CHP #09

"STRINGS & ARRAYS"

→ These instruction are not limited to character array only.

MOVSB, MOVSW, MOVSD:
Move String data: Copy data from memory addressed by ESI to memory addressed by EDI.

CMPSB, CMPSW, CMPSD:
Compare string: Compare the contents of two memory locations addressed by ESI & EDI

SCASB, SCASW, SCASD:
Scan String: Compare the accumulator (AL, AX, or EAX) to the contents of memory addressed by EDI.

STOSB, STOSW, STOSD:
Store String data: Store the accumulator content into memory addressed by EDI.

LODSB, LODSW, LODSD:
Load Accumulator from String: Load memory addressed by ESI into the accumulator.

REPEAT PREFIX :

Repeat prefix, uses ECX as a counter.

REP → Repeat while ECX > 0

REPZ, REPE → Repeat while ZF = 1 & ECX > 0

REPNZ, REPNE → Repeat while ZF = 0 & ECX > 0

DIRECTION FLAG :

Value of DL	Effect on ESI & EDI	Address sequence
Clear	Incremented	Low - high
Set	Decremented	High - low

- cld → clear direction flag
- std → set "

MOVSB, MOVSW, MOVSD :

MOVSB → Copy Bytes

MOVSW → 4 words

MOVSD → 4 doublewords

Copy String :

```
cld                     ;clr direction flag
mov esi, OFFSET string1
mov edi, OFFSET string2
mov ecx, 10
rep movsb               ;mov 10 bytes
```

- moves 10 bytes from string2 to string2
- ESI & EDI are automatically incremented by directional flag when 'movsb' repeats itself.

```
.data
source DWORD 20 DUP(0FFFFFFFFh)
target DWORD 20 DUP(?)
.code
cld
mov ecx, LENGTH of source
mov esi, OFFSET source
mov edi, " target
rep movsd               ;copy doublewords
```

● CMPSB, CMPSW, CMPSD:

```
.data
    source DWORD 1234h
    target  4    5678h
.code
    move  esi, 0 — sc
    mov   edi, 0 — target
    cmpsd              ; comp- doublewords
    ja    L1           ; jump if source > tar
```

→ To compare multiple DWORDs

```
mov   esi, OFF- source
mov   edi,  4  target
cld
mov   ecx, LENGTHOF source

repe cmpsd        ; repeat while
                    equal.
```

↓

repeats until ECX=0 or a pair of DWORDS is found to be different.

● SCASB, SCASW, SCASD:

→ useful when looking for a single value in a string or array.

Scan For Matching Character:

- Searching in string 'ALPHA', looking for letter 'F'. If letter is found EDI points one position beyond matching character. If letter not found, JNZ exit.

```
.data
    ALPHA BYTE "ABCDEFGH", 0
.code
    mov edi, OFFSET alpha
    mov al, 'F'
    mov ecx, LENGTHOF alpha

    cld
    repne scasb   ; repeat while not eq
    jnz  quit     ; quit if letter not found
    dec  edi      ; agr found then
                    decr edi.
```

→ jesi letter found hoga to ZF=1 hojayega or JNZ execute nhi hoga.

→ agr letter found nhi hota to ZF=0 rhega.

Foundation for Advancement of Science & Technology

"Selected String Procedures"

① STOSB, STOSW, STOSD :

Useful when filling all values of array by a style value.

```
.data
    Count = 100
    string BYTE Count DUP (?)

.code
    mov al, 0FFh
    mov edi, OFFSET string
    mov ecx, Count
    cld
    rep stosb        ; fill with contents of AL
```

② LODSB, LODSW, LODSD :

See example in pg # 337.

• Str_compare :

```
INVOKE Str_compare, ADDR string1,
                    ADDR string2
```

→ Comparasion is case sensitive.

• Flags affected by the procedure.

Relation	CF	ZF	Branch if True
S1 < S2	1	0	JB (jump Below)
S1 = S2	0	1	JE (if eq)
S1 > S2	0	0	JA (if above)

• Str_length :

→ Returns length of string in EAX.

```
INVOKE Str_length ADDR string
```

• Str_copy :

→ copy from a null terminated string from src to target.

→ Target must be large enough to hold source.

```
INVOKE Str_copy, ADDR str1, ADDR str2
```

- **Str_trim :**

→ removes all occurrences of a selected trailing character from a NULL terminated string

INVOKE Str_trim, ADDR string, char_to_trim

- **Str_ucase :**

→ converts a string to all upper case. Returns no value.

INVOKE Str_ucase, ADDR string

— 2-D ARRAYS —

Two types of Representation :

↳ Row Major

10	20	30	40
50	60	70	80
90	A0	B0	C0
D0	E0	F0	

10	20	30	40	50	60	70	80	90	. .	F0

1st row 2nd row

↳ Col. Major

10	50	90	D0	20	60	A0	F0	30	. . .

1st col 2nd col.

Base-Indexed Operands :

→ A base-index operand adds the value of two reg (called base & index), producing an offset address.

→ In 32-bit mode any general purpose reg can be used as base & index

- data

```
arr DWORD 1000h, 2000h, 3000h.
```

- code

```
mov ebx, OFFSET arr
mov esi, 2
mov ax, [ebx+esi]      ; AX = 2000h

mov edi, OFFSET arr
mov ecx, 4
mov ax, [edi+ecx]      ; AX = 3000h
```

2-D Array :

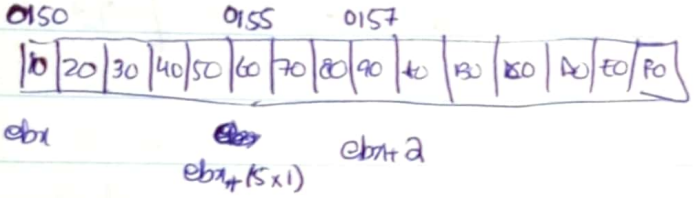
→ When accessing 2-D arr in row major, the ROW offset is held in BASE reg & COL offset in the INDEX reg.

```
tableB BYTE 10h, 20h, 30h, 40h, 50h
RowSize = ($ - tableB)
       BYTE 60h, 70h, 80h, 40h, 40h
       BYTE CB0h, CC0h, DA0h, DE0h, EF0h
```

Foundation for Advancement of Science & Technology

Let we want to find val at (1, 2) i.e 80h.

row_index = 1
col_index = 2
mov ebx, OFFSET arr
add ebx, RowSize * row_index
mov esi, col_index
mov al, [ebx + esi * TYPE arr]



Base-Index-Displacement Oper:

[base + index + displ.]
displ.[base + index * TYPE arr]

→ Displacement can be the name of a variable or a constant expression.

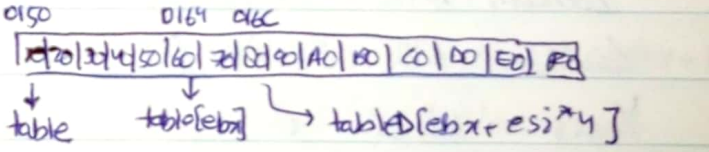
tableD DWORD 10h, 20h, 30h, 40h, 50h
RowSize = ($ - tableD)
DWORD 60h, 70h, 80h, 90h, A0h
DWORD B0h, C0h, D0h, E0h, F0h

Calculating Row Sum:

RowSum PROC USES ebx ecx edx esi
; Recieves EBX = table offset, EAX = row index
; ECX = rowsize in bytes
; Ret EAX return sum.

mul ecx ; rowindex * rowsize
add ebx, eax ; row offset
mov eax, 0
mov esi, 0
L1: movzx edx, BYTE PTR[ebx+esi]
add eax, edx
inc esi ; next byte in row
Loop L1
ret

mov ebx, RowSize
mov esi, 2 ; col_index
mov eax, tableD[ebx + esi * TYPE tableD]



RowSize = 0014 h = 20

Structures & Macros
Chapter # 10

→ The variables in structure are called fields

Name STRUCT
    field declaration
Name ENDS

• Field Initializers :
↳ Undefined : The ? operator leaves the field contents undefined
↳ String Literals : Character strings enclosed in ""
↳ Integers
↳ Arrays : The DUP operator

Employee STRUCT
    Idnum BYTE "00000000"
    LastName BYTE 30 DUP(0)
    Years WORD 0
    SalaryHistory DWORD 0,0,0,0
Employee ENDS

"00000000"	(NULL)	0	0	0	0	0
Idnum	LastName	Yr	Salary History			

• Declaring Structure Variable :

identifier structuretype <initializer-list>

→ identifiers follows the same rules as other variable.

→ The empty <> brackets cause the structure to obtain default field values.

worker Employee <> ; default values
worker2 Employee <"555223333">
↳ only numId is set (#3)
↳ can also be used
worker3 Employee <,"Samraze">
↳ Idnum is skipped
worker4 Employee <,,,2 DUP(2000)>
↳ First 2 values initialize with 2000 & rest of 2 with zeros.

Foundation for Advancement of Science & Technology

• Array Of Structure

```
NumPoints = 3
AllPoints COORD NumPoints DUP (<0,0>)
```

→ TYPE Employee : 60
→ SIZEOF Employee : 60
→ SIZEOF Worker : 60

→ TYPE Employee.SalaryHistory : 4
→ LENGTHOF Emp " : 4
→ SIZEOF " : 16
→ TYPE Employee.Years : 2

Refrencing To Members:

```
.data
    worker Employee <>
.code
    mov dx, worker.Years
    mov worker.SalaryHistory, 2000 ;1st salary
    mov [worker.SalaryHistory+4], 3000 ;2nd sal
```

Use the Offset Operator:

```
mov edx, OFFSET workes.LastName
```

Indirect & Indexed Operands

```
mov esi, OFFSET worker
mov ax, (Employee PTR [esi]).Years
```

```
mov ax, [esi].Years X
```

bcz humne years ko ye nhi btaya k wo kis structure k liye defined h.

Indexed Operands:

```
.data
    department Employee 5 DUP(<>)
.code
    mov esi, TYPE Employee ;index=1
    mov department[esi].Years, 4
```

1200 1250
1230
18 30

Looping through an Array:

- data

```
AllPoints COORD 3 DUP(<0,0>)
```

- code

```
mov edi,0
mov ecx, 3
mov ax,1

L1:
mov (COORD PTR AllPoints[edi]).X ,ax
mov (COORD PTR AllPoints[edi]).Y , ax
add , TYPE COORD
inc ax
Loop L1
endif
```

- NESTED STRUCTURES

```
Rectangle Struct
    UpperLeft COORD <>
    LowerRight COORD <>
Rectangle ENDS

rect1 Rectangle <>
rect2  "    {}
rect3  "    {{10,10},{50,50}}
rect4  "    <<10,4>,<50,7>>
```

direct reference to structure variable

```
mov rect1.UpperLeft.X , 10
```

Indirect reference

```
mov esi, OFFSET rect1
mov (Rectangle PTR[esi]).UpperLeft.Y,10
```

The OFFSET operator can return pointers to individual structure fields

```
→ mov edi, OFFSET rect2.LowerRight
mov (COORD PTR [edi]).X, 50
→ mov edi, OFFSET rect2.LowerRight.X
mov WORD PTR [edi], 50
```

— Unions —

→ All the fields in the union start at the same OFFSET.

→ The storage size of union is eq. to the length of it's longest field.

```
Unionname UNION
    union-field
unionname ENDS
```

If it's inside a structure

```
struct_name STRUCT
    structure_fields
    UNION unionname
        unionfields
    ENDS
struct_name ENDS
```

→ Each field in a union can have a single initializer.

→ Initializer if used must have consistent values.

```
Integer Union
    D DWORD 1
    W WORD 5
    B BYTE 8
Integer ENDS
```

- data

myInt Integer <>

Now the default values of myInt.D, myInt.W, myInt.B will all set to 1. Because the largest datatype has 1 value.

→ Structure Containing Union:

```
FileInfo STRUCT
    FileID Integer <>
    FileName BYTE 64 DUP (?)
FileInfo ENDS
```

or

```
FileInfo STRUCT
    Union FileID
        D DWORD ?
        W WORD ?
        B BYTE ?
    ENDS
    FileName ____ DUP (?)
FileInfo ENDS
```

→ Declaring & Using Union Variables

```
val1 Integer <12345678h>
val2 Integer <100h>
val3 Integer <>
```

To use a union variable, you must supply the name of one of the variant fields.

→

Foundation for Advancement of Science & Technology

```
mov  val3.B , al
mov  val3.W , ax
mov  val3.D , eax
```