بسم الله الرّحمن الرّحيم

EE213 COMPUTER ORGANIZATION AND ASSEMBLY LANGUAGE
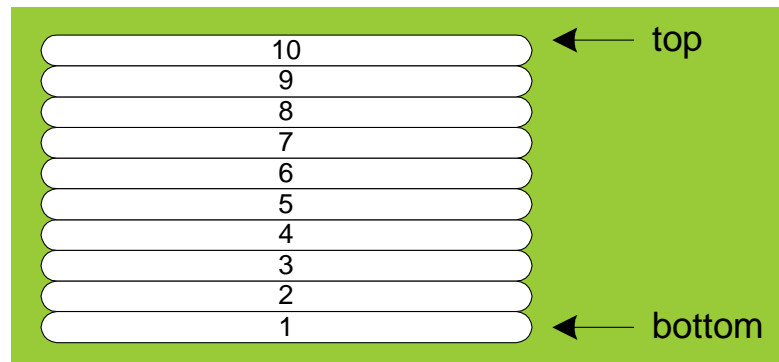
FALL 2018

5

PROCEDURES

OUTLINES

- 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

10	← top
9	
8	
7	
6	
5	
4	
3	
2	
1	← bottom

5.1 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.

- **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*.

- ESP always points to the last value to be added to, or *pushed* on, the top of stack.
  - We rarely manipulate ESP directly; instead, it is indirectly modified by instructions such as CALL, RET, PUSH, and POP

Offset

00001000	**00000006**	← ESP = 00001000h
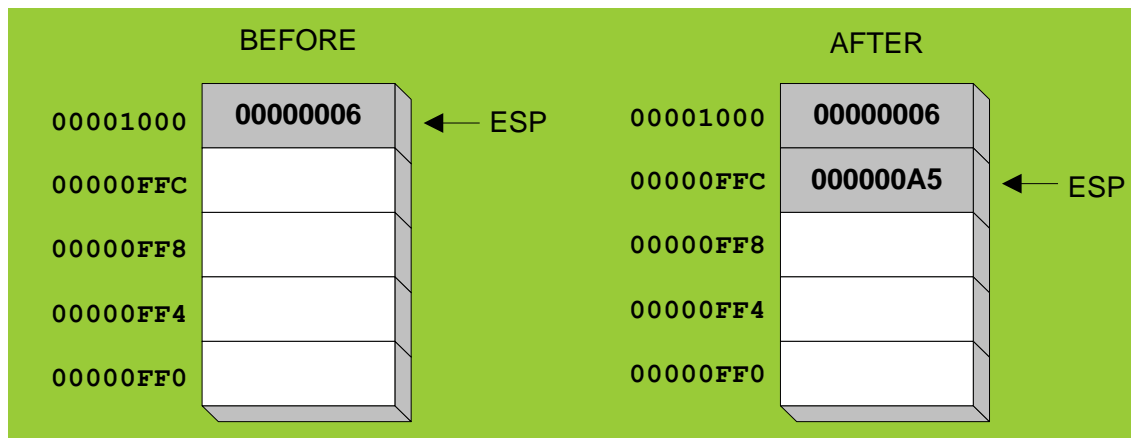| **00000FFC**	
00000FF8	
00000FF4	
00000FF0	

PUSH Operation

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

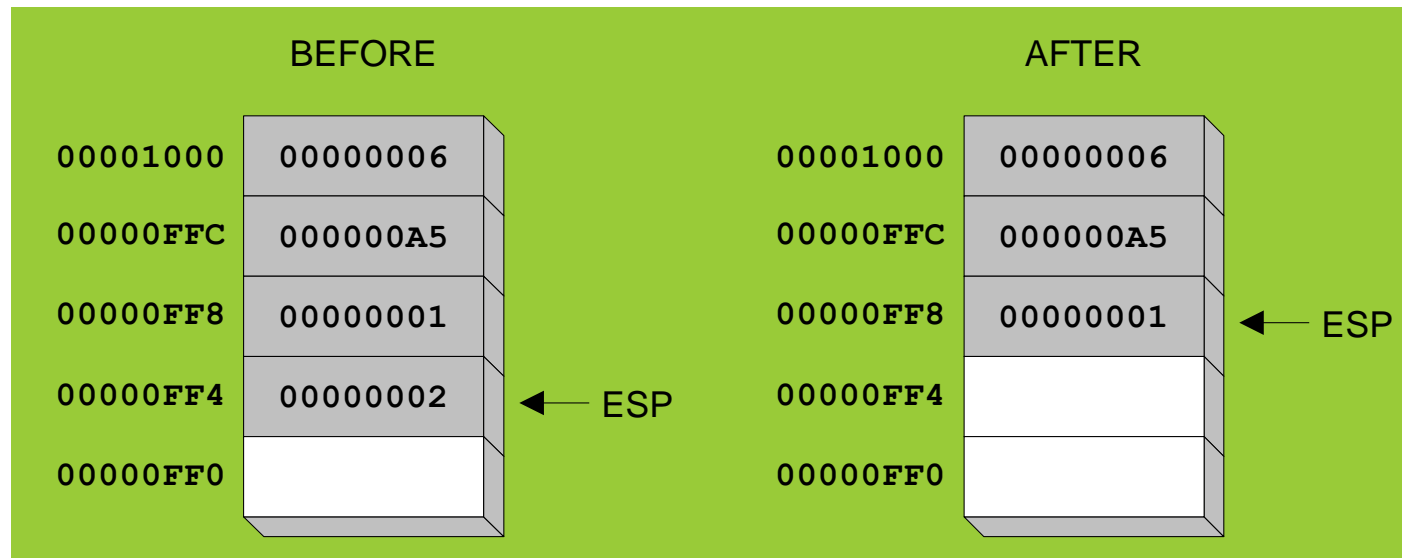|	BEFORE			AFTER	
00001000	**00000006**	← ESP	00001000	**00000006**	
00000FFC			00000FFC	**000000A5**	← ESP
00000FF8			00000FF8		
00000FF4			00000FF4		
00000FF0			00000FF0		

• Same stack after pushing two more integers:

Offset

00001000	00000006
00000FFC	000000A5
00000FF8	00000001
00000FF4	00000002	← ESP
00000FF0	
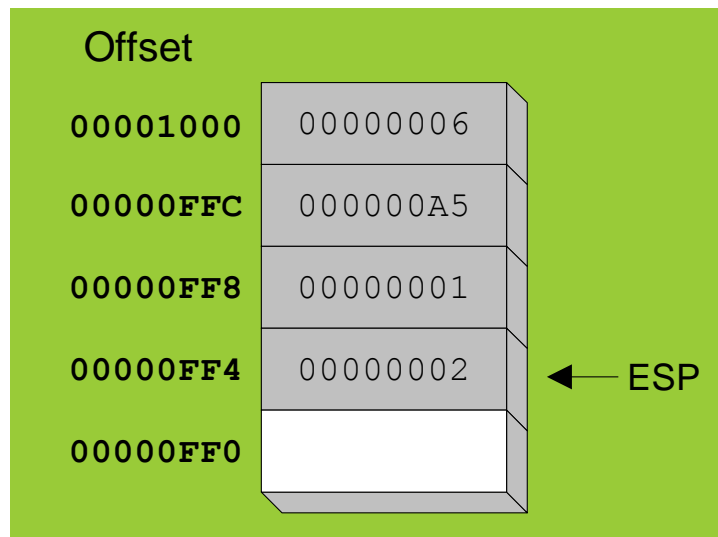
• runtime stack grows downward in memory, from higher addresses to lower addresses.

POP Operation

- A *pop operation* removes a value from the stack. After the value is popped from the stack, the stack pointer is incremented (by the stack element size) to point to the next-highest location in the stack.

	BEFORE			AFTER	
00001000	00000006		00001000	00000006	
00000FFC	000000A5		00000FFC	000000A5	
00000FF8	00000001		00000FF8	00000001	← ESP
00000FF4	00000002	← ESP	00000FF4		
00000FF0			00000FF0		

STACK APPLICATIONS

- A stack makes a convenient temporary save area for registers when they are used for more than one purpose.

- When the CALL instruction executes, the CPU saves the current subroutine's return address on the stack.

- When calling a subroutine, you pass input values called *arguments* by pushing them on the stack.

- The stack provides temporary storage for local variables inside subroutines.

PUSH AND POP INSTRUCTIONS

- The **PUSH** instruction first decrements ESP and then copies a source operand into the stack.
  - A 16-bit operand causes ESP to be decremented by 2.
  - A 32-bit operand causes ESP to be decremented by 4.

```
PUSH  reg/mem16
PUSH  reg/mem32
PUSH  imm32
```

- The **POP** instruction first copies the contents of the stack element pointed to by ESP into a 16- or 32-bit destination operand and then increments ESP.
  - If the operand is 16 bits, ESP is incremented by 2; if the operand is 32 bits, ESP is incremented by 4:

```
POP reg/mem16
POP reg/mem32
```

PUSHAD,PUSHA, POPAD, POPA

- The **PUSHAD** instruction pushes all of the 32-bit general-purpose registers on the stack in the following order: EAX, ECX, EDX, EBX, ESP (value before executing PUSHAD), EBP, ESI, and EDI.

- The **POPAD** instruction pops the same registers off the stack in reverse order.

- Similarly, the **PUSHA** instruction, pushes the 16-bit general-purpose registers (AX, CX, DX, BX, SP, BP, SI, DI) on the stack in the order listed. The POPA instruction pops the same registers in reverse.

```
.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

; 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

    INVOKE ExitProcess,0
main ENDP
END main
```

EXAMPLE: REVERSING A STRING

5.2 DEFINING AND USING PROCEDURES

- A complicated problem is usually divided into separate tasks (subroutines).
  - We typically use the term *procedure* to mean a subroutine.

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

- Must be assigned a name (valid identifier).

```
main PROC
    .
main ENDP
```

- 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
```

- **Labels in Procedures** are visible only within the procedure in which they are declared.
  - In the following example, the label named *Destination* must be located in the same procedure as the `JMP` instruction:

    ```
    jmp Destination
    ```

- It is possible to work around this limitation by declaring a *global label*, identified by a double colon (::) after its name:

    ```
    Destination::
    ```

CALL AND RET INSTRUCTIONS

- The **CALL** instruction calls a procedure by directing the processor to begin execution at a new memory location
  - Pushes offset of next (instruction after call) 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

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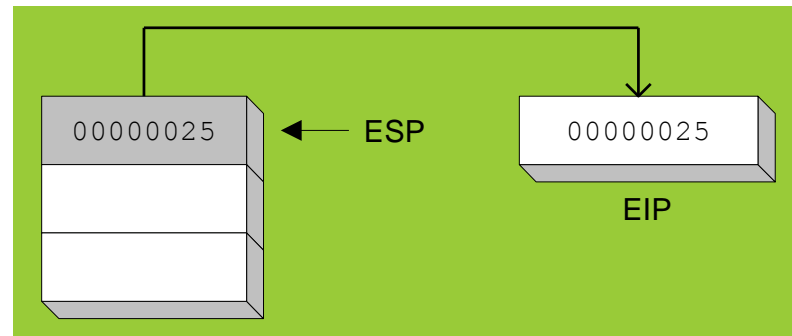
The CALL instruction pushes 00000025 onto the stack, and loads 00000040 into EIP
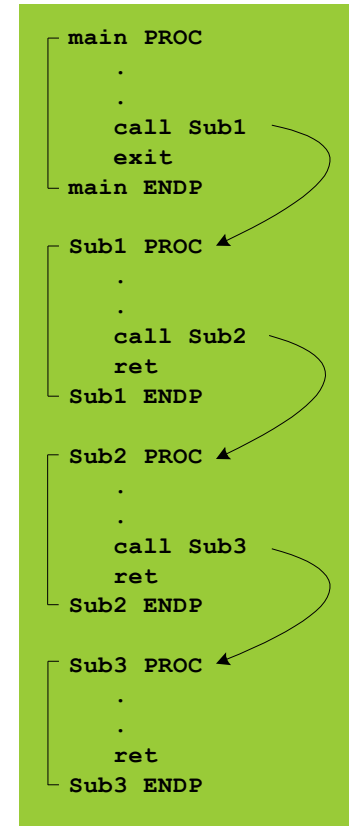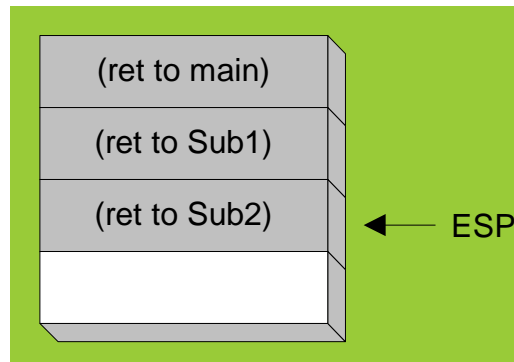

The RET instruction pops 00000025 from the stack into EIP

(stack shown before RET executes)

- A *nested procedure call* occurs when a called procedure calls another procedure before the first procedure returns.

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

(ret to main)
(ret to Sub1)
(ret to Sub2)	← ESP

```
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
```

In assembly language, it is common to **pass arguments inside general-purpose** registers.

```
.data
theSum  DWORD  ?
.code
main PROC
    mov    eax,10000h                    ; argument
    mov    ebx,20000h                    ; argument
    mov    ecx,30000h                    ; argument
    call   Sumof                         ; EAX = (EAX + EBX + ECX)
    mov    theSum,eax                    ; save the sum

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

SUMMARY

- Stack

- Stack Operations
  - PUSH
  - POP

- Defining and using Procedures

- Call and Ret instructions