**PROC** is a statement used to indicate the beginning of a procedure or subroutine. **ENDP** indicates the end of the procedure. Syntax:

ProcedureName PROC Attribute

. . .

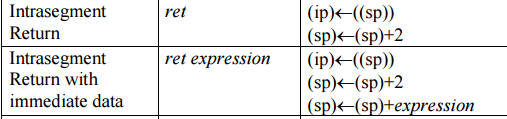
RET

ProcedureName ENDP

ProcedureName may be any valid identifier. Attribute is NEAR if the Procedure is in the same code segment as the calling program; or FAR if in a different code segment. To call any procedure, we write:

CALL ProcedureName

When a procedure is called, the contents of **IP** register are pushed onto stack and **IP** is set to the first instruction of called procedure. When **RET** instruction is executed **IP** and **SP** are updated as shown in table given below. If an optional pop value has been specified, **RET** instruction add that value to SP. This feature may be used to discard parameters pushed onto the stack before the execution of the **CALL** instruction.

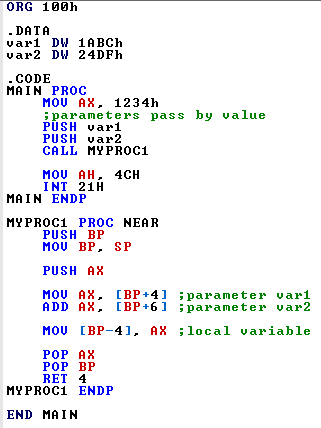


**Procedure Parameters & Variables:**

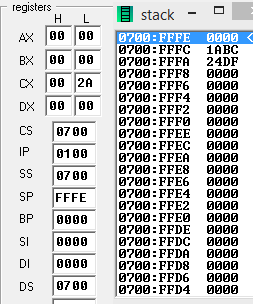
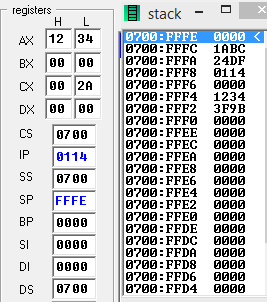
High-level languages pass parameters by placing them on the stack. Parameter on the stack can be passed by Value or by Reference. *Passing by Value* means to put a copy of each parameter value on the stack. *Passing by Reference* means to put a copy of each parameter offset on the stack. Parameters on the stack can then be accessed by procedures by using displacements or a stack-frame structure. To access parameters from the stack, a marker to the stack frame is required. **BP** is commonly used by procedures, but need to be pushed before. Parameters are accessed at **[BP+Disp.]** after a push of **BP** and then copying **SP** to **BP**.

Procedures often need local memory space. The stack area can be used to allocate space dynamically for the procedure with the space de-allocated when the procedure concludes. To allocate space for local variables, subtract from SP the number of bytes needed after setting-up the stack frame marker (**BP**). Then, local variables can be accessed at **[BP-number]** and the parameters at **[BP+number]**. Local variables are released by moving **BP** back to **SP** (MOV SP, BP).

**Example 1– Passing parameters by value and using local variables:**



**Before Execution: After Execution:**

**Recursion:**

* A recursive procedure is a procedure that calls itself.
* Functions can be written in two ways:
  + Iterative: keep repeating until a task is done *e.g.,* loop counter reaches limit
  + Recursive: Solve a large problem by breaking it up into smaller and smaller pieces until you can solve it; combine the results.
* A recursive function must have two properties:
  + There must be a certain (base) criteria for which function doesn’t call itself.
  + Each time function does call itself (directly or indirectly), it must closer to the base criteria.

**Example 1– Compute Factorial Using Recursion:**

PROCEDURE FACTORIAL

IF N = 1

THEN

RESULT=1

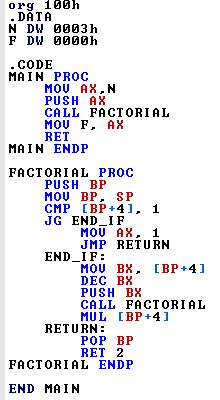
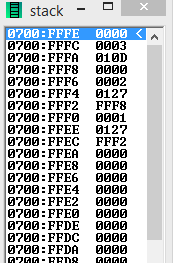
ELSE

CALL FACTORIAL

RESULT = N \* RESULT

END\_IF

RETURN

** **

**Example 2– Product Fibonacci**

Write a recursive subroutine to find the product-fibonacci (PF(n)) of a number stored in AX register. The recursive formula is **PF (n) = PF (n-1) \* PF (n-2)** and base case is PF(0)=1 and PF(1)=2. For example, PF(2)=PF(1)\*PF(2), PF(3)=PF(2)\*PF(1)=PF(1)\*PF(0)\*PF(1)=2\*1\*2=4, so on and so far.

**Solution:**

MAIN PROC

MOV AX, 3

PUSH AX

CALL PF

RET

MAIN ENDP

PF PROC

**PUSH BP**

**MOV BP, SP**

PUSH BX

PUSH CX

MOV BX, [BP+4]

BASECASE0:

CMP BX, 0

JNE BASECASE1

MOV AX, 1

JMP EXITPF

BASECASE1:

CMP BX, 1

JNE RECURSIVE

MOV AX, 2

JMP EXITPF

RECURSIVE:

DEC BX

PUSH BX

**CALL PF**

MOV CX, AX

DEC BX

PUSH BX

**CALL PF**

MUL CX

EXITPF:

POP CX

POP BX

**POP BP**

**RET 2**

PF ENDP