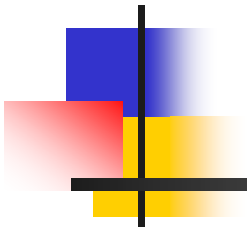
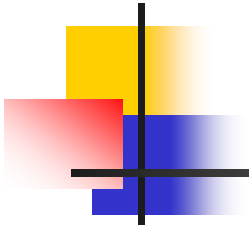


Computer Organization & Assembly Languages

Procedure

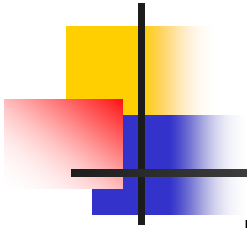
Pu-Jen Cheng

Adapted from the slides prepared by Kip Irvine for the book, Assembly Language for Intel-Based Computers, 5th Ed.



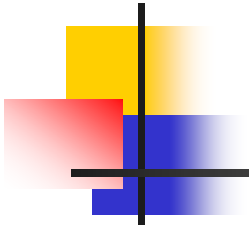
Chapter Overview

- **Linking to an External Library**
- The Book's Link Library
- Stack Operations
- Defining and Using Procedures



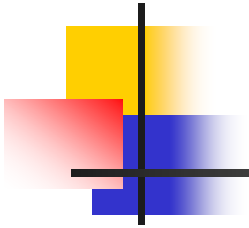
The Book's Link Library

- Link Library Overview
- Calling a Library Procedure
- Linking to a Library
- Library Procedures – Overview
- Six Examples



Link Library Overview

- A file containing procedures that have been compiled into machine code
  - constructed from one or more OBJ files
- To build a library, . . .
  - start with one or more ASM source files
  - assemble each into an OBJ file
  - create an empty library file (extension .LIB)
  - add the OBJ file(s) to the library file, using the Microsoft LIB utility



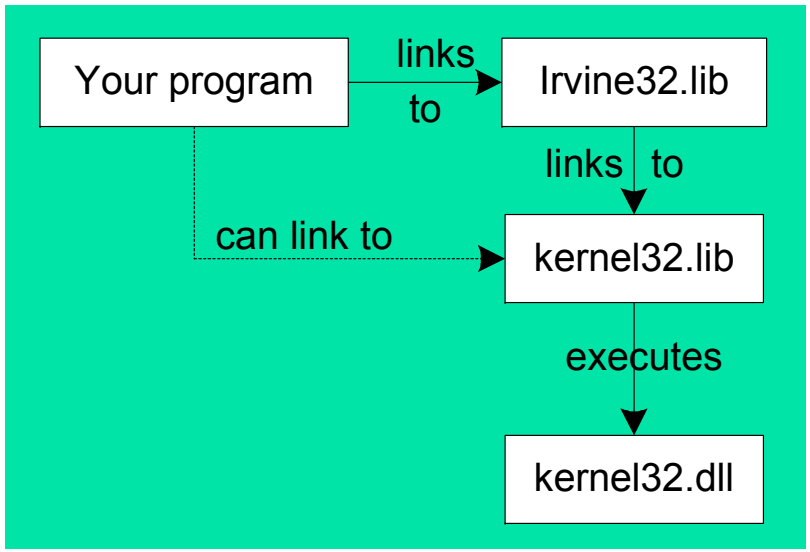
Calling a Library Procedure

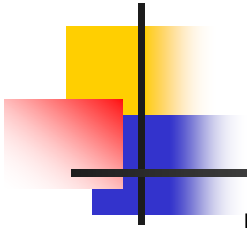
- Call a library procedure using the CALL instruction. Some procedures require input arguments. The INCLUDE directive copies in the procedure prototypes (declarations).
- The following example displays "1234" on the console:

```
INCLUDE Irvine32.inc
.code
   mov eax,1234h              ; input argument
   call WriteHex              ; show hex number
   call Crlf                  ; end of line
```

Linking to a Library

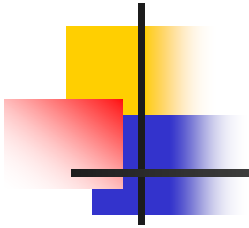
- Your programs link to Irvine32.lib using the linker command inside a batch file named make32.bat.
- Notice the two LIB files: Irvine32.lib, and kernel32.lib
  - the latter is part of the Microsoft *Win32 Software Development Kit (SDK)*





What's Next

- Linking to an External Library
- **The Book's Link Library**
- Stack Operations
- Defining and Using Procedures



Library Procedures - Overview

CloseFile – Closes an open disk file

Clrscr - Clears console, locates cursor at upper left corner

CreateOutputFile - Creates new disk file for writing in output mode

Crlf - Writes end of line sequence to standard output

Delay - Pauses program execution for *n* millisecond interval

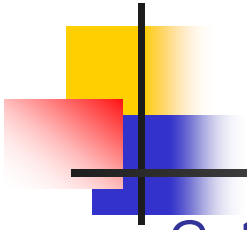
DumpMem - Writes block of memory to standard output in hex

DumpRegs – Displays general-purpose registers and flags (hex)

GetCommandtail - Copies command-line args into array of bytes

GetMaxXY - Gets number of cols, rows in console window buffer

GetMseconds - Returns milliseconds elapsed since midnight



Library Procedures - Overview (cont.)

GetTextColor - Returns active foreground and background text colors in the console window

Gotoxy - Locates cursor at row and column on the console

IsDigit - Sets Zero flag if AL contains ASCII code for decimal digit (0–9)

MsgBox, MsgBoxAsk – Display popup message boxes

OpenInputFile – Opens existing file for input

ParseDecimal32 – Converts unsigned integer string to binary

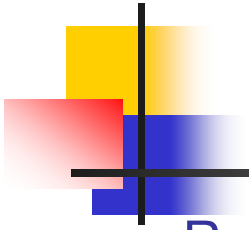
ParseInteger32 - Converts signed integer string to binary

Random32 - Generates 32-bit pseudorandom integer in the range 0 to FFFFFFFFh

Randomize - Seeds the random number generator

RandomRange - Generates a pseudorandom integer within a specified range

ReadChar - Reads a single character from standard input



Library Procedures - Overview (cont.)

ReadFromFile – Reads input disk file into buffer

ReadDec - Reads 32-bit unsigned decimal integer from keyboard

ReadHex - Reads 32-bit hexadecimal integer from keyboard

ReadInt - Reads 32-bit signed decimal integer from keyboard

ReadKey – Reads character from keyboard input buffer

ReadString - Reads string from standard input, terminated by [Enter]

SetTextColor - Sets foreground and background colors of all subsequent console text output

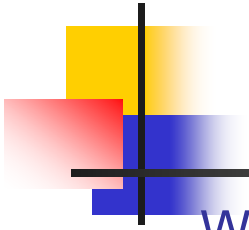
StrLength – Returns length of a string

WaitMsg - Displays message, waits for Enter key to be pressed

WriteBin - Writes unsigned 32-bit integer in ASCII binary format.

WriteBinB – Writes binary integer in byte, word, or doubleword format

WriteChar - Writes a single character to standard output



Library Procedures - Overview (cont.)

WriteDec - Writes unsigned 32-bit integer in decimal format

WriteHex - Writes an unsigned 32-bit integer in hexadecimal format

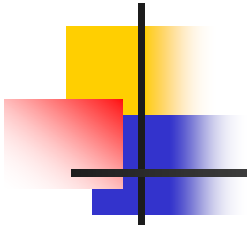
WriteHexB – Writes byte, word, or doubleword in hexadecimal format

WriteInt - Writes signed 32-bit integer in decimal format

WriteString - Writes null-terminated string to console window

WriteToFile - Writes buffer to output file

WriteWindowsMsg - Displays most recent error message generated by MS-Windows



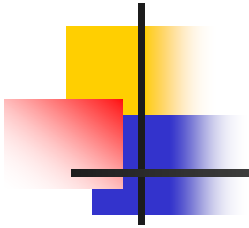
Example 1

Clear the screen, delay the program for 500 milliseconds, and dump the registers and flags.

```
.code
    call Clrscr
    mov  eax,500
    call Delay
    call DumpRegs
```

Sample output:

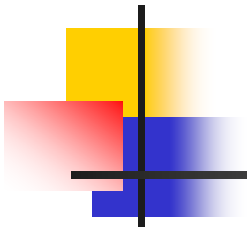
```
EAX=00000613 EBX=00000000 ECX=000000FF EDX=00000000
ESI=00000000 EDI=00000100 EBP=0000091E ESP=000000F6
EIP=00401026 EFL=00000286 CF=0 SF=1 ZF=0 OF=0
```



Example 2

Display a null-terminated string and move the cursor to the beginning of the next screen line.

```
.data
str1 BYTE "Assembly language is easy!",0

.code
   mov  edx,OFFSET str1
   call WriteString
   call Crlf
```



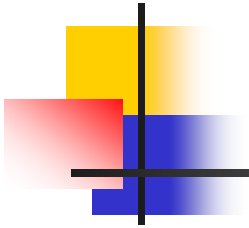
Example 3

Display an unsigned integer in binary, decimal, and hexadecimal, each on a separate line.

```
IntVal = 35
.code
   mov  eax,IntVal
   call WriteBin            ; display binary
   call Crlf
   call WriteDec            ; display decimal
   call Crlf
   call WriteHex            ; display hexadecimal
   call Crlf
```

Sample output:

```
0000 0000 0000 0000 0000 0000 0010 0011
35
23
```

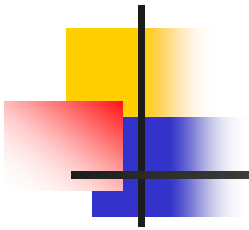


Example 4

Input a string from the user. EDX points to the string and ECX specifies the maximum number of characters the user is permitted to enter.

```
.data
fileName BYTE 80 DUP(0)

.code
   mov edx,OFFSET fileName
   mov ecx,SIZEOF fileName - 1
   call ReadString
```

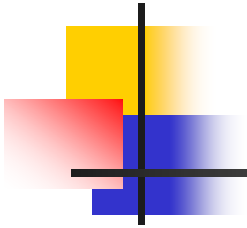
A null byte is automatically appended to the string.



Example 5

Generate and display ten pseudorandom signed integers in the range 0 – 99. Pass each integer to WriteInt in EAX and display it on a separate line.

```
.code
    mov ecx,10                  ; loop counter

L1: mov  eax,100                ; ceiling value
    call RandomRange            ; generate random int
    call WriteInt               ; display signed int
    call Crlf                   ; goto next display line
    loop L1                     ; repeat loop
```

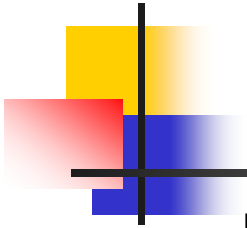


Example 6

Display a null-terminated string with yellow characters on a blue background.

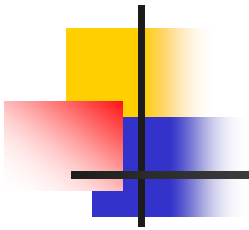
```
.data
str1 BYTE "Color output is easy!",0

.code
   mov  eax,yellow + (blue * 16)
   call SetTextColor
   mov  edx,OFFSET str1
   call WriteString
   call Crlf
```

The background color is multiplied by 16 before being added to the foreground color.



What's Next

- Linking to an External Library
- The Book's Link Library
- **Stack Operations**
- Defining and Using Procedures

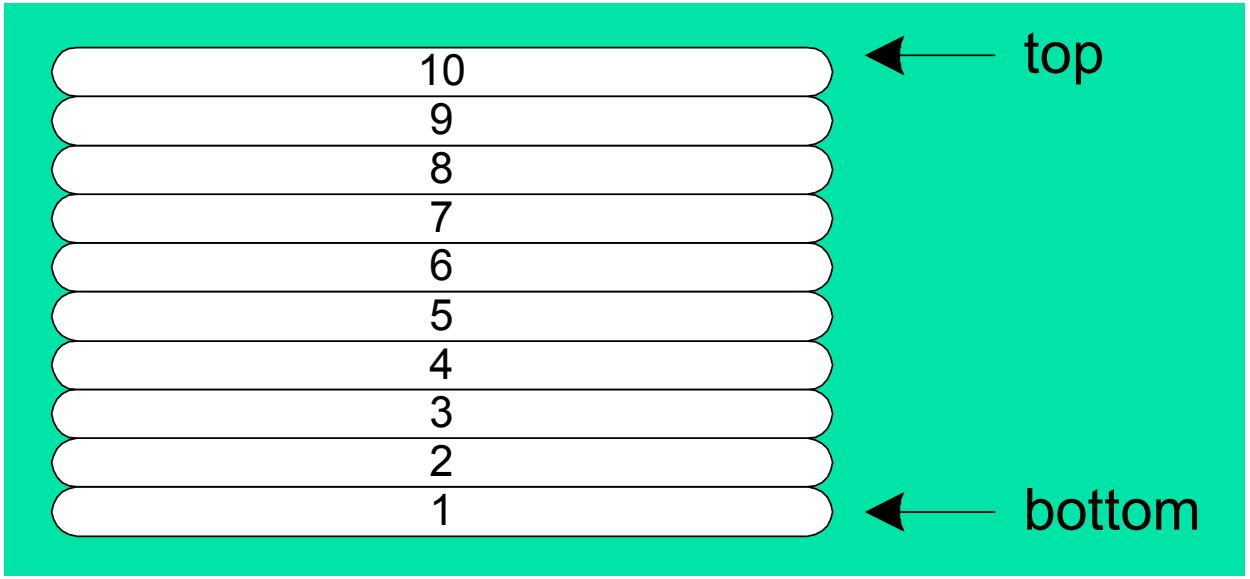


Stack Operations

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

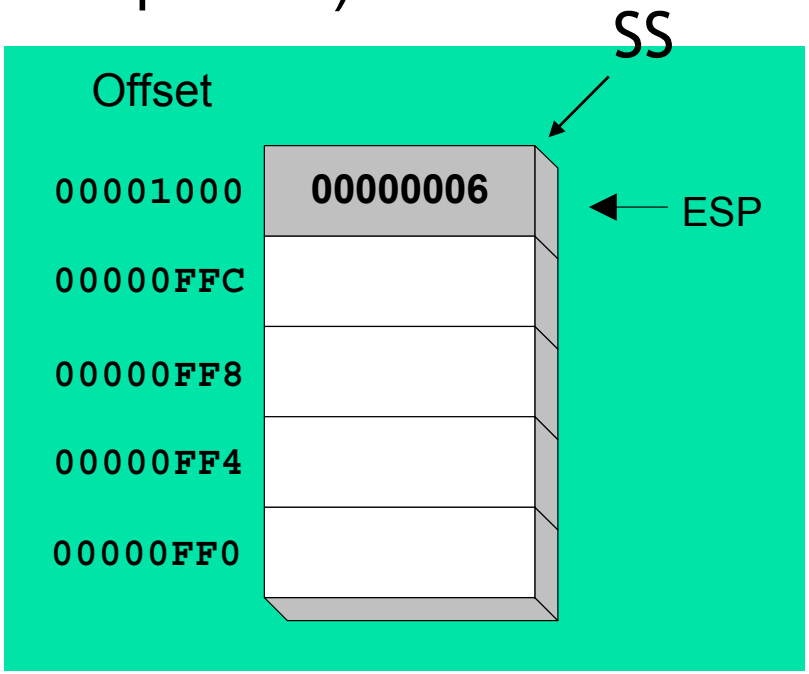
Runtime Stack

- 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

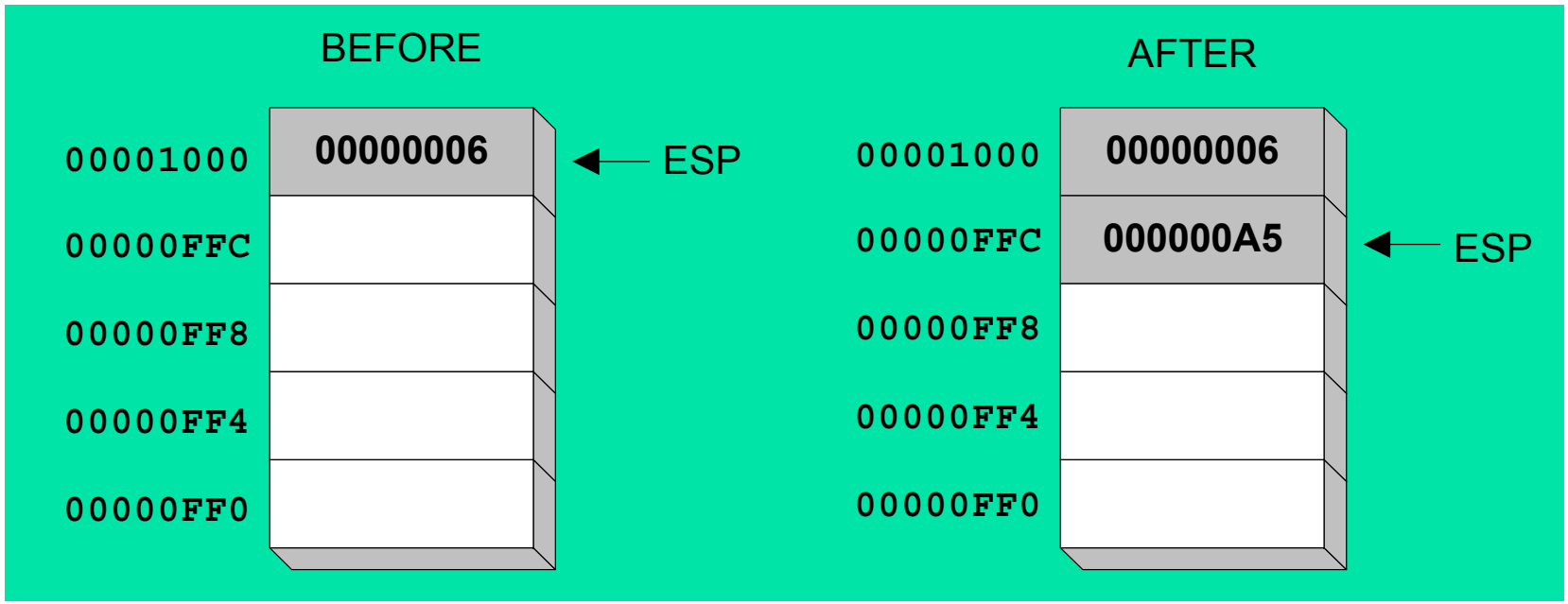
- Managed by the CPU, using two registers
  - SS (stack segment)
  - ESP (stack pointer) *



* SP in Real-address mode

PUSH Operation

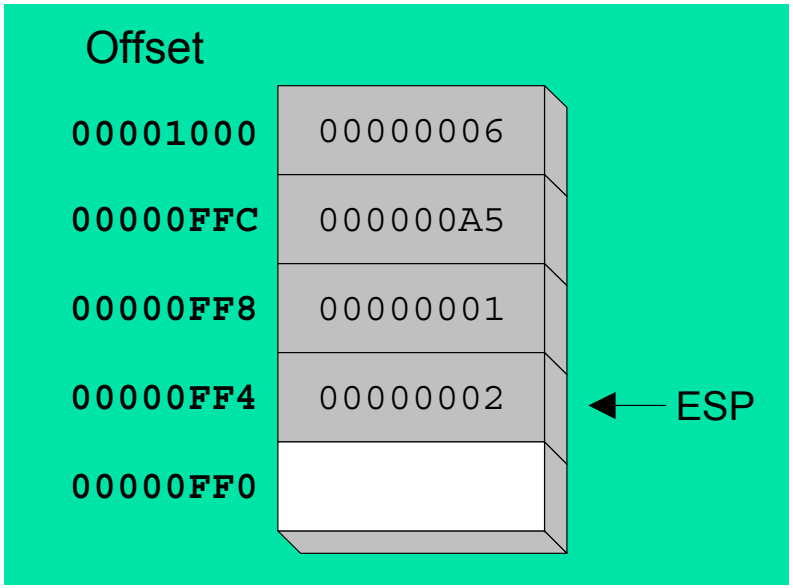
- A 32-bit push operation decrements the stack pointer by 4 and copies a value into the location pointed to by the stack pointer.



PUSH 0A5h

PUSH Operation (cont.)

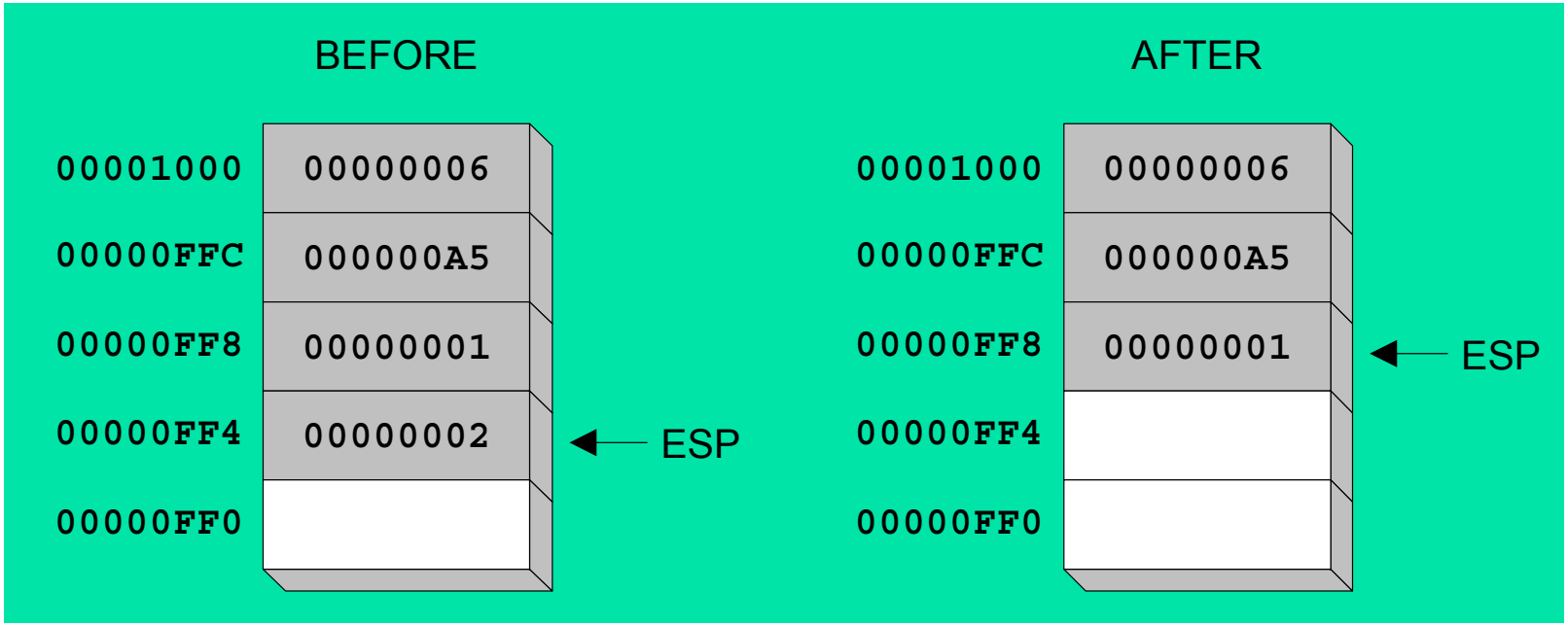
- Same stack after pushing two more integers:



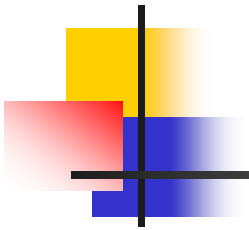
The stack grows downward. The area below ESP is always available (unless the stack has overflowed).

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

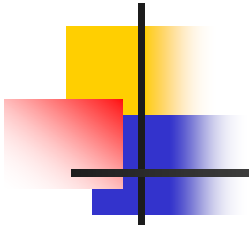


Pop EAX
EAX = 00000002



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 Sacks

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

Example: Using PUSH and POP

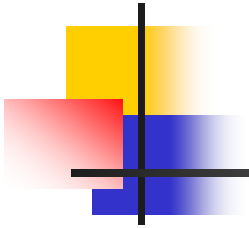
Save and restore registers when they contain important values. PUSH and POP instructions occur in the opposite order.

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

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

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

Example: Nested Loop

When creating a nested loop, 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

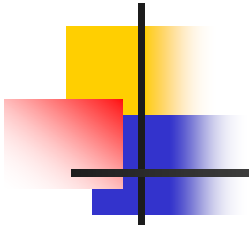
- PUSHFD and POPFD
  - push and pop the EFLAGS register
- PUSHAD pushes the 32-bit general-purpose registers on the stack
  - order: EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI
- POPAD pops the same registers off the stack in reverse order
  - PUSHA and POPA do the same for 16-bit registers

Example: Reversing String

```
.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
```

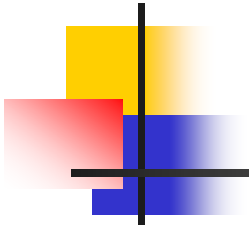
Example: Reversing String (cont.)

```
; 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

  exit
main ENDP
END main
```



What's Next

- Linking to an External Library
- The Book's Link Library
- Stack Operations
- **Defining and Using Procedures**



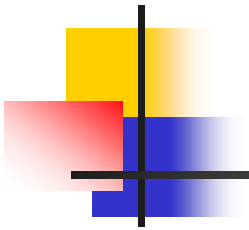
Defining and Using Procedures

- 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

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

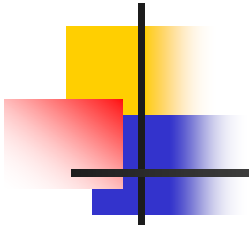
- The CALL instruction calls a procedure
  - pushes offset of next instruction on the stack
  - copies the address of the called procedure into EIP

```
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

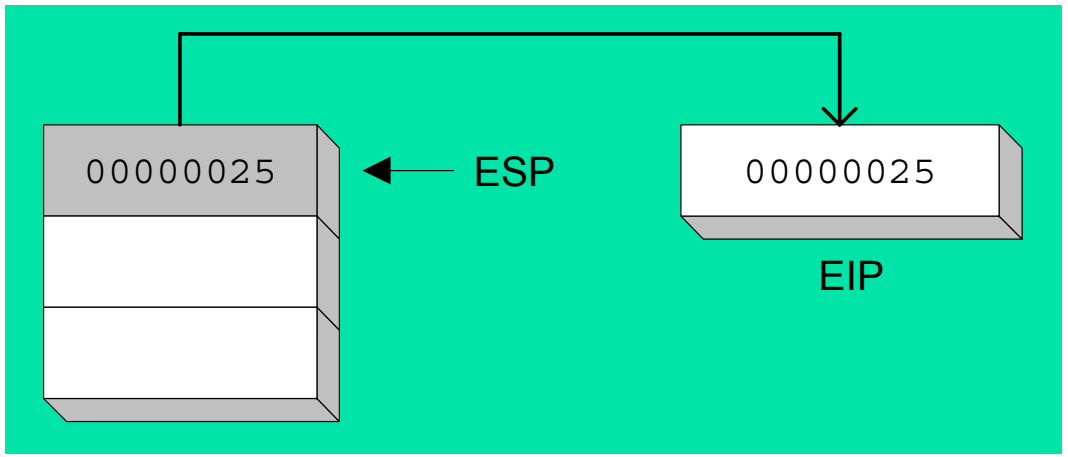
00000040 is the offset of the first instruction inside MySub

```
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

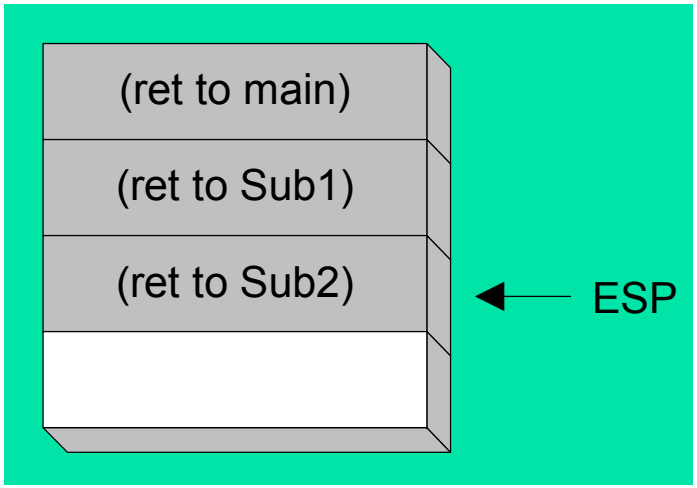


(stack shown before RET executes)

Nested Procedure Calls

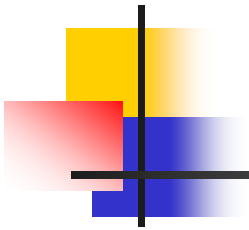
```
main PROC
    .
    .
    call Sub1
    exit
main ENDP

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:



How Is Program Control Transferred?

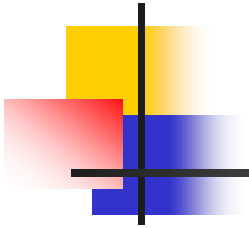
```
Offset(hex)  machine code(hex)
                                main:
                                     . . . .
00000002     E816000000              call     sum
00000007     89C3                    mov      EBX,EAX
                                     . . . .
                                     ; end of main procedure

                                sum:
0000001D     55                      push     EBP
                                     . . . .
                                     ; end of sum procedure

                                avg:
                                     . . . .
00000028     E8F0FFFFFF              call     sum
0000002D     89D8                    mov      EAX,EBX
                                     . . . .
                                     ; end of avg procedure
```



Local and Global Labels

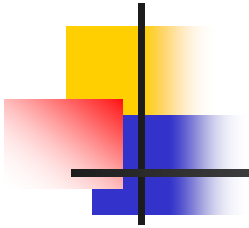
A local label is visible only to statements inside the same procedure. A global label is visible everywhere.

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

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



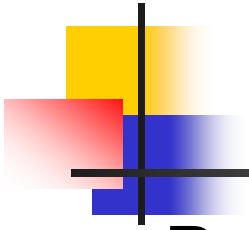
Procedure Parameters

- 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



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

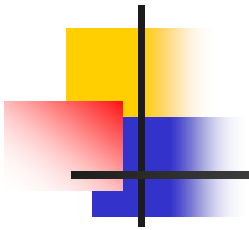
```
ArraySum PROC
    mov esi,0                       ; array index
    mov eax,0                       ; set the sum to zero
    mov ecx,LENGTHOF myarray        ; set number of elements

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.)

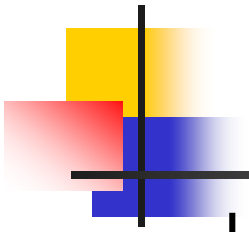
This version of ArraySum returns the sum of any doubleword array whose address is in ESI. The sum is returned in EAX:

```
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:

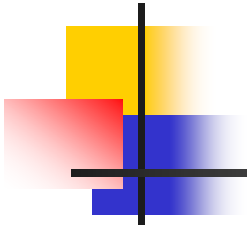
```
ArraySum PROC
    push esi
    push ecx
    .
    .
    pop ecx
    pop esi
    ret
ArraySum ENDP
```

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
```

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



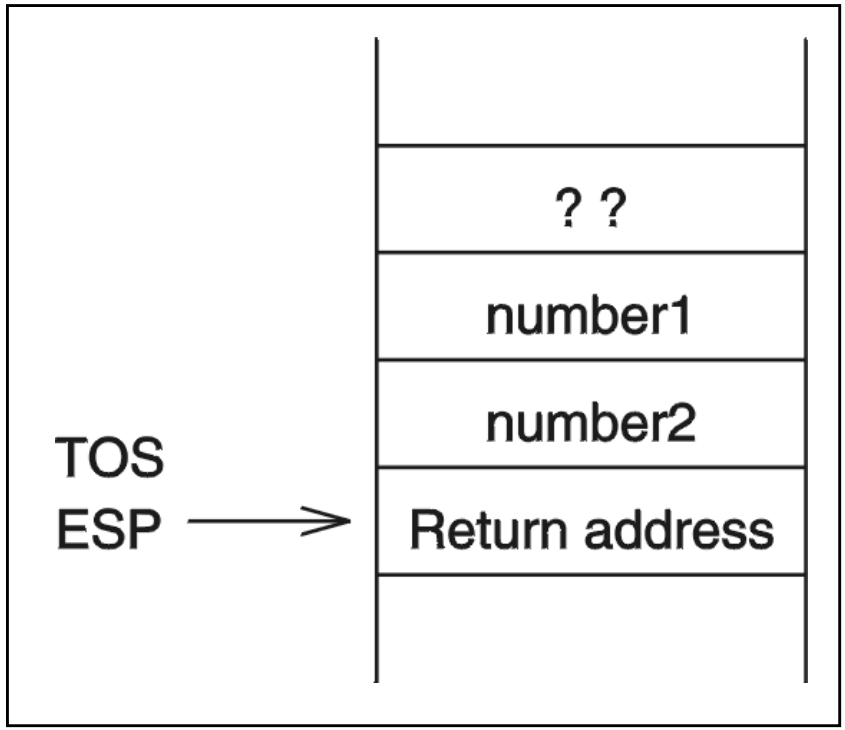
Pros and Cons of the Register Method

- 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

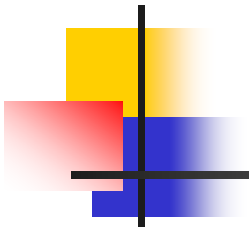
Parameter Passing: Stack Method

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

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



Accessing Parameters on the Stack

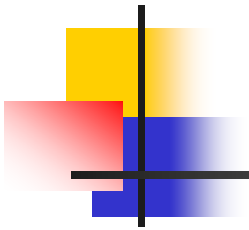


- Parameter values are buried inside the stack
- We can use the following to read **number2**

  mov EBX,[ESP+4]

  Problem: The ESP value changes with **push** and **pop** operations
  - Relative offset depends of the stack operations performed
- Is there a better alternative?
  - Use EBP to access parameters on the stack

Using BP Register to Access Parameters



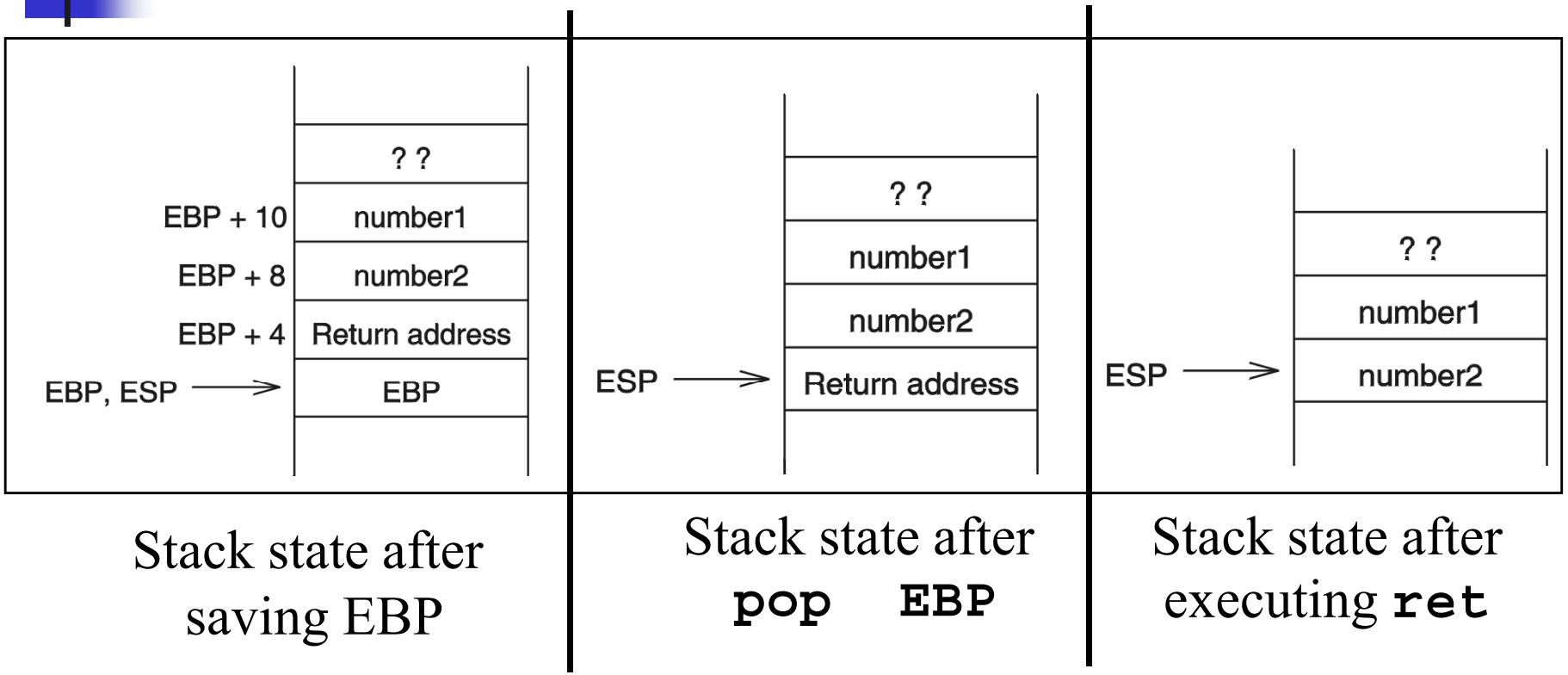
- Preferred method of accessing parameters on the stack is

```
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`**

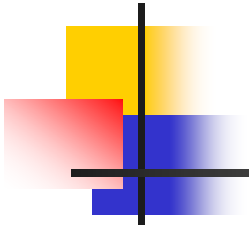
Clearing the Stack Parameters (cont.)

- 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

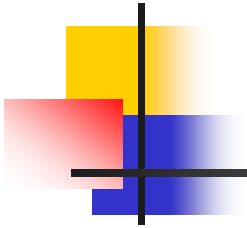
      EIP = SS:ESP
      ESP = ESP + 4 + optional-integer
  - Add the constant to ESP in calling procedure (C uses this method)

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



Housekeeping Issues

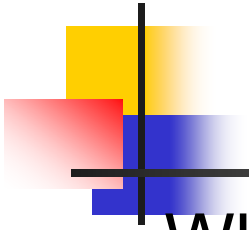
- Who should clean up the stack of unwanted parameters?
  - Calling procedure
    - Need to update ESP with every procedure call
    - Not really needed if procedures use fixed number of parameters
    - C uses this method because C allows variable number of parameters
  - Called procedure
    - Code 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)

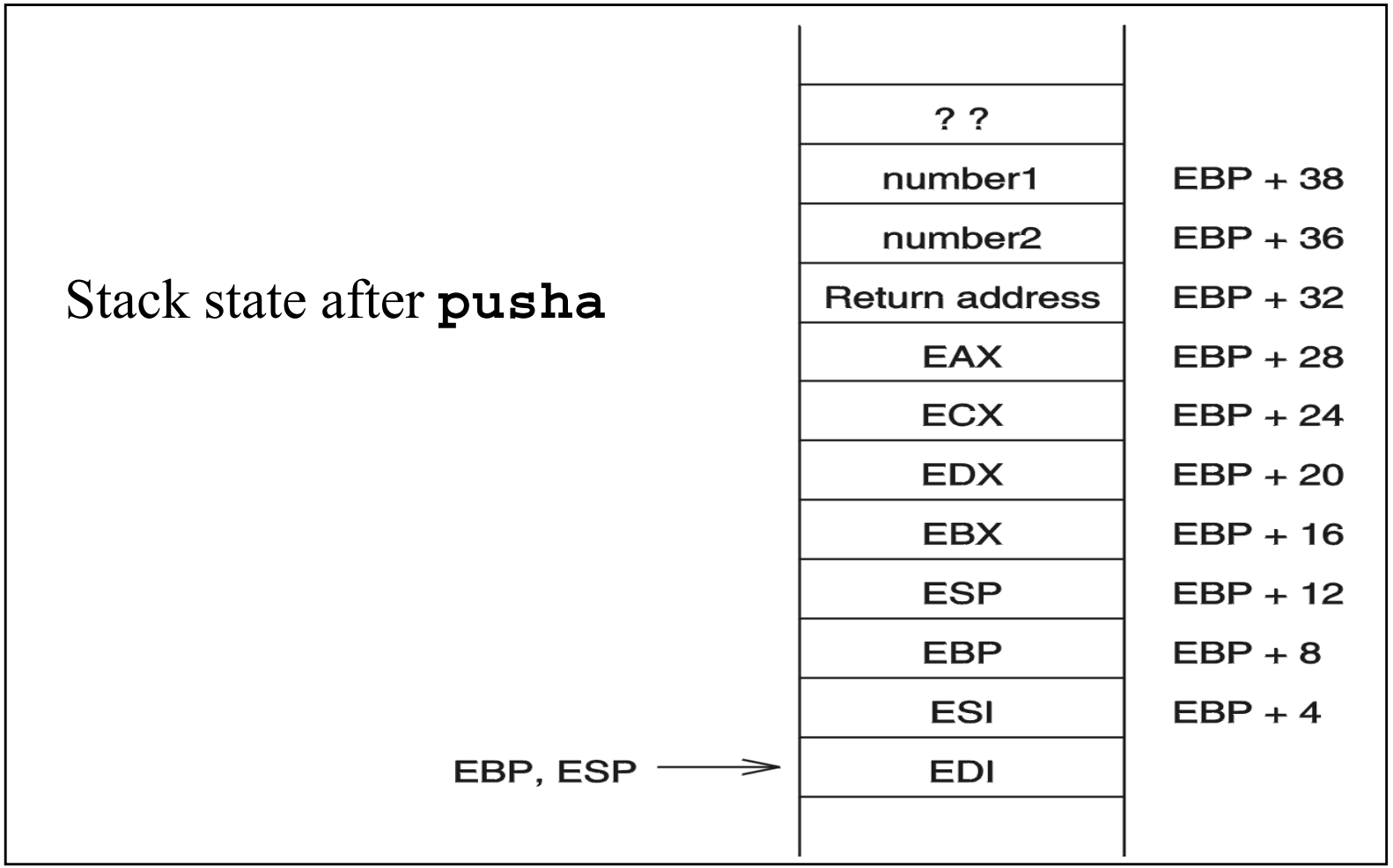
Housekeeping Issues (cont.)



- 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

Housekeeping Issues (cont.)

Stack state after **`pusha`**



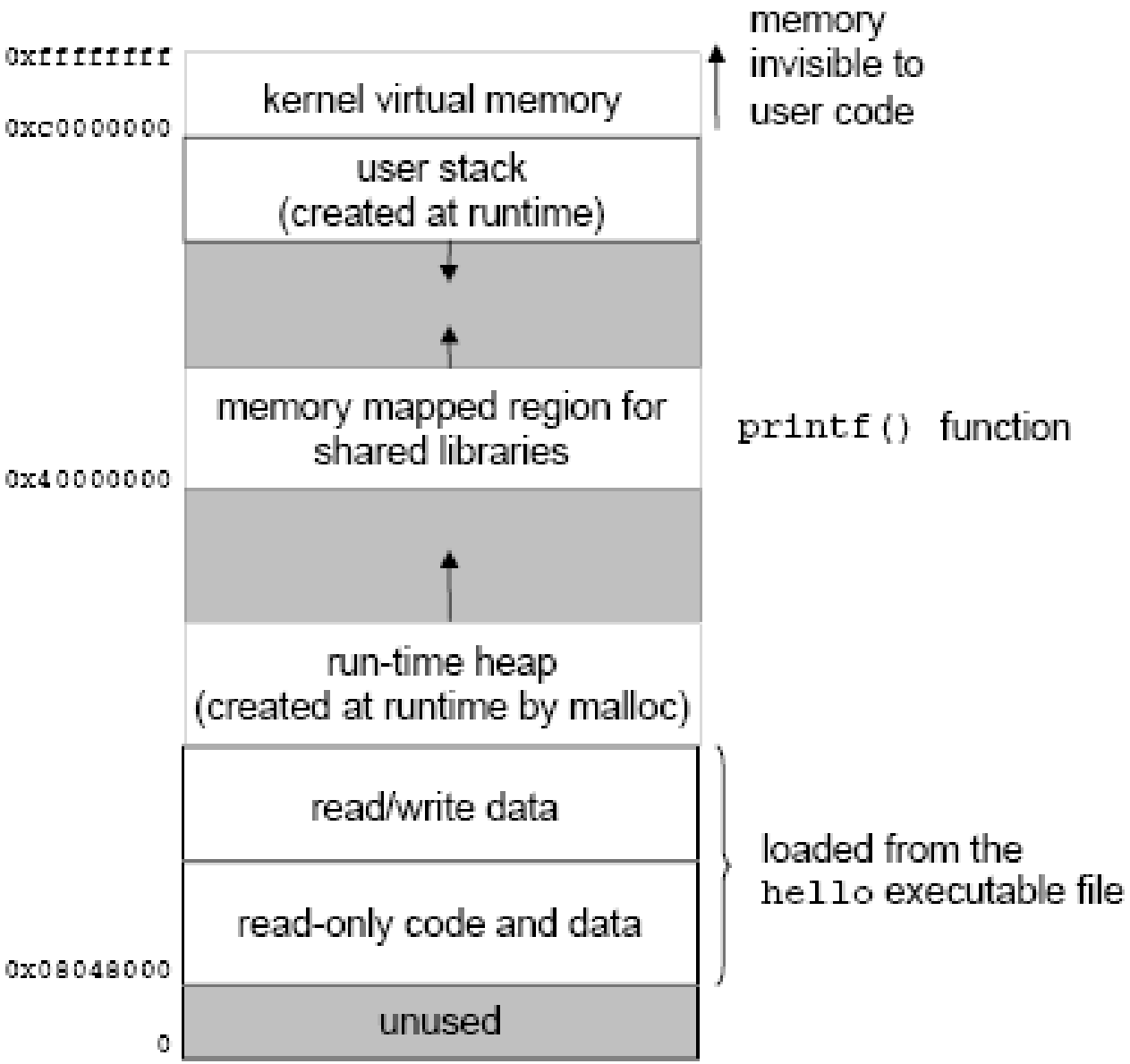


Figure 1.13: **Linux process virtual address space.**