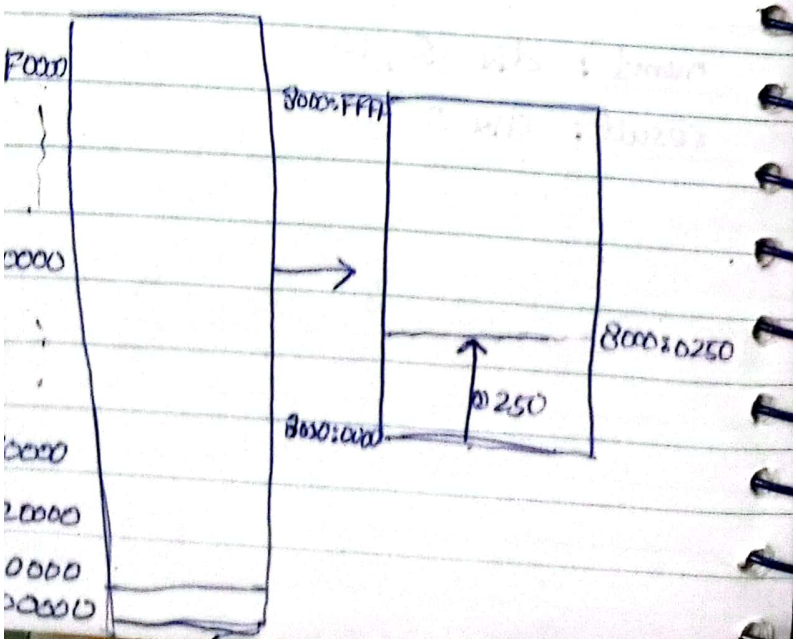
REAL ADDRESS MODE

→ In this mode x86 processor can access 1 MB using 20-bit address in the range 0 to FFFFF.

→ 16-bit register in the Intel 8086 could not hold 20-bit addresses.

→ All of the memory is divided into 64 KB units called segments.

"An analogy is a large building in which segment represents the building's floor. A person can ride the elevator to a particular floor, get off, and begin following the room numbers to locate a room. The offset of a room can be thought of as a distance from the elevator to the room."



DATA TYPES

→ Ek memory block mein kaha jana hai use segment kehte hai or us segment mein kaha jana hai wo offset hai.

Programs in real-address mode, the linear (physical) (absolute) address is 20 bits. Programs cannot use linear address directly, so they are expressed using 2 16-bit integers.

— A 16 bit segment value, placed in one of the segment register
— A 16-bit offset value

→ CPU automatically converts segment: offset value to 20-bit linear address.

TYPES OF Segment Reg.

- CS → contains 16-bit code segment address
- DS → 16-bit data segment address
- SS → " stack " "
- ES → Extra segment.

BYTE	→	8 bit unsigned integer
SBYTE	→	" signed "
WORD	→	16 bit unsigned " [For also be a near or farther in real address mode]
SWORD	→	" signed "
DWORD	→	32 bit unsigned "
SDWORD	→	" signed "
FWORD	→	48 bit integer [For pointer in protected mode]
QWORD	→	64-bit integer
TBYTE	→	80 bit (10 byte) integer
REAL4	→	32-bit (4 byte) IEEE short real
" 8	→	64 " (8 ") " long "
10	→	80 " (10 ") " extended.

OR

DB	→	8-bit integer	can be used for both signed & unsigned
DW	→	16 " "	
DD	→	32 " "	
DQ	→	64 " "	
DT	→	define 80 bit (10 byte) integer	

→ At least one initializer is required in data definition, even if it is zero.

like val1 BYTE X
val1 BYTE 0 ✓
or
val1 BYTE ?

Multiple Initializers:

If multiple initializer are used in the same data definition, it's label refers only to the offset of the first initializer.

offset	value
0000	10
0001	20
0002	30
0003	40

list BYTE 10, 20, 30, 40

0000 0001 0002 0003

→ Not all data definition require label.

list BYTE 10, 20, 30
BYTE 40, 50, 60

→ In the following example, list1 & list2 have same contents.

list1 BYTE 10, 32, 41h, 00100010b
list2 BYTE 0Ah, 20h, 'A', 22h

String → greeting BYTE "Hello", 0
greeti BYTE 'Hi', 0
↓ act as '10' in assembly

⇒ The line continuation character (\) concatenates two source code lines into a single statement. It must be the last character on line.
The following 2 statements are same.

greeting BYTE "Hello"
and
greeting \
BYTE "Hello"

DUP Operator:

→ Used to allocated storage for multiple data items. Useful for allocating space for an ARRAY or STRING, and can be used with initialized or unitialized data:

BYTE 20 DUP (0) ; 20 bytes, all equal to 0
BYTE 20 DUP (?) ; 20 bytes, unitialize
BYTE 4 DUP ("STACK") ; 20 bytes:

"STACKSTACKSTACKSTACK"
5 5 5 5
20 bytes

Array BYTE 3 (1,2,3)

ARRAY OF Words:

mylist Word 1, 2, 3, 4, 5

offset	value
0000	1
0002	2
0004	3
0006	4
0008	5

each word occupies 2 byte so we will increment by 2.

or

array WORD 5 DUP(?)

ARRAY OF DWORD:

myList DWORD 1, 2, 3, 4

0000 0004 0008 000C

increment by 4 bcz each DWORD occupies 4byte

Defining Real Number Data

rVal1	REAL4	-1.2
rVal2	REAL8	3.2E-260
rVal3	REAL10	4.6E+4096
ShortArray	REAL4	20 DUP (0.0)

The DD, DQ & DT directives can also define real number.

Little Endian

x86 processor store & retrive data from memory using little endian order (low to high). The least significant byte is stored at the first memory.

Consider little endian representation of 12345678h.

0000	78
0001	56
0002	34
0003	12

Declaring Unitialized Data:

The .DATA? directive declares unitallized data. When defining a large block of unitialized data, the .DATA? directive reduces the size of a compiled program.

```
.data
small_Array DWORD 10 DUP(0)
.data?
bigArray DWORD 5000 DUP(?)
```

Equal Sign Directive:

name = expression

where name can be any identifier (symbol). Symbols are not variable as they don't reserve storage.

and expression is any integer expression

COUNT = 500.

now this count can be used anywhere. This simply means we can write count instead of 500.

→ array DWORD COUNT DUP(0)

→ COUNT = 5

```
mov al, COUNT
COUNT = 10
mov al, COUNT
COUNT = 6
mov al, COUNT
```

Current Location Counter:

$ is called " " " .

The following declaration declares a variable selfPtr and initializes it with it's own location counter.

```
selfPtr WORD $
```

Calculating size of Array:

```
List BYTE 10,20,30,40
ListSize = ($ - List)
```

ListSize is calculated by subtracting the offset of list from the current location counter.

But ListSize should be exactly after the list

```
List WORD 1000h, 2000h, 3000h, 4000h
Size = ($ - List) / 2
```

↓ bcz each word is 2 byte

EQU Directive:

* name EQU expression

→ In the first format, expression must be a valid integer expression

- number EQU 10+10
- number EQU 11.643

* name EQU symbol

→ In the second format, symbol is an existing symbol name, already defined with = or EQU.

- number2 EQU number

* name EQU <text>

→ In the third format, any text may appear within the brackets <--->

```
pressKey EQU <"Press any key",0>
.
.
.data
prompt BYTE pressKey
```

prompt: variable

⇒ No redefinition is allowed for EQU directive.

CHAPTER #04

OPERAND TYPE:

Following are 3 types of operand.

→ Immediate – uses a numeric literal expression

→ Register – uses a man—

→ Memory

reg8 → 8 bit general purpose register (AH, AL, BH, BL ---)

reg16 → 16 bit general purpose register (AX, BX, CX, DX, SI, DI, BP, SP)

reg32 → 32 bit -- (EAX, EBX, --- ESI, EDI--)

reg → Any general purpose reg

sreg → 16-bit seg registers, CS, DS, SS ES, FS, GS

imm → 8, -16, or 32 bit immediate value

imm8 → 8-bit immediate byte value

imm16 → 16 " " word "

imm32 → 32 " " double " "

MOV Instruction:

rules for MOV:

→ Both operands must be of same size

→ Both operands cannot be memory operands

→ CS, EIP, and IP cannot be destination operand.

→ An immediate value cannot be moved to a segment register.

[mov, var1, var2] X

[mov ax, var1
 mov var2, ax] ✓

Overlapping of Values:

```
.data
oneByte   BYTE   78h
oneWord   WORD   1245h
oneDWord  DWORD  12345678h
.code
mov   eax,0        ;eax = 00000000h
mov   al,oneByte   ;eax = 00000078h
mov   ax,oneWord   ;eax = 00001245h
mov   eax,oneDWord ;eax = 12345678h
mov   ax,0         ;eax = 12340000h
```

LAHF & SAHF Instruction:

The LAHF (Load status flag into AH) instruction copies the low byte of EFlags register into AH. Following flags are copied:
Sign, Zero, Auxilary, Carry, Parity

```
.data
  saveflag BYTE ?
.code
  lahf                    ; load flags into AH
  mov saveflag, ah ; save the ma variable
```

SAHF (store AH into status flag) instruction copies AH into the low Byte of EFlags register.

```
mov ah, saveflag ; load save flag into ah
sahf             ; copy into flag register
```

XCHG Instruction:

The XCHG (exchange data) instruction exchanges the contents of two operands.
There are three variants:

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

The rules are same as MOV instr.
→ it doesnot accept immediate operands.
→ It can be used for sorting array when exchanging the elements of array.

```
xchg ax,bx ; exchange 16-bit reg
xchg ah,al
xchg var1, bx
xchg eax, ebx
```

For using two memory operands.

```
mov ax, var1
xchg ax, val2
mov val1, ax
```

Direct Offset Operand:

arrayB BYTE 10h, 20h, 40h, 60h

```
mov al, [arrayB]     al = 10h
mov al, [arrayB+1]   al = 20h
"  "  , [arrayB+2]   al = 40h
       arrayB+3      al = 60h
```

→ For WORD we use +2, +4, +6 ...
→ For DWORDS " " +4, +8, +12 ...

MOVZX Instruction:

The MOVZX instruction (move with zero-extend) copies the contents of a source operand into a destination operand.

Agr humein 16-bit ki value 32 bit reg mein move krni hai to movzx use krenge q k ye 16-bit k baad k tamam bytes ko '0' assign kr dega.

→ It is used with unsigned integers.

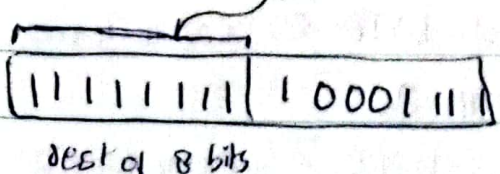
→ The destination must be a register

```
movzx  eax, ax ;
       32   16
```

```
.data
byteVal BYTE 10001111b
.code
   movzx ax, byteVal   AX=0000000010001111
```

MOVSX Instruction:

This instruction fills the upper half of the destination with a copy of source operand's sign bit



agr sign bit '0' hoti to ye zero se fill hojate.

→ isse hoga ye k -ve value negative hi store hogi agr Movzx se store krte to zeros lag jate bawajood isse ghalat value store hoti.

→ destination must be a register.

" ADDITION & SUBRACTION "

INC & DEC Instruction:

INC & DEC add 1 & sub 1 respectively from a single operand.

```
.data
   myWord WORD 1000h
.code
   inc MyWord
   mov bx, myWord ; myWord= 1001 h
   dec bx         ; BX = 1000h
```

ADD & SUB Instruction.

```
.data
   var1 DWORD 10000h
   var2 DWORD 20000h
.code
   mov eax, var1 ; EAX = 10000h
   add eax, var2 ; EAX = 30000h
   sub eax, var1 ; EAX = 20000h
```

Overflow, Sing, Zero, Auxiliary Carry, Parity flags are changed acc. to the value of destination operand.

→ The INC & DEC instruction donot affect the Carry flag.

NEG Instruction:

This instruction reverse the sign of a number by converting the number to it's 2's complement. The following operands are permitted.

NEG reg
NEG mem

OFFSET OPERATOR:

It returns the offset of a data label. The offset represents distance in bytes, of the label from the begining of data segment

```
.data
  bval  BYTE ?
  wVal  WORD ?
  dVal  DWORD ?
.code
  mov eax, OFFSET bval  ;eax = 00404000
  "   "  ,  "   wVal   ;eax = 00404001
  "   "  ,  "   dVal   ;eax = 00404003
```

```
.data
  myArray WORD 1,2,3,4,5
.code
  mov eax, OFFSET myArray+4
```

↳ Now eax contains the address of third integer of array.

```
.data
  BigAss DWORD 500 DUP(?)
  pAss   DWORD BigAss
```

Now pAss is pointing to begining of BigAss. (Creating a pointer)

```
  mov eax, pAss
```

Now loading the value into a register.

ALIGN OPERATOR:

This directive aligns a variable on a byte, word, double word, or paragraph.

```
  ALIGN bound
```

Bound can be 1, 2, 4 or 16.

It is used to set the address of variable in even number as it is easy for CPU to process.

```
  bVal  BYTE ?   00404000
  ALIGN 2
  wVal  WORD ?   00404002
  ALIGN 2
  bVal2 BYTE ?   00404004
```

ALIGN 4
dVal DWORD 00404008
~~ALIGN 4~~
~~sVal BYTE 00404008~~

Note that dVal would have offset 00404005 but ALIGN 4 bumps it to 08.

PTR OPERATOR:

Use to override the declared size of an operand.

If we want to move the lower 16-bits of a DOUBLE word into AX then there would be an error.

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

But WORD PTR makes it possible to move lower order word (5678h) into AX

```
mov ax, WORD PTR myDouble
```

1234 isliye move nhi hoa q ke x86 little endian follow krta hai.

Little Endian: Least Significant Byte stored 1st in the memory and Most Significant stored at last in the memory

Suppose we want to store 12345678h in a memory having starting address 1000h.

1000	78h
1001	56h
1002	34h
1003	12h

In Stack Lower Bytes holds Lower data & Higher Bytes hold Higher data.

⇒ Moving Smaller Values Into Larger Destination:

If we want to move two small values into a large destination

```
.data
  wordlist WORD 5678h, 1234h
.code
  move eax,DWOR PTR wordlist
  ;eax = 1234 5678h
         ah al
```

TYPE OPERATOR:

This operator returns size in bytes of a single element of a variable

eg:

TYPE of byte = 1

" " word = 2

" " doubleword = 4

LENGTHOF	byte1	3
"	array1	30+2
"	array2	5*3
"	array3	4
"	digitStr	9 (last one is zero bcz of null count)

LENGTHOF OPERATOR:

This operator counts the no. of elements in an array, defined by the values appearing on the same list on the same line as it's label.

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

When nested DUP operators are used in an array decleration LENGTHOF returns product of two counters.

⇒

```
MyArray BYTE  10,20,30,40,50
        BYTE  60,70,80,90,1
```

LENGTHOF MyArray = 5 (as lines are split)

⇒

```
MyArray BYTE  10,20,30,
              40,50,60
```

Now cnt will be 6.

SIZEOF OPERATOR

This returns a value that is equivalent to multiplying LENGTHOF by TYPE.

```
.data
intArr WORD 32 DUP(?)
.code
mov eax, SIZEOF intArr ; eax = 64
```

as 32 x 2 = 64

length, Byte size

LABEL DIRECTIVE :

this directive lets you insert a label and give it a size attribute without allocating any storage. Common use of LABEL is to provide alternative name and size attribute to the variable declared next to it in the ~~code~~ data segment.

```
.data
val16  LABEL WORD
val32  DWORD 12345678h
.code
mov ax, val16          ; ax = 5678h
mov dx, [val16+2]      , dx = 1234
```

val32 ke liye ek doosra naam val16 rakh diya hai val16 & val32 dono same address contain krte hai.

```
inc [esi]              ; error
```

[as assembler doesn't know to which whether esi points to byte word, or doubleword]

```
inc BYTE PTR [esi]     ; No error
```

now PTR confirms the operand size

INDIRECT ADDRESSING

We use register as a pointer (called indirect addressing) and manipulate register's value.

indirect operand → operand using indirect address

INDIRECT OPERANDS

Protected Mode: In this mode, an indirect operand can be any 32-bit general purpose register (EAX, EBX, ECX, EDX, ESI, EDI, EBP, & ESP) surrounded by brackets.

```
.data
byteVal BYTE 10h
.code
mov esi, OFFSET byteVal
mov al, [esi]          , AL = 10h
mov [esi], 20h,        now at the address which ESI contain 20h is stored.
```

Real-Address Mode: A 16-bit register hold the offset variable (SI, DI, BX, or BP)

Avoid BP unless you are using it to index into the Stack.

Adding 32-bit Integer Using Arry.

```
.data
arr DWORD 10000h, 2000h, 3000h
.code
mov esi, OFFSET arr
mov eax, [esi]
add esi, 4
add eax, [esi]
add esi, 4
add eax, [esi]
```

Offset	Value	
10200	10000	[esi]
10204	20000	[esi] + 4
10208	30000	[esi] + 8

Scale Factors In Indexed Oper.

```
.data
arrayD DWORD 100h, 200h, 300h, 400h
```

(0 1 2 3)

To get the element 400h we would multiply subscript (3) by 4 (size of double word)

```
.code
mov esi, 3 * TYPE arrayD
mov eax, arrayD[esi]
```

```
.data
Arr DWORD 1,2,3,4
.code
mov esi, 3
mov eax, Arr[esi * 4]  or [esi * TYPE arr]
                ; eax = 4
```

"POINTERS"

There are two types of pointer NEAR & FAR Pointer. Their sizes are affected by the processors current mode (16-bit real or 32-bit protected).

	16-Bit Mode	32-Bit Mode
NEAR Pointer	16-bit offset from the begining of data segment	32-bit offset from the begining of data segment
FAR Pointer	32-bit segment-offset address	48-bit segment-offset address

Foundation for Advancement of Science & Technology

78	0000
56	0001
34	0002
12	0003

arrayB BYTE 10h, 20h, 30h
arrayW WORD 1000h, 2000h, 3000h
ptrB DWORD arrayB
ptrW DWORD arrayW

we can also write, both are correct
ptrB DWORD OFFSET arrayB

.data
myDouble DWORD 12345678h
0003 0002 0001 0000
.code
mov al, BYTE PTR myDouble AL = 78h
→ BYTE PTR myDouble ke jo '0' offset par Byte hogi wo store krega. Little endian hi wajase Least Significant Byte 0000 offset par stored hai.

⇒ mov al, BYTE PTR myDouble+1 AL = 56
ab ulta challenge myDouble+1 matlab 0000+1 waali Byte jo hai 56.
→ mov al, BYTE PTR myDouble+2 AL = 34
mov ax, WORD PTR myDouble AX = 5678
→ reverse mein jo phela word hoga wo store hojayega
mov ax, WORD PTR myDouble+1 AX = 1234

.data
myByte BYTE 12h, 34h, 56h, 78h
0001 0002 0003 0004
.code

mov ax, WORD PTR [myByte] AX = 3412
→ array mein 1st element 1st Byte hota hai to pheli Byte uthai AX k lower address mein daal diya phir doosri byte uthai or use higher mein daal diya.

Foundation for Advancement of Science & Technology

(Jump & Loop)

Unconditional Transfers: Control is transfer to a new location in all cases; a new address is loaded into the IP, causing execution to continue at a new address. The JUMP instruction does this

JMP destination

where destination can be any label.

```
00041  top:
         .
         .
       jmp top   ; this will be repeated endlessly as no condition
```

Now on reaching jump the jump will load 00041 in IP so it will move towards 00041.

Conditional Transfers: the program branches if a certain condition is True. The CPU interprets T/F condition based on values of ECX & Flags registers.

LOOP Instruction:

Also known as LOOP acc to ECX Counter, repeats ~~itself~~ a block of statements a specific no. of times. ECX is automatically used as a counter & is decremented each time the loop repeats itself.

LOOP destination

The destination must be within -128 to +127 bytes.

This LOOP instruction follow ~~the~~ steps

→ First decrement ECX by 1

→ Check if ECX = 0

→ If ECX ≠ 0 then jump is taken to the destination (label)

→ else if ECX = 0 loop is terminated & control passes to the next instr. following loop.

```
        mov  ax, 0
        mov  ecx, 5
L1:
        inc  ax      // runs 5 times
        loop L1

; ax = 5
; ecx = 0
```

→ If we want to modify ecx inside a loop we should then store it and then restore it.

```
.data
  ctr DWORD ?
.code
    mov ecx,100     // setting ecx=100
  top:
      mov ctr,ecx   //storing ecx to ctr
      mov ecx,50    //modifying ecx.
      .
      .
      mov ecx,ctr   //restoring ecx
      loop top
```

Nested loops:

```
.data
  ctr DWORD ?
.code
    mov ecx,10      ; ecx for outerloop
  outerloop:
      mov ctr,ecx   ; saving value
      mov ecx,5     ; ecx for inner loop
  innerloop:
        .
        .
      loop innerloop
      mov ecx,ctr   ; ecx = outervalue
    loop outerloop
```

Summing an integer array:

```
.data
arr  DWORD  1000h,2000h,3000h,4000h
.code
    mov  eax,0
    mov  ebx, OFFSET arr
    mov  esi,0
    mov  ecx, LENGTHOF arr
  top:
      add  eax, arr[esi* TYPE arr]
      inc  esi
      loop top
    exit
main ENDP
END main
```

Copying String:

```
.data
source BYTE "Hello World",0
target BYTE SIZEOF source DUP(0)
.code
    mov esi,0
    mov ecx, $-SIZEOF source
```

```
top:
    mov  al, source[esi]
    mov  target[esi], al
    inc  esi
    loop top
exit
main ENDP
END main
```