CSE331L _2 - Variables, I/O, Array

Topics to be covered in this class:

- Creating Variables
- Creating Arrays.
- Creating constants
- Introduction to INC, DEC, LEA. instructions.
- Learn how to access memory.

Creating variables:

Syntax for a variable declaration:

```
name DB value
name DW value
```

DB - stands for Define Byte.

DW - stands for Define Word.

- name - can be any leter or digit combination. though it should start with a leter. It's possible to declare unnamed variables by not specifying the name (this variable will have an address but no name).

- Value - can be any numeric value in any supported numbering system (hexadecimal, binary or decimal). or "?" symbol for variables that are not initialized.

Creating constants:

Constants are just like variables, but they exists only until your program is compiled (assembled). After definition of a constants it's value cannot be changed. to define constants EQU dérective is ~~s~~used:

name EQU <any expression> .

for example: K EQU 5

MOV AX, k .

Creating Arrays:

Arrays canbe seen as chains of variables. A text string is an example of abyte array. each character is represent as an ASCII code value.(0 - 255).

Here are some array définition example:

a DB 48h, 65h, 6Ch, 6Ch, 6Fh, 00h .

b DB 'Hello', 0 .

- You can access the value of any element in array using square brackets, for example:

MOV AL, a[3] .

- You can also use any of the memory

index registers BX, SI, DI, BP, for example:

```
MOV SI, 3
MOV AL, a[SI].
```

- If you need to declare a large array you can use DUP operator.

The syntax for DUP:

number DUP (values(s))

number - number of duplicates to make (any constant value).

value - expression that DUP will duplicate.

for example: c DB 5 DUP(9)
is an alternative way of declaring - c DB 9,9,9,9,9.
One more example:
d DB 5 DUP(1,2) .
is an alternative way of declaring - d DB 1,2,1,2,1,2,1,2,1,2

Memory access

To access memory, we can use these 4 registers: BX, SI, DI, BP. combining these 4 registers inside [ ] symbols, we can get different memory locations.

[BX+SI] [BX+DI] [BP+SI] [BP+DI]	[SI] [DI] d16 (Varriable offset only) [BX]	[BX+SI+d8] [BX+DI+d8] [BP+SI+d8] [BP+DI+d8]
[SI+d8] [DI+d8] [BP+d8] [BX+d8]	[BX+SI+d16] [BX+DI+d16] [BP+SI+d16] [BP+DI+d16]	[SI+d16] [DI+d16] [BP+d16] [BX+d16]

- Displacement canbe an immediate value or offset of a varriable, or even both. If there are several values, assembler evaluates all values and calculate a singe immidiate value.
- Displacement canbe inside or outside of the [ ] symbols, assembler generates the same machine code for both ways.
- Displacement is a signed value, so it can be both positive and negetive.

Instructions:

Instruction	Operands	Description
INC	REG MEM	Increment. Algorithm: operand = operand + 1 Example: MOV AL, 4 INC AL; AL=5 RET
DEC	REG MEM	DECrement. Algorithm: Operand = operand - 1 Example: MOV AL, 86 DEC AL ; AL=85 RET
LEA	REG, MEM	Load effective address Algorithm: REG = address of memory (offset) Example: Mov BX, 35h MOV DI, 12h. LEA SI, [BX+DI]

Declaring Array: Array Name DB Size DUP(?).

Value initialize: arr1 DB 50 DUP(5, 10, 12).

Index values: mov bx, offset arro.

```
mov [bx], 6 ;inc bx
mov [bx+1], 10
mov [bx+9], 9 .
```

OFFSET:

"Offset" is an assembler directive in x86 assemb language. It actually means "address" and is a way of handling the overloading the "mov" instruction.

Allow me to illustrate the usage-

1. mov si, offset varible

2. mov si, variable.

The first line load SI with the address of variable. The second line load the value stored at the address of variable.

As a matter of style, when I wrote x86 assembler I would write it this way -

1. mov si, offset variable.

2. mov si, [variable].

The square bracket aren't ~~n~~necessary but they made it much clearer when loading the content rather than the address.

LEA is an instruction that load the "offset Variables" when adjusting the address between 16 and 32 bits as necessary. "LEA (16 bit registers) (32 bits registers)" loads the lower 16 bits of the addresses into the registers, and "LEA (32 bit registers), (16 bit registers)" loads the 32-bit address zero extended to 32 bits