EE 2003 COAL

Date: M T W T F S S

Assembly → statement → (X+8) + Z = Y
↳ complex statement

Instruction → ADD → only one action perform.

MOV AL, 00H → hexa

If H not present then decimal

MOV AL, 00H
destination source.

HIGH LL	Assembly	Machine code
X = (Y+4) *3	ADD Y, 4	Instruction or bytes
statement	MUL Y, 3	1010, 111 0101
	MOV X, Y	

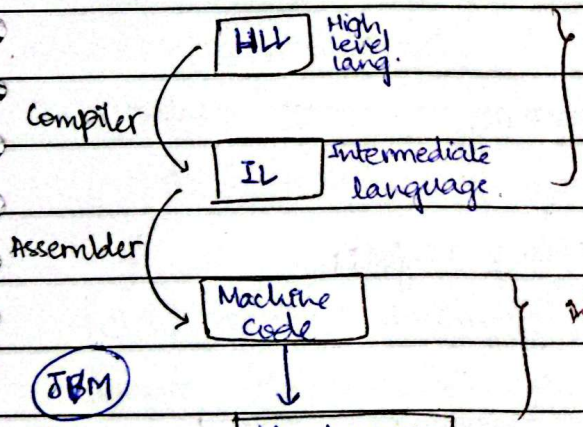
Instruction

HIGH LL TO ASSEMBLY :- 1 to many relation.

ASSEMBLY TO MACHINE CODE :- one to one relation

* In Assembly, no datatype. (Has memory spaces)

VIRTUAL MACHINE :- translate + interpret



L1 (HIGH LEVEL LANGUAGE) — VIRTUAL MACHINE VM1

L0 (MACHINE CODE) — VIRTUAL MACHINE VM0

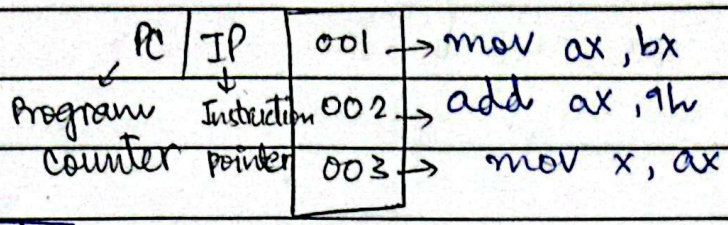
VM :- software program that emulates the function of some other physical or virtual computer.

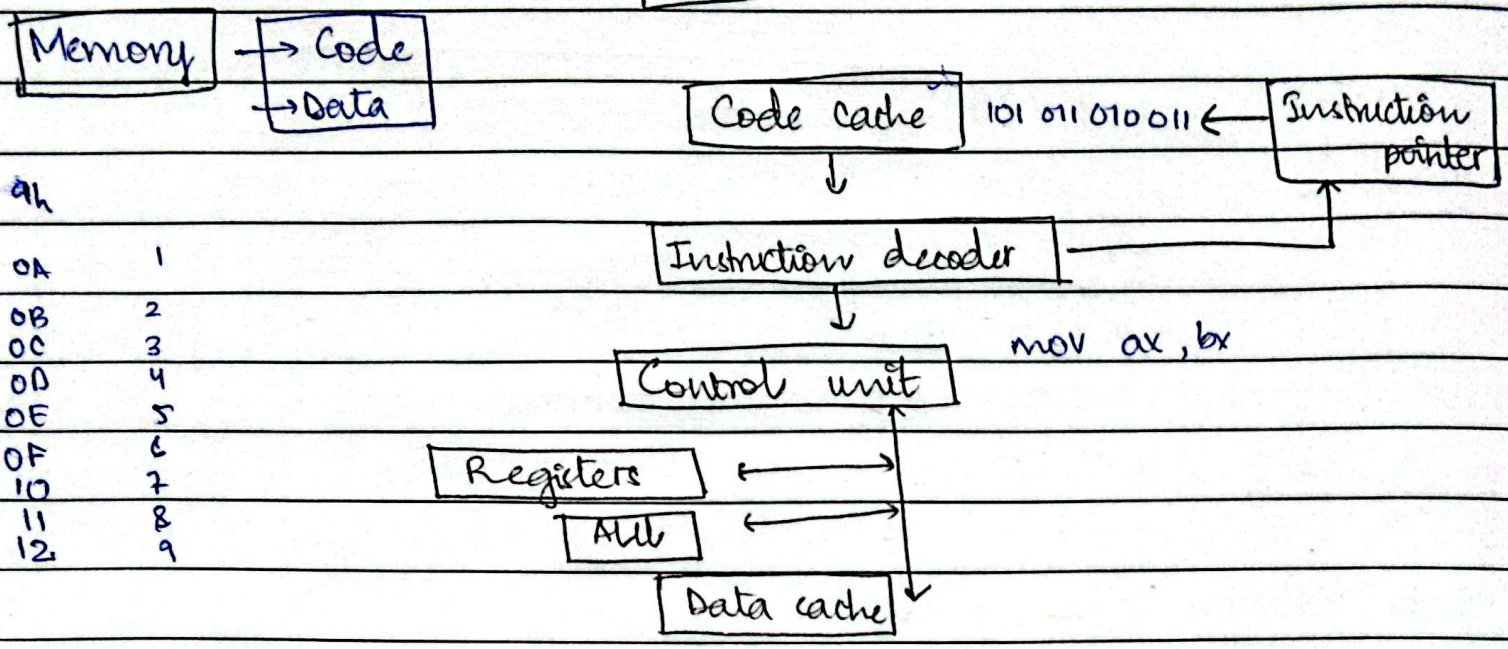
High-level language
Assembly language — VM1
Instruction set Architecture (ISA) — VM0
Digital Logic

Date: M T W T F S S

Code:- mov ax, bx
add ax, 9h
mov x, ax

Include Irvine 32.inc
data
x byte 0h
code
main PROC





9h	
0A	1
0B	2
0C	3
0D	4
0E	5
0F	6
10	7
11	8
12	9

FETCH - DECODE - EXECUTE :-

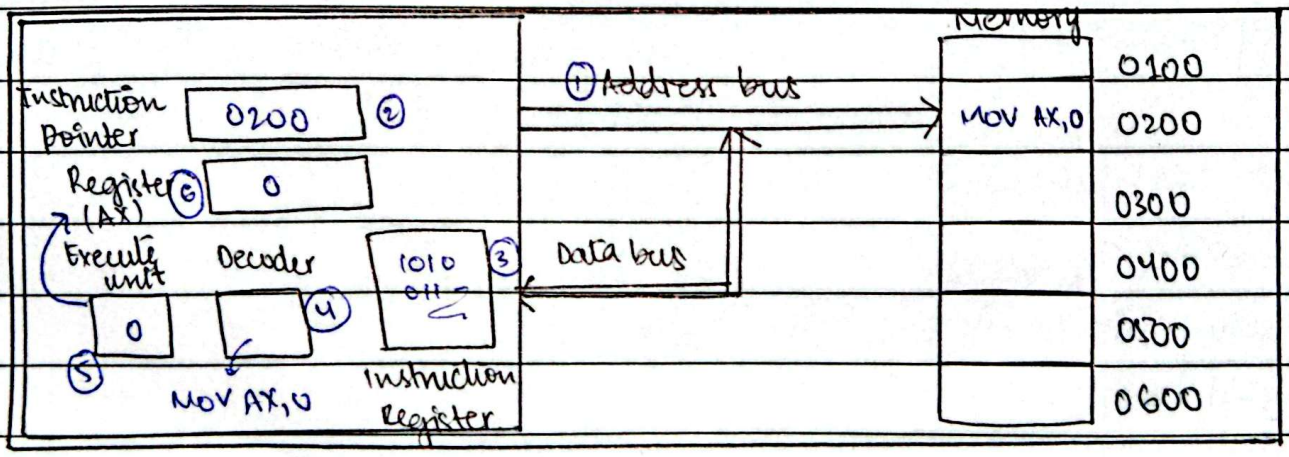
For execution, data is loaded into main memory.

Fetch :- processor fetches the instruction from memory.

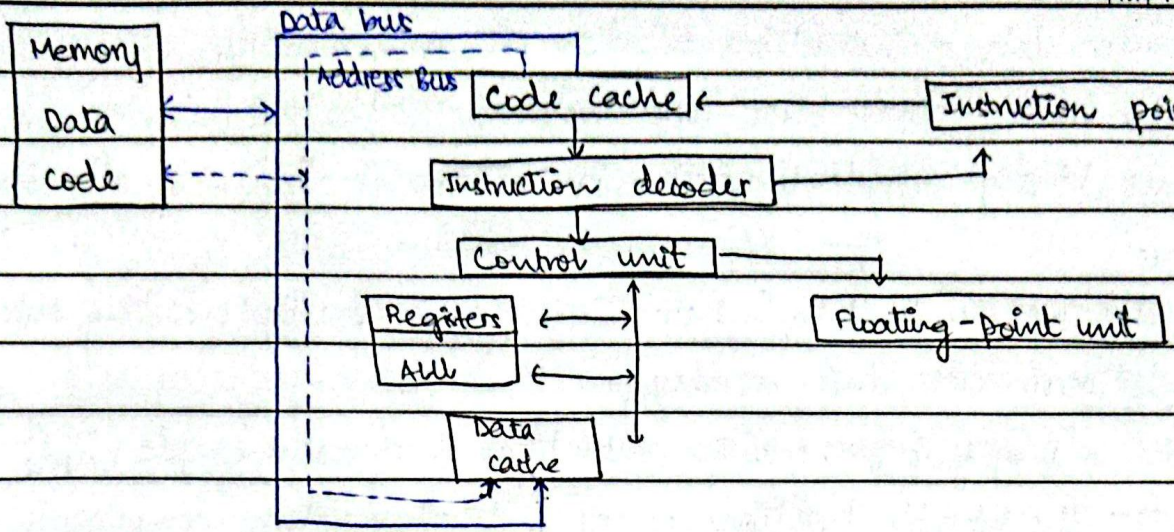
Instruction is transferred from memory to instruction register.

Decode :- Instruction is decoded by processor.

Execute :- processor executes instruction.



Date:
M T W T F S S



Memory stores both code (instructions) and data.
Bus → pathway for transferring data b/w diff. parts of CPU.
Instruction pointer → points to location in memory of next instruction to be executed.
Code cache :- fast memory that stores frequently used instructions.
it speeds up execution speed time by reducing time it takes to retrieve info. from main memory.
Instruction decoder :- translates 0101... signals into signals that CPU control unit can understand.
Control unit :- coordinates with ALU & floating point unit.
Registers :- temporarily hold data that's being processed.
ALU :- performs arithmetic & logical operations on data.
Data cache :- stores frequently used data for faster access.
After data is executed, goes to output device for display / data stored depends on instruction.

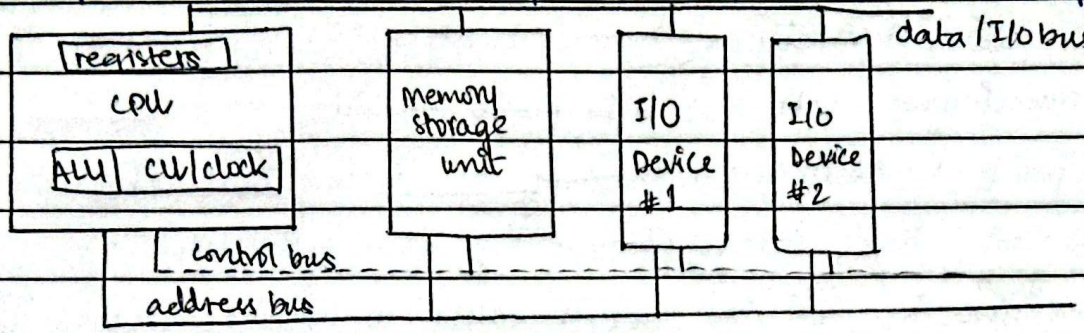
loop
fetch next instruction
advance the instruction pointer (IP)
decode the instruction
If memory operand needed, read value from memory.
execute the instruction
if result is memory operand, write result to memory.
continue loop

Bus:- group of parallel wires that transfer data from one part of comp. to another.

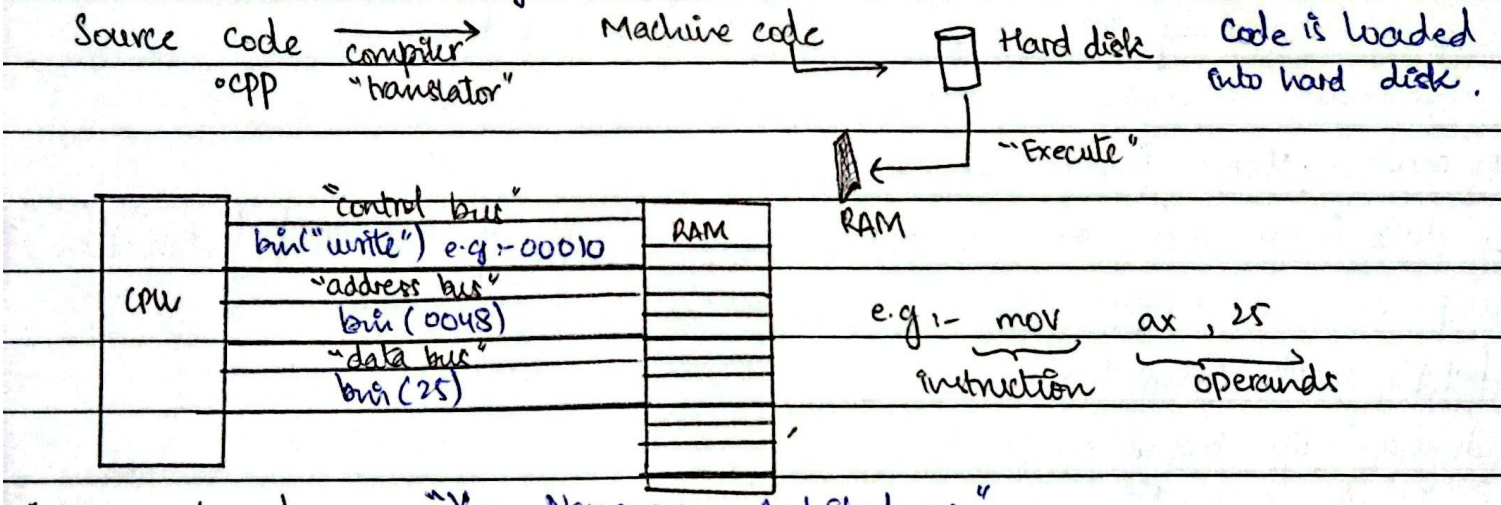
Clock is used to trigger events.
↳ measures time of single operation.
↳ synchronization

Date: M T W T F S S

Data bus → transfer data n instructions between CPU and memory.
I/O bus :→ transfer data between CPU and system input/output devices.
Control bus → uses binary signals to synchronize actions of all devices attached to system bus.
Address bus → holds address of data & instruction when currently executing instruction transfers data between CPU and memory.
Clock :- The basic unit of time for machine instructions is a machine cycle (or clock cycle).
Clock synchronizes the internal operations of CPU with other system components.



Reading from Memory :- The CPU must wait one or more cycle until operands have been fetched from memory before current instruction can complete its execution.



e.g :- mov ax , 25
instruction operands

✓ All are based on "Von Neumann Architecture".
- Code and data both stored in memory.
- General purpose
- Fetch, decode, execute.

Reading a single value from memory involves 4 steps :-
① Place the address of the value you want to read on the address bus.

Date: M T W T F S S

② Assert (change the value of) processor's RD (read) pin.
③ wait one clock cycle for memory chips to respond.
④ copy data from data bus into destination operand.
Each of these steps require a single CLK cycle.

CACHES ... WHY ARE CACHES NEEDED?

Coz RAM is very slow as compared to CPU's processing speed.
Cache is high speed memory, reduces amount of time spent in reading & writing.
When CPU begins to execute a program, it loads the next thousand instructions into cache.
If it happens to be a loop in that block of code, the same instruction will be in cache.
when processor is able to find its data in cache memory, its called cache hit.
If CPU tries to find something in cache & it's not there, its called cache miss.

Cache memory
- Level-1 cache (primary cache)
- Level-2 cache (secondary cache)

* Cache memory is faster than RAM.
  - ↳ made of static RAM
  - ↳ Dynamic RAM. (needs to be refreshed constantly).

LOADING & EXECUTING A PROGRAM:-

✓ OS searches for program's filename in current disk directory. If it cannot find the name there, it searches a predetermined list of directories (called paths) for filename.
✓ If OS fails to find program filename, it issues an error message.
✓ If program file is found, the OS retrieves info. about program's file from disk directory including file size, its physical location on disk drive.
✓ Memory block is allocated based on program's size & location (descriptor table).
✓ OS also adjust values of pointers within program so they contain addresses of program data.
✓ OS begins execution (entry point).

Date: M T W T F S S

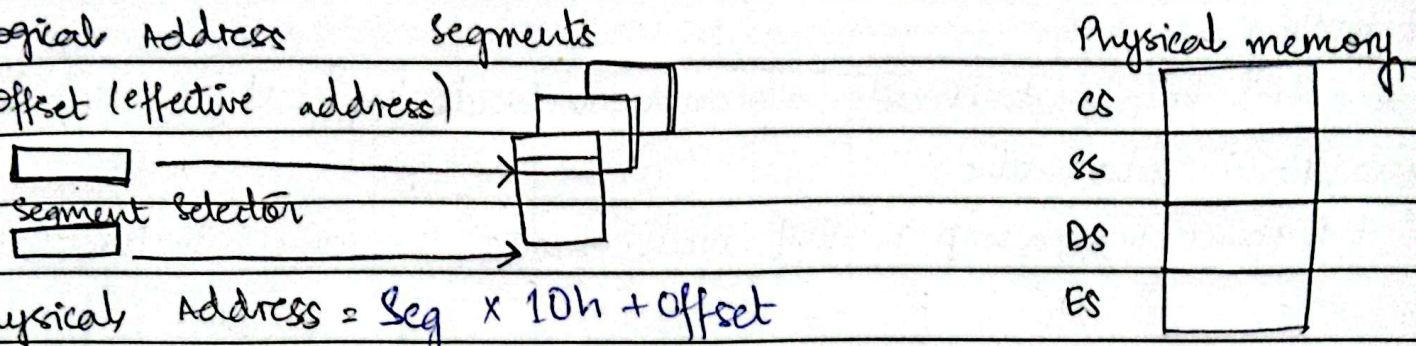
✓ As soon as program begins running, it is called a process.

✓ OS assigns process an identification number (process ID), which is used to keep track of it while running.

✓ OS's job is to track the execution of process & to respond to requests for system resources → memory, disk files and input/output devices.

✓ When process ends, it is removed from memory.

PHYSICAL ADDRESS, SEGMENT, OFFSET :-



Physical Address = Seg × 10h + Offset

Q. 0A51 : CD90h ⟶ seg : offset

Physical Address: P.A = Seg × 10h + Offset

= 0A51 × 10h + CD90h

= 0A510h + CD90h

= 172A0h

5+D = 1 sum, 2 carry

carry: 1 1

0 A 5 1 0 h

+ C D 9 0

1 7 2 A 0

Q. Address = 4A37Bh. offset = ?, Segment = 40FFh, see hexadecimal addition rules.

P.A = Seg × 10h + Offset

Offset = P.A − Seg × 10h

= 4A37B − 40FF×10h

= 4A37B − 40FF0h

= 938B0h

16 → 16+7 = 23, 23−15 = 8

2+16 = 18, 18−15 = 3

4 9 A 2 B 7 B

− 4 0 F F 0

0 9 3 8 B

Code Segment :- holds the base address for all executable instructions in program.

Data Segment :- holds the base addresses for variables. This segment stores data for program.

Extra Segment :- extra data segment used for shared data.

Stack Segment :- holds base address for stack. Also stores interrupt & subroutine return addresses.

Date: M T W T F S S

Lecture:-

1111 1111 FF ← Byte

1111 1111, 1111 1111 FFFF ← Word

1byte 1byte

1111 1111, 1111 1111, 1111 1111, 1111 1111 FFFF FFFF ← DWord

1byte 1byte 1byte 1byte

CF = 1 → (+ve)

AC = 1 Auxilary carry Flag

PF = 0 Parity Flag

ZF = 1 Zero Flag

SF = 0 Sign Flag

OF = 0 → (-ve) Overflow Flag.

FF + 01 = 1 00

Carry Flag (CF)

1111 1111 + 01 = 1 0000 0000

CF ZF = 1

If there is any carry blw 4th and 5th bit then Auxilary carry flag is set to 1.

If CF = 1 then OF = 0, If OF = 1 then CF = 0.

6 status flags:- 1- CF
2- AF
3- PF
4- ZF
5- OF
6- SF

Control Flags:- 1- Trap flag
2- Interrupt enable flag
3- Direction flag

Status flags reflect the outcomes of arithmetic and logical operations performed by CPU.

CF → Carry Flag: set by carry out of MSB.

PF → Parity Flag: set if result has even parity. (if even number of 1's)

ZF → Zero Flag: set if result = 0

SF → Sign Flag: MSB of result

REGISTERS:- high speed storage locations directly inside CPU, can be accessed much more quickly than conventional memory.

(1) Mnemonics/opcode; instruction with no operand ← NOP/STC/CLC

(2) opcode/mnemonics; One operand: Inc Reg/Memory 8/16/32, Dec Reg/Memory 8/16/32

(3) Opcode/Mnemonic destination (R/M) Source (R/M); 2 operands ← MOV / ADD / SUB / MUL / DIV (M → M ✗)

(4) opcode/Mnemonic Destination Source Direction; 3 operands

Date: M T W T F S S

16 bit 8 bit

Mov ax, bl ✗
Add bx, ah ✗
} Not compatible
Assembler will give error

3) Instruction → 2 operands.

R	Immediate
R 8/16/32	R 8/16/32
R 8/16/32	M 8/16/32
M 8/16/32	R 8/16/32
M	M ✗
Destination	Source

R : Register
M : Memory

4) Movsb → to move string byte
Movsd
Movsw

Movsw , Dest2 , Source1 , 0/1 ; 3 operands

Opcode | Destination | Source | Direction

{ Source 1 byte " " ;
Dest2 byte " " ;

LABEL :-

CODE LABEL | DATA LABEL

L1: | array DWORD 1024, 2048
this is data label

code label /loop/jmp

code/~~data~~ segment

L3:

data
→ arr1 byte " " dup(0) , 0 (null)
→ arr2 byte " " dup(0) "

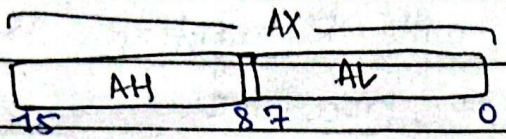
code
mov ecx, length(arr1)
mov si, offset arr1
mov di, offset arr2

L1: mov al, [si]
mov di, al
inc si
inc di
loop L1 } similar to goto statements in C.

Date: M T W T F S S

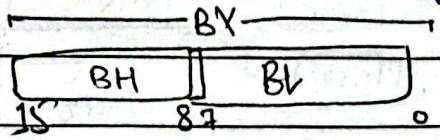
General Purpose Registers :- AX (Accumulator), BX (Base), Counter (CX), DX (Data)

AX [Accumulator Register] → arithmetic, Logic and movement operations.



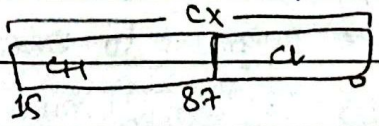
AL (lower byte) 1byte (16 bits)
AH (Higher byte) 1 byte (16 bits)
AX (32 bits) 2 bytes

BX [Base Register] → arithmetic, logic and movement operations.
→ contains address of other memory location.



BL (lower byte) 1byte
BH (Higher byte) 1 byte
BX 2 bytes
EBX 4 bytes

CX [Counter Register] → arithmetic, logic & data movement operations.
→ Used for loop or counter purpose.



CL
CH
CX
ECX

DX [Data Register] → arithmetic, logic & data movement operations.
→ used in Multiplication & division operations.

DL
DH
DX
EDX

Segment Registers :- CS (Code segment), DS (Data segment), SS (Stack segment), ES (Extra segment)

Index Registers :-

Source Index (SI) :- used to store an offset address for source operand.
SI register work with DS (Data segment) DS : SI.

Destination Index (DI) :- used to store an offset address for destination operand
DI register work with ES (Extra segment) ES : DI

Both SI & DI registers are used in string movements instructions.

Pointer Registers :-

Instruction Pointer (IP) :- IP always work with CS register. CS : IP.
Always point to next instruction to be executed.

SS → Stack segment
CS → code segment

Date:
M T W T F S S

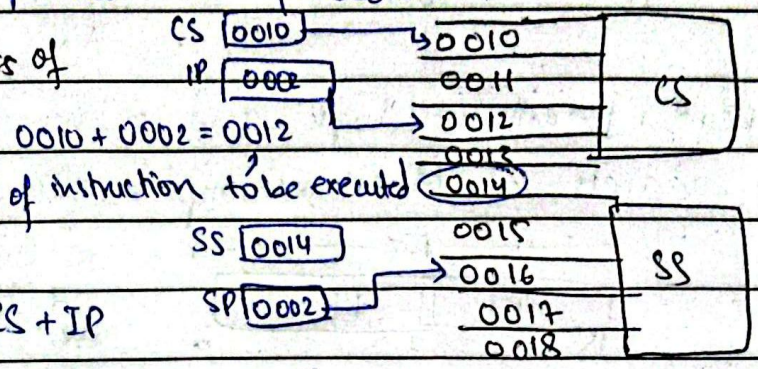
Stack Pointer (SP):- SP register work with SS register SS:SP.
Always points to top item on the stack.

Base Pointer (BP):- BP register work with SS register. SS : BP.
Used to access parameters passed to procedure.

Base address → starting address of segment.

Offset address → distance from the base location.

Address of next instruction = CS + IP



MODES

Real Address mode:- only 1 MB of memory can be addressed, from hexadecimal 00000 to FFFFF. Processor can run only one program at a time, but can momentarily interrupt the program to process requests (called interrupts) from peripherals. MSDOS OS (operating system) runs in real address mode & Windows 95 and 98 can be booted into this mode.

Protected mode:- Processor can run multiple programs at the same time. It assigns each process (running program) a total of 4GB of memory. Each program can be assigned its own reserved memory area, & programs are prevented from using each other's code and data. Ex:- MS-Windows and Linux.

Virtual mode:- computer runs in protected mode and creates a virtual 8086 machine with its own 1-MB address space that simulates an 80x86 computer running in real address mode. for Ex:- Windows NT and 2000.

MOV INSTRUCTION. Lecture:-

Date: M T W T F S S

Mov Reg 8/16, Reg 8/16

Mov al, bl / ax, bx

Mov mem 8/16/32, Reg 8/16/32

Mov reg 8/16, mem 8/16

.data

var1 WORD [?] — when value is undefined/uninitialized.

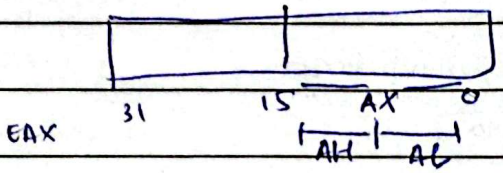
registers stores value and memory.

that value can be address

BYTE 00

WORD 00 00

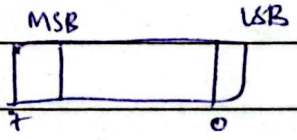
DWORD 0000 0000h



(S)WORD

signed → can save both +-

Either Negative or positive.

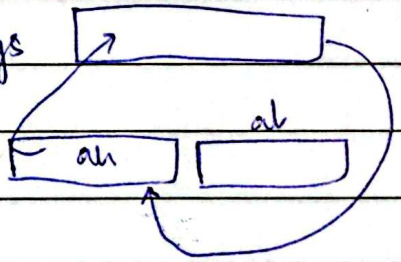


movzx

lahf and sahf → Instruction with no operand.

load save

↳ NOP, CLC, STC, lahf, sahf




```
.data
saveflags BYTE ?
.code
lahf
mov saveflags, ah
```

```
mov ah, saveflags
sahf
```

XCHG → exchange instruction

Instead of 3 instructions just use xchg instruction xchg ax, cx ←

```
mov bx, ax
mov ax, cx
mov cx, bx
```

```
mov ax, 9h
mov cx, 15h
```

direct naam k through access kr rhe.

Date: M T W T F S S

Direct offset:-

arrB byte 5 DUP(0) // arrayB BYTE 10h, 20h, 30h, 40h, 50h

Q. Access through direct offset, indirect offset and index.

mov al, [arrayB] ; AL = 10h mov al, [arrayB] ≡ mov al, [arrayB+0]

al/ah/bl/bh ← any Byte register

mov al, [arrayB+1] ; AL=20h

mov al, [arrayB+2] ; AL=30h

mov al, [arrayB+20] ; AL=?? Garbage value.

2 bytes

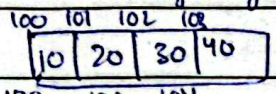
arrayW WORD 100h, 200h, 300h

mov ax, [arrayW] ; AX = 100h

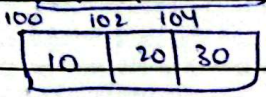
mov ax, [arrayW+2] ; AX = 200h

2 bytes jump kregar.

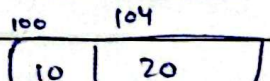
Initially stores offset/ starting address of arrB



arrW



arrD



100 → isk pass address ajaega

mov esi, offset arrB — INDIRECT OFFSET

mov al, [esi+0] AL = 10h

mov al, [esi+1] AL = 20h

mov esi, offset arrW

mov edi, esi

mov ax, [edi] / [edi+0]

inc edi; ~> unity increment only adds one byte so we write edi, 2 for 2 bytes

add edi, 2

NEG :-

comp. does 2's complement at the backend

mov ax, 10h

NEG reg 8/16/32 neg ax AX = -10

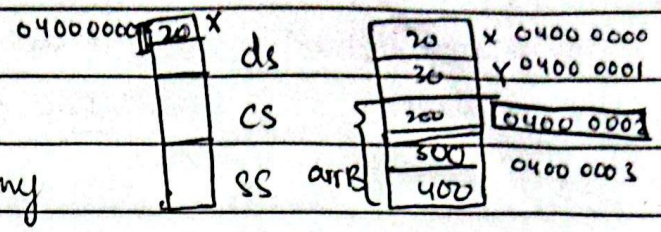
NEG mem 8/16/32 neg ax AX = 10

Date: M T W T F S S

OFFSET :-

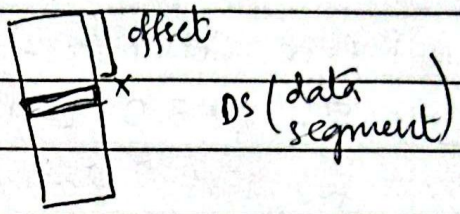
mov esi, offset X ← any variable

mov esi, offset arrB

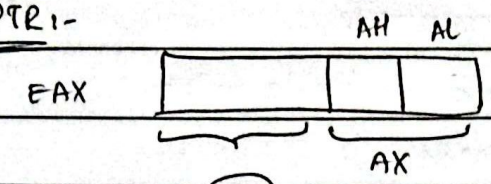


offset → returns starting address of any variable

↳ or distance, in bytes, of the label from beginning of data segment.



POINTER PTR :-



mov ax, WORD PTR myDouble

to access 16 bits / 2 bytes

any 16 bits — starting ki bhi hosakte
middle/end

.data

myDouble DWORD 12345678h

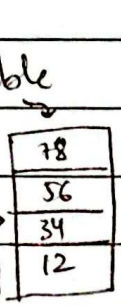
1 2 3 4 — 4th byte

1st byte 2nd byte 3rd byte

mov ax, WORD PTR myDouble

↳ Type Casting

↳ Output : 5678



mov ax, WORD PTR [myDouble+2]

↳ Output: 1234

Example of little endian

little endian — Big endian

depends on architecture

WEEK 03 + WEEK 04

Date: M T W T F S S

Integer constants :-

h → hexadecimal — If no radix is given, it is decimal.

q/o → octal — Ex:- 26 → 26d

d → decimal

b → binary

REAL NUMBER CONSTANTS :- [sign] integer . [integer] [exponent]

Ex:- 2. , +3.0 , -44.2E+05 , 26.ES — E [+/-] integer

Atleast one digit & decimal point are required.

CHARACTER :- single quotes — STRING :- single or double quotes

RESERVED WORDS :- Special meaning in MASM.

Ex:- Mnemonics (ADD, SUB, MUL), Register names (ax, bx), Directive, Attributes (BYTE, WORD), Operators, predefined symbols

Identifier :- programmer-chosen name. It might identify a variable, constant, procedure or code label.

Identifier → variable names.

An identifier cannot be same as assembler reserved word.

first character can be A-Z, a-z, _, @, ?, $.

DIRECTIVES :- commands to the assembler that direct the assembly process.

Ex:- .data , DWORD , .code , .stack

Directives don't execute at runtime, are not case sensitive.

.data → identifies the area of the program containing variables.

.code → " " " " " " containing executable instructions.

.stack → " " " " " " containing runtime stack, setting its size.

→ Different assemblers have different directives. NASM, MASM not same.

INSTRUCTION :- statement that becomes executable when a program is assembled.

Syntax : [label:] mnemonic [operands] [;comment].

LABEL : act as place markers, marks the address (offset) of code and data.

→ Follow identifier rules.

Data label, Code label

Date: M T W T F S S

Data label :- identifier :- count DWORD 100

identifier/name of variable/data label

Code label :- target of jump and loop instructions.

loop ← L1 : (followed by colon

= code

INSTRUCTION FORMAT EXAMPLE :-

① NOP [No Operation Instruction] :- STC (set carry flag)

② One operand :- INC/DEC inc eax , inc myvar

register 8/16/32 memory 8/16/32

③ Two Operand :- Ex: ADD ebx , ecx

reg , mem / mem , reg
reg , reg

Ex :- SUB myvar , 25

mem , constant
reg , constant expression

ADD eax , 25 * 36

mem , mem X

Destination operand Source operand

PROGRAM :-

,TITLE, Add and Subtract

Title directive marks entire line as a comment.

; This is my program All text to right of ; is ignored by assembler / used for comments

INCLUDE Irvine32.inc INCLUDE directive copies necessary definition

.code

main PROC PROC directive identifies the beginning of procedure.

name of procedure

MOV / ADD } instructions

calls a procedure that displays current value of CPU registers.

call DumpRegs ; displays registers.

exit - macro command defined in Irvine32.inc include file that provides a simple way to end program.

main ENDP → ENDP directive marks the end of main procedure.

END main END directive marks the last line of program to be assembled.

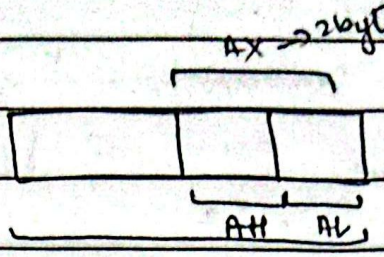
INTRINSIC DATA TYPES :- 8/16/32 . Assembler is not case sensitive.

BYTE, SBYTE, WORD, SWORD, DWORD, SDWORD

dword , Dword , DWORD → same no issue !!!

Date: M T W T F S S

OVERLAPPING VALUES :-



EAX → 16 bit

```
.data
oneByte   BYTE  78h
oneWord   WORD  1234h
oneDword  DWORD 12345678h
.code
    mov eax,0          EAX = 00000000h
    mov al, oneByte    EAX = 00000078h
    mov ax, oneWord    EAX = 00001234h
    mov eax, oneDword  EAX = 12345678h
    mov ax,0           EAX = 12340000h
```

COPYING SMALLER VALUES TO LARGER ONES :- mov ax, al ✗ invalid

```
.data                  → 2bytes → decimal
count WORD 1
.code
    mov ecx,0          00000000
    mov cx, count      0000 0001
```

what happens if we try this approach with signed integer -16?

```
.data
  signedval SWORD -16 ⟶  FFF0h (-16)
.code
    mov ecx,0
    mov cx, signedval   ECX = 0000FFF0h  → but in decimal this would be +65,520.
```

✗

✓ ECX = FFFF FFF0h (-16)

MOVZX :- move with zero extend. only used with unsigned integers.

```
.data
  byteval  BYTE 10001111b
.code
  movzx  ax, byteval
         2bytes
```

Source: 1 0 0 0 1 1 1 1 b

0

Destination: 0 0 0 0 0 0 0 0 1 0 0 0 1 1 1 1

MOVSX :- move with sign extend. only used with signed integers.

```
EX:-  mov bx, 0A69h      BX = 0A69
      movsx eax, bx      EAX = FFFF0A69
      movsx edx, bl      EDX = FFFFFF69
      movsx cx, bl       CX = FF69
```

Source: 1 0 0 0 1 1 1 1

Destination: 1 1 1 1 1 1 1 1 1 0 0 0 1 1 1 1

LAHF and SAHF :- Load status flag into AH (LAHF), store AH into status flag (SAHF).

Date: M T W T F S S

The following flags are copied: Sign, Zero, Auxilary carry, parity & Carry.

```
.data
saveflags BYTE ?
.code
lahf
mov saveflags,ah
mov ah, saveflags
sahf
```

XCHG instruction :- exchanges the content of two operands

```
XCHG reg , reg
     reg , mem
     mem , reg
```

Ex:-
```
xchg ax,bx
xchg ah,al
xchg var1, bx
xchg eax, ebx
```

Q. To exchange two memory operands. val1 , val2 — xchg mem, mem ✗ invalid

↳ combine mov & xchg instruction.

① mov ax, val1
② xchg ax, val2
③ mov val1, ax

val1 = 5, val2 = 10] Assume
ax = 5] mov
val2 = 5]
ax = 10] xchg
val1 = 10] mov
val1 = 10
val2 = 5

DIRECT OFFSET OPERAND :-

```
arrayB BYTE 10h,20h,30h,40h,50h
mov al, [arrayB+1]    ; AL = 20h
mov al, [arrayB+2]    ; AL = 30h
mov al, [arrayB +20]  AL = ??  Range checking → logic bugs.
```

Ex:-
```
.data  arrayW WORD 100h,200h,300h
.code  mov ax, arrayW      ; AX = 100h
       mov ax, [arrayW +2] ; AX = 200h
```

Ex:-
```
.data  arrayD DWORD 1000h,2000h
.code  mov eax, arrayD     ; EAX = 1000h
       mov eax, [arrayD+4] ; EAX, 2000h
```

INC | DEC :-

reg , mem

Date: M T W T F S S

Ex:- .data myword WORD 1000h

.code mov myword ; myword = 1001h

mov bx, myword

dec bx ; bx = 1000h

Overflow, Sign, Zero, Auxilary carry, parity flags are changed according to the value of destination operand.

* INC/DEC instruction donot affect Carry flag.

NEG:- reverses the sign of number by converting the number to 2's complement.

Implement: Neg reg/mem

Implementing Arithmetic Expressions :-

Rval = -Xval + (Yval - Zval)

```
Rval    SDWORD ?        mov eax, Xval
Xval    SDWORD 26       neg eax          ; EAX = -26
Yval    SDWORD 30       mov ebx, Yval
Zval    SDWORD 40       sub ebx, Zval    ; EBX = -10
                        add eax, ebx
                        mov Rval, eax    ; -36
```

OFFSET OPERATOR:- distance in bytes of the label from beginning of data segment.

```
.data   bval  BYTE ?      .code  mov esi, OFFSET bval  ; ESI = 4000
        wval  WORD ?             mov esi, OFFSET wval  ESI = 4001
        dval  DWORD ?            mov esi, OFFSET dval  ESI = 4003
        dval2 DWORD ?            mov esi, OFFSET dval2 ESI = 4007
```

Offset returns the starting address.

.data myarray WORD 1, 2, 3, 4, 5

.code mov esi, OFFSET myarray+4. ESI = 3 -> address of 3rd integer

Date: M T W T F S S

```
.data   bigarray DWORD 500 DUP(?)
        parray DWORD bigarray
```

This means parray points to the bigarray.

```
mov esi, parray        ESI = holds address of beginning of big array.
```

PTR OPERATOR :-

```
.data   mydouble DWORD 12345678h
.code   mov ax, mydouble ; error
```

(ax → 2 bytes, mydouble → 4 bytes)

word PTR makes it possible.

```
mov ax, WORD PTR mydouble          ; 5678h
mov ax, WORD PTR [mydouble+2]      ; 1234h
```

Moving smaller values into larger destinations

```
.data   wordlist WORD 5678h, 1234h
.code   mov eax, DWORD PTR wordlist   ; EAX = 12345678h
```

TYPE OPERATOR :- returns size, in bytes of single element of variable.

```
.data   var1 BYTE ?     → 1
        var2 WORD ?     → 2
        var3 DWORD ?    → 4
```

(Type)

LENGTH OF OPERATOR :- counts the number of elements in an array.

```
.data   byte1  BYTE  10,20,30          → 3
        array1 WORD 30 DUP(?),0,0      30+2
        array2 WORD 5 DUP(3 DUP(?))    5*3
        array3 DWORD 1,2,3,4           4
        str    BYTE "12345678",0       9
```

(Value (lengthof))

SIZEOF OPERATOR :- = length of * time

```
.data   array WORD 32 DUP(0)           32*2
.code   mov eax, SIZEOF array ; EAX = 64
```

ARRAYS :-

```
.data   arrayB BYTE 10h, 20h, 30h
.code   mov esi, OFFSET arrayB
        mov al, [esi]      ; AL = 10h
        inc esi
```

Date: M T W T F S S

```
mov al, [esi]      ; AL = 20h
inc esi
mov al, [esi]      ; AL = 30h
```

WORD ↓ Instead of inc esi do add esi, 2

INDEXED OPERANDS :-

```
.data   arrayW  WORD  1000h, 2000h, 3000h
.code   mov esi, OFFSET arrayW
        mov ax, [esi]
        mov ax, [esi+2]
        mov ax, [esi+4]
```

SCALE FACTORS :- ↳ `mov eax, arrayW[esi*4]`

Lecture :-

* Instead of esi you can use edi or ebx.

PTR OPERATOR :-

```
.data   myDouble  DWORD  12345678h
        varW      WORD   670Ah
mov ax, WORD PTR [myDouble +3]
```

78	0040 0000
56	0040 0001
34	0040 0002
12	0040 0003
0A	
67	

PTR → ↑ memory mapping

will read 2 bytes ↓

0040 0000 + 3 = 0040 0003

↳ output : 0A12h

```
.data
    arrayW  WORD  1000h, 2000h, 3000h
.code  mov esi, OFFSET arrayW
       mov ax, [esi]        AX = 1000h
       add esi, 1
       mov ax, [esi]  → output = 0010
```

00
10
00
20
00
30