Department of Computer Science and Engineering

Course Code: CSE341	**Credits: 1.5**
Course Name: Microprocessors	**Semester: Summer' 25**

Lab 06
Macros and Procedures

I. Topic Overview:

The lab is designed to introduce the students to get the basic idea of Macros and Procedures. In this lab, we'll discuss two program structures called a macro and procedure and understand how these two work.

II. Lesson Fit:

In order to do the lab with ease, the student must have completed all the previous labs.

III. Learning Outcome:

After this lecture, the students will be able to:

A. Use Macro and Procedure.

B. Program using Macros and Procedures.

C. Differ between Macro and Procedure.

IV. Anticipated Challenges and Possible Solutions

A. Students might get confused between Macros and Procedures.

1. In Procedures the code exists in one place and when that procedure is called the control is passed to that place each time. So we are using the same code here but written only once.
2. In macros, the actual code corresponding to the macro is inserted at the calling place at compile time, which is similar to writing the same code again and again.

V. Acceptance and Evaluation

If a task is a continuing task and one couldn't finish within time limit, then he will continue from there in the next Lab, and if it is a one Lab task then it will be given as a home work and in the next Lab you have to submit the code and have to face a short viva. A deduction of 30% marks is applicable for late submission. The marks distribution is as follows:

Code: 50%

Viva: 50%

VI. Activity Detail

A. Hour: 1

Discussion: Macro

A **procedure** is run during program execution. When a procedure is called, the program jumps to the procedure(control is transferred to the procedure), runs its instructions, and then comes back to where it was originally called.

A **macro** is different – it works at assembly time (Assembly time is to assemblers what compilation time is to compilers). When a macro is used, the assembler directly copies the macro's instructions into the program where it's called. This means that when the program runs, there is no transfer of control (just to remember, no jumping to another place) — the code is already there.

You can think of a macro as a **block of code that has been given a name**. This block can have instructions, assembler directives(also known as pseudo-ops, eg. MACRO, DB, DW, ENDM etc.), comments, or even include other macros.

Syntax:

```
macro_name   MACRO   a1, a2, … an

        Instructions

        ……………..

ENDM
```

- **macro_name** → any name you choose for the macro
- **MACRO** → tells the assembler that the macro starts here
- **ENDM** → tells the assembler that the macro ends here
- **a1, a2, … an** → optional arguments that act like placeholders in the macro

Example 1: Macro to Move One Word Variable into Another

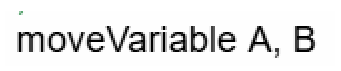
We will create a macro called **moveVariable** that copies the value from one word variable (B) to another (A).

```
.MODEL SMALL

; Macro definition
moveVariable MACRO var1, var2          ; var1 = destination, var2 = source
   push var2                          ; save var2 on stack
   pop  var1                          ; pop the value from stack, store into var1
ENDM

.STACK 100H

.DATA
A dw 2
B dw 5

.CODE
MAIN PROC

   ; Initialize DS
   MOV AX, @DATA
   MOV DS, AX

   ; Use the macro to move B into A
   moveVariable A, B                  ;calling macro - A is received by var1, B by var2

   ; Print value of A (convert to ASCII first)
   mov dx, A
   add dx, 48
   mov ah, 2
   int 21h

;exit to DOS
 MOV AX, 4C00h
 INT 21h

MAIN ENDP
END MAIN
```

Here, the name of the macro is **moveVariable** and **var1, var2** are the dummy arguments. **A, B** are the real arguments.

To use the macro in a program we invoke it within the **code segment**. Keep in mind that the macro must be defined **prior** to invoking(calling) it anywhere in the program.

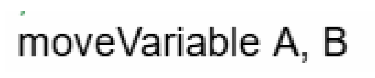
When the assembler encounters the macro name:



```
moveVariable A, B
```

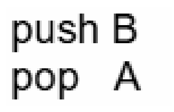
it expands the macro i.e. it copies the macro statements into the program at the position of invocation and while doing so replaces each dummy argument by the corresponding actual argument.

So in this case, to expand this macro, assembler would copy the macro statements into the program at the position of the call, replacing var1 by A and var2 by B.



```
moveVariable A, B
```

is thus replaced by



```
push B
pop  A
```

Try printing the contents in variable A to see if it holds the value of variable B.

Example 2: Macros that Invoke Other Macros

A macro itself can also invoke another macro.

For example, we have 2 macros named **addNumbers** & **printNumber** that add 2 numbers and print the output. **addNumbers** is invoked first from the main PROCEDURE, and **printNumber** is then invoked from the **addNumbers** macro in the following example.

Example : A macro that adds 2 numbers and prints the output.

```
.MODEL SMALL

; ----- Macro to add two numbers -----
addNumbers MACRO num1, num2, result
   mov al, num1
   add al, num2
   mov result, al
   printNumber result
ENDM

; ----- Macro to print a number (0–9) -----
printNumber MACRO num
   mov dl, num
   add dl, 48        ; Convert to ASCII
   mov ah, 2
   int 21h
ENDM

.DATA
a db 3
b db 5
```

```
sum db ?

.STACK 100h
.CODE
MAIN PROC
    mov ax, @DATA
    mov ds, ax

    ; Call first macro
    addNumbers a, b, sum  ; Adds a + b and prints result


    ; Exit to DOS
    mov ax, 4C00h
    int 21h
MAIN ENDP
END MAIN
```

Problems: 1 - 9

B. Hour: 2

Discussion: Procedures

A procedure is a set of instructions that performs a specific task and can be **reused** in a program. You can call multiple times by its name whenever you need that task done, instead of repeating code. This makes your program cleaner and easier to understand.

After the procedure finishes its job, it returns control back to the point in the program where it was called.

1. **Syntax**

```
PROC procedure_name type

        ; body of procedure (instructions)

         RET

ENDP procedure_name
```

Example: **Adding 2 numbers, this time using a procedure**

```
.MODEL SMALL
.STACK 100h
.DATA
   num1 DW 5      ; first number
   num2 DW 4     ; second number

.CODE
MAIN PROC
   MOV AX, @DATA
   MOV DS, AX      ; initialize data segment

   ; Load numbers into registers
   MOV AX, num1
   MOV BX, num2

   CALL AddNumbers ; call procedure to add AX and BX

   MOV DL, AL        ;number is small enough so using AL
   ADD DL,30h         ;ASCII conversion
   MOV AH,2
   INT 21h             ;prints 9 in this case

   ; Exit to DOS
   mov ax, 4C00h
   int 21h
MAIN ENDP       ;notice we ended MAIN PROCEDURE here


; Procedure to add two numbers in AX and BX, result in AX
AddNumbers PROC NEAR
   ADD AX, BX                ; AX = AX + BX
   RET
AddNumbers ENDP

END MAIN     ;notice we ended MAIN block here
```

How PROC and ENDP Work:

I. The **PROC** statement is used to define the start of a procedure or subroutine in assembly language. It tells the compiler that a new block of reusable code is beginning.

   The procedure is given a name, which is a valid identifier, so you can call it from anywhere in your program.

II. The **ENDP** statement marks the end of that procedure.

Procedure Types:

- **NEAR:** Procedure is in the **same memory segment** as the calling code.
- **FAR:** Procedure is in a **different memory segment**.

If you don't specify the type, **NEAR** is the default. The main procedure is usually treated as **FAR**.

Note: PROC and **ENDP** are compiler directives, meaning they are not turned into machine code. Instead, the compiler uses them to keep track of the procedure's starting and ending addresses. This helps when the procedure is called from somewhere else in the program.

CALL (Direct & Indirect Calls) and RET Instructions :

I. **CALL instructions:**

The CALL instruction is used to invoke (jump to) a procedure in assembly language. There are two main forms:

- **Direct CALL:**

  CALL **procedure_name** **[as seen above]**

  This directly calls a procedure by its name (known at compile time).

- **Indirect CALL:**

  CALL **address_expression**
  (e.g CALL BX — BX has the memory location of the procedure)

  Here, the address of the procedure is stored in a register or memory location, and the call is made through that.

II. **RET Instruction**

Syntax: **RET pop_value**

The RET instruction is used to return from a procedure back to the point where it was called.

- When a procedure is called, the **return address** (the next instruction after CALL) is pushed onto the stack.
- When RET executes, it pops the return address (saved during the CALL) from the stack back into the Instruction Pointer (IP), so the program knows where to continue execution.
- Optionally, **pop_value** can be specified as **N**, where N is an integer. It then removes N extra bytes from the stack, which is useful when parameters are pushed onto the stack before the call.

Execution of a CALL :SUMMARY

- The current IP (Instruction Pointer) is pushed onto the stack as the return address.
- Control jumps to the procedure's address by updating IP.
- For a **FAR** call, both Code Segment (CS) and IP are saved and updated.
- After the procedure finishes, RET brings the control back to the calling point.

Parameter Passing:

One simple and fast way to pass values to procedures is by using CPU **registers**.

- Before calling the procedure, load input values into registers.
- Inside the procedure, use those registers directly.
- Place the result in a register to use after returning.

Here is another example of a procedure that receives two parameters in AL and BL registers, multiplies these parameters and returns the result in AX register:

Example: Procedure that Multiplies Two Numbers

```
MOV AL, 1      ; first value
MOV BL, 2      ; second value

CALL m2        ; call procedure to multiply AL and BL
CALL m2
CALL m2
CALL m2

RET          ; return to operating system

m2 PROC
   MUL BL     ; multiply AL by BL, result in AX
   RET        ; return to caller
m2 ENDP

END
```

In the above example, the value in the **AL** register is updated every time the procedure is called, while the **BL** register remains unchanged. This means the procedure effectively calculates 2^4 (2 raised to the power of 4).

As a result, the final value stored in the **AX** register is **16** (which is **10h** in hexadecimal).

Task: **Problems:** 10-18

C. Hour: 3

Discussion: Macros vs. Procedures

Aspect	Procedure	Macro
How to call/ use	Use the **CALL** instruction.	Just type the macro name directly.
Example of call	**CALL MyProc**	**MyMacro**
Location in memory	Stored at a specific address in memory.	Expanded directly in the program code.
Effect of multiple calls	CPU jumps to one memory location each time; uses **RET** to return. Stack stores return address.	Expanded each time at the call site, increasing executable size if used many times.
Passing parameters	Use stack or general-purpose registers to pass parameters.	Pass parameters directly after macro name. For example, **MyMacro 1, 2, 3**.
End marking directive	End marked by **ENDP procedure_name.**	End marked simply by **ENDM**.

Lab 6 Activity List

1. Write a macro to calculate the factorial of a number.
2. Write a macro to reverse a string using an array.
3. Write a macro to identify the maximum between 2 numbers and use that macro to figure out the maximum between n amount of numbers.
4. Write a macro to calculate x^n.
5. Write a macro that prints all the prime numbers upto a given number.
6. Write a program that checks whether a value exists in the array or not using macro.
7. Write a procedure to display a character string. The string is the procedure parameter.
8. Write a procedure to reverse a string.
9. Write a procedure to identify the maximum number between 3 numbers.
10. Write a procedure that prints all the prime numbers upto a given number.
11. Write a program that checks whether a value exists in the array or not using procedure.