Procedure and Macros in 8086

Procedure:

In a program, we very frequently face situations where there is a need to perform the same set of task again and again. So, for that instead of writing the same sequence of instructions, again and again, they are written separately in a subprogram. This subprogram is called a procedure. With the help of procedures, we can very well implement the concept of modular programming in our code. Also, whenever we need to execute the instructions mentioned in the procedure, we can simply make a CALL to it. Therefore, with the help of procedures, the duplicity in the instructions can be avoided.

The **Syntax for a procedure** is as follows:

```
Procedure_name PROC  [near / far]
    Instruction 1
    Instruction 2
    - - - - - - - - - - - -
    - - - - - - - - - - - -
    Instruction n
Procedure_name ENDP
```

- Here, the **PROC** is a keyword to define that the set of instructions enclosed by the given name is a procedure.
- The **ENDP** keyword defines that the body of the procedure has been ended.
- All the instructions lying between these two keywords are the instructions that belong to the procedure and will be executed whenever a **CALL** to the procedure is made.
- The keyword near or far defines the range of code within which the procedure is defined. If it is defined in the same segment as the rest code, then near is used.
- If it is defined in some other segment, then the keyword far is used for it.

The CALL to a procedure can be made in the following way,

```
CALL procedure_name
```

At the end of the procedure, the **RET** instruction is used. This instruction will cause the execution to be transferred to the program from which the call to the procedure was made.

Macro:

A **Macro** is a set of instructions grouped under a single unit. It is another method for implementing modular programming in the 8086 microprocessors (The first one was using Procedures).

The **Macro** is different from the Procedure in a way that unlike calling and returning the control as in procedures, the processor generates the code in the program every time whenever and wherever a call to the **Macro** is made.

A **Macro** can be defined in a program using the following assembler directives: **MACRO** (used after the name of Macro before starting the body of the Macro) and **ENDM** (at the end of the Macro). All the instructions that belong to the Macro lie within these two assembler directives. The following is the syntax for defining a **Macro in the 8086 Microprocessor**:

```
Macro_name  MACRO  [ list of parameters ]
    Instruction 1
    Instruction 2
    - - - - - - - - - - - -
    - - - - - - - - - - - -
    - - - - - - - - - - - -
    Instruction n
ENDM
```

And a call to Macro is made just by mentioning the name of the Macro:

```
Macro_name [ list of parameters]
```

It is optional to pass the parameters in the Macro. If you want to pass them to your macros, you can simply mention them all in the very first statement of the Macro just after the directive: MACRO.

The advantage of using Macro is that it avoids the overhead time involved in calling and returning (as in the procedures). Therefore, the execution of Macros is faster as compared to procedures. Another advantage is that there is no need for accessing stack or providing any separate memory to it for storing and returning the address locations while shifting the processor controls in the program.

But it should be noted that every time you call a macro, the assembler of the microprocessor places the entire set of Macro instructions in the mainline program from where the call to Macro is being made. This is known as Macro expansion. Due to this, the program code (which uses Macros) takes more memory space than the

code which uses procedures for implementing the same task using the same set of instructions.

Hence, it is better to use Macros where we have small instruction sets containing less number of instructions to execute.