Computer Organization and Assembly Language (COAL)

Lecture 5

Dr. Naveed Anwar Bhatti

Webpage: naveedanwarbhatti.github.io

Procedures

- **Stack Operations**
- Defining and Using Procedures

Stack Operations

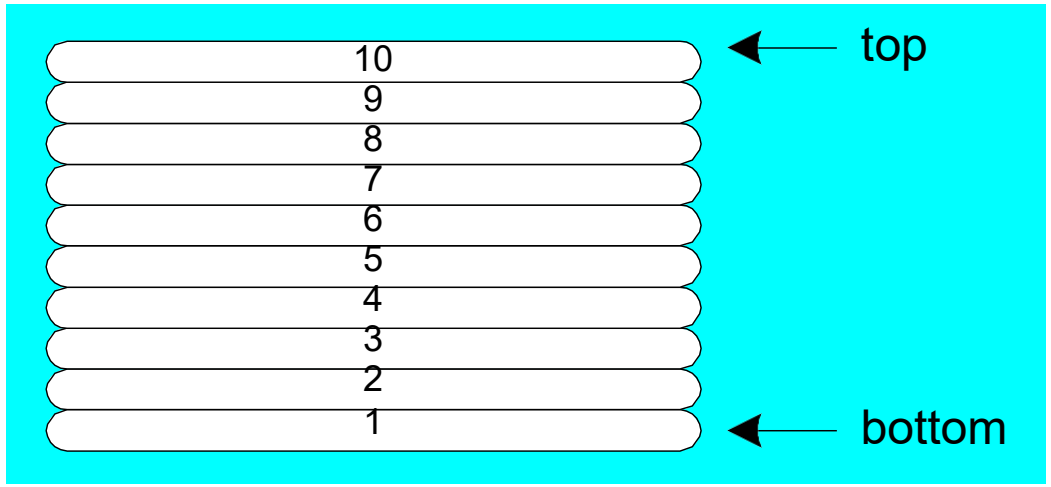
- Runtime Stack
- PUSH Operation
- POP Operation
- PUSH and POP Instructions
- Using PUSH and POP
- Example: Reversing a String
- Related Instructions

Stack and Stack Pointers

- The stack is a section of **RAM** used by the CPU to store information temporarily
  - This information could be data or an address
- The CPU needs this storage area because there are only a limited number of registers
- There must be a register inside the CPU to point to stack in the memory
- The register used to access the stack is called the SP (stack pointer) register.

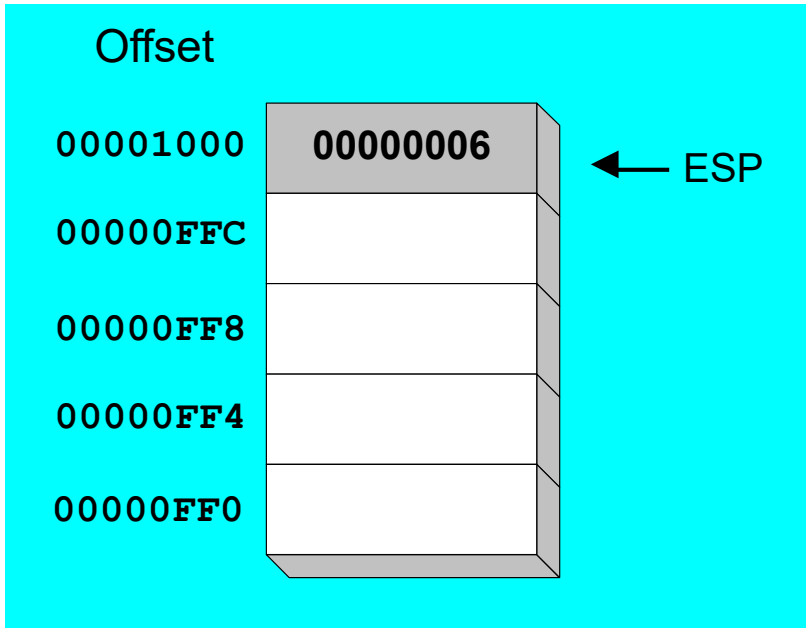
Runtime Stack

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



Runtime Stack

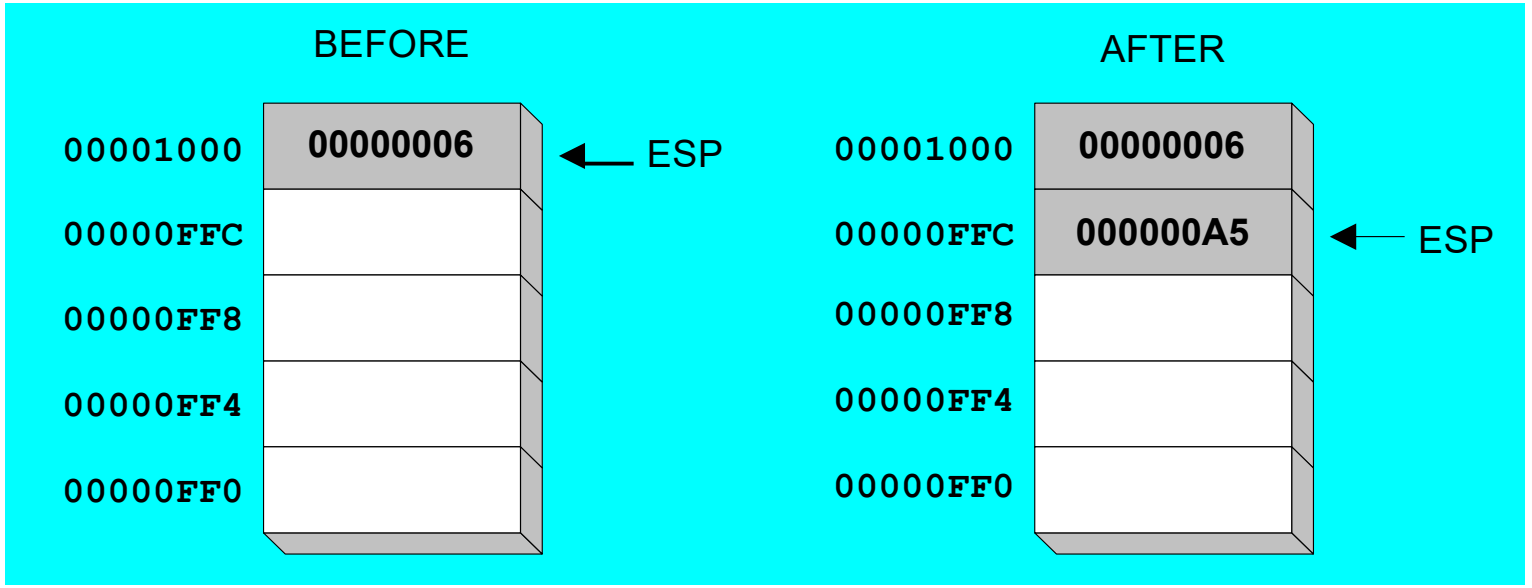
- Managed by the CPU, using two registers
  - SS (stack segment)
  - ESP (stack pointer) *



* SP in Real-address mode

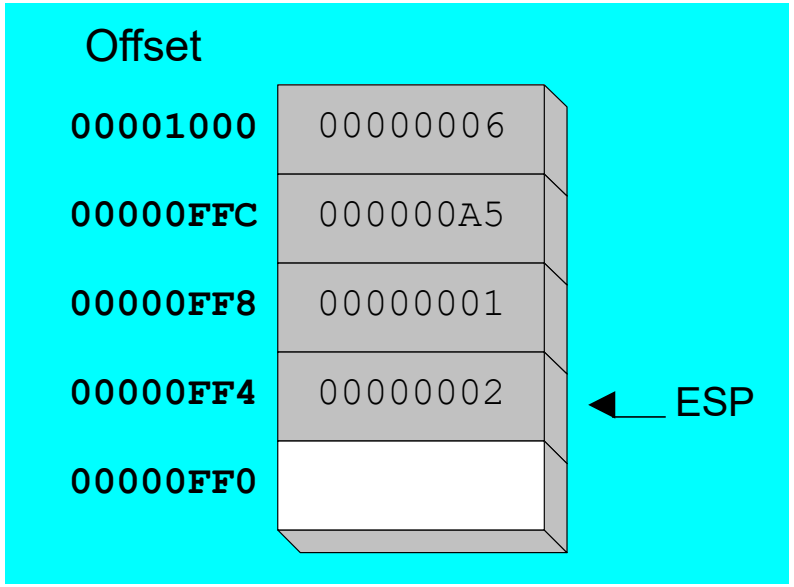
PUSH Operation (1 of 2)

- A **32-bit push** operation decrements the stack pointer by **4** and copies a value into the location pointed to by the stack pointer.



PUSH Operation (2 of 2)

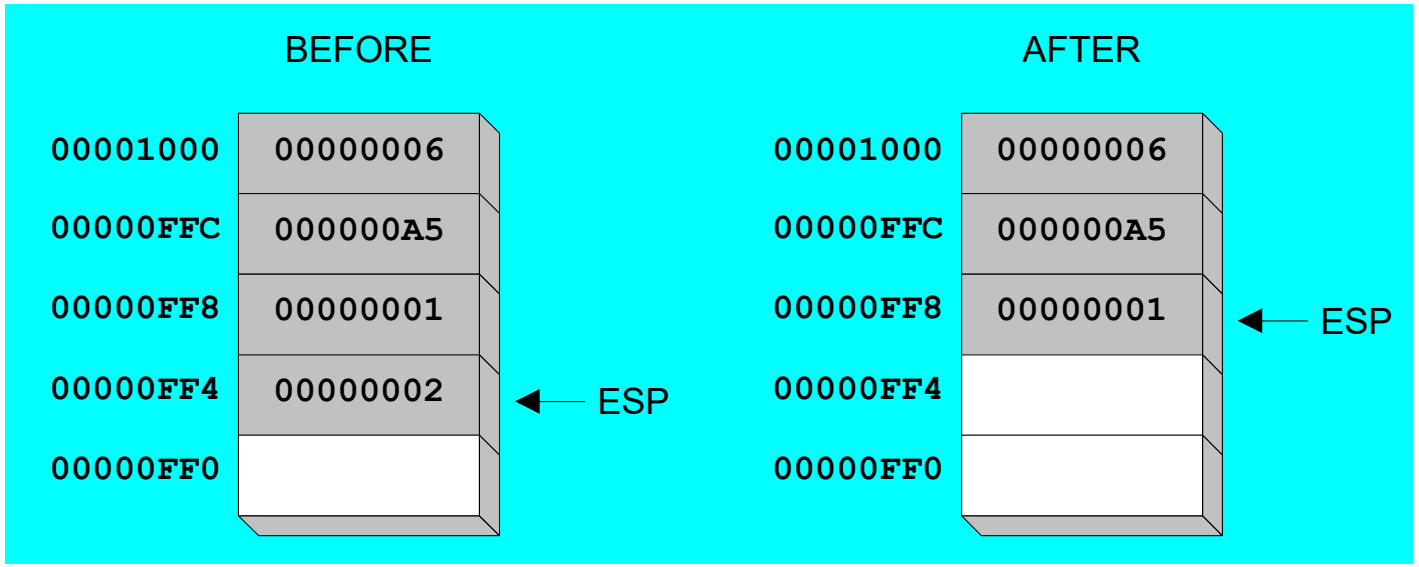
- Same stack after pushing two more integers:



The stack grows **downward**. The area below ESP is always available (unless the stack has overflowed).

POP Operation

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



PUSH and POP Instructions

- PUSH syntax:
  - PUSH *r/m16*
  - PUSH *r/m32*
  - PUSH *imm32*
- POP syntax:
  - POP *r/m16*
  - 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.

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

mov  esi,OFFSET dwordVal          ; display some memory
mov  ecx,LENGTHOF dwordVal
mov  ebx,TYPE dwordVal
.....
```

Using PUSH and POP

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

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

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

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

Recall Nested Loop Example

Remember the nested loop we created in nested loop lecture?

If you need to code a loop within a loop, you must save the outer loop counter's **ECX value**. In the following example, the outer loop executes 100 times, and the inner loop 20 times.

```
.data
count DWORD ?
.code
    mov ecx,100             ; set outer loop count
L1:
    mov count,ecx           ; save outer loop count
    mov ecx,20              ; set inner loop count
L2: .
    .
    loop L2                 ; repeat the inner loop
    mov ecx,count           ; restore outer loop count
    loop L1                 ; repeat the outer loop
```

Nested Loop with Push/Pop

Remember the nested loop we created in nested loop lecture?

It's easy to **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
```

Copying a String Example

The following code copy a string from source to target:

```
.data
source  BYTE  "This is the source string",0
target  BYTE  SIZEOF source DUP(0)

.code
    mov  si,0                       ; index register
    mov  cx,SIZEOF source           ; loop counter
L1:
    mov  al,source[si]              ; get char from source
    mov  target[si],al              ; store it in the target
    inc  si                         ; move to next character
    loop L1                         ; repeat for entire string
```

Example: Reversing a String

- RE-program the code using Push/Pop statements to reverse the string

```
.data
source  BYTE  "This is the source string",0
target  BYTE  SIZEOF source DUP(0)

.code
    mov  si,0                    ; index regis
    mov  cx,SIZEOF source        ; loop counter
L1:
    mov  al,source[si]           ; get char from source
    mov  target[si],al           ; store it in the target
    inc  si                      ; move to next character
    loop L1                      ; repeat for entire string
```

Solution:
1. Use a loop with indexed addressing
2. Push each character on the stack
3. Start at the beginning of the string, pop the stack in reverse order, insert each character back into the string

What's Next

- Stack Operations

  Defining and Using Procedures
- Linking to an External Library

Defining and Using Procedures

- Creating Procedures
- Documenting Procedures
- Example: SumOf Procedure
- CALL and RET Instructions
- Nested Procedure Calls
- Local and Global Labels
- Procedure Parameters
- Flowchart Symbols
- USES Operator

Creating 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
```

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
```

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)

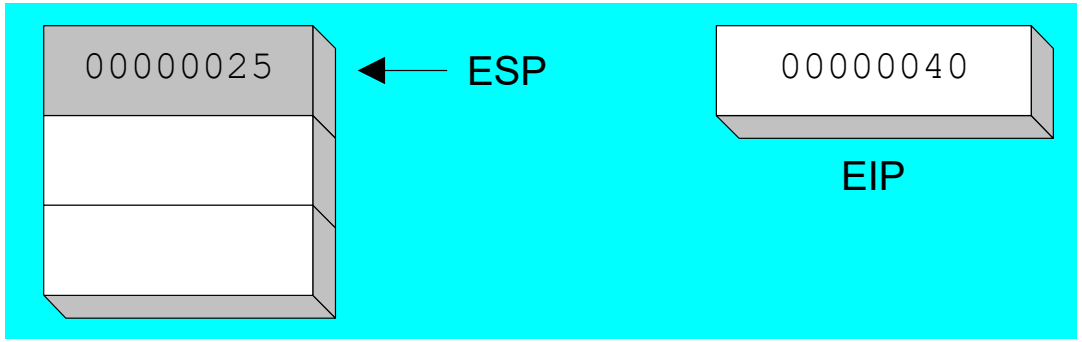
0000025 is the offset of the instruction immediately following the CALL instruction

00000040 is the offset of the first instruction inside MySub

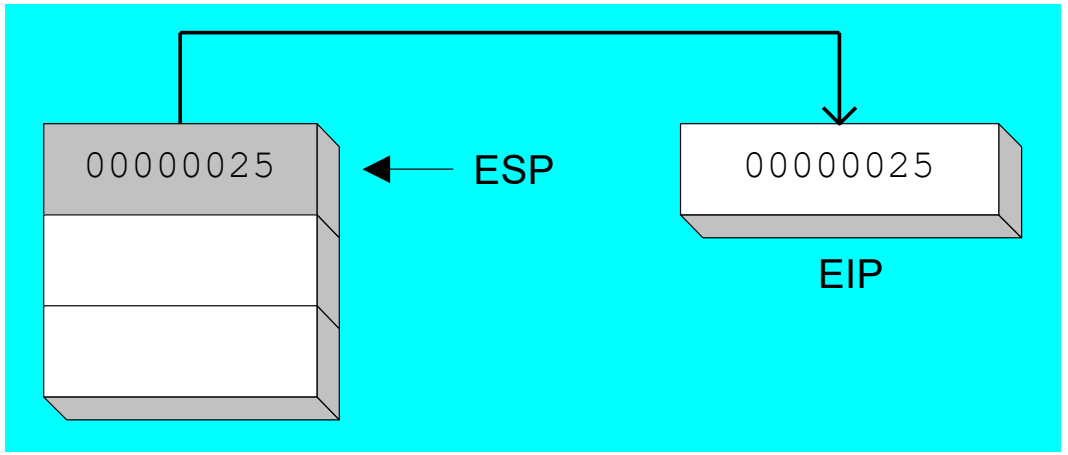
```
main PROC
    00000020 call MySub
    00000025 mov eax,ebx
    .
    .
main ENDP

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 00000025 from the stack into EIP

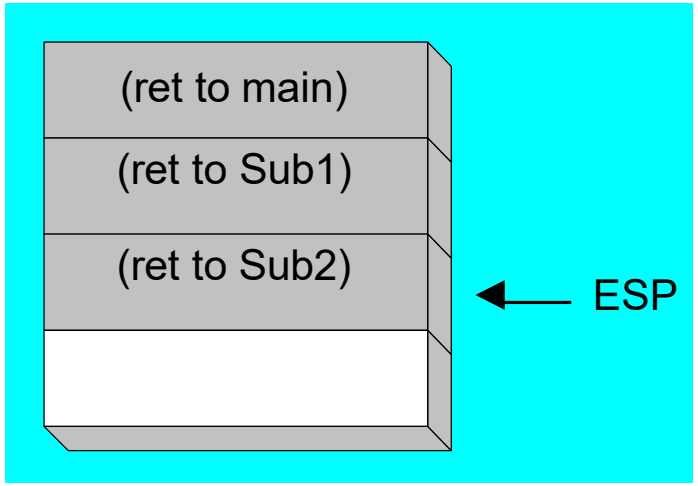


(stack shown before RET executes)

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

A local label is visible only to statements inside the same procedure. A global label is visible everywhere.

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

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

Procedure Parameters (1 of 3)

- A good procedure might be usable in many different programs
  - but not if it refers to **specific variable names**
- Parameters help to make procedures flexible because parameter values can change at runtime

Procedure Parameters (2/3)

```
; Testing the ArraySum procedure (TestArraySum.asm)

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

.data
array DWORD 10000h,20000h,30000h,40000h,50000h
theSum DWORD ?

.code
main PROC
     mov   esi,OFFSET array     ; ESI points to array
     mov   ecx,LENGTHOF array   ; ECX = array count
     call  ArraySum             ; calculate the sum
     mov   theSum,eax           ; returned in EAX

     INVOKE ExitProcess,0
main ENDP

;-----------------------------------------------------
; ArraySum
; Calculates the sum of an array of 32-bit integers.
; Receives: ESI = the array offset
; ECX = number of elements in the array
; Returns: EAX = sum of the array elements
;-----------------------------------------------------

ArraySum PROC
     push  esi                  ; save ESI, ECX
     push  ecx
     mov   eax,0                ; set the sum to zero
```

Procedure Parameters (3/3)

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

    pop   ecx                 ; restore ECX, ESI
    pop   esi
    ret                       ; sum is in EAX
ArraySum ENDP

END main
```

USES Operator

- Lists the registers that will be preserved

```
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
```

When not to push a register

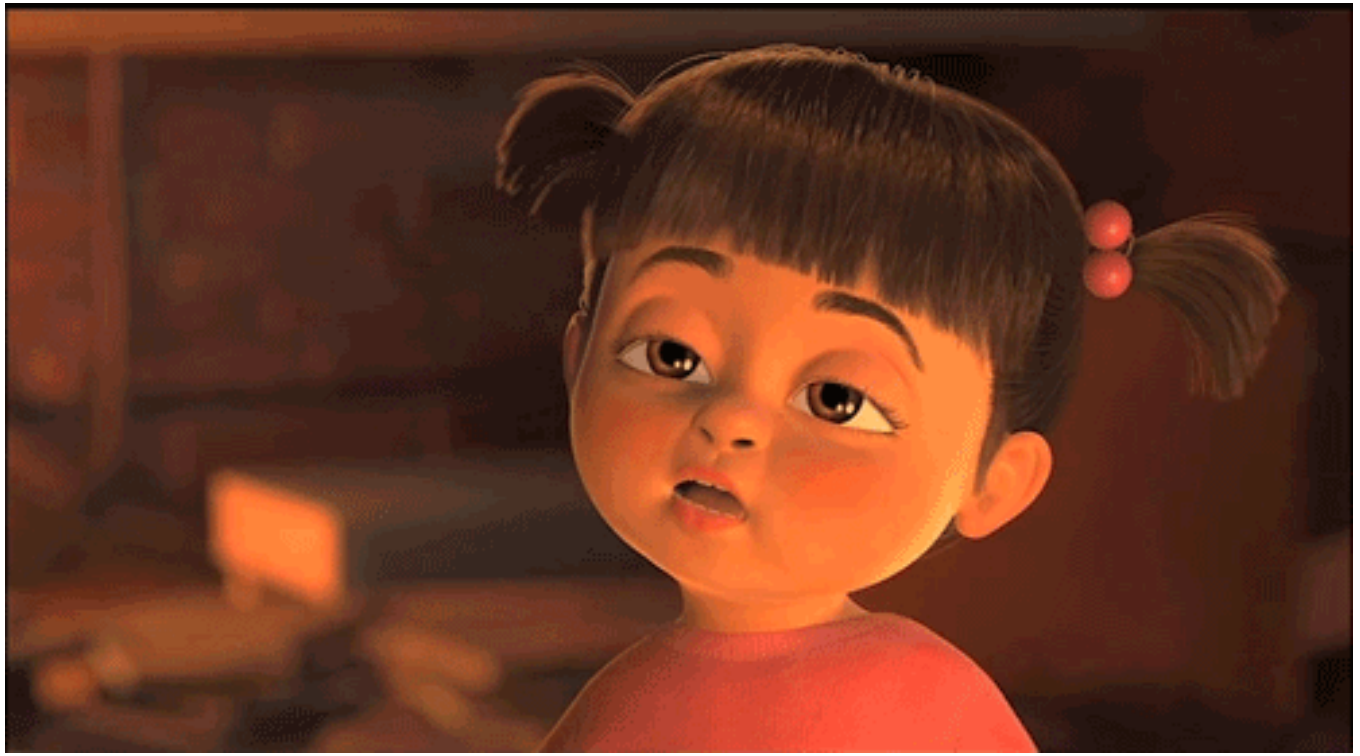
The sum of the three registers is stored in EAX on line (3), but the POP instruction replaces it with the starting value of EAX on line (4):

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

Summary

- Procedure – named block of executable code
- Runtime stack – LIFO structure
  - holds return addresses, parameters, local variables
  - PUSH – add value to stack
  - POP – remove value from stack

Thanks a lot



If you are taking a Nap, **wake up**.....Lecture Over