Subprogram 101

ICS312

Machine-Level and Systems Programming

Henri Casanova (henric@hawaii.edu)

Subprograms

- Subprograms (i.e., functions, procedures, methods) make programs modular, promoting code reuse
  - We use them a lot in high-level code
    - From a software engineering perspective, it's bad practice to write long sequences of lines of code
    - So we end up putting a lot of code in subprograms
    - Some companies even require that subprograms be shorter than some fixed number of lines of code
- We are going to see how to define and call subprograms in assembly
  - Useful to write large(r) assembly programs, if you're into that
  - **More importantly,** will allow us to understand how subprograms work in higher-level languages and opens up understanding of all kinds of deeper concepts
- But first, let's just review the concept of indirection

Indirection

- Registers can hold “data” or “addresses”
  - mov eax, L vs. mov eax, [L]
  - 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
- Addresses are 32-bit and can be stored in any 32-bit register
- Multiple indirections is “easy”, but we cannot do [[[eax]]] in NASM assembly and instead need multiple steps:

```
mov ebx, [eax] ; one indirection
mov ebx, [ebx] ; two indirections
mov ebx, [ebx] ; three indirections
```

- Storing addresses into registers 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 and return a value

Subprograms are a lot of Work

- Every time you call a function in any high-level programing language (Java, Python), a lot of things happen
  - Way more instructions than you think
- We are going to built it all from nothing, first using only what we have learned so far this semester
- And we'll "grow" what we have until we have a full-fledged subprogram implementation
  - Which will be very similar to what a C or C++ compiler would generate

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 subprogram 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 subprogram 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
return_here:
   . . .
   . . .
func:
   add eax, ebx      ; eax = eax + ebx
   jmp return_here   ; “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
return_here:
   . . .
   . . .
func:
   add eax, ebx      ; eax = eax + ebx
   jmp return_here   ; “return” to the instruction
                     ; after the call
```

Why is this not really a subprogram?

“By hand” subprogram

```
   . . .
   mov   eax, 12     ; first operand = 12
   mov   ebx, 14     ; second operand = 14
   jmp    func       ; “call” the function
return_here:
   . . .
   . . .
func:
   add eax, ebx
   jmp return_here
```

Why is this not really a subprogram?

Because the subprogram can only be called from one place!!!

Multiple Calls?

- We want to be able 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
return_here_1:
    . . .
    jmp    func   ; "call" the function
return_here_2:
    . . .
func:
    . . .
    jmp    ???    ; where should we return???
```

Multiple Calls?

- To fix our previous example, we need to remember the place where the function should return, i.e., the address of the instruction after the call instruction
- This could be done by storing this address in, say, ecx and then just do **`jump ecx`**
- You can store any address in a register, for instance:

```
        mov eax, here
        jmp eax
        . . .
here:
        . . .
```

- Let's live-code a program that computes the size in bytes of some **`add`** instructions
- Let's look at our function call/code…

A Better Function Call

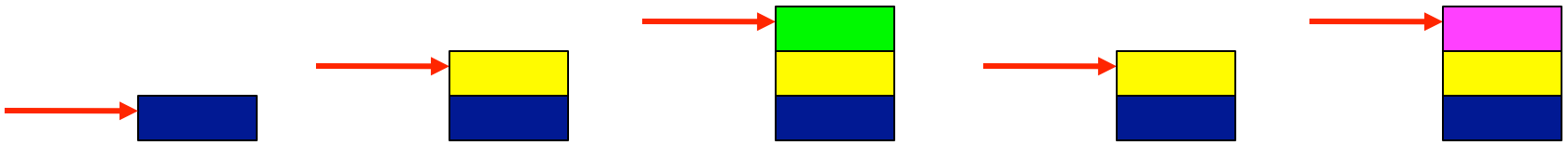
```
. . .
  mov  ecx, return_here_1; store the return address
  jmp   func              ; "call" the function
return_here_1:
. . .
  mov  ecx, return_here_2; store the return address
  jmp   func              ; "call" the function
return_here_2:
. . .
   func:
. . .
  jmp   ecx          ; return
```

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 so few registers to begin with :(
- Solution:
  - A stack and two new instructions: **`call`** and **`ret`**

A Stack

- A stack is a Last-In-First-Out data structure
- It 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 an element on top of the stack and updates the stack pointer
  - Pop: gets the element from the top of the stack and updates the stack pointer
- The processor has “tools” (registers, instructions) to maintain one particular stack, the “runtime stack”

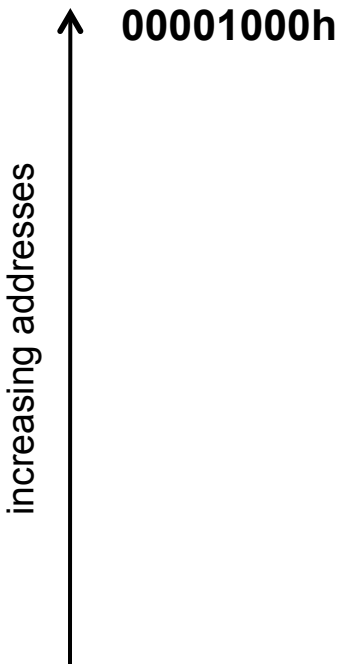


The Runtime Stack

- A stack in RAM that allows pushing/popping of 4-byte elements
  - Not “quite” true, but true in this course
- **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

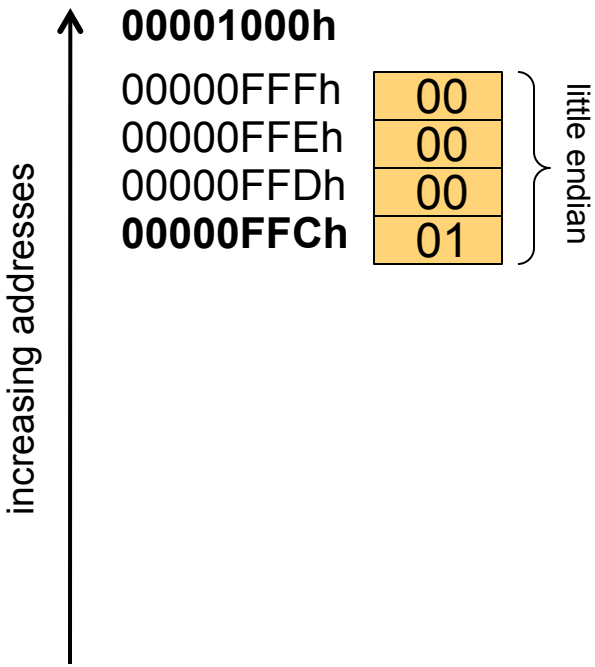
- Assuming that ESP=00001000h



Example Stack Instructions

- Assuming that ESP=00001000h

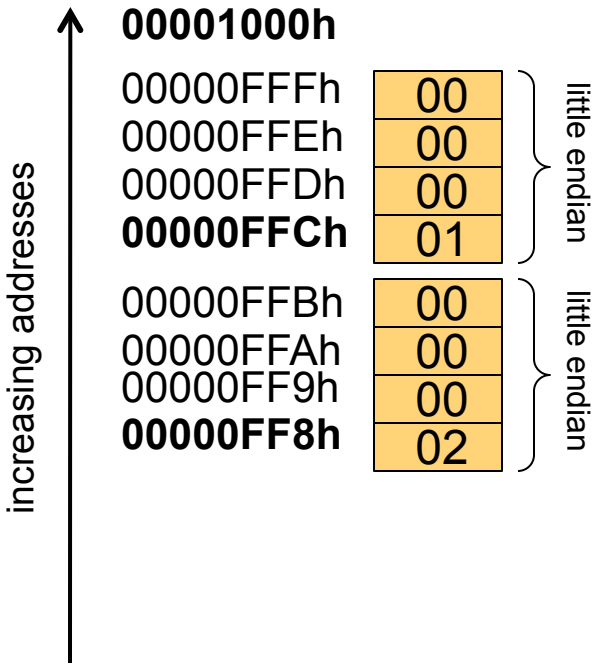
```
push   dword  1        ; ESP = 00000FFCh
```



Example Stack Instructions

- Assuming that ESP=00001000h

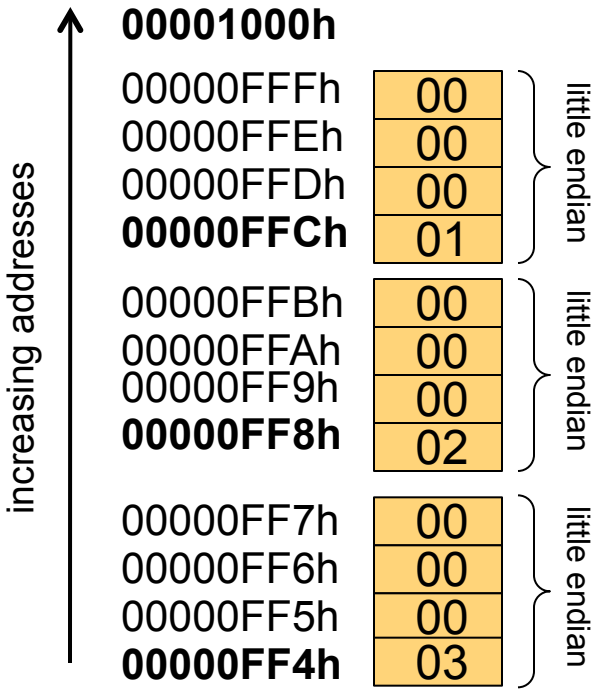
```
push   dword  1        ; ESP = 00000FFCh
push   dword  2        ; ESP = 00000FF8h
```



Example Stack Instructions

- Assuming that ESP=00001000h

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

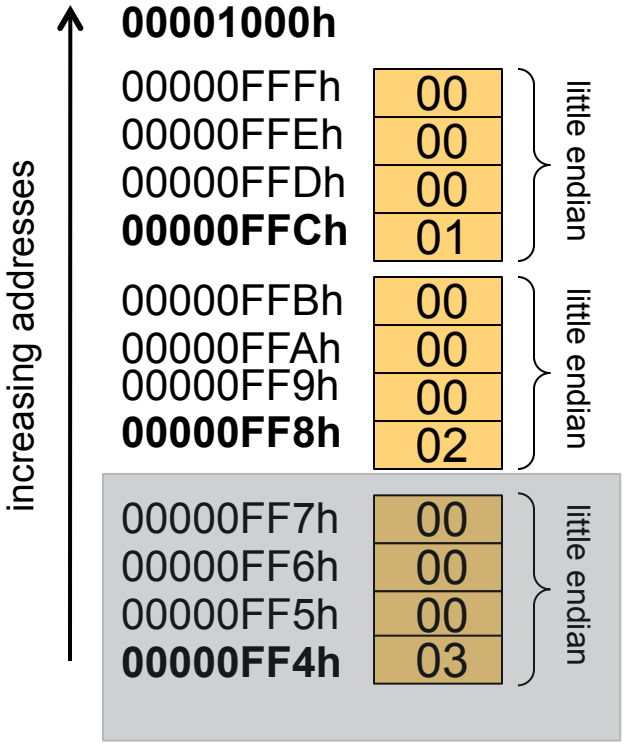


Example Stack Instructions

- Assuming that ESP=00001000h

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

pop    eax     ; EAX = 3, ESP = 00000FF8h
```



Then popped value is not erased from the stack!

It stays there, and will be overwritten whenever there are pushes

That's why it's shown grayed out above

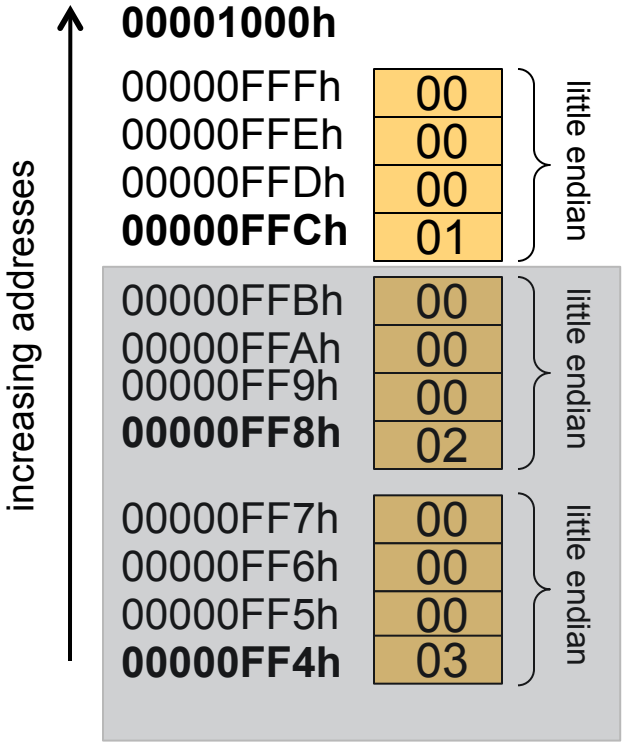
It's really an array and a pointer that masquerade as a stack data structure

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, ESP = 00000FF8h
pop    ebx     ; EBX = 2, ESP = 00000FFCh
```

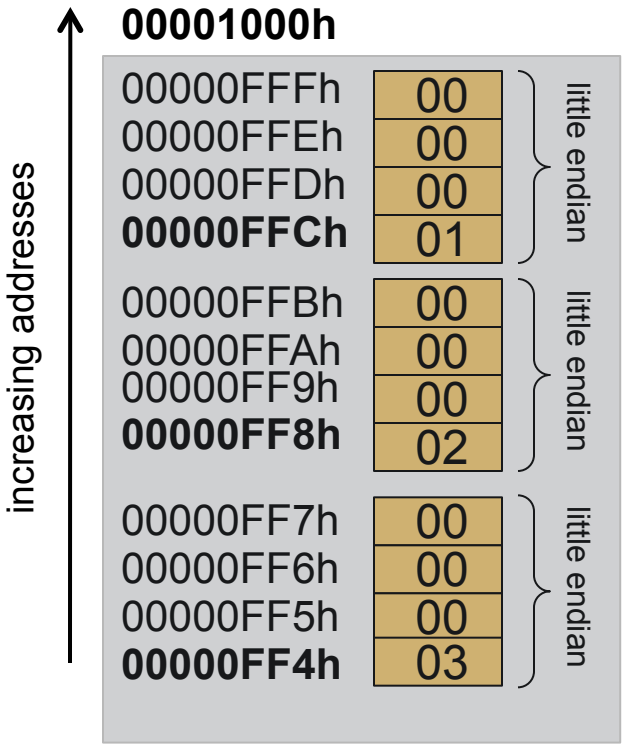


Example Stack Instructions

- Assuming that ESP=00001000h

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

pop    eax     ; EAX = 3, ESP = 00000FF8h
pop    ebx     ; EBX = 2, ESP = 00000FFCh
pop    ecx     ; ECX = 1, ESP = 00001000h
```



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!
  - Even if you "run out" of registers, using ESP to store your data is not an option
- Its value is updated automatically by **`push`** and **`pop`** instructions, and a few other instructions
- In a few very specific and well-known cases we'll update it by hand (future lecture notes)

PUSHA and POPA

- For subprograms, the stack it used to save/restore register values
- Say your program uses ebx and calls a function written by somebody else
- You have no idea whether that function uses ebx, but if it does, your ebx will be corrupted
- One easy solution:
  - push ebx onto the stack
  - call the function and let it do its thing until it returns
  - pop the stack to restore the ebx value
- The x86 offers two convenient instructions
  - `pusha`: pushes EAX, EBX, ECX, EDX, ESI, EDI, and EBP onto the stack
  - `popa`: pops the stack to restore them all
- We can say “save all my registers” and “restore all my registers”
  - Overkill and expensive, but safe and easy
  - NO good compiler will every use pusha/popa

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” C 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 attempt 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 (to retrieve the return address)
  - Unconditionally jumps to that address (returning from a function)

Without `call` and `ret`

```
    . . .
    mov  ecx, return_here_1 ; store the return address
    jmp   func              ; "call" the function
    return_here_1:
    . . .
    mov  ecx, return_here_2 ; store the return address
    jmp   func              ; "call" the function
    return_here_2:
    . . .
func:
    . . .
    jmp   ecx          ; 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               ; pops value 12 and
                      ; interprets it as
                      ; a return address!!
```

- This bugs is also an exploit technique (stay tuned)

One example

- Let’s write together a program in which the main program calls a function f that calls a function g
  - And try to see what happens if we “corrupt” the stack…

Important Takeways

- There are instructions to maintain a “runtime stack” in memory
  - ESP always contains the address of the item “on top” of the stack
  - **`push`** and **`pop`** instructions updated ESP
- The return address of a subprogram call is pushed on the stack by the **`call`** instruction and popped off the stack by the **`ret`** instruction

Conclusion

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