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 returns results
  - 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;
}
```

Result type

Formal parameter list

Return function result

- The above function sumof can be called as follows:

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

Actual parameter list

Defining a Procedure in Assembly

- Assembler provides two directives to define procedures
  - **PROC** to define name of procedure and mark its beginning
  - **ENDP** to mark end of procedure
- A typical procedure definition is

```
procedure_name   PROC
          . . .
     ; procedure body
          . . .
procedure_name   ENDP
```

- **`procedure_name`** should match in PROC and ENDP

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**: 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 EAX, EBX, and ECX
  - Computes and returns result in register EAX

```
;-----------------------------------------------
; Sumof:    Calculates the sum of three integers
; Receives: EAX, EBX, ECX, the three integers
; Returns:  EAX = sum
; Requires: nothing
;-----------------------------------------------
sumof PROC
   add  EAX, EBX          ; EAX = EAX + second number
   add  EAX, ECX          ; EAX = EAX + third  number
   ret                    ; return to caller
sumof ENDP
```

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

The Call Instruction

- 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**
  - Caller passes actual parameters in EAX, EBX, and ECX
  - Before calling procedure **sumof**

```
mov   EAX, num1      ; pass first  parameter in EAX
mov   EBX, num2      ; pass second parameter in EBX
mov   ECX, num3      ; pass third  parameter in ECX
call  sumof          ; result is in EAX
mov   sum, EAX       ; save result in variable sum
```

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

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
  - Jumps into the first instruction inside procedure
  - **ESP = ESP – 4; [ESP] = RA; EIP = procedure address**
- The **ret** (return) instruction does the following
  - Pops return address from stack
  - Jumps to return address: **EIP = [ESP]; ESP = ESP + 4**

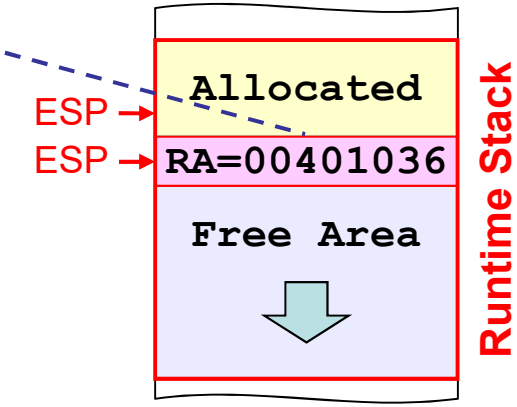
Details of CALL and Return

Address	Machine Code	Assembly Language
		`.CODE`
		`main PROC`
00401020	A1 00405000	`mov EAX, num1`
00401025	8B 1D 00405004	`mov EBX, num2`
0040102B	8B 0D 00405008	`mov ECX, num3`
00401031	E8 0000004B	`call sumof`
00401036	A3 0040500C	`mov sum, EAX`
. . .	. . .	. . .
		`exit`
		`main ENDP`
		`sumof PROC`
00401081	03 C3	`add EAX, EBX`
00401083	03 C1	`add EAX, ECX`
00401085	C3	`ret`
		`sumof ENDP`
		`END main`

IP-relative call

```
EIP = 00401036
    + 0000004B
EIP = 00401081
```

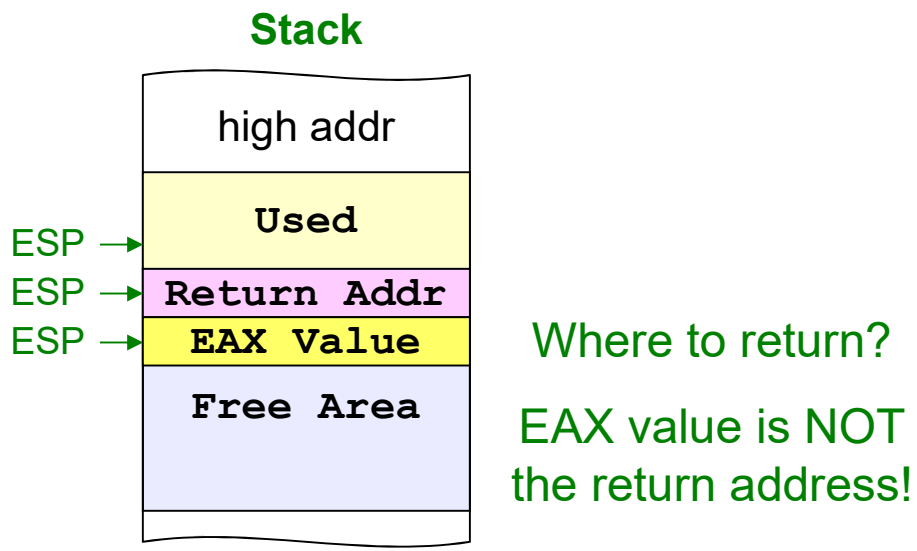
Before Call ESP = 0012FFC4
After Call ESP = 0012FFC0
After Ret (Return) ESP = 0012FFC4



Don't Mess Up the Stack !

- Just before returning from a procedure
  - Make sure the stack pointer ESP is pointing at return address
- Example of a messed-up procedure
  - Pushes EAX on the stack before returning
  - Stack pointer ESP is NOT pointing at return address!

```
main PROC
    call messedup
    . . .
    exit
main ENDP
messedup PROC
    push EAX
    ret
messedup ENDP
```



Nested Procedure Calls

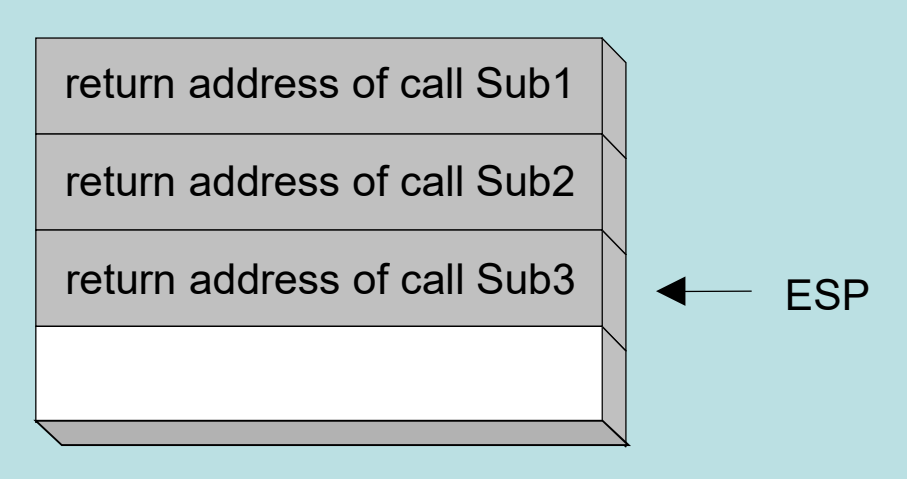
```
main PROC
    .
    .
    call Sub1
    exit
main ENDP

Sub1 PROC
    .
    .
    call Sub2
    ret
Sub1 ENDP

Sub2 PROC
    .
    .
    call Sub3
    ret
Sub2 ENDP

Sub3 PROC
    .
    .
    ret
Sub3 ENDP
```

By the time Sub3 is called, the stack contains all three return addresses



Parameter Passing

- 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
- Two types of storage areas used
  - Registers: general-purpose registers are used (register method)
  - Memory: stack is used (stack method)
- Two common mechanisms of parameter passing
  - Pass-by-value: parameter **value** is passed
  - Pass-by-reference: **address** of parameter is passed

Passing Parameters in Registers

```
;-----------------------------------------------------
; ArraySum: Computes the sum of an array of integers
; Receives: ESI = pointer to an array of doublewords
;           ECX = number of array elements
; Returns:  EAX = sum
;-----------------------------------------------------
ArraySum PROC
    mov eax,0                   ; set the sum to zero
L1: add eax, [esi]              ; add each integer to sum
    add esi, 4                  ; point to next integer
    loop L1                     ; repeat for array size
    ret
ArraySum ENDP
```

ESI: **Reference** parameter = array address

ECX: **Value** parameter = count of array elements

Preserving Registers

- Need to preserve the registers across a procedure call
  - Stack can be used to preserve register values
- Which registers should be saved?
  - Those registers that are modified by the called procedure
    - But still used by the calling procedure
  - We can save all registers using **`pusha`** if we need most of them
    - However, better to save only needed registers when they are few
- Who should preserve the registers?
  - Calling procedure: saves and frees registers that it uses
    - Registers are saved before procedure call and restored after return
  - Called procedure: **preferred method** for modular code
    - Register preservation is done in one place only (inside procedure)

Example on Preserving Registers

```
;-----------------------------------------------------
; ArraySum: Computes the sum of an array of integers
; Receives: ESI = pointer to an array of doublewords
;           ECX = number of array elements
; Returns:  EAX = sum
;-----------------------------------------------------
ArraySum PROC
    push esi                ; save esi, it is modified
    push ecx                ; save ecx, it is modified
    mov  eax,0              ; set the sum to zero
L1: add  eax, [esi]         ; add each integer to sum
    add  esi, 4             ; point to next integer
    loop L1                 ; repeat for array size
    pop  ecx                ; restore registers
    pop  esi                ; in reverse order
    ret
ArraySum ENDP
```

No need to save EAX. Why?

USES Operator

- The **USES** operator simplifies the writing of a procedure
  - Registers are frequently modified by procedures
  - Just list the registers that should be preserved after **USES**
  - Assembler will **generate** the **push** and **pop** instructions

```
ArraySum PROC USES esi ecx
    mov  eax,0
L1: add  eax, [esi]
    add  esi, 4
    loop L1
    ret
ArraySum ENDP
```

```
ArraySum PROC
    push esi
    push ecx
    mov  eax,0
L1: add  eax, [esi]
    add  esi, 4
    loop L1
    pop  ecx
    pop  esi
    ret
ArraySum ENDP
```