What is a Stack?

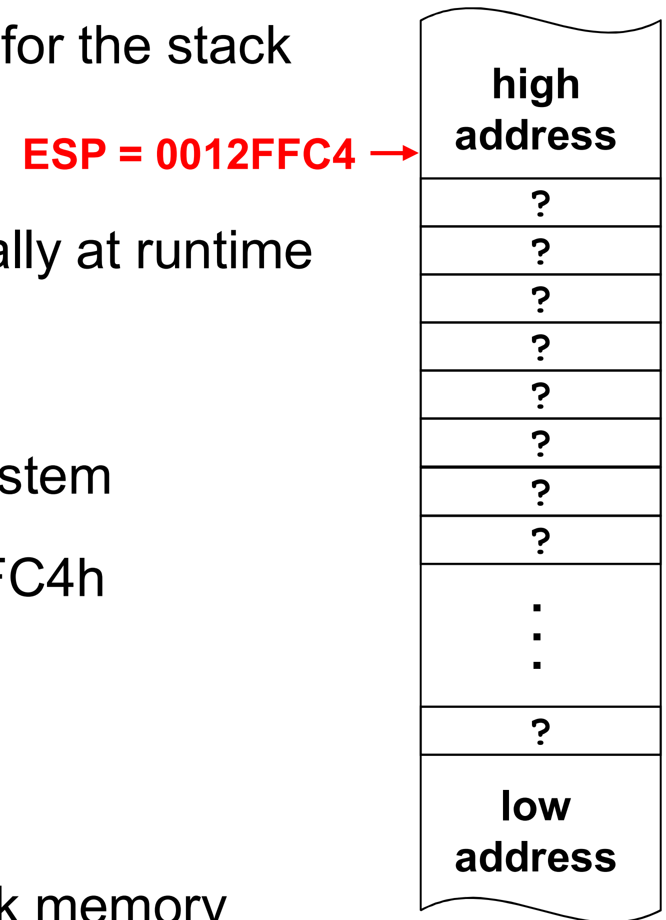
- Stack is a **Last-In-First-Out** (LIFO) data structure
  - Analogous to a stack of plates in a cafeteria
  - Plate on **Top of Stack** is directly accessible
- Two basic stack operations
  - **Push**: inserts a new element on top of the stack
  - **Pop**: deletes top element from the stack
- View the stack as a linear array of elements
  - Insertion and deletion is restricted to one end of array
- Stack has a maximum capacity
  - When stack is **full**, no element can be pushed
  - When stack is **empty**, no element can be popped

Runtime Stack

- **Runtime stack:** array of consecutive memory locations
- Managed by the processor using two registers
  - Stack Segment register **SS**
    - Not modified in protected mode, **SS** points to segment descriptor
  - Stack Pointer register **ESP**
    - For 16-bit real-address mode programs, **SP** register is used
- **ESP** register points to the **top of stack**
  - Always points to last data item placed on the stack
- Only words and doublewords can be pushed and popped
  - But not single bytes
- Stack grows **downward** toward lower memory addresses

Runtime Stack Allocation

- **.STACK** directive specifies a runtime stack
  - Operating system allocates memory for the stack
  - Runtime stack is initially empty
  - The stack size can change dynamically at runtime
- Stack pointer **ESP**
  - **ESP** is initialized by the operating system
  - Typical initial value of **ESP** = 0012FFC4h
- The stack grows **downwards**
  - The memory below **ESP** is free
  - **ESP** is decremented to allocate stack memory



Stack Instructions

- Two basic stack instructions:
  - `push source`
  - `pop  destination`
- **`Source`** can be a word (16 bits) or doubleword (32 bits)
  - General-purpose register
  - Segment register: CS, DS, SS, ES, FS, GS
  - Memory operand, memory-to-stack transfer is allowed
  - Immediate value
- **`Destination`** can be also a word or doubleword
  - General-purpose register
  - Segment register, except that **`pop CS`** is NOT allowed
  - Memory, stack-to-memory transfer is allowed

Push Instruction

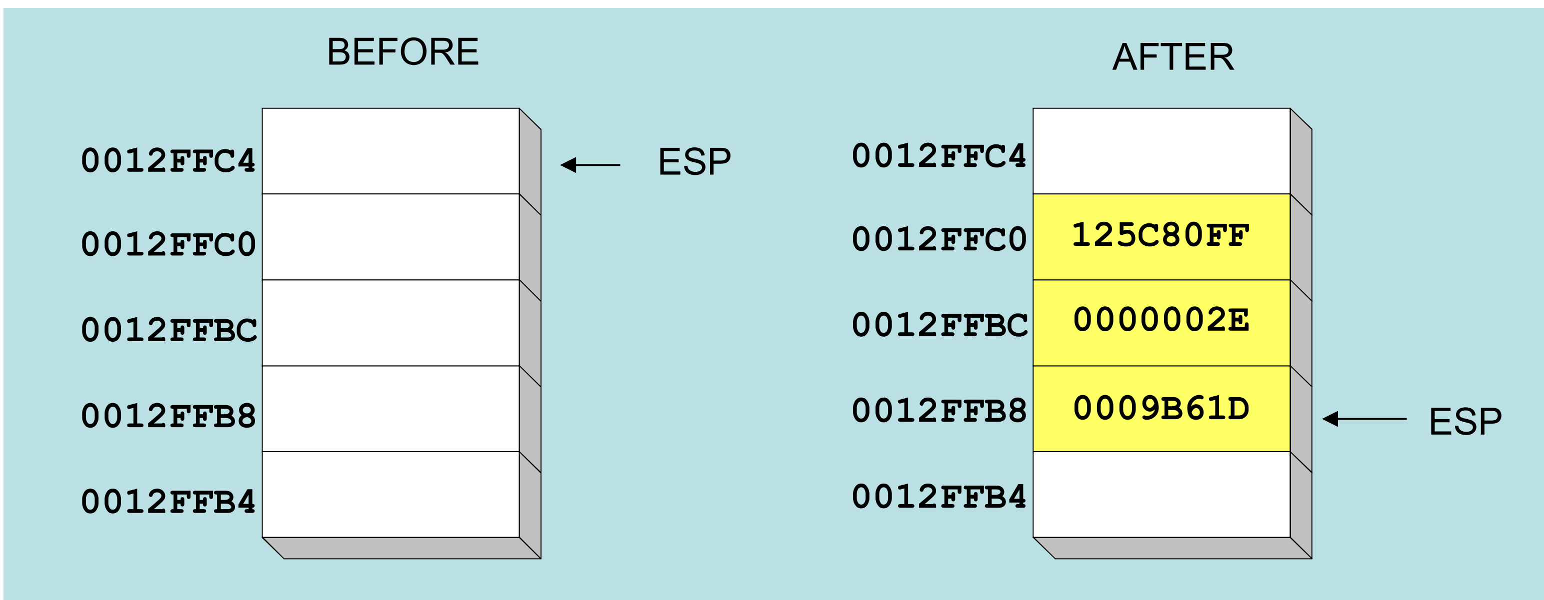
- **Push** `source32 (r/m32 or imm32)`
  - **ESP** is first decremented by **4**
    - **ESP = ESP – 4** (stack grows by 4 bytes)
  - 32-bit source is then copied onto the stack at the new **ESP**
    - **[ESP] = source32**
- **Push** `source16 (r/m16)`
  - **ESP** is first decremented by **2**
    - **ESP = ESP – 2** (stack grows by 2 bytes)
  - 16-bit source is then copied on top of stack at the new **ESP**
    - **[ESP] = source16**
- Operating system puts a limit on the stack capacity
  - **Push** can cause a **Stack Overflow** (stack **cannot grow**)

Examples on the Push Instruction

- Suppose we execute:
  - PUSH EAX ; EAX = 125C80FFh
  - PUSH EBX ; EBX = 2Eh
  - PUSH ECX ; ECX = 9B61Dh

The stack grows downwards

The area below ESP is free



Pop Instruction

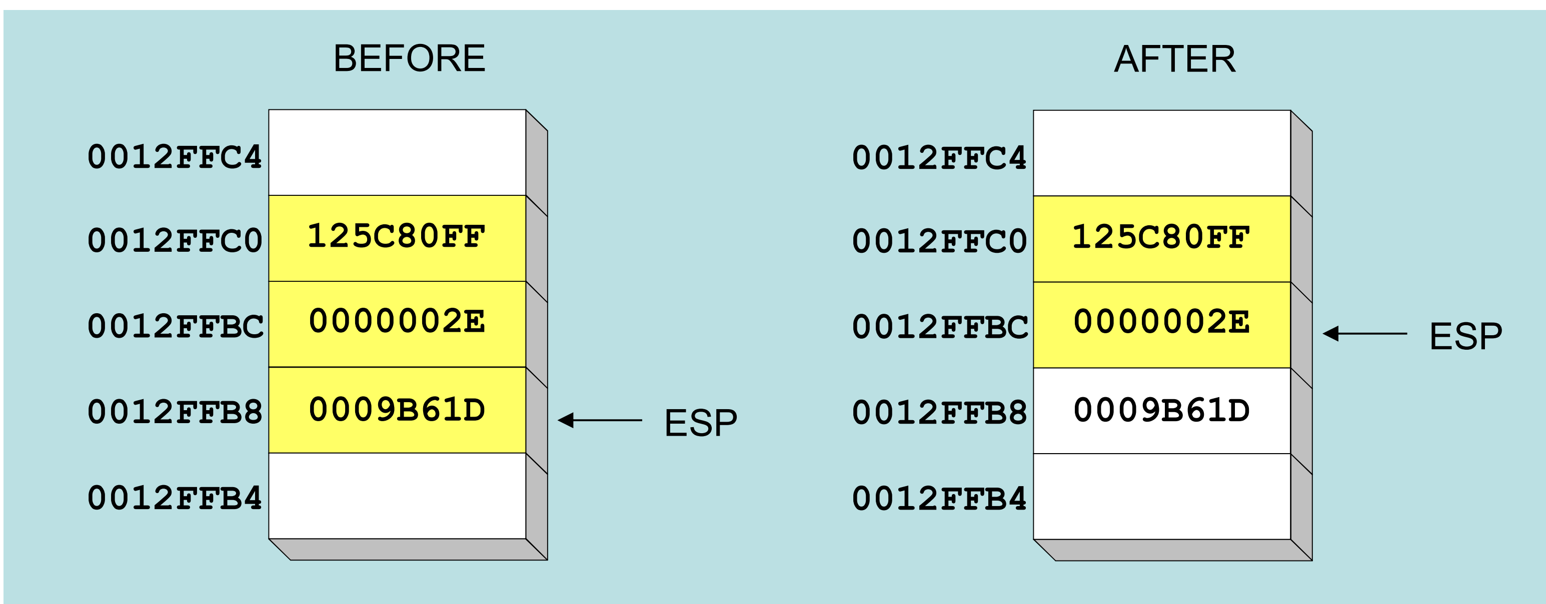
- **Pop** **`dest32 (r/m32)`**
  - 32-bit doubleword at ESP is first copied into dest32
    - **dest32 = [ESP]**
  - ESP is then incremented by 4
    - **ESP = ESP + 4** (stack shrinks by 4 bytes)
- **Pop** **`dest16 (r/m16)`**
  - 16-bit word at ESP is first copied into dest16
    - **dest16 = [ESP]**
  - ESP is then incremented by 2
    - **ESP = ESP + 2** (stack shrinks by 2 bytes)
- Popping from an empty stack causes a **stack underflow**

Examples on the Pop Instruction

- Suppose we execute:
  - POP SI ; SI = B61Dh
  - POP DI ; DI = 0009h

The stack shrinks upwards

The area at & above ESP is allocated



Uses of the Runtime Stack

- Runtime Stack can be utilized for
  - Temporary storage of data and registers
  - Transfer of program control in procedures and interrupts
  - Parameter passing during a procedure call
  - Allocating local variables used inside procedures
- Stack can be used as temporary storage of data
  - Example: exchanging two variables in a data segment

```
push var1  ; var1 is pushed
push var2  ; var2 is pushed
pop  var1  ; var1 = var2 on stack
pop  var2  ; var2 = var1 on stack
```

Temporary Storage of Registers

❖ Stack is often used to free a set of registers

```
push EBX        ; save EBX
push ECX        ; save ECX
. . .
; EBX and ECX can now be modified
. . .
pop  ECX        ; restore ECX first, then
pop  EBX        ; restore EBX
```

❖ Example on moving DX:AX into EBX

```
push DX         ; push most significant word first
push AX         ; then push least significant word
pop  EBX        ; EBX = DX:AX
```

Example: Nested Loop

When writing a nested loop, push the outer loop counter ECX before entering the inner loop, and restore ECX after exiting the inner loop and before repeating the outer 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
    . . .               ; inner loop
    loop L2             ; repeat the inner loop

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

Push/Pop All Registers

- **pushad**
  - Pushes all the 32-bit general-purpose registers
  - EAX, ECX, EDX, EBX, ESP, EBP, ESI, and EDI in this order
  - Initial ESP value (before pushad) is pushed
  - ESP = ESP – 32
- **pusha**
  - Same as pushad but pushes all 16-bit registers AX through DI
  - ESP = ESP – 16
- **popad**
  - Pops into registers EDI through EAX in reverse order of pushad
  - ESP is not read from stack. It is computed as: ESP = ESP + 32
- **popa**
  - Same as popad but pops into 16-bit registers. ESP = ESP + 16

Stack Instructions on Flags

- Special Stack instructions for pushing and popping flags
  - **`pushfd`**
    - Push the 32-bit EFLAGS
  - **`popfd`**
    - Pop the 32-bit EFLAGS
- No operands are required
- Useful for saving and restoring the flags
- For 16-bit programs use **`pushf`** and **`popf`**
  - Push and Pop the 16-bit FLAG register