# LAB 11: Procedures or functions

**OBJECTIVES:**

Working with procedures

**Pre-Lab: Task**

Procedures or functions are very important to a program. They help divide a program into logical and more manageable parts. They are the core for procedural programming paradigm.

A function call is a sort of branching instruction that breaks the execution sequence. It requires the value of **IP** to point to some other part of the code containing the function. After the called function finishes execution, it must resume execution to the last function that called it. Further, it must also return to the same line of code in calling function from where it was called. To ensure this, at the time of function call, the value of **IP** is saved on stack by **call** instruction. At the end of called function, the **ret** instruction makes the **IP** get its previous value from the stack thus gets the calling instruction address and resumes execution there.

The general syntax for procedure definition is

**PROCNAME PROC {NEAR OR FAR}**

**<STATEMENTS>**

**PROCNAME ENDP**

Where NEAR or FAR is optional that defines whether the function resides in the same code segment.

Near calls and returns transfer control between procedures in the same code segment. Far calls and returns pass control between different segments, so a far call also pushes **CS**, register value on the stack and **far ret** pops back **CS** value along with the **IP** register value.

Assembler uses {NEAR or FAR} operands to determine if you're calling this procedure with a near or far call instruction, **default is NEAR** if one is not mentioned. Consider the following two procedures:

NPROC PROC NEAR

MOV AX, 0

RET

NPROC ENDP

FPROC PROC FAR

MOV AX, 0FFFFH

RET

FPROC ENDP

CALL NPROC ; CALLING NEAR PROC

CALL FPROC ; CALLING FAR PROC

The assembler automatically generates a three-byte (near) call for the first call instruction above because it knows that NPROC is a near procedure. It also generates a five-byte (far) call instruction for the second call because FPROC is a far procedure. Within the procedures, the assembler automatically converts all **ret** instructions to near or far returns depending on the type of routine.

We will use NEAR procedures here.

**Program:**

The program below defines a main function. It further defines two functions; the NEWLINEPROC that outputs a new line on the console and PRNTCHAR procedure that just prints a character.

Note that the procedures have no arguments. Argument passing requires the knowledge of stack management with respect to function calling mechanism, which will be covered later.

. MODEL SMALL

. STACK 100H

.DATA

STR DB 'CALLING A PROCEDURE FOR NEW LINE$'

NEWLINE DB 0AH, 0DH, '$'

. CODE

MAIN PROC

MOV AX, @DATA

MOV DS, AX

MOV AH, 9

MOV DX, OFFSET STR

INT 21H

CALL NEWLINEPROC ; CALLS FUNCTION NEWLINEPROC

CALL PRNTCHAR

RET ; RETURN IS MUST, OTHERWISE CONTROL WILL FALL THROUGH

MAIN ENDP

PRNTCHAR PROC ; PROCEDURE DEFINITION STARTS

MOV AH, 2

MOV DL, 'B'

INT 21H

RET ; RETURN TO THE FUNCTION THAT CALLED IT

PRNTCHAR ENDP  ; PROCEDURE DEFINITION ENDS

NEWLINEPROC PROC ; PROCEDURE DEFINITION STARTS

MOV AH, 9

MOV DX, OFFSET NEWLINE

INT 21H

RET ; PROCEDURE DEFINITION ENDS

NEWLINEPROC ENDP ; PROCEDURE DEFINITION ENDS

END MAIN

The above program starts by defining two strings str 'calling a procedure for new line$' and newline 0ah, 0dh, '$'.

Next it defines two other functions; the NEWLINEPROC to outputs a new line on the console and PRNTCHAR procedure to prints a character. The main function then calls both these functions as:

**CALL NEWLINEPROC**

**CALL PRNTCHAR**

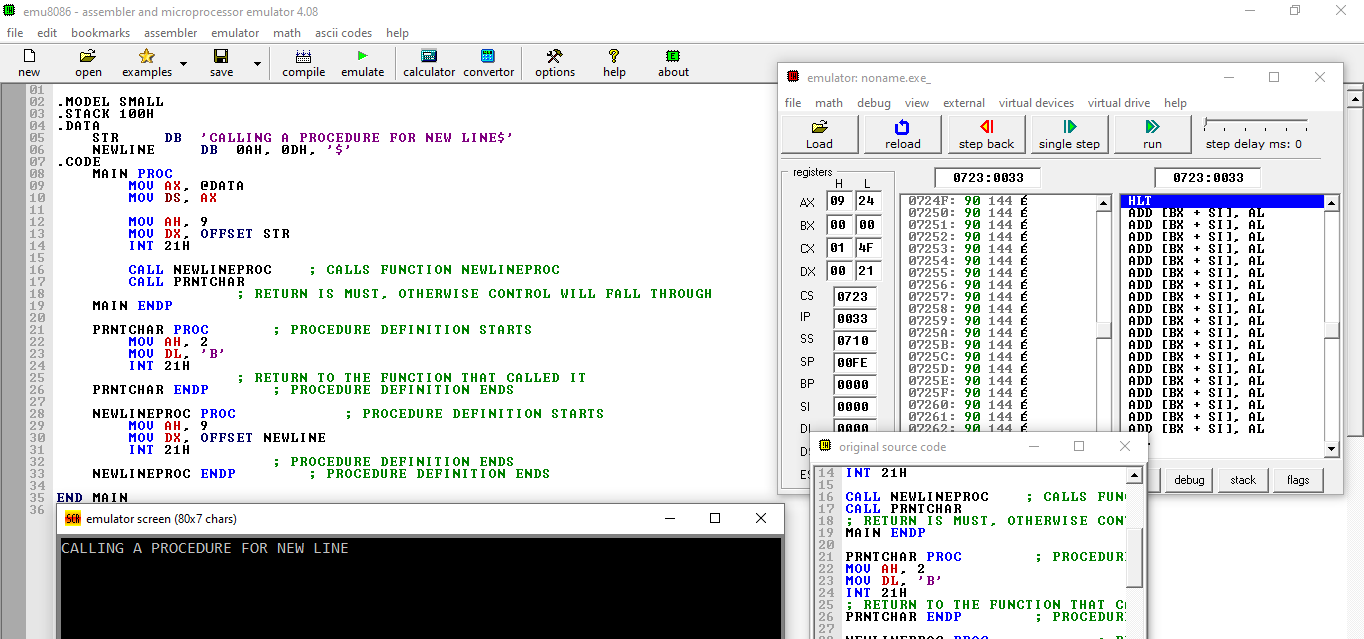
When each function finishes execution, it returns the control back to the main function from where it was called.

**In-Lab Task:**

**Activity 1**

Try implementing the above program but without using the **ret** statement at the end of functions and observe the output.

**Control Falls through and result of last function is printed and program terminates.**

****

**Activity 2:**

**Write a function that prints alphabets in reverse order, without using array.**

**CODE:**

.model small

.stack 100h

.data

; No data required for this program

.code

main proc

; Initialize data segment

mov ax, @data

mov ds, ax

; Initialize the starting character 'Z' into register

mov cx, 26 ; Counter for 26 alphabets

mov dl, 90 ; Start with ASCII value of 'Z'

print\_loop:

; Output the character in DL

mov ah, 02h ; Function to print character in DL

int 21h ; DOS interrupt to print character

; Decrement the character

dec dl ; Move to previous character in ASCII

; Decrement the counter

loop print\_loop ; Loop until CX = 0

; Exit to DOS

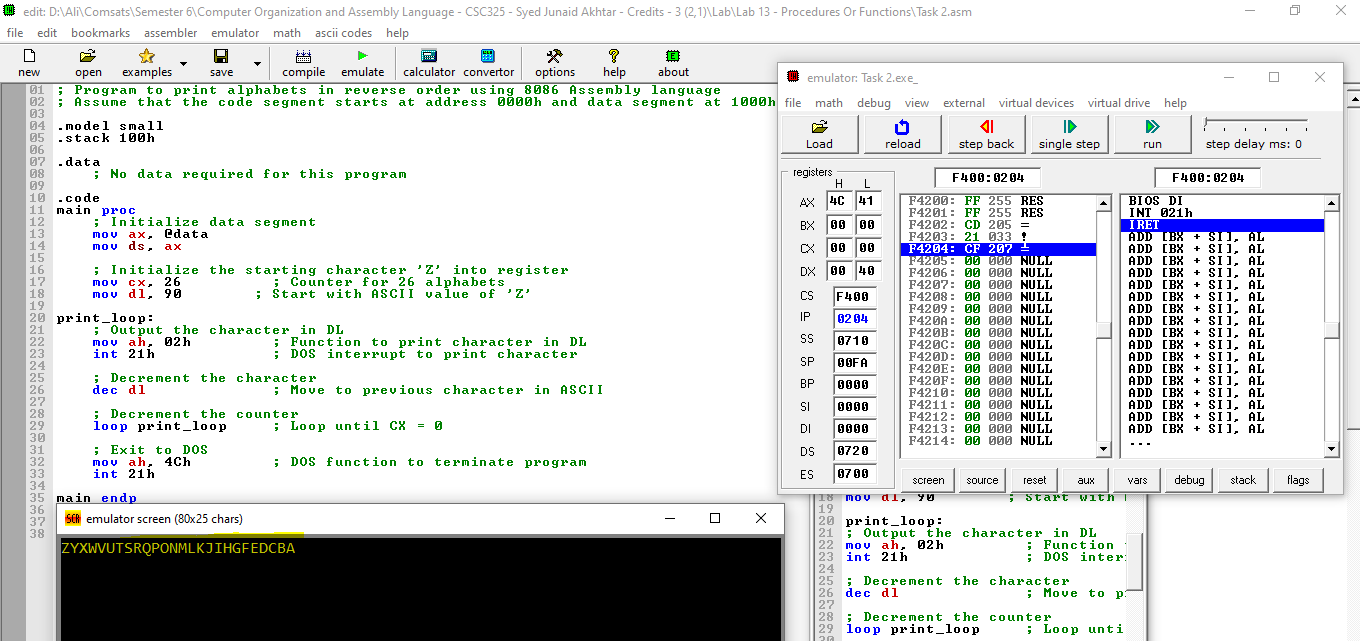
mov ah, 4Ch ; DOS function to terminate program

int 21h

main endp

end main

MAIN



**Activity 3**

**Write a function to print a multi-digit number.**

**CODE:**

.model small

.stack 100h

.data

number dw 12345 ; Multi-digit number to print

.code

main proc

; Initialize data segment

mov ax, @data

mov ds, ax

; Load the number to AX

mov ax, number

; Convert number to string

mov bx, 10 ; Base 10 for decimal conversion

xor cx, cx ; Clear CX for digit count

convert\_loop:

xor dx, dx ; Clear DX for division

div bx ; AX / 10, quotient in AX, remainder in DX

push dx ; Push remainder (digit) onto stack

inc cx ; Increment digit count

test ax, ax ; Check if quotient is zero

jnz convert\_loop ; If not zero, continue loop

print\_loop:

pop dx ; Pop digit from stack

add dl, '0' ; Convert to ASCII

mov ah, 02h ; Function to print character in DL

int 21h ; DOS interrupt to print character

loop print\_loop ; Loop until CX = 0

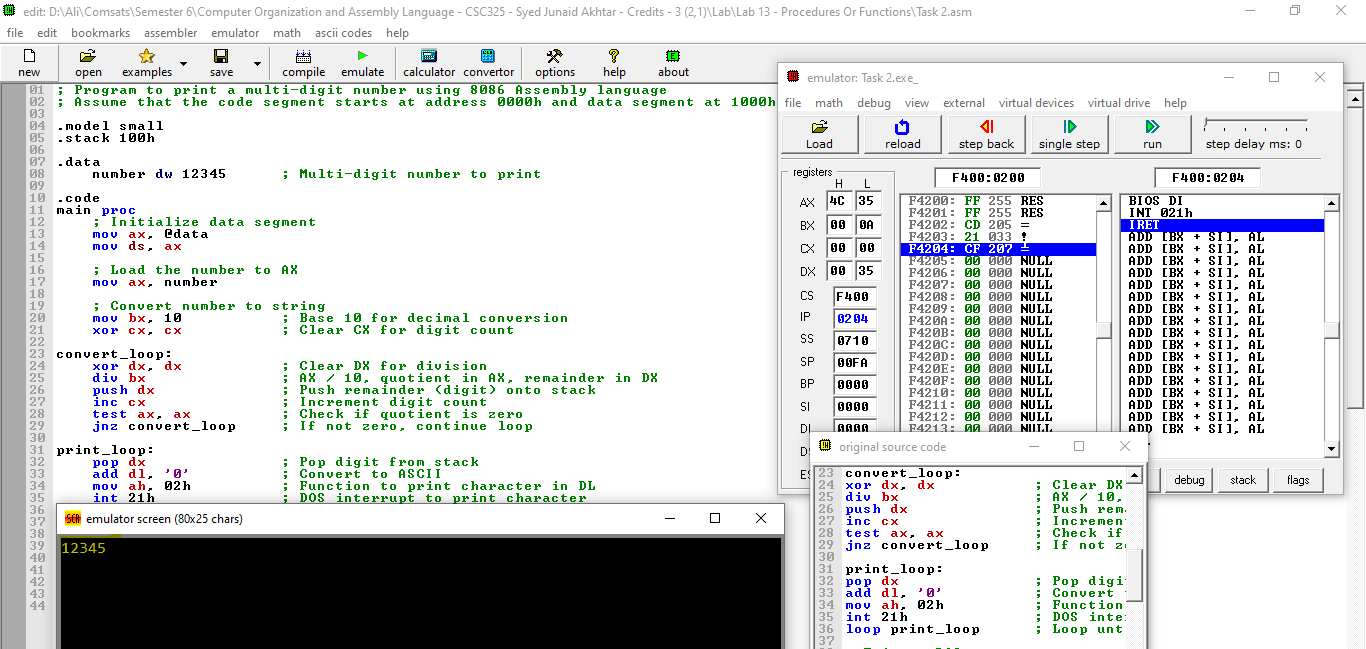
; Exit to DOS

mov ah, 4Ch ; DOS function to terminate program

int 21h

main endp

end main



**Post-Lab Assignment**

1. What happens if RET statement is missed at the end of a function?

The program control falls through and the next function or the following line number executes.

1. How does a call instruction like CALL someFunction works?

The CALL instruction performs the following actions:

1. Pushes the return address onto the stack.
2. Updates the IP to the address of the subroutine.
3. Upon encountering the RET instruction in the subroutine, pops the return address off the stack.
4. Updates the IP to the return address, resuming execution immediately after the CALL.
5. What is the difference between a near and far procedure.

 **Near Procedures**: Used within the same segment. Faster and use less stack space. CALL and RET instructions handle only the offset part of the address.

 **Far Procedures**: Used for subroutines in different segments. More flexible but slower and use more stack space. CALL and RET instructions handle both segment and offset parts of the address.

1. How do functions help make a program manageable?

Functions help make a program manageable by encapsulating specific tasks into reusable, modular units, and by promoting code organization and readability, making it easier to debug and maintain.

**Critical Analysis /Conclusion**

|  |
| --- |
|  |

|  |  |  |  |
| --- | --- | --- | --- |
| **Performance (15Marks)** | | **Viva (5 Marks)** | **Total/20** |
| **Pre-Lab Exercise** | **/3** |  |  |
| **Performance** | **/4** |  |  |
| **Results** | **/3** |
| **Critical Analysis** | **/2** |
| **Post Lab Exercise** | **/3** |
| **Comments** | | | |