# Department of Computer Science and Engineering

| Course Code: CSE341 | Credits: 1.5 |
|---|---|
| Course Name: Microprocessors | Semester: Summer 21 |

**Lab 08**
**Macros and Procedures**

## I. Topic Overview:

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

## 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 actual corresponding to the Macro which means the code is inserted to 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 called at execution time; controls transfer to the procedure and returns after executing the statements. A macro is invoked at assembly time (Assembly time is to assemblers what compilation time is to compilers). The assembler copies the macros statements into the program at the position of the invocation. When the program executes, there is no transfer of control. A macro is a block of text that has been given a name. The text may consist of instructions, pseudo-ops, comments, or references to other macros.

1. **Syntax**:

**macro_name MACRO a1,a2,.... an**

**statements**

**ENDM**

The macro_name can be any arbitrary user supplied name for the macro whereas the pseudo-ops MACRO and ENDM indicate the beginning and the end of the macro respectively. A1, a2, a3,... an are optional list of dummy arguments to be used by the macro.

**Example 1:** Define a macro to move a word variable B into another word variable A.

```
.model small

moveVariable macro  var1, var2 ;arguments must be memory words or 16 bit registers

    push var2

    pop var1

endm

.data

A dw 2

B dw 5

.stack 300h

.code

mov ax, @data

mov ds, ax

moveVariable A, B

mov dx, A

add dx, 48

mov ah, 2

int 21h

mov ax, 4ch

int 21h
```

Here, the name of the macro is moveVariable and arg1, arg2 are the dummy arguments. To use the macro in a program we invoke it within the code segment. It is to be kept in mind that the macro must be defined prior to invoking it anywhere in the program. When the assembler encounters the macro name, 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.

If we want to invoke the moveVariable macro, we have to write the name of the macro followed by the actual arguments. In order to copy the word variable B into the word variable A, we have to invoke the macro within the code segment as shown:

moveVariable A, B

To expand this macro, assembler would copy the macro statements into the program at the position of the call, replacing arg1 by A and arg2 by B. The result is

Push A

Pop B

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

**2. Macros that invoke other macros**

A macro make invoke another macro. Suppose, for example, we have two macros that save and restore three registers. These macros are invoked by the macro in the following example.

**Example** : A macro that copies a string.

```
.model small
saveReg macro R1, R2, R3
   push R1
   push R2
   push R3
endm

restoreReg macro S1,S2,S3
   pop S1
   pop S2
   pop S3
```

```
endm

copy macro source, destination, length
   saveReg CX, SI, DI
   lea SI, source
   lea DI, destination
   CLD
   MOV CX, length rep
   movsb restoreReg
   DI, SI, CX
endm

.data
str2 dw "ABC$"
str1 dw "XYZ$"

.stack 3000h

.code

mov ax, @data
mov ds, ax
mov es, ax

copy str2, str1, 3 ;copies str2 to str1
;print string1
mov ah, 9
lea dx, str1
int 21h
```

```
mov ax, 4ch
int 21h
```

**Problems:** 1 - 9

B. **Hour: 2**

**Discussion: 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.

1. **Syntax**

```
PROC name type
        ; body of procedure
ret
ENDP name
```

**Example**:

```
ORG 100h
CALL m1
MOV AX, 2
RET ; return to operating system.

m1 PROC
MOV BX, 5
RET ; return to caller.
m1 ENDP

END
```

The above example calls procedure m1, does MOV BX, 5, and returns to the next instruction after CALL: MOV AX, 2.

```
PrintString PROC FAR
MOV AH, 09h
INT 21h
RET ; return of the procedure
PrintString ENDP
```

**PROC** is a statement used to indicate the beginning of a procedure or subroutine.

**ENDP** indicates the end of the procedure.

**name** may be any valid identifier.

type can be **NEAR** (in same segment) or **FAR** (in a different segment) -- if omitted, **NEAR** is assumed. Main Procedure is always **FAR** (implicit)

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

a. **Direct and Indirect**

The **CALL** keyword invokes a procedure. This keyword has two forms which are **direct** and **indirect.**

a. **Direct,**

CALL **name**

where **name** is the name of a procedure.

b. **Indirect,**

CALL **address_expression**

where ***address_expression*** specifies a register or memory location containing the address of a procedure.

b. **RET Instruction**

To return from a procedure, the instruction

**ret pop_value**

is executed

1. The integer argument **pop_value** is optional
2. **ret** causes the stack to be popped into IP

3. If **pop_value N** is specified, it is added to SP -- in effect removes N additional bytes from the stack.

## 2. Execution of a CALL

a. The return address to the calling program (the current value of the IP) is saved on the stack.
b. IP get the offset address of the first instruction of the procedure (this transfers control to the procedure).
c. FAR procedures must process CS:IP instead of just IP.

## 3. Parameter Passing

There are several ways to pass parameters to procedure, the easiest way to pass parameters is by using registers, 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**

```
ORG 100h

MOV AL, 1

MOV BL, 2

CALL m2

CALL m2

CALL m2

CALL m2

RET ; return to operating system.


m2 PROC

MUL BL ; AX = AL * BL.

RET ; return to caller.

m2 ENDP
```

```
END
```

In the above example value of AL register is update every time the procedure is called, BL register stays unchanged, so this algorithm calculates 2 in power of 4, so final result in AX register is 16 (or 10h).