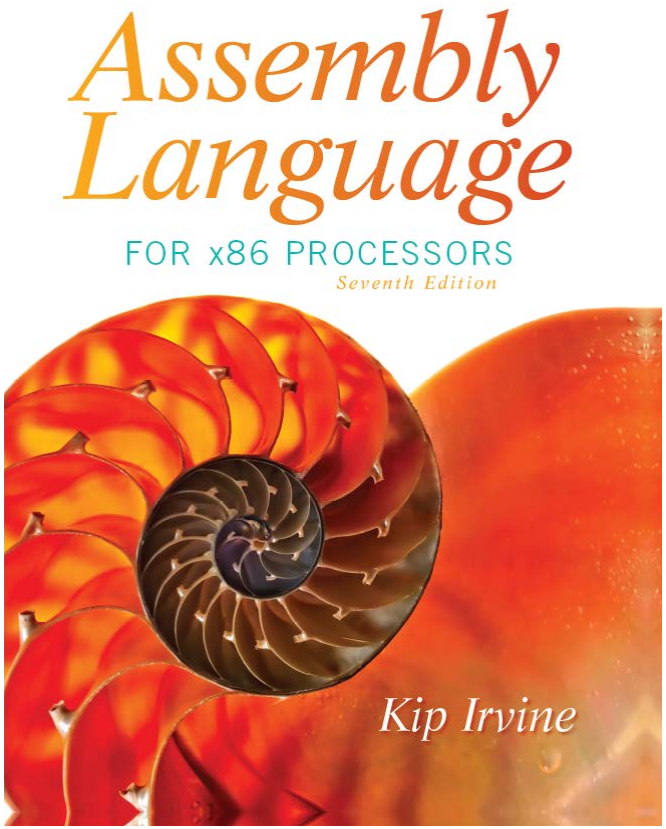
Assembly Language for x86 Processors

Seventh Edition



Chapter 5

Procedures

Chapter Overview

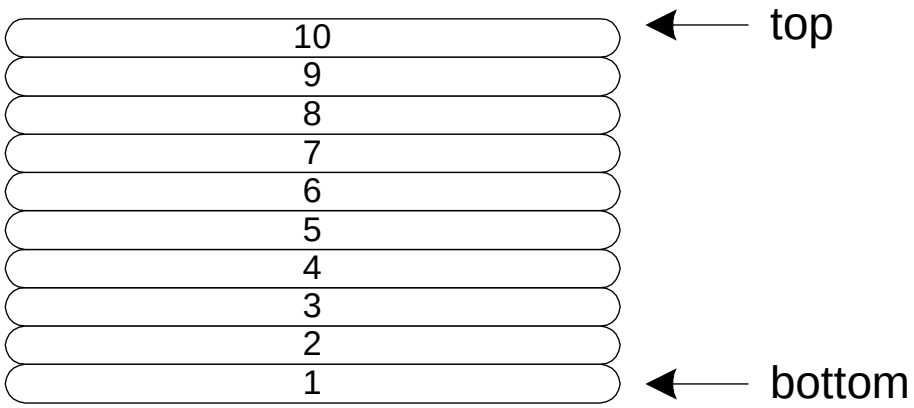
- Stack Operations
- Defining and Using Procedures

Stack Operations

- Runtime Stack
- PUSH Operation
- POP Operation
- PUSH and POP Instructions
- Using PUSH and POP
- Related Instructions

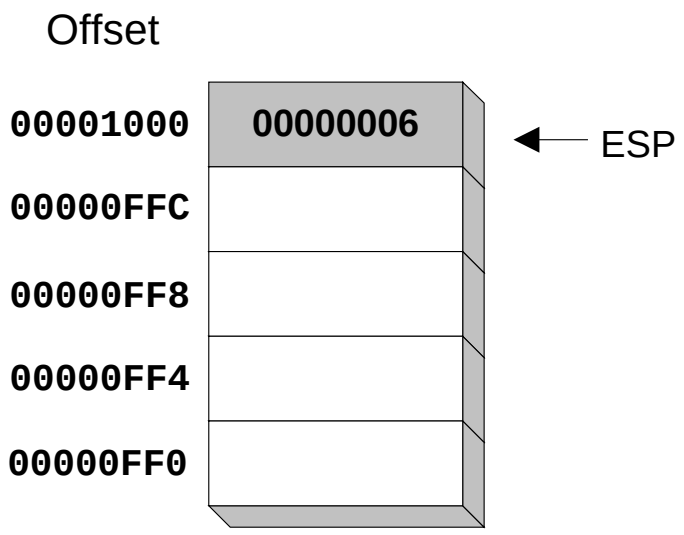
Runtime Stack (1 of 2)

- Imagine a stack of plates . . .
  - plates are only added to the top
  - plates are only removed from the top
  - LIFO structure



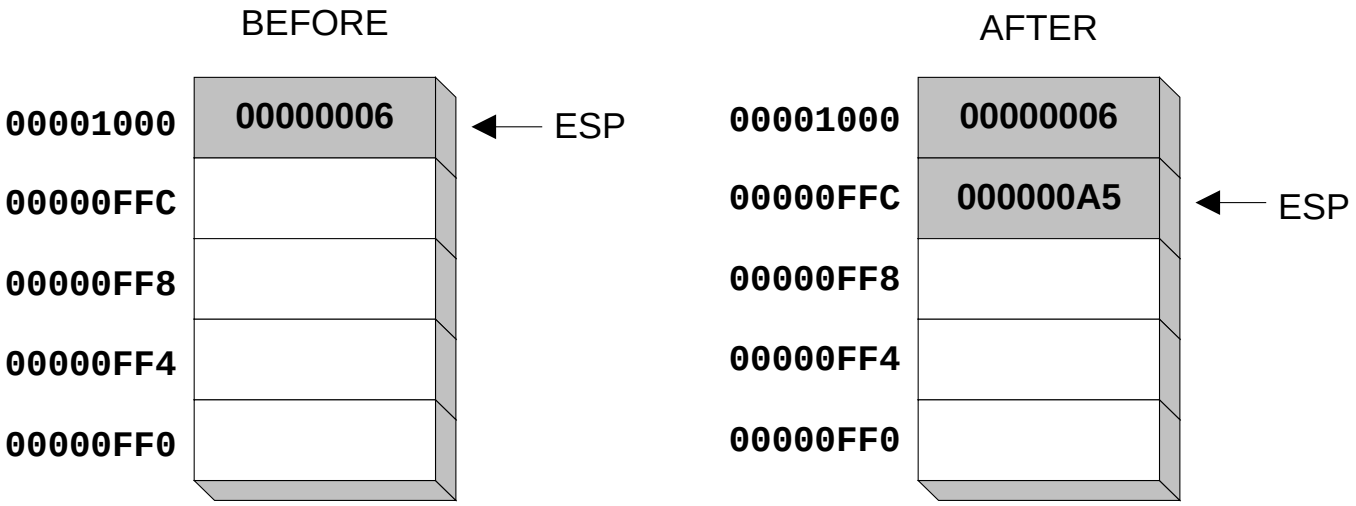
Runtime Stack (2 of 2)

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



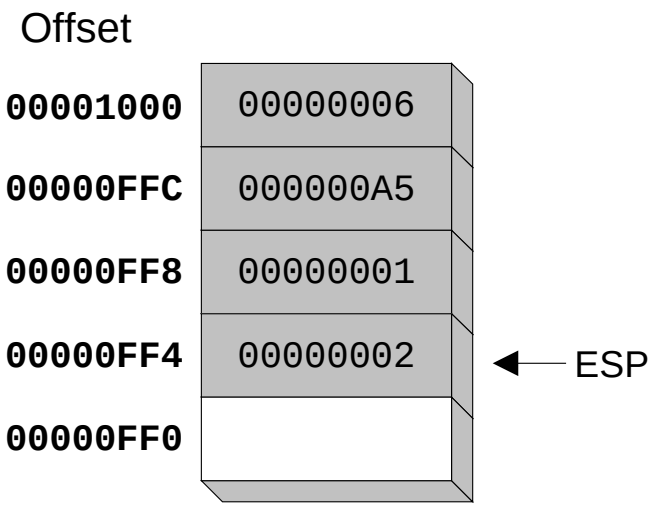
PUSH Operation (1 of 2)

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



PUSH Operation (2 of 2)

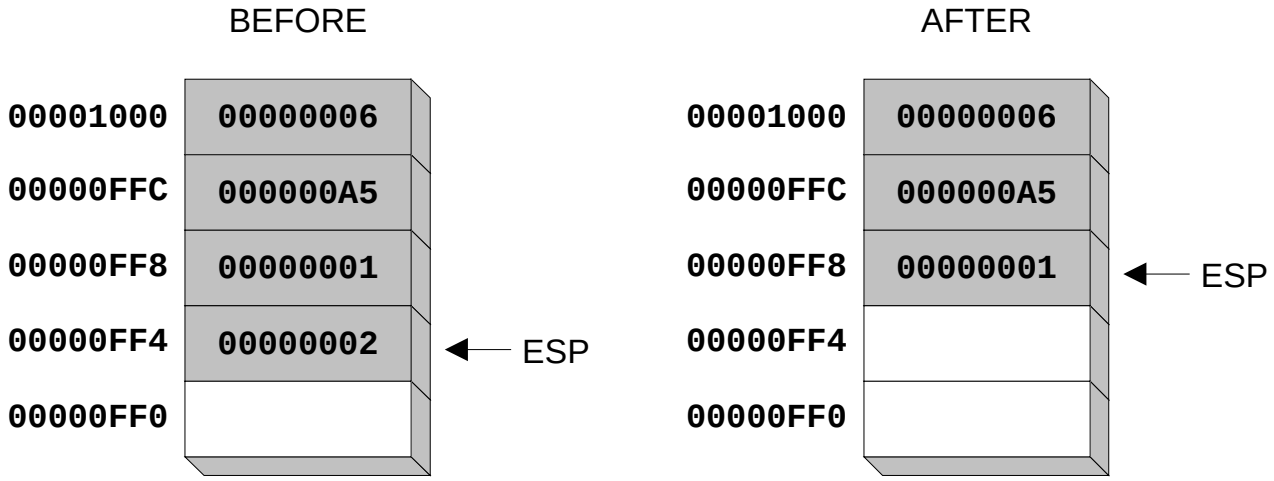
- Same stack after pushing two more integers:



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



PUSH and POP Instructions

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

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
```

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
```

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
```

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

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

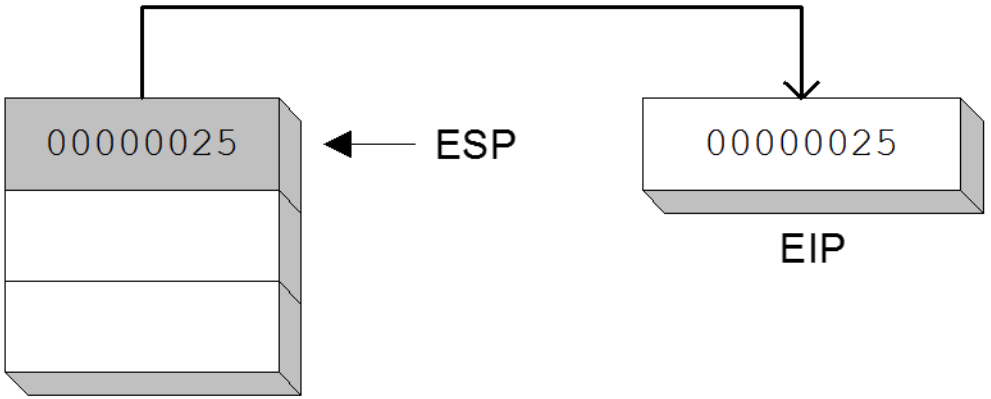
```
main PROC
    00000020 call MySub
    00000025 mov eax,ebx
    .
    .
main ENDP

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

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)

Nested Procedure Calls

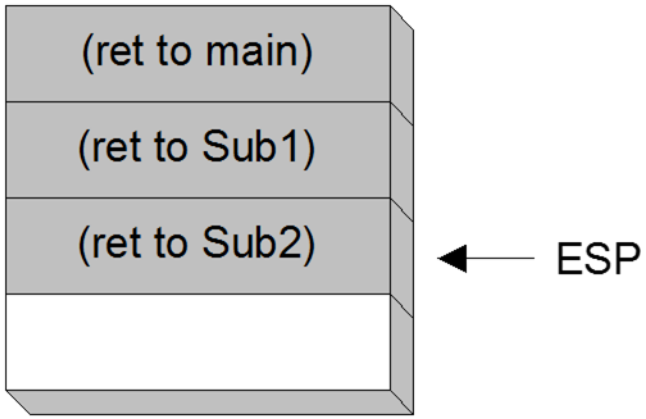
```
main PROC
    .
    .
    call Sub1
    exit
main ENDP

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
```

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

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
```

Procedure Parameters (3 of 3)

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
```

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
```