CSE 331L / EEE 332L: Microprocessor Interfacing & Embedded System (Lab 3)

Section: 7 & 8, Fall 2019

SIngle-key Input:

```
ORG 100H

MOV AH, 1
INT 21H
MOV BL, AL

RET
```

Multiple-key Input:

```
ORG 100H

MOV AH, 1

INT 21H          ;1st input
MOV BL, AL

INT 21H          ;2nd input
MOV BL, AL

INT 21H          ;3rd input
MOV BL, AL
```

Single-key Output:

```
ORG 100H

MOV AH, 2
MOV DL, BL          ;output the value loaded in register BL
INT 21H

RET
```

Multiple-key Output:

```
ORG 100H

MOV AH, 2

MOV DL, BL          ;output the value loaded in register BL
INT 21H

MOV DL, 35H         ;output the ascii value of 35H that is 5
INT 21H

MOV DL, 20H         ;prints a space (ascii code of 20H is a space)
INT 21H

RET
```

Example: Take 2 numbers as input, add them and show the answer.

```
ORG 100H
MOV AH, 1

INT 21H
MOV BL, AL      ;1st input

INT 21H
MOV CL, AL      ;2nd input

ADD BL, CL      ;add the two values (summation in hex value)
SUB BL, 30H     ;subtract 30h from the sum to get the ascii of the required result

MOV AH, 2

MOV DL, OAH
INT 21H

MOV DL, ODH
INT 21H

MOV DL, BL
INT 21H

RET
```

Program Structure

- **Code Segment:** holds the instructions of the program, instructions are organized as procedures.

  Procedure is a part of code that can be called from your program in order to make some specific task. Procedures make program more structural and easier to understand. Generally procedure returns to the same point from where it was called.

The syntax for procedure declaration:

```
name PROC
    ; here goes the code
    ; of the procedure ...
  RET
  name ENDP
```

name is the procedure name, the same name should be in the top and bottom, this is used to check the correct closing of procedures.

You already know that RET instruction is used to return the control to the operating system. The same instruction is used to return from procedure (actually operating system sees your program as a special procedure).

PROC and ENDP are compiler directives, so they are not assembled into any real machine code. Compiler just remembers the address of procedure.

CALL instruction is used to call a procedure.

Example:

```
ORG   100H

MOV   AL, 1
CALL  m2       ; m2 is the name of the procedure
RET            ; return to operating system.

m2    PROC     ; start of procedure m2
   ADD  Al, BL
   RET             ; return to caller.
m2    ENDP       ; closing of a procedure
```

MAIN procedure:

```
ORG 100H

MAIN PROC
        ;BODY

        MOV AH, 4CH        ;terminate function number
        INT 21H            ;executes the function number 4ch
MAIN ENDP

;other procedures

END MAIN
```

- **Data Segment:** contains all the variable definitions
- **Stack Segment:** contains a block of memory to store the stack, needs enough space to store.

Memory Models: SMALL, MEDIUM, COMPACT, LARGE, HUGE (Determines the size of code and data of the program)

Program Structure	**Example**
.MODEL SMALL .STACK 100H .DATA ;variables and constants .CODE MAIN PROC ;instructions of main procedure MAIN ENDP ; other procedures END MAIN	.MODEL SMALL .STACK 100H .DATA VAR1 DB 5EH ;variable VAR2 DW 8DC6h k EQU 5 ;constant .CODE MAIN PROC MOV AX, 6

	CALL SUM ;procedure call
	
	MOV AH, 4CH
	INT 21H
	MAIN ENDP
	
	SUM PROC
	ADD AX, 2 ;return to caller
	RET
	SUM ENDP
	
	END MAIN

Variables:

- Declaration:

  Variable _Name DB/DW initial_value

  - DB = Define Byte
    Example:
    VAR1 DB 5EH

  - DW = Define Word
    Example:
    VAR2 DW 8DC6h

  - DB/DW are variable types

- Variable _Name:
  Can be any letter or digit combination, though it **must start with a letter.**
  It's possible to declare unnamed variables by not specifying the name using “?” (this variable will have an address but no name)

- Initial_value:
  can be any numeric value:
    - Decimal number ends with an optional “D”/”d”
    - Binary number ends with “B”/”b”
    - Hex number ends with “H”/”h” and must start with a decimal digit. Otherwise assembler would be unable to decide whether the data represents a number or a string. (Ex: 2AH, 5C4H, 1ABCH, 0ABCDH)
    - Numbers may have signs
    - “?” denotes an uninitialized byte/word

Creating Constants:

Constants are just like variables, but they exist only until your program is compiled (assembled) because no memory is allocated for EQA names. After definition of a constant its value cannot be changed. To define constants EQU (equates) directive is used:

constant_name EQU constant_value (numeric value / string)

Example:

```
k EQU 5

MOV BX, 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).
  Example:

  ```
  a DB 48h, 65h, 6Ch, 6Ch, 6Fh, 00h
  b DB 'Hello', 0
  ```

- You can access the value of any element in an array using square brackets
  Example:

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

- You can also use any of the memory index registers BX, SI, DI, BP
  Example:

  MOV SI, 3
  MOV CL, 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.

  Example:

  c DB 4 DUP(9) ;is an alternative way of declaring: c DB 9, 9, 9, 9

  d DB 4 DUP(1, 2) ;is an alternative way of declaring: d DB 1, 2, 1, 2, 1, 2, 1, 2

Example: Define a byte variable with the value 35H and print the value of the variable.

```
ORG 100H

.MODEL SMALL
.STACK 100H

.DATA
        VAR DB 35H

.CODE
MAIN PROC
        MOV AX, @DATA
        MOV DS, AX

        MOV AH, 2
        MOV DL, VAR
        INT 21H

        MOV AH, 4CH
        INT 21H
```

```
MAIN ENDP

END MAIN
```

@Data is the name of the data segment defined by .DATA. The assembler translates the name @DATA into a segment number. Two instructions are needed because a number (the data segment number) may not be moved directly into a segment register.

INT 2lh, function 9, expects the offset address of the character string to be In DX. To get it there, we use a new instruction: / LEA destination, source where destination is a general register and source is a memory location. LEA stands for "Load Effective Address." It puts a copy of the source offset address into the destination. For example, LEA DX, MSG puts the offset address of the variable MSG into UX.

Example: Define a byte variable with a string “hello” and print it.

```
ORG 100H

.MODEL SMALL
.STACK 100H


.DATA

        VAR DB 5H
        MSG1 DB "HELLO!$"

.CODE

MAIN PROC
        MOV AX, @DATA
        MOV DS, AX

        MOV AH, 2
        MOV DL, VAR
        INT 21H
```

```
        MOV DL, 20H
        INT 21H

        MOV AH, 9
        LEA DX, MSG1
        INT 21H

        MOV AH, 4CH
        INT 21H
MAIN ENDP

END MAIN
```

INT 2lh, function 9, expects the offset address of the character string to be in DX. To get it there, we use a new instruction:

LEA Destination, Source

where destination is a general register and source is a memory location. LEA stands for "Load Effective Address." It puts a copy of the **source offset address** into the destination.

Example:

LEA DX, MSG1

puts the offset address of the variable MSG into DX.

Instruction	**Operands**	**Descriptions**
LEA	REG, MEM	Load Effective Address. Algorithm: REG = address of memory (offset)

Tasks

1. Take two numeric inputs, load the **greater value in register BH** and the **smaller in CH**. Subtract the value in **CH from that of BH** and show the output in a **newline**.
2. Define a string with lower-case characters and print them in uppercase characters.