CSE331L_3 – Variables, I/O, Array

Topics to be covered in this class:

- Creating Variables
- Creating Arrays
- Create Constants
- Introduction to INC, DEC, LEA instruction
- Learn how to access Memory.

Creating Variable:

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 letter or digit combination, though it should start with a letter. 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 exist only until your program is compiled (assembled). After definition of a constant its value cannot be changed. To define constants EQU directive is used:

name EQU < any expression >

For example:

```
K EQU 5
MOV AX, k
```

Creating Arrays

Arrays can be seen as chains of variables. A text string is an example of a byte array, each character is presented as an ASCII code value (0-255).

Here are some array definition examples:

```
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 (value(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 four registers: BX, SI, DI, BP. Combining these registers inside [ ] symbols, we can get different memory locations.

[BX + SI] [BX + DI] [BP + SI] [BP + DI]	[SI] [DI] d16 (variable 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 can be an immediate value or offset of a variable, or even both. if there are several values, assembler evaluates all values and calculates a single immediate value.
- Displacement can be 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 or negative.

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 arr0
mov [bx], 6 ;inc bx
mov [bx+1],10
mov [bx+9], 9
```

OFFSET:

"Offset" is an assembler directive in x86 assembly language. It actually means "address" and is a way of handling the overloading of the "mov" instruction. Allow me to illustrate the usage -

```
1.mov si,offset variable
2.mov si,variable
```

The first line loads SI with the *address* of variable. The second line loads SI with 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 brackets aren't necessary, but they made it much clearer while loading the contents rather than the address.

LEA is an instruction that load "offset variable" while adjusting the address between 16 and 32 bits as necessary. "LEA (16-bit register),(32-bit address)" loads the lower 16 bits of the address into the register, and "LEA (32-bit register),(16-bit address)" loads the 32-bit register with the address zero extended to 32 bits.