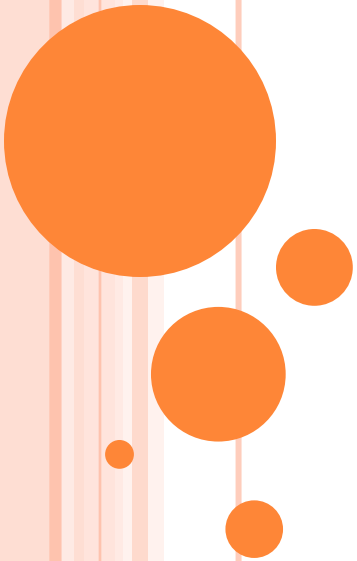


Anita's Super Awesome Recitation Slides

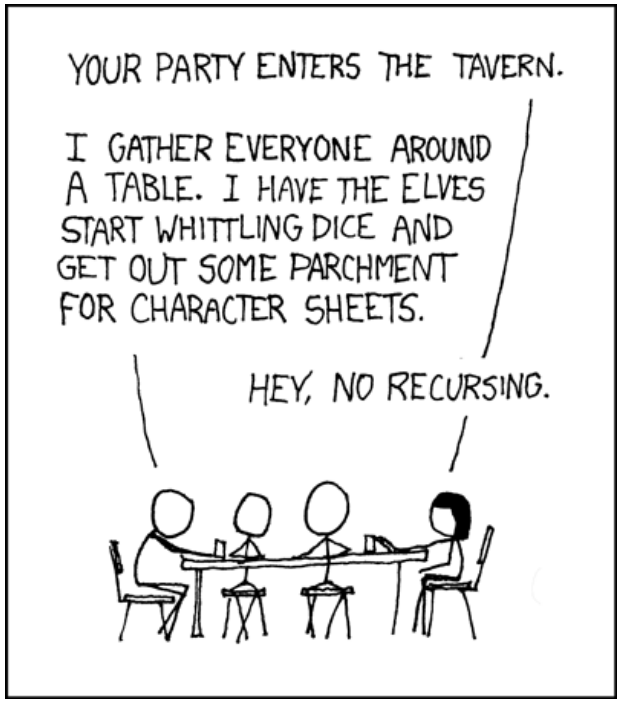
15/18-213: Introduction to Computer Systems
Stacks and Buflab, 11 Jun 2013

Anita Zhang, Section M

What's New (or Not)

- Bomblab is due tonight, 11:59 PM EDT
  - Your late days are wasted here
  - Student: "But if you wait until the last minute, then it only takes a minute!"
    - Not (quite) true
- Buflab comes out tonight, 11:59 PM EDT
  - Hacking the stack
- Stacks will be on the exams
  - They're tough at first, but I believe in you ☺

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

- Basic Assembly Review
  - Terminology
- Stacks
  - IA32 Stack Discipline
    - And a couple asides
  - Function Call Overview
  - Stack Walkthrough
  - Differences between x86 (IA32) and x86_64
- Buflab Quick Start
  - Essential Items of Business
  - Miscellany
- Demo

Definitions and Conventions

- Register
  - Some place in hardware that stores bits
    - Like boxes on the side of memory
- Caller save
  - 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
  - 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

Aside: Why both?

- Why do we have both caller and callee save?
  - Performance
  - Not all registers need to be saved

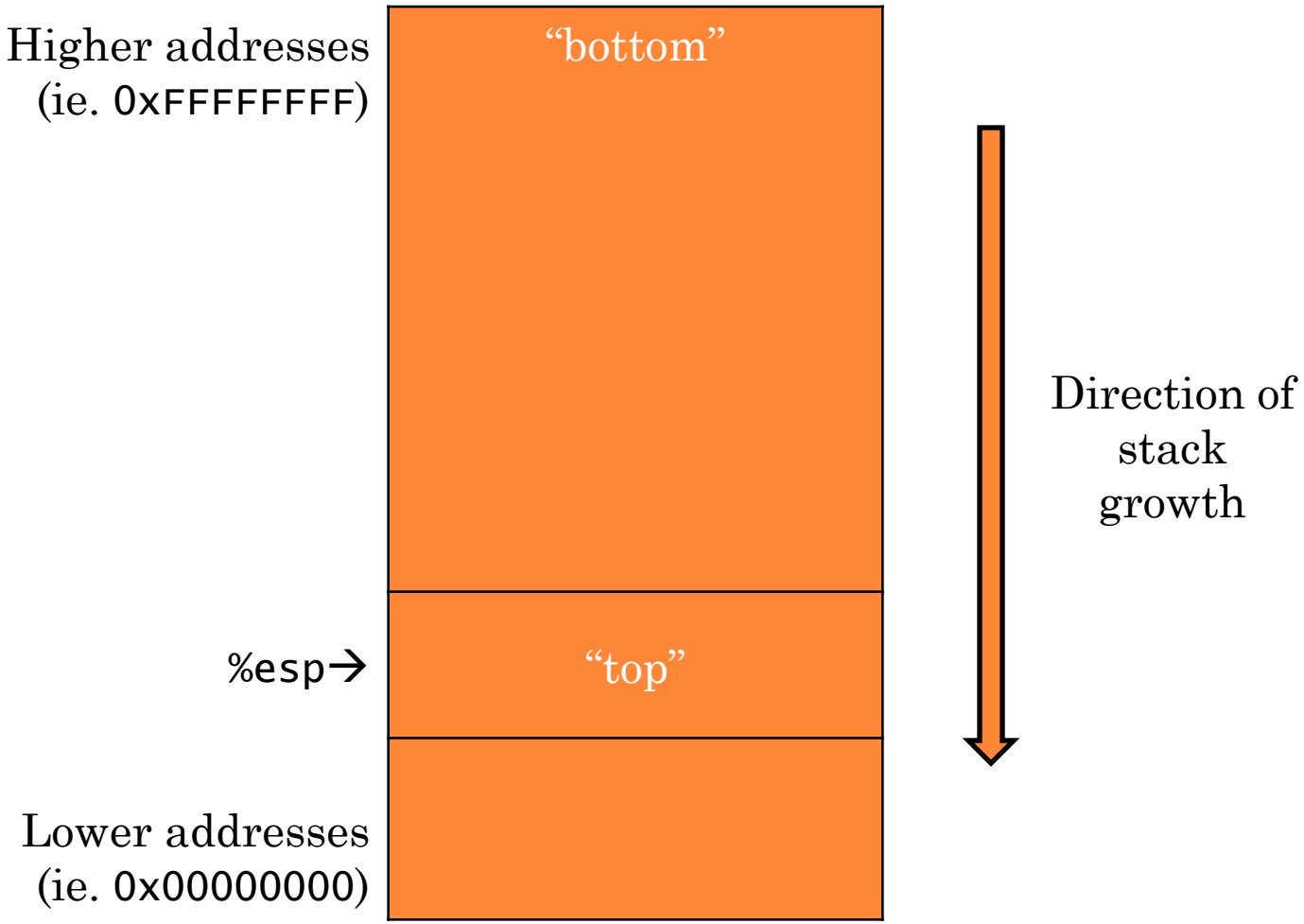
IA32 Registers

- 6 general purpose registers
  - Caller save
    - `%eax, %ecx, %edx`
    - Saved by the caller of a function
  - Callee save
    - `%ebx, %edi, %esi`
    - Saved by the callee of a function

More IA32 Registers

- Base Pointer
  - `%ebp`
  - Points to the "bottom" of the stack frame
    - AKA the location of old `%ebp` that gets pushed on entry
- Stack Pointer
  - `%esp`
  - Points to the "top" of the stack
    - Usually whatever was last pushed on the stack
- Instruction Pointer (Program Counter)
  - `%eip`
  - Points to the next instruction to be executed

IA32 Terminology



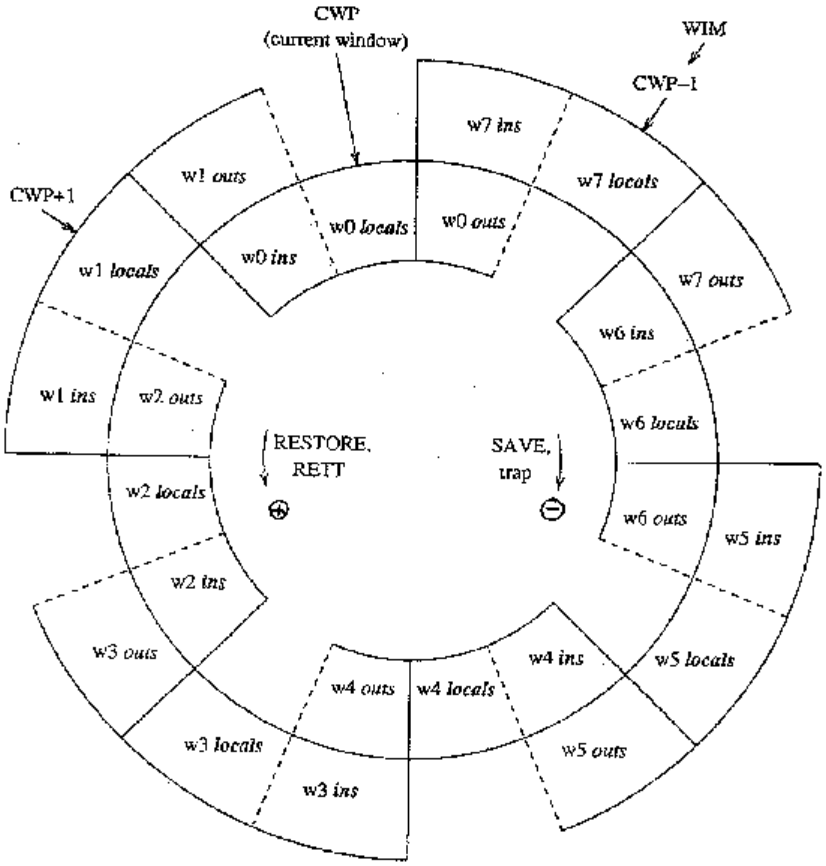
Aside Stuff

- 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 (weird, I know)
  - Stacks grow down/ up depending on what is implemented
    - Infinitely confusing to the newly initiated

Aside: Direction of Growth

- Stack direction REALLY doesn't matter
  - Direction of growth is dependent on the processor
  - May be selectable for up/down
  - …Or some other direction…?

BAM! Circular Stack!



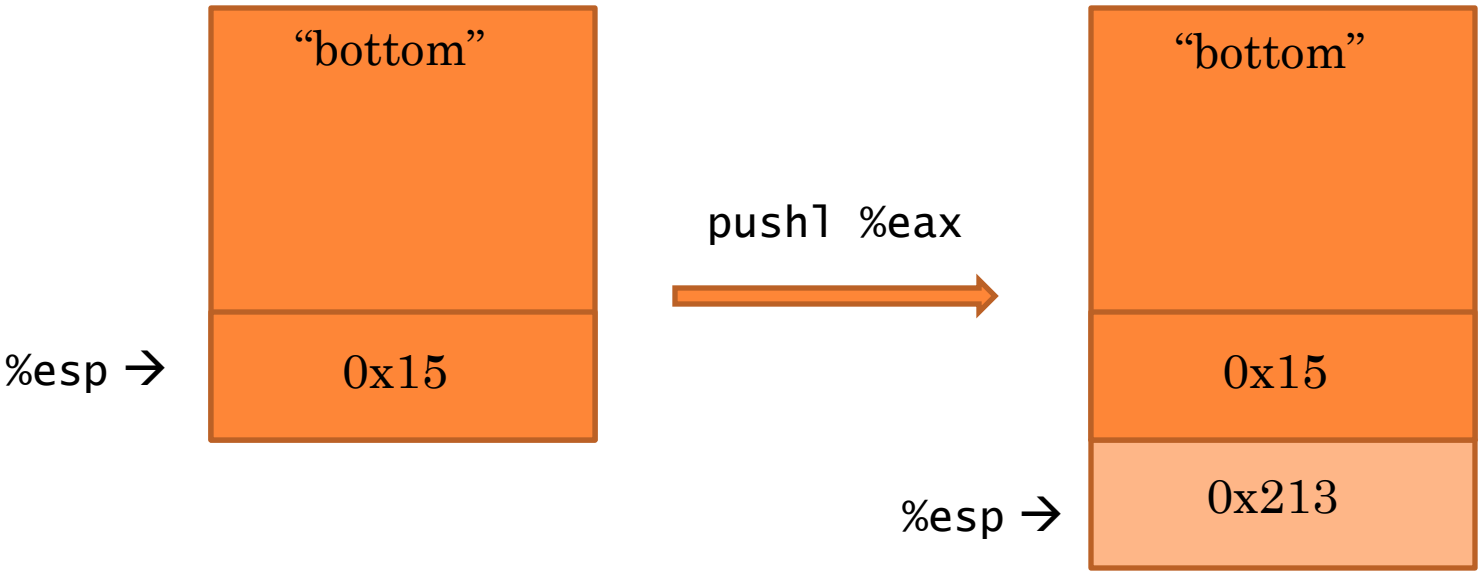
SPARC (scalable processor architecture) Architecture

Aside: Direction of Growth

- Examples from StackOverflow
  - x86 down
  - SPARC in a circle
  - System z in a linked list (down, at least for zLinux)
  - ARM selectable
  - PDP11 down

What Happens in IA32

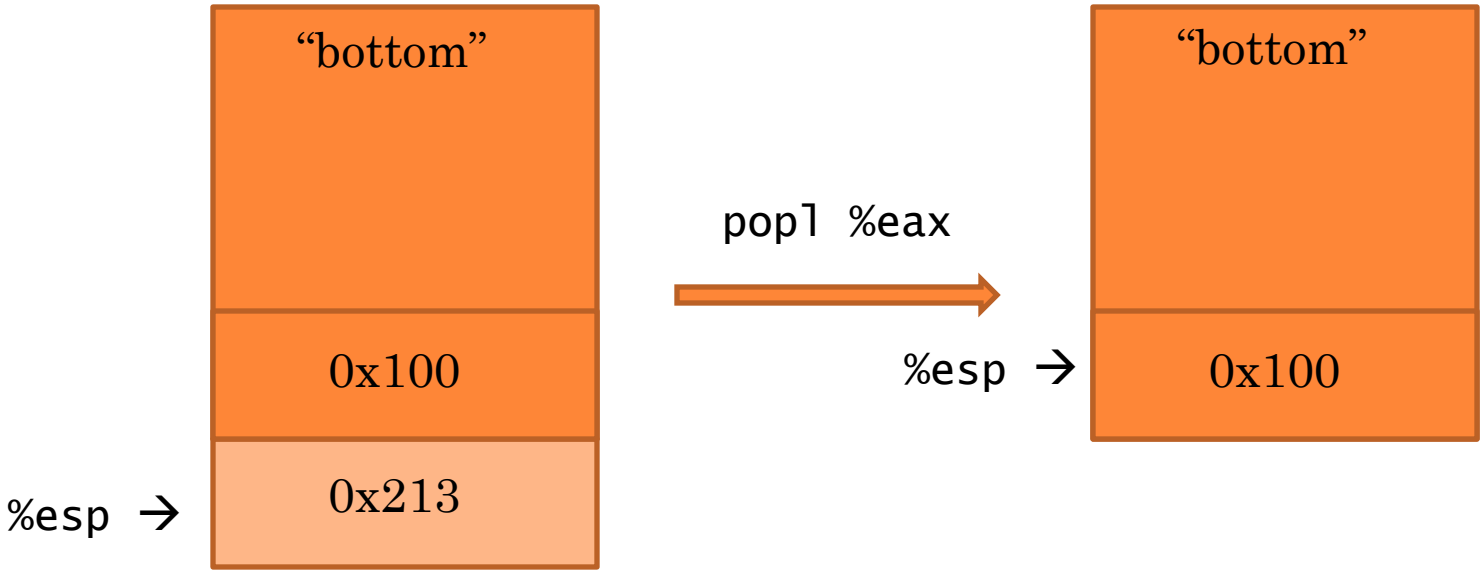
- Pushing on the stack



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

What Happens in IA32

- Popping off the stack

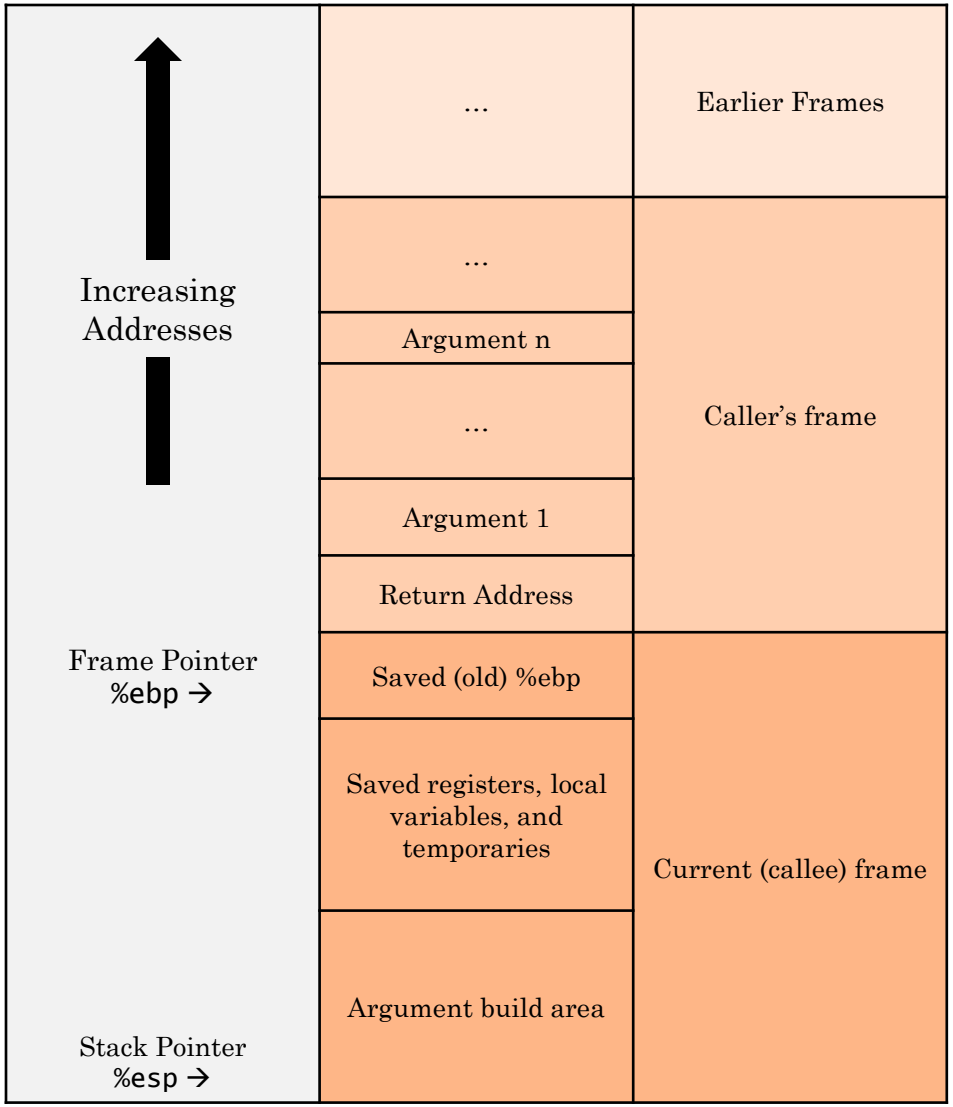


- In general, `popl` translates to (in AT&T syntax):
  - `movl (%esp), dest`
    `addl $0x4, %esp`

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…



Function Call Caller (IA32)

- Caller
  - Save (push) relevant caller save registers
  - Push arguments
  - Call function
- Caller after function return
  - "Remove" (add to `%esp` or pop) arguments
  - Restore (pop) saved caller save registers

Function Call Callee (IA32)

- Callee
  - Push %ebp (save stack frame)
  - Move %esp into %ebp
  - Save (push) callee save registers it wants to use
- Callee before return
  - Restore (pop) callee save registers previously saved
  - Move %ebp into %esp
    - Moves stack pointer to the saved %ebp
  - Restore (pop) %ebp

Function Call More (IA32)

- Implied operations
  - "call" implicitly pushes return address
    - Return address is always of the instruction after the call
  - "ret" implicitly pops return address into `%eip`
    - Becomes the next instruction to execute!

Stack Frames in Action

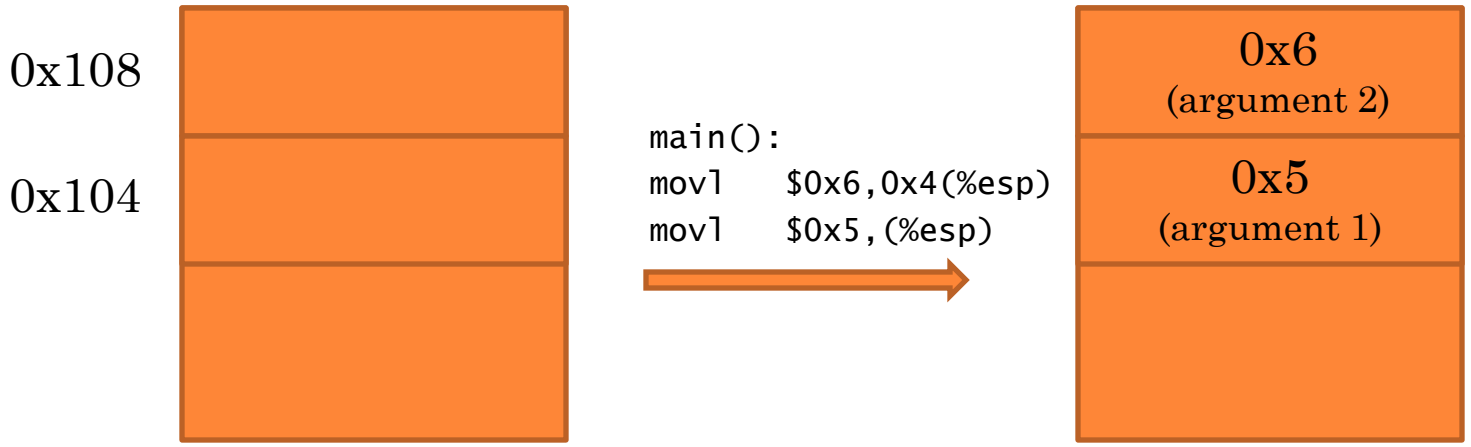
C Code	Disassembly

```
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: Arguments

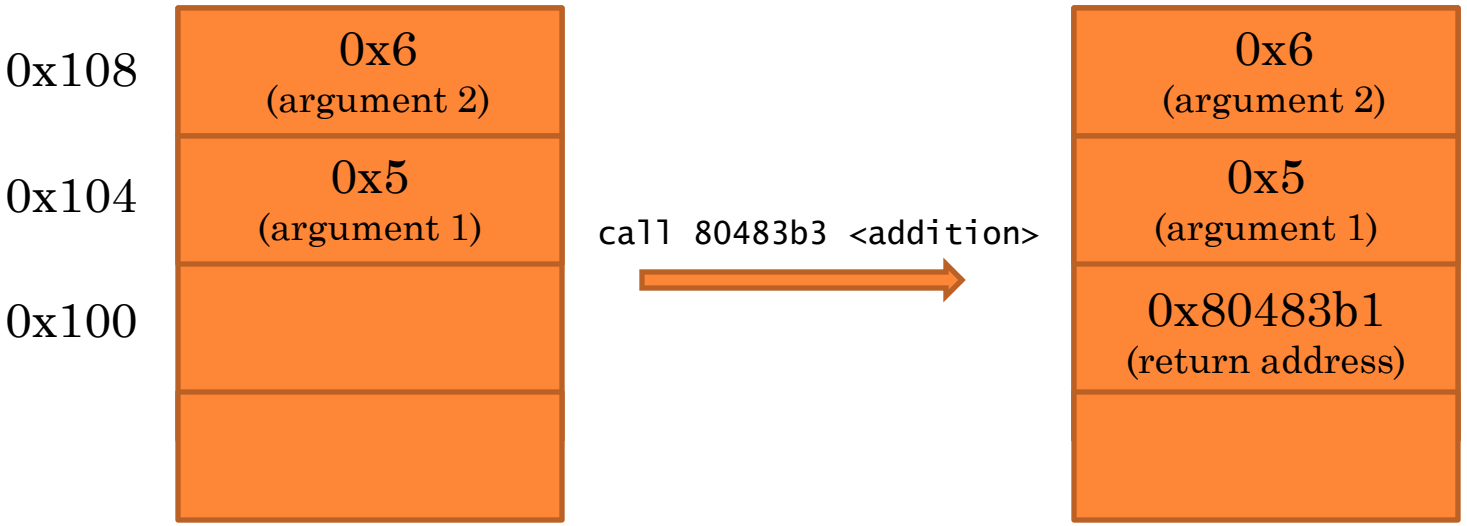
- Build the arguments (special note: 2 instructions are executed in this example)



Before	After
%esp = 0x104 %ebp = 0x200 %eip = 0x804839d	%esp = 0x104 %ebp = 0x200 %eip = 0x80483ac

Breakdown: Function Call

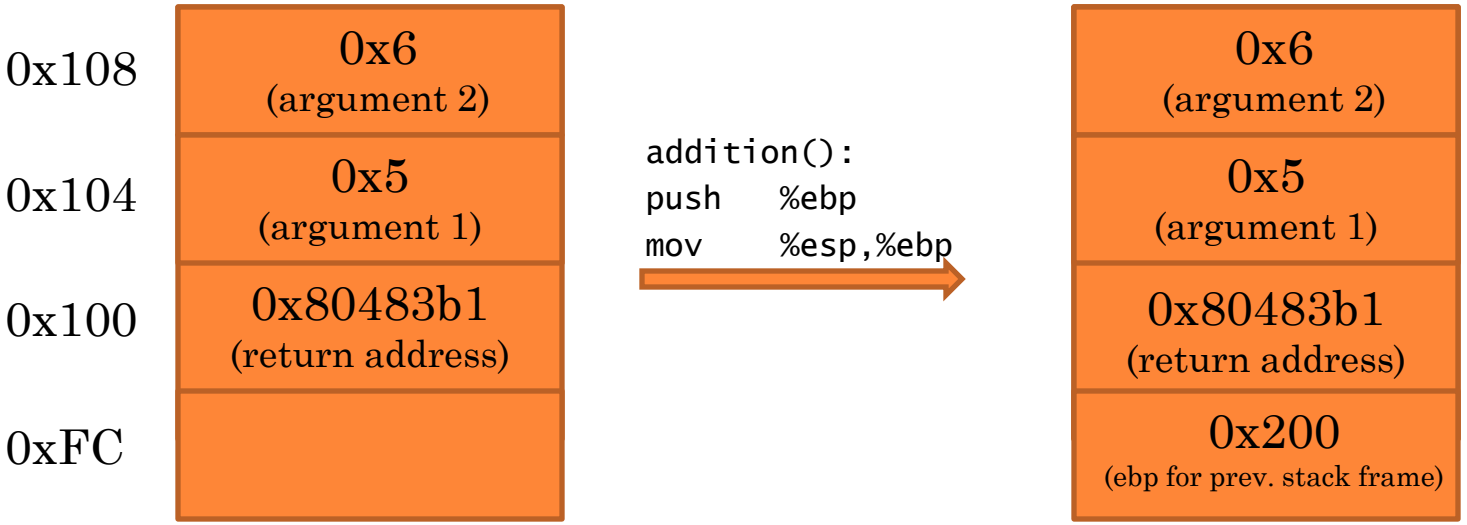
- Call the function



Before	After
%esp = 0x104 %ebp = 0x200 %eip = 0x80483ac	%esp = 0x100 %ebp = 0x200 %eip = 0x80483b3

Breakdown: Callee Set-up

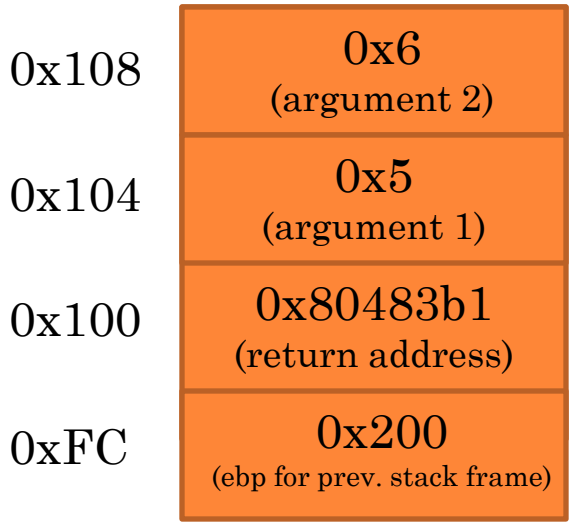
- Stack frame set up for the callee
  (special note: 2 instructions are executed in this example)



Before	After
%esp = 0x100 %ebp = 0x200 %eip = 0x80483b3	%esp = 0xFC %ebp = 0xFC %eip = 0x80483b6

Break From the Example.. Kind of

- 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)`

Let's Review the Code Again

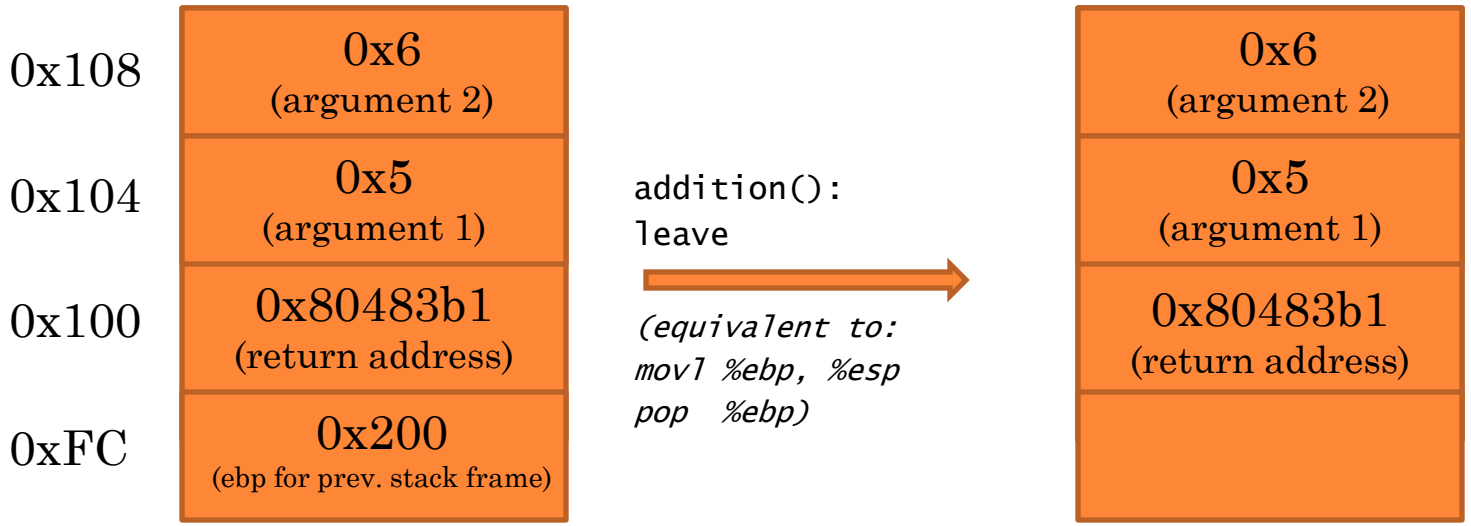
C Code	Disassembly

```
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

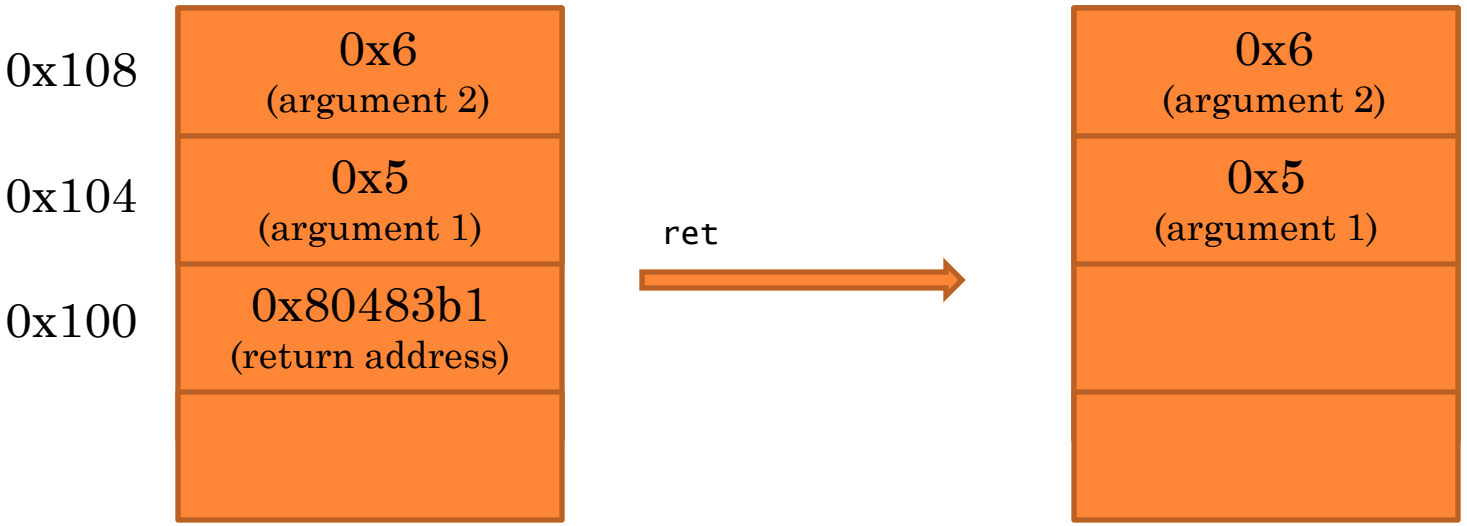
- Preparing to return from a function



Before	After
%esp = 0xFC %ebp = 0xFC %eip = 0x80483bf	%esp = 0x100 %ebp = 0x200 %eip = 0x80483c0

Breakdown of the Example

- Return from a function



Before	After
%esp = 0xFC %ebp = 0x200 %eip = 0x80483c0	%esp = 0x104 %ebp = 0x200 %eip = 0x80483b1

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

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
  - And other potential issues you shouldn't worry about

BUFLAB

- A series of exercises asking you to overflow the stack and change execution
  - You do this with inputs that are super long and write over stack values
- 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
  - Disassembling
    - `> objdump -d bufbomb > outfile`
  - Don't forget that GDB still exists!
- Put your byte code exploit into a text file
- Later, write a few lines of (corruption) assembly
  - Compiling
    - `> gcc -m32 -c example.s`
  - Get the byte codes
    - `> objdump -d example.o > outfile`

Feeding Byte Codes

- Option 1: Pipes
  - `> cat exploitfile | ./hex2raw | ./bufbomb -t andrewID`
- Option 2: Redirects
  - `> ./hex2raw < exploitfile > exploit-rawfile`
  - `> ./bufbomb -t andrewID < exploit-rawfile`
- Option 3: Redirects in GDB
  - `> gdb bufbomb`
  - `(gdb) run -t andrewID < exploit-rawfile`

BUFLAB

- The writeup contains (pretty much) everything you need
  - How to use the tools
  - How to write corruption code
  - Even tells you how to solve the level (at a high level)!
- Please don't ask questions answered by the writeup
  - Or I will make this sad face: (TT _ TT)
- The writeup is on Autolab
  - Couple links down from the handout

BUFLAB TOOLS

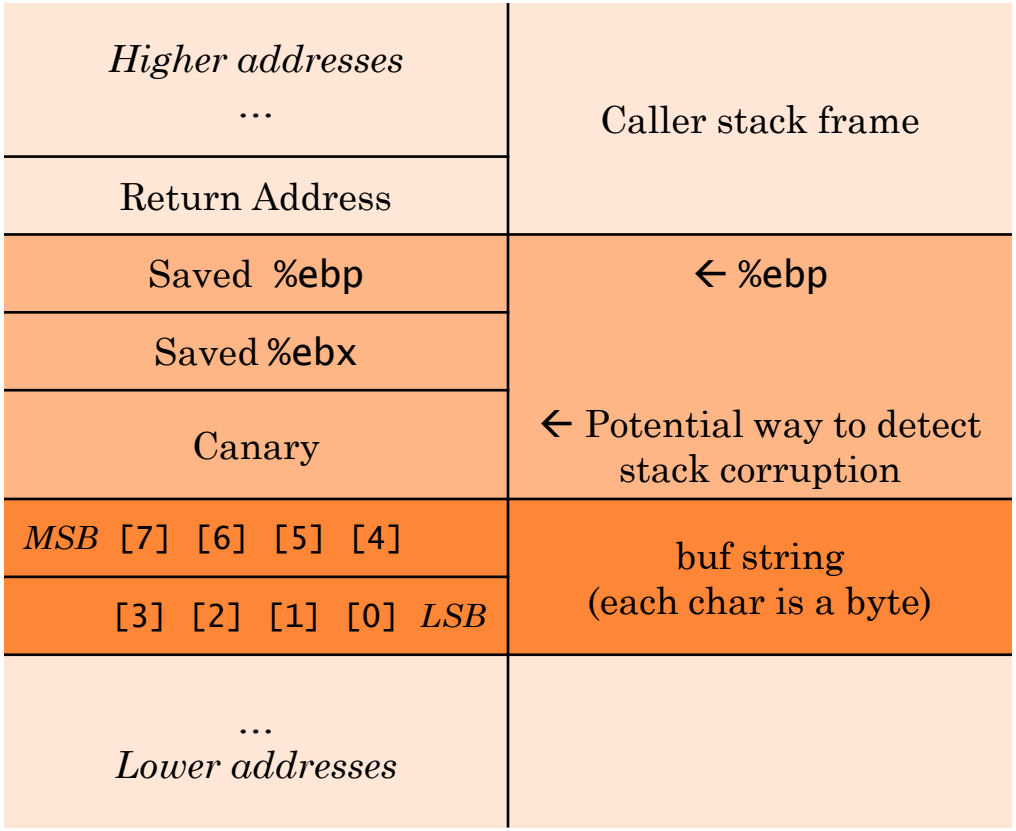
- `./makecookie` *`andrewID`*
  - Makes a unique "cookie" based on your Andrew ID
- `./hex2raw`
  - Use the hex generated from assembly to pass raw strings into bufbomb
- `./bufbomb -t` *`andrewID`*
  - The actual program to attack
  - Always pass in with your Andrew ID so your score is logged

Potential Points of Failure

- Don't use byte value 0A in your exploit
  - ASCII for newline
  - `Gets()` will terminate early if it sees this
- Multiple exploits submitted for the same level always takes the latest submission
  - So if you pass correctly, but accidently pass the wrong exploit later, just pass the correct one again
- If you manage to execute your exploit….
  - GDB will say weird things
  - "Can't access memory…" etc.
  - Just ignore it and keep going
- Don't forget the –n flag on the last level

A Lesson on Endianness

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



Miscellany but Necessary

- Canaries
  - Attempts to detect overrun buffers
  - Sits at the end of the buffer (array)
    - If the array overflows, *hopefully* we detect this with a change in the canary value….
- Nop sleds
  - The "nop" instruction means "no-op/ no operation"
    - In computer architecture it's like "pipeline bubbles"
  - Consider many nop byte codes in an exploit
    - If an actual exploit is placed at the end of the nop sled, it allows for a less precise return address

Stolen Credits & Questions Slide

- xkcd: Tabletop Roleplaying
- StackOverflow: Supporting Recursion
- Understanding the SPARC Architecture
- StackOverflow: Direction of Stack Growth
- CS:APP p. 220 – Stack Frame Structure
- *Smashing the Stack for Fun and Profit*
- CS:APP p.262 – NOP sleds
- CS:APP p.263 – Stack Frame with a canary
- Upcoming Double Mocha Latte Picture

Demo Time!

- Byte code format
- Byte code feeding
- Example assembly
- Compiling assembly
  - Not quite assembling
- Assembly to byte code