a short sequence of instructions that uses REPEAT-UNTIL construct to copy the contents of byte sized memory block A ⟶ byte sized memory block B, until 00H is moved.

MACRO's :-

- A group of instructions that perform a certain task.
- It is inserted in a program at the pt. of usage by ASSEMBLER.
- There is no need to 'call' a Macro (as is req^d for a procedure)
- It is created in the same way as a new op. code is created.

- ends with ENDM.
- To begin a macro: keyword : MACRO.

ex :- ADDSTR MACRO SUM, TOTAL

(ADDSTR → name of macro; MACRO → keyword; SUM, TOTAL → variables.)

```
ADD   AX, BX
MOV   AX, CX
ENDM
```

→ written in the beginning

Now, whenever ADDSTR is used in a program, all these instructions are executed.

* There is no change in IP & moving/calling is not there. So, macro is faster over procedure.

On writing name of macro the instructions of macro are copied into program by assembler.

name of macro

eg. MOVE MACRO A,B
PUSH AX
MOV AX,B
MOV A,AX
ENDM

writing Macro

macro used.

MOVE VAR1, VAR2 → will be treated as A, B & instructions will execute as shown.

0000	50	1	PUSH AX
0001	A1	1	MOV AX, VAR2
0004	A3	1	MOV VAR1, AX
0007	58	1	POP AX

They are written instead of A & B.

This is automatically copied & executed as soon as MOVE VAR1, VAR2 is written.

Section - 8.2

USING KEYBOARD & VIDEO DISPLAY

- how to use keyboard and video display connected to PC in a program.

* Reading values / keys from keyboard in a DOS program
  ↳ use of a fn :- Interrupt
  ↳ INT 21 H.

* Reading a key with an (echo) → means, we'll get the display

```
KEY  PROC  FAR          ← name of proc.
     MOV   AH, 1        } directs the user to
     INT   21H.         } enter value from keyboard
                          (4th like scanf in C)
     OR    AL, AL       } clearing AL
     JNZ   KEY1
     INT   21H
     STC                } set carry → If C=0, or C=0, then, the meaning of a carry is understood.
KEY1 :
     RET
KEY  ENDP
```

displays
read & echo a char.
calls a procedure that processes DOS fn calls

Puffin
Date
Page

* Just like C lang. has printf & scanf, in assembly language :-

printf : mov AH, 2
INT 21H
↳ The value stored in DL is displayed

scanf : mov AH, 1
INT 21H
↳ The value gets stored in AL (which is inputted by user).

* When value is stored, it is always taken as ASCII value. This value is
  - Extended ASCII : If C = 1
  - Normal ASCII : If C = 0.

* Reading a key without a display (= Echo)
↳ fn call no. = 06H.

eg:
```
keys proc near
    mov AH, 6
    mov DL, 0FFH
    int 21H
    JE keys
```
} the instructions that will read a char.

```
OR  AL, AL
JN  Keys1
INT 21 H
STC
Keys1 :
    ret
Keys endp
```

* Note : for reading a key without display :

```
MOV AH, 6
MOV DL, OFFH
```
} necessary

* Read an entire line with an echo
  - → fn call no. for reading entire line : 0A H
  - → reads upto 255 characters read from keyboard
  - → It'll continue to accept character until 0D H is typed.
  - → DS : DX addresses the keyboard buffer.

address stored in buffer.

buffer 1 → holds count of characters

buffer 2 → holds all the characters.

eg :
```
.model small
.data
BUF1  DB  257 DUP(?)
BUF2  DB  257 DUP(?)
.code
.startup
```

```
mov BUF1, 255
mov DX, offset BUF1
Call Line

MOV BUF2, 255
mov DX, offset BUF2
Call Line
.exit

Line Proc Near
    mov AH, DA H
    int 21 H
    Ret
line endp
    END
```

- Writing to the video display with DOS functions :
  - Displaying one ASCII character :
    - fn call no :- 02 H or 06 H .
      - display 1 char. at a time
    - :- 09 H
      - display entire string of chars.

combin" of them moves the cursor to next line at left margin of the screen

- :- 0D H
  - displays carriage return.
- :- 0A H
  - displays a line feed

eg :
```
.model tiny
.code
disp macro A
    mov AH, 06H
    mov DL, A        → displaying any parameter
    int 21H
    endm
disp 0DH
1   mov AH, 06H
1   mov DL, 0DH
1   int 21H
    disp 0AH
1   mov AH, 06H
1   Mov DL, 0AH
1   INT 21H
.exit
end
```

whenever 1 is written, these lines are inserted in program by assembler

= Here, A is placed in DL & A is displayed.

* Displaying a Character string.

- → String : a series of ASCII coded chars.
- → Each string ends with a null char '$'
- → End of string is given by calling : 09H. (for $)
- → Before executing INT 21H, fn call no. 09H requires that DS:DX addresses the char. string.

eg: .model small
.data
MES DB 13, 10, 10, 'A test line .$'
.code
.startup
mov AH, 09
mov DX, offset MES
int 21 H
.exit
end

✱ Using BIOS video function calls:
Basic i/p o/p sys.

→ Diff. b/w DOS & Video BIOS fn calls:
DOS fn call : INT 21 H -
BIOS fn call : INT 10 H .

① DOS: fn calls read & display a character with ease, but cursor positioning is difficult.

DOS	BIOS
1- Reads & displays a char. with ease, but cursor pos^n is difficult.	1- Allows more control over the video display.
2- take more time to execute	2- Faster (takes less time to execute.
3- F^n calls do not allow cursor placement	3- Allow cursor placement

* Cursor Position:
Video BIOS f^n call no. 03 H
BH : page no.
DH : Row no.
DL : Column no.

- BH : Page number : 0 used
  1-7 not used.
  ↳ recognized by CGA, EGA & VGA adapters.
- DH : Row number : 0 : uppermost
  ↳ (0-24) 24 lowermost
- DL : Column no. : 0th column — left hand pg.
  ↳ (0-79)

* Home position: Row 0, Column 0.
- Starting with home pos^n, use DOS to 2000 character of blank space on video display.
  ↳ 80 characters per line × 25 character lines
- Finally, cursor goes back to home pos^n.

- eg:

```
.model tiny
.code
home macro          > home cursor macro.
mov AH, 2           > f^n 02 H.
mov BH, 0           > page 0
mov DX, 0           > row 0, line 0.
int 10H             > home cursor
                    <- Video BIOS f^n call
endm
```

being used from the declared macro

```
• startup
home                    → home cursor
1  mov AH,2
1  mov BH,0
1  mov DX,0
1  int 10H
mov CX, 25*80           → load char. count
mov AH,6                → select fr 06 H
mov DL,' '              → select a space
MAIN 1:
    int 21H             → display a space
    loop main 1         → repeat 2000 times.
    home
1  mov AH,2
1  mov BH,0
1  mov DX,0
1  int 10H
   exit
   end
```

* read & display or clear screen

- The speed of the above program is very slow. So, instead of using only INT 06 H to clear screen, we use
  06 H along with ~~001~~ AL: 00H to blank screen
- For faster clear & home cursor program, use :
  - 08H: reads char. attributes for blanking screen.
  - DX: loaded with screen size, 4FH (79) and 19H (25).

* MOV AH, 600 : selects scroll function.

eg: Program that clears screen & homes the cursor :-

```
.model tiny
.code
home macro
mov AH, 2
mov BH, 0
mov DX, 0
int 10H
endm

.startup
mov BA, 0
mov AH, 8
int 10H  ---> read video attribute
mov BL, BH ---> load pg. no.
mov BH, AH
mov CX, 0 ---> load attributes
mov DX, 184F H ---> line (DH) 25, column (DL) 79
mov AX, 600H ---> select scroll fn.
int 10H ---> scroll screen.
home
1 mov AH, 2
1 mov BH, 0
1 mov DX, 0
1 int 10H

.exit
end
```

8. A program that displays AB followed by a carriage return and line feed combination using DISP macro

```
.model tiny
.code
disp macro var
mov DL, VAR
mov AH, 06
int 21 H
endm.
.startup
disp A
1 mov DL, A
1 mov AH, 06
1 int 21H
MOV AL, 'B'
DISP AL
1 MOV DL, AL
1 MOV AH, 06
1 INT 21 H.
DISP 13          (Carriage return)
1 MOV DL, 13
1 MOV AH, 06
1 INT 21 H.
DISP 10          (line feed)
1 MOV DL, 10
1 MOV AH, 06
1 INT 21 H.
.exit
end
```

Chapter - 9 (Theoretical)

8086/8088 Hardware Specifications

SECTION 9.1
→ PIN-OUTS & PIN FUNCTIONS.

8086
- ✓ 16 bit μP
- ✓ 16 bit data bus. (8088 : 8 bit data bus)

- Power supply requirements :-
  +5 V ± 10%.
  - 8086 draws max current 360 mA
  - 8088 " " " 340 mA

- Connecting a μp : i/p current requirements should be known.

- ✓ i/p & o/p characteristics of pins : in table 9.1 & 9.2 resp.

- ✓ Self :- Pg-305, 306

SECTION 9.3 : BUS BUFFERING & LATCHING:

✓ Related to different Bus connections.

→ Demultiplexing the busses.

→ one signal separated to give multiple signals (opposite of Multiplexing)

* Address/data bus of 8086/8088 is multiplexed to reduce no. of pins req^d for IC.

* BUSES

Address Data Control

* Buses must be present in order.

* Demultiplexing 8088

→ Latches req^d :- 74LS373 74LS573

connecting these latches to 8088, o/p is demux signals

* Demultiplexing 8086

* Buffered Sys:

→ fully buffered sys :- Basically, boosting the signals so that their reception is proper at the target

Fully buffered 8088 → 74LS244, 245, 373 } These latches, connected to 8088 o/p, signal is boosted.

Date 26/5/13
Page 1

Chapter - 10

TOPICS TO STUDY

- Section - 10.1
  (From beginning till ROM Memory)
  (all definitions)
  (diff. b/w static RAM & dynamic RAM)

- Section - 10.2
  (From beginning till PLD)
  (few examples in b/w → go through)

- Pg - 356, 357
  (all the contents on that pg.)

- Pg - 379
  (Isolated & memory mapped i/o)

- Pg - 382
  (Handshaking)
  (an example in this topic : do that too)