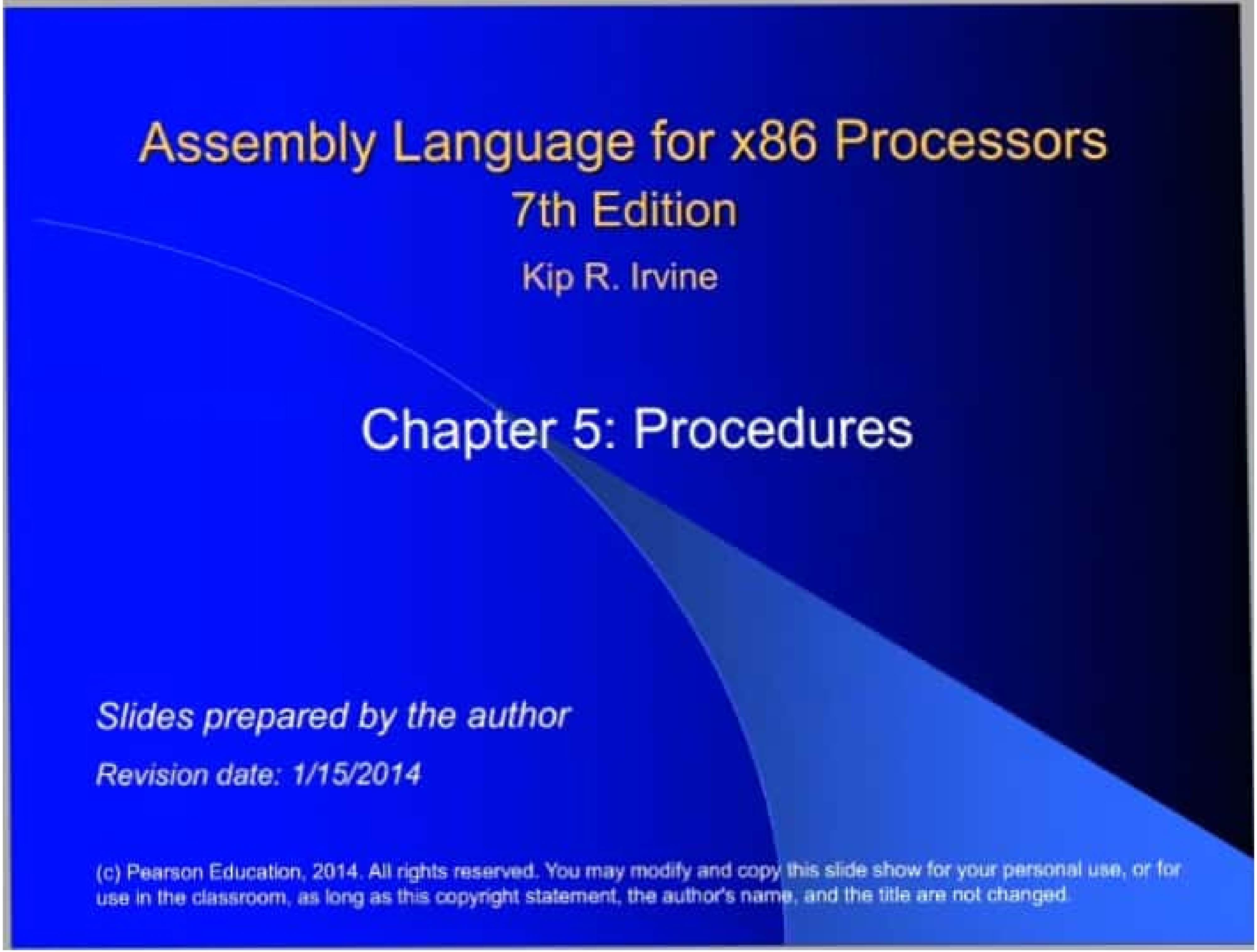


Assembly Language for x86 Processors

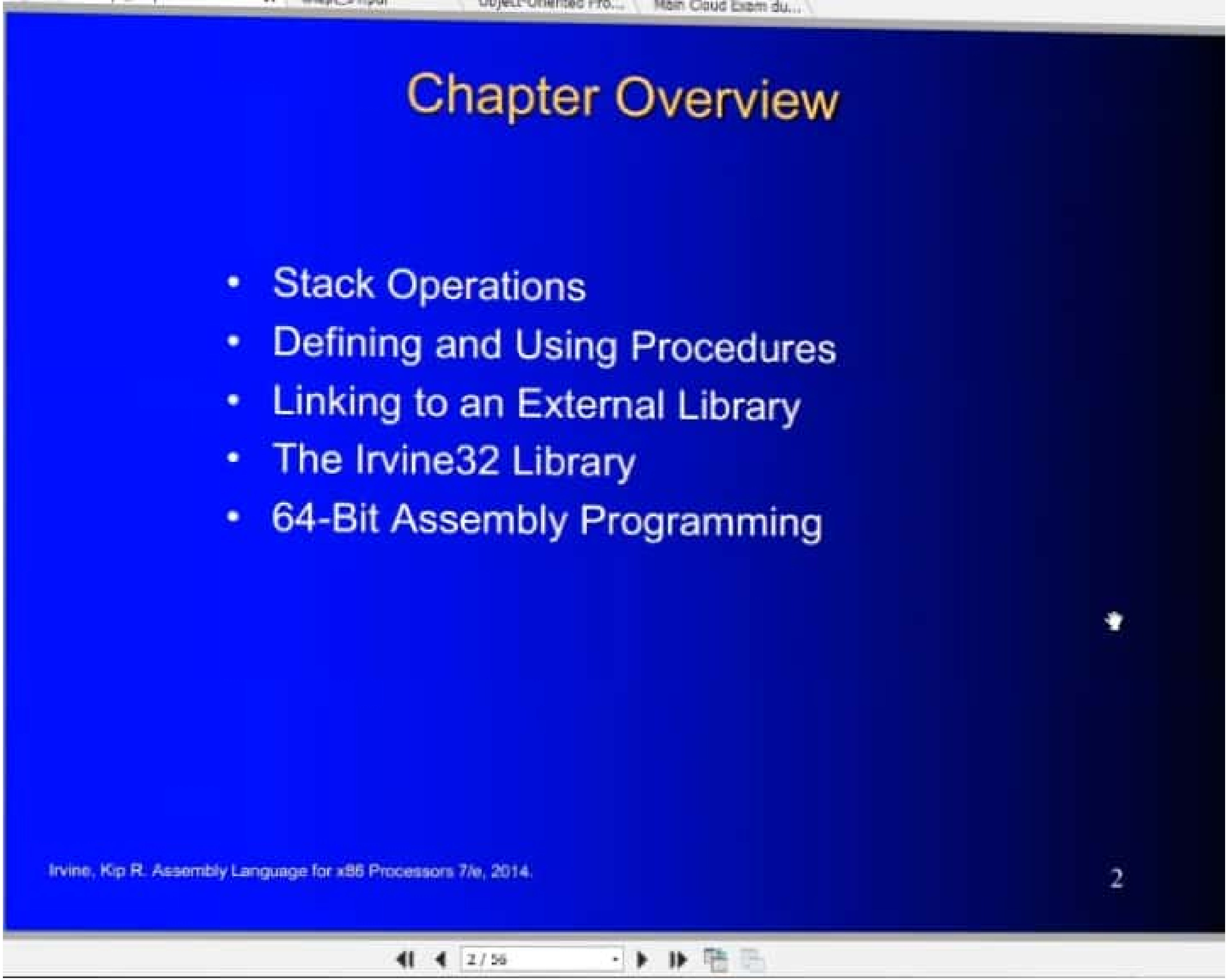
7th Edition

Kip R. Irvine

Chapter 5: Procedures

Slides prepared by the author

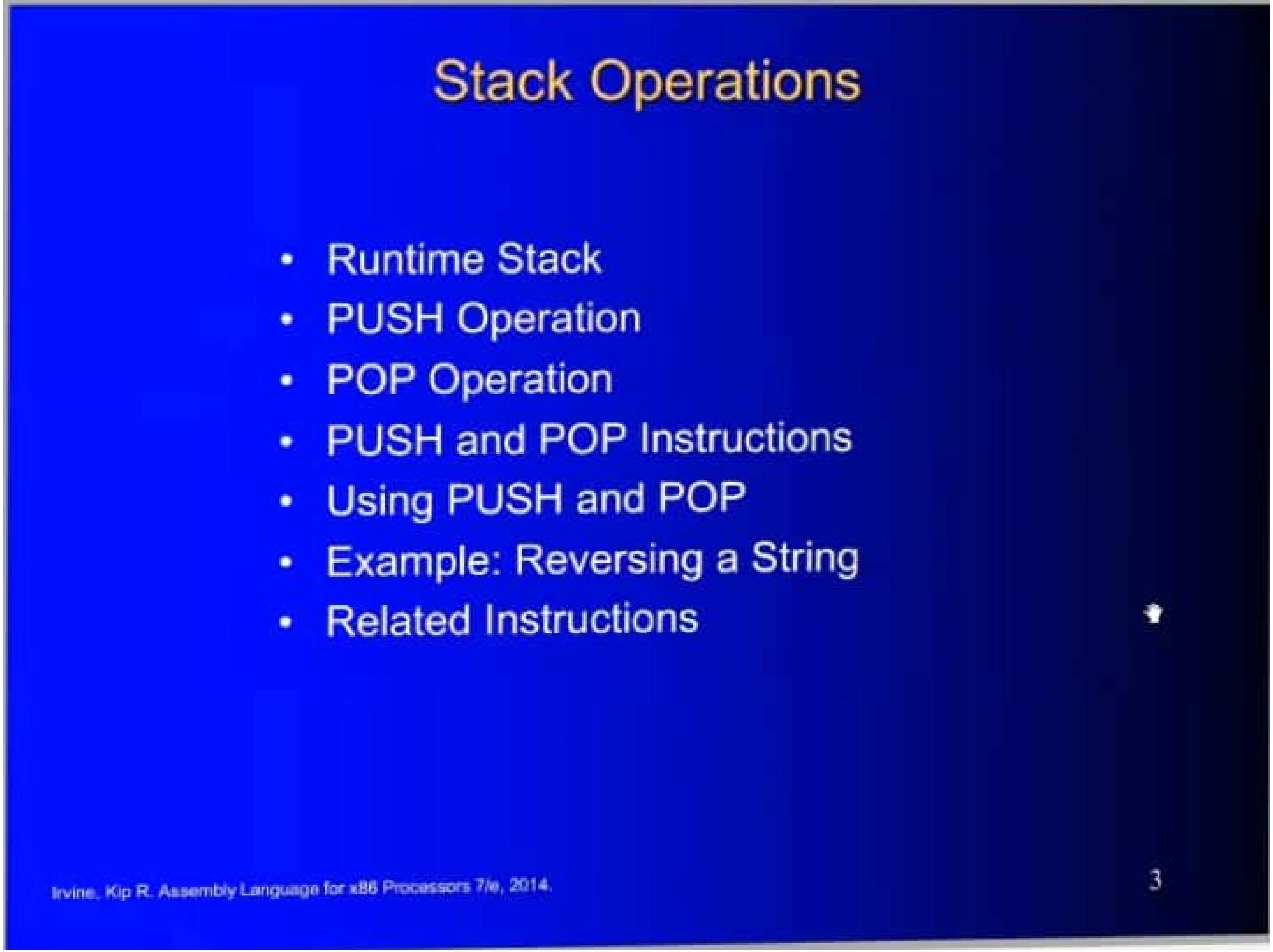
Revision date: 1/15/2014



Chapter Overview

- Stack Operations
- Defining and Using Procedures
- Linking to an External Library
- The Irvine32 Library
- 64-Bit Assembly Programming

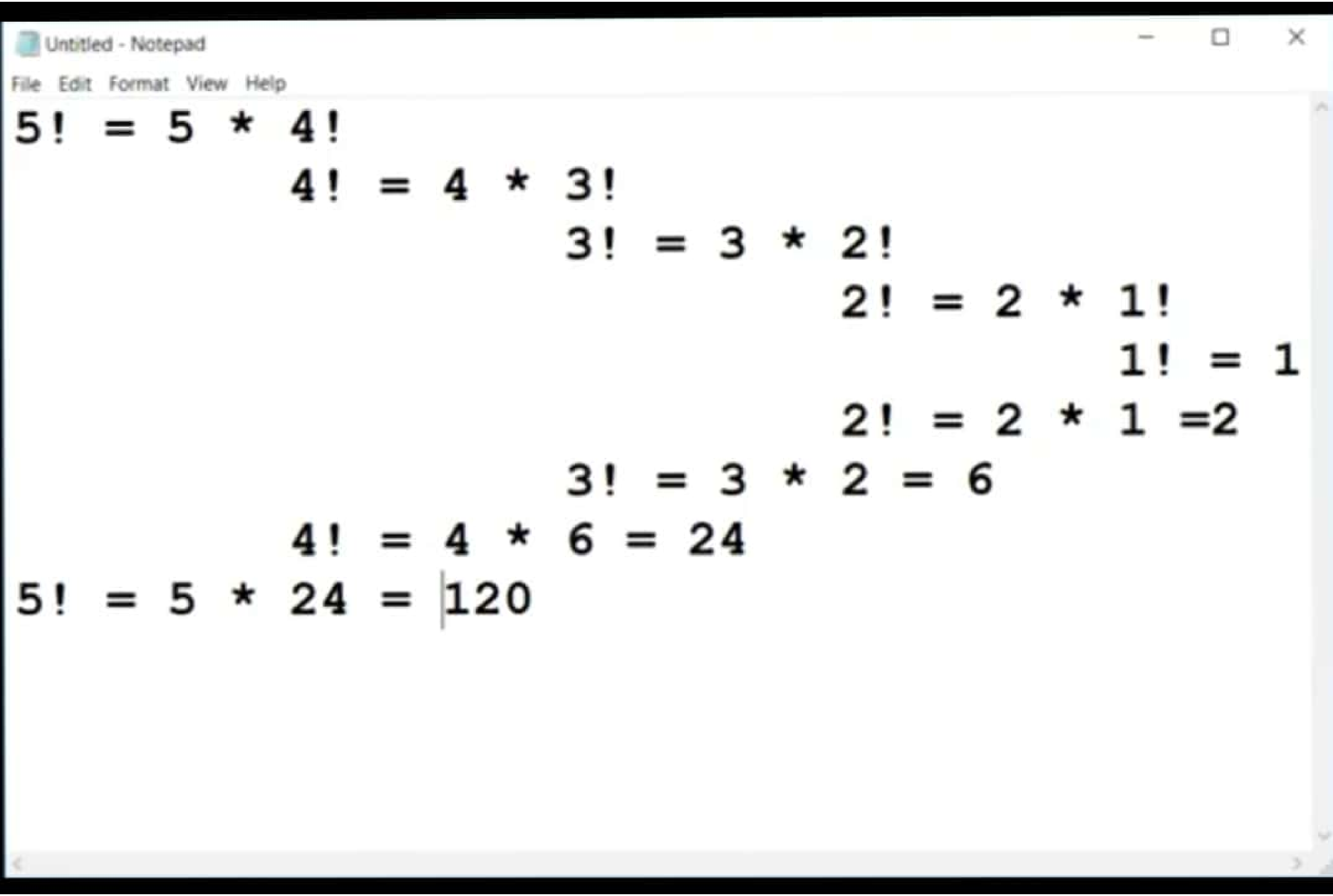
Irvine, Kip R. Assembly Language for x86 Processors 7/e, 2014.



Stack Operations

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

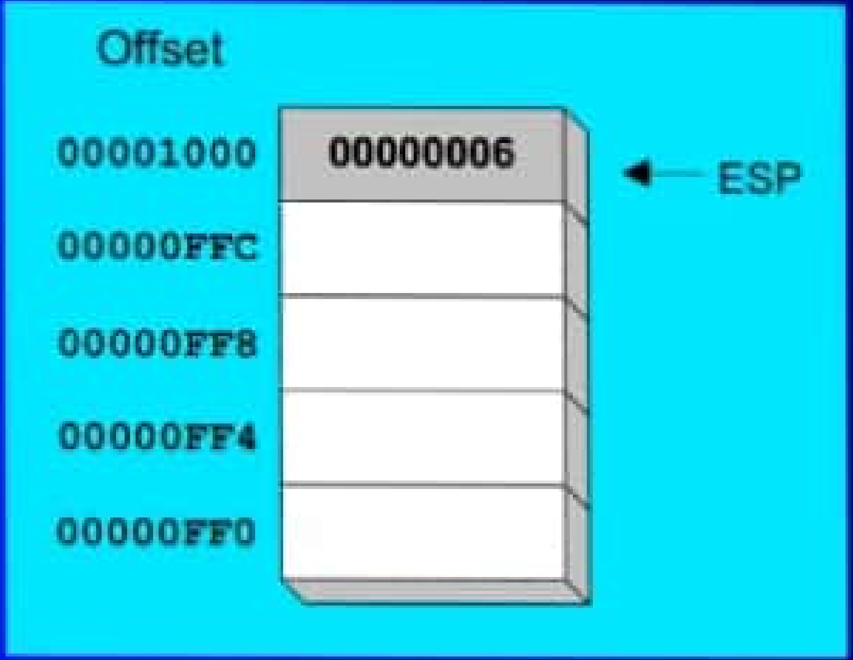
Irvine, Kip R. Assembly Language for x86 Processors 7/e, 2014.



```
5! = 5 * 4!
        4! = 4 * 3!
                3! = 3 * 2!
                        2! = 2 * 1!
                                1! = 1
                        2! = 2 * 1 =2
                3! = 3 * 2 = 6
        4! = 4 * 6 = 24
5! = 5 * 24 = 120
```

Runtime Stack

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



Offset

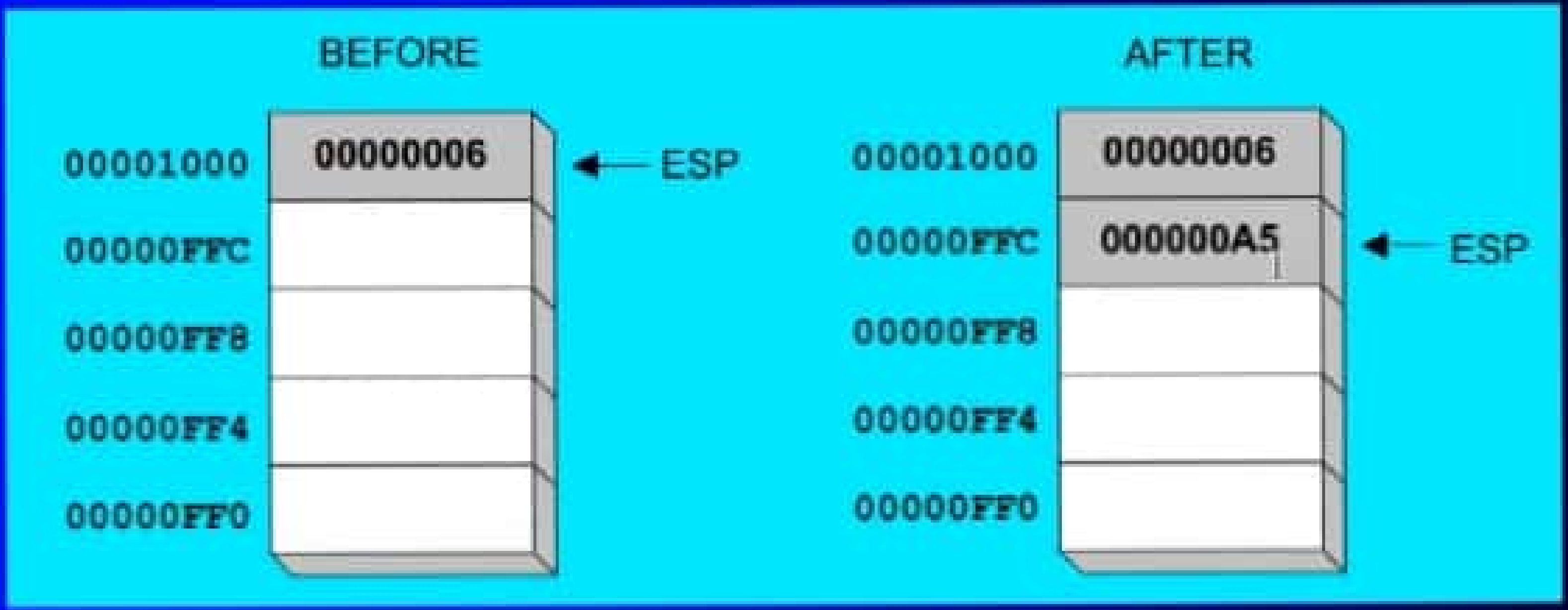
00001000	00000006
00000FFC	
00000FF8	
00000FF4	
00000FF0	

* SP in Real-address mode

Irvine, Kip R. Assembly Language for x86 Processors 7/e, 2014.

PUSH Operation (1 of 2)

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



BEFORE

00001000 00000006 ← ESP

00000FFC

00000FF8

00000FF4

00000FF0

AFTER

00001000 00000006

00000FFC 000000A5 ← ESP

00000FF8

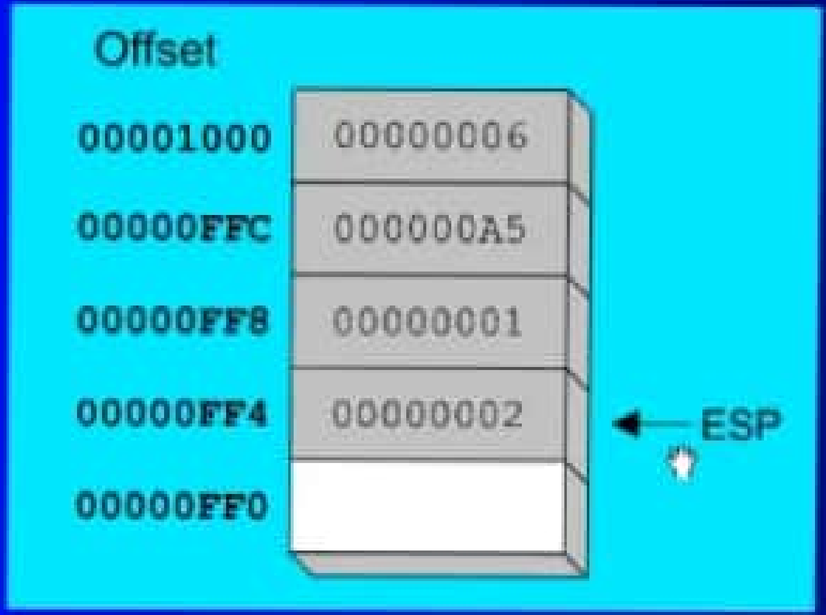
00000FF4

00000FF0

Irvine, Kip R. Assembly Language for x86 Processors 7/e, 2014.

PUSH Operation (2 of 2)

- Same stack after pushing two more integers:



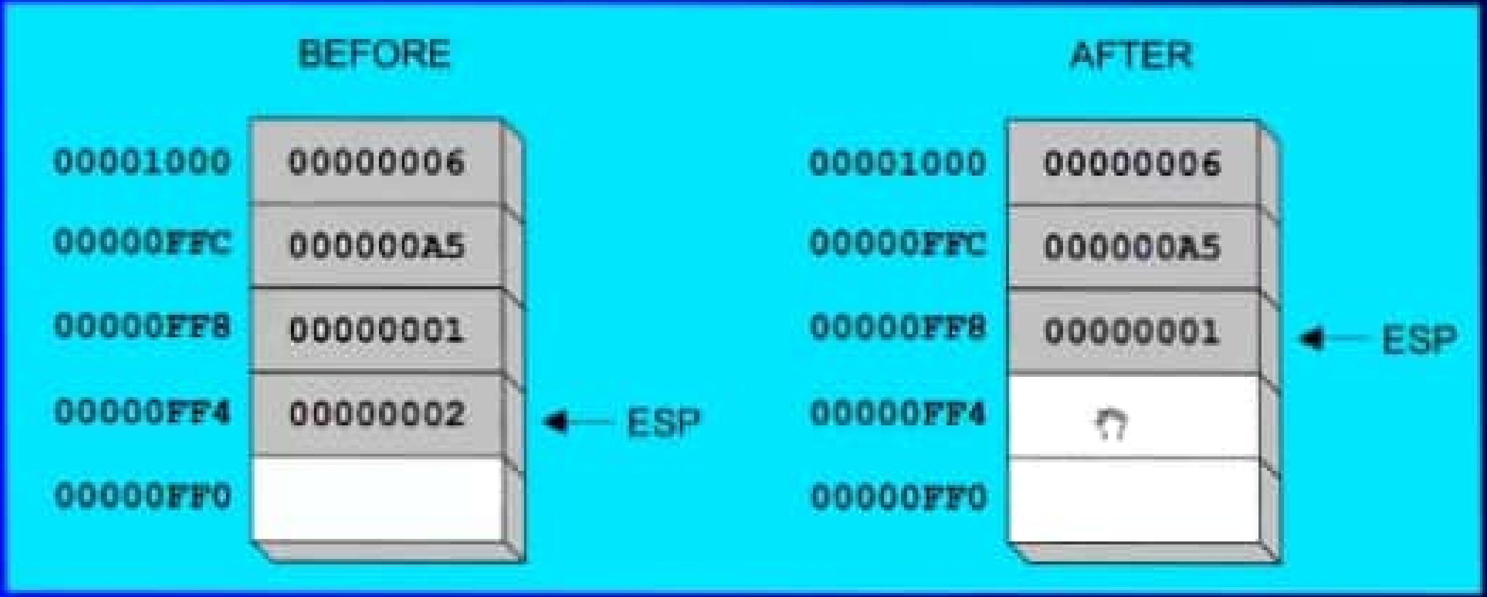
Offset		
00001000	00000006	
00000FFC	000000A5	
00000FF8	00000001	
00000FF4	00000002	← ESP
00000FF0		

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

Irvine, Kip R. Assembly Language for x86 Processors 7/e, 2014.

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



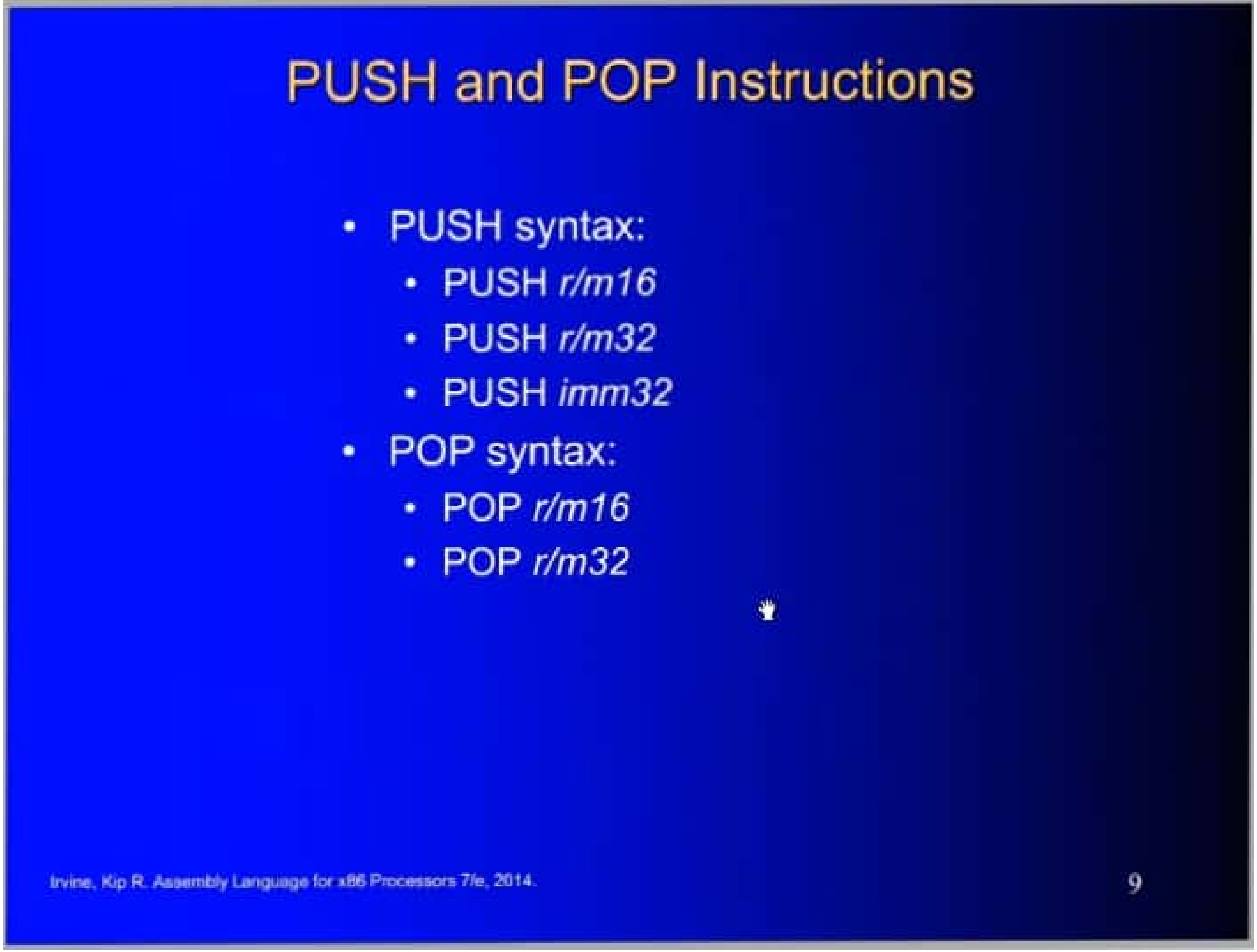
BEFORE

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

AFTER

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

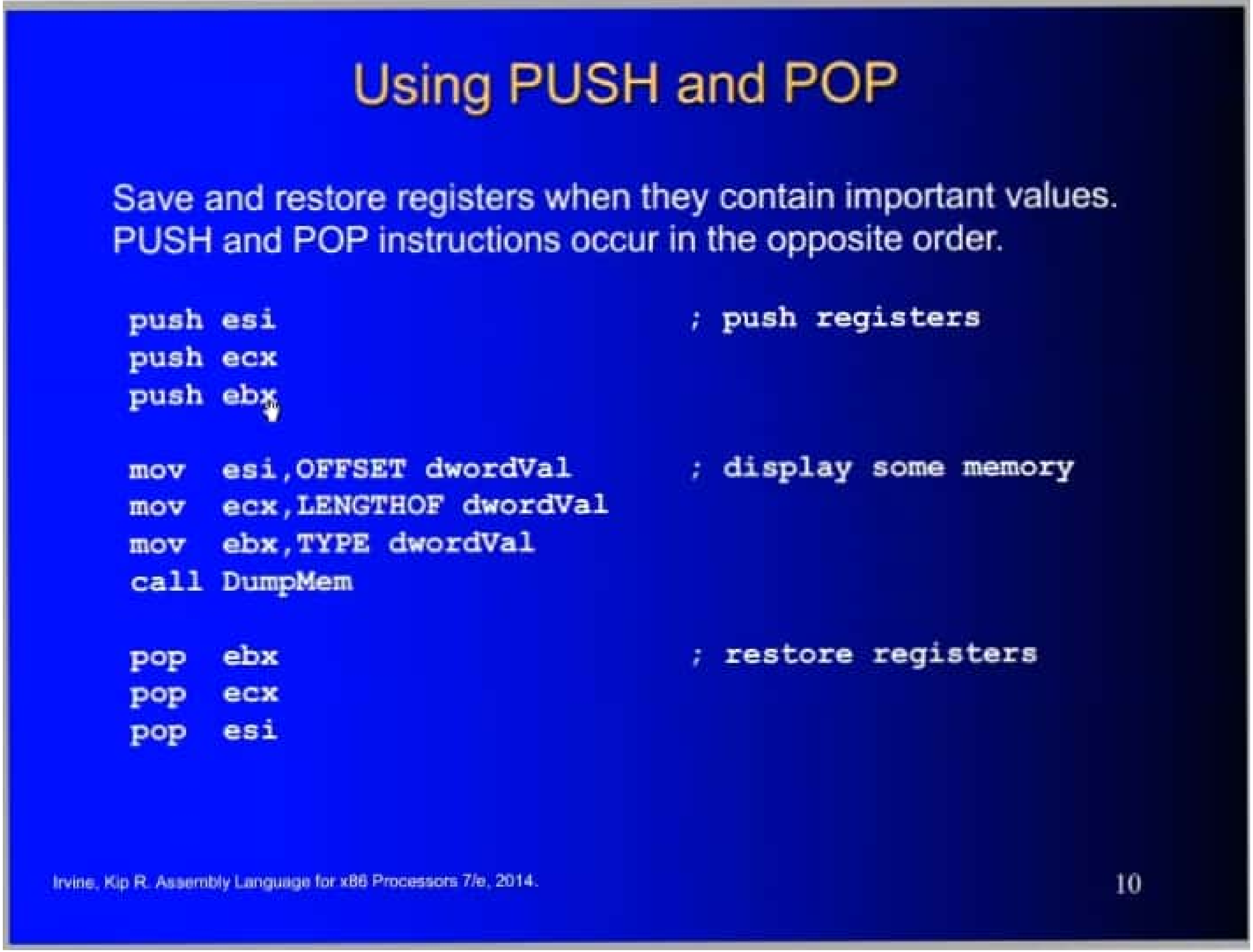
Irvine, Kip R. Assembly Language for x86 Processors 7/e, 2014.



PUSH and POP Instructions

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

Irvine, Kip R. Assembly Language for x86 Processors 7/e, 2014.



Using PUSH and POP

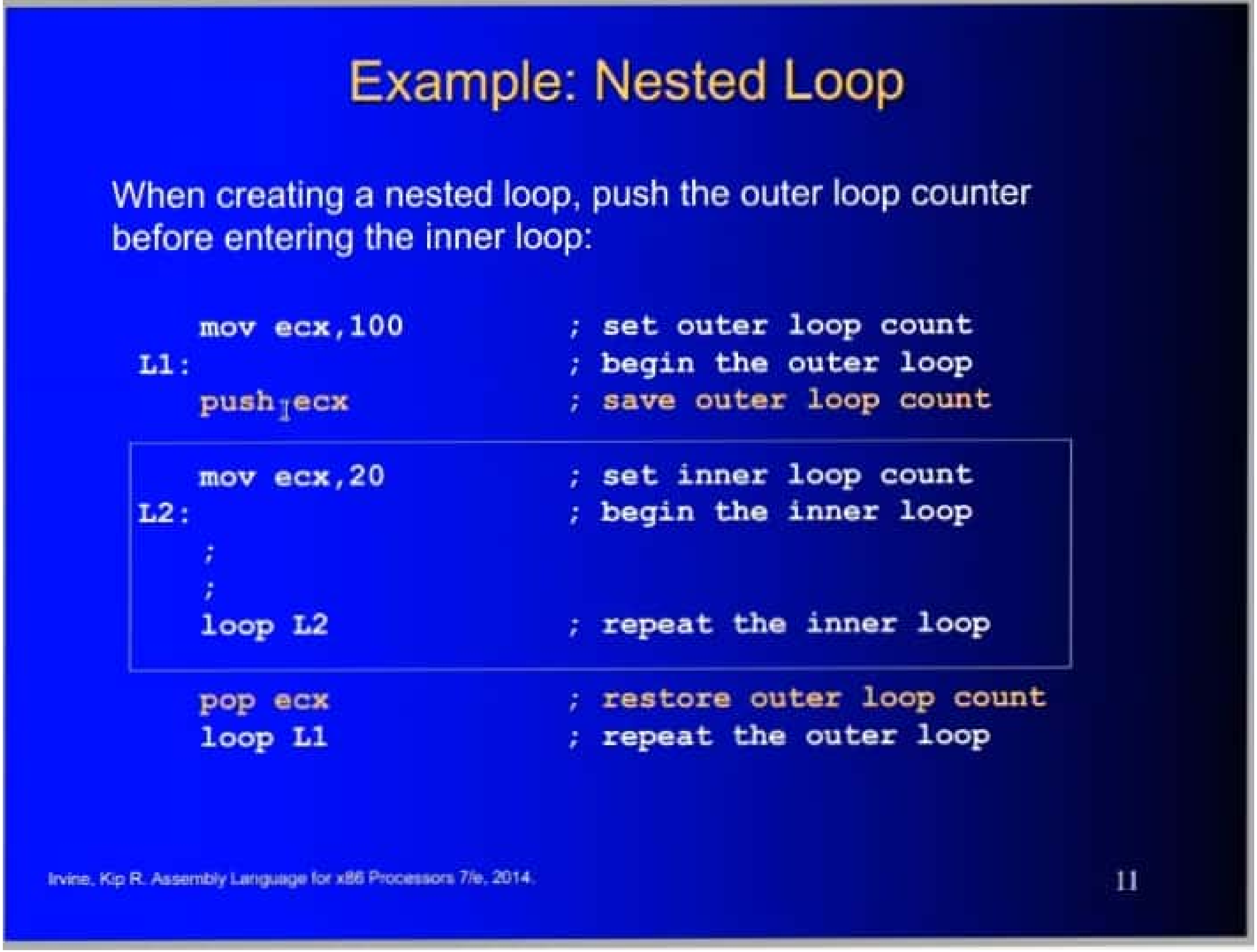
Save and restore registers when they contain important values. PUSH and POP instructions occur in the opposite order.

```
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
pop  esi
```

Irvine, Kip R. Assembly Language for x86 Processors 7/e, 2014.



Example: Nested Loop

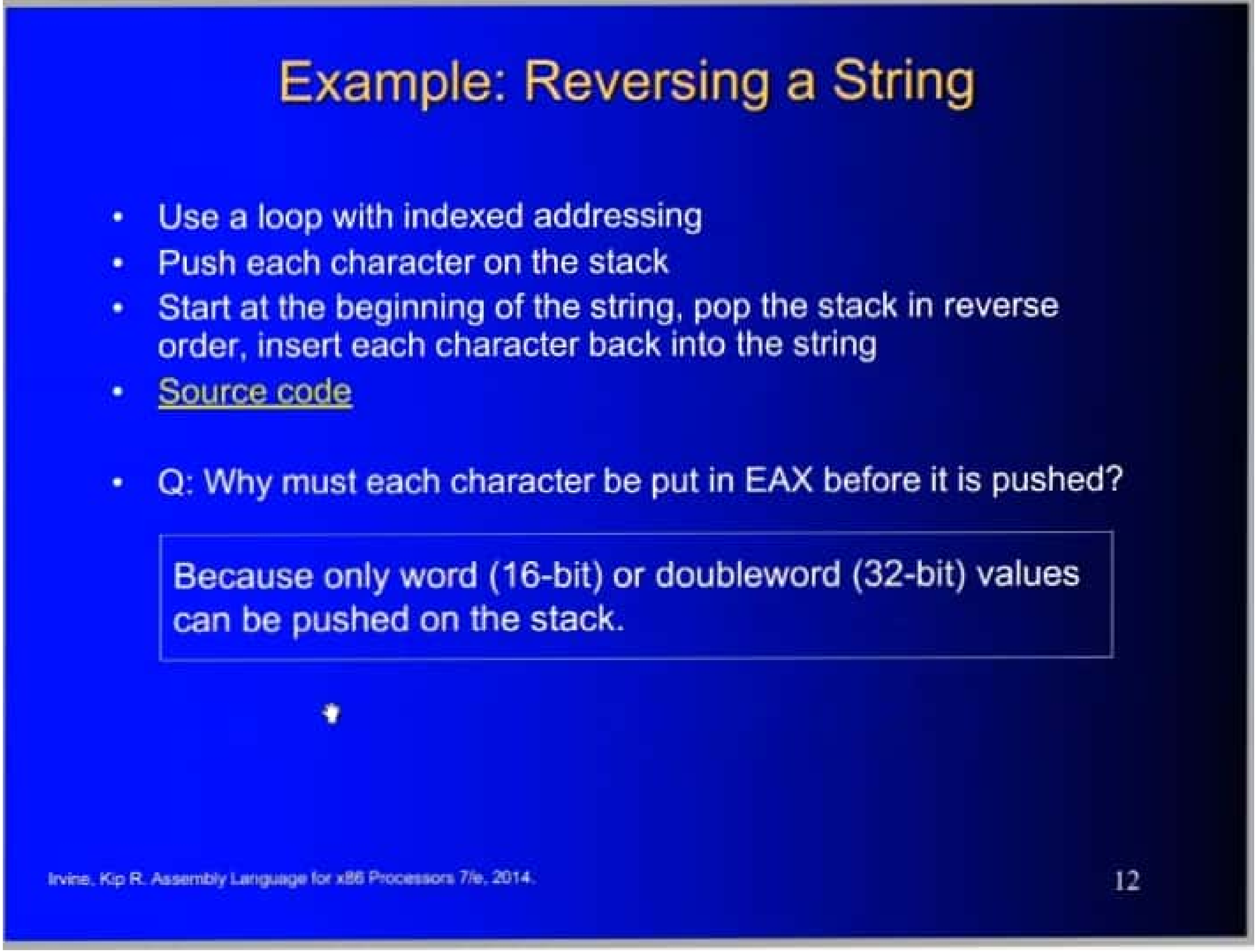
When creating a nested loop, push the outer loop counter before entering the inner loop:

```
    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
```

Irvine, Kip R. Assembly Language for x86 Processors 7/e, 2014.

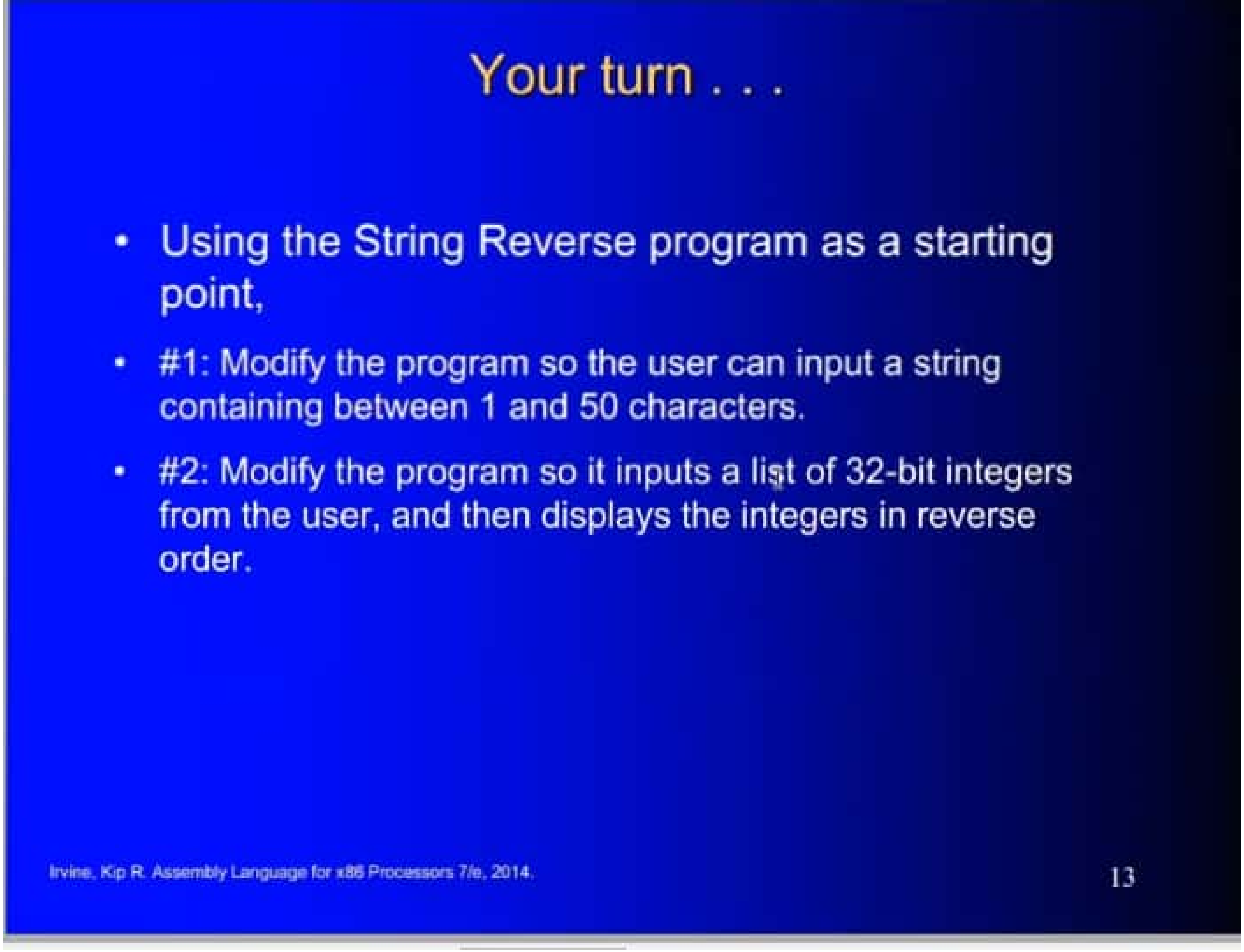


Example: Reversing a String

- 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
- Source code

- Q: Why must each character be put in EAX before it is pushed?

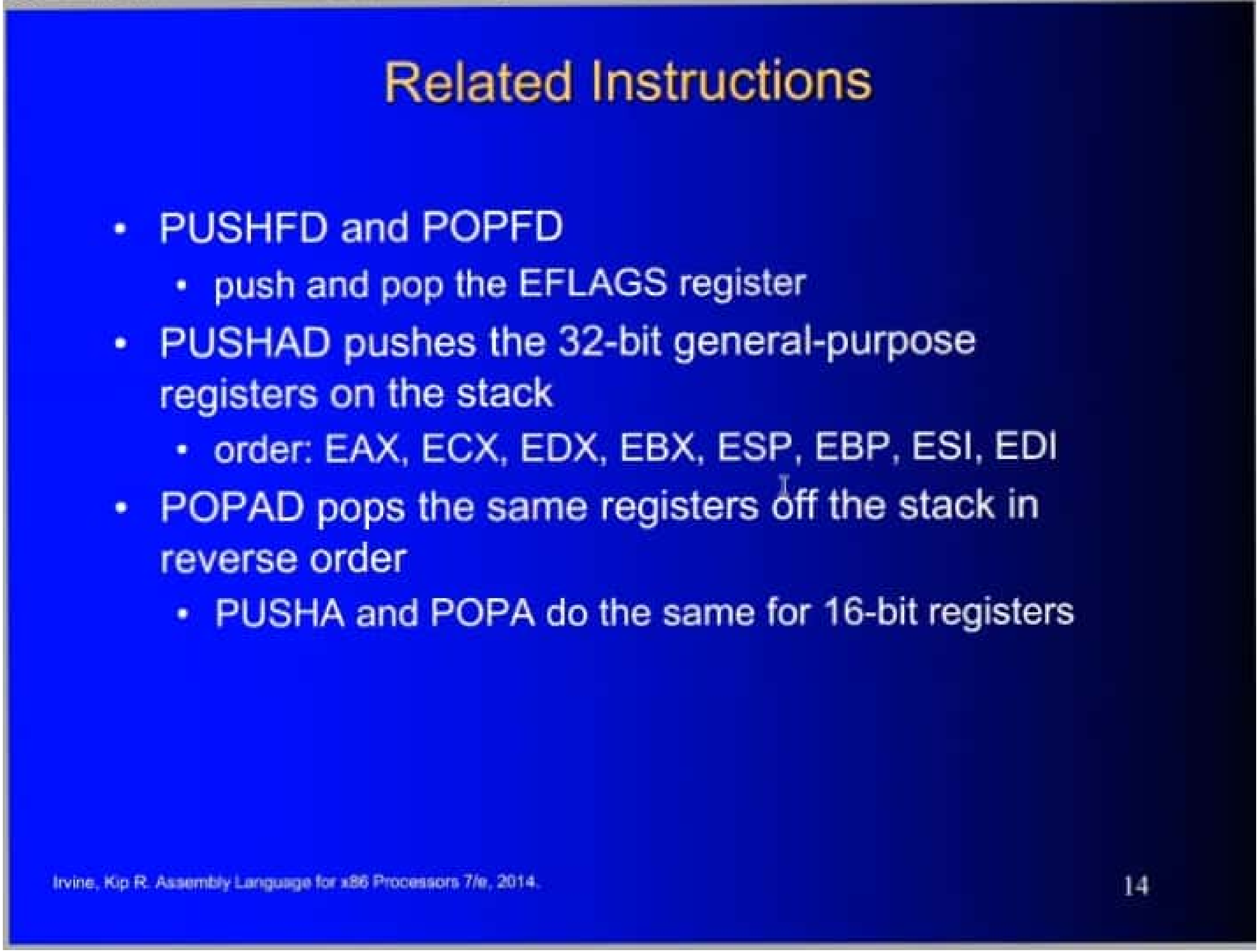
Because only word (16-bit) or doubleword (32-bit) values can be pushed on the stack.



Your turn . . .

- Using the String Reverse program as a starting point,
- #1: Modify the program so the user can input a string containing between 1 and 50 characters.
- #2: Modify the program so it inputs a list of 32-bit integers from the user, and then displays the integers in reverse order.

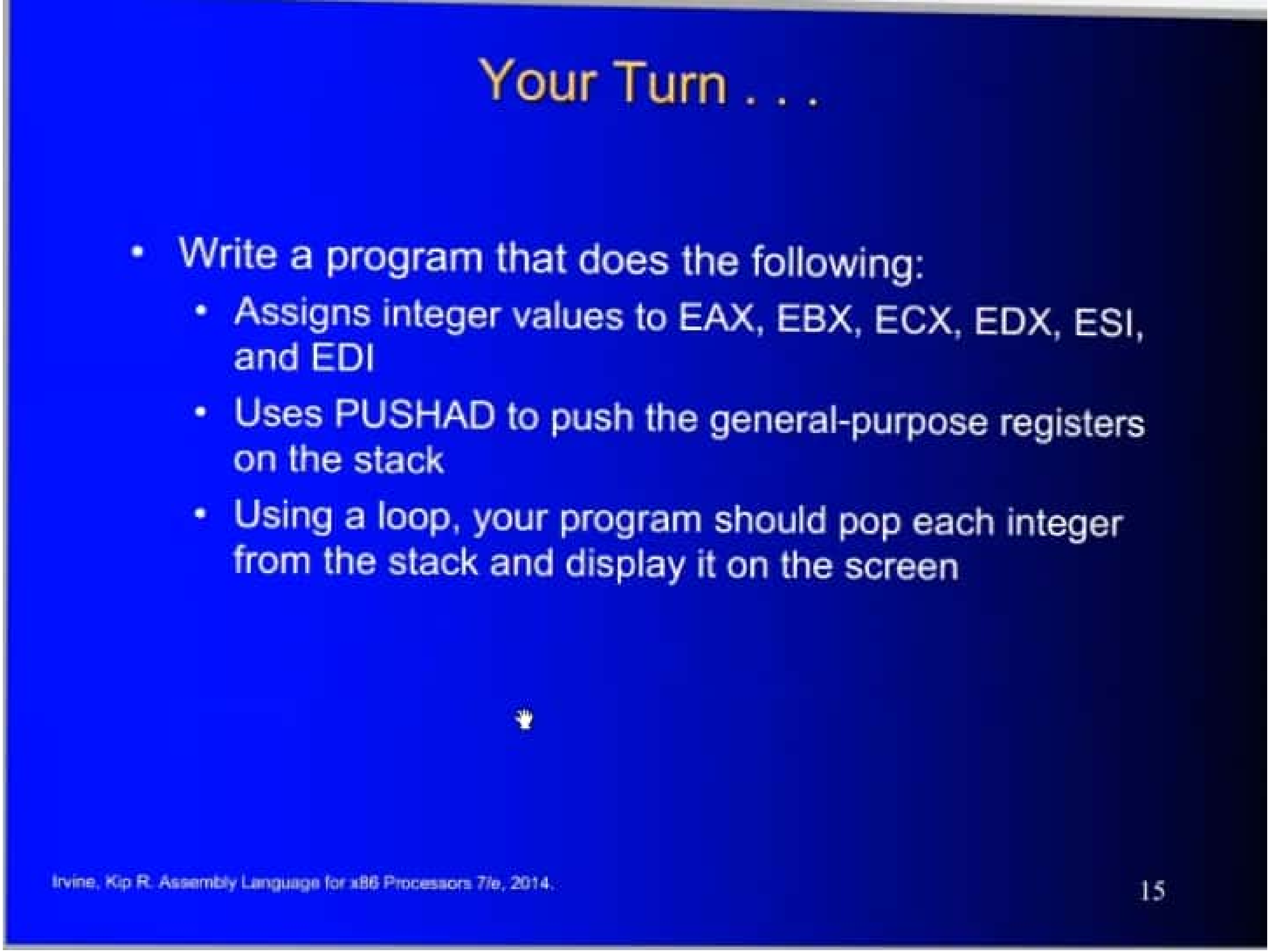
Irvine, Kip R. Assembly Language for x86 Processors 7/e, 2014.



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

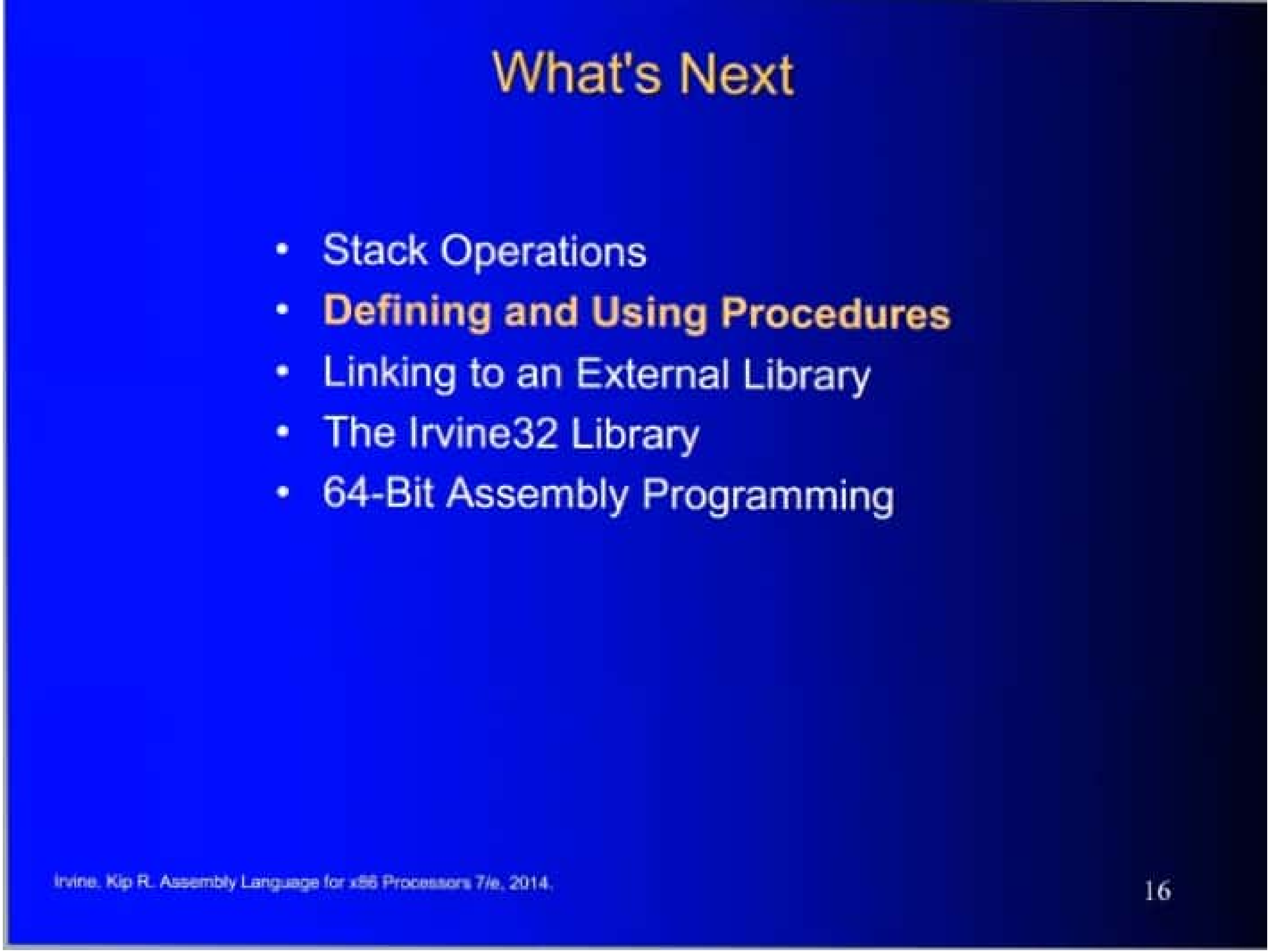
Irvine, Kip R. Assembly Language for x86 Processors 7/e, 2014.



Your Turn . . .

- Write a program that does the following:
  - Assigns integer values to EAX, EBX, ECX, EDX, ESI, and EDI
  - Uses PUSHAD to push the general-purpose registers on the stack
  - Using a loop, your program should pop each integer from the stack and display it on the screen

Irvine, Kip R. Assembly Language for x86 Processors 7/e, 2014.



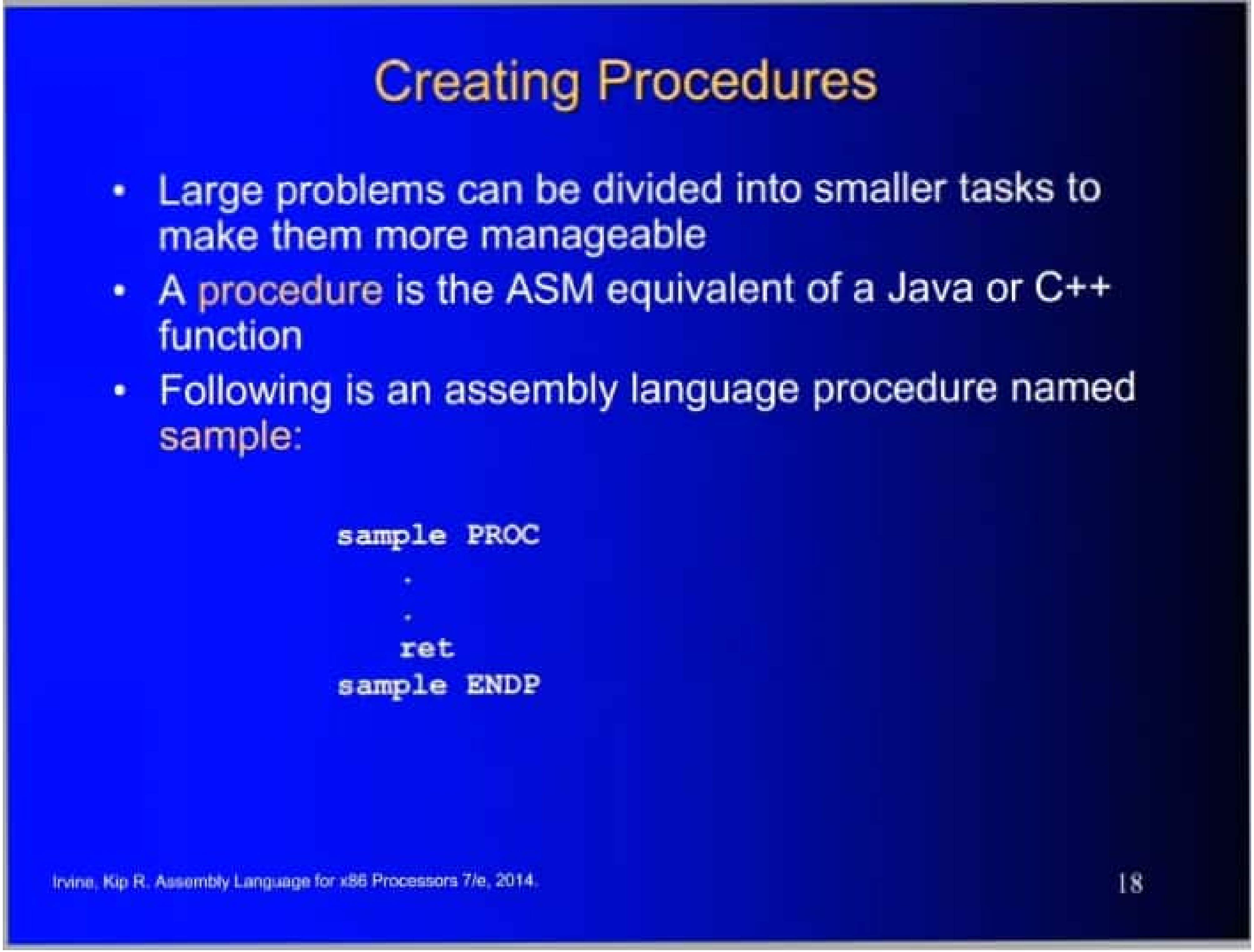
What's Next

- Stack Operations
- **Defining and Using Procedures**
- Linking to an External Library
- The Irvine32 Library
- 64-Bit Assembly Programming

Irvine, Kip R. Assembly Language for x86 Processors 7/e, 2014.

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

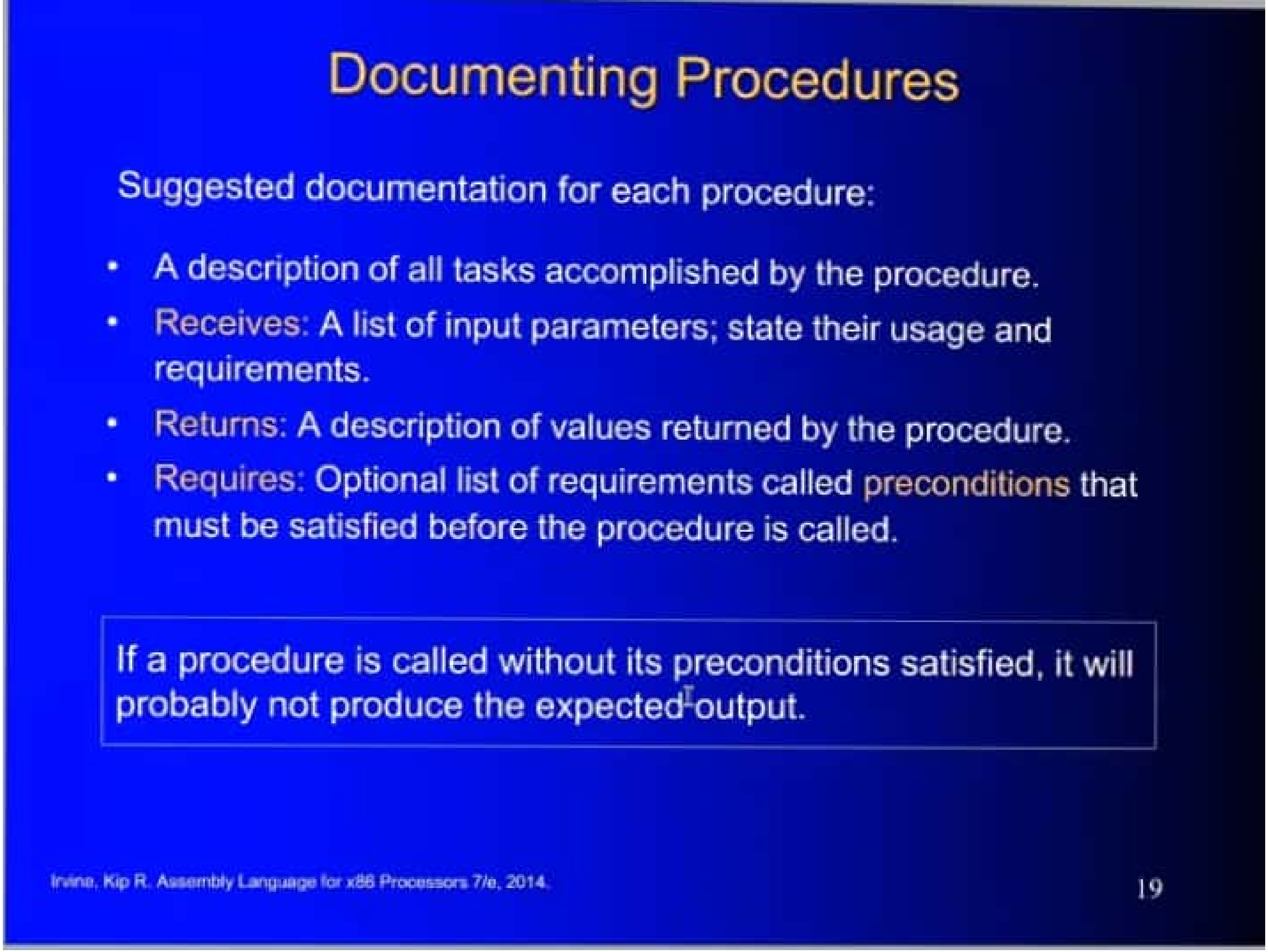


Creating Procedures

- 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
```

Irvine, Kip R. Assembly Language for x86 Processors 7/e, 2014.



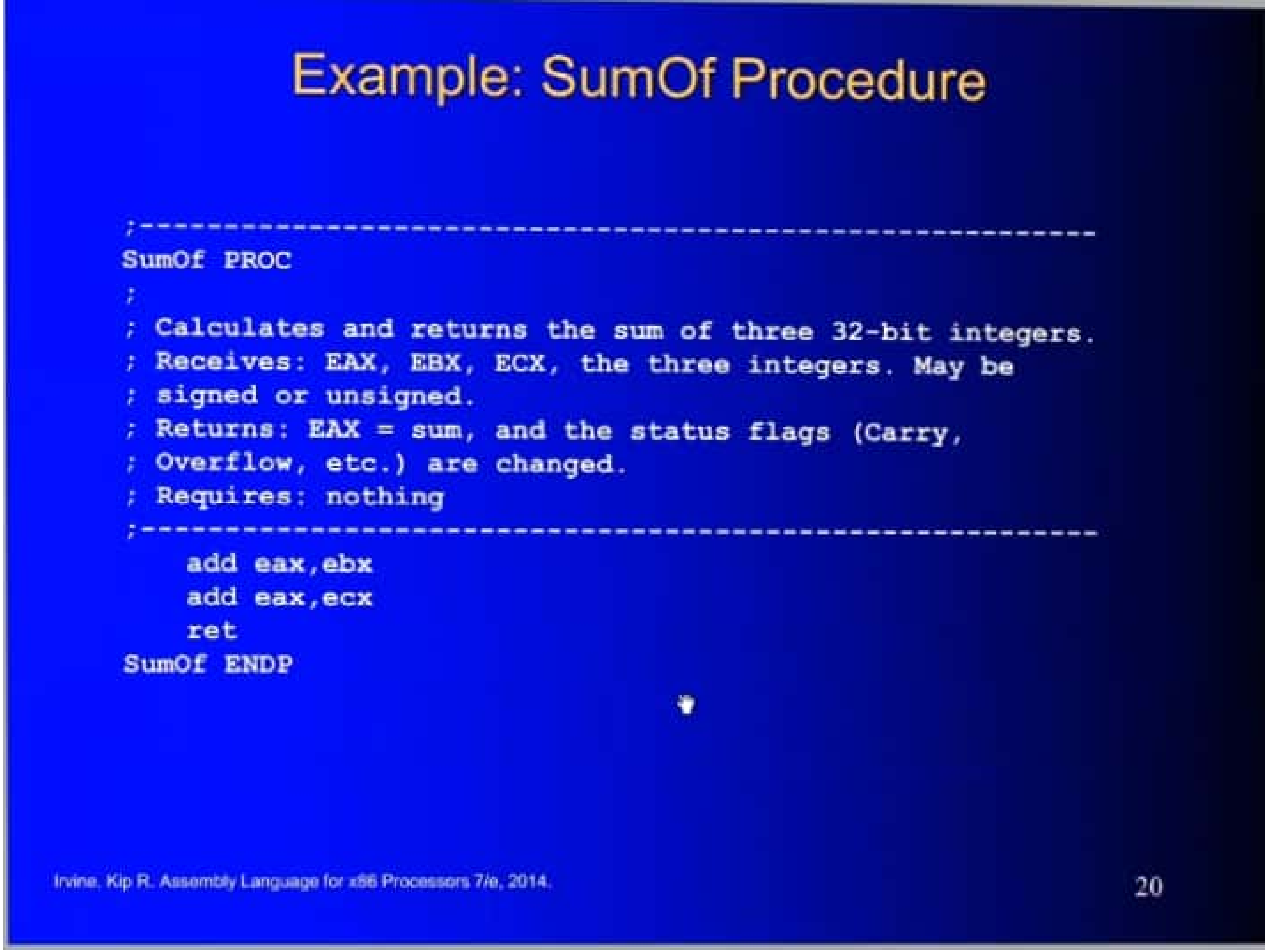
Documenting Procedures

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.

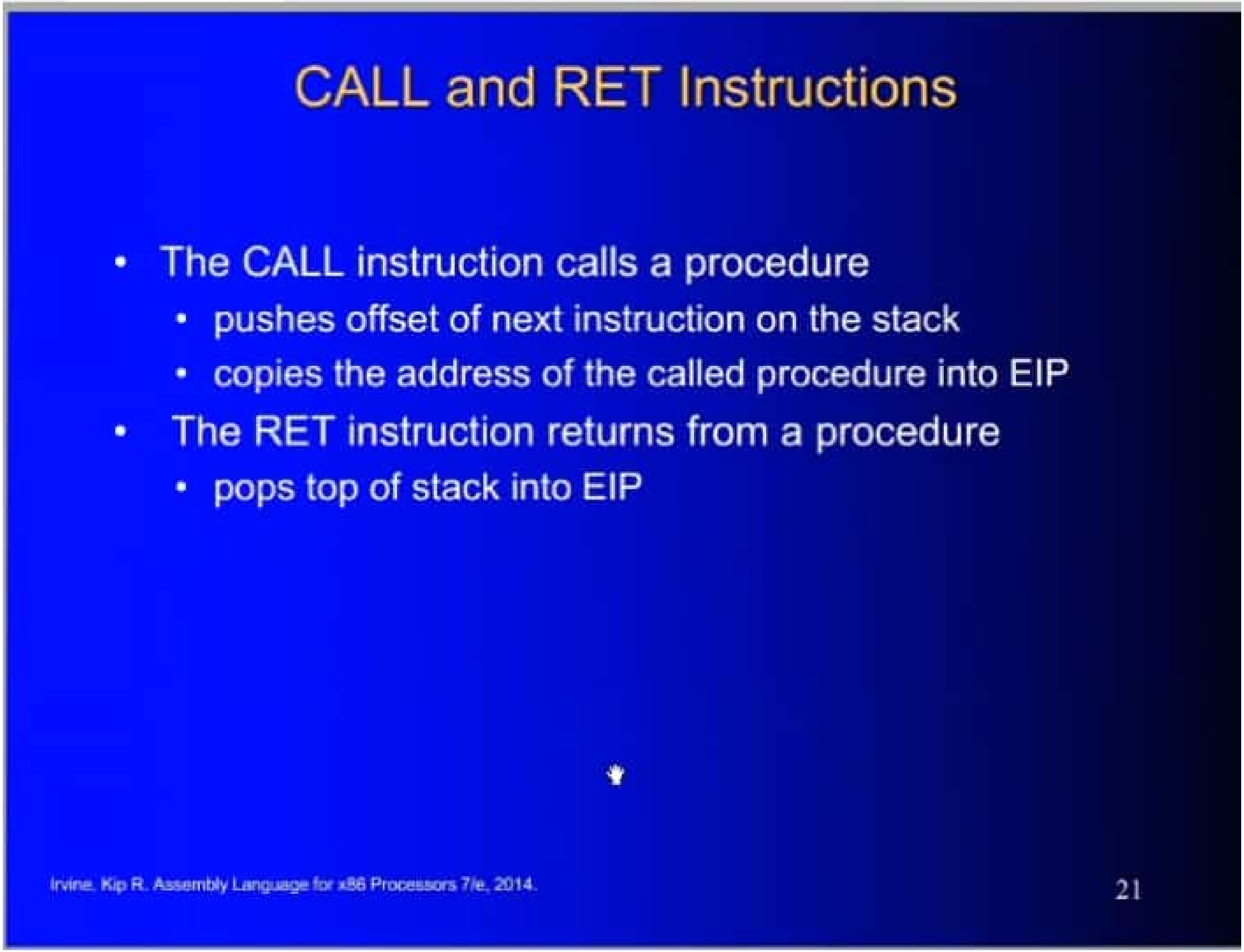
Irvine, Kip R. Assembly Language for x86 Processors 7/e, 2014.



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
```

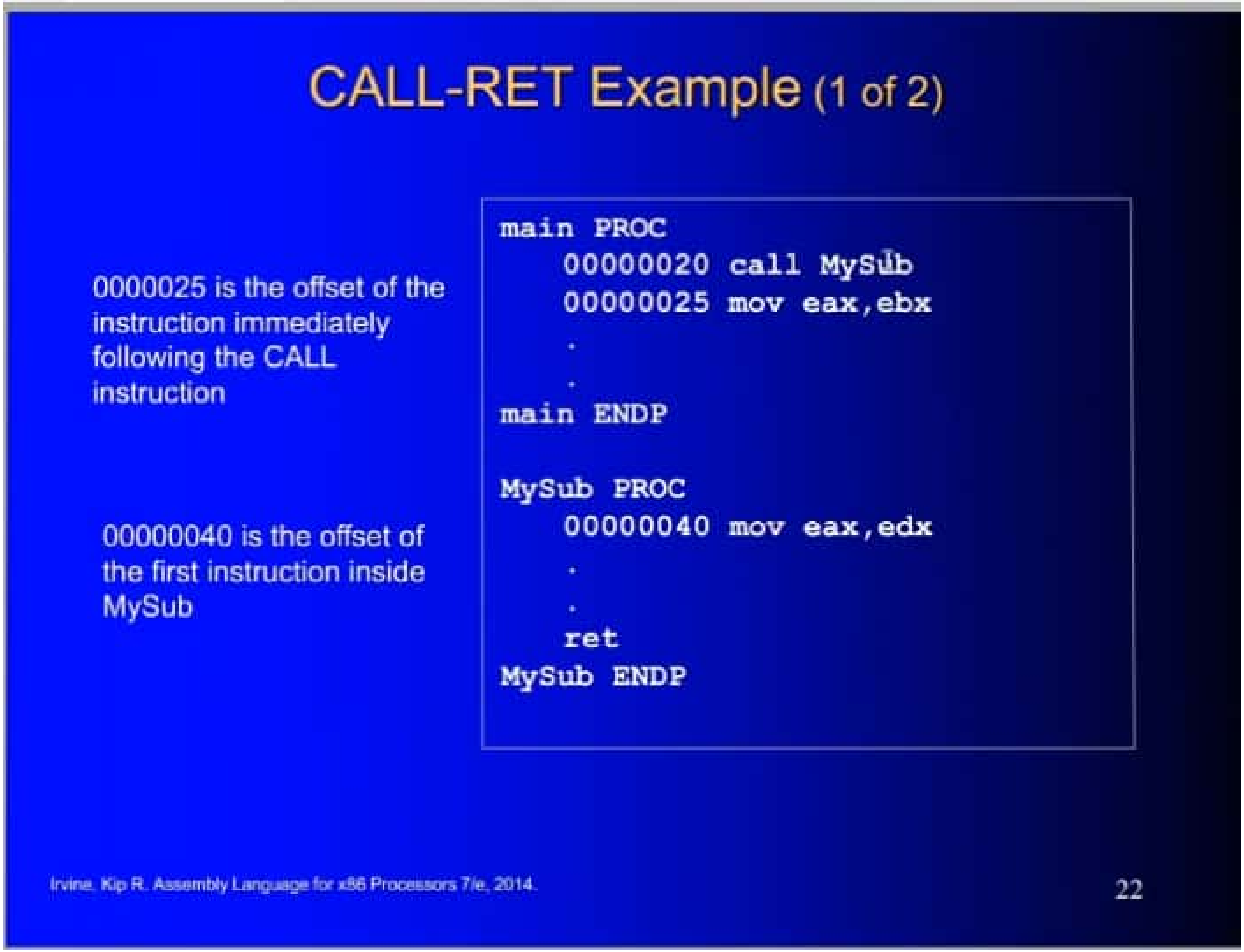
Irvine, Kip R. Assembly Language for x86 Processors 7/e, 2014.



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

Irvine, Kip R. Assembly Language for x86 Processors 7/e, 2014.



CALL-RET Example (1 of 2)

0000025 is the offset of the instruction immediately following the CALL instruction

00000040 is the offset of the first instruction inside MySub

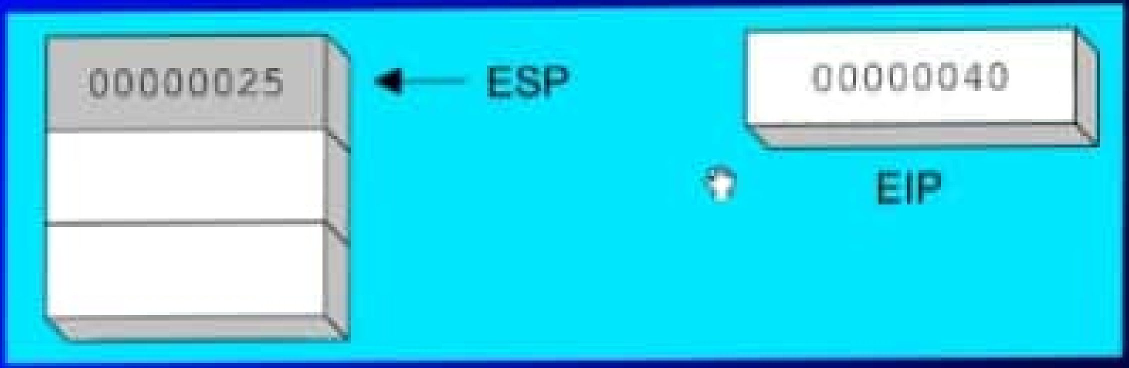
```
main PROC
    00000020 call MySub
    00000025 mov eax,ebx
    .
    .
main ENDP

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

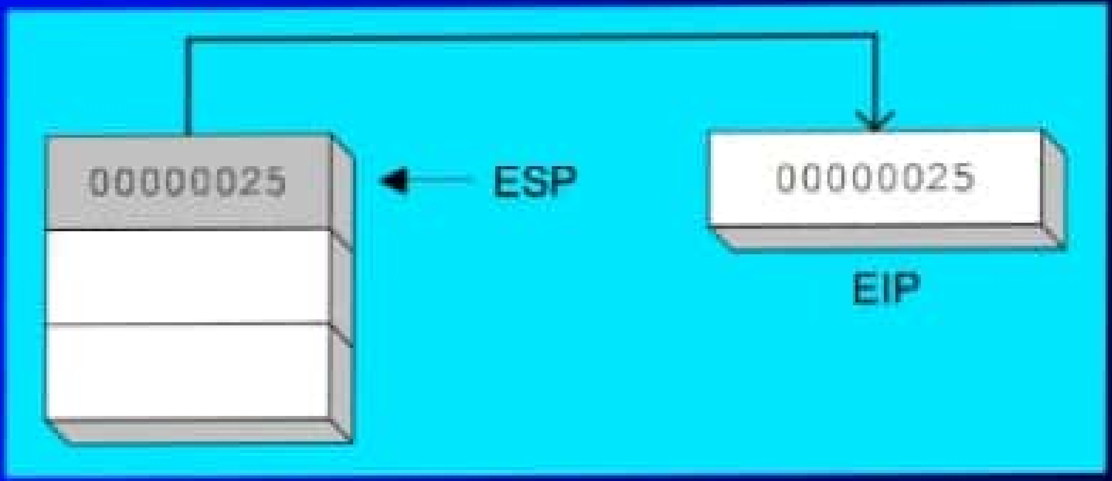
Irvine, Kip R. Assembly Language for x86 Processors 7/e, 2014.

CALL-RET Example (2 of 2)

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

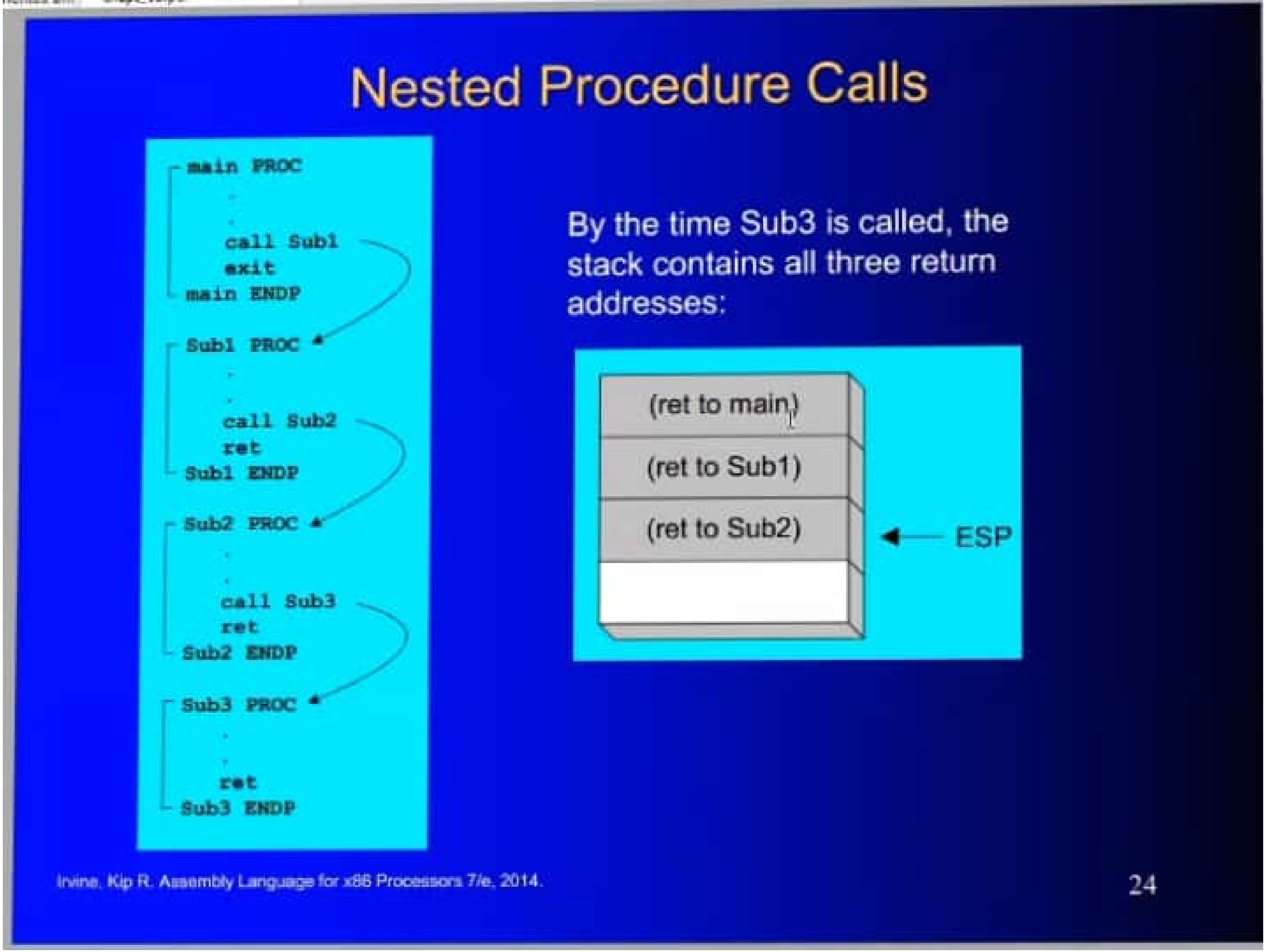


The RET instruction pops 00000025 from the stack into EIP



(stack shown before RET executes)

Irvine, Kip R. Assembly Language for x86 Processors 7/e, 2014.



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:

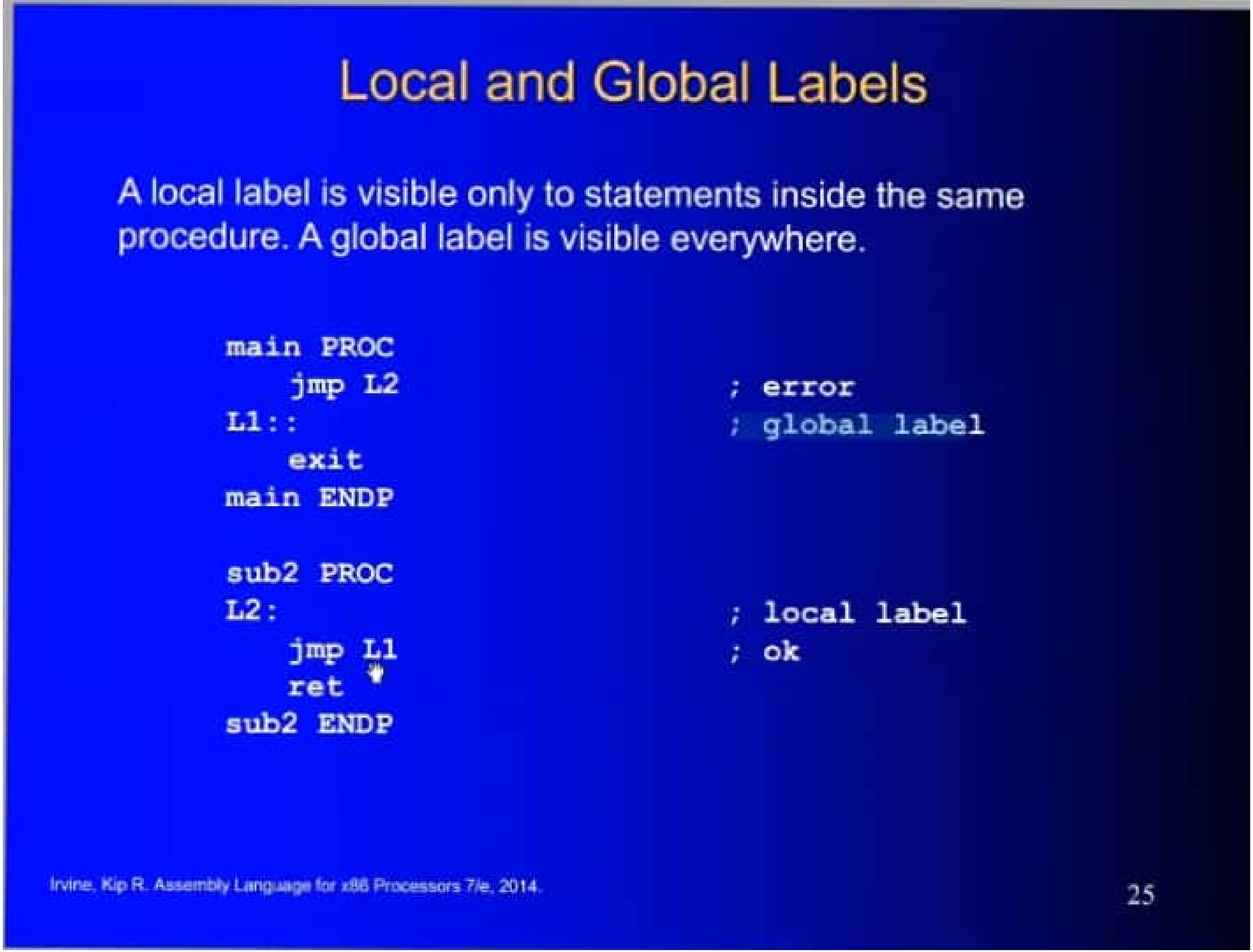
(ret to main)

(ret to Sub1)

(ret to Sub2)

ESP

Irvine, Kip R. Assembly Language for x86 Processors 7/e, 2014.



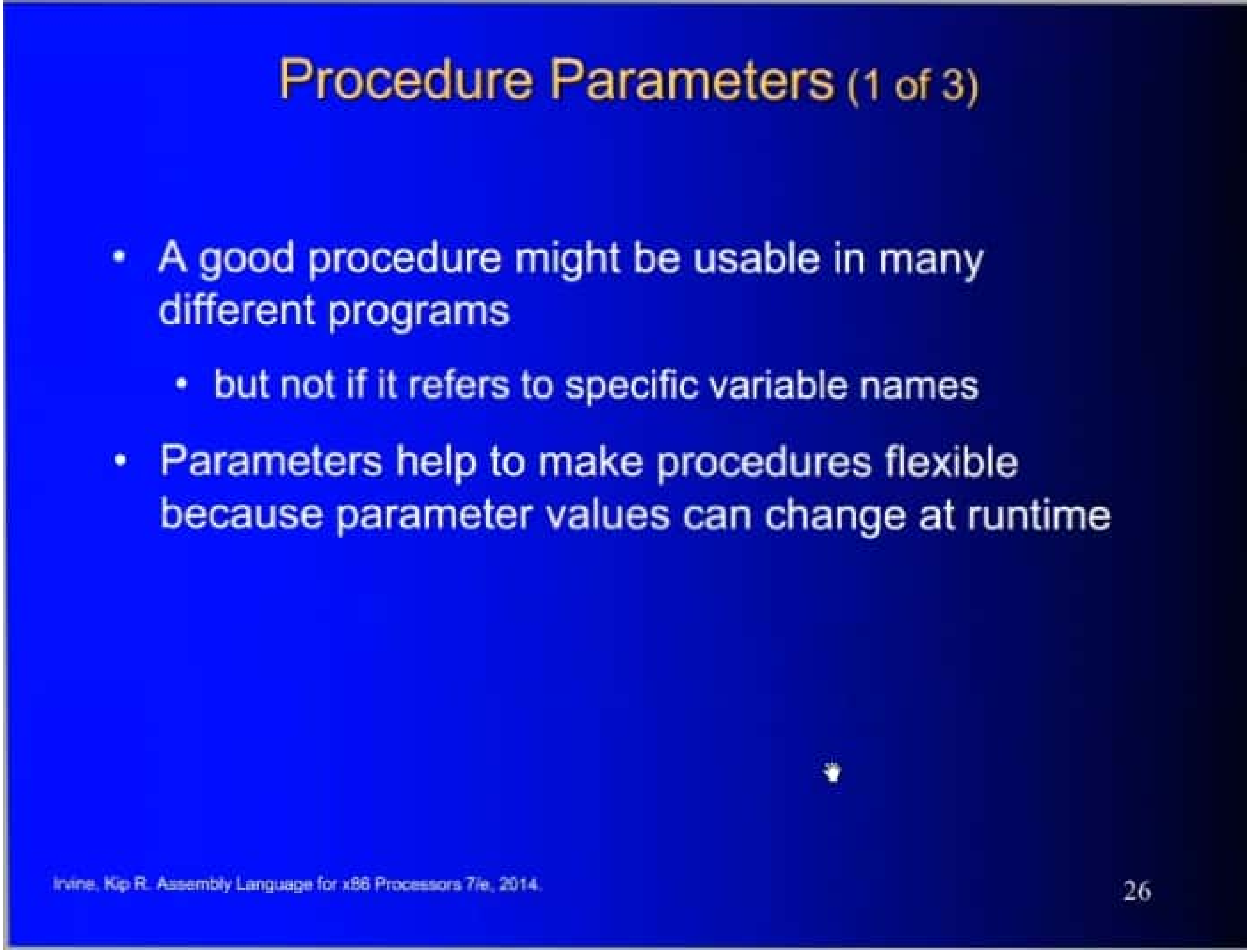
Local and Global Labels

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
```

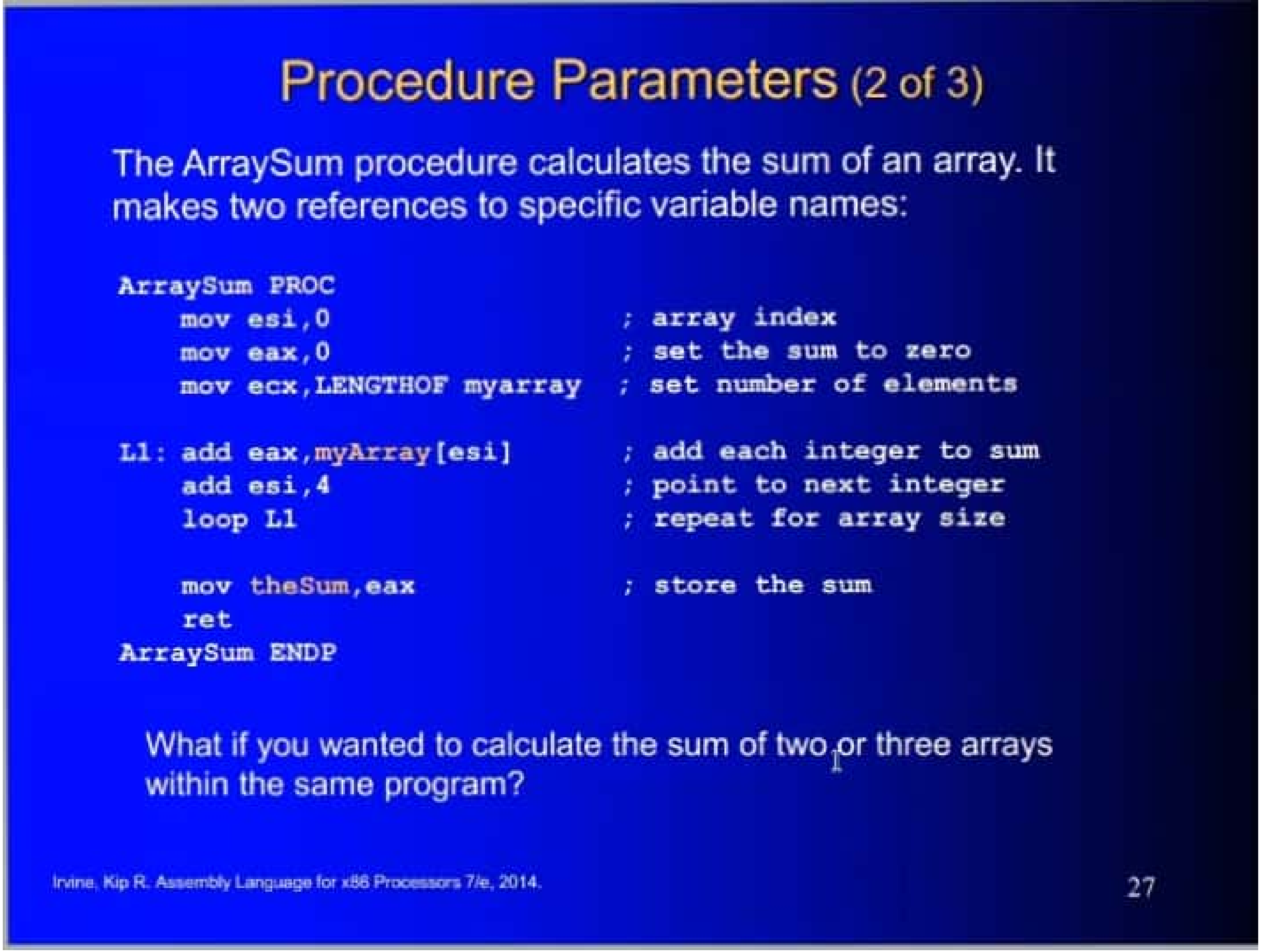
Irvine, Kip R. Assembly Language for x86 Processors 7/e, 2014.



Procedure Parameters (1 of 3)

- 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

Irvine, Kip R. Assembly Language for x86 Processors 7/e, 2014.



Procedure Parameters (2 of 3)

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

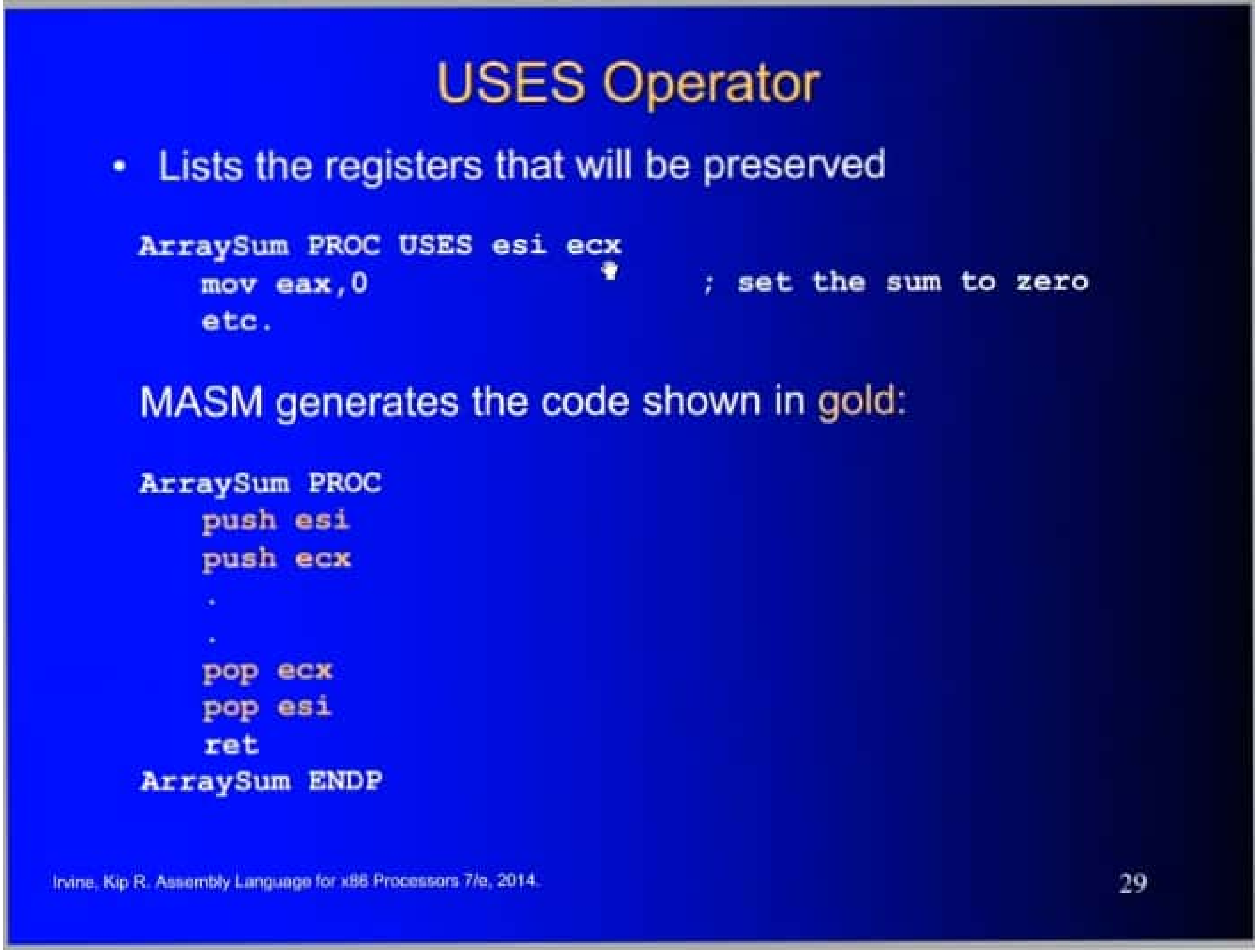
```
ArraySum PROC
    mov esi,0                   ; array index
    mov eax,0                   ; set the sum to zero
    mov ecx,LENGTHOF myarray    ; set number of elements

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?

Irvine, Kip R. Assembly Language for x86 Processors 7/e, 2014.



USES Operator

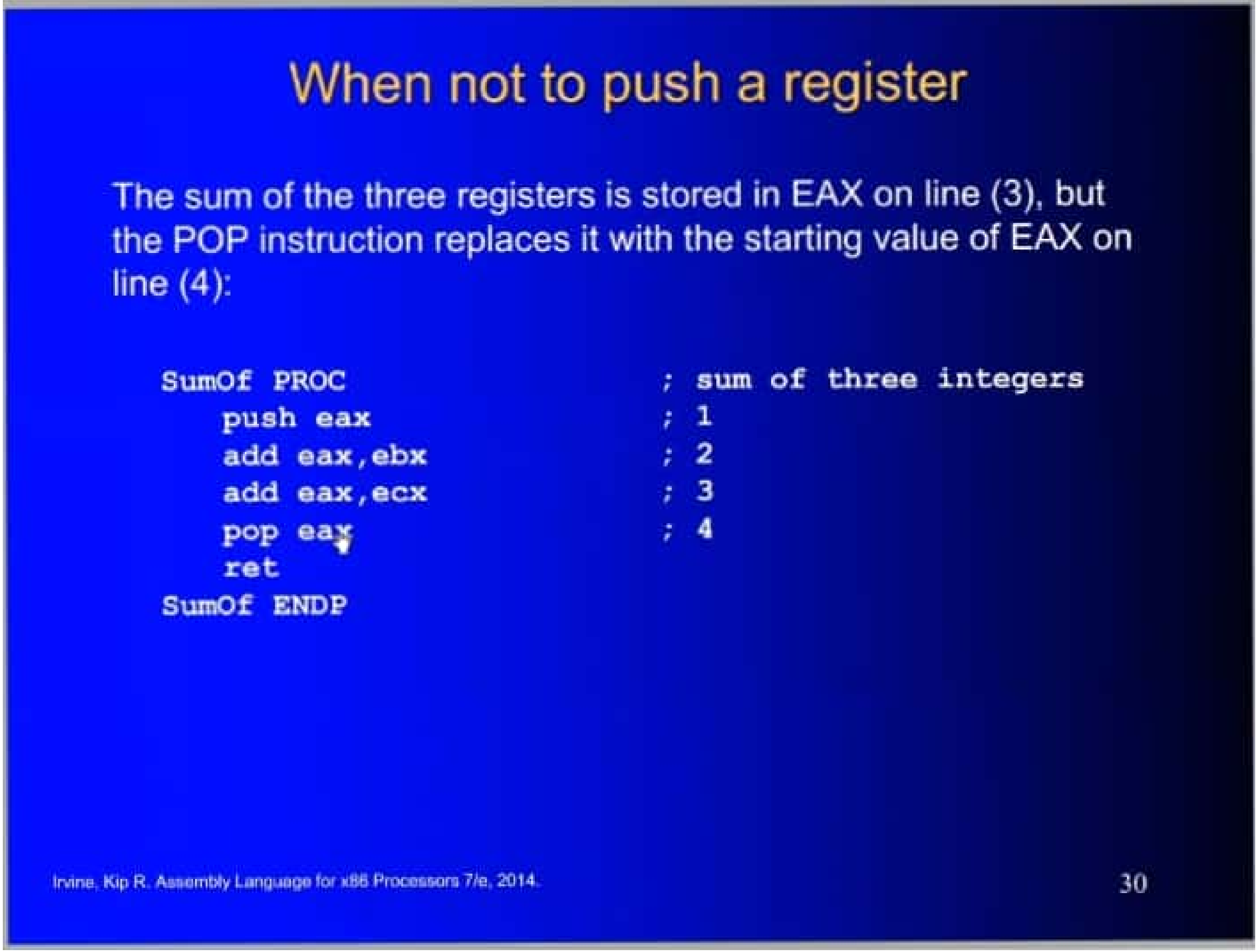
- Lists the registers that will be preserved

```
ArraySum PROC USES esi ecx
    mov eax,0                    ; set the sum to zero
    etc.
```

MASM generates the code shown in gold:

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

Irvine, Kip R. Assembly Language for x86 Processors 7/e, 2014.



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
```

Irvine, Kip R. Assembly Language for x86 Processors 7/e, 2014.