CSE 331/EEE 332 (Microprocessor Interfacing & Embedded System Lab)

Lab 02 : Creating variables, constants and arrays; Learn how to access Memory; Instructions: INC, DEC, LEA

Lab Officer : M. A. Muhiminul Islam

Topics to be covered in class today:

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

Differences between com file and exe file.

COM files are text files while EXE files are binary

COM files cannot exceed 64K, so large programs are usually stored in EXE files.

MODEL directive

The traditional memory models recognized by many languages are small, medium, compact, large, and huge.

- Tiny-model programs run only under MS-DOS. Tiny model places all data and code in a single segment. Therefore, the total program file size can occupy no more than 64K.
- Small model supports one data segment and one code segment. All data and code are near by default.
- Medium model supports multiple code and single data segment.
- Compact model supports multiple data segments and a single code segment.
- Large model supports multiple code and multiple data segments. All data and code are far by default.
- Huge model implies that individual data items are larger than a single segment, but the implementation of huge data items must be coded by the programmer.

Memory Model	Size of Code	Size of Data
TINY	Code + Data < 64KB	Code + data < 64KB
SMALL	Less than 64KB	Less than 64KB
MEDIUM	Can be more than 64KB	Less than 64 KB
COMPACT	Less than 64KB	Can be more than 64KB
LARGE	Can be more than 64K	Can be more than 64KB
HUGE	Can be more than 64K	Can be more than 64KB

Difference between Stack 100h and Org 100h.

stack 100h reserves 100h bytes for stack. org 100h sets the current address to 100h, that is the address the assembler is assuming. Note that stack 100h applies to exe files, it's going to be written in a header so the loader provides that much stack for you. org 100h typically applies to com files, because those are loaded at address 100h.

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 duplicate 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

Instruction	Operands	Description
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]

Task 1

Create two arrays of size 5. Load one of the arrays with random numbers of your choice. The second arrays should be kept blank. Copy the contents of the first array into the second array in **reverse** order. You must not use loops to accomplish this task.