Subprogram 101

ICS312

Machine-Level and Systems Programming

Henri Casanova (henric@hawaii.edu)

Subprograms

- Subprograms (functions, procedures, methods) are key to making programs easier to read and write (code reuse)
  - They are used all the time in high-level code, and in fact it's bad practice to write many lines of code in sequence instead of resorting to subprogram calls
  - Some companies require that subprograms be shorter than some fixed number of line of codes
- We are going to see how to define and call subprograms in assembly
  - Useful to write large(r) assembly programs
  - **More importantly,** will allow us to understand how subprograms work in higher-level languages
- But first, let's just review the concept of indirection

Indirect Addressing

- So far we have seen one way to address the content of memory
  - Define a symbol, i.e., an address, in the .bss or .data segment

```
L       dd      FA123BDEh
```

  - Use that symbol as an address so that we can access content

```
mov     eax, [L]
```

- L is an address in memory, and [L] is the content of whatever is stored at that address
  - The mov instruction above knows that eax is 32-bit, so it will read 32 bits starting at address L
- We have also used registers to store addresses

```
mov     eax, L          ; eax stores the address
inc     eax             ; modify the address
mov     bx, [eax]       ; put the 2 bytes starting at
                        ; address eax into bx
```

Indirect Addressing

- Registers can hold "data" or "addresses"
  - Not keeping this straight leads to horrible bugs
  - The processor will happily apply whatever operation on whatever data as long as data sizes are correct
  - e.g., if you think that a register contains an integer, but in fact it stores the address of the integer in memory, then your arithmetic operations on that integer will return very strange results
- Since addresses are 32-bit, only the EAX, EBX, ECX, EDX, ESI, and EDI registers can be used to store addresses in a program
- Storing addresses into a register is what makes it possible to implement our first subprogram

What is a subprogram?

- A subprogram is a piece of code that starts at some address in the text segment
- The program can jump to that address to “call” the subprogram
- When the subprogram is done executing it jumps back to the instruction after the call, and the execution resumes “as if nothing had happened”
- The subprogram can take parameters
- Let’s see how we can implement this using only what we’ve seen so far in the course

Example Subprogram

- Say we want to write a subprogram that computes some numerical function of two operands and “returns” the result
  - e.g., because we need to compute that function often and code duplication is evil
- We will write the program so that when it is called, the first operand is in eax and the second in ebx, and when it returns the result is in eax
  - This is a convention that we make, and that should be documented in the code
- Calling the program can then be done via a simple jmp instruction
- Let’s look at the code

"By hand" subprogram

```
    . . .
    mov  eax, 12      ; first operand = 12
    mov  ebx, 14      ; second operand = 14
    jmp  func; "call" the function
ret:
    . . .
    . . .
func:
    add eax, ebx      ; do something with eax and ebx
                      ; put result in eax
    jmp ret           ; "return" to the instruction
                      ; after the call
```

“By hand” subprogram

```
    . . .
    mov  eax, 12      ; first operand = 12
    mov  ebx, 14      ; second operand = 14
    jmp  func; “call” the function
ret:
    . . .
    . . .
func:
    add eax, ebx      ; do something with eax and ebx
                      ; put result in eax
    jmp ret           ; “return” to the instruction
                      ; after the call
```

Why is this not really a subprogram?

Multiple Calls?

- Typically we want to call a function from multiple places in a program
- The problem with the previous code is that the function always returns to a single label!

```
        . . .
        jmp   func        ; "call" the function
ret1:
        . . .
        jmp   func        ; "call" the function
ret2:
        . . .
func:
        . . .
        jmp   ???         ; where do we return???
```

A Better Function Call

- To fix our previous example, we need to remember the place where the function should return!
- This can be done by storing the address of the instruction after the call in a register, say, register ecx
- The code for the function then can just return to whatever instruction ecx points to
  - Again, this is a convention that we decide as a programmer and that we must remember

A Better Function Call

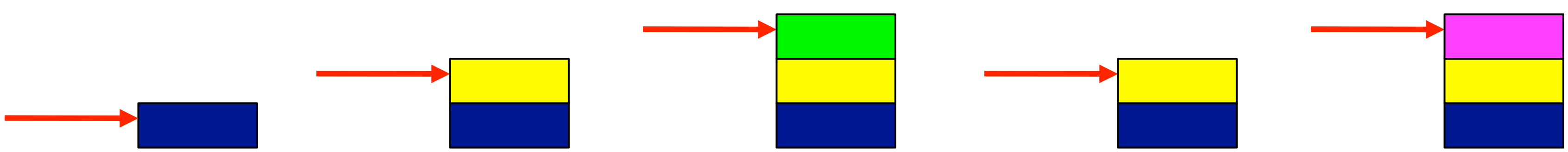
```
        . . .
        mov  ecx, ret1          ; store the return address
        jmp  func               ; “call” the function
    ret1:
        . . .
        mov  ecx, ret2          ; store the return address
        jmp  func               ; “call” the function
    ret2:
        . . .
    func:
        . . .
        jmp  ecx                ; return
```

All Good, but ...

- So at this point, we can do any function call
- We just need to decide on and document a convention about which registers hold the
  - input parameters
  - return value
  - return address
- The problem is that this gets very cumbersome
  - It requires a bunch of “ret” labels all over the code
    - The textbook shows how the return address can be computed numerically, but it is very awkward
  - It forces the programmer to constantly keep track of registers and be careful to save and restore important values
    - We already have few registers
- Solution:
  - A stack
  - Two new instructions: CALL and RET

The Stack

- A stack is a Last-In-First-Out data structure
- Provides two operations
  - Push: puts something on the stack
  - Pop: removes something from the stack
- Defined by the address of the "element" at the top of the stack, which is stored in the so-called "stack pointer"
  - Push: puts the element on top of the stack and update the stack pointer
  - Pop: gets the element from the top of the stack and update the stack pointer
- The processor has "tools" (registers, instructions) to maintain one stack
  - It's called the "runtime stack"



The Runtime Stack

- Our stack will only allow pushing/popping of 4-byte elements
  - Note “quite” true, but a much safer approach/convention
- The stack pointer is always stored in the ESP register
- Initially the stack is empty and the ESP register has some value
- The stack grows downward (i.e., toward lower addresses)
- **Pushing an element:**
  - Decrease ESP by 4 and write 4 bytes at address ESP
  - Examples: push eax push dword 42
- **Popping an element:**
  - Get the value from the top of the stack into a register and increase ESP by 4
  - Examples: pop eax pop ebx
- **Accessing an element:**
  - Read the 4 bytes at address ESP
  - Example: mov eax, [esp]

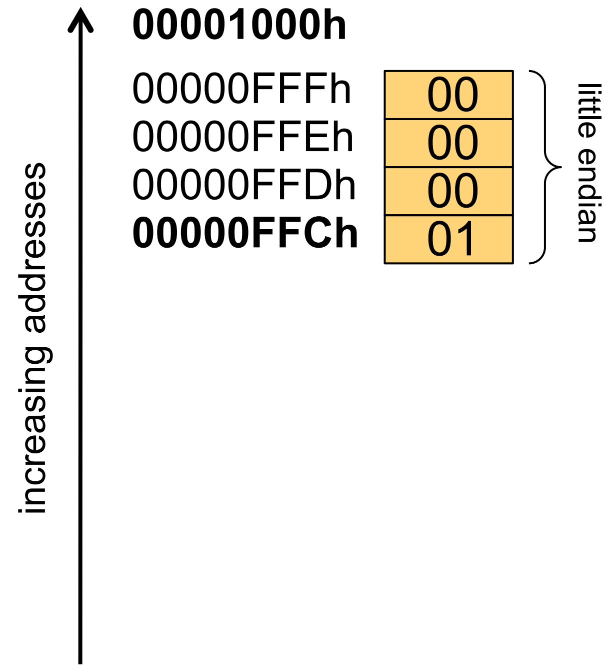
Example Stack Instructions

- Assuming that ESP=00001000h

Example Stack Instructions

- Assuming that ESP=00001000h

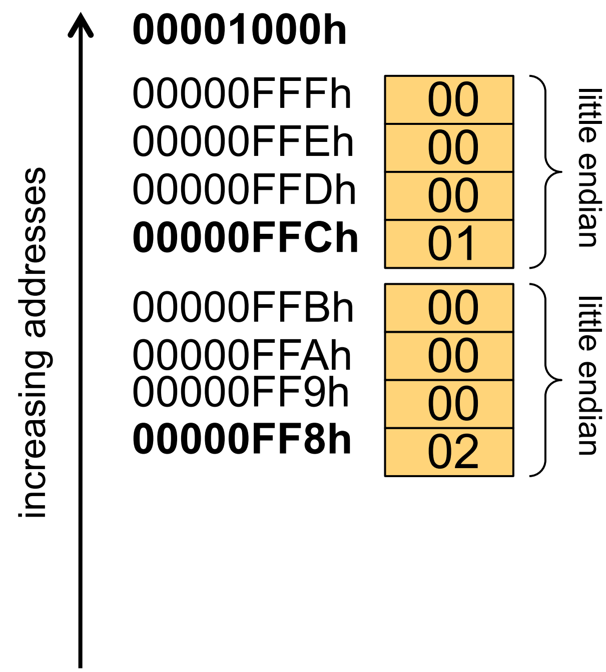
push dword 1 ; ESP = 00000FFCh



Example Stack Instructions

- Assuming that ESP=00001000h

```
push    dword   1        ; ESP = 00000FFCh
push    dword   2        ; ESP = 00000FF8h
```



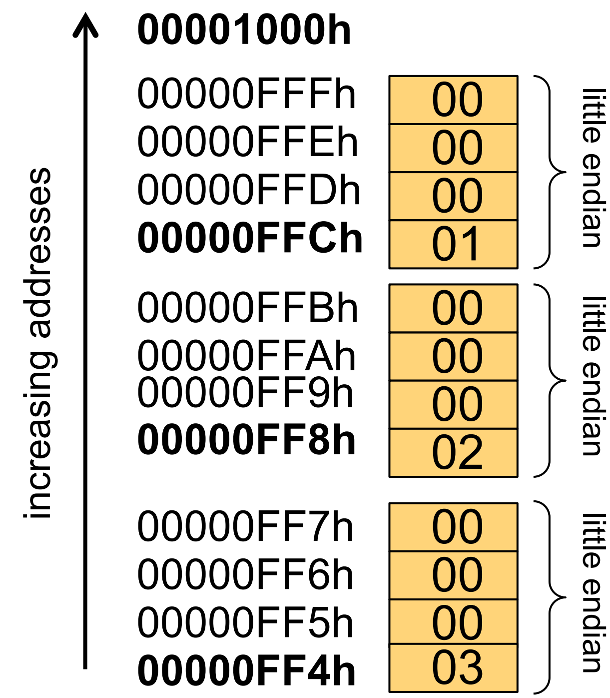
Example Stack Instructions

- Assuming that ESP=00001000h

push dword 1 ; ESP = 00000FFCh

push dword 2 ; ESP = 00000FF8h

push dword 3 ; ESP = 00000FF4h

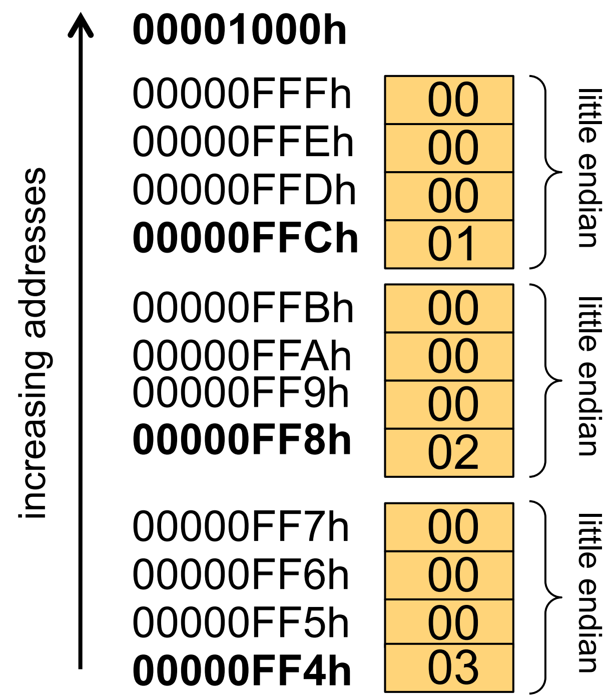


Example Stack Instructions

- Assuming that ESP=00001000h

```
push    dword   1       ; ESP = 00000FFCh
push    dword   2       ; ESP = 00000FF8h
push    dword   3       ; ESP = 00000FF4h

pop     eax             ; EAX = 3
pop     ebx             ; EBX = 2
pop     ecx             ; ECX = 1
```



The ESP Register

- The ESP register always contains the address of the element at the **top** of the stack
  - which is the “bottom” of the figure in the previous slide since the stack grows towards lower addresses
- IMPORTANT: Do not use ESP for anything else!
  - If you “run out” of registers, using ESP to store your data is not a good option at all
- Its value is updated by calls to push and pop
- In a few very specific and well-known cases we’ll update it by hand
  - See this in a few slides

PUSHA and POPA

- For subprograms, a key use of the stack is to save/restore register values
- Say your program uses eax and calls a function written by somebody else
- You have no idea (or don't care to know) whether that function uses eax
  - But if it does, your eax will be corrupted
- One easy solution:
  - push eax onto the stack
  - call the function and let it do its thing until it returns
  - pop eax to restore its value
- The x86 offers two convenient instructions
  - PUSHA: pushes EAX, EBX, ECX, EDX, ESI, EDI, and EBP onto the stack
  - POPA: restores them all and pops the stack
- It's now simple to say "save all my registers" and "restore all my registers"
  - Probably overkill, but safe and easy

Recall the NASM Skeleton

```
    ; include directives

segment .data
    ; DX directives

segment .bss
    ; RESX directives

segment .text
    global asm_main
    asm_main:
        enter   0,0
        pusha
        ; Your program here
        popa
        mov     eax, 0
        leave
        ret
```

Save the registers since they may have been in use by the “driver” program

Restore the registers so that the “driver” program will not be disrupted by the call to function asm_main

The CALL and RET Instructions

- One of the annoying things with our previous subprogram was that we had to manage the return address
  - In our example we stored it into the ECX register
- Two convenient instructions can do this for us
- CALL:
  - Pushes the address of the next instruction on the stack
  - Unconditionally jumps to a label (calling a function)
- RET:
  - Pops the stack and gets the return address
  - Unconditionally jumps to that address (returning from a function)

Without CALL and RET

```
        . . .
        mov  ecx, ret1        ; store the return address
        jmp  func             ; "call" the function
ret1:
        . . .
        mov  ecx, ret2        ; store the return address
        jmp  func             ; "call" the function
ret2:
        . . .
func:
        . . .
        jmp  ecx              ; return
```

With CALL and RET

```
    . . .
    call   func              ; call the function
    . . .

    call   func              ; call the function
    . . .

func:
    . . .
    ret                      ; return
```

With CALL and RET

```
    . . .
    call  func          ; call the function
    . . .

    call  func          ; call the function
    . . .

func:
    . . .
    ret                 ; return
```

Looks almost like high-level code

Recall the NASM Skeleton

```
    ; include directives

segment .data
    ; DX directives

segment .bss
    ; RESX directives

segment .text
        global asm_main
    asm_main:
        enter   0,0
        pusha
        ; Your program here
        popa
        mov     eax, 0
        leave
        ret
```

Returns from function asm_main

Nested Calls

- The use of the stack enables nested calls
  - Return addresses are popped in the reverse order in which they were pushed (Last-In-First-Out)
- Warning: one must be extremely careful to pop everything that's pushed on the stack inside a function
- Example of erroneous use of the stack:

```
func:
        mov   eax, 12        ; eax = 12
        push  eax            ; put eax on the stack
        ret                  ; pop eax and interpret
                             ; it as a return address!!
```

Conclusion

- The next set of lecture notes will talk about everything we can do with the stack
  - Much more than just storing return addresses!