Faculty of Computers & Informatics

Assembly Language

Lecture 5 – Procedures

Ahmed Sallam

Slides based on original lecture slides by Dr. Mahmoud Elgayyar

Outline

- ***Linking to External Library***
  - The Irvine library
- *Stack Operations*
  - Runtime Stack
  - PUSH, POP instructions
- *Defining and Using Procedures*
  - PROC directive
  - CALL and RET instructions

Assembly & Execution

- *We write .asm files containing ASCII (i.e., text) versions of our program*
- *MASM* ***assembles*** *our .asm file into a .obj file – unlinked, Intel32 binary code*
- *All the .obj files are* ***linked*** *to create an executable – a .exe file*
- *The .exe file is* ***loaded*** *into main memory, addresses are resolved, and the program is executed*

Do you remember ?! (Example: Adding and Subtracting Integers 2nd version)

```
INCLUDE Irvine32.inc







.code
main PROC
     mov eax,10000h ;EAX=10000h
     add eax,40000h ;EAX = 50000h
     sub eax,20000h ;EAX = 30000h
     call DumpRegs  ;display registers
     exit
main ENDP
END main
```

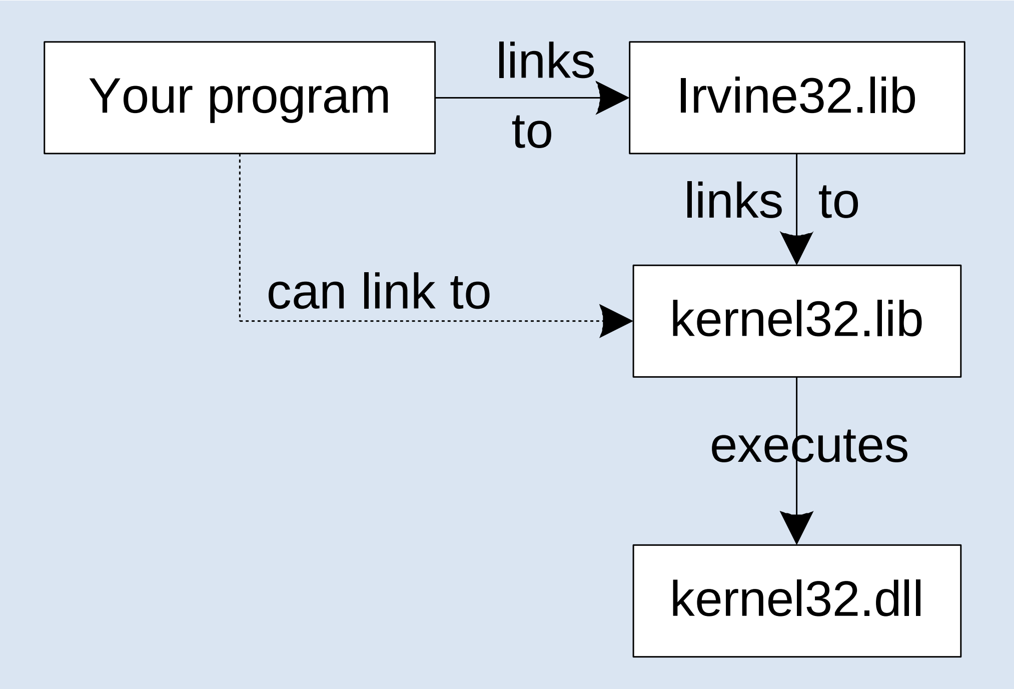
```
.386
.model flat, stdcall
.stack 4096
ExitProcess PROTO, dwExitCode:DWORD
DumpRegs PROTO



.code
main PROC
     mov eax,10000h ;EAX = 10000h
     add eax,40000h ;EAX = 50000h
     sub eax,20000h ;EAX = 30000h
     call DumpRegs  ;display registers
     INVOKE ExitProcess, 0
main ENDP
END main
```

Link Library Overview

- ***Procedure**: Same thing as a "method" in java or a "function" in C*
- ***Link Library**: A bunch of .obj files merged together*
- *A **file** containing compiled procedures*
  - OBJ files are assembled from ASM source files
  - Library built using the Microsoft LIB utility (or similar tool)
- *`Irvine32.lib` is an example of a link library*
- *Library is linked (i.e., joined) to your .asm file when you build your project*

Linking to a Library

- *Notice the two LIB files: `Irvine32.lib` and `kernel32.lib` (Part of the Microsoft Win32 SDK)*



Calling a Procedure

- *Call/Use a (library) procedure using the CALL instruction*
- *Some procedures require input arguments, which must be pre-placed in the proper location => a register*
- *The INCLUDE directive copies in the procedure prototypes (same thing as #include <stdio.h>)*

```
INCLUDE Irvine32.inc
.code
    mov  eax, 1234h           ; input argument
    call WriteHex             ; show hex number
    call Crlf                 ; end of line
```

Library Procedures

Clrscr	Clears console, locates cursor at upper left corner
Crlf	Writes end of line sequence to standard output
Delay	Pauses program execution for n millisecond interval
DumpMem	Writes block of memory to standard output in hex
DumpRegs	Displays general-purpose registers and flags (hex)
GetCommandtail	Copies command-line args into array of bytes
GetDateTime	Gets the current date and time from the system
MsgBox, MsgBoxAsk	Display popup message boxes
IsDigit	Sets Zero flag if AL contains ASCII code for decimal digit (0–9)
ParseDecimal32	Converts unsigned integer string to binary
ParseInteger32	Converts signed integer string to binary
Random32	Generates 32-bit pseudorandom integer in the range 0 to FFFFFFFFh

Library Procedures

OpenInputFile	Opens existing file for input
CloseFile	Closes an open disk file
CreateOutputFile	Creates new disk file for writing in output mode
ReadFromFile	Reads input disk file into buffer
WriteToFile	Writes buffer to output file
ReadChar , ReadDec , ReadHex , ReadInt , ReadString	Read from standard input
WriteChar , WriteDec , WriteString , WriteHex , WriteInt , WriteBin	Write to standard output
Str_compare , Str_copy , StrLength , Str_trim , Str_ucase	String operations
WaitMsg	Displays message, waits for Enter to be pressed

DON'T memorize!! Just know what can be done and be able to look them up for argument/parameter details (pgs 134-149)

Example 1

- *Clear the screen, delay the program for 500 milliseconds, and dump the registers and flags*

```
.code
    call Clrscr
    mov  eax,500
    call Delay
    call DumpRegs
```

Sample output:

```
EAX=00000613 EBX=00000000 ECX=000000FF EDX=00000000
ESI=00000000 EDI=00000100 EBP=0000091E ESP=000000F6
EIP=00401026 EFL=00000286 CF=0 SF=1 ZF=0 OF=0
```

Example 2

- *Display a null-terminated string and move the cursor to the beginning of the next screen line.*

```
.data
    str1 BYTE "Bus Strikes Really Suck!",0


.code
    mov  edx,OFFSET str1
    call WriteString
    call Crlf
```

Avoiding `call Crlf`

- *Display a null-terminated string and move the cursor to the beginning of the next screen line (use embedded CR/LF)*

```
.data
   str1 BYTE "The lab was too long!",0Dh,0Ah,0

.code
   mov  edx, OFFSET str1
   call WriteString
```

Example 3

- *Display an unsigned integer in binary, decimal, and hexadecimal, each on a separate line*

```
testVal = 35
.code
   mov  eax, testVal
   call WriteBin           ; display binary
   call Crlf
   call WriteDec           ; display decimal
   call Crlf
   call WriteHex           ; display hexadecimal
   call Crlf
```

Sample output:

```
0000 0000 0000 0000 0000 0000 0010 0011
35
23
```

Example 4

- *Input a string from the user*
  - EDX points to the string and ECX specifies the maximum number of characters the user is permitted to enter
  - Note: null (zero) byte is automatically added by ReadString

```
.data
   fileName BYTE 80 DUP(0)

.code
   mov  edx, OFFSET fileName
   mov  ecx, SIZEOF fileName ;readstring will read sizeof-1
   call ReadString
```

Example 5

- *Generate and display ten pseudo-random (semi-random) signed integers in the range 0 – 99*
  - Pass each integer to WriteInt (via EAX) and display it on a separate line

```
.code
   mov ecx,10                ; loop counter
genNum:
   mov  eax,100              ; ceiling value
   call RandomRange          ; generate random int
   call WriteInt             ; display signed int
   call Crlf                 ; goto next display line
   loop genNum               ; repeat loop
```

Review Questions

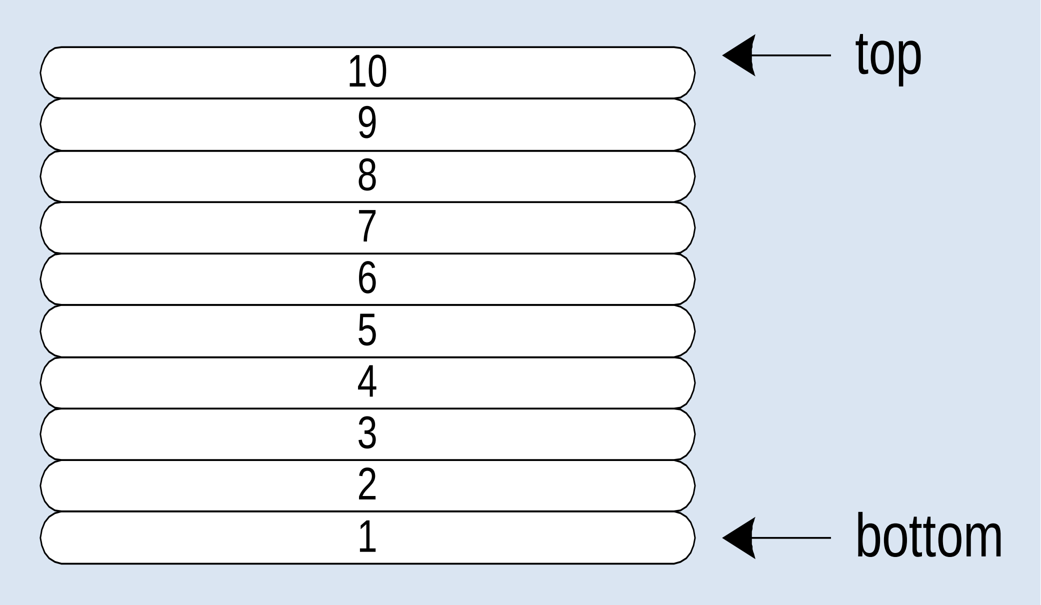
1. *What types of statements are inside the Irvine32.inc file?*
2. *(True/False): A link library consists of assembly language source code.*
3. *Write statements that cause a program to pause for 700 milliseconds.*
4. *Write statements that prompt the user for an identification number and input a string of digits into an array of bytes.*

Outline

- ***Linking to External Library***
  - The Irvine library
- ***Stack Operations***
  - Runtime Stack
  - PUSH, POP instructions
- *Defining and Using Procedures*
  - PROC directive
  - CALL and RET instructions

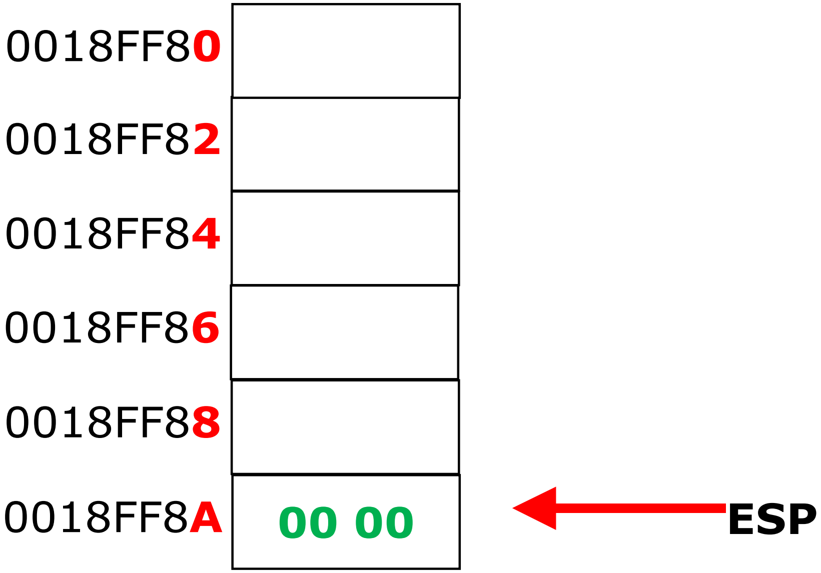
Runtime Stack

- *Imagine a stack of plates:*
  - plates are only added to the top = "*pushed*" on the stack
  - plates are only removed from the top = "*pulled*" from the stack
  - LIFO structure – "Last In, First Out"



x86 CPU Stack

- *Managed by the CPU, using two registers*
  - SS (stack segment) – Segment being used for stack
  - ESP (stack pointer) – Pointer/Address/Offset of TOP of Stack
  - In reality, the stack pointer starts at the highest location in the stack segment



0018FF80

0018FF82

0018FF84

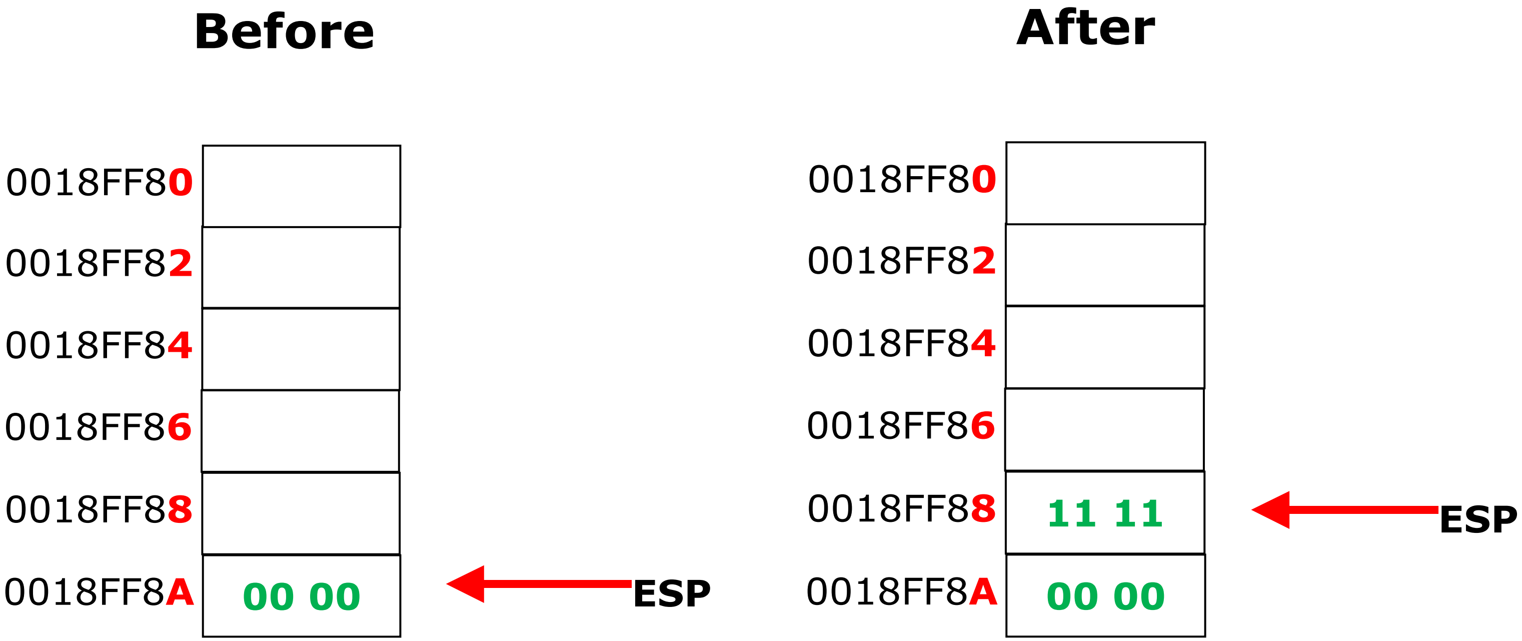
0018FF86

0018FF88

0018FF8A 00 00 ← ESP

PUSH

1. *A 16-bit push operation decrements the stack pointer by 2, and*
2. *Copies a value into the location pointed to by the stack pointer*



Before

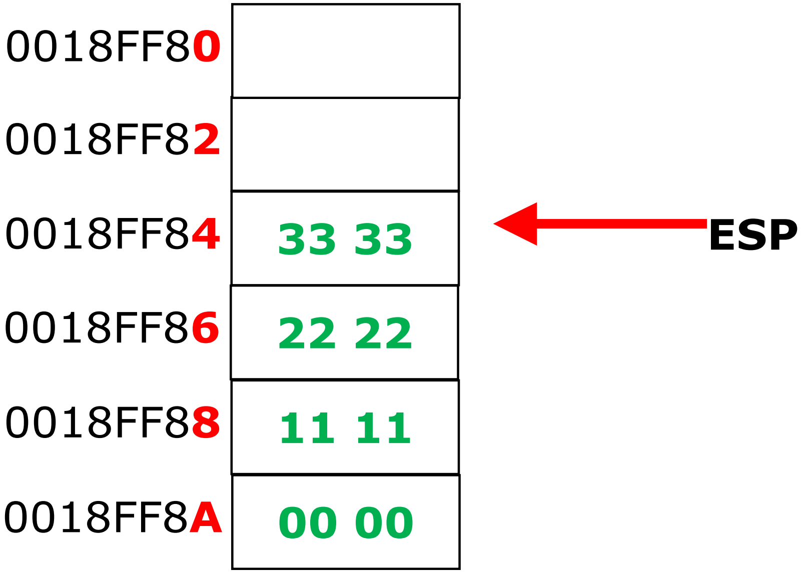
0018FF80		
0018FF82		
0018FF84		
0018FF86		
0018FF88		
0018FF8A	00 00	ESP

After

0018FF80		
0018FF82		
0018FF84		
0018FF86		
0018FF88	11 11	ESP
0018FF8A	00 00	

More Pushing

- *After pushing two more integers:*

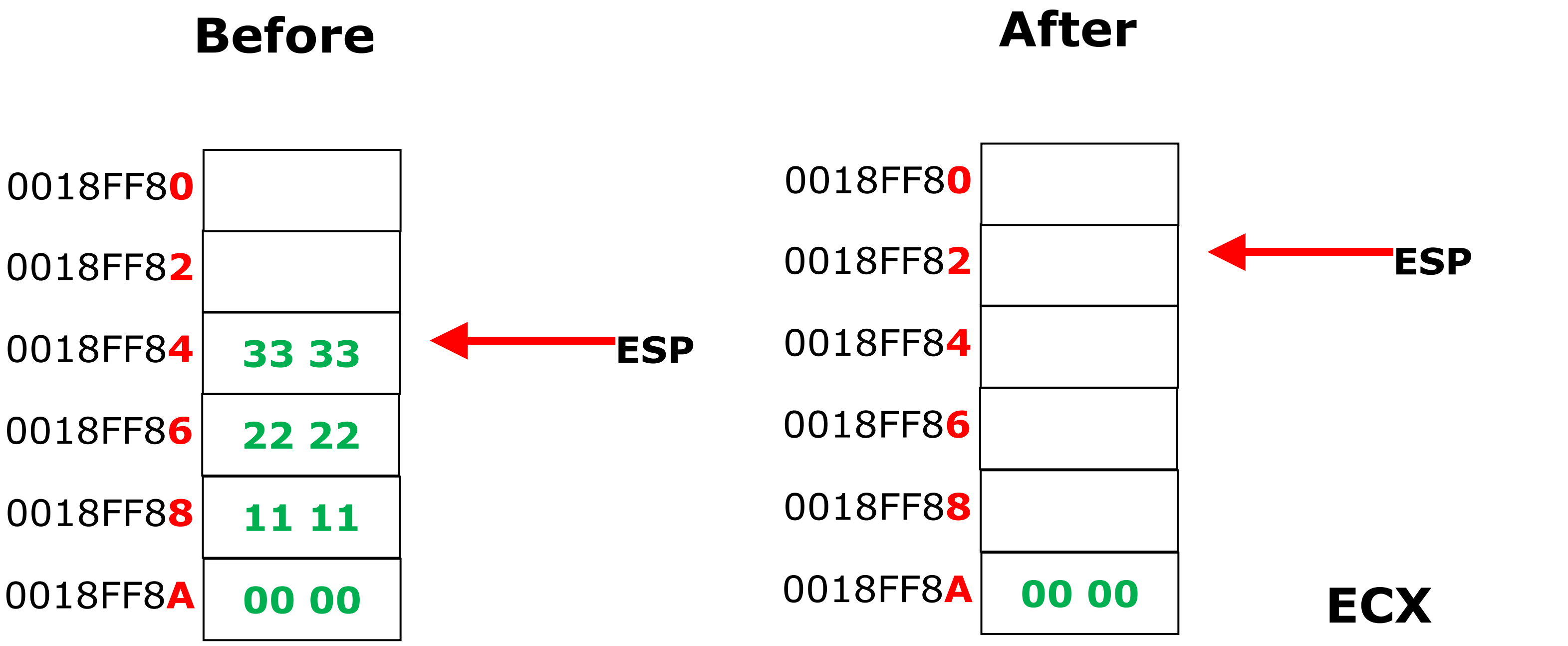


0018FF80		
0018FF82		
0018FF84	33 33	ESP
0018FF86	22 22	
0018FF88	11 11	
0018FF8A	00 00	

- The stack grows downward (into LOWER addresses/offsets)
- The area below ESP is always available (unless the stack overflows)
- Overflow: When segment is filled (and no more space is available)

POP

1. *Copies value at `stack[ESP]` into a register or variable, and*
2. *Adds n to ESP, where n is either 2 or 4 (depending on size of destination)*



Before

0018FF80	
0018FF82	
0018FF84	33 33 ← ESP
0018FF86	22 22
0018FF88	11 11
0018FF8A	00 00

After

0018FF80	
0018FF82	← ESP
0018FF84	
0018FF86	
0018FF88	
0018FF8A	00 00

ECX

PUSH, POP Formats

- *PUSH* **syntax:**
  1. PUSH *r/m16* *r/m = register/memory*
  2. PUSH *r/m32*
  3. PUSH *imm32*
- *POP* **syntax:**
  1. POP *r/m16*
  2. POP *r/m32*

Using PUSH and POP

- *Save and restore registers when they contain important values*
- *PUSH and POP instructions occur in the opposite order (LIFO)*

```
push esi                              ; push registers
push ecx
push ebx

mov  esi,OFFSET dwordVal              ; display some memory
mov  ecx,LENGTHOF dwordVal
mov  ebx,TYPE dwordVal
call DumpMem

pop  ebx                              ; restore registers
pop  ecx
pop  esi
```

The **DumpMem** procedure writes a range of memory to the console window in hexadecimal. Pass it the starting address in ESI, the number of units in ECX, and the unit size in EBX

Example: Nested Loop

- *Idea:*
  - Use stack to save loop counter of outer loop when in inner loop
  - push the outer loop counter before entering the inner loop

```
    mov   ecx, 100          ; set outer loop count
outer:                      ; begin the outer loop
    push ecx                ; save outer loop count

    mov   ecx, 20           ; set inner loop count
inner:                      ; begin the inner loop
    …  Code for inner loop goes here  …
    loop inner              ; repeat the inner loop

    pop   ecx               ; restore outer loop count
    loop outer              ; repeat the outer loop
```

Related Instructions

1. *PUSHFD and POPFD push and pop the EFLAGS register*

2. *PUSHAD pushes the 32-bit general-purpose registers on the stack*

   order: EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI

3. *POPAD pops the same registers off the stack in reverse order*

4. *PUSHA and POPA do the same for 16-bit registers*

Review Questions

1. *When a 32-bit value is pushed on the stack, what happens to ESP?*
2. *(True/False) Only 32-bit values should be pushed on the stack when using the Irvine32 library.*
3. *(True/False) Only 16-bit values should be pushed on the stack when using the Irvine16 library.*
4. *(True/False) Local variables in procedures are created on the stack.*
5. *Challenge: Suppose there were no PUSH instruction. Write a sequence of two other instructions that would accomplish the same as PUSH EAX.*

Outline

- ***Linking to External Library***
  - The Irvine library
- ***Stack Operations***
  - Runtime Stack
  - PUSH, POP instructions
- ***Defining and Using Procedures***
  - PROC directive
  - CALL and RET instructions

Creating Procedures

- *Large problems can be divided into smaller tasks to make them more manageable*
- *A* ***procedure*** *is the ASM equivalent of a Java Method, C/C++ Function, Basic Subroutine, or Pascal Procedure*
- *Same thing as what is in the Irvine32 library*
- *The following is an assembly language procedure named `sample`:*

```
sample PROC
    … Code for procedure goes here …
    ret
sample ENDP
```

SumOf Procedure

```
;-----------------------------------------------------
;
; Calculates and returns the sum of three 32-bit ints
; Receives: EAX, EBX, ECX, the three integers
;           may be signed or unsigned.
; Returns:  EAX = sum
;           status flags are changed.
; Requires: nothing
;-----------------------------------------------------
SumOf PROC

   add eax,ebx

   add eax,ecx

   Ret

SumOf ENDP
```

CALL-RET Example

0000025 is the offset of the instruction immediately following the **CALL** instruction

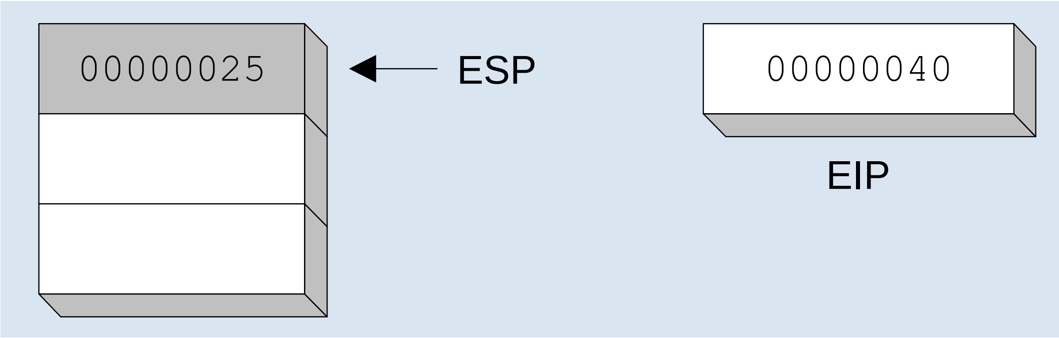
00000040 is the offset of the first instruction inside **MySub**

```
main PROC
    00000020 call MySub
    00000025 mov eax,ebx
    .
    .
main ENDP

MySub PROC
    00000040 mov eax,edx
    .
    .
    ret
MySub ENDP
```

CALL-RET in Action

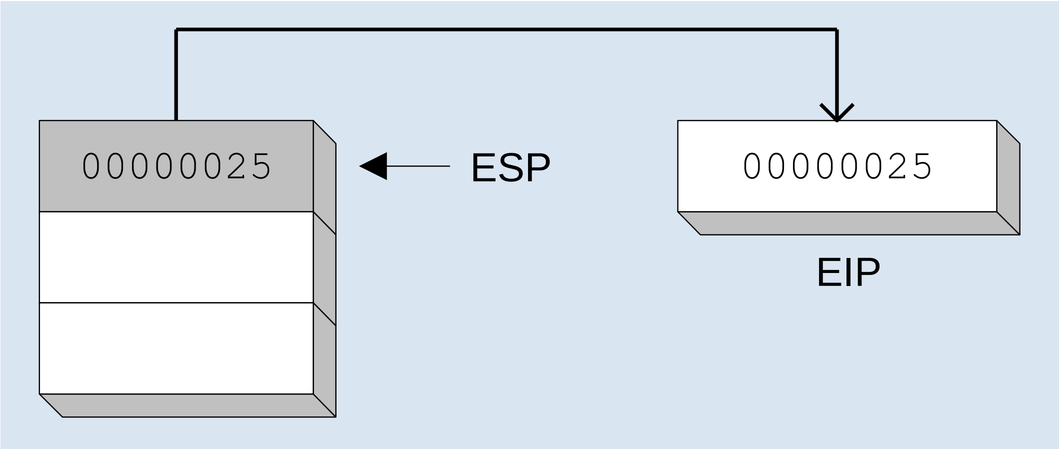
The **CALL** instruction pushes 00000025 onto the stack, and loads 00000040 into EIP

```
CALL =
   PUSH eip
   MOV EIP, OFFSET proc
```



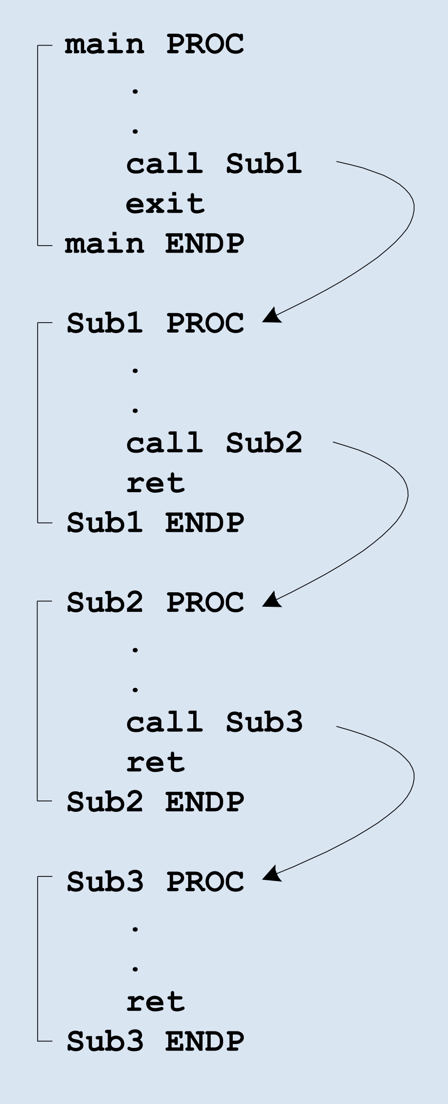
The **RET** instruction pops 00000025 from the stack into EIP

```
RET = POP eip
```



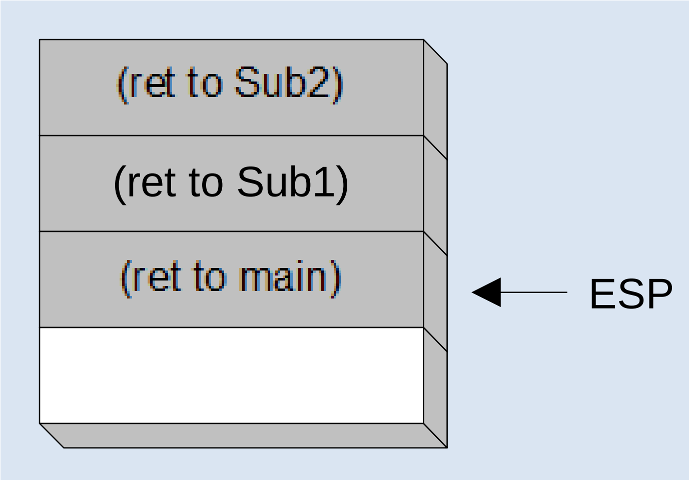
(stack shown before **RET** executes)

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:

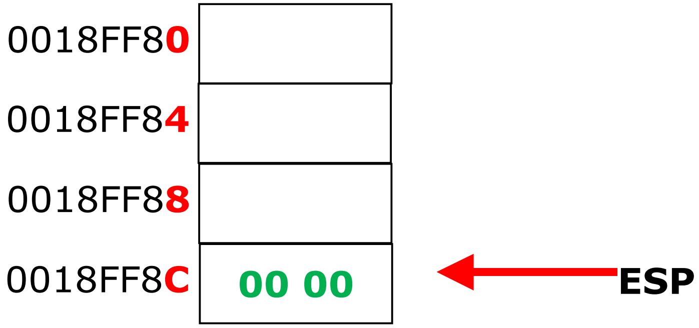


Local and Global Labels

1. *A local label is visible only inside the same procedure*

2. *A global label is visible everywhere*

```
main PROC
    jmp L2          ; error
    call sub2
L1::                ; global label
    exit
main ENDP
sub2 PROC
L2:                 ; local label
    jmp L1          ; legal, but stupid
    ret             ; When is ret ever called?
sub2 ENDP
```



0018FF80

0018FF84

0018FF88

0018FF8C 00 00

ESP

Without Parameters

- *The ArraySum procedure calculates the sum of an array. It makes two references to specific variable names:*

```
ArraySum PROC
    mov esi,0                      ; array index
    mov eax,0                      ; set the sum to zero
    mov ecx,LENGTHOF myArray       ; set number of elements

forEach:
    add eax,myArray[esi]           ; add each integer to sum
    add esi,4                      ; point to next integer
    loop forEach                   ; repeat for array size

    mov theSum,eax                 ; store the sum
    ret
ArraySum ENDP
```

This procedure needs parameters so that the array name and result location can be passed in/out and permit the function to be used with different arrays.

With Parameters

- *This version of ArraySum returns the sum of any doubleword array whose address is in ESI. The sum is returned in EAX:*

```
; Add an array of doublewords
; ESI = address of array, ECX = no. of elements
; Returns: EAX = sum; ECX, ESI, & flags changed

ArraySum PROC
    mov eax,0                          ; set the sum to zero

forEach:
    add eax,[esi]                      ; add each integer to sum
    add esi,4                          ; point to next integer
    loop forEach                       ; repeat for array size

    ret
ArraySum ENDP
```

USES Operator

- *Lists the registers that are used by a procedure*
- *MASM inserts code that will try to preserve them*

```
ArraySum PROC USES esi ecx
    mov eax,0                       ; set the sum to zero
    etc.
```

MASM generates the code shown in gold:

```
ArraySum PROC
    push esi
    push ecx
    .
    .
    pop ecx
    pop esi
    ret
ArraySum ENDP
```

Review Questions

1. *(True/False): It is possible to define a procedure inside an existing procedure.*
2. *What would happen if the RET instruction was omitted from a procedure?*
3. *(True/False): The CALL instruction pushes the offset of the CALL instruction on the stack.*
4. *(True/False): In protected mode, each procedure call uses a minimum of 4 bytes of stack space.*
5. *(True/False): The USES operator only generates PUSH instructions, so you must code POP instructions yourself.*

Summary

- *Linking to External Library*
  - The Irvine library
- *Stack Operations*
  - Runtime Stack
  - PUSH, POP instructions
- *Defining and Using Procedures*
  - PROC directive
  - CALL and RET instructions