6. Procedure and Macro in Assembly Language Program

16 Marks

Syllabus:

6.1 Defining Procedure - Directives used, FAR and NEAR, CALL and RET instructions, Reentrant and recursive procedures, Assembly Language Programs using Procedure

6.2 Defining Macros.
Assembly Language Programs using Macros.

Procedures:

While writing programs, it may be the case that a particular sequence of instructions is used several times. To avoid writing the sequence of instructions again and again in the program, the same sequence can be written as a separate subprogram called a procedure. Each time the sequence of instructions needs to be executed, CALL instruction can be used. CALL instruction transfers the execution control to procedure. In the procedure, a RET instruction is used at the end. This will cause the execution to be transferred to caller program.

Often large programs are split into number of independent tasks which can be easily designed and implemented. This is known as modular programming.

Advantages:

1. Programming becomes simple.
2. Reduced development time – as each module can be implemented by different persons.
3. Debugging of smaller programs and procedures is easy.
4. Reuse of procedures is possible.
5. A library of procedures can be created to use and distribute.

Disadvantages:

1. Extra code may be required to integrate procedures.
2. Liking of procedures may be required.
3. Processor needs to do extra work to save status of current procedure and load status of called procedure. The queue must be emptied so that instructions of the procedure can be filled in the queue.

Defining procedures:

Assembler provides PROC and ENDP directives in order to define procedures. The directive PROC indicates beginning of a procedure. Its general form is:

```
Procedure_name  PROC  [NEAR|FAR]
```

NEAR | FAR is optional and gives the types of procedure. If not used, assembler assumes the procedure as near procedure. All procedures are defined in code segment. The directive ENDP indicates end of a procedure. Its general form is:

```
Procedure_name  ENDP
```

For example,

```
Factorial  PROC  NEAR
. . .
. . .
. . .
Factorial  ENDP
```

CALL instruction:

CALL instruction s used to call a procedure for execution. Before a procedure is called, the CALL instruction saves the address of instruction, which is next to CALL instruction, on stack. The procedure call is of two types:

- Intra segment or near call
- Inter segment or far call

A near call is used to call a procedure within same code segment. A far call is used to call a procedure which is in different code segment. For near call, only value of IP is saved on stack. For a far call, value of CS and IP is stored on stack.

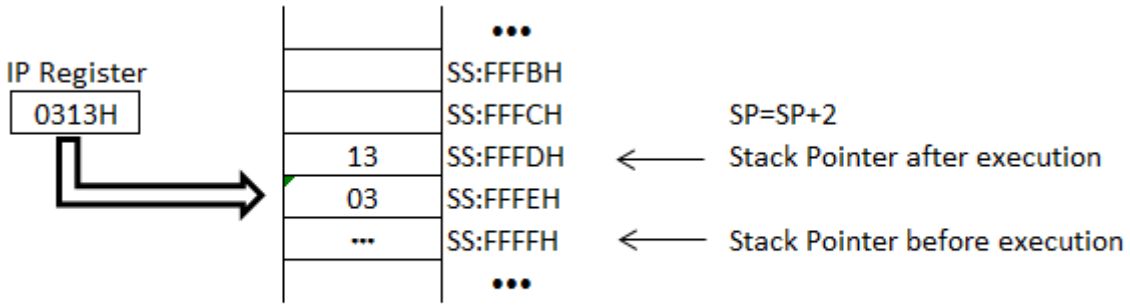


Fig. Near Call - only contents of IP are saved

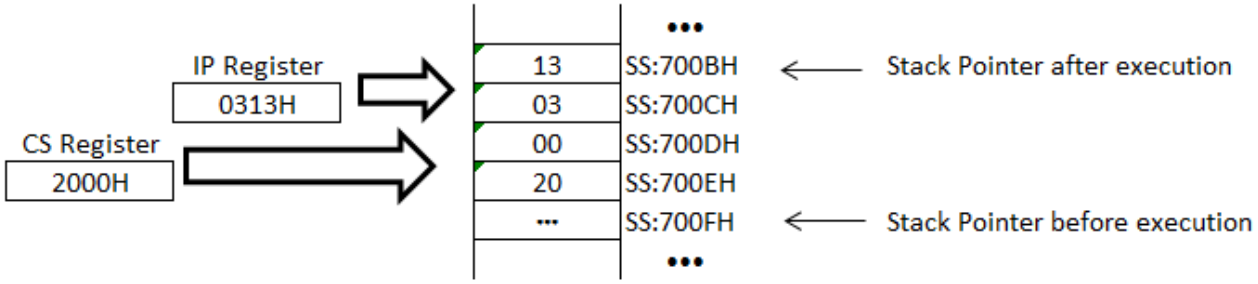


Fig. Far Call - contents of CS and IP are saved

RET instruction:

This instruction is used to transfer program execution from procedure back to the calling program. Depending upon type of call, RET instruction may be of two types:

- Near RET or intra segment return
- Far RET or inter segment return

If procedure is declared as near, RET instruction copies a word from top of stack (TOS) into IP. If procedure is declared as far, RET instruction copies a word from top of stack (TOS) into IP, stack pointer is decremented by 2 and again a word is copied from TOS into CS register.

Example: Program using procedures

Following example shows how to define and use near procedure (i.e. a procedure within same code segment).

```
ASSUME CS:CODE, DS:DATA, SS:STACK_SEG

DATA SEGMENT
NUM1 DB 50H
NUM2 DB 20H
ADD_RES DB ?
SUB_RES DB ?
DATA ENDS

STACK_SEG SEGMENT
     DW 40 DUP(0)   ; stack of 40 words, all initialized to zero
TOS LABEL WORD
STACK_SEG ENDS

CODE SEGMENT
START:    MOV AX, DATA   ; initialize data segment
          MOV DS, AX
          MOV AX, STACK_SEG   ; initialize stack segment
          MOV SS, AX
          MOV SP, OFFSET TOS  ; initialize stack pointer to TOS
          CALL ADDITION
          CALL SUBTRACTION
          MOV AH, 4CH
          INT 21H
          ADDITION PROC NEAR
          MOV AL, NUM1
          MOV BL, NUM2
          ADD AL, BL
          MOV ADD_RES, AL
          RET
```

```
            ADDITION ENDP
            SUBTRACTION PROC
            MOV AL, NUM1
            MOV BL, NUM2
            SUB AL, BL
            MOV SUB_RES, AL
            RET
            SUBTRACTION ENDP
CODE ENDS
END START
```

Passing parameters to and from procedures:

The data values or addresses passed between procedures and main program are called parameters. There are four ways of passing parameters:

Passing parameters in registers:

Parameter values can be stored in registers. When procedure is called, in procedures these registers can be accessed. Similarly, return values can be stored in registers and accessed in main program.

Passing parameters in dedicated memory locations:

This is another way of passing parameters to procedures. Specific memory locations identified by names can be used to store parameter values in caller procedure. Later in called procedures these values can be used with the name of memory locations.

Passing parameters with pointers passed in registers:

Instead of passing actual values in registers, pointers which point to these values can be used. The offset address of variable is stored in SI, DI or BX register. These registers can be used to access values pointed by them.

Passing parameters using the stack:

To pass parameters to a procedure using the stack, push the parameters on the stack somewhere in the main program before calling the procedure. In the procedure, instructions can be used in order to read the parameter values from the stack.

Example: Passing parameters in registers

```
; Program to add two numbers by passing them as parameters in
; registers to procedure
ASSUME CS:CODE, DS:DATA, SS:STACK_SEG
DATA SEGMENT
NUM1 DB 50H
NUM2 DB 20H
```

```
ADD_RES DB ?
DATA ENDS
STACK_SEG SEGMENT
     DW 40 DUP(0)   ; stack of 40 words, all initialized to zero
TOS LABEL WORD
STACK_SEG ENDS
CODE SEGMENT
START:    MOV AX, DATA   ; initialize data segment
          MOV DS, AX
          MOV AX, STACK_SEG   ; initialize stack segment
          MOV SS, AX
          MOV SP, OFFSET TOS  ; initialize stack pointer to TOS
          MOV BL, NUM1
          MOV BH, NUM2
          CALL ADDITION
          MOV ADD_RES, AL
          MOV AH, 4CH
          INT 21H
          ADDITION PROC NEAR
          ADD BL, BH
          MOV AL, BL          ; return value is copied in AL
          RET
          ADDITION ENDP
CODE ENDS
END START
```

Macros:

Whenever it is required to use a group of instructions several times in a program, there are two ways to use that group of instructions: One way is to write the group of instructions as a separate procedure. We can call the procedure whenever it is required to execute that group of instructions. But disadvantage of using a procedure is we need stack. Another disadvantage is that time is required to call procedures and return to calling program.

When the repeated group of instruction is too short or not suitable to be implemented as a procedure, we use a MACRO. A macro is a group of instructions to which a name is given. Each time a macro is called in a program, the assembler will replace the macro name with the group of instructions.

Advantages:

1. Macro reduces the amount of repetitive coding.
2. Program becomes more readable and simple.

3. Execution time is less as compared to calling procedures.
4. Reduces errors caused by repetitive coding.

Disadvantage of macro is that the memory requirement of a program becomes more.

Defining macros:

Before using macros, we have to define them. Macros are defined before the definition of segments. Assembler provides two directives for defining a macro: MACRO and ENDM.

MACRO directive informs the assembler the beginning of a macro. The general form is:

```
Macro_name  MACRO  argument1,  argument2, …
```

Arguments are optional.

ENDM informs the assembler the end of the macro. Its general form is : ENDM

Example: Addition of two 16-bit numbers using macro

```
ADDITION  MACRO  NO1, NO2, RESULT
  MOV AX, NO1
  MOV BX, NO2
  ADD AX, BX
  MOV RESULT, AX
ENDM
ASSUME CS:CODE, DS:DATA
DATA SEGMENT
  NUM1 DW 1000H
  NUM2 DW 2000H
  RES DW ?
DATA ENDS
CODE SEGMENT
START:    MOV AX, DATA   ; initialize data segment
          MOV DS, AX
          ADDITION NUM1, NUM2, RES
          MOV AH, 4CH
          INT 21H
CODE ENDS
END START
```

In the above example three arguments are used with macro.

Example: To display strings using macro

```
DISPLAY   MACRO  MESSAGE
PUSH AX
PUSH DX
MOV AH, 09H
LEA DX, MESSAGE
INT 21H
POP DX
POP AX
ENDM
ASSUME CS:CODE, DS:DATA
DATA SEGMENT
  MSG1 DB 'Microprocessor and programming$'
  MSG2 DB 10,13,'Using macros$'
  MSG3 DB 10,13,'It eliminates repetitive coding$'
DATA ENDS
CODE SEGMENT
START:    MOV AX, DATA   ; initialize data segment
          MOV DS, AX
          DISPLAY MSG1
          DISPLAY MSG2
          DISPLAY MSG3
          MOV AH, 4CH
          INT 21H
CODE ENDS
END START
```

Delay loops: ***(Not in syllabus in 'G' scheme)***

In order to create delay loops, the steps given below are followed.

1. Determine the exact required delay.
2. Select instructions for delay loop.
3. Find out the number of T-states (clock cycles) required for execution of selected instructions.
4. Find out the time period of clock frequency at which microprocessor is running (T) i.e. duration of single T-state.

5. Find out time required to execute the loop once. This can be calculated by multiplying the period T (which is found in step 4) with the number of T-states required (n) to execute the loop once.
6. Find out the count N by dividing the required time delay T_d by the duration for execution of the loop once (n * T).

Count N = required delay T_d / n * T

Example: Write a procedure to generate a delay of 100ms using an 8086 system that runs on 10 MHz frequency.

Solution:

1) The required delay T_d = 100 ms = 100 x 10^{-3} seconds

2)

Instructions selected	**T-states required to execute**
i) MOV CX, COUNT	4
ii) DEC CX	2
iii) NOP	3
iv) JNZ LABEL	16

3) Period T = 1 / 10 MHz

= 1 / 10 x 10^{-6}

= 0.1 micro second

4) No. of cycles required to execute the loop once = 2 + 3 + 16 = 21 T-states

(Note: MOV CX, COUNT will not be in loop.)

Therefore Time required to execute loop once = n x T = 21 x 0.1 micro second = 2.1 micro second

5) Count N = required delay T_d / n * T

N = T_d / n x T

= 100 x 10^{-3} / 2.1 x 10^{-6}

= (0.1 / 2.1) x 10^{6}

= 0.047619 x 10^{6}

= 47619

= BA03H

```
PROC DELAY NEAR
MOV CX, BA03H
UP: DEC CX
NOP
JNZ UP
```

```
RET
DELAY ENDP
```

Example: 1 second delay

Considering CPU speed = 10 MHz , therefore, 1 T-state = 0.1 μ sec

T_d = 1 sec

Instructions selected	**T-states required to execute**
i) MOV BX, COUNT1	4
ii) MOV CX, COUNT2	4
iii) DEC BX	2
iv) DEC CX	2
v) JNZ LABEL	16
vi) NOP	3
vii) RET	8

Let COUNT2 = 8000H = 32768

T = 1 / 10 MHz = 0.1 μ sec

There are two loops, inner loop requires

T1 = 0.1 μ sec x 4 + (2+3+16) x 32768 x 0.1 μ sec

= 0.0688132 sec

Outer loop requires

T2 = 0.0688132 + (16 +2) x 0.1 μ sec

= 0.068815

COUNT1 = T_d / T2

= 1 / 0.068815

= 14.5317 ≈ 15 = 000FH

```
ASSUME CS:CODE
CODE SEGMENT
DELAY  PROC
       MOV BX, COUNT1          ; COUNT1 = 000FH
ABOVE: MOV CX, COUNT2          ; COUNT2 = 8000H
UP:    NOP
       DEC CX
       JNZ UP
       DEC BX
       JNZ ABOVE
```

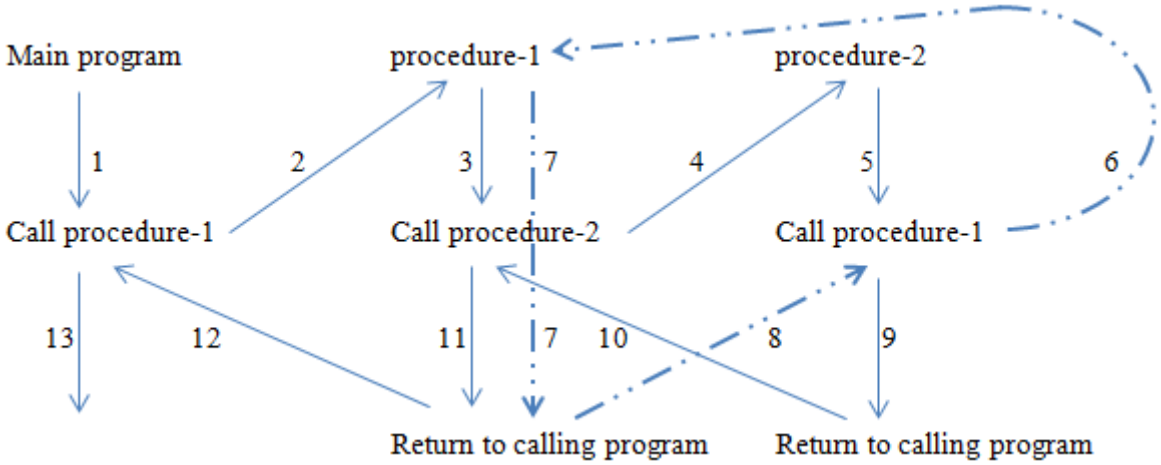
```
        RET
DELAY   ENDP
CODE    ENDS
```

Reentrant and recursive procedures:

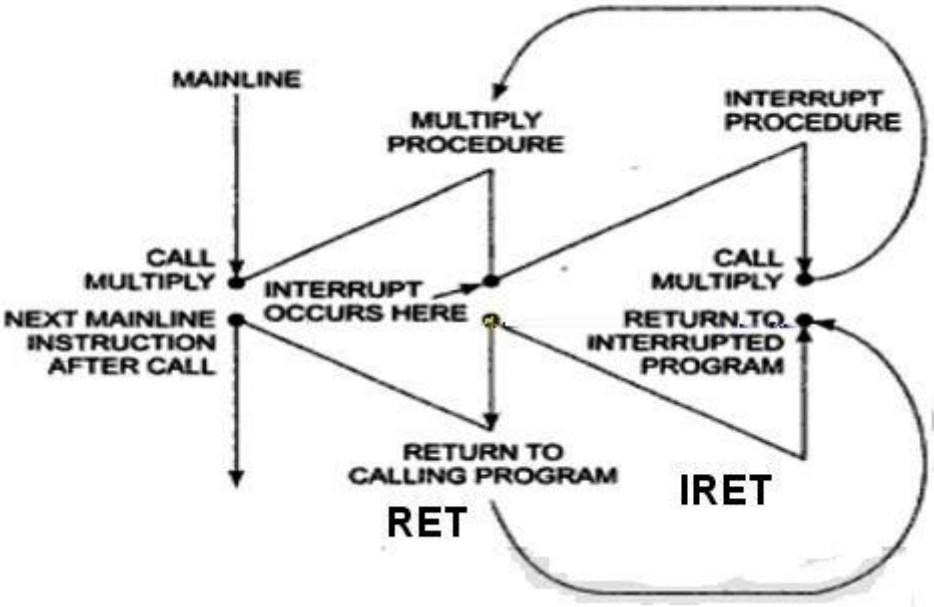
A procedure is said to be re-entrant, if it can be interrupted, used and re-entered without losing or writing over anything.

In some situation it may happen that procedure-1 is called from main program, procrdure-2 is called from procedure-1, and procrdure-1 is again called from procdure-2.

In this situation program execution flow re-enters in the procedure1. These types of procedures are called re-entrant procedures.



OR



1) The main program calls the multiply procedure.

2) In between execution of multiply procedure, when an interrupt occurs, 8086 branches to interrupt service routine

3) ISR routine then calls the multiply procedure when it needs it. The RET instruction at the end of multiply procedure returns execution to the ISR.

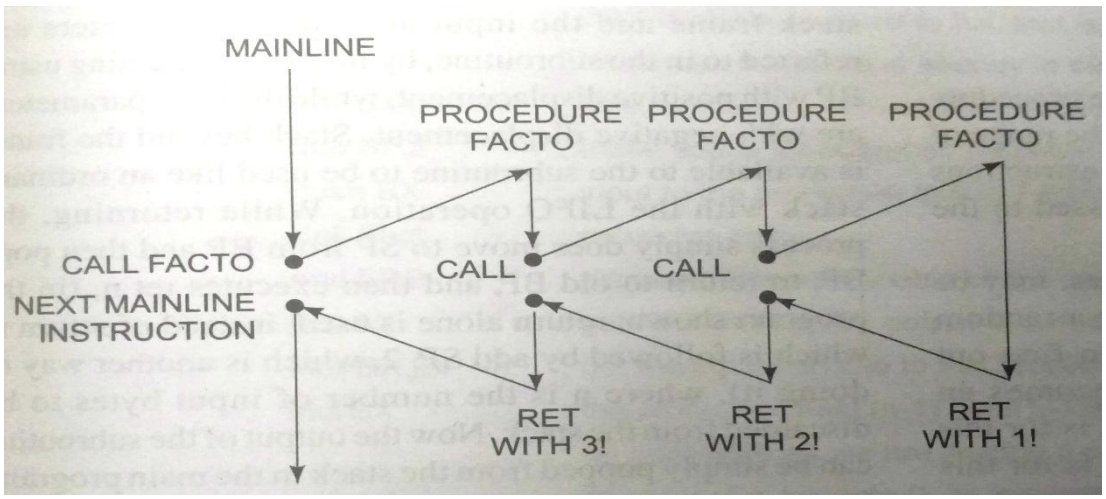
4) After execution of ISR, an IRET instruction returns program execution to the multiply procedure at the instruction next to the point where the interrupt occurred.

5) Once again multiply is executed with data values of the main program.

6) A RET at the end of multiply returns execution to the main program.

Recursive Procedures:

A recursive procedure is a procedure which calls itself. Here, the program sets aside a few locations in stack for the storage of the parameters which are passed each time the computation is done and the value is returned. Each value returned is then obtained by popping back from the stack at every RET instruction when executed at the end of the procedure.



Example:

```
ASSUME    CS:CODE, DS:DATA
DATA      SEGMENT
          N   DB 04H
          RES DW ?
DATA      ENDS
CODE      SEGMENT
START:    MOV AX, DATA
          MOV DS, AX
          MOV AL, N
          MOV AH, 00H
          CALL FACT
          INT 3
FACT      PROC
```

```
        CMP AX, 01      ;If N=1, FACT=1 Else FACT=N*FACT(N-1)
        JZ EXIT
        PUSH AX
        DEC AX          ;N-1
        CALL FACT       ;FACT(N-1)
        POP AX
        MUL RES         ;N*FACT(N-1)
        MOV RES, AX     ;RES=FACTORIAL
        RET
EXIT:
        MOV RES, 01
        RET
FACT    ENDP
CODE    ENDS
END     START
```

Difference between Near Call and Far Call Procedures:

Near Call	Far Call
A near call is a call to procedure which is in same code segment.	A far call is a call to procedure which is in different code segment.
The content of CS is not stored.	The content of CS is also stored along with IP.
In near call, content of SP is decremented by 2 and contents of offset address IP is stored.	In far call, contents of SP are decremented by 2 and value of CS is loaded. Then SP is again decremented by 2 and IP is loaded.
Example: CALL Delay	Example: CALL FAR PTR Delay

Difference between Macro and Procedure:

Macro	Procedure
Macro is a small sequence of code of the same pattern, repeated frequently at different places, which perform the same operation on different data of the same data type.	Procedure is a series of instructions is to be executed several times in a program, and called whenever required.
The MACRO code is inserted into the program, wherever MACRO is called, by the assembler.	Program control is transferred to the procedure, when CALL instruction is executed at run time.
Memory required is more, as the code is inserted at each MACRO call	Memory required is less, as the program control is transferred to procedure.
Stack is not required at the MACRO call.	Stack is required at Procedure CALL.
No overhead time required.	Extra overhead time is required for linkage between the calling program and called procedure.
Parameter passed as the part of statement which calls macro.	Parameters passed in registers, memory locations or stack.
RET is not used	RET is required at the end of the procedure
Macro is called using: <Macro_name> [argument list]	Procedure is called using: CALL <Procedure_name>
Directives used: MACRO, ENDM, LOCAL	Directives used: PROC, ENDP, FAR, NEAR
. Example: Macro_name MACRO ------ ------ instructions ------ ENDM	Example: Procedure_Name PROC -------- Procedure Statements -------- Procedure_Name ENDP