Stack Operations and Procedures

Computer Organization and Assembly Language

Computer Science Department

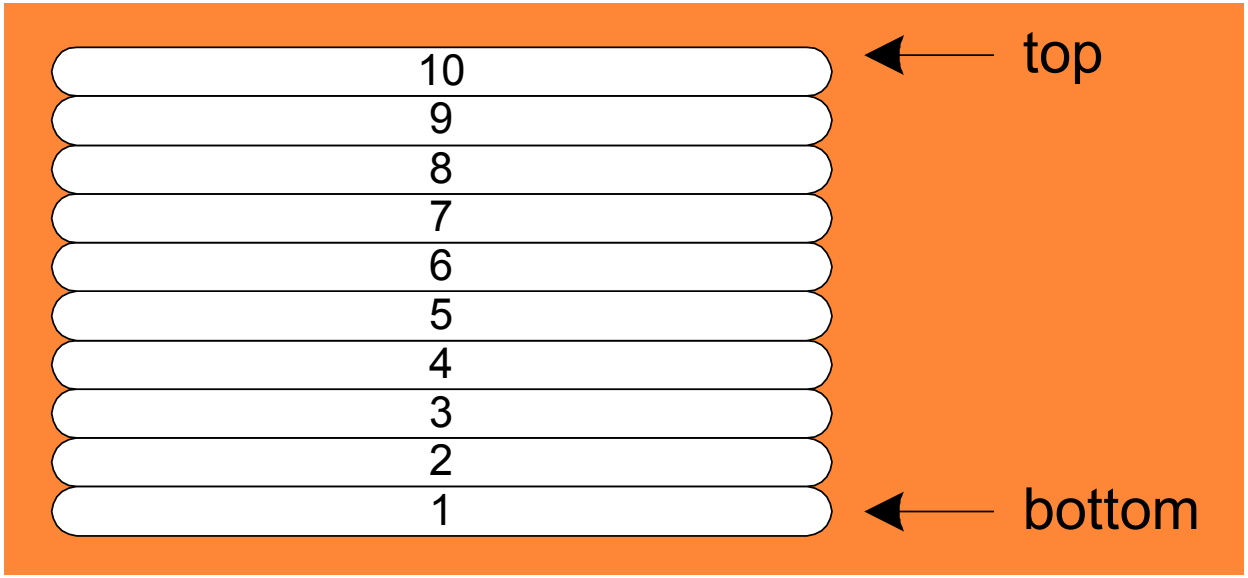
National University of Computer and Emerging Sciences Islamabad

Stack Operations

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

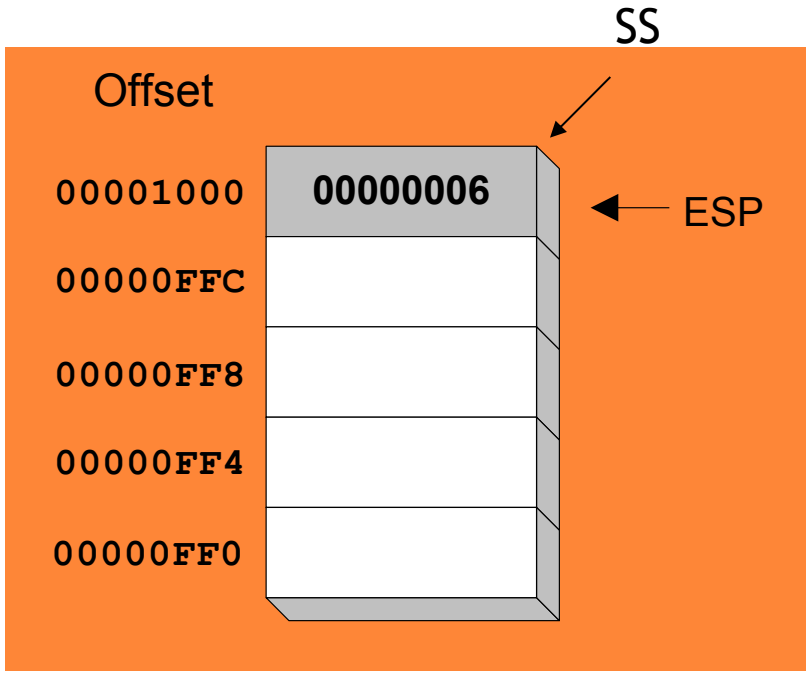
Runtime Stack

- Imagine a stack of plates . . .
  - plates are only added to the top
  - plates are only removed from the top
  - LIFO (Last-In, First-Out) structure
  - Push & pop operations



Runtime Stack

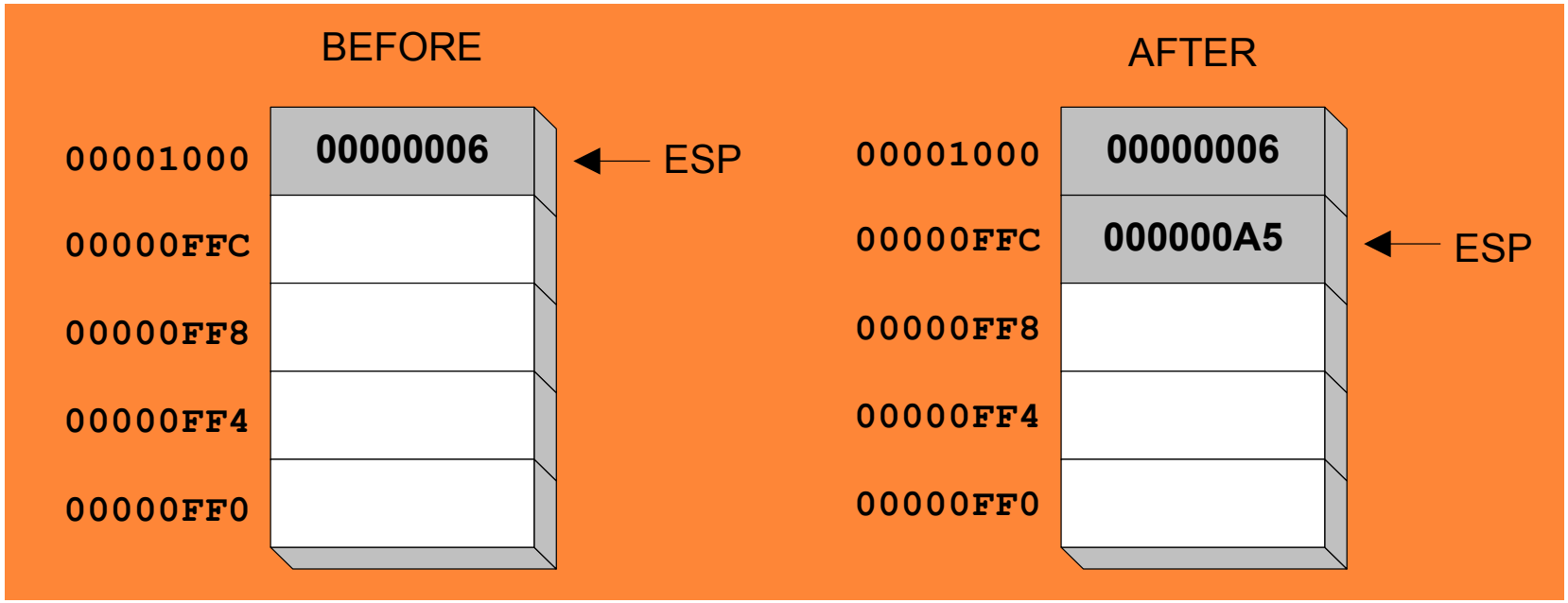
- Managed by the CPU, using two registers
  - SS (stack segment)
  - ESP (stack pointer) *



* SP in Real-address mode

PUSH OPERATION

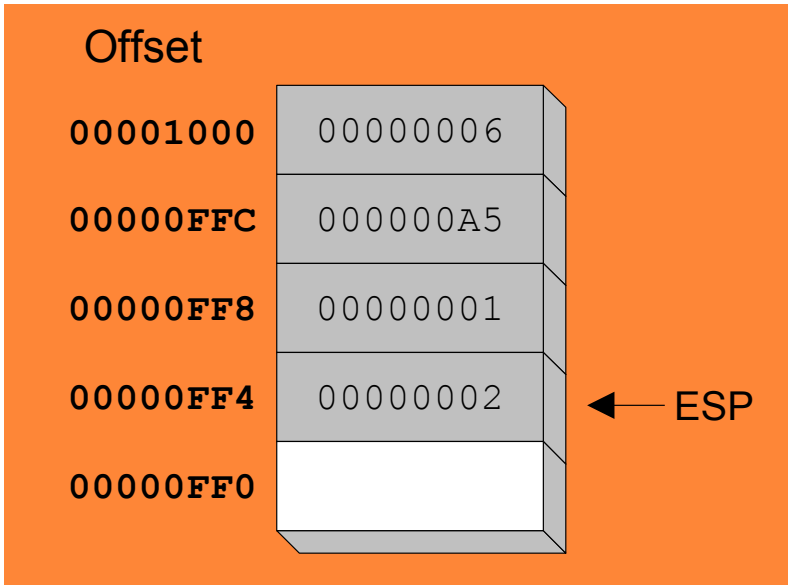
- A 32-bit push operation decrements the stack pointer by 4 and copies a value into the location pointed to by the stack pointer.



PUSH 0A5h

PUSH OPERATION (CONT.)

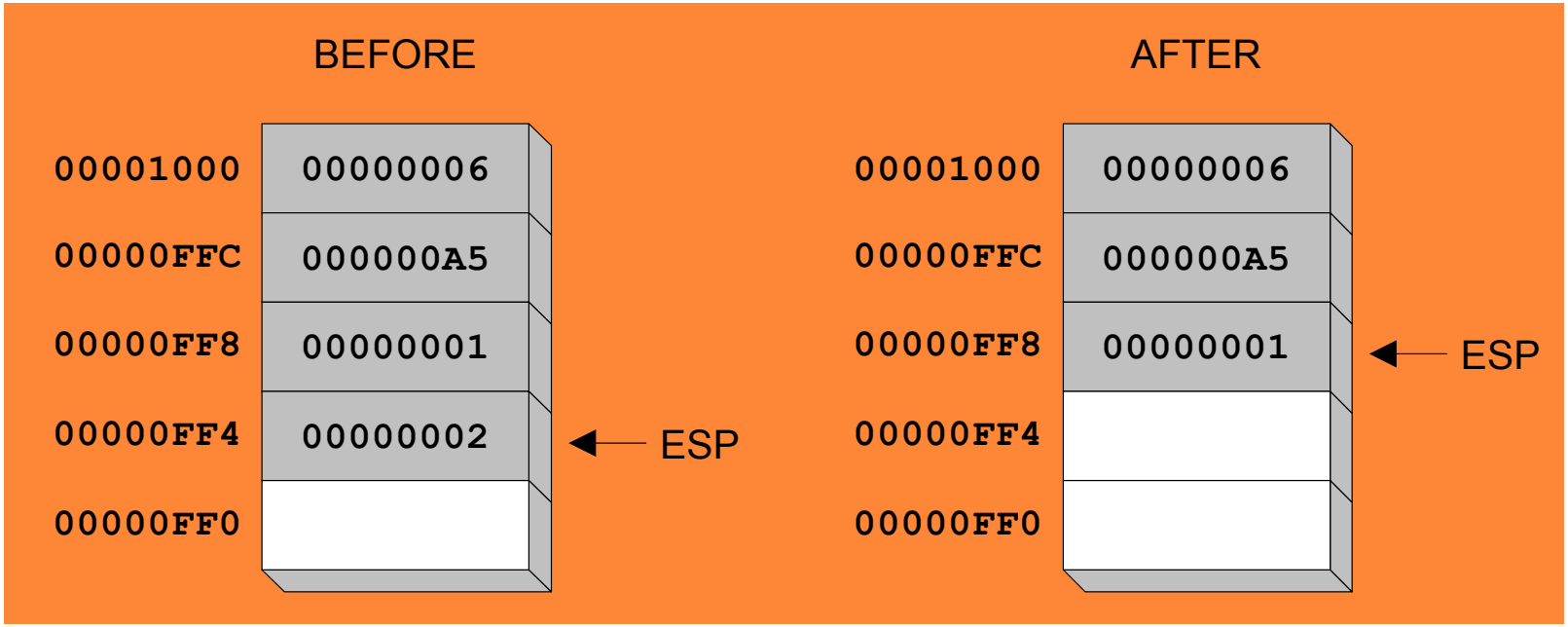
- Same stack after pushing two more integers:



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



Pop EAX
EAX = 00000002

PUSH and POP Instructions

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

When to Use Stacks

- To save and restore registers
- To save return address of a procedure
- To pass arguments
- To support local variables

Defining and Using Procedures

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

Creating Procedures

- Procedure
  - A named block of statements that ends in a return statement
- 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
```

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

```
ESP = ESP - 4          ; push return address
SS:ESP = EIP           ; onto the stack
EIP = EIP + relative offset (or displacement)
                       ; update EIP to point to procedure
```

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

```
EIP = SS:ESP           ; pop return address
ESP = ESP + 4          ; from the stack
```

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

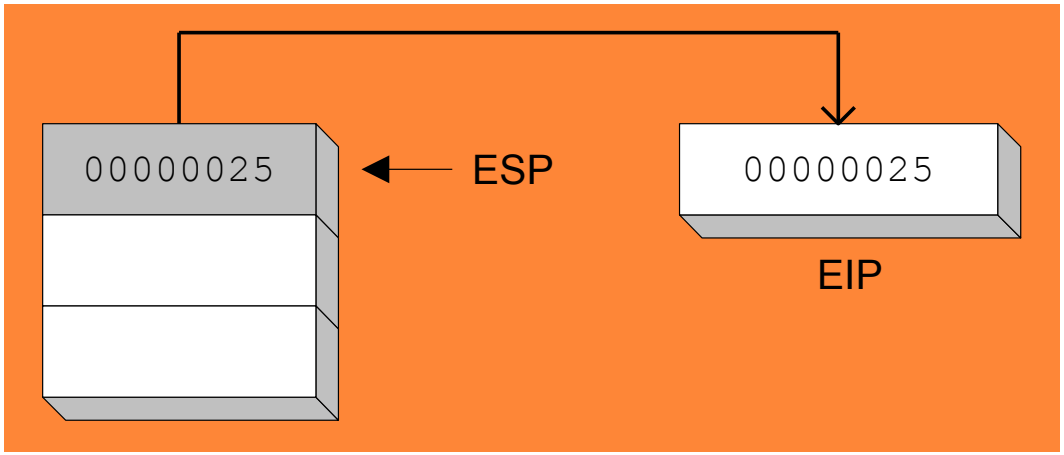
```
main PROC
    00000020 call MySub
    00000025 mov eax,ebx
    .
    .
main ENDP

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

CALL-RET EXAMPLE (CONT.)

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

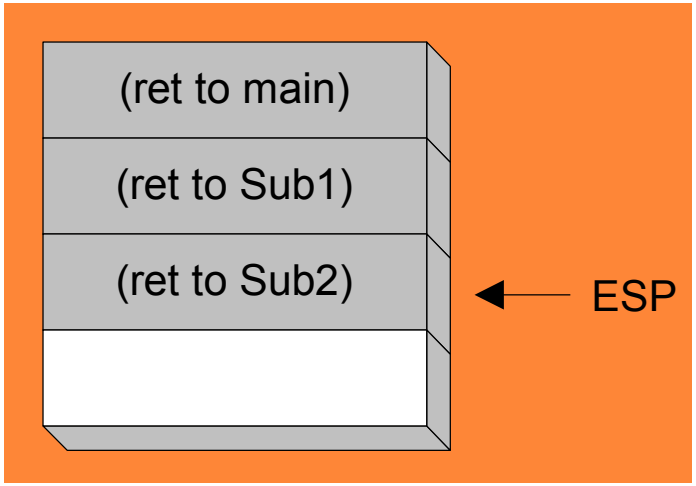
```
main PROC
    .
    .
    call Sub1
    exit
main ENDP

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
```

Parameter Passing Mechanisms

- Call-by-value
  - Receives only values
  - Similar to mathematical functions
- Call-by-reference
  - Receives pointers
  - Directly manipulates parameter storage

Parameter Passing

- Parameter passing is different and complicated than in a high-level language
- In assembly language
  - You should first place all required parameters in a mutually accessible storage area
  - Then call the procedure
- Types of storage area used
  - Registers (general-purpose registers are used)
  - Memory (stack is used)
- Two common methods of parameter passing:
  - Register method
  - Stack method

Parameter Passing: Register Method

The ArraySum procedure calculates the sum of an array. It makes two references to specific variable names:

Call-by-reference

```
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
```

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

Procedure Parameters (cont.)

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
```

CALLING ARRAYSUM

```
.data
array DWORD 10000h, 20000h, 30000h,
  40000h
theSum DWORD ?
.code
main PROC
   mov      esi, OFFSET array
   mov      ecx, LENGTHOF array
   call     ArraySum
  mov       theSum, eax
```

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 **blue**:

```
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
```

Call-by-value

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

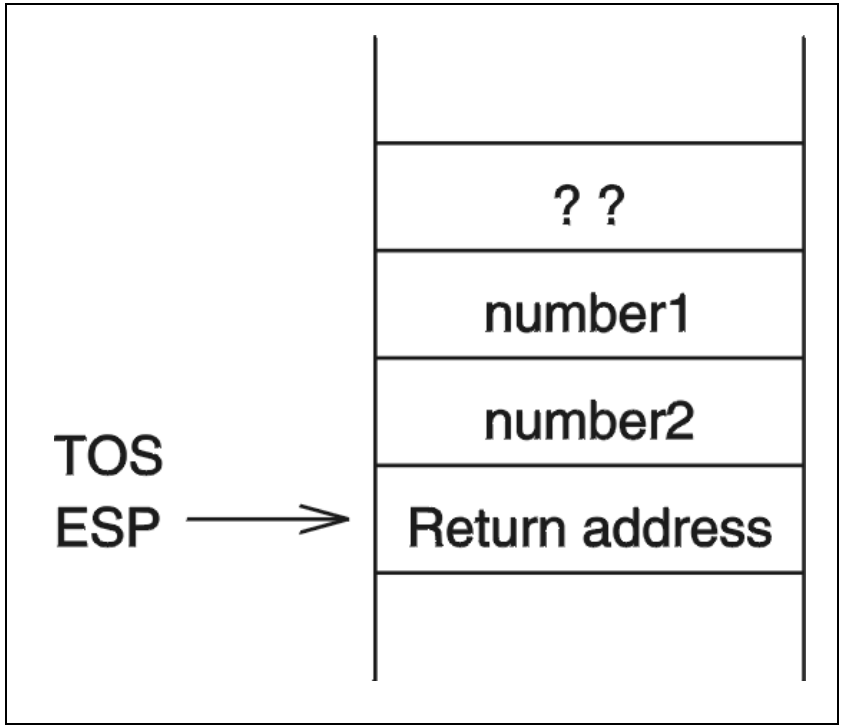
Pros and Cons of the Register Method

- Advantages
  - Convenient and easier
  - Faster
- Disadvantages
  - Only a few parameters can be passed using the register method
    - Only a small number of registers are available
  - Often these registers are not free
    - freeing them by pushing their values onto the stack negates the second advantage

Parameter Passing: Stack Method

- All parameter values are pushed onto the stack before calling the procedure
- Example:

```
push    number1
push    number2
call    sum
```



Accessing Parameters on the Stack

- Parameter values are buried inside the stack
- We can use the following to read **number2**

```
mov    EBX,[ESP+4]
```

Problem: The ESP value changes with **push** and **pop** operations

  - Relative offset depends on the stack operations performed

- Is there a better alternative?
  - Use EBP to access parameters on the stack

Using BP Register to Access Parameters

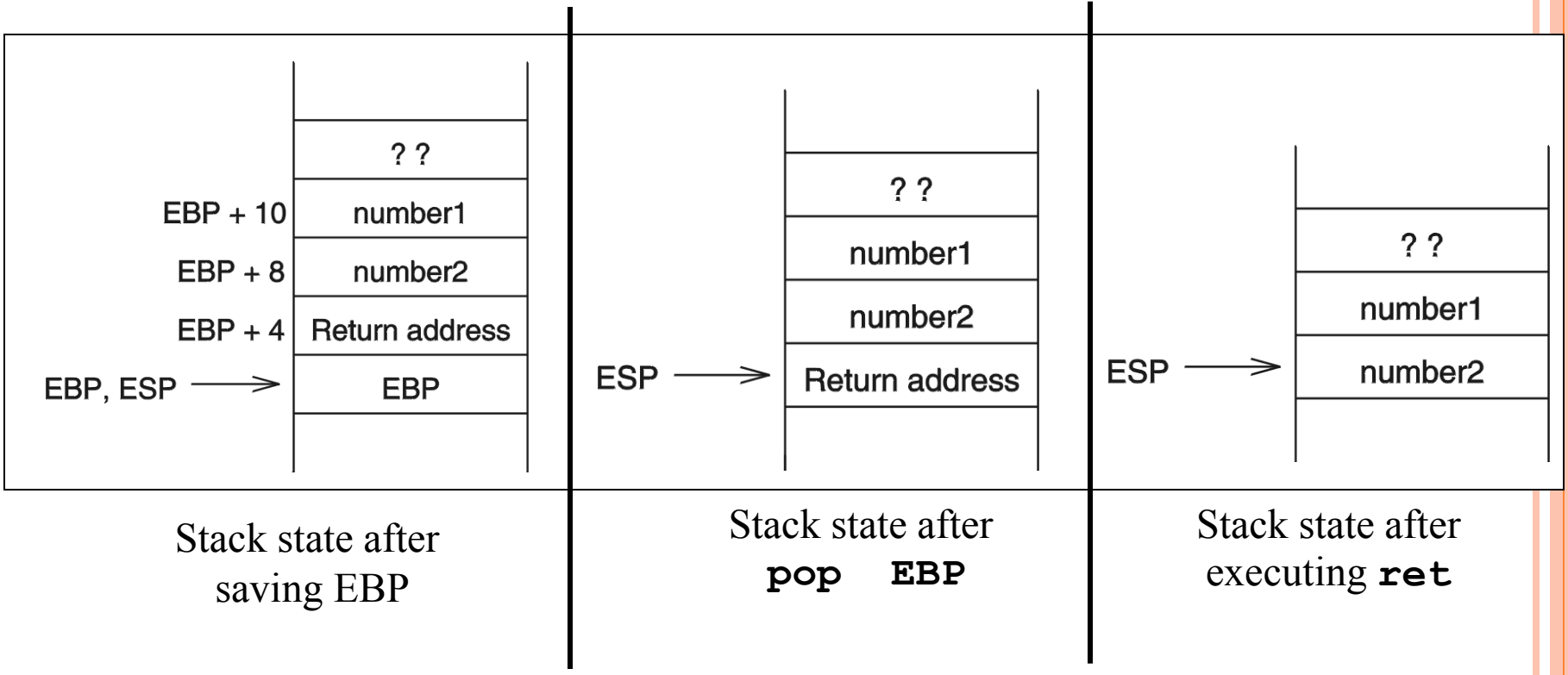
- Preferred method of accessing parameters on the stack is

```
mov   EBP,ESP
mov   EAX,[EBP+4]
```

  to access **number2** in the previous example
- Problem: BP contents are lost!
  - We have to preserve the contents of BP
  - Use the stack (caution: offset value changes)

```
push  EBP
mov   EBP,ESP
```

Clearing the Stack Parameters



Stack state after saving EBP

Stack state after **`pop EBP`**

Stack state after executing **`ret`**

Clearing the Stack Parameters (cont.)

- Two ways of clearing the unwanted parameters on the stack:
  - Use the optional-integer in the **ret** instruction
    - in the previous example, you can use

```
ret    4
```

EIP = SS:ESP
ESP = ESP + 4 + optional-integer

  - Add the constant to ESP in calling procedure (C uses this method)

```
push    number1
push    number2
call    sum
add     ESP,4
```

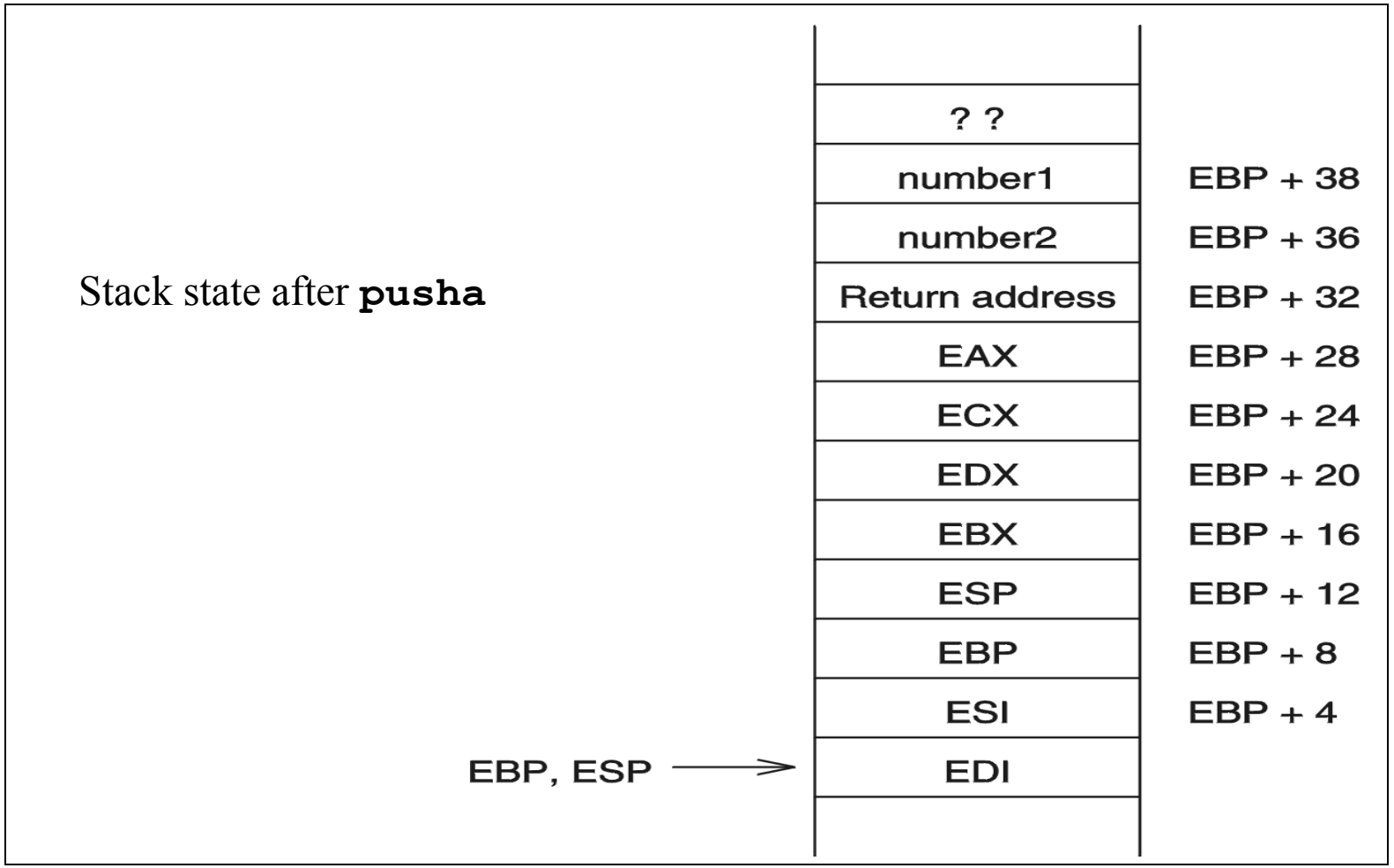
Housekeeping Issues

- Who should clean up the stack of unwanted parameters?
  - Calling procedure
    - Need to update ESP with every procedure call
    - Not really needed if procedures use fixed number of parameters
    - C uses this method because C allows variable number of parameters
  - Called procedure
    - Code becomes modular (parameter clearing is done in only one place)
    - Cannot be used with variable number of parameters

Housekeeping Issues (cont.)

- Need to preserve the state (contents of the registers) of the calling procedure across a procedure call.
  - Stack is used for this purpose
- Which registers should be saved?
  - Save those registers that are used by the calling procedure but are modified by the called procedure
    - Might cause problems as the set of registers used by the calling and called procedures changes over time
  - Save all registers (brute force method) by using **pusha**
    - Increased overhead (**pusha** takes 5 clocks as opposed 1 to save a register)

Housekeeping Issues (cont.)



Housekeeping Issues (cont.)

- Who should preserve the state of the calling procedure?
  - Calling procedure
    - Need to know the registers used by the called procedure
    - Need to include instructions to save and restore registers with every procedure call
    - Causes program maintenance problems
  - Called procedure
    - Preferred method as the code becomes modular (state preservation is done only once and in one place)
    - Avoids the program maintenance problems mentioned

Reference

- Assembly Language for x86 Processors
  - Sec 5.1,5.2
  - Chapter#5
  - Sec 8.1,8.2,8.3 (16/32 Bits only)
  - Chapter#8
    - Kip R. Irvine
    - 7th edition