# Lecture #1: Nested Subroutines

By

Ayman Elnaggar

# Objectives

- Understand the need for modular design
- Know about procedures (Subroutines) and how they are structured and used
- Understand the structure of the Stack and know how procedures efficiently access it
- Know how to write efficient assembly programs that define and call procedures
- Understand how Nested Subroutines work
- Be familiar with some Performance considerations

# Modular Design

- Modular design is one of the cornerstones of structured programming.
- A modular program contains blocks of code (subroutines) that can be used in other sections of the program or you can use vendor-supplied *library* of useful subroutines
- Given the rising costs of software development, modular design will become more important as they are easy to debug, maintain, reuse, and divide the task among different design groups.
- Modules have different meanings to different people, herein you can assume that the terms module, ***subroutine***, ***procedure***, and ***function*** are all synonymous.

# Program Design using Procedures

Program Design involves the Following:

- Break large tasks into smaller ones
- Use a hierarchical structure based on procedure calls
- Test individual procedures separately

# What is the Procedure ?

# Procedures

- A **procedure** is a logically self-contained unit of code
  - Called sometimes a **function**, **subprogram**, or **subroutine**
  - Receives a **list of parameters**, also called **arguments**
  - Performs computation and might **return results (*function*)**
  - Plays an important role in modular program development
- Example of a procedure (called function) in C language

```
int sumof ( int x,int y,int z ) {
    int temp;
    temp = x + y + z;
    return temp;
}
```

Formal parameter list

Result type

Return function result

- The above function *sumof* can be called as follows:

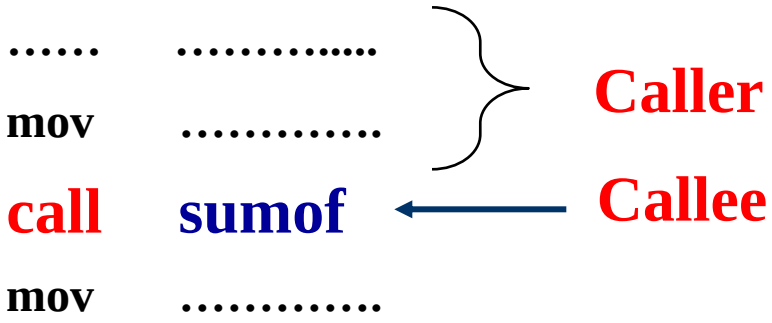
```
sum = sumof( num1,num2,num3 );
```

Actual parameter list

# How to access a procedure in Assembly Language

- To invoke a procedure, the **call** instruction is used
- The **call** instruction has the following format

  **call procedure_name**
- Example on calling the procedure **sumof**
  - parameters should have been saved in a known location before calling procedure **sumof** . Also the result should have been saved in a known location after **sumof** finishes




- **call sumof** will call the procedure **sumof**

# How to access a procedure in Assembly Language

- Each time a program calls a procedure, the same set of operations is repeated:
  - Pass parameters
  - Preserve return address, and caller register state
  - Store callee local variables
  - Pass results
  - Release any used memory

# Assigned Roles ??

1. Who passes the parameters and how
2. Who saves the return address
3. Who preserves caller state (registers)
4. How does callee access parameters
5. How does callee save and access local variables
6. How does caller access return value(s);
7. How do we deal with nested procedures
8. Who should clean the stack

# How We Passes Parameters

- Passing Parameters in Registers
  - Pros: Convenient, easier to use, and faster to access
  - Cons: Only few parameters can be passed
    - A small number of registers are available
    - Often these registers are used and need to be saved on the stack
    - Pushing register values on stack negates their advantage. **Why ?**

# How We Passes Parameters

- Passing Parameters on the Stack
  - Pros: Many parameters can be passed
    - Large data structures and arrays can be passed
    - Nesting Subroutines are easy to implement
  - Cons: Accessing parameters is not simple
    - More overhead and slower access to parameters
- However, it is the preferred choice by most programming languages

# What is a Stack?

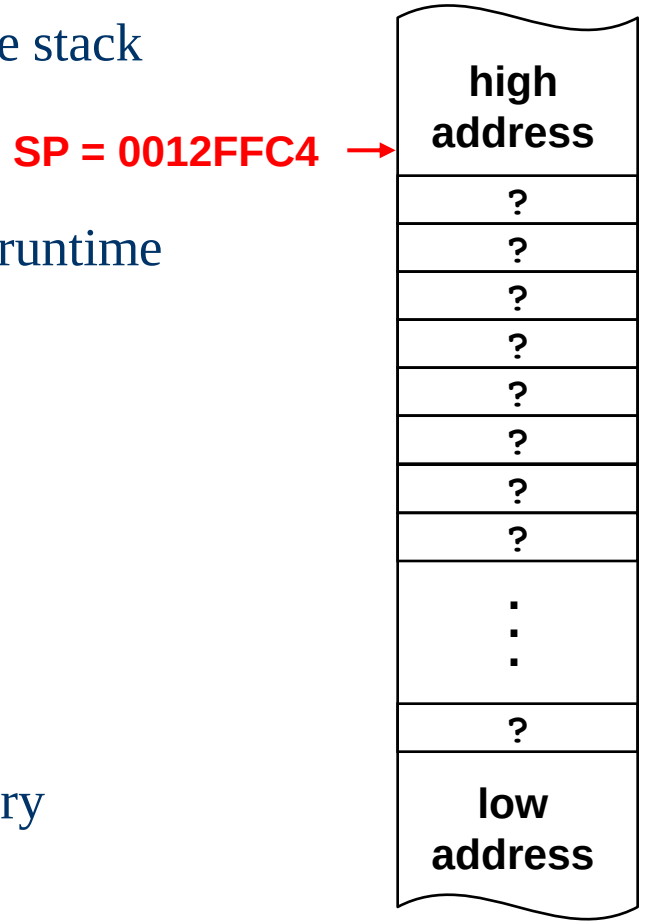
- Stack is a **Last-In-First-Out** (LIFO) data structure
  - Analogous to a stack of plates in a cafeteria
  - Plate on **Top of Stack** is directly accessible
- Two basic stack operations
  - **Push**: inserts a new element on top of the stack
  - **Pop**: deletes top element from the stack
- View the stack as a linear array of elements
  - Insertion and deletion is restricted to one end of array
- Stack has a maximum capacity
  - When stack is **full**, no element can be pushed
  - When stack is **empty**, no element can be popped

# How to Access Stack

- **stack:** array of consecutive memory locations
- Managed by the processor using two registers
  - Base Pointer **bP**
    - Points to the entry point to the procedure
  - Stack Pointer register **SP**
    - Points to the last data pushed to (top of) the stack
- Stack grows **downward** toward lower memory addresses

# Runtime Stack Allocation

- runtime stack
  - Operating system allocates memory for the stack
  - Runtime stack is initially empty
  - The stack size can change dynamically at runtime
- Stack pointer **SP**
  - **SP** is initialized by the operating system
- The stack grows **downwards**
  - The memory below **SP** is free
  - **SP** is decremented to allocate stack memory



# How to Access Stack

- Two basic stack instructions:
  - **push source**
  - **pop  destination**
- **Source** can be:
  - A register
  - Immediate value
- **Destination** can be also:
  - A register

# Push Instruction

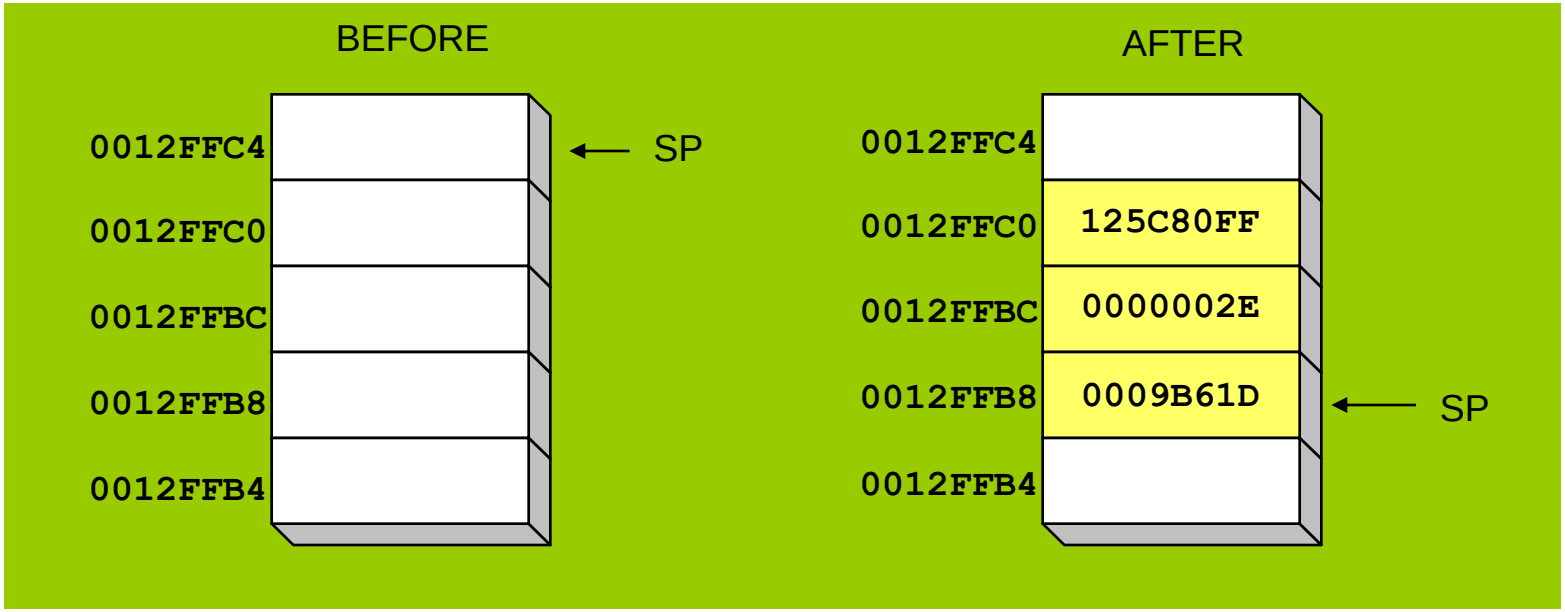
- **Push source (normally a 32-bit)**
  - **SP** is first decremented by **4**
    - **SP = SP – 4** (stack grows by 4 bytes)
  - 32-bit source is then copied onto the stack at the new **SP**
    - **[SP] = source32**
- Operating system puts a limit on the stack capacity
  - **Push** can cause a **Stack Overflow** (stack **cannot grow**)

# Examples on the Push Instruction

- Suppose we execute:
  - PUSH R1 ; R1 =125C80FFh
  - PUSH R2 ; R2 = 2Eh
  - PUSH R3 ; R3 = 9B61Dh

The stack grows downwards

The area below SP is free



# Pop Instruction

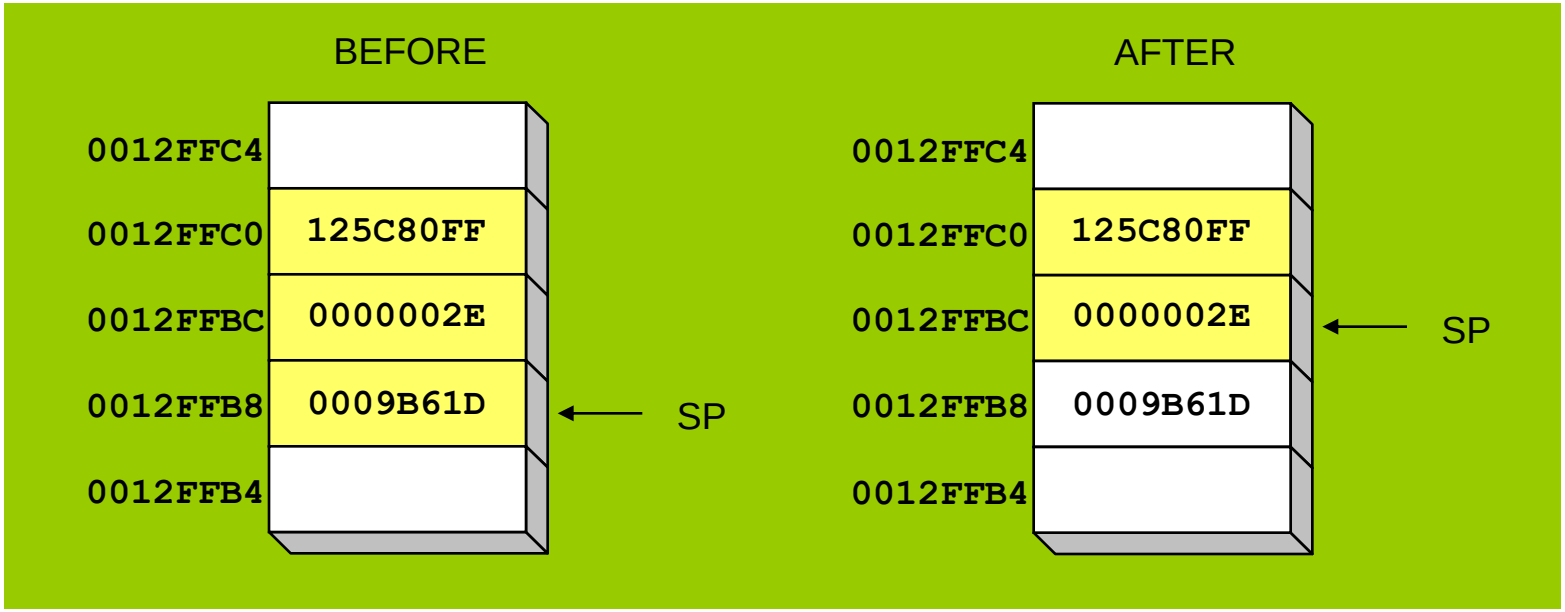
- **Pop dest**
  - 32-bit doubleword at SP is first copied into dest
    - **dest = [SP]**
  - SP is then incremented by 4
    - **SP = SP + 4** (stack shrinks by 4 bytes)
- Popping from an empty stack causes a **stack underflow**

# Examples on the Pop Instruction

- Suppose we execute:
  - POP R3 ; R3 = 9B61Dh

The stack shrinks upwards

The area at & above SP is allocated



# Uses of the Runtime Stack

- Runtime Stack can be utilized for
  - Temporary storage of data and registers
  - Transfer of program control in procedures and interrupts
  - Parameter passing during a procedure call
  - Allocating local variables used inside procedures

- Stack can be used as temporary storage of data
  - Example: exchanging two variables in a data segment

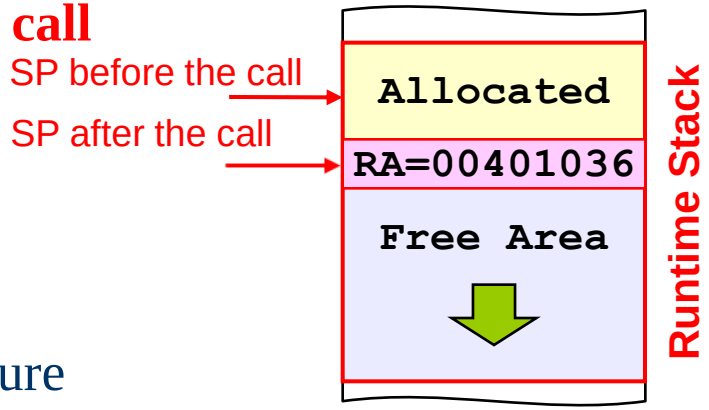
```
push var1; var1 is pushed
push var2; var2 is pushed
pop  var1; var1 = var2 on stack
pop  var2; var2 = var1 on stack
```

# How a Procedure Call / Return Works

- How does a procedure know where to return?
  - There can be multiple calls to same procedure in a program
  - Procedure has to return differently for different calls
- It knows by saving the **return address** (RA) on the stack
  - This is the **address of next instruction** after **call**
- The **call** instruction does the following
  - Pushes the **return address** on the stack

    **SP = SP – 4; [SP] = RA**
  - Jumps into the first instruction inside procedure

    **PC = procedure address**
- The **ret** (return) instruction does the following
  - Pops return address from stack
  - Jumps to return address: **PC = [SP]; SP = SP + 4**

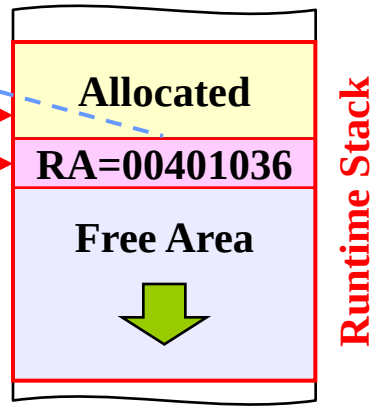



Just before returning from a procedure Make sure SP is pointing at RA

# How a Procedure Call / Return Works

| Address | Machine Code | Assembly Language |
|---|---|---|
| | | .CODE |
| | | main PROC |
| ..... | | |
| ..... | | |
| ..... | | |
| 00401031 | E8 0000004B | call sumof |
| 00401036 | A3 0040500C | mov ..... |
| . . . | . . . | . . . |
| | | exit |
| | | main ENDP |
| | | sumof PROC |
| 00401081 | 03 C3 | ..... |
| 00401083 | 03 C1 | ..... |
| 00401085 | C3 | ret |
| | | sumof ENDP |
| | | END main |

| Before Call |
|---|
| SP = 0012FFC4 |
| After Call |
| SP = 0012FFC0 |
| After Ret (Return) |
| SP = 0012FFC4 |



# 1. Who Passes the Parameters?

- When a subprogram is invoked, the caller code and the callee must agree on how to pass data between them. High-level languages have standard ways to pass data known as calling conventions.
- Each programming language supports a calling convention that allows one to create subprograms that can be called at any point of a program safely even inside the subprogram itself (nested subroutines).

# Calling Conventions

- Programming language dependent
- Defines the standard for passing arguments
- Caller and Callee need to agree (such as who cleans the stack)
- Enforced by compiler
- Important when using 3rd party libraries
- Different styles ↔ different advantages

# Calling Convention

- C
  - procedure arguments pushed on stack in reverse order (right to left)
  - Caller cleans up the stack (variable number of parameters such as **printf**)
  - _name (for example, **_AddTwo**)
- PASCAL
  - arguments pushed in forward order (left to right)
  - Callee cleans up the stack

# Calling (HLL to LL) Conventions

- It is often difficult for the same subroutine to be called by different languages.
- A subroutine called by Pascal, for instance, expects its parameters to be in a different order from the subroutine called by C.
- One principle seems to be universal: when a HLL calls a subroutine, it pushes its arguments on the stack before executing a CALL instruction.
- But beyond this basic principle, languages vary in their calling conventions.

# Parameter Passing in Assembly

- Parameter passing in assembly language is different
  - More complicated than that used in a high-level language
- In assembly language
  - Place all required parameters in an accessible storage area
  - Then call the procedure

# Who Preserve Registers

- Caller: saves and frees registers that it uses
  - Calling Procedures have to know the details of the called one
  - Registers are repeatedly saved before procedure call and restored after return
- Callee: **preferred method** for modular code
  - Register preservation is done in one place only (inside procedure)

# So How Do Procedures uses the Stack ?

- Consider the following max procedure

```
int max ( int x, int y, int z ) {
        int temp = x;
        if (y > temp) temp = y;
        if (z > temp) temp = z;
        return temp;
}
```

Calling procedure: **mx = max(num1, num2, num3)**

**Stack Parameters (Caller Job)**

```
push num3
push num2
push num1
call max
mov  mx, R1
```

**Reverse Order**

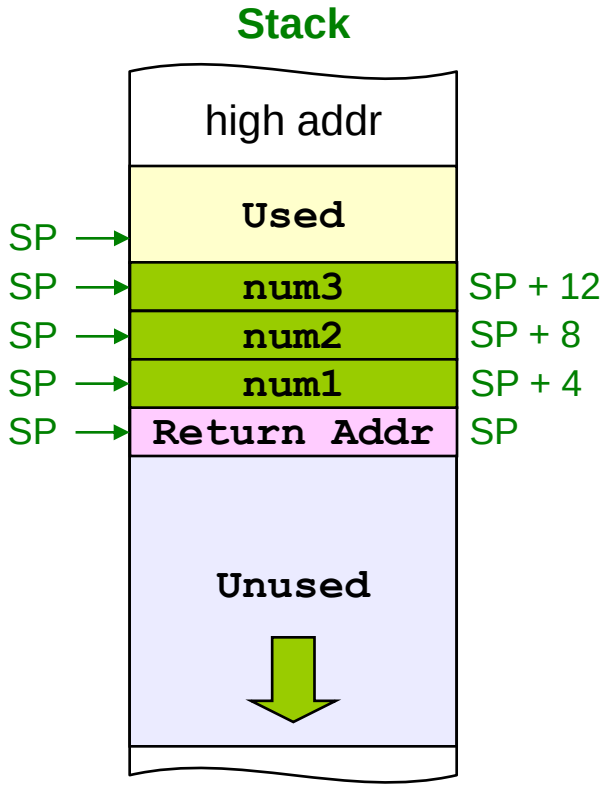
# Caller Pass Parameters on the Stack

- Calling procedure pushes parameters on the stack
- Procedure max receives parameters on the stack
  - Parameters are pushed in reverse order
  - Parameters are located relative to SP

Passing Parameters on the stack

```
push num3
push num2
push num1
call max
mov  mx,R1
add  SP,12
```

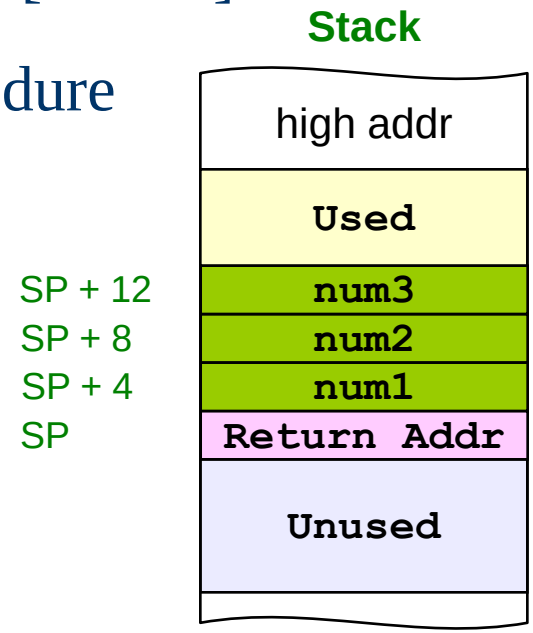
```
max PROC
    mov R1,[SP+4]
    cmp R1,[SP+8]
    jge @1
    mov R1,[SP+8]
@1: cmp R1,[SP+12]
    jge @2
    mov R1,[SP+12]
@2: ret
max ENDP
```



# Accessing Parameters on the Stack

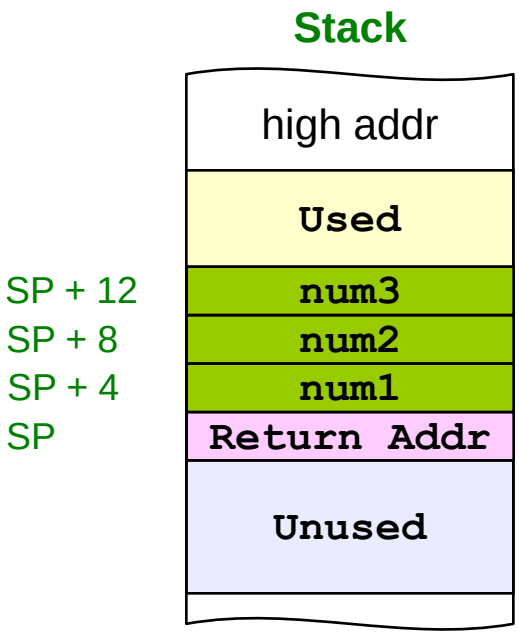
- When parameters are passed on the stack
  - Parameter values appear after the return address
- We can use SP to access the parameter values
  - [SP+4] for num1, [SP+8] for num2, and [SP+12] for num3
  - However, SP might change inside procedure

(if the procedure saved used Registers or local variables as will see later)



# Accessing Parameters on the Stack

- A better choice is to use the BP register
  - BP is called the base pointer
  - BP does not change during procedure
  - Start by copying SP into BP
  - Use BP to locate parameters
  - BP must be restored when a procedure returns



# How we use Base Pointer Register

- **Use the Base (frame) pointer instead.**
- **BP** is often called the base pointer or frame pointer because it holds the base address of the stack frame.
- **BP** does not change value during the procedure.
- A procedure can then safely access stack parameters using constant offsets from **BP**.
  - Example: **[bp + 8]**
- **BP** must be preserved before it is used, incase we have nested procedures or recursion.
- **BP** must be restored to its original value when a procedure returns.

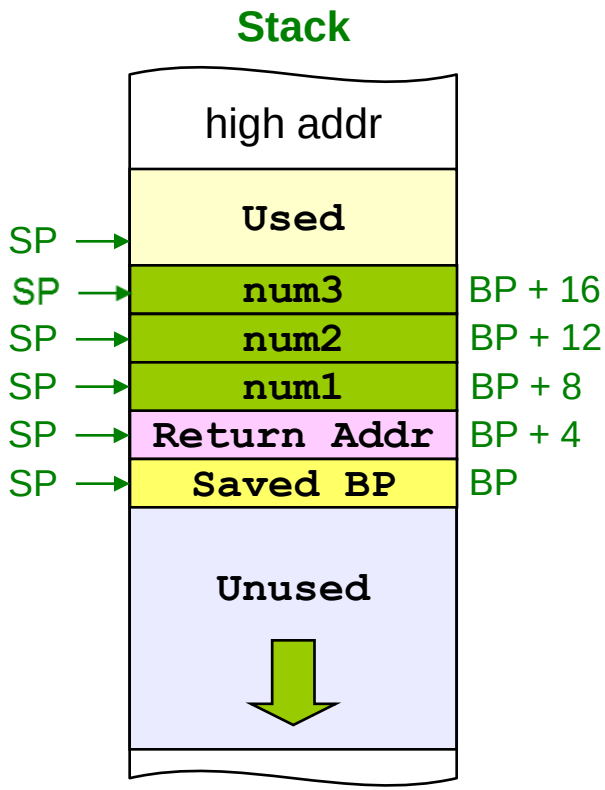
# How Callee access Parameters Using Base Pointer Register

- BP is used to locate parameters on the stack
- Like any other register, BP must be saved before use

Passing Parameters on the stack

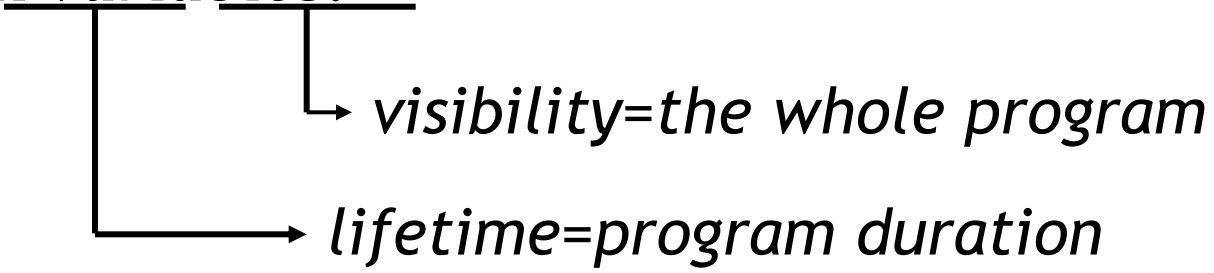
```
push num3
push num2
push num1
call max
mov  mx,R1
add  SP,12
```

```
max PROC
    push BP
    mov  BP, SP
    mov R1,[SP+8]
    cmp R1,[SP+12]
    jge @1
    mov R1,[SP+12]
@1: cmp R1,[SP+16]
    jge @2
    mov R1,[SP+16]
@2: pop BP
    ret
max ENDP
```



# How Callee access Local variables

- The variables defined in the data segment can be taken as *static global variables*.




- A local variable Come into existence when the procedure is invoked and Disappear when the procedure terminates
- Advantages of local variables:
  - Restricted access: easy to debug, less error prone
  - Efficient memory usage
  - Same names can be used in two different procedures
  - Essential for recursion

# Local Procedure Variables

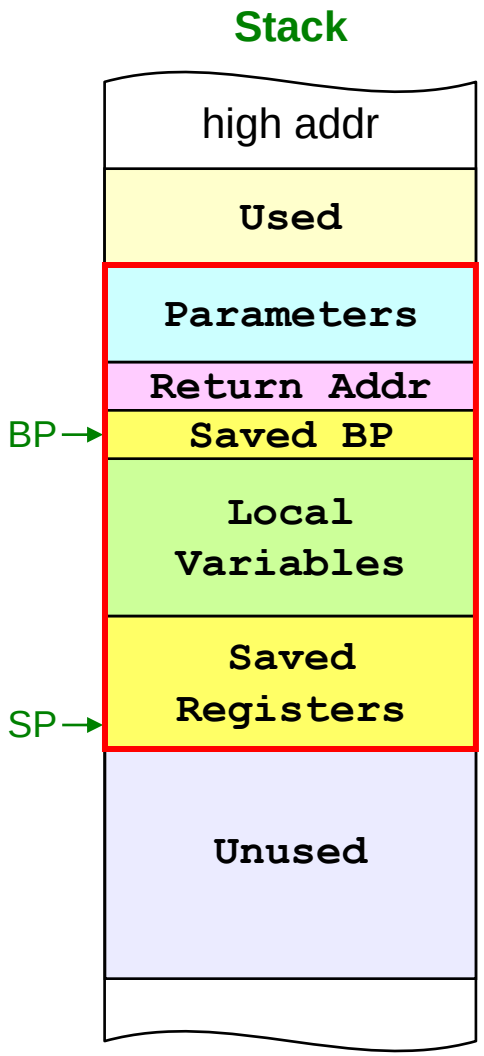
- Local procedure variables are dynamic in nature
- Cannot reserve space for local variables in data segment
  - Because such space allocation is static (wasted!! Why?)
    - Remains active even after returning from the procedure call
  - Also because it does not work with recursive procedures
- Local variables can be stored in registers or on the stack
  - Registers are best used for local variables when ...
    - Variables are small in size and frequently used (e.g. loop counter)
  - Local variables are stored on the stack when …
    - They are large in size (e.g. arrays) or cannot fit in registers
    - Recursive calls are made by the procedure

# Stack Frame

- Also known as an *activation record*
- Area of the stack set aside for a procedure's return address, passed parameters, saved registers, and local variables
- Created by the following steps:
  - Calling program pushes arguments on the stack and calls the procedure.
  - The called procedure pushes BP on the stack, and sets BP to SP.
  - If local variables are needed, a constant is subtracted from SP to make room on the stack.

# How the Stack Frame looks like after all

- For each procedure call
  - Caller pushes parameters on the stack
  - Return address is saved by CALL instruction
  - Procedure saves BP and sets BP to SP
  - Local variables are pushed on the stack
  - Registers are saved by the procedure
- Stack Frame
  - Area on the stack reserved for …
    - Parameters
    - Return address
    - Saved registers
    - Local variables
  - Designed specifically for each procedure



# Who Should Clean up the Stack?

- When returning from a procedure call …
  - Who should remove parameters and clean up the stack?
- Clean-up can be done by the calling procedure (Caller)
  - **First instruction in the caller after the call instruction**
  - **The SP is now pointing to the first parameters in stack**
  - **add SP, 4xn**

  **will clean up stack (where n is the number of parameters previously pushed by the caller_**
- Some efficient compilers might call several procedures and keep track of how many parameters have been pushed. It then uses just one add instruction to clean all left parameters at once.

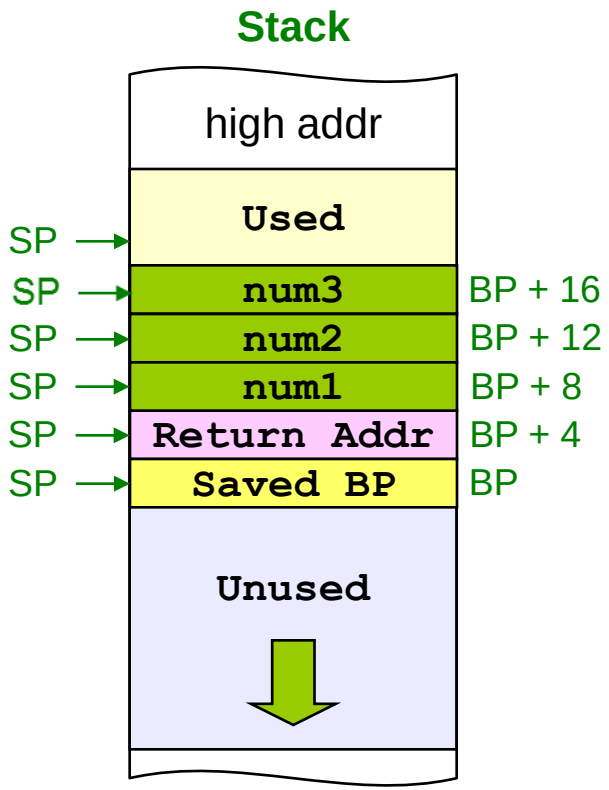
# How Caller Cleans Up the Stack

- BP is used to locate parameters on the stack
- Like any other register, BP must be saved before use

Passing Parameters on the stack

```
push   num3
push   num2
push   num1
call   max
mov    mx,R1
add    SP,12
```

```
max  PROC
     push BP
     mov  BP, SP
     mov  R1,[SP+8]
     cmp  R1,[SP+12]
     jge  @1
     mov  R1,[SP+12]
@1:  cmp  R1,[SP+16]
     jge  @2
     mov  R1,[SP+16]
@2:  pop BP
     ret
max  ENDP
```



# Who Should Clean up the Stack?

- Callee can also clean the stack.
- Useful for recursion
- Return instruction is used to clean up stack
  - **ret *n*** **; *n* is an integer constant**
  - Actions taken
    - SP = [SP]
    - SP = SP + 4+***n***

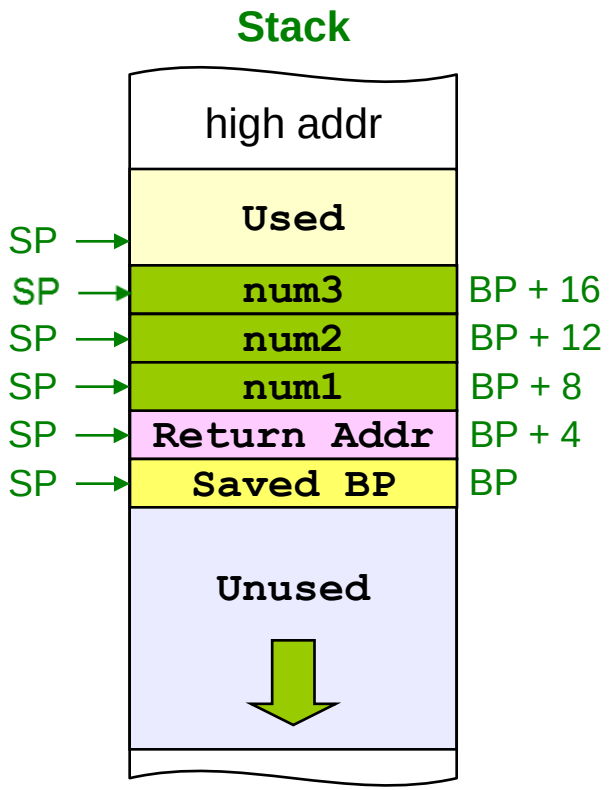
# How Callee Cleans Up the Stack

- BP is used to locate parameters on the stack
- Like any other register, BP must be saved before use

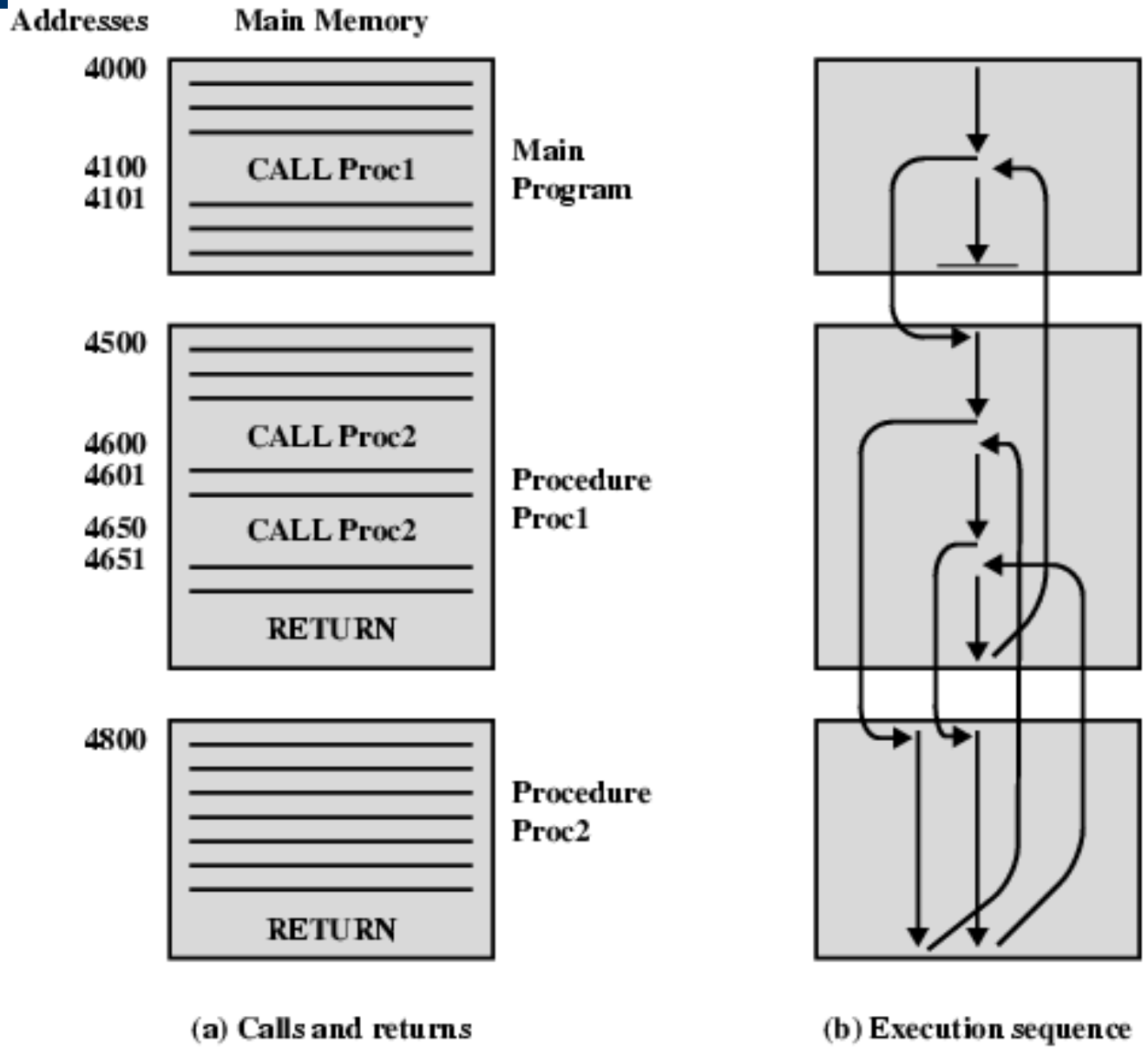
Passing Parameters on the stack

```
push   num3
push   num2
push   num1
call   max
mov    mx,R1
```

```
max  PROC
     push BP
     mov  BP, SP
     mov  R1,[SP+8]
     cmp  R1,[SP+12]
     jge  @1
     mov  R1,[SP+12]
@1:  cmp  R1,[SP+16]
     jge  @2
     mov  R1,[SP+16]
@2:  pop BP
     ret 12
max  ENDP
```

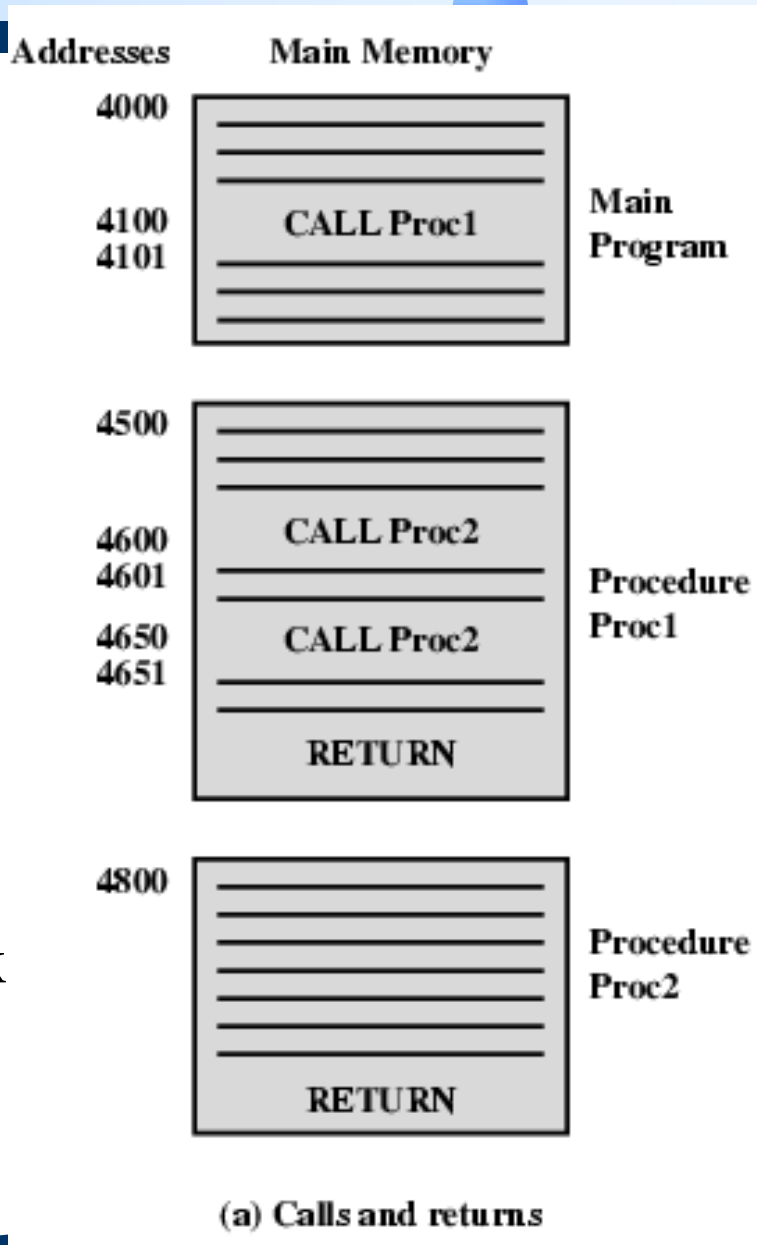


# Nested Procedure Calls



# Nested Procedure Calls

- Let us say the main code called Proc1.
- Proc 1 has two parameters Par1 and Par2 and uses Ro to R3
- Proc 2 has one parameter Par3 and uses R0 and R1
- Proc 2 has been called twice from Proc 1
- Parameters are referenced at offset 8, 12, etc…
- The last instructions in each Procedure restores the old value of the frame pointer and the values of the other registers used, and pops the return address from the stack into the PC



# Nested Procedure Calls

```
Main
……
Push Par2
Push Par1
Call Proc1
Add SP,8
…..
```

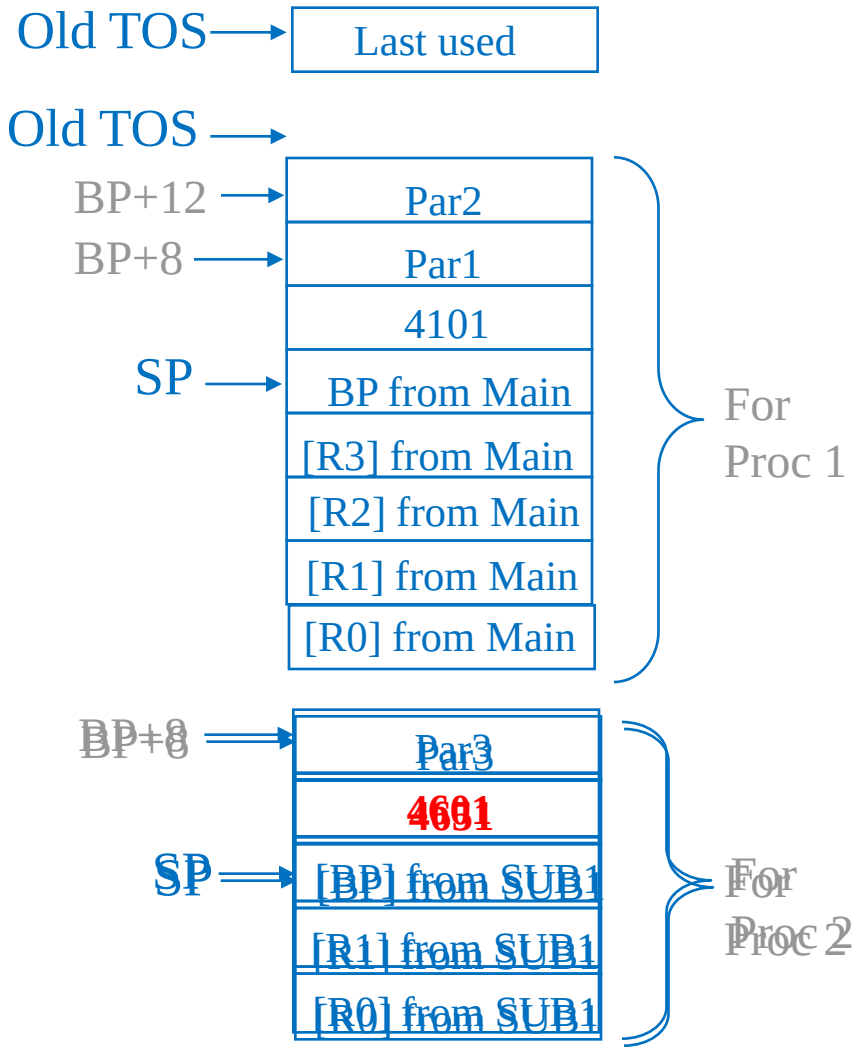
4101

4601

4651

```
Proc1
Push BP
Mov BP,SP
Push R0 –R3
……
Push Par3
Call Proc2
Add SP,4
…..
…..
Push Par3
Call Proc2
Add SP,4
…..
Pop R3-R0
Pop BP
ret
```

```
Proc2
Push BP
Mov BP,SP
Push R0 –R1
……
……
Pop R1-R0
Pop BP
ret
```



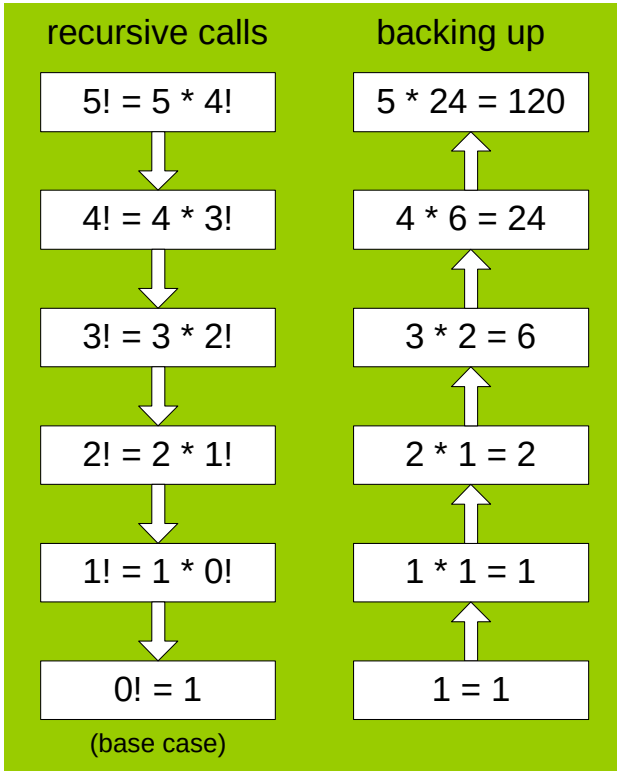
# Recursion: Calculating Factorial (1 of 3)

This function calculates the factorial of integer *n*

A new value of *n* is saved in each stack frame

```
int factorial(int n)
{
      if(n == 0)
        return 1;
      else
        return n * factorial(n-1);
}
```

As each call instance returns, the product it returns is multiplied by the previous value of *n*



# Calculating Factorial (2 of 3)

```
Factorial PROC
        push bp
        mov  bp,sp
        mov  R1,[bp+8]          ; R1 = n
        cmp  R1,0               ; n > 0?
        ja   L1                 ; yes: continue
        mov  R1,1               ; no: return 1
        jmp  L2
L1:     dec  R1
        push R1                 ; Factorial(n-1)
        call Factorial
ReturnFact:
        mov  R2,[bp+8]          ; get n
        mul  R1,R2              ; R1 = R1 * R2
L2:     pop  bp                 ; return R1
        ret  4
Factorial ENDP
```

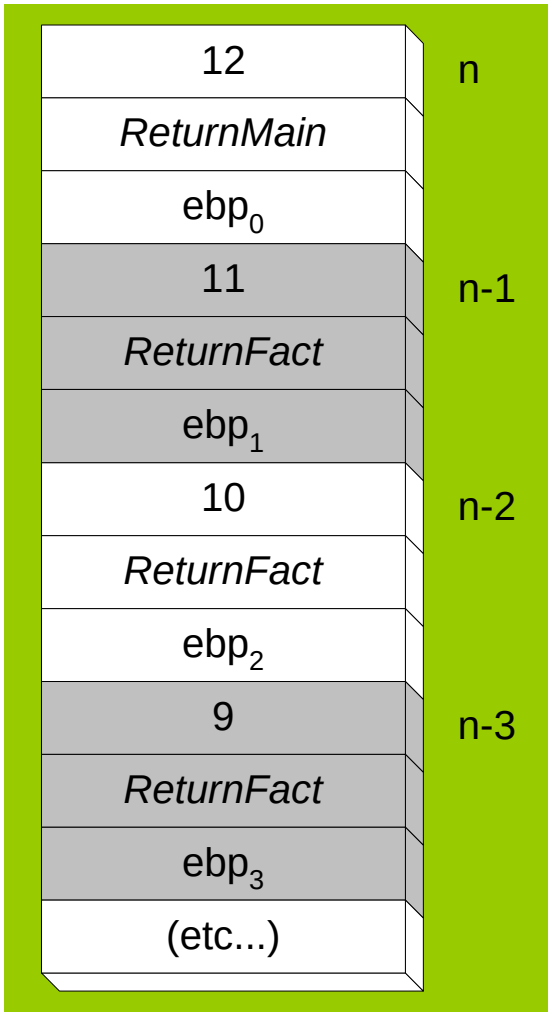
# Calculating Factorial (3 of 3)

Suppose we want to calculate 12!

This diagram shows the first few stack frames created by recursive calls to Factorial

Each recursive call uses 12 bytes of stack space for:

- parameter *n*
- return address
- saved ebp



# Documenting Procedures

- Suggested Documentation for Each Procedure:
  - **Does:** Describe the task accomplished by the procedure
  - **Receives**: Describe the input parameters
  - **Returns**: Describe the values returned by the procedure
  - **Requires**: Optional list of requirements called **preconditions**
- Preconditions
  - Must be satisfied **before** the procedure is called
  - If a procedure is called without its preconditions satisfied, it will probably not produce the expected output

# Example of a Procedure Definition

- The **sumof** procedure receives three integer parameters
  - Assumed to be in bp+4, bp+8, bp+12
  - Computes and returns result in register bp+12

```
;-----------------------------------------------
; Sumof:    Calculates the sum of three integers
; Receives: bp+4, bp+8, bp+12 the three integers in traverse order
; Returns:   sum = bp+12,
; Requires: nothing
;-----------------------------------------------
sumof PROC
  ……..                          ; Body of Proc
  ……..
   ret                          ; return to caller
sumof ENDP
```

- The **ret** instruction returns control to the caller

# Performance Issues

- After showing the overhead of using Procedures for modular design, how is it affecting the performance?
- Procedure In-line expansion
- Trends in recent processors architectures

# Summary

- We have learned about Procedures
- We have learned about Stacks
- We have covered calling conventions
- We learned about nested Procedure calls and Recursion Procedure calls
- We have learned how to write and call Procedures in in Assembly Language.