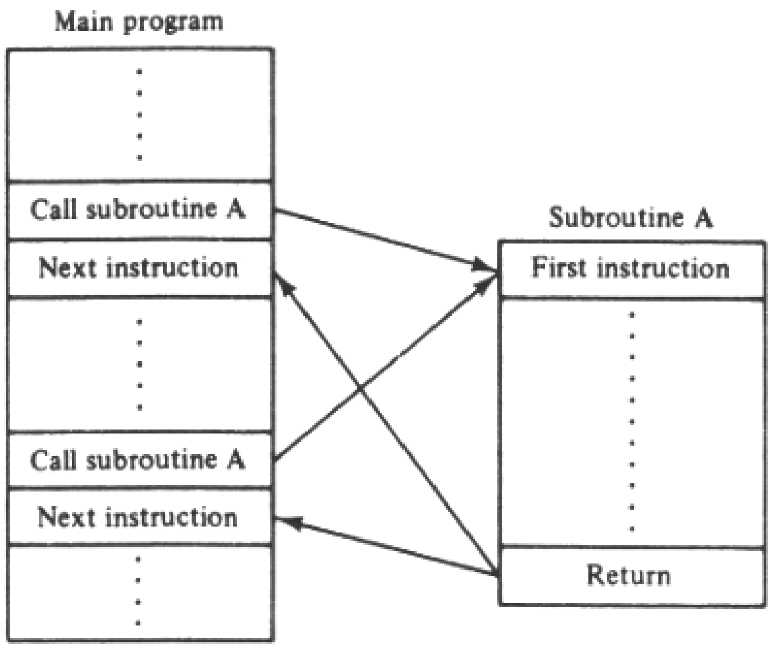
8- SUBROUTINES AND SUBROUTINE – HANDLING INSTRUCTIONS:

A subroutine is a special segment of program that can be called for execution from any point in a program. Figure (a) illustrates the concept of a subroutine. Here we see a program structure where one part of the program is called the **main program**. In addition to this, we find a group of instructions attached to the main program, known as a subroutine.

The subroutine is written to provide a function that must be performed at various points in the main program. Instead of including this piece of code in the main program each time the function is needed, it is put into the program just once as a subroutine. An assembly language subroutine is also referred to as *a procedure*.



(a)

Wherever the function must be performed, a single instruction is inserted into the main body of the program to **"call"** the subroutine. Remember that the logical address CS:IP identifies the next instruction to be fetched for execution. Thus, to branch to a subroutine that starts elsewhere in memory, the value in either ” IP “ or “CS and IP” must be modified.

After executing the subroutine, we want to return control to the instruction that immediately follows the one called the subroutine. To facilitate this return operation, return linkage is saved when the call takes place. That is, the original value of “IP” or “ IP and CS” must be preserved.
A return instruction is included at the end of the subroutine to initiate the *return sequence* to the main program environment. In this way, program execution resumes in the main program at the point where it left off due to the occurrence of the subroutine call.
The instructions provided to transfer control from the main program to a subroutine and return control back to the main program are called subroutine-handling instructions.

CALL and RET Instructions

There are two basic instructions in the instruction set of the 8086 for subroutine handling: the *call* (**CALL**) and *return* (**RET**) instructions. **CALL** instruction is used to call the subroutine. **RET** instruction must be included at the end of the subroutine to initiate the return sequence to the main program environment.

Call instruction:

Just like the **JMP** instruction, **CALL** allows implementation of two types of operations: the **intrasegment call** and the **intersegment call**.
The CALL instruction is shown in Fig.(b), and its allowed operand variations are shown in Fig.(c).

Mnemonic	Meaning	Format	Operation	Flags Affected
CALL	Subroutine call	CALL operand	Execution continues from the address of the subroutine specified by the operand. Information required to return back to the main program such as IP and CS are saved on the stack.	None

(b)

Operand
Near-proc
Far-proc
Mem 16
Reg 16
Mem 32

(c)

It is the operand that initiates either an intersegment or an intrasegment call. The operands **Near-proc, Mem.l6,** and **Reg.l6** all specify **intrasegment** calls to a subroutine. In all three cases, execution of the instruction causes the contents of **IP** to be saved on the stack. The saved value of **IP** is the offset address of the instruction that **immediately follows** the CALL instruction. After saving this return address, a new **16-bit value**, which is specified by the instruction's operand and corresponds to the storage location of the **first instruction in the subroutine**, is loaded into **IP** types of operands represent different ways of specifying a new value of **IP**. Using **a Near-proc , Mem.l6, and Reg.l6** operand, and the subroutine is located in the same code segment.

Ex:

CALL 1234h

CALL BX

CALL [BX]

- CALL 1234H

Here 1234H identifies the starting address of the subroutine. It is encoded as the difference between l234H and the updated value of IP-that is, the IP for the instruction following the CALL instruction.

- CALL BX

When this instruction is executed, the contents of BX are loaded into IP and execution continues with the subroutine starting at the physical address derived from the current CS and the new value of IP.

- CALL [BX]

By using various addressing modes of the 8086, an operand that resides in memory is used as the call to offset address. This represents a Meml6 type of operand. This instruction has its subroutine offset address at the memory location whose physical address is derived from the contents of DS and BX. The value stored at this memory location is loaded into IP Again the current contents of CS and the new value in IP point to the first instruction of the subroutine.

The other type of CALL instruction, the **intersegment call**, permits the subroutine to reside in another code segment. It corresponds to the **Far-proc** and **Mem32** operands. These operands specify both a new offset address
for IP and a new segment address for CS. In both cases, execution of the call instruction causes the contents of the CS and IP registers to be saved on the stack, and the new values are loaded into IP and CS. The saved values of CS and IP permit return to the main program from a different code segment.

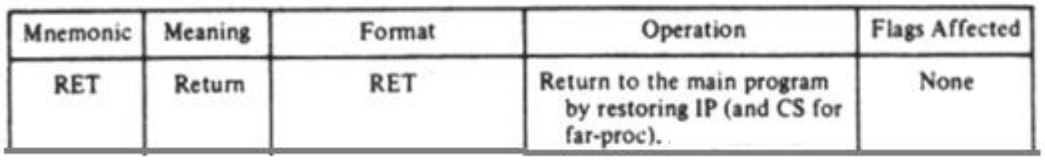
Far-proc represents a 32-bit immediate operand that is stored in the four bytes that follow the opcode of the call instruction in program memory. These two words are loaded directly from code segment memory into IP and CS with execution of the CALL instruction.
On the other hand, when the operand is Mem32, the pointer for the subroutine is stored as four consecutive bytes in data memory. The location of the first byte of the pointer can be specified indirectly by one of the 8086's memory addressing modes.

CALL DWORD [DI]

Here the physical address of the first byte of the 4-byte pointer in memory is derived from the contents of DS and DI.

<u>RET instruction</u>

Every subroutine must end by executing an instruction that returns control to the main program. This is the return (**RET**) instruction.



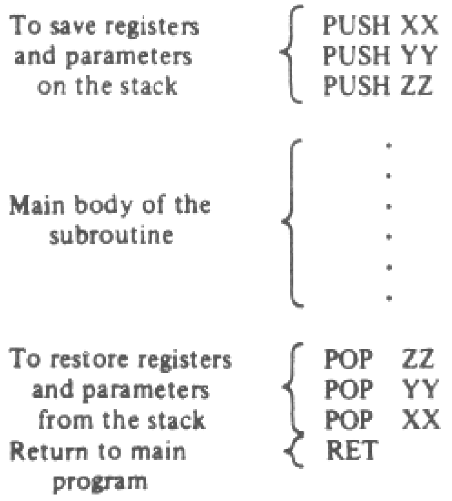
Mnemonic	Meaning	Format	Operation	Flags Affected
RET	Return	RET	Return to the main program by restoring IP (and CS for far-proc).	None

Note that its execution causes the value of **IP** or both the values of **IP and CS** that were saved on the stack to be returned back to their corresponding registers.

In general, an intrasegment return results from an intrasegment call and an intersegment return results from an intersegment call. In this way, program control is returned to the instruction that immediately follows the call instruction in program memory.

PUSH AND POP INSTRUCTION

To save the contents of certain registers or some other main program parameters. Pushing them onto the stack saves these values. Typically, these data correspond to registers and memory locations that are used by the subroutine. In this way, their original contents are kept intact in the stack segment of memory during the execution of the subroutine. Before a return to the main program takes place, the saved registers and main program parameters are restored. Popping the saved values from the stack back into their original locations does this. Thus, a typical structure of a subroutine is that shown in Fig. below:


To save registers and parameters on the stack
Main body of the subroutine
To restore registers and parameters from the stack
Return to main program

```
PUSH XX
PUSH YY
PUSH ZZ
   .
   .
   .
   .
   .
   .
POP   ZZ
POP   YY
POP   XX
RET
```

The instruction that is used to save parameters on the stack is the *push* (**PUSH**) instruction, and that used to retrieve them is the *pop* (**POP**) instruction.

Note in Fig.(a) and (b) that the standard **PUSH** and **POP** instructions can be written with a **general-purpose register**, a **segment register** (**excluding CS**), or a storage **location in memory** as their operand.

Mnemonic	Meaning	Format	Operation	Flags Affected
PUSH	Push word onto stack	PUSH S	((SP)) ← (S) (SP)← (SP)-2	None
POP	Pop word off stack	POP D	(D) ← ((SP)) (SP)←(SP)+2	None

(a)

Operand (S or D)
Register Seg-reg (CS illegal) Memory

(b)

Execution of a PUSH instruction causes the data corresponding to the operand to be pushed onto the top of the stack. For instance, if the instruction is

PUSH AX

the result is as follows:

$$((SP) - 1) \leftarrow (AH)$$
$$((SP) - 2) \leftarrow (AL)$$
$$(SP) \leftarrow (SP) - 2$$

This shows that the two bytes of AX are saved in the stack part of memory and the stack pointer is decremented by two so that it points to the new top of the stack.

On the other hand, if the instruction is

POP AX

its execution results in

$(AL) \leftarrow ((SP))$

$(AH) \leftarrow ((SP) + 1)$

$(SP) \leftarrow (SP) + 2$

In this manner, the saved contents of AX are restored in the register.

Ex:

Write a procedure named ***Square*** that squares the contents of **BL** and places the result in **BX**.

Solution:

```
Square:  PUSH AX
         MOV AL, BL
         MUL BL
         MOV BX, AX
         POP AX
         RET
```

To square the number in BL, we use the 8-bit multiply instruction, MUL. This instruction requires the use of register AX for its operation. Therefore, at entry of the procedure, we must save the value currently held in AX. Pushing its contents to the stack with the instruction does this

```
PUSH AX
```

Now we load AX with the contents of BL using the instruction

```
MOV AL, BL
```

To square the contents of AL, we use the instruction

```
MUL BL
```

Which multiplies the contents of AL with the contents of BL and places the result in AX. The result is the square of the original contents of BL. To place the result in BX, use the instruction

```
MOV BX, AX
```

This completes the square operation; but before returning to the main part of the program, the original contents of AX that are saved on the stack are restored with the pop instruction

POP AX

Then a return instruction is used to pass control back to the main program:

RET

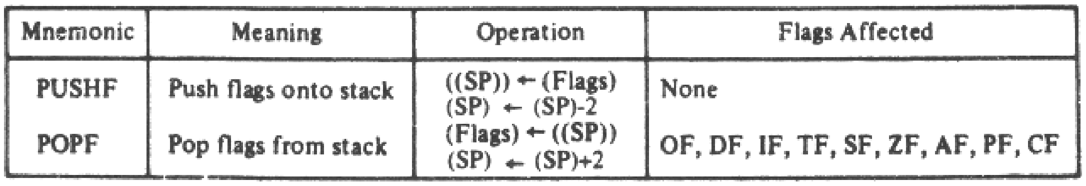
Ex: write a program that computes **$y = (AL)^2 + (AH)^2 + (DL)^2$** places the result in **CX**. Make use of the **SQUARE subroutine** defined in the previous example. (Assume result y doesn't exceed 16 bit)

Solution:

```
MOV  CX, 0000H
MOV BL,AL
CALL  Square
ADD  CX, BX
MOV  BL,AH
CALL  Square
ADD  CX, BX
MOV  BL,DL
CALL  Square
ADD  CX, BX
HLT
```

PUSHF & POPF instructions:

Use to save the contents of the flag register, and if we save them, we will later have to restore them. These operations can be accomplished with *push flags* (**PUSHF**) and *pop flags* (**POPF**) instructions, respectively, as shown in Fig. below:



Mnemonic	Meaning	Operation	Flags Affected
PUSHF	Push flags onto stack	((SP)) ← (Flags) (SP) ← (SP)-2	None
POPF	Pop flags from stack	(Flags) ← ((SP)) (SP) ← (SP)+2	OF, DF, IF, TF, SF, ZF, AF, PF, CF

Note that **PUSHF** saves the contents of the flag register on the top of the stack. On the other hand, **POPF** returns the flags from the top of the stack to the flag register.