Lecture 5: Procedures

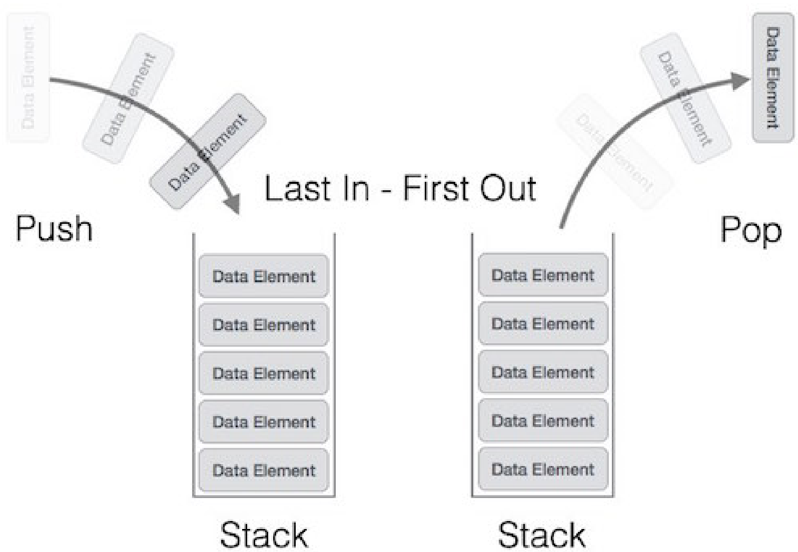
Table of contents

1 Objectives:

- Understand the run-time stack operations.
- Learn how to define a procedure and how to call it.

2 Stack Operations

- A stack is a data structure that is often called **LIFO** structure (Last-In, First-Out) because the last value put into the stack is always the first value taken out.
- A stack is associated with two operations: **Push** and **Pop**.
  - Push is an operation to add an element into the stack (at the top)
  - Pop is an operation to remove one element from the stack (from the top)



2.1 Run-time Stack

- Each process (a running program) has a ***run-time stack***, which is a memory block managed directly by the CPU.
- In 32-bit mode, the ESP register (known as the stack pointer) holds a 32-bit offset into some location on the stack.
- ESP always points to the top of the run-time stack.
- ESP can be manipulated ***manually*** or by CPU instructions: `PUSH`, `POP`, `CALL` and `RET`.

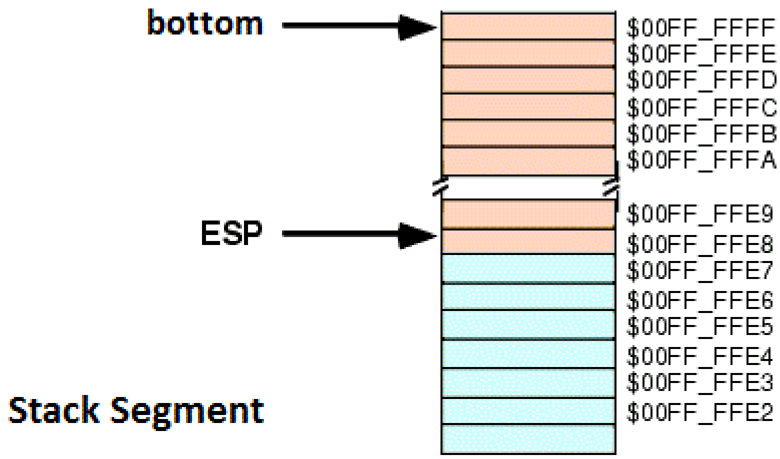


Figure 1: ESP points to the top of the stack. The area above ESP (in light red) is inside the stack. The area below the ESP (light blue) is empty.

2.2 `PUSH` Instruction

Syntax

```
PUSH    <source>
```

- The source operand can be either 16-bit value or 32-bit value.
- The push operation performs the following two steps (in order):

1. It decrements the stack pointer (ESP) by the appropriate amount according to the size of the source operand:
   - if the source operand is 32-bit, the stack pointer is decremented by a value of 4
   - if the source operand is 16-bit, the stack pointer is decremented by a value of 2.
2. It copies a value into the location in the stack referenced by the stack pointer.

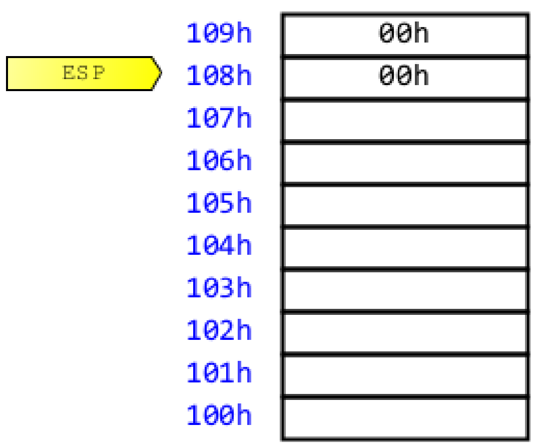
- The valid formats are:

```
push    reg/mem32
push    reg/mem16
push    imm32
```

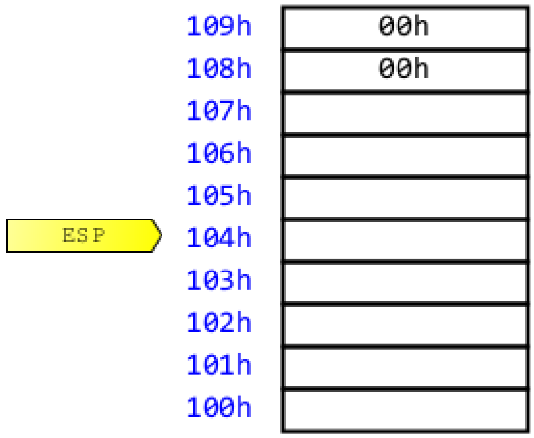
- Example:

```
push    2024    ; 2024 => 000007E8h
```

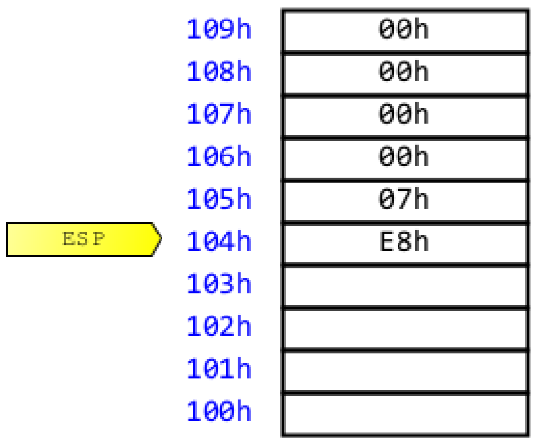
1. Let us consider the following run-time stack as shown below; before executing `push` instruction:



2. When `push 2024` is executed, the first step is to decrement ESP by 4.



3. Then, the value `2024` , which will be treated as `imm32`, will be copied onto the stack at the location referenced by `ESP`:



4. The above diagram shows the stack after pushing a total of four bytes.

> **Note**
>
> The area of the stack below ESP (at lower addresses) is logically empty, and will be overwritten the next time the current program executes any instruction that pushes a value on the stack.

2.3 POP Instruction

Syntax

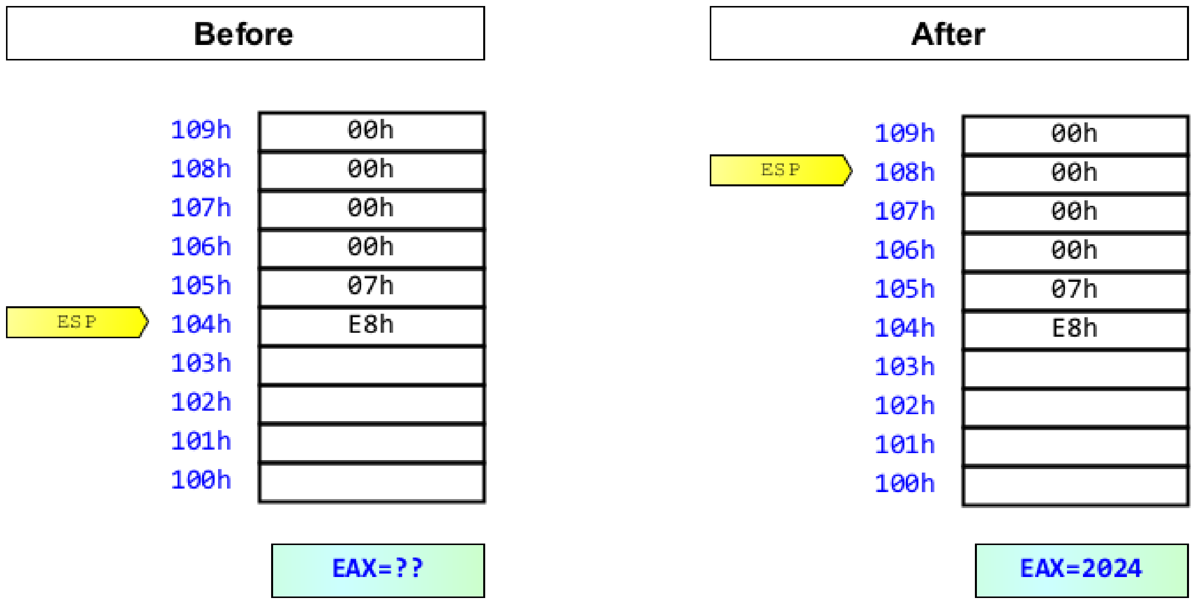
```
POP   <destination>
```

- The destination operand can be of size either 16-bit or 32-bit.
- The valid formats are:

```
pop   reg/mem16
pop   reg/mem32
```

- The pop operation performs the following steps (in order):
  1. It copies the value in the stack referenced by the stack pointer into the destination operand. The number of bytes to be copied is determined by the size of the destination operand.
     – If the size of destination operand is 16-bit, then two bytes are tranferred into the destination operand.
     – If the size of destination operand is 32-bit, then four bytes are transferred into the destination operand
  2. It increments the stack pointer by the appropriate amount according to the size of the destination operand (i.e, either 2 bytes or 4 bytes).
- Example: Let us consider the previous state of the run-time stack. The following instruction pop the top value (which is 2024), and put it into EAX register:

```
pop   eax
```



2.3.1 Flags Affected

Both push and pop instructions have no effect on EFLAGS register.

2.4 PUSHA and POPA Instructions

Syntax

```
PUSHA        ; no operand

POPA         ; no operand
```

- PUSHA (push all) pushes the contents of the 16-bit general-purpose registers onto the stack. The registers are stored on the stack in the following order: AX, CX, DX, BX, SP, BP, SI, and DI.
- POPA (pop all) pops words from the stack into the 16-bit general purpose registers in the following order: DI, SI, BP, BX, DX, CX, and AX. The value on the stack for SP register is ignored.

2.5 PUSHAD and POPAD Instructions

Syntax

```
PUSHAD        ; no operand

POPAD         ; no operand
```

- PUSHAD (push all doublewords) pushes the contents of the 32-bit general-purpose registers onto the stack. The registers are stored on the stack in the following order: EAX, ECX, EDX, EBX, ESP (original value), EBP, ESI, and EDI.
- POPAD (pop all doublewords) pops doublewords from the stack into the 32-bit general purpose registers in the following order: EDI, ESI, EBP, EBX, EDX, ECX, and EAX. The value on the stack for ESP register is ignored.

2.6 PUSHFD and POPFD Instructions

Syntax

```
PUSHAD        ; no operand                POPAD        ; no operand
```

- PUSHFD (push flag doubleword) pushes the contents of the 32-bit EFLAGS register onto the stack.
- POPFD (pop flag doubleword) pops a doubleword from the stack into the 32-bit EFLAGS register.

2.7 Stack Applications

2.7.1 Invoking C Standard Library

- We can invoke C standard functions from inside assembly program, such as `printf()`.
- The instruction `CALL` is used to invoke a function. The following program prints out "Welcome to Assembly Language" to screen monitor (output console).

```
        global _main
        extern _printf      ; tell assembler that printf() is an
                            ; external function

        section .data
message db      "Welcome to Assembly Language", 0

        section .code
_main:
        push    message     ; push message as argument
        call    _printf     ; call printf()
        add     esp, 4      ; to pop out message

        ret
```

2.7.2 Reverse a String

- Let's look at a program that loops through a string and pushes each character on the stack. It then pops the letters from the stack and stores them back into the same string variable. Because the stack is a LIFO (last-in, first-out) structure, the letters in the string are reversed:

```
                global   _main
                extern   _printf        ; C standard function

                section .data
msg             db       "Hello World!", 0

                section .text

_main:
                mov      esi, 0
L1:             movzx    ax, BYTE [esi + msg]
                cmp      ax, 0
                jz       ENDL1
                push     ax
                inc      esi
                jmp      L1
ENDL1:
                mov      ecx, esi
                mov      esi, 0
L2:             pop      ax
                mov      [esi + msg], al
                inc      esi
                loop     L2

                push     msg
                call     _printf
                add      esp, 4

                xor      eax, eax
                ret
```

2.7.3 Memory to Memory Transfer

- Let translate the following statement into assembly language using run-time stack:

```
y = x;        /* assign x to y */
```

```
        push   DWORD[x]
        pop    DWORD[y]
```

- Let us also swap two values x and y using run-time stack:

```
        push   DWORD[x]
        push   DWORD[y]
        pop    DWORD[x]
        pop    DWORD[y]
```

2.7.4 Nested Loop

- Recall that you need to maintain the counter value(s) of the outer loop(s)

```
        mov    ecx, 100      ; outer loop index
OUTER:
        push   ecx           ; store outer loop index
        mov    ecx, 20
INNER:
        .
        .
        loop   INNER

        pop    ecx           ; restore outer loop index
        loop   OUTER
```

3 Procedures

- A complicated problem is usually divided into separate tasks before it can be understood, implemented, and tested effectively.
- This is known as ***Procedure-Oriented Programming (POP)***.

- In assembly language, we typically use the term procedure to mean the assembly language implementation of POP paradigm.
- In other programming languages, procedures are often called **functions** or **subroutines**.
- In NASM, a procedure can be placed anywhere in TEXT segment; before `_main` label, after `ret` instruction, or between `_main` and `ret`.
- NASM has no built-in directive to support procedures. However, MASM, the other assembler, has built-in directive: `PROC` and `ENDP`.

3.1 `CALL` and `RET` Instructions

Syntax

```
        CALL    <destination>
```

- We know that the CPU fetches the instruction in memory pointed by EIP register, also known as **Program Counter (PC)**.
- After fetching the instruction, the PC is automatically incremented.
- When the instruction is executed, the PC will be already pointing to the next instruction.
- Therefore, the `CALL` instruction performs the following operations:
  - it pushes the current value of PC onto the run-time stack. Then,
  - It jumps to the destination address.
- The `RET` instruction simply pops from stack into the PC. Therefore, the EIP register will execute the instruction that follows the `CALL` instruction.

Example 3.1. In the following example, the procedure `initialize` sets EAX, EBX, ECX and EDX to zero. In the main program, we call this procedure to initialize data registers to zero.

```
initialize:
      xor   eax, eax
      xor   ebx, ebx
      xor   ecx, ecx
      xor   edx, edx
      ret                  ; end of initialize procedure

_main:
      call  initialize
      .
      .
      .
      ret                  ; end of main program
```

3.1.1 How the CPU Executes `CALL` and `RET` Instructions

- The following diagrams show a program in execution along with its run-time stack. All addresses are shown in decimal format (not hexadecimal).
- In Figure 2, the program is fetching $instruction_1$, which is located at address 4000.
- When the CPU starts to execute $instruction_1$, the EIP is pointing to $instruction_2$, as shown in Figure 3. After instruction execution, the CPU will fetch $instruction_2$.
- When the CPU starts to execute $instruction_2$, the EIP is incremented and pointing to the instruction at location 4008, as shown in Figure 4.
- After executing $instruction_2$, the CPU will fetch `CALL foo` instruction. Before executing this instruction, the EIP is incremented to 4010, as shown in Figure 5.
- The `CALL` instruction, when executed, will perform the following two operations:
  - first, It pushes the current value of EIP onto the stack.

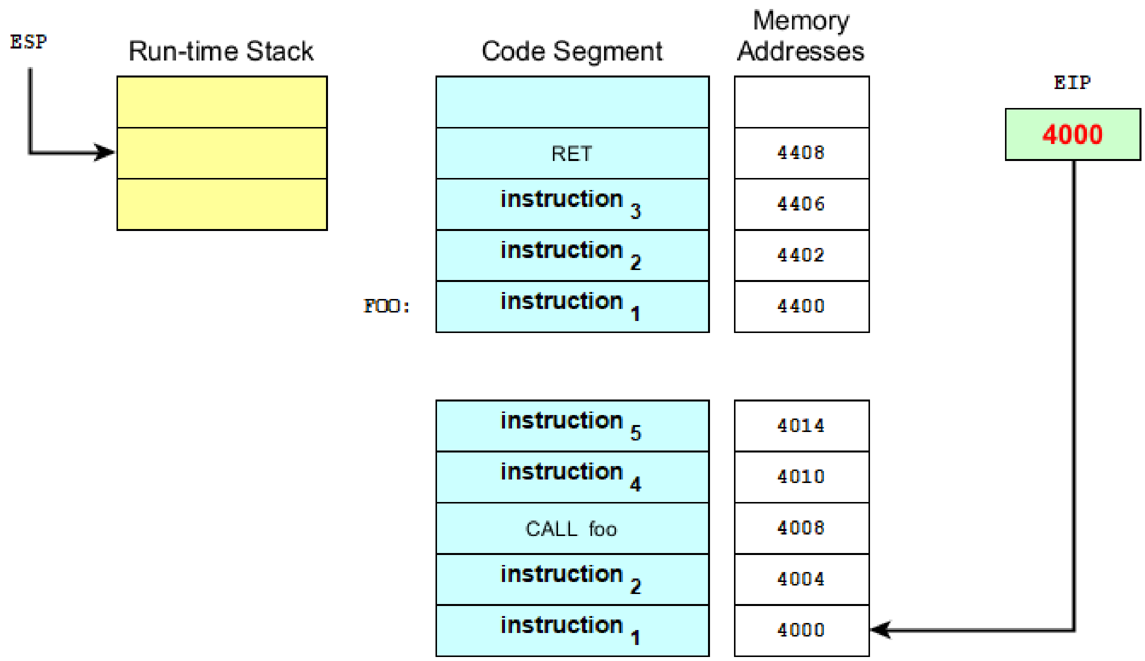


Figure 2

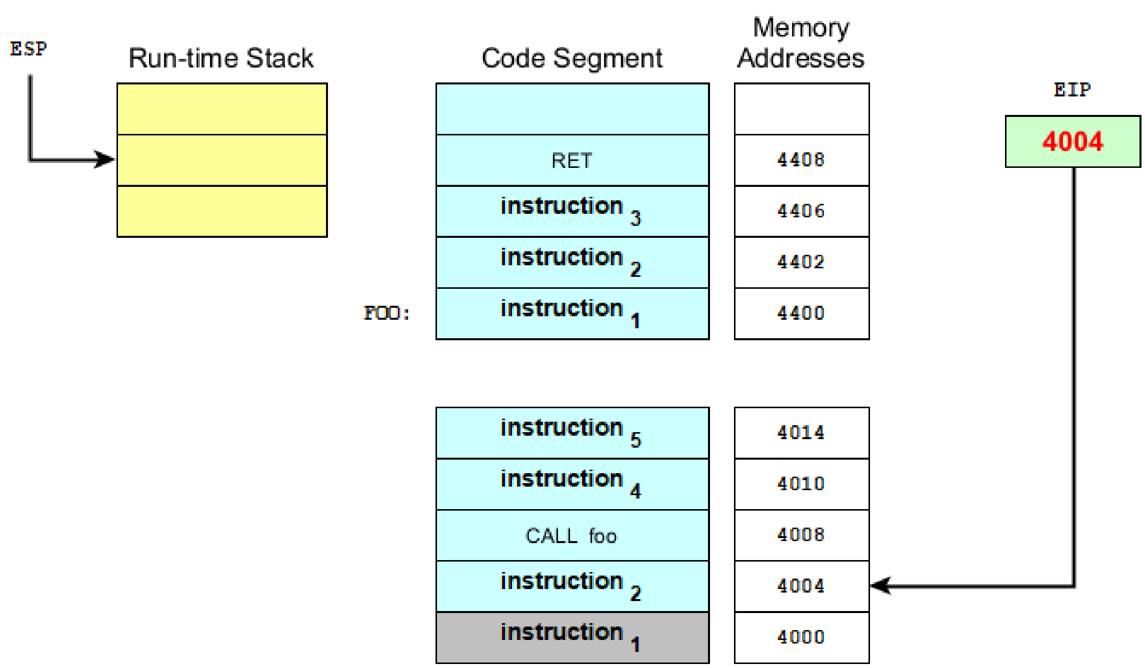


Figure 3

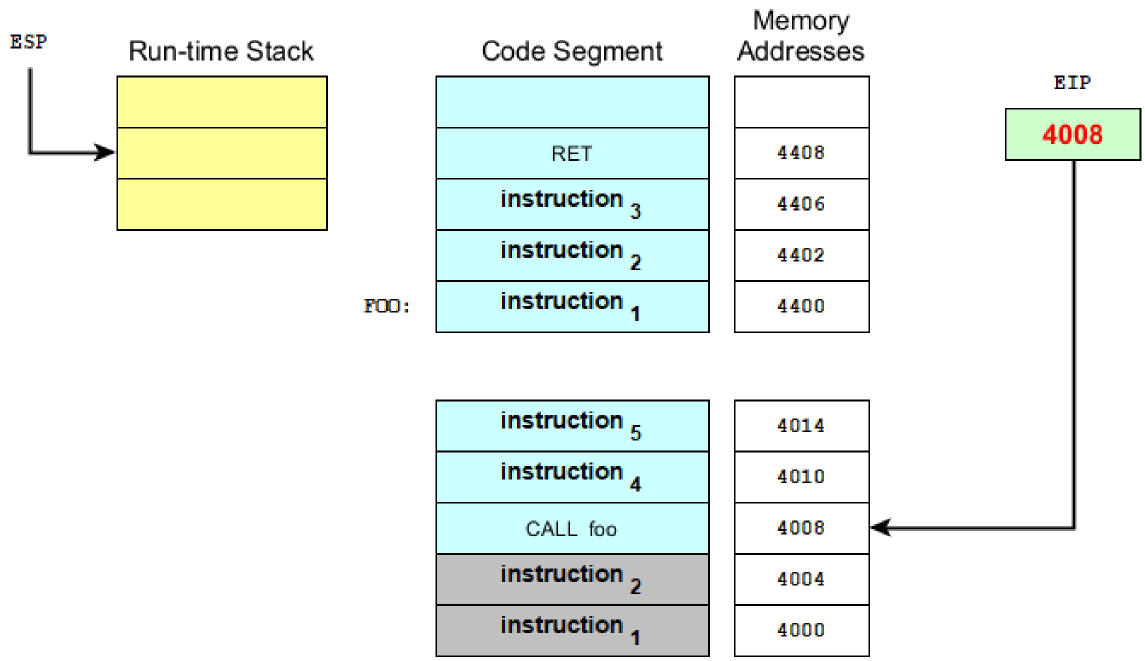


Figure 4

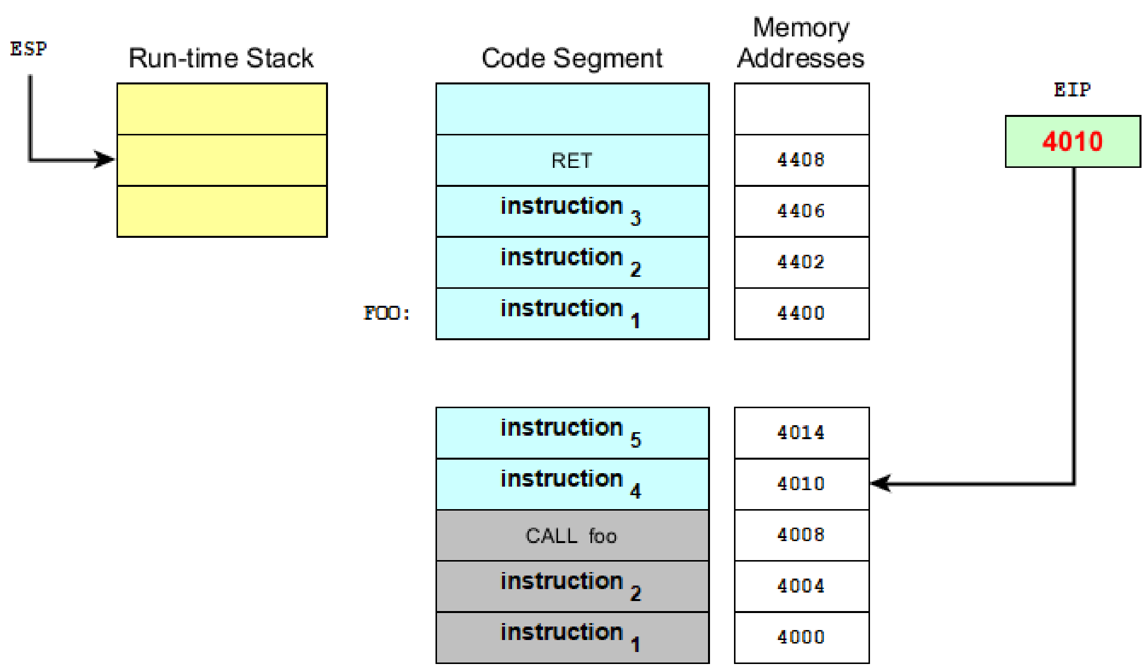


Figure 5

– second, It replaces the value of EIP with the offset address FOO (which has the value 4400). So, the CPU will branch to that address, as illustrated in Figure 6.

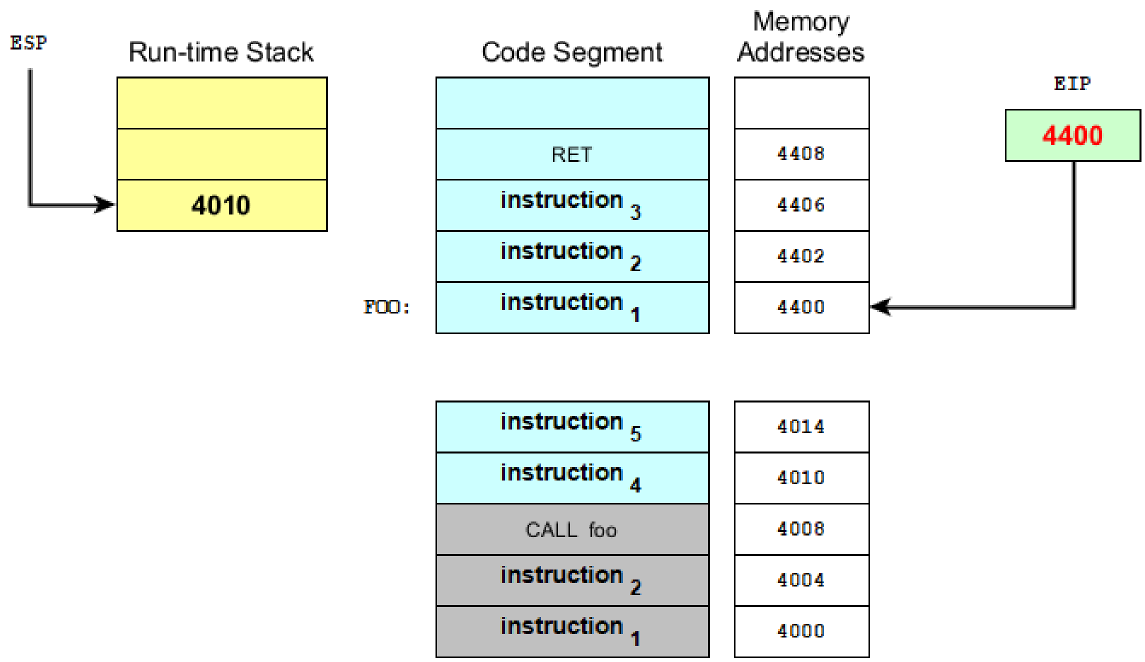


Figure 6

- Logically, the CPU will start to execute the procedure FOO.
- The CPU will execute $instruction_1$, $instruction_2$, and $instruction_3$ of procedure FOO, which are located at 4400, 4402, and 4406, respectively.
- After executing $instruction_3$ of procedure FOO, the EIP is pointing to the next instruction, which is RET. See Figure 7.
- When the CPU executes RET instruction, it will implicitly perform the following instruction: pop EIP. This implies that the value in EIP will be replaced with the value pointed by ESP. Then, the ESP will be incremented by 4. as shown in Figure 8. Consequently, the CPU will branch back to the instruction that follows the CALL instruction.

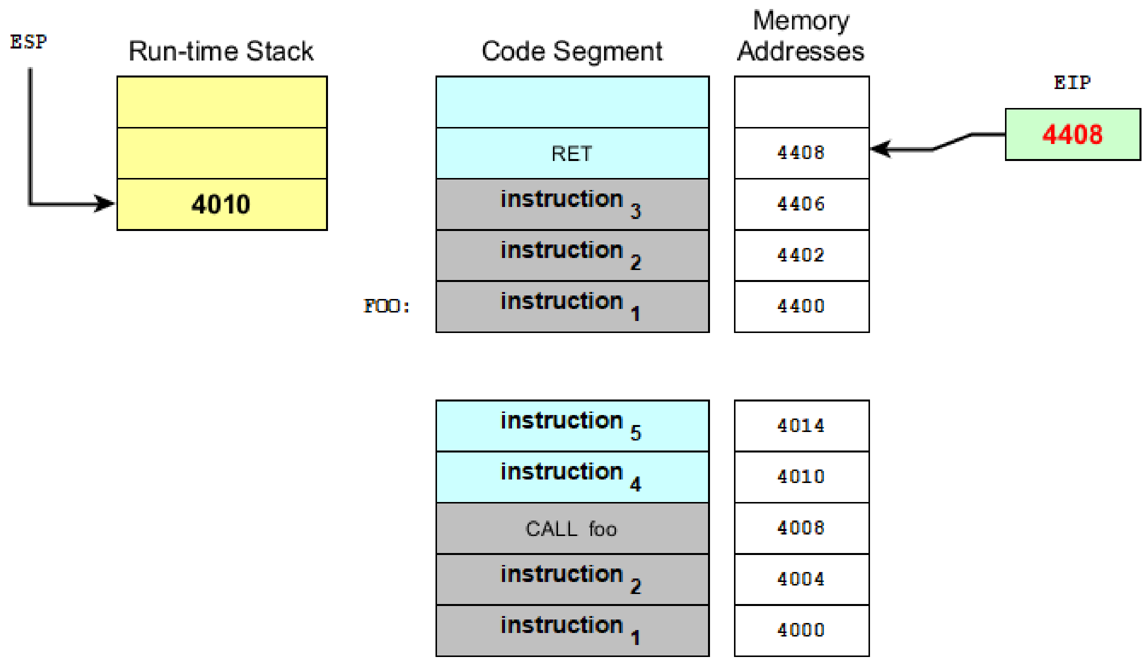


Figure 7

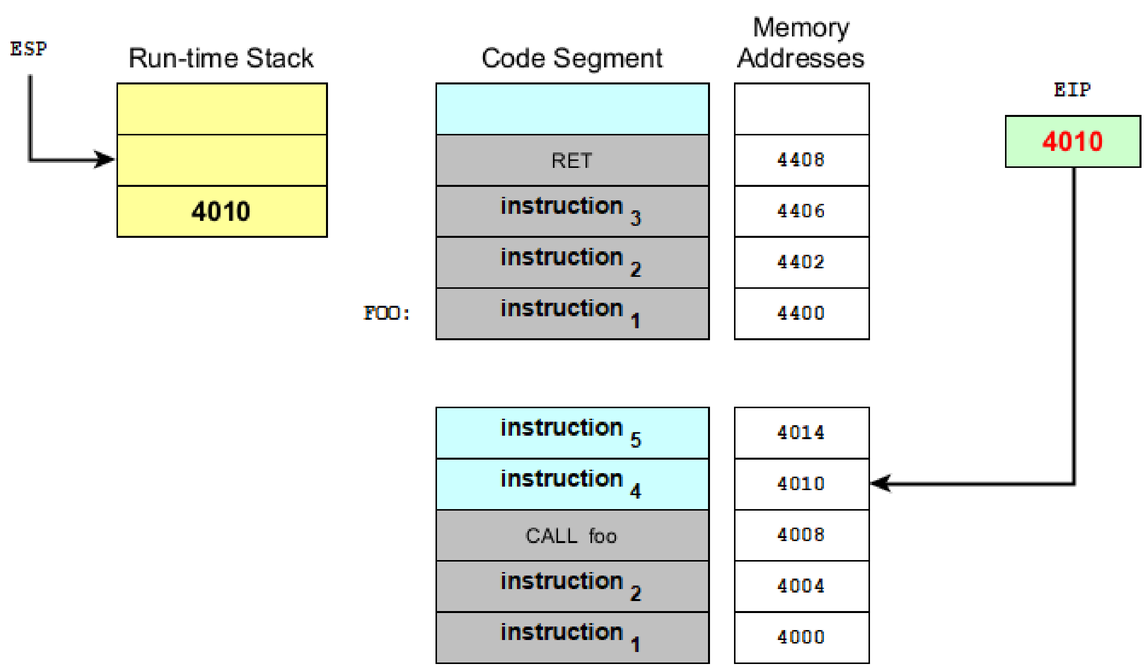


Figure 8

Stack Buffer Overflow

The EIP register cannot be accessed directly by software; it is controlled implicitly by control-transfer instructions (such as JMP, Jcc, CALL, and RET), interrupts, and exceptions. The only way to read the EIP register is to execute a CALL instruction and then read the value of the return instruction pointer from the procedure stack. The EIP register can be loaded indirectly by modifying the value of a return instruction pointer on the procedure stack and executing a return instruction (RET or IRET).
A stack buffer overflow can be caused deliberately as part of an attack known as **stack smashing**. If the run-time stack is filled with data supplied from an *untrusted user* then that user can corrupt the stack in such a way as to **inject executable code** into the running program and take control of the process. This is one of the oldest and more reliable methods for attackers to gain unauthorized access to a computer.

3.2 Nested Procedure Call

- A called procedure can call another procedure before the first procedure returns. This is known as *nested procedure call*.
- Suppose the `main` procedure calls a procedure named `Sub1`. While `Sub1` is executing, it calls the `Sub2` procedure. While `Sub2` is executing, it calls the `Sub3` procedure. The process is shown in Figure 9.

3.3 Passing Register Arguments to Procedures

- Suppose we have two vectors: `vec1` and `vec2`. The size of the two vectores are defined in a constant label named `VECSIZE`.
- The procedure `addVectors` adds two vectors (i.e., `vec1 + vec2`) and stores the resulted vector in `vec1`. Here is the code:

```
addVectors:
      mov   esi, 0
      mov   ecx, VECSIZE
```

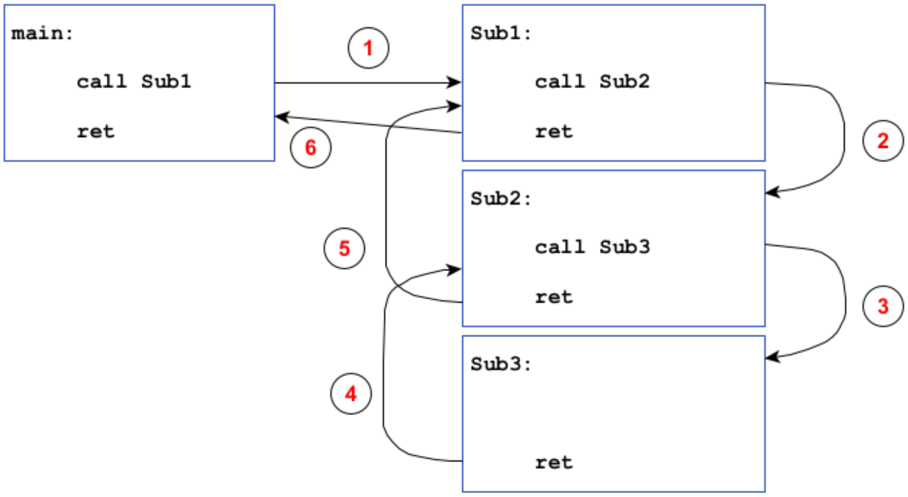


Figure 9

```
top:
      mov   eax, [esi*4 + vec1]
      add   eax, [esi*4 + vec2]
      mov   [esi*4 + vec1], eax
      inc   esi
      loop  top
```

- If you call the procedure, it could only add two specific vectors (`vec1` and `vec2` ONLY). If we have another two vectors, then you need to write another procedure.
- It's not a good idea to include references to specific variable names inside the procedure.
- A better approach is to pass the offset of an array to the procedure and pass an integer specifying the number of array elements. Here is a better solution:

```
addVectors:
      push    edi          ; EDI is pointing to the destination vecto
      push    esi          ; EDI is pointing to the source vector
      push    ecx          ; ECX has the number of elements in the tu
top:
      mov     eax, [edi]
```

```
        add     eax, [esi]
        mov     [edi], eax
        add     esi, 4          ; goto next element
        add     edi, 4          ; goto next element
        loop    top

        pop     ecx             ; restore ECX
        pop     esi             ; restore ESI
        pop     edi             ; restore EDI
        ret
```

- The registers ESI, EDI and ECX are known as ***arguments***.
- In the main program, you should call the procedure as follows:

```
        mov    edi, vec1      ; destination vector (first argument)
        mov    esi, vec2      ; source vector (second argument)
        mov    ecx, VECSIZE   ; Array size (third argument)
        call   addVector
```

- In assembly language, it is common to pass arguments inside general–purpose registers.

3.3.1 Saving and Restoring Registers

- In the addVector example, ECX, EDI and ESI were pushed on the stack at the beginning of the procedure and popped at the end.
- You should save and restore registers modified by a procedure so that the calling program can be sure that none of its own register values have been overwritten.
- The exception to this rule pertains to registers used as return values, usually EAX. Do not push and pop them.

3.4 Documenting Procedures

- It is a good practice to document each procedure you have created. The document should contain at least:

    1. A description of all tasks accomplished by the procedure.
    2. **Receives**: A list of parameters; state their usage and requirements.
    3. **Returns**: A description of values returned by the procedure.
    4. **Requires**: Optional list of requirements called preconditions that must be satisfied before the procedure is called.

- Here is an example:

```
;-----------------------------------------------------------------
; 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 (Cary, Overflow,
;            etc) are changed.
; Requires : nothing
;-----------------------------------------------------------------

SumOf:                  ; beginning of procedure
        add eax, ebx
        add eax, ecx
        ret             ; end of procedure
```

Exercise

Write an assembly program to add two vectors `A` and `B` to obtain vector `C`. The program starts by getting the values of `A` and `B` from a user, then it prints out the vector `C`.

Solution:

The general algorithm for this problem is as follows:

1. Read vector A

2. Read vector B
3. Computer C such that C[i] = A[i] + B[i]
4. Print vector C

Thus, we divide our problem into three sub-problems:

1. Reading a vector from a console
2. Adding two vectors
3. Printing out a vector to the console

Hence, the structure of our program is shown in Figure 10.

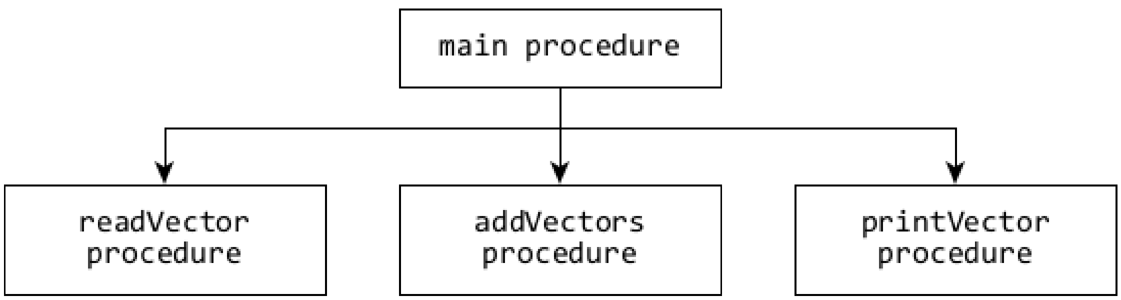


Figure 10

In this exercise, we are going to write a separate file for each procedure:

1. `readvector.asm`, which contains `readVector` procedure.
2. `addvectors.asm`, which contains addVectors procedure.
3. `printvector.asm`, which contains printVector procedure.
4. `main.asm`, which contains the main procedure.

```
;----------------------------------------------------
; File: readvector.asm
;----------------------------------------------------
            global  readVector
            extern  _scanf
            extern  _printf

            section .data
scan_fmt    db  "%d", 0

            section .code
;----------------------------------------------------
; Read a vector from a console user
;
; Receives:
;   EAX:    Input prompt
;   EDI:    A pointer to the vector
;   ECX:    Number of elements in the vector
;
; Returns:
;   A vector pointed by EDI register
;
; Requires: Nothing
;-----------------------------------------------------
readVector:
            ; store EDI and ECX
            push    edi
            push    ecx

            ; print input prompt
            push    eax
            call    _printf
            pop     eax

            ; restore EDI and ECX
            pop     ecx
            pop     edi
top:
            push    ecx         ; store it again
            push    edi
            push    scan_fmt
            call    _scanf
```

```
;------------------------------------------------
; File: addvectors.asm
;------------------------------------------------
            global addVectors

            section .code

;------------------------------------------------
; Add two vectors
;
; Receives:
;   EAX:    A pointer to the first vector
;   EBX:    A pointer to the second vector
;   EDI:    A pointer to the resulted vector
;   ECX:    Number of elements in the vector
;
; Returns:
;   The new vector pointed by EDI
;
; Requires: Nothing
;-------------------------------------------------
addVectors:
            ; save all data registers
            pushad
top:
            mov     edx, [eax]
            add     edx, [ebx]
            mov     [edi], edx

            add     eax, 4
            add     ebx, 4
            add     edi, 4
            loop    top

            ; restore all data registers
            popad

            ret
```

```
;-------------------------------------------------
; File: printvector.asm
;-------------------------------------------------
            global  printVector

            extern  _printf

            section .data
printf_fmt  db  "%d", 13, 10, 0

            section .code
;-------------------------------------------------
; Print out vector to the console screen
;
; Receives:
;   EAX:    Output prompt
;   EDI:    A pointer to the vector
;   ECX:    Number of elements in the vector
;
; Returns: Nothing
;
; Requires: Nothing
;--------------------------------------------------
printVector:
            ; store EDI and ECX
            push    edi
            push    ecx

            push    eax
            call    _printf
            pop     eax

            ; restore EDI and ECX
            pop     ecx
            pop     edi
top:
            push    ecx
            push    edi

            push    DWORD [edi]
            push    printf_fmt
            call    _printf
```

```
;----------------------------------------------------
; File: main.asm
;----------------------------------------------------
            global _main
            extern readVector
            extern printVector
            extern addVectors

            section .data
prompt1     db      "Enter vector A:", 13, 10, 0
prompt2     db      "Enter vector B:", 13, 10, 0
output_msg  db      "Vector C:", 13, 10, 0

VECSIZE     equ     5    ; number of elements in the vector

            section .bss
vecA        resd    VECSIZE
vecB        resd    VECSIZE
vecC        resd    VECSIZE

            section .code
;----------------------------------------------------
; The main procedure
;----------------------------------------------------
_main:
            mov     eax, prompt1
            mov     edi, vecA
            mov     ecx, VECSIZE
            call    readVector

            mov     eax, prompt2
            mov     edi, vecB
            mov     ecx, VECSIZE
            call    readVector

            mov     eax, vecA
            mov     ebx, vecB
            mov     edi, vecC
            mov     ecx, VECSIZE
            call    addVectors

            mov     eax, output_msg
```