EE-2003 Computer Organization & Assembly Language

INSTRUCTOR

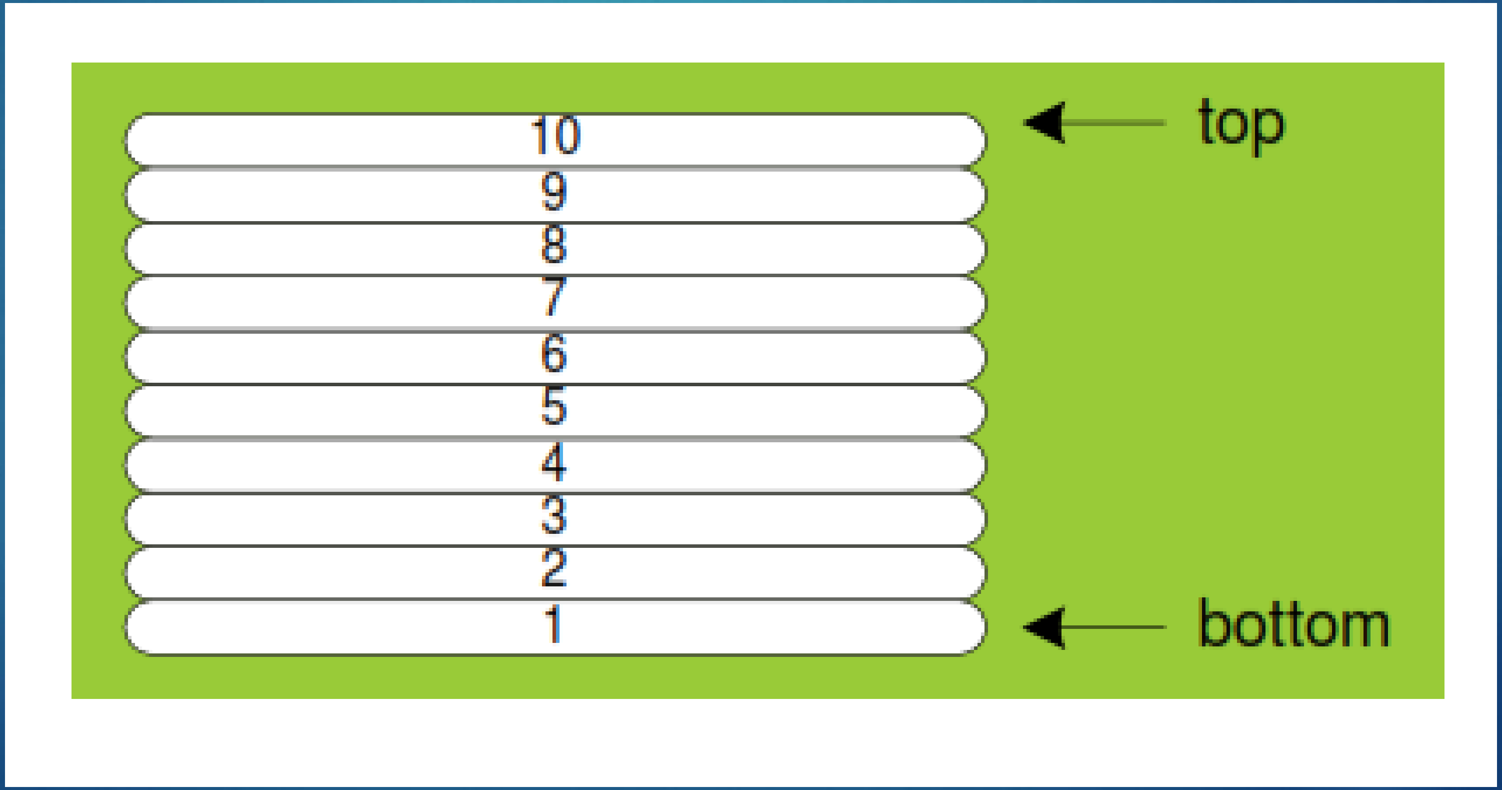
Engr. Aashir Mahboob
Lecturer, Department of Computer Science
FAST NUCES (Karachi)
aashir.mahboob@nu.edu.pk

CH# 05 PROCEDURES

OUTLINE

- Stack Operations
- Defining and Using Procedures

- Imagine a stack of plates . . .
- plates are only added to the top
- plates are only removed from the top
- LIFO structure



STACK OPERATIONS

- •A *stack data structure follows the same principle as a stack of plates:*
- • New values are added to the top of the stack, and existing values are removed from the top.
- •A stack is also called a LIFO structure (*Last-In, First-Out*) *because the last* value put into the stack is always the first value taken out.

STACK OPERATIONS

- LIFO (Last-In, First-Out) data structure.
- push/pop operations

- You probably have had experiences on implementing it in high-level languages.

- Here, we concentrate on *runtime stack*, directly supported by hardware in the CPU. It is essential for calling and returning from procedures.

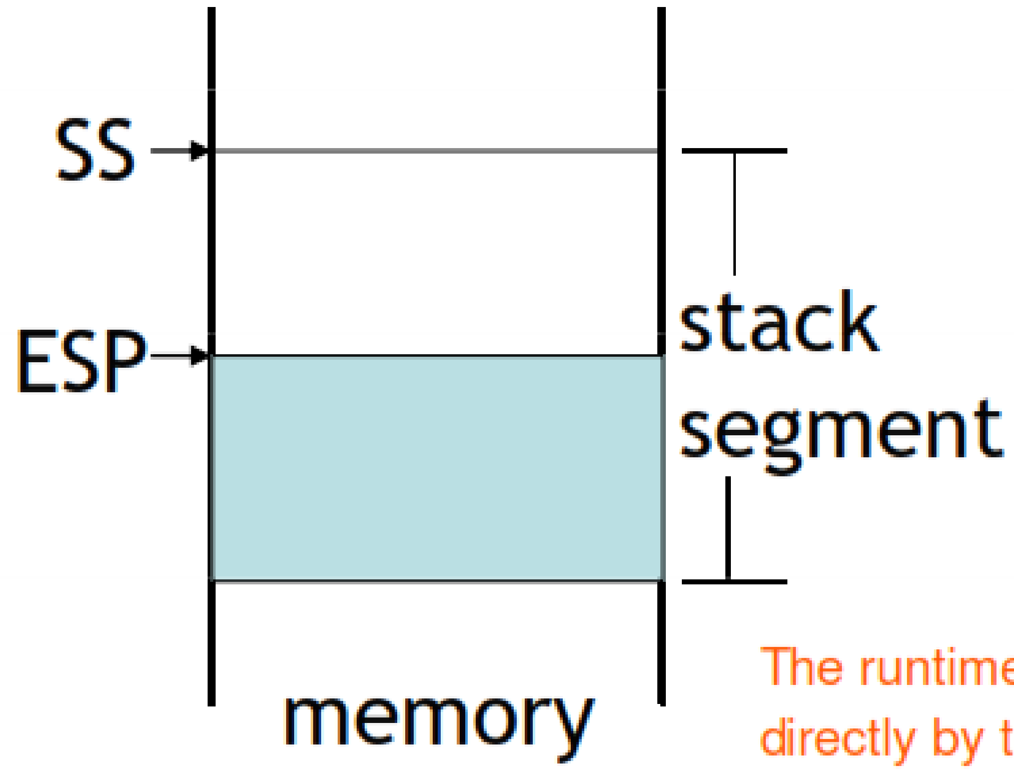
The runtime stack stores information about the active subroutines of a computer program.

Runtime Stack

- The *runtime stack* is a memory array managed directly by the CPU, using the ESP (extended stack pointer) register, known as the *stack pointer register.*

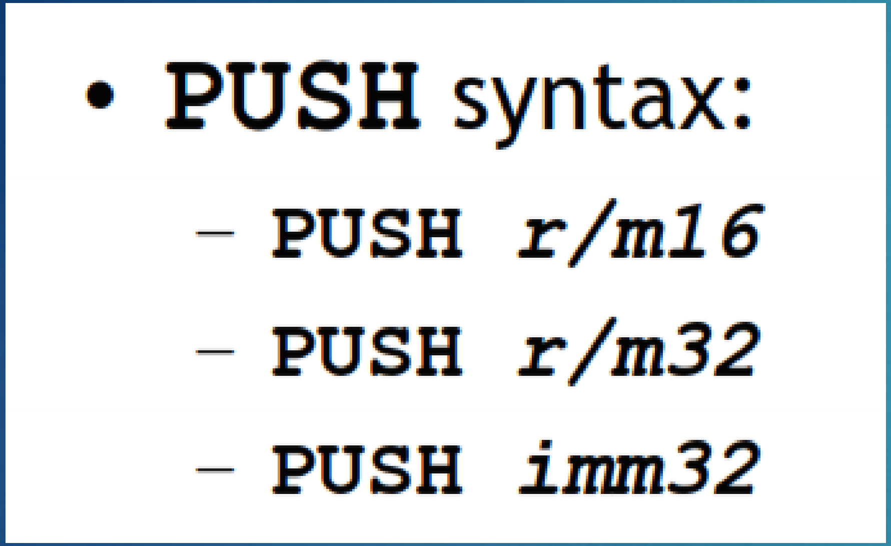
- Managed by the CPU, using two registers
  - SS (stack segment)
  - ESP (stack pointer) * : point to the top of the stack usually modified by **`CALL`**, **`RET`**, **`PUSH`** and **`POP`**

Runtime Stack

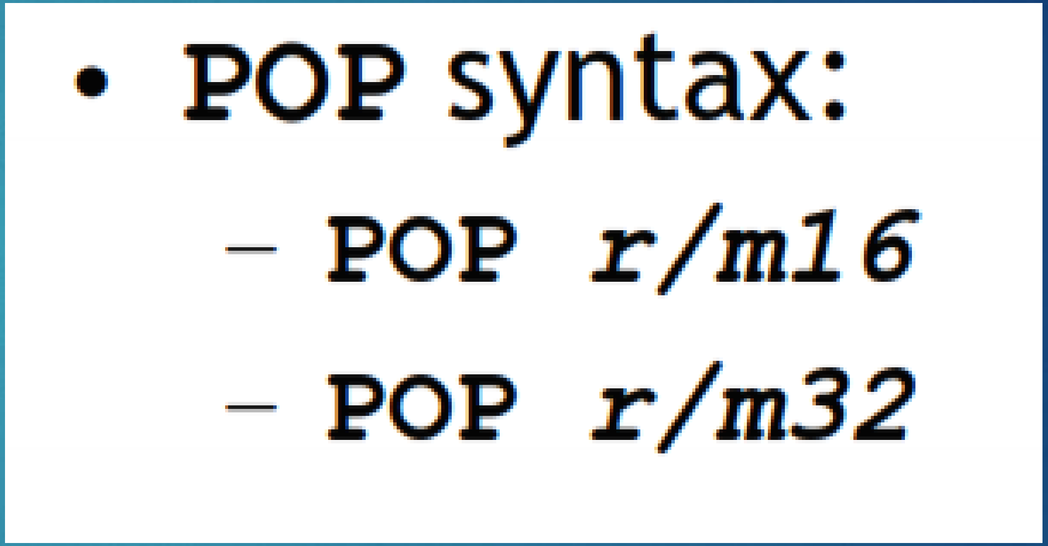


The runtime stack is a memory array managed directly by the CPU, using the ESP register, known as the stack pointer register.

PUSH and POP instructions



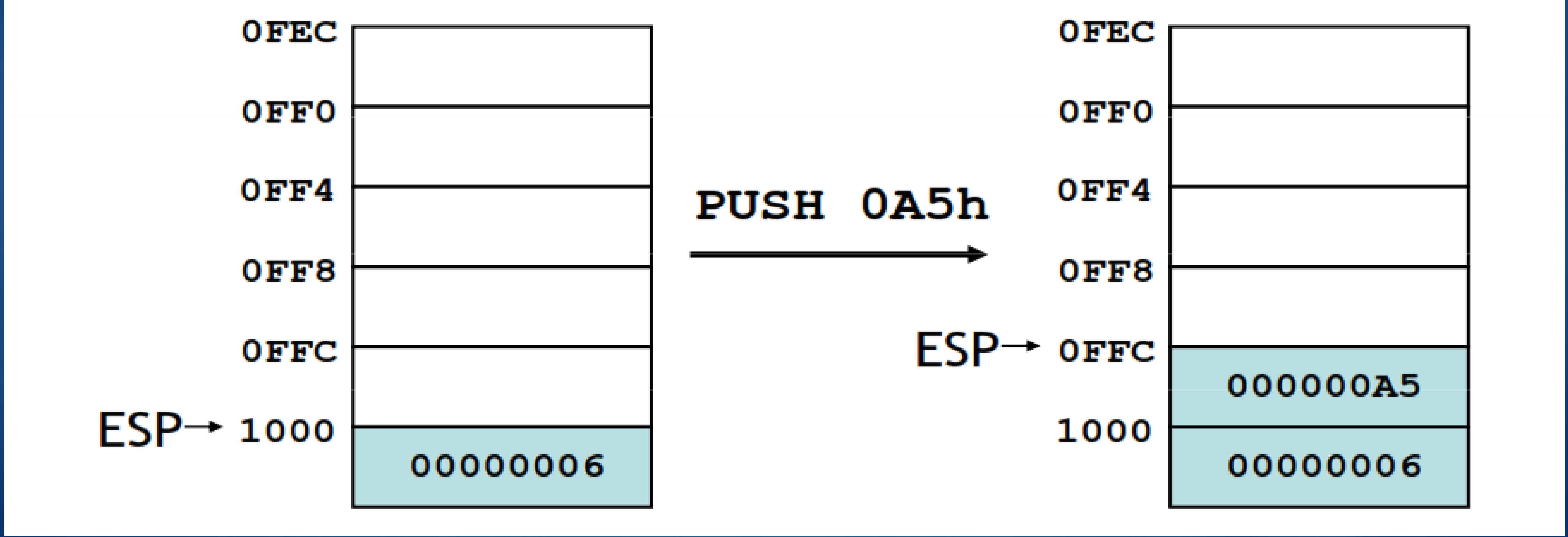
- **PUSH** syntax:
  - `PUSH` *`r/m16`*
  - `PUSH` *`r/m32`*
  - `PUSH` *`imm32`*



- **POP** syntax:
  - `POP` *`r/m16`*
  - `POP` *`r/m32`*

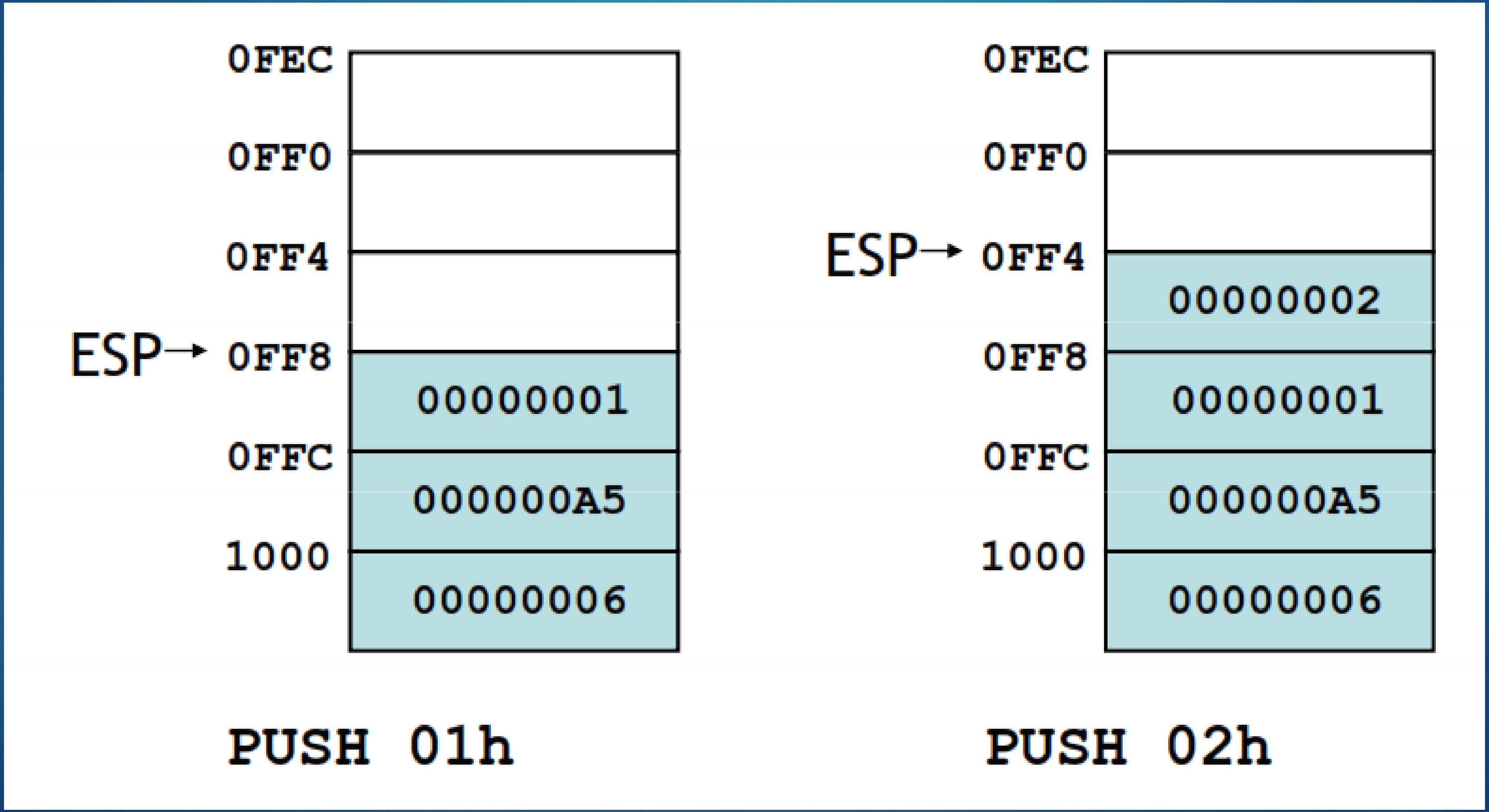
PUSH operation

- A **push** operation decrements the stack pointer by 2 or 4 (depending on operands) and copies a value into the location pointed to by the stack pointer.



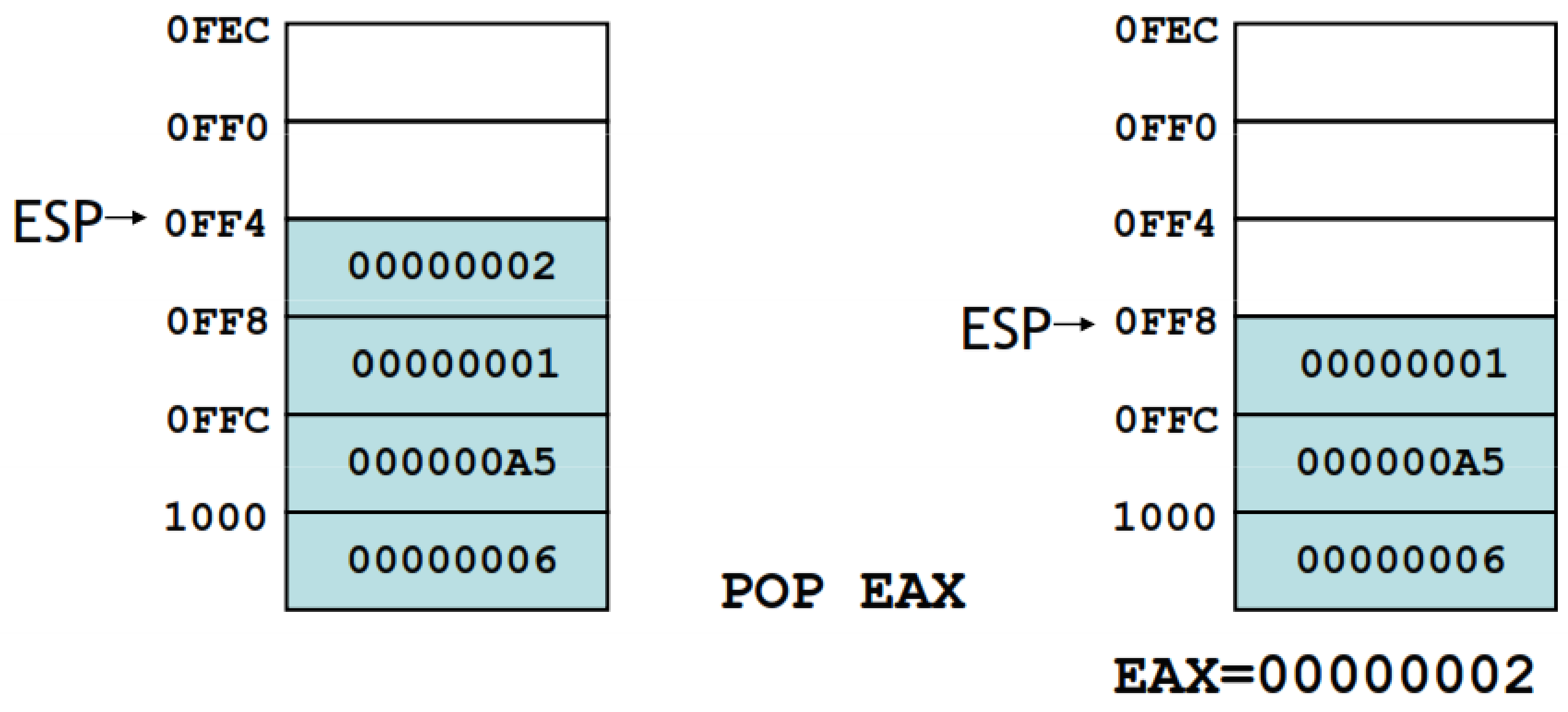
PUSH operation

- The same stack after pushing two more integers:



POP operation

- Copies value at stack[ESP] into a register or variable.
- Adds *n* to ESP, where *n* is either 2 or 4, depending on the attribute of the operand receiving the data



When to use stacks

- Temporary save area for registers
- To save return address for CALL

- To pass arguments
- Local variables

- Applications which have LIFO nature, such as reversing a string

Example of using stacks

Save and restore registers when they contain important values. Note that the **PUSH** and **POP** instructions are in the opposite order:

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

mov esi,OFFSET dwordVal       ; starting OFFSET
mov ecx,LENGTHOF dwordVal;  number of units
mov ebx,TYPE dwordVal ;size of a doubleword
call DumpMem                  ; display memory

pop ebx                       ; opposite order
pop ecx
pop esi
```

Example: Nested Loop

When creating a nested loop, push the outer loop counter before entering the inner loop:

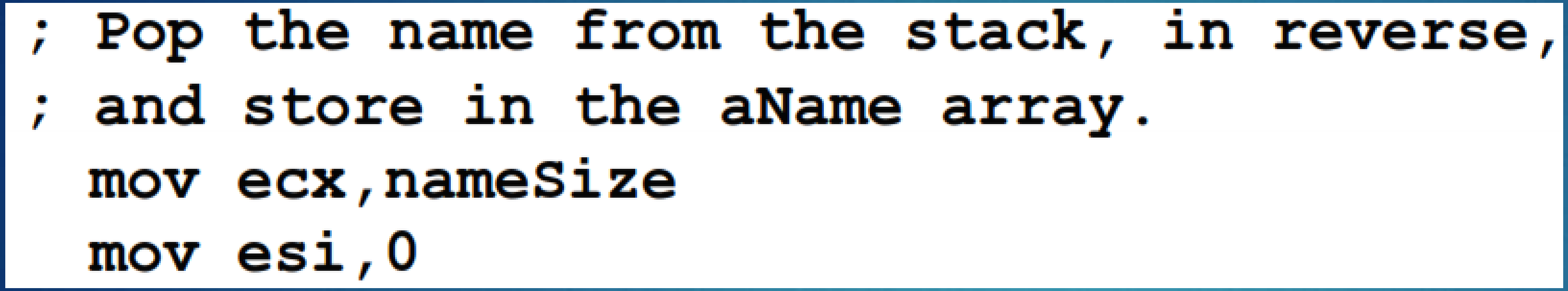
```
   mov ecx,100        ; set outer loop count
L1:                   ; begin the outer loop
   push ecx           ; save outer loop count

   mov ecx,20         ; set inner loop count
L2:                   ; begin the inner loop
    ;
    ;
   loop L2            ; repeat the inner loop

   pop ecx            ; restore outer loop count
   loop L1            ; repeat the outer loop
```

Example: reversing a string

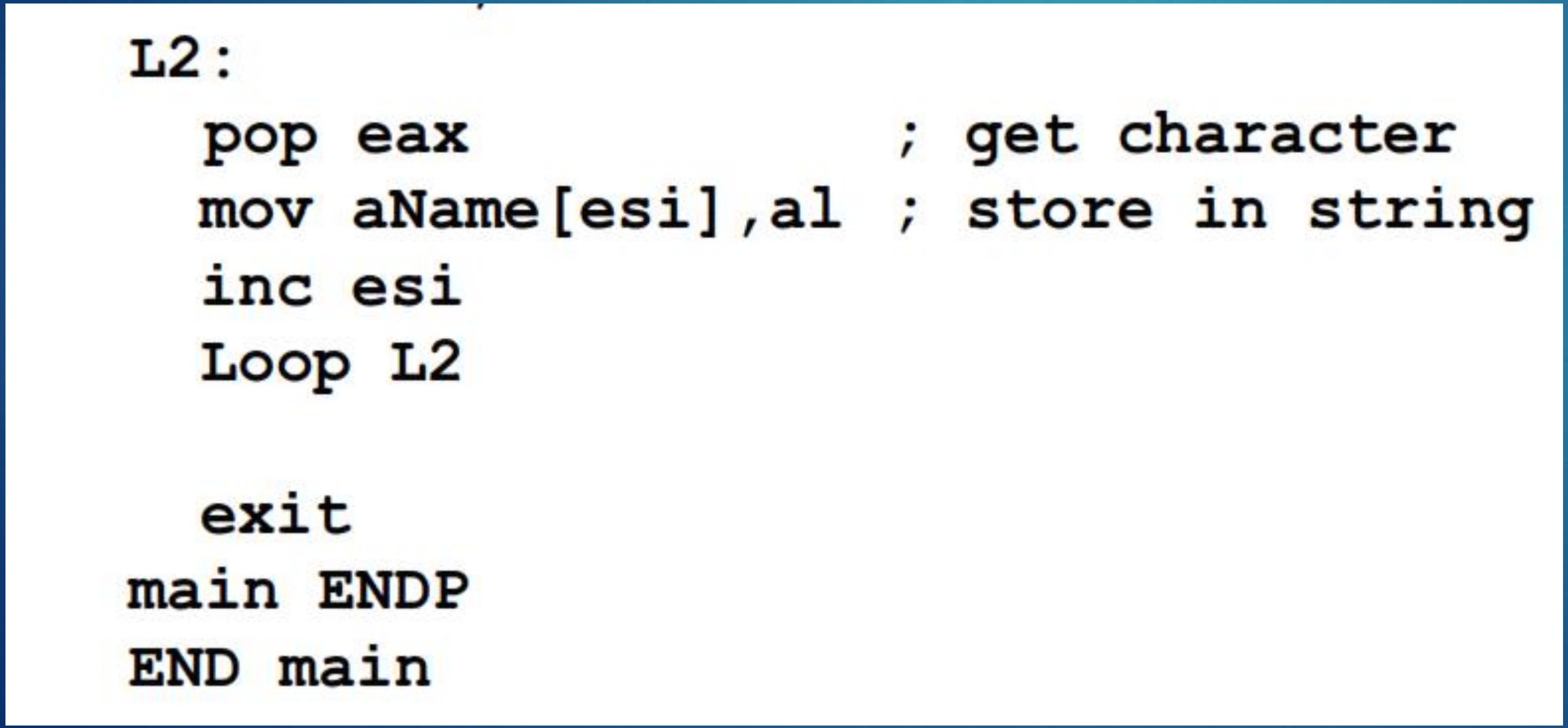
```
 .data
aName BYTE "Abraham Lincoln",0
nameSize = ($ - aName) - 1
```

```
.code
main PROC
; Push the name on the stack.
  mov ecx,nameSize
  mov esi,0
L1:
  movzx eax,aName[esi]    ; get character
  push eax                ; push on stack
  inc esi
  Loop L1
```

Example: reversing a string

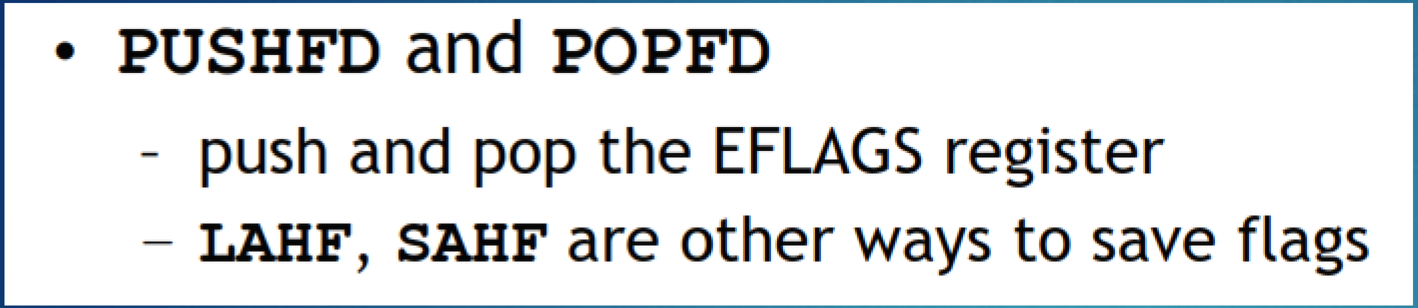


```
; Pop the name from the stack, in reverse,
; and store in the aName array.
  mov ecx,nameSize
  mov esi,0
```

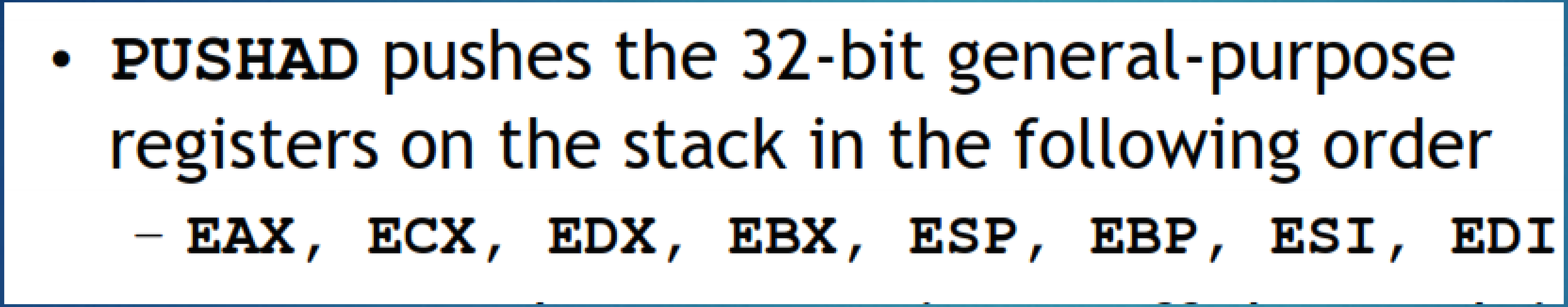


```
  L2:
    pop eax              ; get character
    mov aName[esi],al    ; store in string
    inc esi
    Loop L2

    exit
  main ENDP
  END main
```

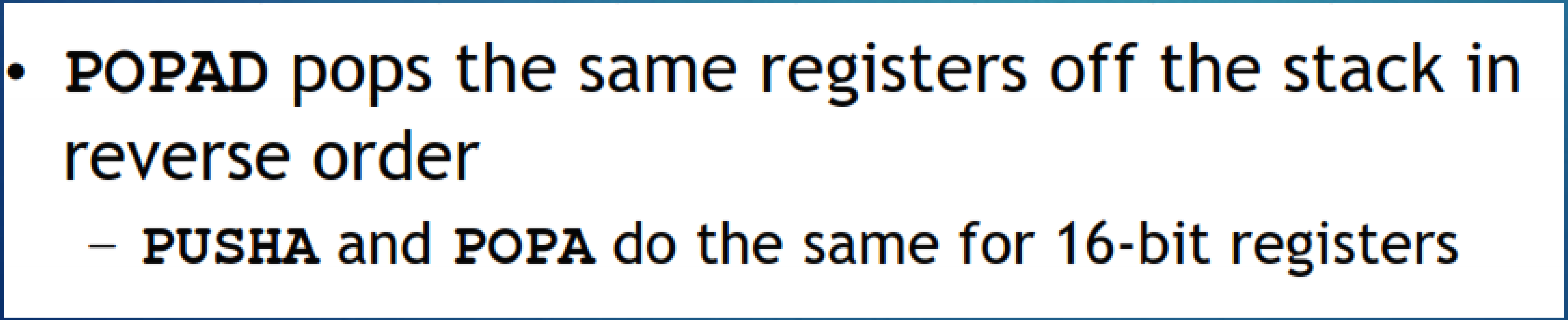
Related instructions



- **PUSHFD** and **POPFD**
  - push and pop the EFLAGS register
  - **LAHF**, **SAHF** are other ways to save flags



- **PUSHAD** pushes the 32-bit general-purpose registers on the stack in the following order
  - **EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI**



- **POPAD** pops the same registers off the stack in reverse order
  - **PUSHA** and **POPA** do the same for 16-bit registers

Defining and using procedures

- Large problems can be divided into smaller tasks to make them more manageable
- A procedure is the ASM equivalent of a Java or C++ function

Following is an assembly language procedure named sample:

```
sample PROC
   .
   .
   ret
sample ENDP
```

A named block of statements that ends with a return.

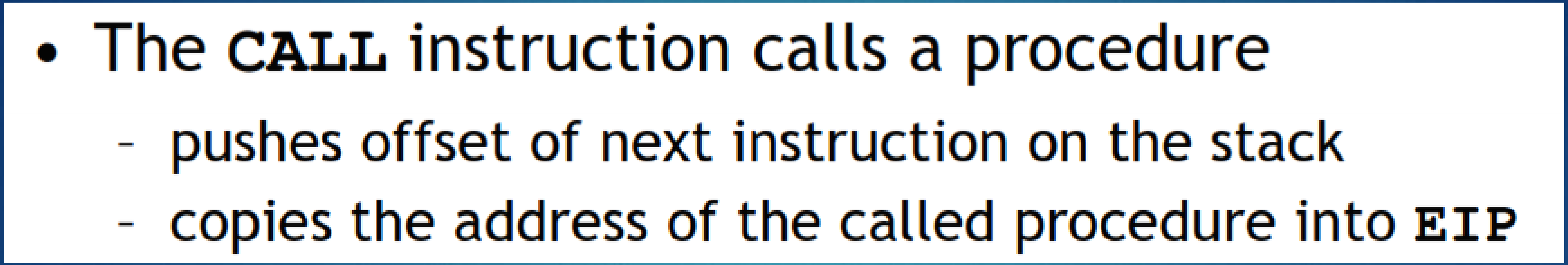
Defining and using procedures

- A procedure is named block of statements that ends in a return statement
  - Declared using PROC and ENDP directives

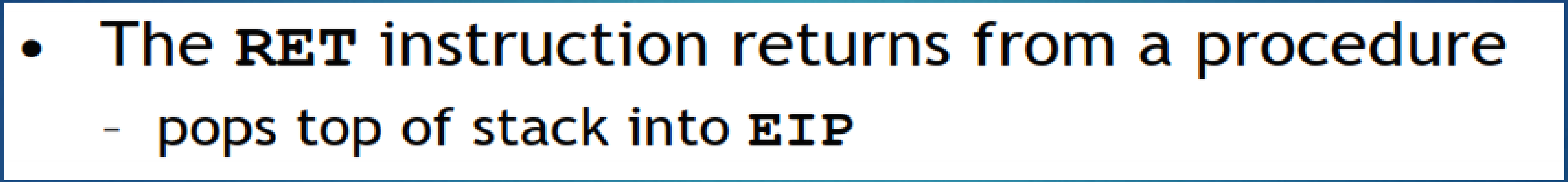
- When you create a procedure other than your program's startup procedure, end it with a RET instruction.
  - RET forces the CPU to return to the location from where the procedure was called:

```
sample PROC
        add eax,ebx
        add eax,ecx
        ret
sample ENDP
```

CALL and RET instructions

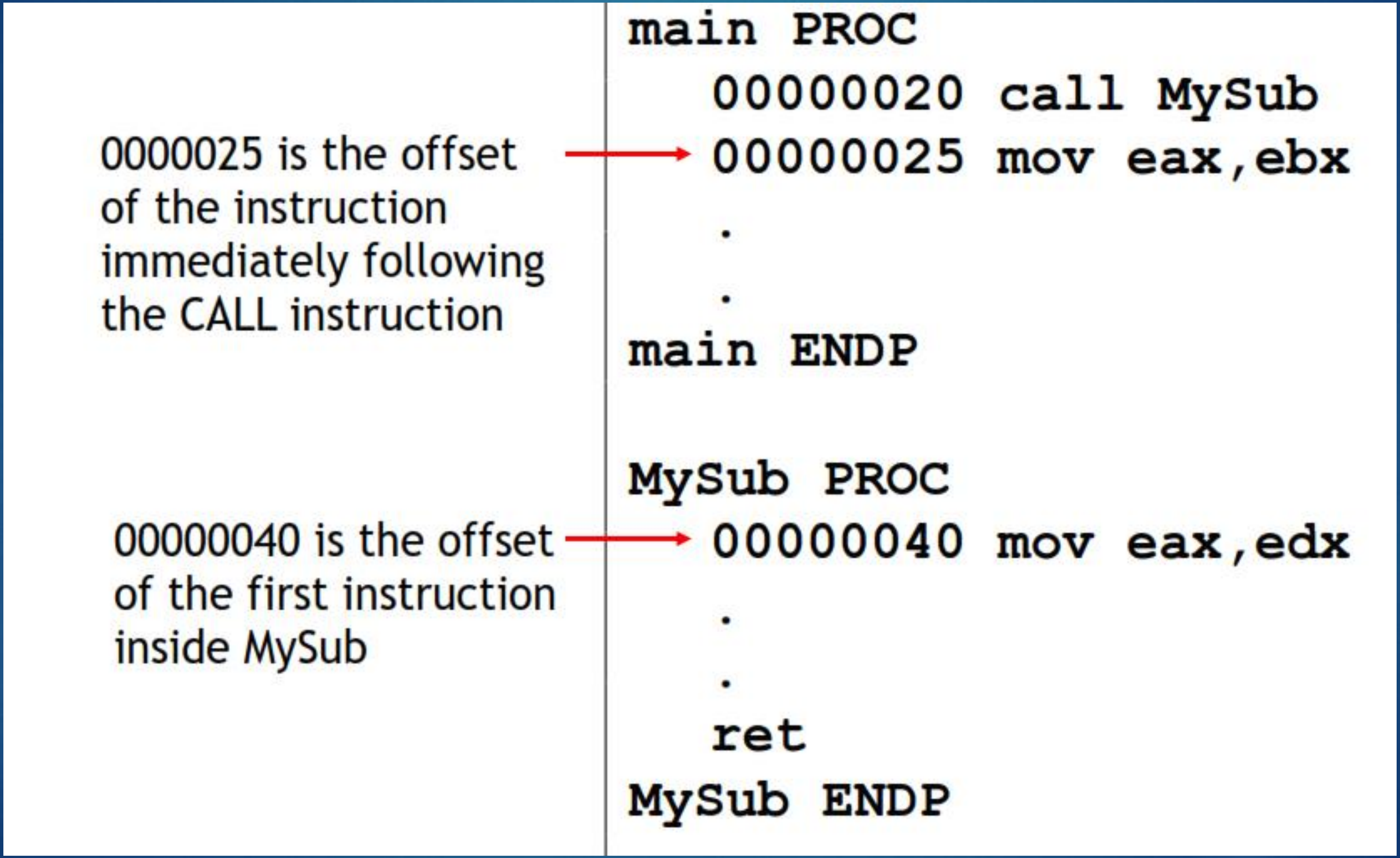


- The **`CALL`** instruction calls a procedure
  - pushes offset of next instruction on the stack
  - copies the address of the called procedure into **`EIP`**



- The **`RET`** instruction returns from a procedure
  - pops top of stack into **`EIP`**

CALL-RET example (1 of 2)



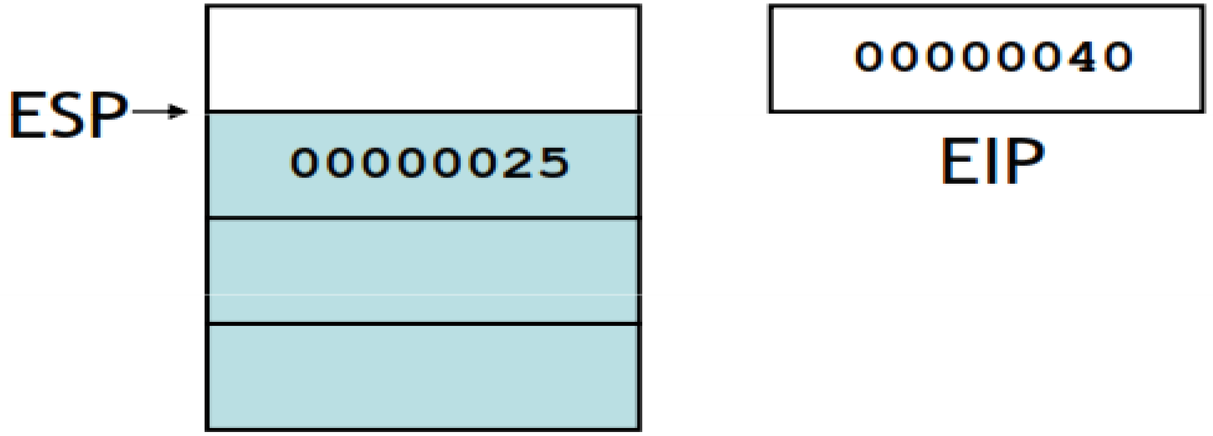
0000025 is the offset of the instruction immediately following the CALL instruction

00000040 is the offset of the first instruction inside MySub

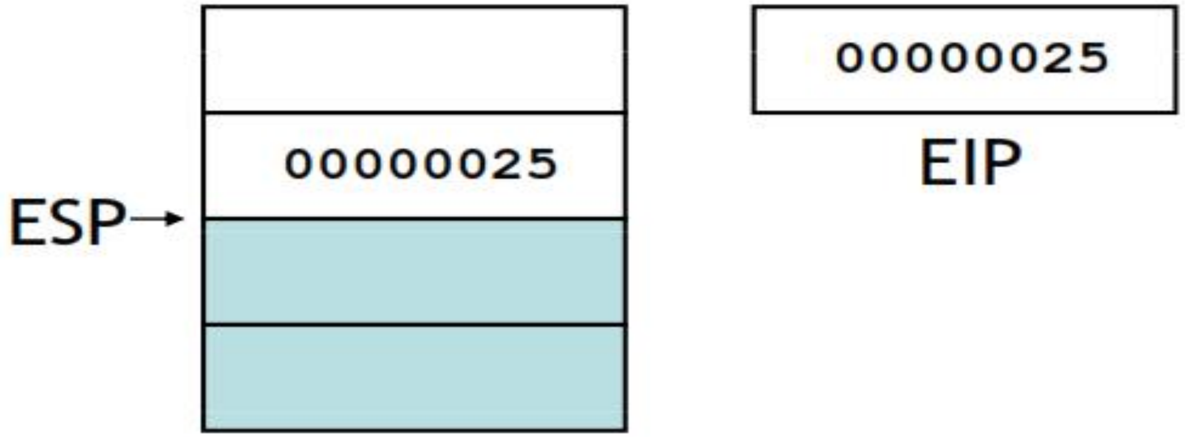
```
main PROC
   00000020 call MySub
   00000025 mov eax,ebx
   .
   .
main ENDP

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

CALL-RET example (2 of 2)

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



The RET instruction pops 0000025 from the stack into EIP



Library Procedures - Overview (1 of 3)

- **CloseFile** – Closes an open disk file
- **Clrscr** - Clears console, locates cursor at upper left corner
- **CreateOutputFile** - Creates new disk file for writing in output mode
- **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
- **GetMaxXY** - Gets number of cols, rows in console window buffer
- **GetMseconds** - Returns milliseconds elapsed since midnight

Library Procedures - Overview (2 of 3)

- **GetTextColor** - Returns active foreground and background text colors in the console window
- **Gotoxy** - Locates cursor at row and column on the console
- **IsDigit** - Sets Zero flag if AL contains ASCII code for decimal digit (0–9)
- **MsgBox, MsgBoxAsk** – Display popup message boxes
- **OpenInputFile** – Opens existing file for input
- **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
- **Randomize** - Seeds the random number generator
- **RandomRange** - Generates a pseudorandom integer within a specified range
- **ReadChar** - Reads a single character from standard input

Library Procedures - Overview (3 of 3)

- **ReadDec** - Reads 32-bit unsigned decimal integer from keyboard
- **ReadFromFile** – Reads input disk file into buffer
- **ReadHex** - Reads 32-bit hexadecimal integer from keyboard
- **ReadInt** - Reads 32-bit signed decimal integer from keyboard
- **ReadKey** – Reads character from keyboard input buffer
- **ReadString** - Reads string from standard input, terminated by [Enter]
- **SetTextColor** - Sets foreground and background colors of all subsequent console text output
- **Str_compare** – Compares two strings
- **Str_copy** – Copies a source string to a destination string
- **StrLength** – Returns length of a string
- **Str_trim** - Removes unwanted characters from a string.

Example 1

TASK:-

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
```

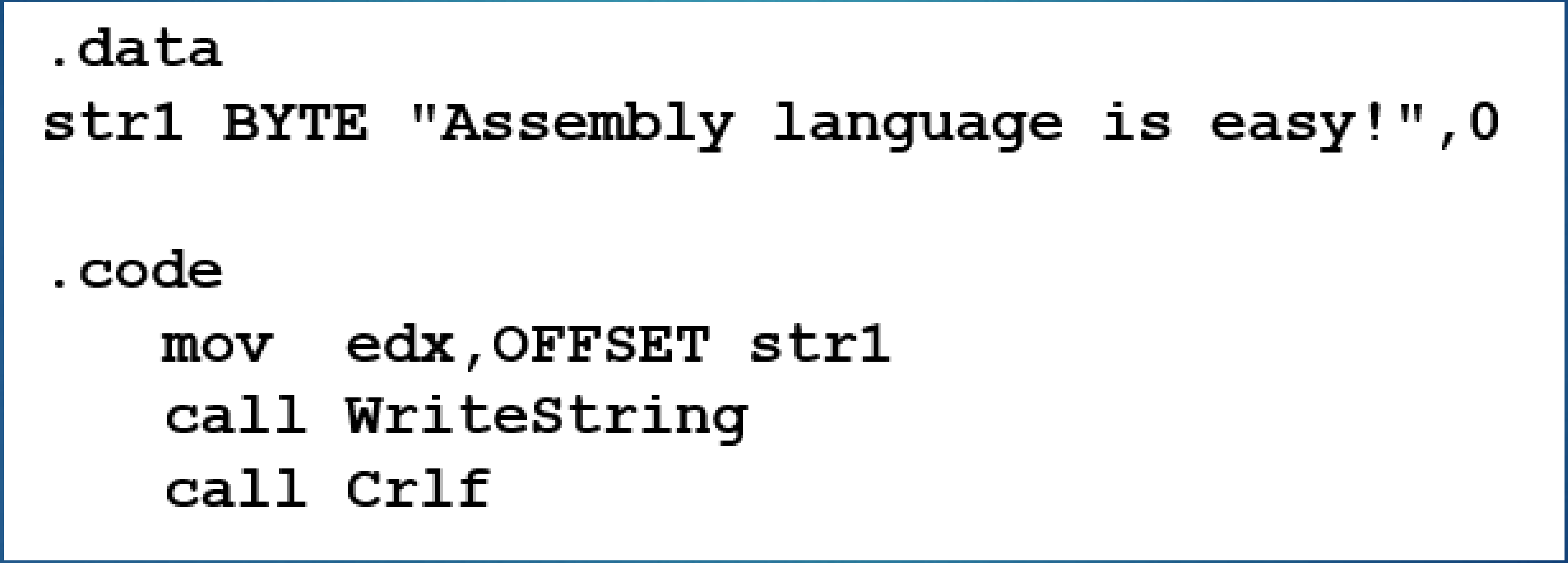
Sample output:

```
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

TASK:-

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



```
.data
str1 BYTE "Assembly language is easy!",0

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

Example 2a

TASK:-

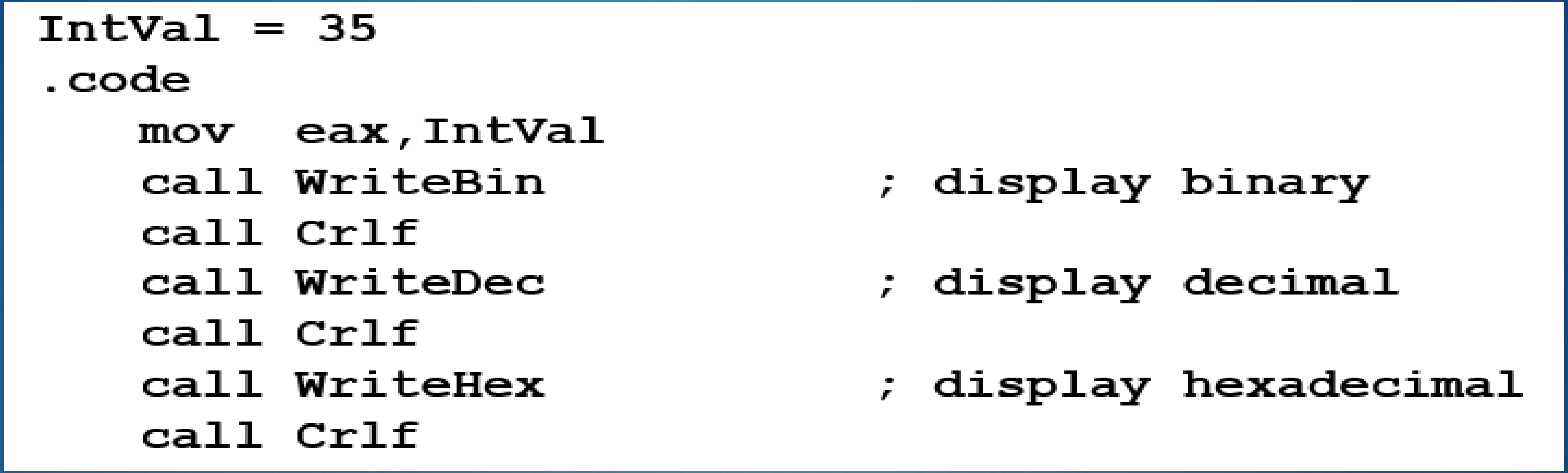
Display a null-terminated string and move the cursor to the beginning of the next screen line (use embedded CR/LF)

```
.data
str1 BYTE "Assembly language is easy!",0Dh,0Ah,0

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

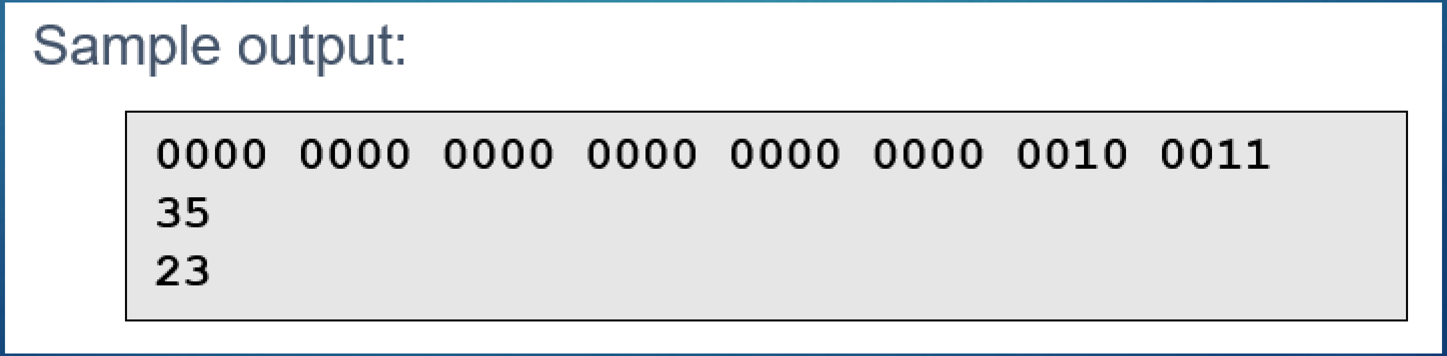
Example 3

TASK:-

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



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



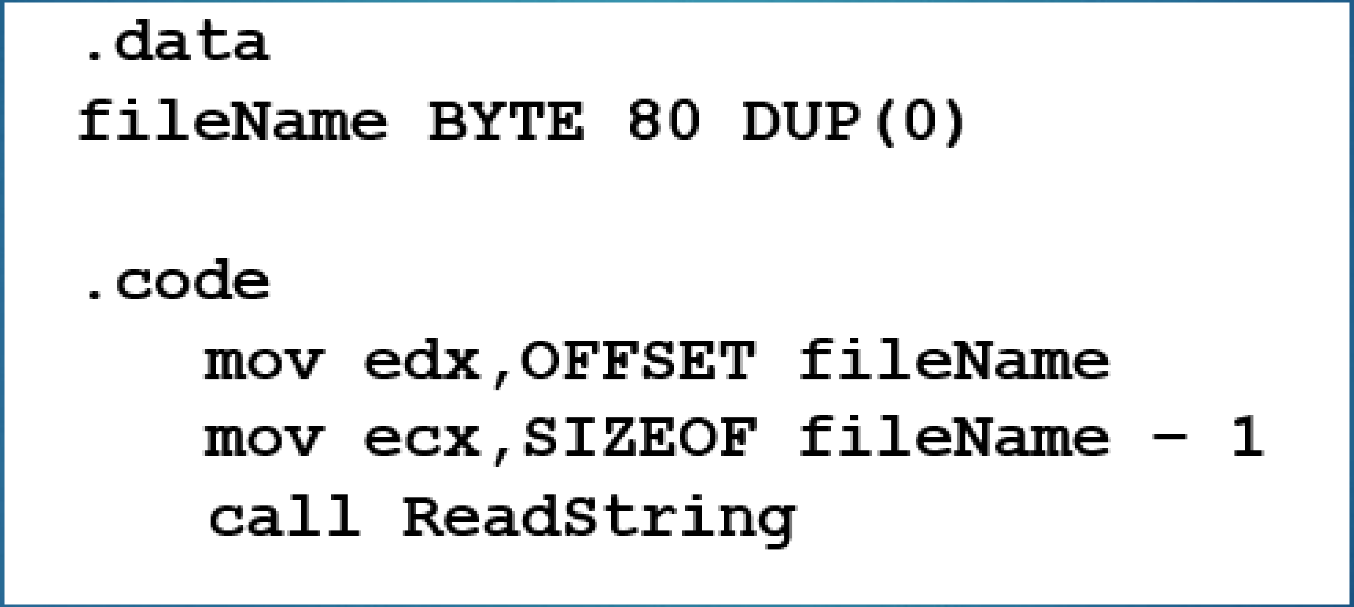
Sample output:

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

Example 4

TASK:-

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.

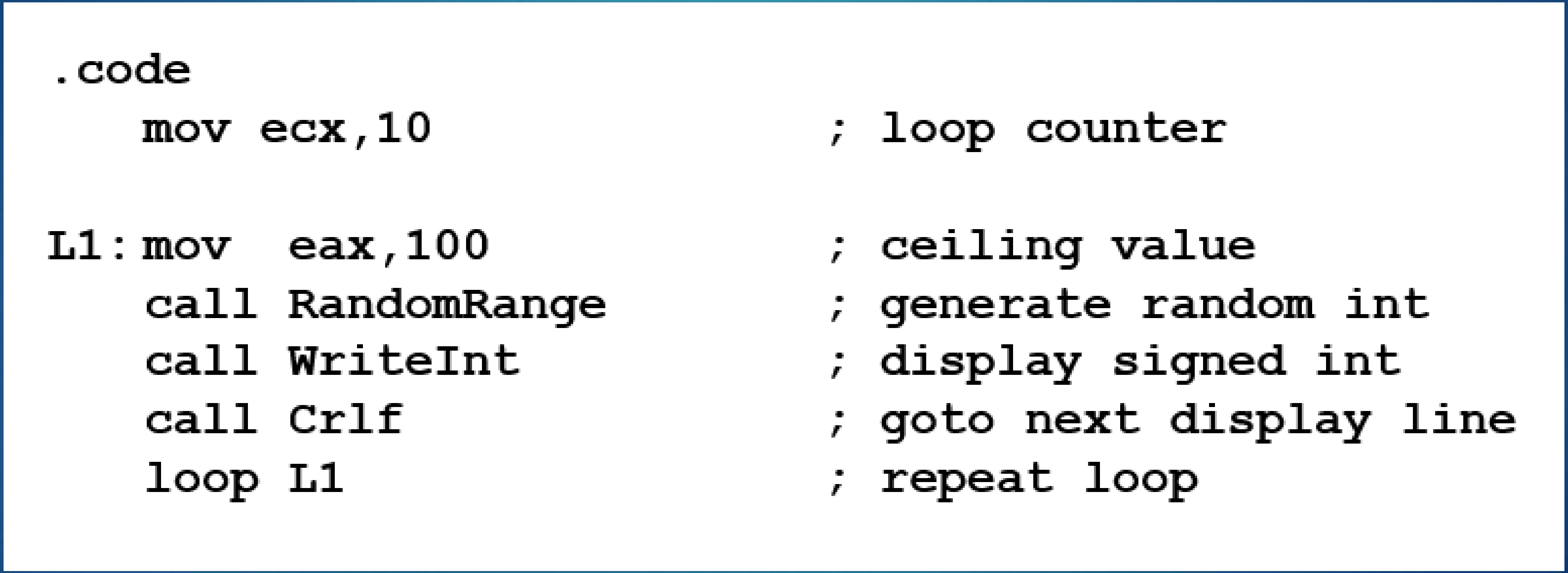


```
.data
fileName BYTE 80 DUP(0)

.code
    mov edx,OFFSET fileName
    mov ecx,SIZEOF fileName - 1
    call ReadString
```

Example 5

TASK:-

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

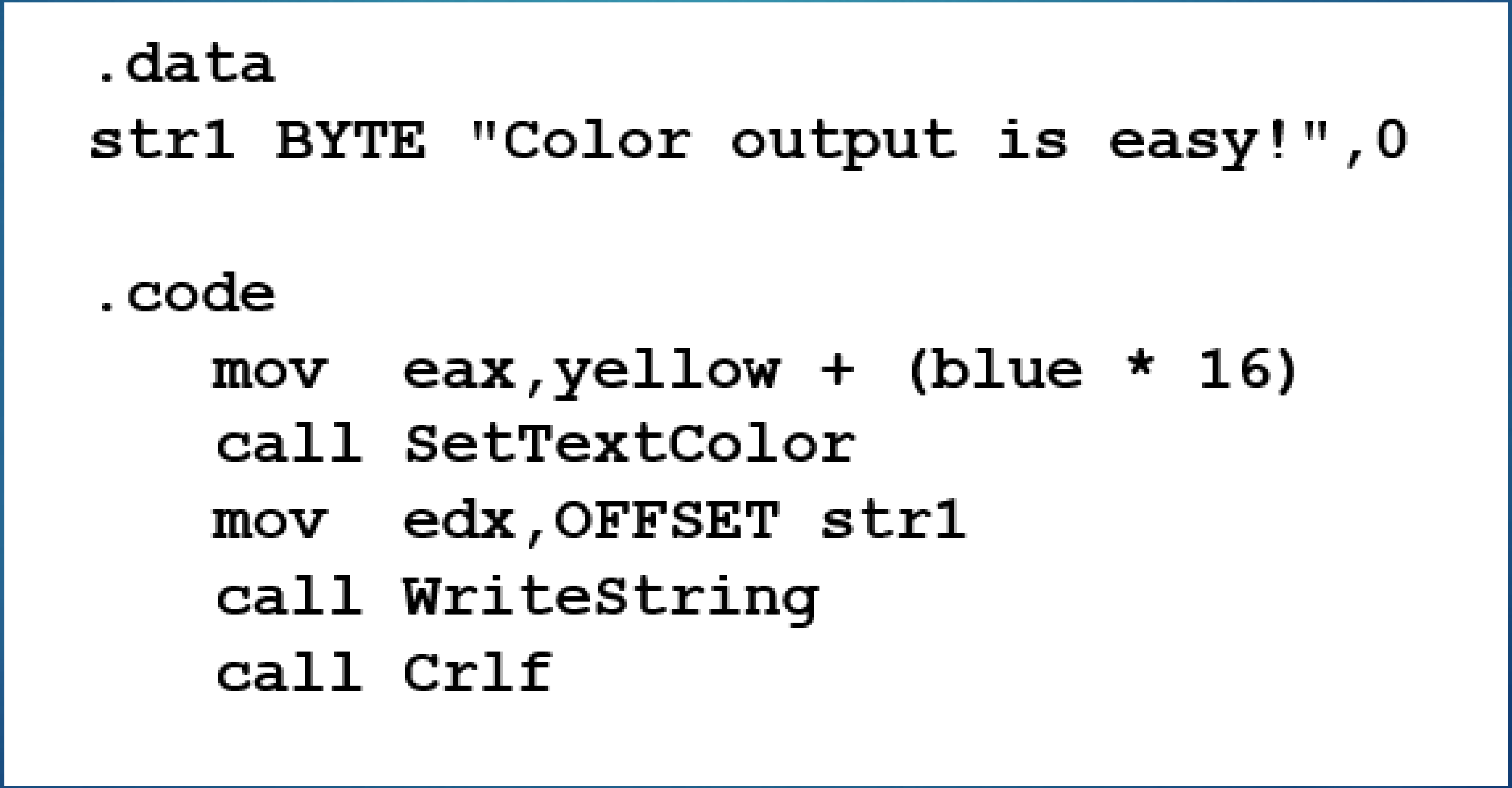


```
.code
    mov ecx,10                  ; loop counter

L1: mov  eax,100                ; ceiling value
    call RandomRange            ; generate random int
    call WriteInt               ; display signed int
    call Crlf                   ; goto next display line
    loop L1                     ; repeat loop
```

Example 6

TASK:-

Display a null-terminated string with yellow characters on a blue background.



```
.data
str1 BYTE "Color output is easy!",0

.code
   mov  eax,yellow + (blue * 16)
   call SetTextColor
   mov  edx,OFFSET str1
   call WriteString
   call Crlf
```

Procedure parameters

- A good procedure might be usable in many different programs
- Parameters help to make procedures flexible because parameter values can change at runtime
- General registers can be used to pass parameters

Documenting Procedures

Suggested documentation for each procedure:

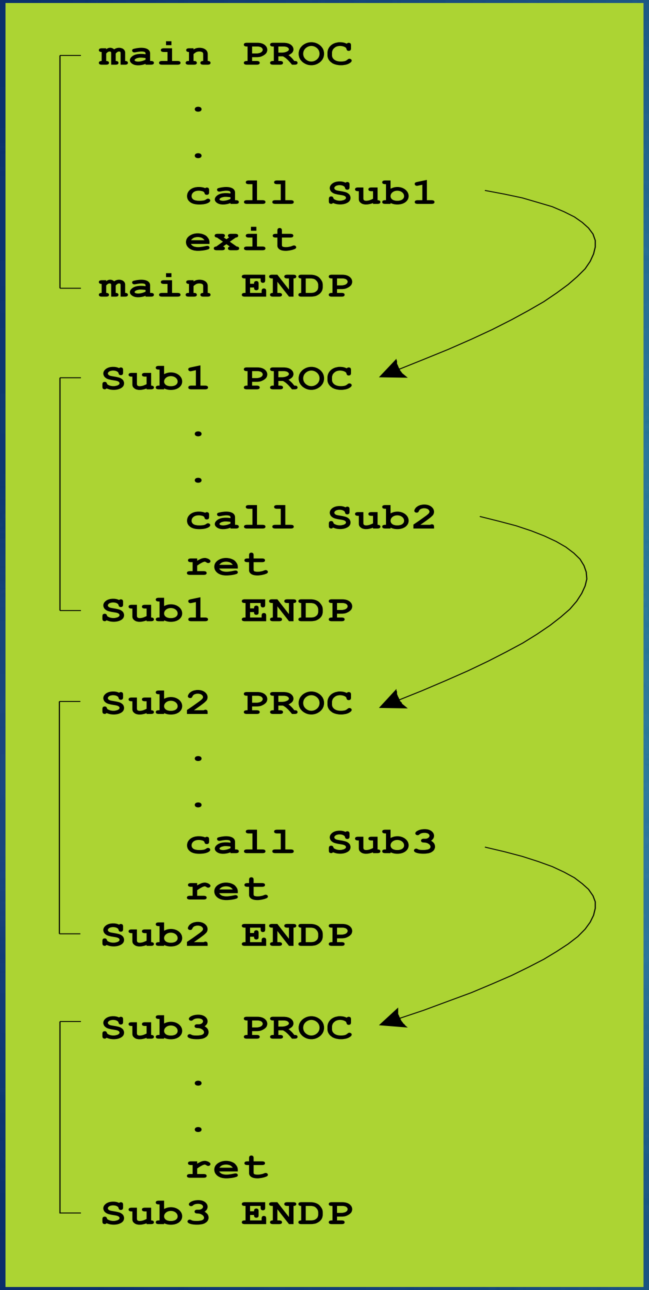
- A description of all tasks accomplished by the procedure.
- **Receives:** A list of input parameters; state their usage and requirements.
- **Returns:** A description of values returned by the procedure.
- **Requires:** Optional list of requirements called preconditions that must be satisfied before the procedure is called.

If a procedure is called without its preconditions satisfied, it will probably not produce the expected output.

Example: SumOf Procedure

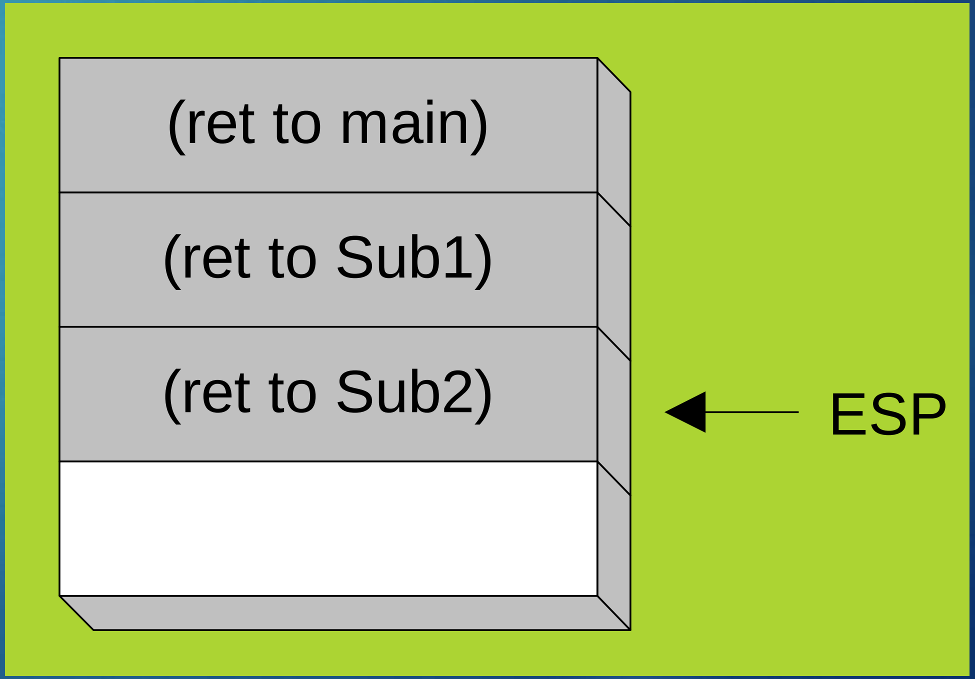
```
;---------------------------------------------------------
SumOf PROC
;
; Calculates and returns the sum of three 32-bit integers.
; Receives: EAX, EBX, ECX, the three integers. May be
; signed or unsigned.
; Returns: EAX = sum, and the status flags (Carry,
; Overflow, etc.) are changed.
; Requires: nothing
;---------------------------------------------------------
   add eax,ebx
   add eax,ecx
   ret
SumOf ENDP
```

Nested Procedure Calls



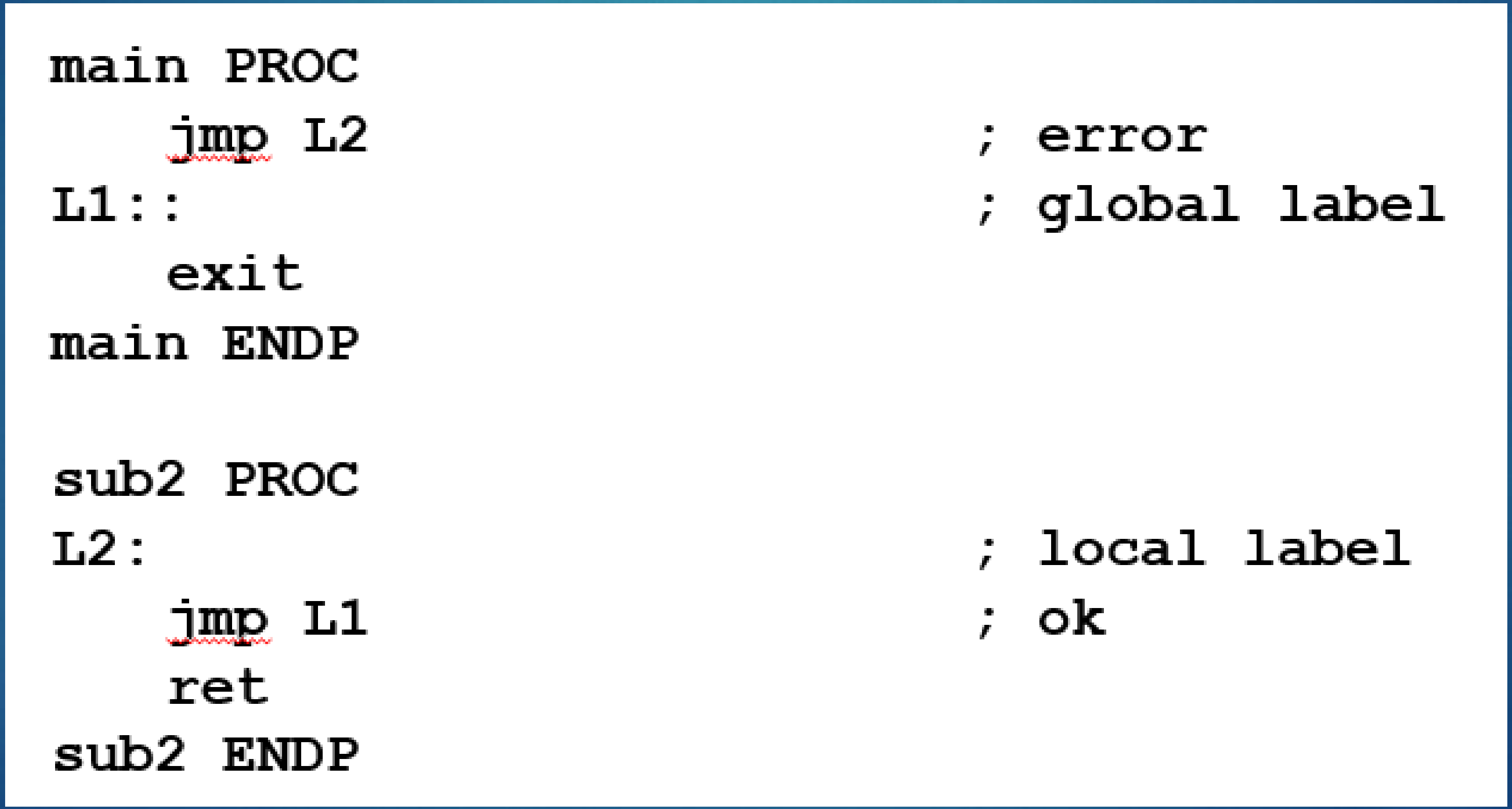
```
main  PROC
      .
      .
      call  Sub1
      exit
main  ENDP

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:



Local and Global labels

- A local label is visible only to statements inside the same procedure
- A global label is visible everywhere . Global label identified by double colon (::)



```
main PROC
    jmp L2                    ; error
L1::                          ; global label
    exit
main ENDP

sub2 PROC
L2:                           ; local label
    jmp L1                    ; ok
    ret
sub2 ENDP
```

Example 7

TASK:-

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

L1: add eax,myArray[esi]        ; add each integer to sum
    add esi,4                   ; point to next integer
    loop L1                     ; repeat for array size

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

EXERCISE

What if you wanted to calculate the sum of two or three arrays within the same program?

Example 8

TASK:-

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

```
ArraySum PROC
; Receives: ESI points to an array of doublewords,
;    ECX = number of array elements.
; Returns: EAX = sum
;-----------------------------------------------------
    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
```

USES Operator

- Lists the registers that will be saved (to avoid side effects) (return register shouldn't be saved)

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

- MASM generates the following code

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

When not to push a register

The sum of the three registers is stored in EAX on line (3), but the POP instruction replaces it with the starting value of EAX on line (4):

```
SumOf PROC                          ; sum of three integers
    push eax                        ; 1
    add eax,ebx                     ; 2
    add eax,ecx                     ; 3
    pop eax                         ; 4
    ret
SumOf ENDP
```

RET Instruction

- Return from subroutine
- Pops stack into the instruction pointer (EIP or IP), control transfers to the target address
- Syntax:
  - RET
  - RET n
- Optional operand n causes n bytes to be added to the stack pointer after EIP (or IP) is assigned a value