Microprocessors (0630371)
Fall 2010/2011 – Lecture Notes # 15

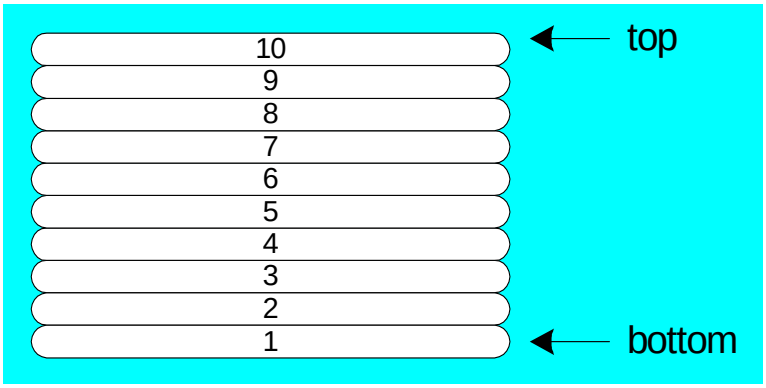
Stack Operations

Objectives of the Lecture

- **Runtime Stack**
- **PUSH Operation**
- **POP Operation**
- **Initializing the Stack**
- **PUSH and POP Instructions**
- **Stack Applications ------Using PUSH and POP**
- **Related Instructions**
- **Example: Reversing a String**

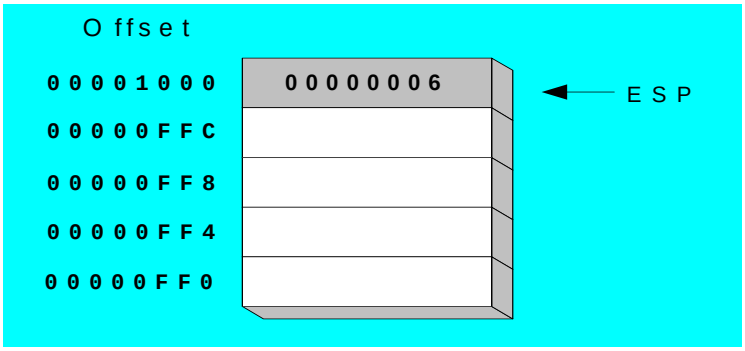
Runtime Stack

- Imagine a stack of plates . . .
  - plates are only added to the top
  - plates are only removed from the top
  - **LIFO** structure (last-in first-out)

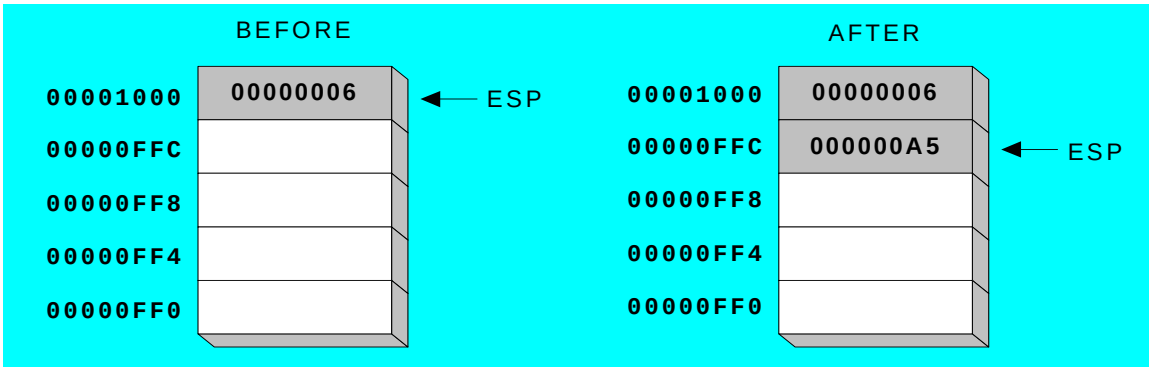


- Managed by the CPU, using two registers
  - **SS** (stack segment)
  - **ESP** (stack pointer) -------(SP in Real-address mode)

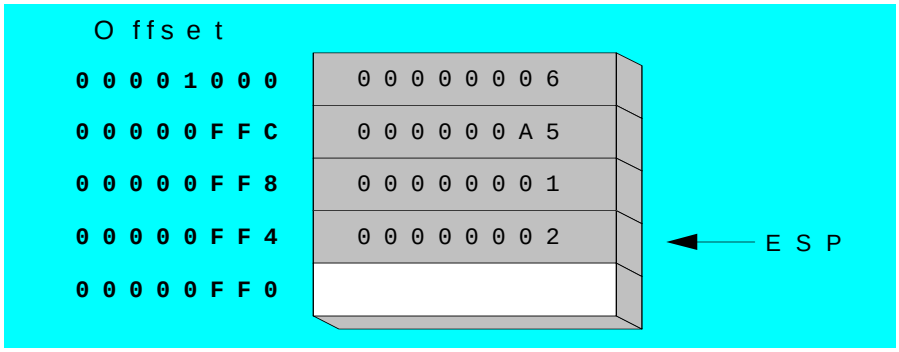
PUSH Operation



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



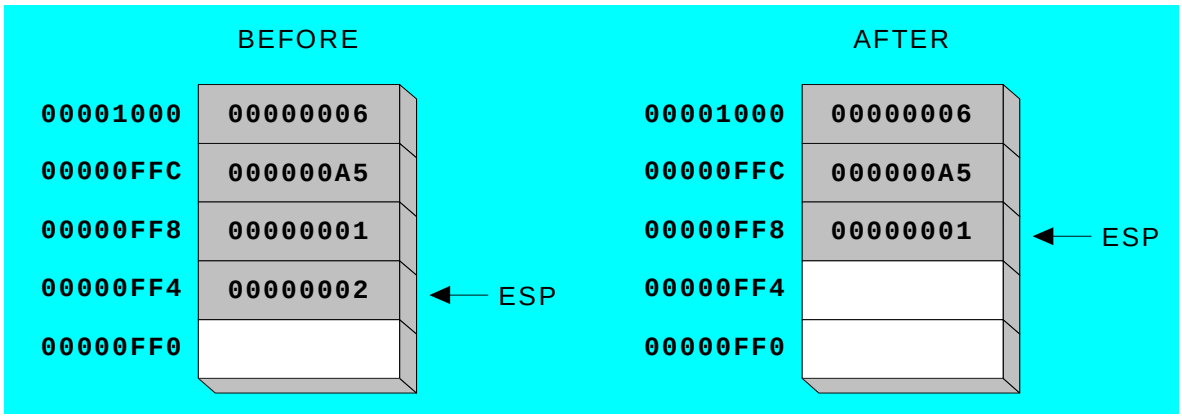
- Same stack after pushing two more integers:



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

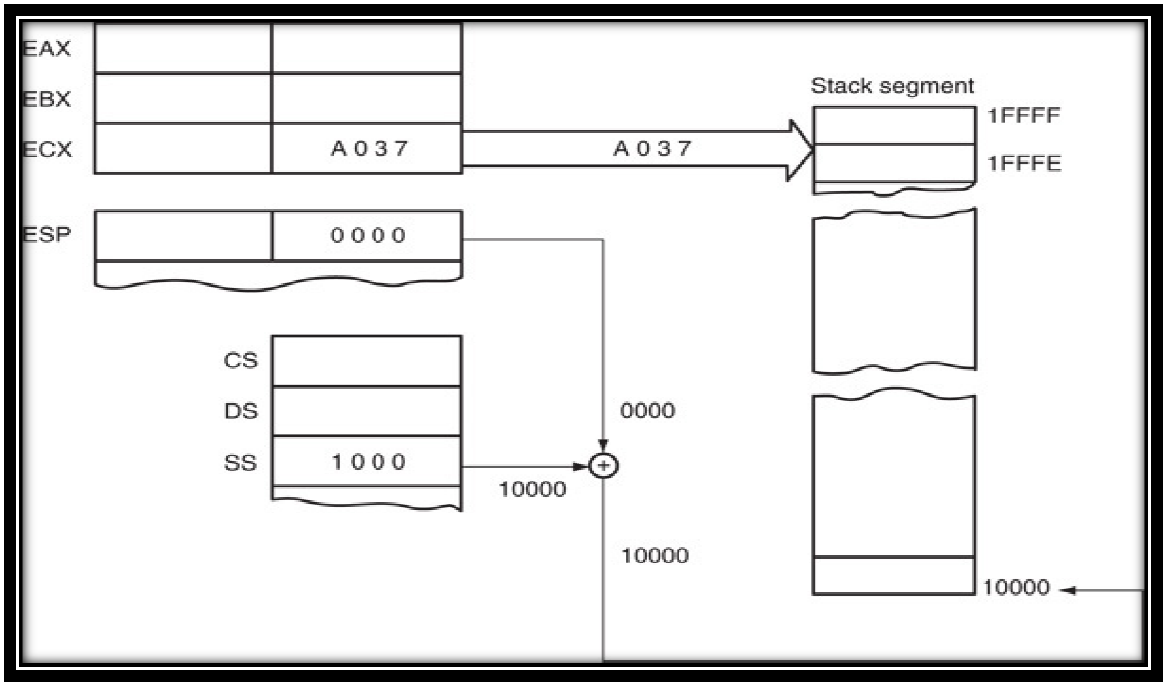
POP Operation

- 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



Initializing the Stack

- When the stack area is initialized, load both the stack segment (SS) register and the stack pointer (SP) register.
- Figure shows how this value causes data to be pushed onto the top of the stack segment with a **PUSH CX** instruction.
- All segments are cyclic in nature.
  - the top location of a segment is contiguous with the bottom location of the segment
- The **PUSH CX** instruction, showing the cyclical nature of the stack segment. This instruction is shown just before execution, to illustrate that the stack bottom is contiguous to the top.



PUSH and POP Instructions

PUSH syntax:

```
PUSH reg/m16
PUSH reg/m32
PUSH imm32
```

- Always transfers 2 bytes of data to the stack;
  - 80386 and above transfer 2 or 4 bytes

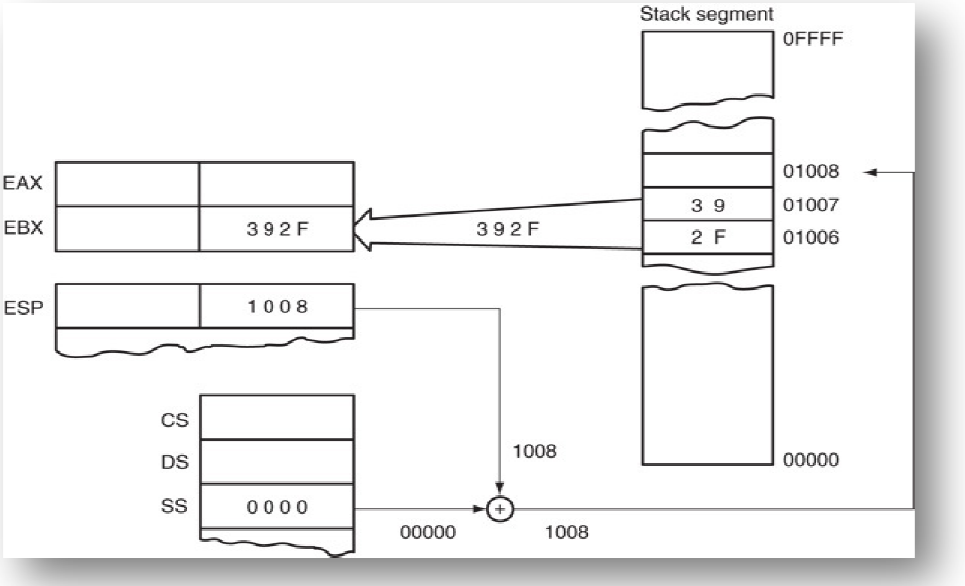
POP syntax:

```
POP reg/m16
POP reg/m32
```

- Performs the inverse operation of PUSH.
- POP removes data from the stack and places it in a target 16-bit register, segment register, or a 16-bit memory location.
  - not available as an immediate POP

- **Example:**

```
POP BX
```



Stack Applications ---- Using PUSH and POP

- Saves procedure linking information on the stack
- Local Variables for Calling Procedure.
- Parameters Passed to Called Procedure
- Storing the contents of the registers including the flag register.

Example1: 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
call DumpMem
pop ebx                     ; restore registers
pop ecx
pop esi
```

Example 2: Nested Loop: Remember the nested loop we created in previous 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
```

Related Instructions

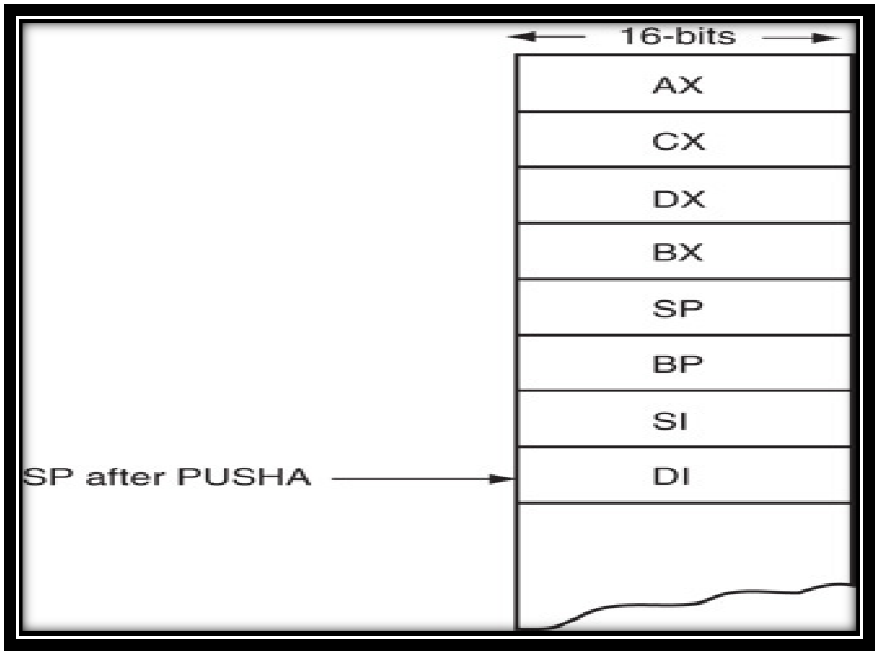
Flags:

- **PUSHF** (**push flags**) instruction copies the contents of the flag register (**FLAGS**) to the stack.
- **POPF** instruction retrieves and loads the flag register (**FLAGS**) from the stack.
- **PUSHFD** and **POPFD**
  - push and pop the **EFLAGS** register

General-purpose registers

- **PUSHA** instruction copies contents of the internal register set, except the segment registers, to the stack.
  - PUSHA (**push all**) instruction copies the registers to the stack in the following order: **AX, CX, DX, BX, SP, BP, SI, and DI**.
- **PUSHAD** pushes the 32-bit general-purpose registers on the stack
  - order: **EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI**
- **POPAD** pops the same registers (32-bit general-purpose registers) off the stack in reverse order
- **POPA** do the same for 16-bit registers
- **Example:**

  PUSHA
  - Requires 16 bytes of stack memory space to store all eight 16-bit registers.
  - After all registers are pushed, the contents of the SP register are decremented by 16.
  - PUSHA is very useful when the entire register set of 80286 and above must be saved.
  - PUSHAD instruction places 32-bit register set on the stack in 80386 - Core2.
  - PUSHAD requires 32 bytes of stack storage



Example: Reversing a String

- Use a loop with indexed addressing
- Push each character on the stack
- Start at the beginning of the string, pop the stack in reverse order, insert each character back into the string

Programming Example

```
TITLE Reversing a String          (RevStr.asm)
; This program reverses a string.
INCLUDE Irvine32.inc
.data
aName BYTE "Abraham Lincoln",0
nameSize = ($ - aName) - 1
.code
main PROC
; Push the name onto 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
; Display the name.
     mov  edx,OFFSET aName
     call Writestring
     call Crlf
     exit
main ENDP
END main
```

Q: Why must each character be put in EAX before it is pushed?
Because only word (16-bit) or double word (32-bit) values can be pushed on the stack.