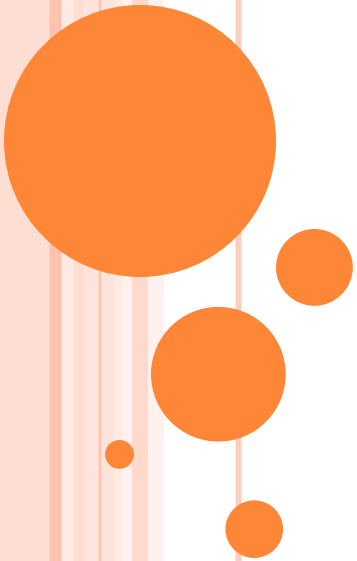


Anita's Super Awesome Recitation Slides

15/18-213: Introduction to Computer Systems
Stacks and Buflab, 11 Feb 2013

Anita Zhang, Section M

What's New (or Not)

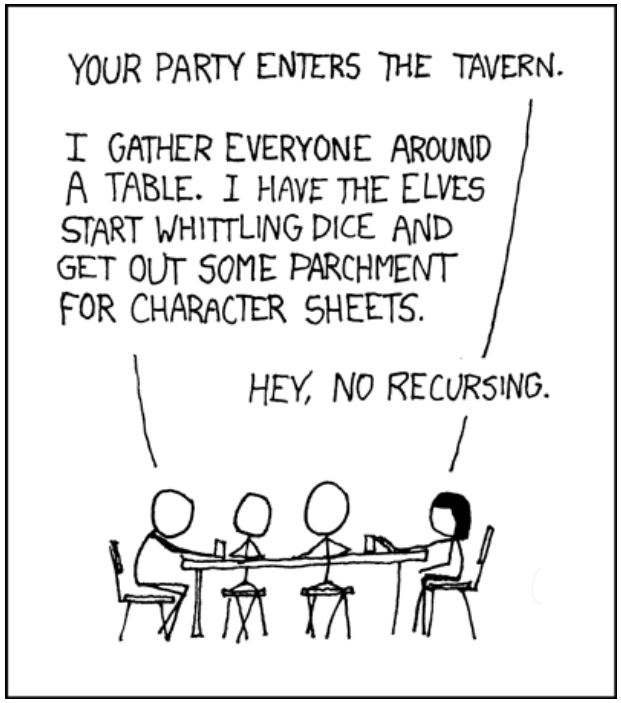
- Style scores for Datalab released
  - We tried to be harsh >:D
- Bomblab is due tomorrow night
  - Your late days are wasted here
  - "If you wait until the last minute, then it only takes a minute!"
- Buflab comes out tomorrow night
  - Hacking the stack
- Stacks will be on the exams
  - They're tough at first, but I believe in you ☺

Gift from Anita

For those of you who asked:

http://www.contrib.andrew.cmu.edu/~anitazha/15213_tips.html

Something, Something Motivation



"In order to support general recursion, a language needs a way to allocate different activation records for different invocations of the same function. That way, local variables allocated in one recursive call can coexist with local variables allocated in a different call." (credits to stack overflow)

Journey Through Time

- Stacks
  - IA32 Stack Discipline
  - More Stack Stuff
  - Stack Walkthrough
  - Differences between x86 (IA32) and x86_64
- Buflab Quick Start
  - Essential Items of Business

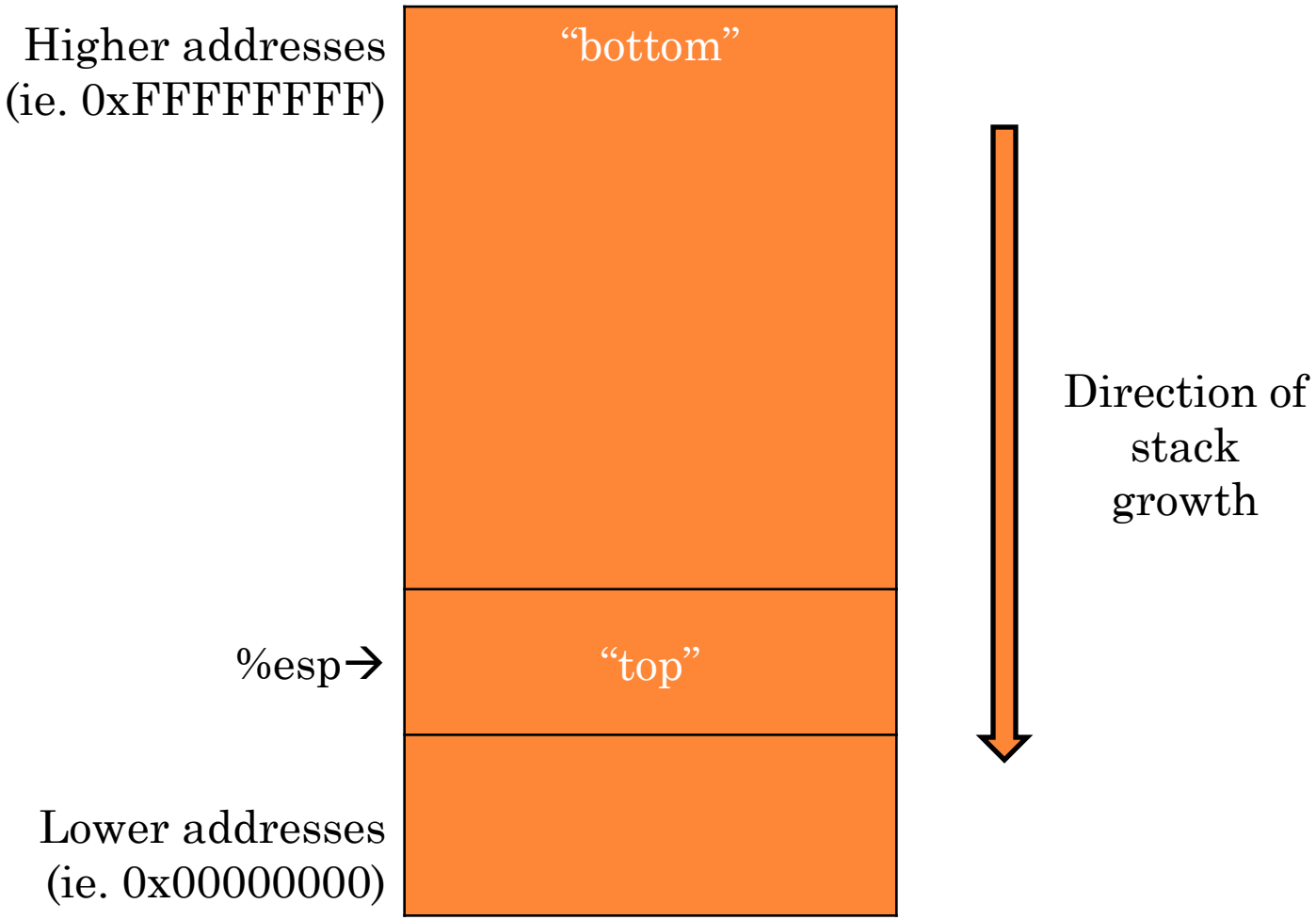
IA32 Registers

- 6 general purpose registers
  - Caller save
    - %eax, %ecx, %edx
    - Saved by the caller of a function
    - Before a function call, the caller must save any caller save register values it wants preserved
  - Callee save
    - %ebx, %edi, %esi
    - Saved by the callee of a function
    - The callee is required to save and restore the values in these registers if it is using them in the function

More IA32 Registers

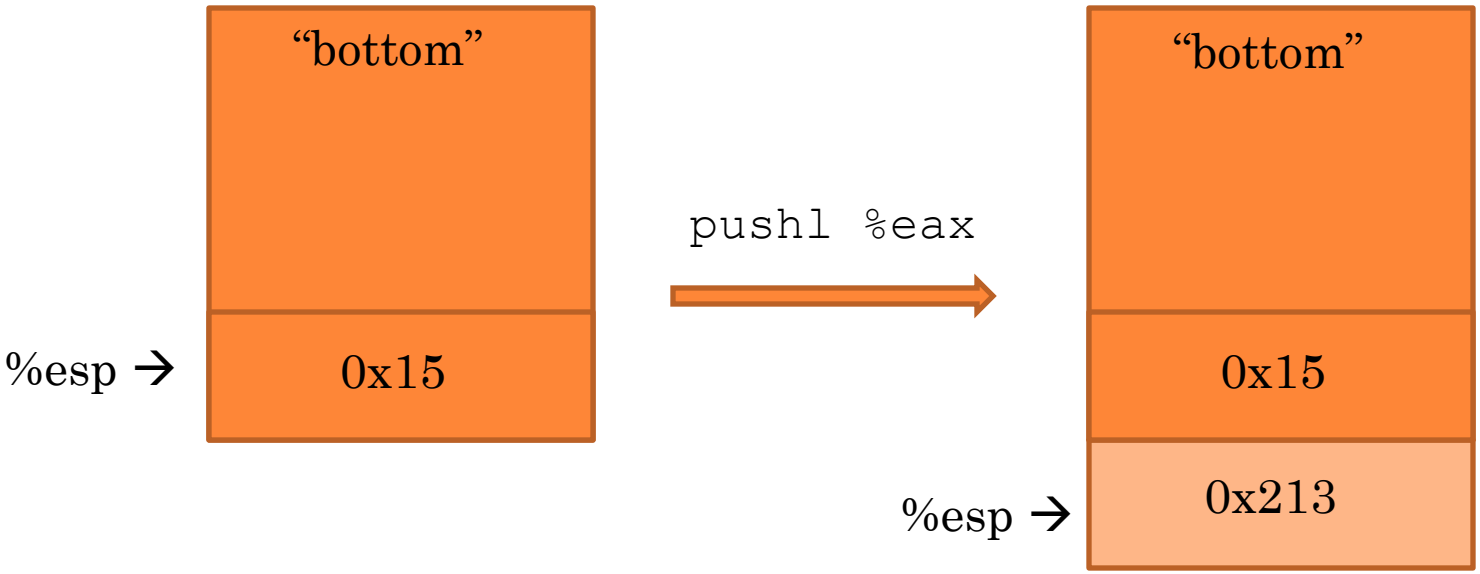
- Base Pointer
  - %ebp
  - Points to the "bottom" of the stack frame
- Stack Pointer
  - %esp
  - Points to the "top" of the stack
- Instruction Pointer (Program Counter)
  - %eip
  - Points to the next instruction to be executed

IA32 Terminology



What Happens in IA32

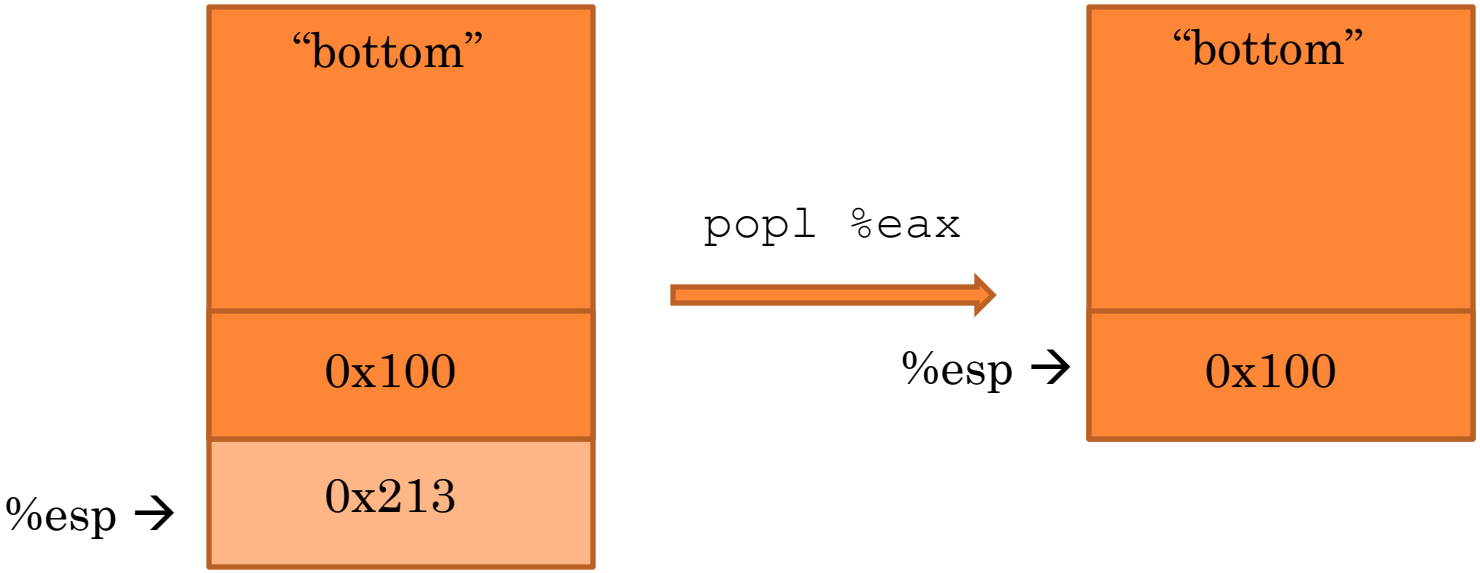
- Pushing on the stack



- In general, pushl translates to (in AT&T syntax):
  - subl $0x4, %esp
    movl src, (%esp)

What Happens in IA32

- Popping off the stack

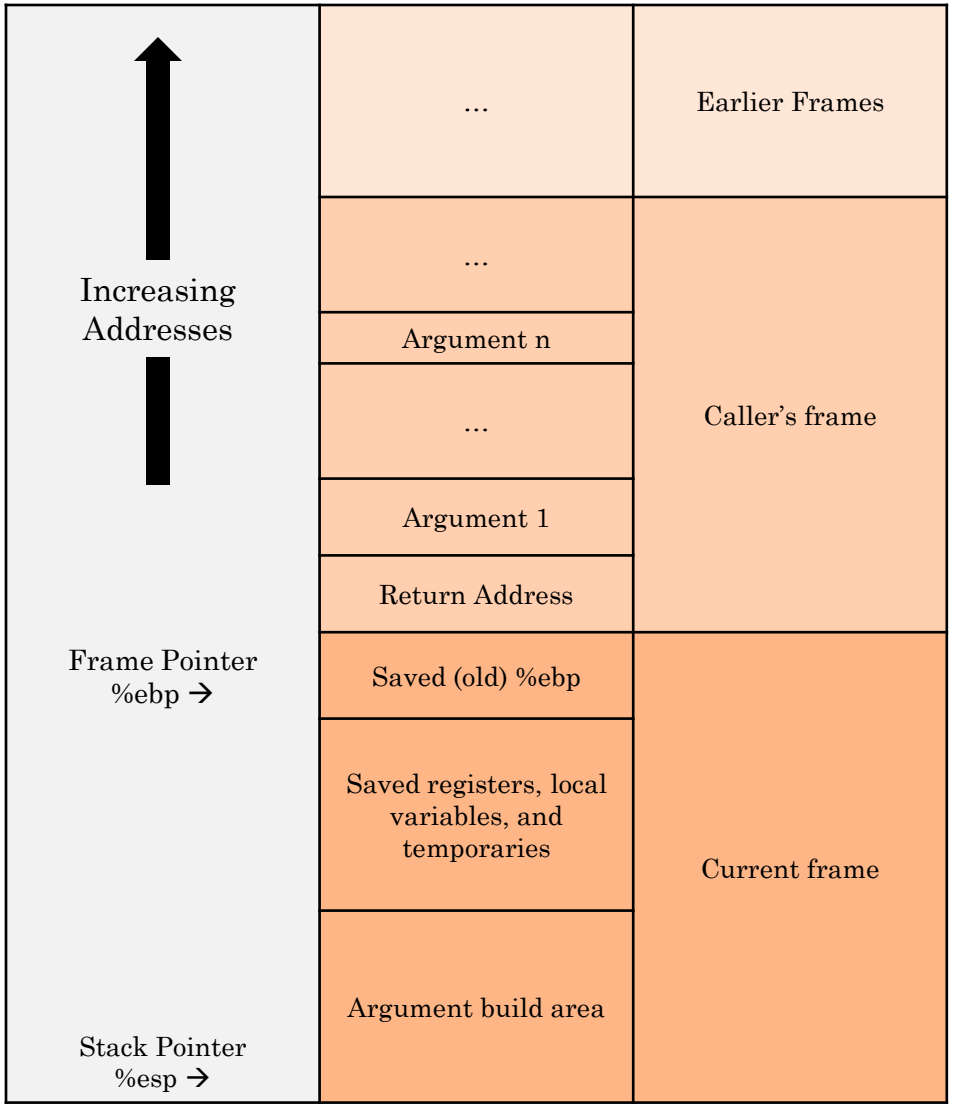


- In general, popl translates to (in AT&T syntax):
  - movl (%esp), dest
    addl $0x4, %esp

Stack Frames Whatchamacallits?

- Every function call gets a "stack frame"
- All the useful stuff can go on the stack!
  - Local variables (scalars, arrays, structs)
    - What the compiler couldn't fit into registers
  - Callee/caller save registers
  - Temporary variables
  - Arguments
- Stacks make recursion work
- Key idea: "Storage for each *instance* of procedure call" (stolen out of 15-410 slides)

So THAT'S What it Looks Like…



Stack Frames in Action

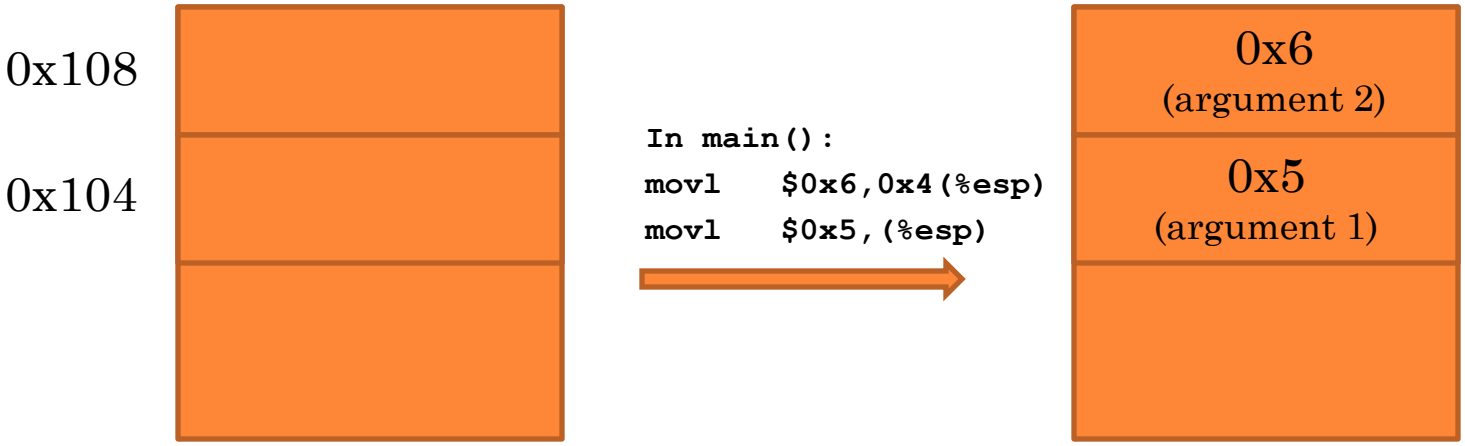
C Code	Disassembly

```
int main() {
    return addition(5, 6);
}

int addition(int x, int y)
{
    return x+y;
}
```

```
08048394 <main>:
 8048394:       55                      push   %ebp
 8048395:       89 e5                   mov    %esp,%ebp
 8048397:       83 e4 f0                and    $0xfffffff0,%esp
 804839a:       83 ec 10                sub    $0x10,%esp
 804839d:       c7 44 24 04 06 00 00    movl   $0x6,0x4(%esp)
 80483a4:       00
 80483a5:       c7 04 24 05 00 00 00    movl   $0x5,(%esp)
 80483ac:       e8 02 00 00 00          call   80483b3 <addition>
 80483b1:       c9                      leave
 80483b2:       c3                      ret

080483b3 <addition>:
 80483b3:       55                      push   %ebp
 80483b4:       89 e5                   mov    %esp,%ebp
 80483b6:       8b 45 0c                mov    0xc(%ebp),%eax
 80483b9:       8b 55 08                mov    0x8(%ebp),%edx
 80483bc:       8d 04 02                lea    (%edx,%eax,1),%eax
 80483bf:       c9                      leave
 80483c0:       c3                      ret
```

Breakdown of the Example

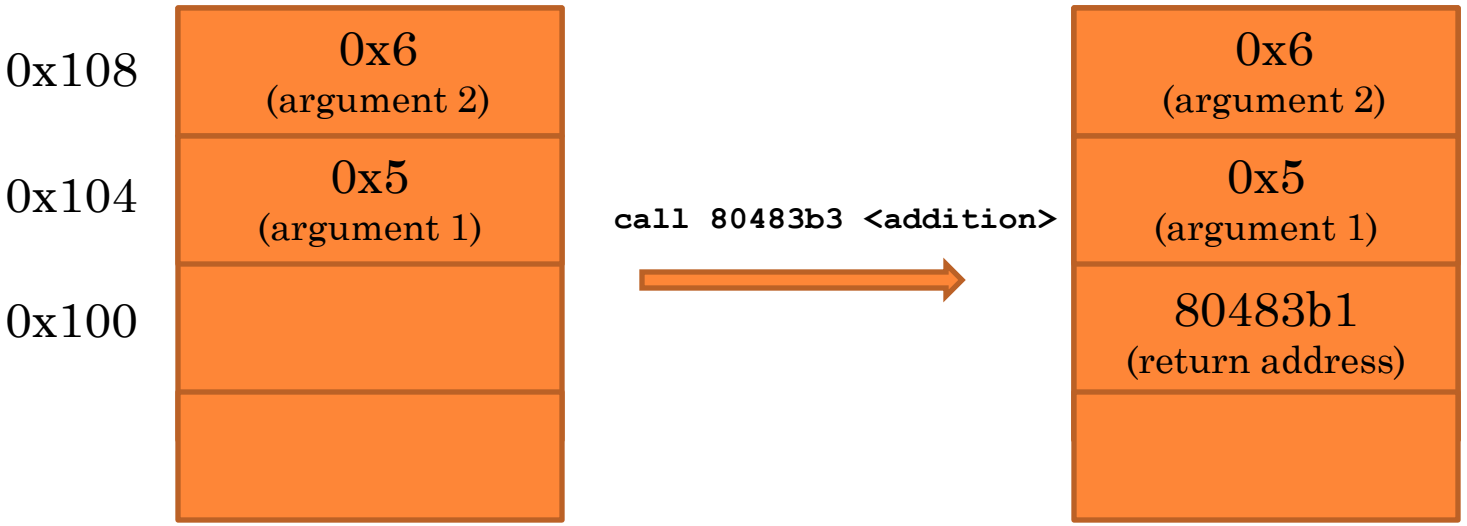
- Build the arguments (special note: 2 instructions are executed in this example)



Before	After
%esp = 0x104 %ebp = 0xffffd418 %eip = 804839d	%esp = 0x104 %ebp = 0xffffd418 %eip = 80483ac

Breakdown of the Example

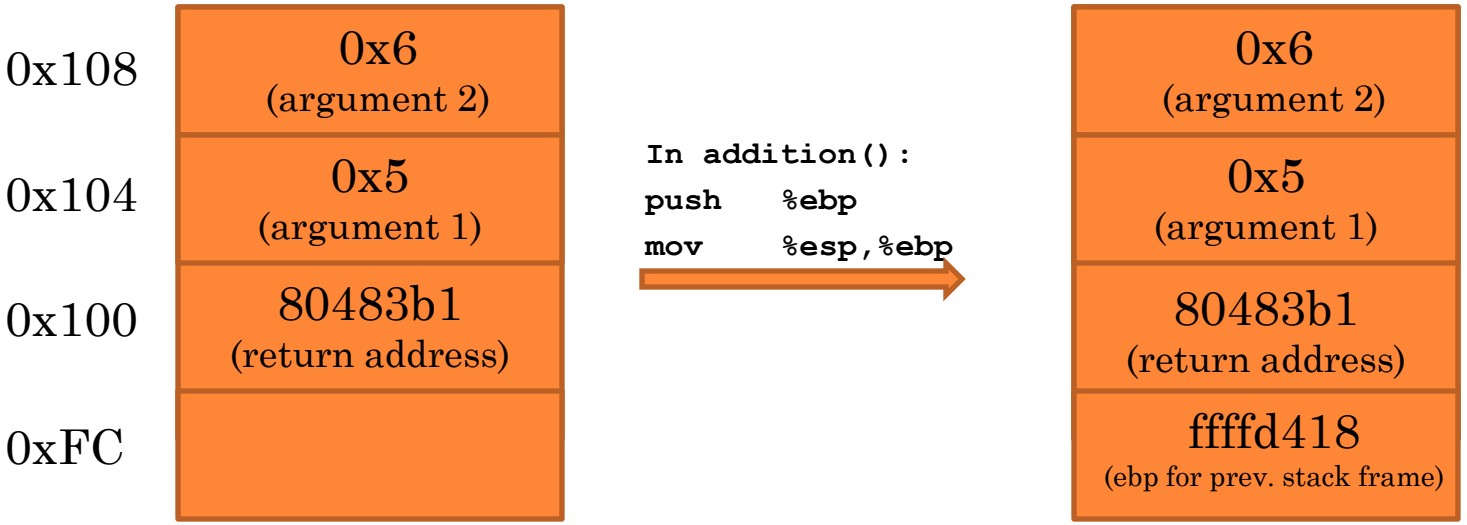
- Call the function



Before	After
%esp = 0x104 %ebp = 0xffffd418 %eip = 80483ac	%esp = 0x100 %ebp = 0xffffd418 %eip = 80483b3

Breakdown of the Example

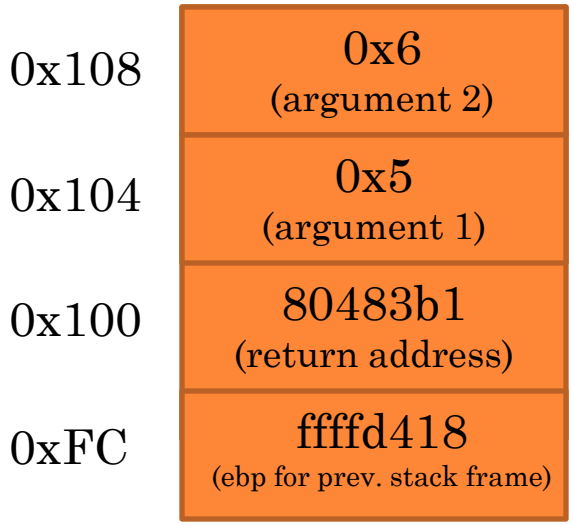
- Stack frame set up for the callee
  (special note: 2 instructions are executed in this example)



Before	After
%esp = 0x100 %ebp = 0xffffd418 %eip = 80483b3	%esp = 0xFC %ebp = 0xFC %eip = 80483b6

Breakdown of the Example

- Accessing an argument



Argument	Location
Argument 2	0xC(%ebp)
Argument 1	0x8(%ebp)

- In the current frame, arguments are accessed via references to %ebp
  - Upon entry, we could also use %esp to get the arguments
  - Notice how argument 1 is at 0x8(%ebp), not 0x4(%ebp)

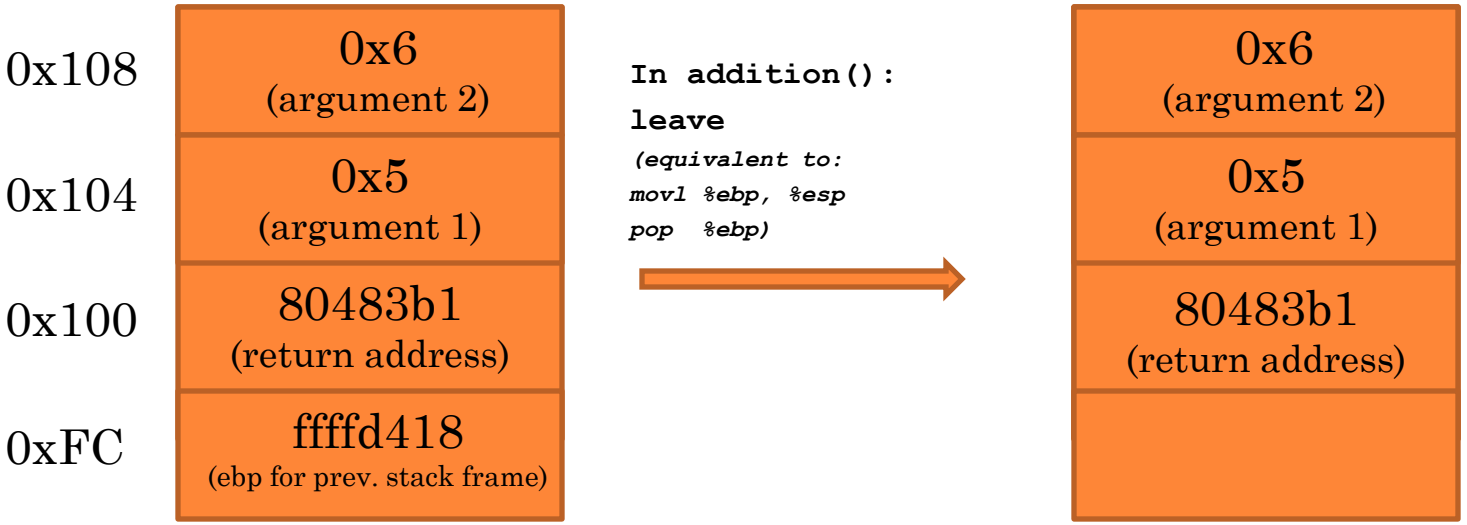
Breakdown of the Example

.....Stuff happens

(side track about how registers are not on the stack when they're pushed, but their values are)

Breakdown of the Example

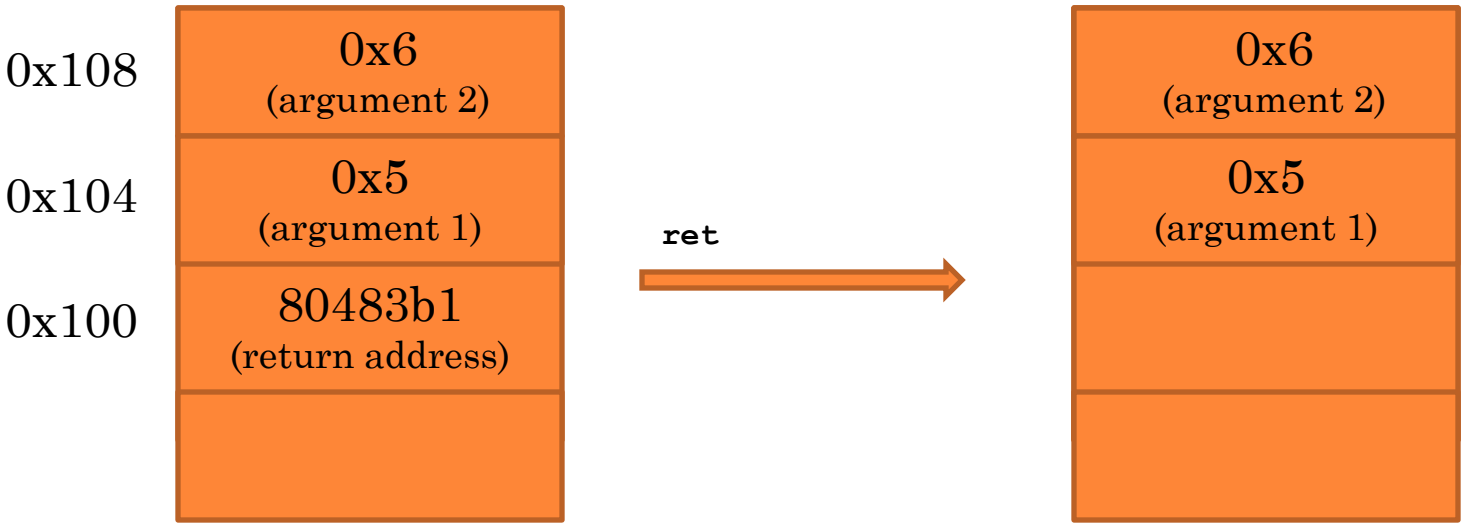
- Preparing to return from a function



Before	After
%esp = 0xFC %ebp = 0xFC %eip = 80483bf	%esp = 0x100 %ebp = 0xffffd418 %eip = 80483c0

Breakdown of the Example

- Return from a function



Before	After
%esp = 0xFC %ebp = 0xffffd418 %eip = 80483c0	%esp = 0x104 %ebp = 0xffffd418 %eip = 80483b1

Stacks and Stuff on x86_64

- Arguments (<= 6) are passed via registers
  - %rdi, %rsi, %rcx, %r8, %r9
  - Extra arguments passed via stack!
    - IA32 stack knowledge still matters!
- Don't need %ebp as the base pointer
  - Compilers are smarter now
- Overall less stack use
  - == Potentially better performance
- 64-bit stack discipline is required knowledge
  - Even if it's not tested on labs

And Floating Point?

- Floating point arguments are complicated
  - Out of the scope of this course
  - Some chips have a separate floating point stack
- Example of complication: x86_64 stack on function entry needs to be 16 byte aligned for floating point
  - Many trickies going on

An Aside

- This class is (strictly) x86(_64)
  - Other architectures may not always have the same convention
    - May use a combination of registers and stack to call functions
    - May not use stacks at all (???)
  - Stacks grow down/ up depending on what is implemented
    - Infinitely confusing to the newly initiated

BUFLAB

- A series of exercises asking you to overflow the stack and change execution
- A paper on stack corruption
  - *Smashing the Stack for Fun and Profit*
- **Incorrect inputs will not hurt your score**
- Basic approach
  - Examine the C code/disassembly
    - objdump -d bufbomb > bufbomb.d
  - Write a few lines of (corruption) assembly
    - Compile with gcc -m32 -c *example.S*
    - Get the byte codes with objdump -d *example.o* > *example.d*
  - Feed byte codes into hex2raw, then into bufbomb

BUFLAB

- The writeup contains (pretty much) everything you need to know about the tools and how to write corruption code
- If you ask a question that is answered in the writeup, I will be sad
  - The other TAs will be sad too, they're just too manly to voice it
- The writeup is on Autolab (separate from the tar?)

BUFLAB TOOLS

- ./makecookie *andrewID*
  - Makes a unique “cookie” based on your Andrew ID
- ./hex2raw
  - Use the hex generated from assembly to pass raw strings into bufbomb
- ./bufbomb -t *andrewID*
  - The actual program to attack
  - Always pass in with your Andrew ID so your score is logged

A Lesson on Endianness

- We're working with little endian
  - Least significant byte is at the lower address

Higher addresses …	Caller stack frame
Return Address	
Saved %ebp	← %ebp
Saved %ebx	
Canary	← Potential way to detect stack corruption
MSB [7] [6] [5] [4]	buf string (each char is a byte)
[3] [2] [1] [0] *LSB*	
… *Lower addresses*	

Stolen Credits & Questions Slide

- http://xkcd.com/244/
- http://stackoverflow.com/questions/14658612/what-properties-must-a-language-have-to-support-recursion
- http://www.cs.cmu.edu/~410/lecture.html
- CS:APP p. 220 – Stack Frame Structure
- CS:APP p.263 – Stack Frame with a canary