CS 33: Week 3 Discussion

x86 Assembly (v1.0)
Section 1G

Announcements

- HW2 due Sunday
- **MT1 this Thursday!**
- Lab2 out

Info

Name: Eric Kim (Section 1G, 2-4 PM, BH 5419)

Office Hours (Boelter 2432)

- Wed (3-4 PM)
- Thurs (12-1 PM)

Feel free to e-mail me at: erickim555@gmail.com

This Week...

- More x86 assembly fun

Exercise: Fun with arithmetic

(a) C code

```
1   int arith(int x,
2             int y,
3             int z)
4   {
5       int t1 = x+y;
6       int t2 = z*48;
7       int t3 = t1 & 0xFFFF;
8       int t4 = t2 * t3;
9       return t4;
10  }
```

Figure 3.8 **C and assembly code for arithmetic routine body.** The stack set-up and completion portions have been omitted.

Convert this function to x86. Assume that: `x at %ebp+8, y at %ebp+12, z at %ebp+16.`
Recall: `addl src dst, imull src dst, andl src dst.`

Exercise: Fun with arithmetic

(a) C code

```
1   int arith(int x,
2             int y,
3             int z)
4   {
5       int t1 = x+y;
6       int t2 = z*48;
7       int t3 = t1 & 0xFFFF;
8       int t4 = t2 * t3;
9       return t4;
10  }
```

(b) Assembly code

```
    x at %ebp+8, y at %ebp+12, z at %ebp+16
1     movl    16(%ebp), %eax          z
2     leal    (%eax,%eax,2), %eax     z*3
3     sall    $4, %eax                t2 = z*48
4     movl    12(%ebp), %edx          y
5     addl    8(%ebp), %edx           t1 = x+y
6     andl    $65535, %edx            t3 = t1&0xFFFF
7     imull   %edx, %eax              Return t4 = t2*t3
```

Figure 3.8 **C and assembly code for arithmetic routine body.** The stack set-up and completion portions have been omitted.

Does this make sense? Note the usage of `leal` and `sall`, rather than simply using `imull`. Using `imull` isn't wrong - but it's good to be able to see why both approaches work!

Function Calling Convention

Function Frames

- When a function is called, a section of the stack is set aside for the function.
- Represented by two registers: the base pointer (%ebp) and the stack pointer (%esp).

%ebp: base pointer

- Points to the "beginning" of the function's stack frame.
- Should not change during function execution, unless another function call is made.

%ebp: Accessing Arguments

- Suppose f() calls g(x,y). Then, g can access its arguments x,y via %ebp:
    - x is at 8(%ebp), and y is at 12(%ebp)

%ebp

- What's at 0(%ebp) and 4(%ebp)?
    - %ebp points to the saved %ebp, ie the f's base pointer.
    - Need to set %ebp to the f's %ebp before returning from g! More on this later.

%ebp

- 4(%ebp) points to the saved return address, i. e. the next instruction to execute after returning from the function.
- The command `ret` updates the %eip (Instruction Pointer)

%esp: Stack Pointer

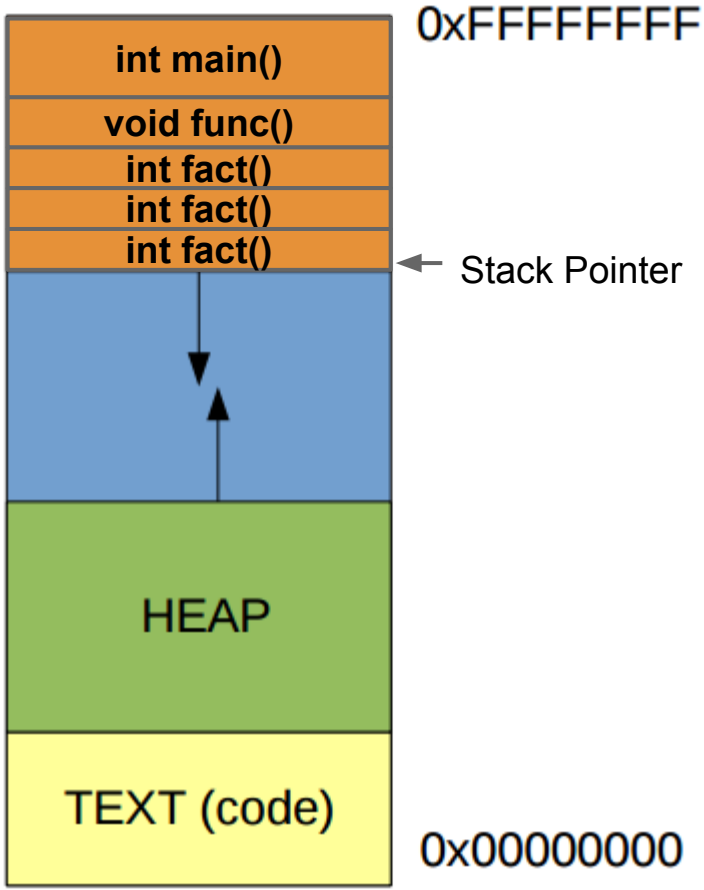
- Points to the "end" of the function frame.
- All of a function's local variables are stored between %ebp and %esp

%esp - Static Allocation

- At the start of a function, we allocate all required storage of local/temp variables by updating %esp

The Stack

- Contains local variables
- LIFO
- Grows “downward”
- Organized in frames



The Stack: pushl and popl

- pushl <SRC>
  - subl $4, %esp
  - movl <SRC> (%esp)
- popl <DST>
  - movl (%esp) <DST>
  - addl $4 %esp

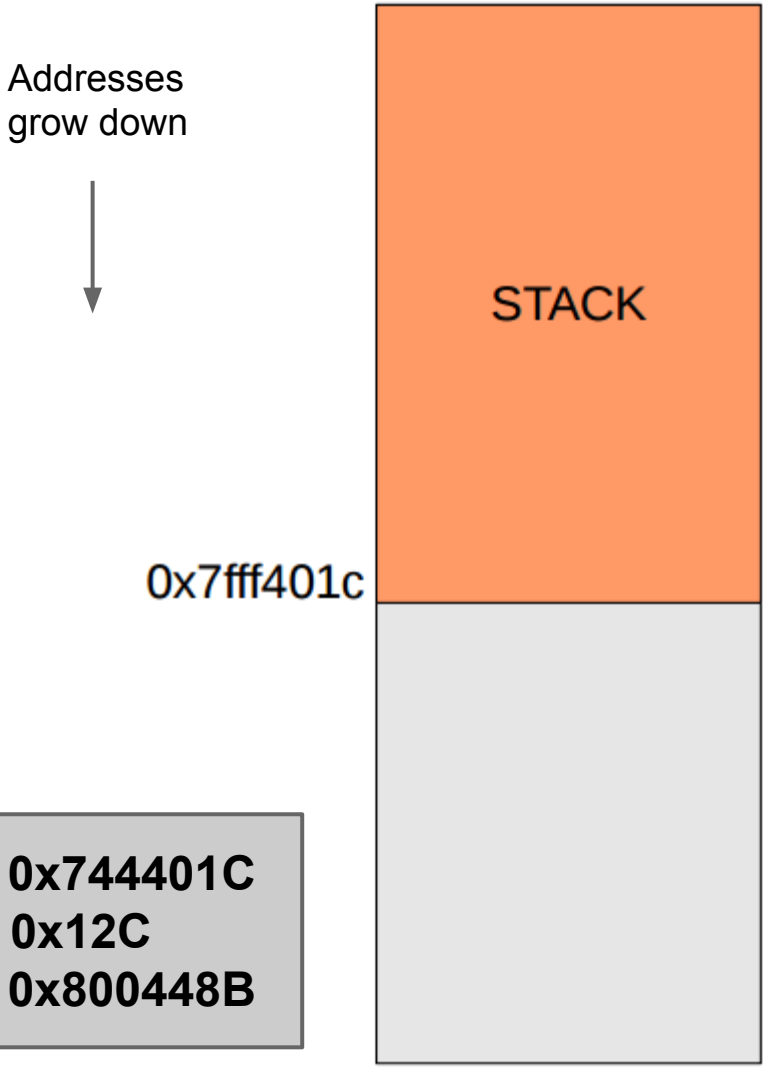
pushl, popl are convenience commands. Could simply use subl, movl,addl, etc. But you'd raise some eyebrows.

The Stack

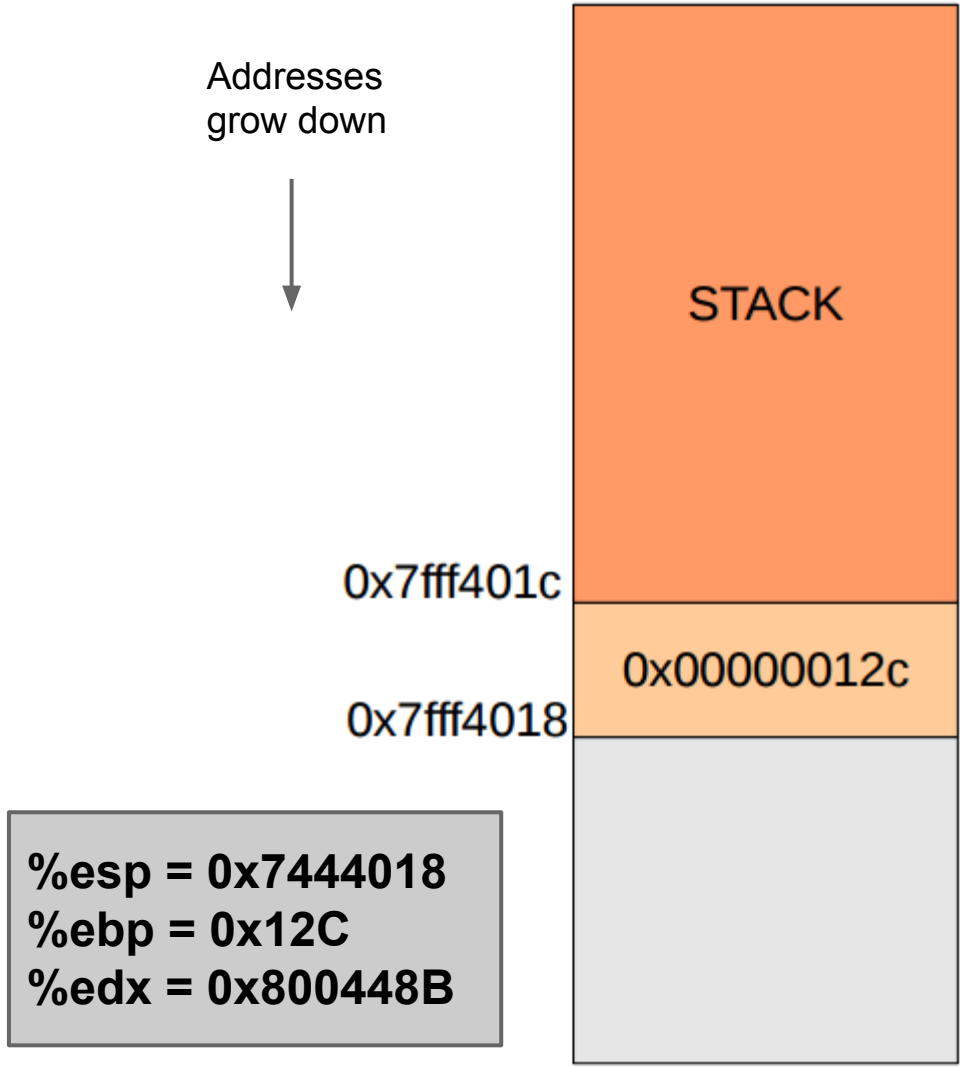
Consider the following stack.

What happens when I do:

pushl %ebp



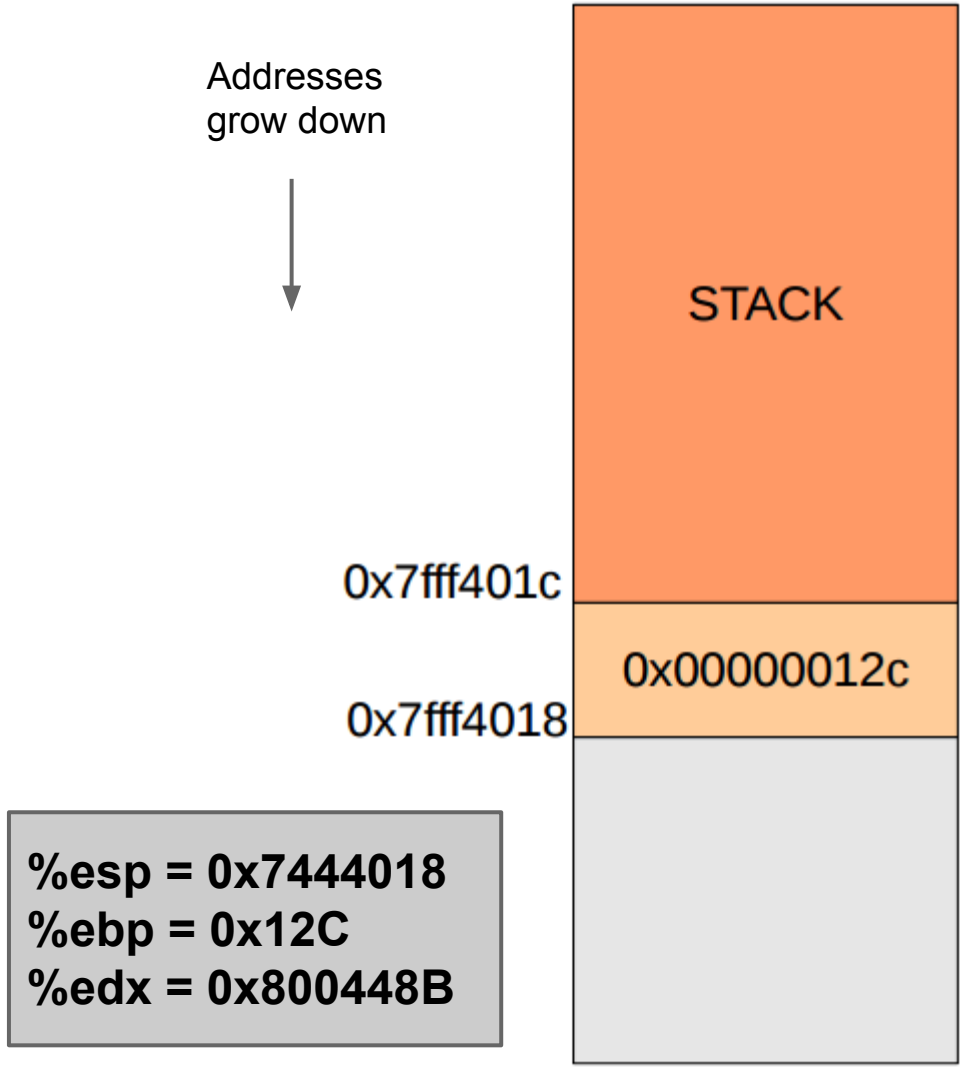
The Stack


Addresses
grow down
STACK
0x7fff401c
0x00000012c
0x7fff4018
%esp = 0x7444018
%ebp = 0x12C
%edx = 0x800448B

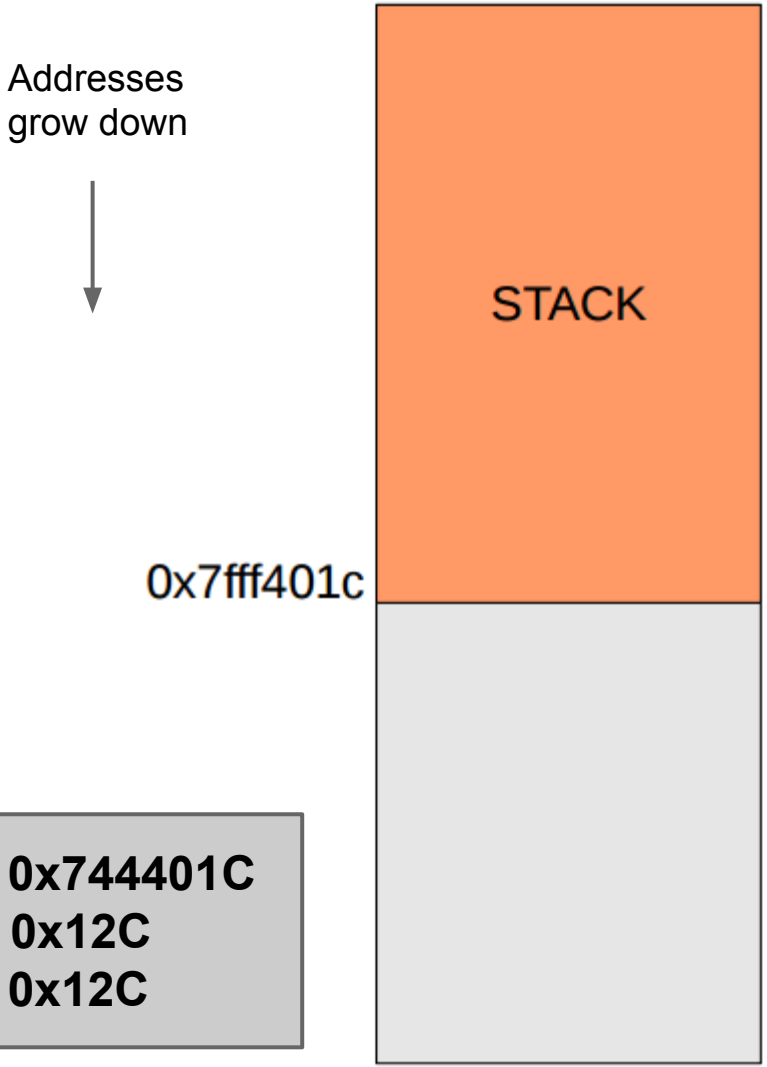
The Stack

What happens when I do:

```
popl %edx
```

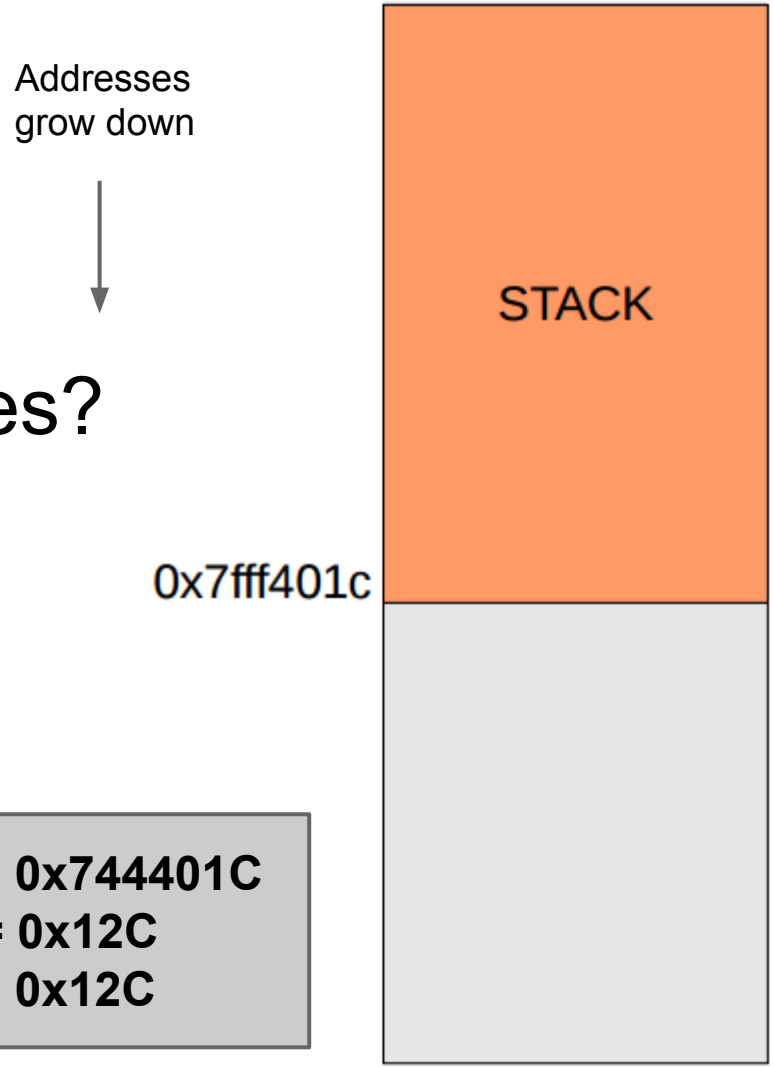


The Stack


Addresses
grow down
STACK
0x7fff401c
%esp = 0x744401C
%ebp = 0x12C
%edx = 0x12C

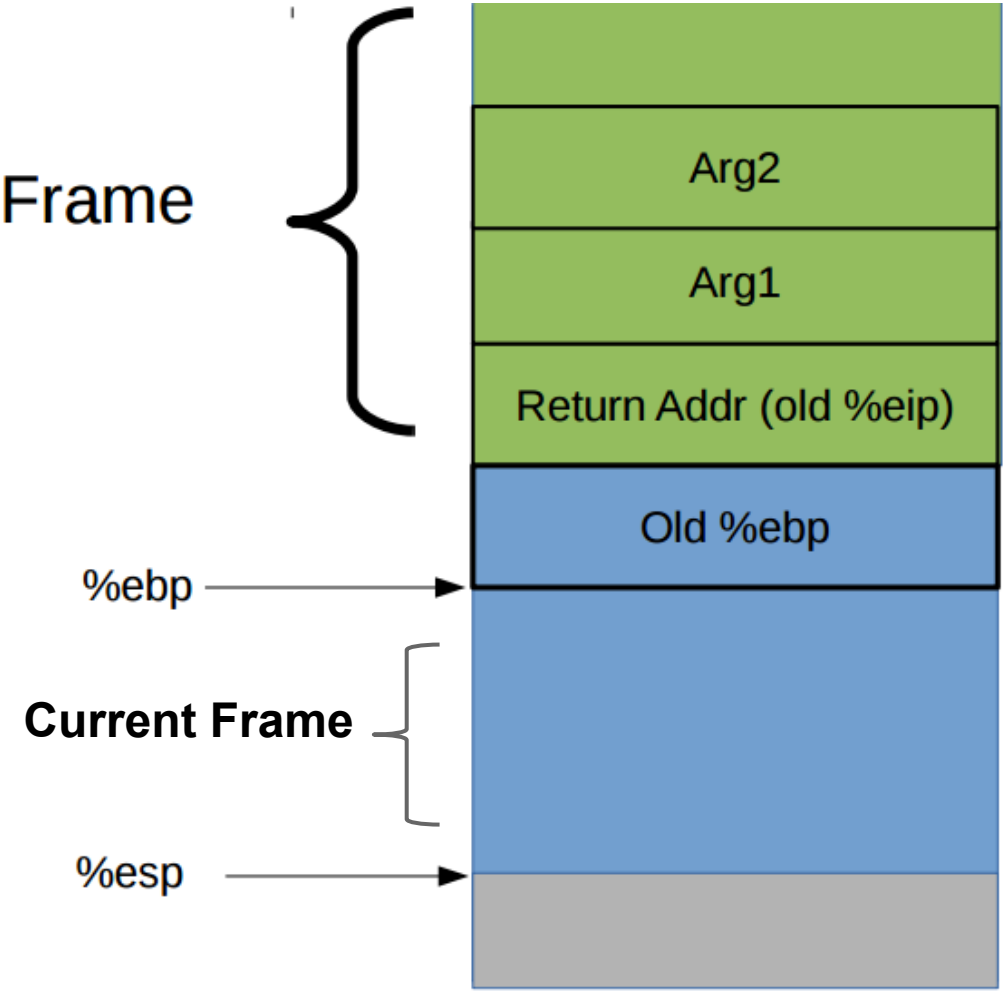
The Stack

- How do we create frame abstractions for procedures?



Stack Frames

- %ebp
  - base pointer
  - bottom of frame
- %eip
  - instruction pointer



x86 Calling Conventions

- **Caller** saved registers
  - pushed to the stack before function call is made
  - restored after the callee exits
  - %eax, %ecx, %edx
  - arguments: $0x8(%ebp), %0xc(%ebp), %0x10(%ebp) ...
- **Callee** saved registers
  - pushed to the stack at the start of the function
  - restored before the callee exits
  - %ebx, %edi, %esi
  - return value in %eax

Even More Assembly

- call <label>
  - pushl %eip
  - movl <label (function address)> %eip
- leave
  - movl %ebp, %esp
  - popl %ebp
- ret
  - popl %eip

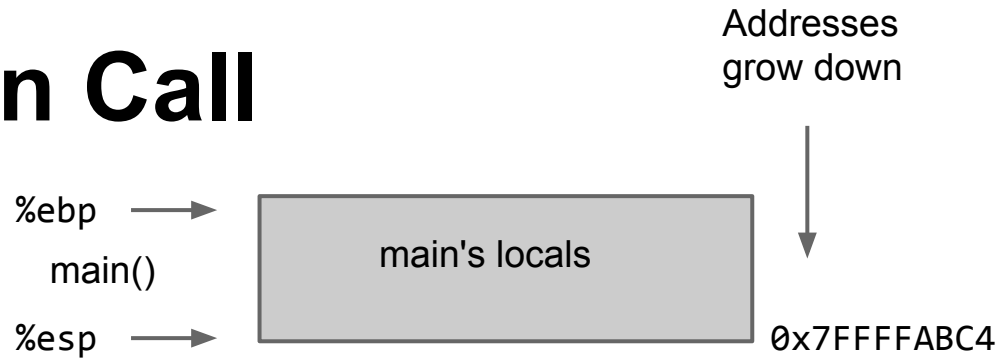
Example: Function Call

```
int f() {
    int x = 42;
    int foo = g(x);
    return foo + 5;
}
int g(int num) {
    return num+10;
}
```

```
In main():
    f();
```

Example: Function Call

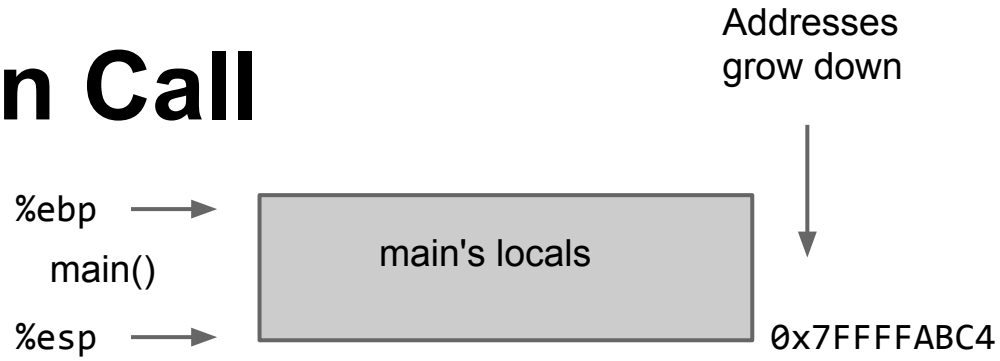
```
int f() {
    int x = 42;
    int foo = g(x);
    return foo + 5;
}
int g(int num) {
    return num+10;
}
```



In main(), about to call f().

Example: Function Call

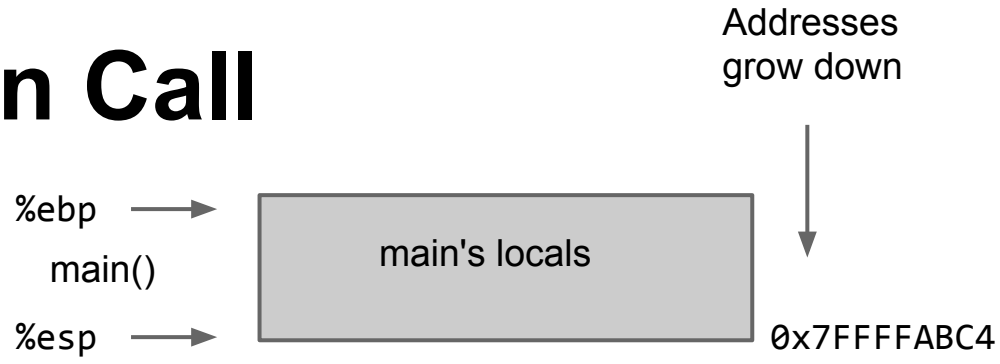
```
int f() {
    int x = 42;
    int foo = g(x);
    return foo + 5;
}
int g(int num) {
    return num+10;
}
```



In main(), about to call f().
1. Prepare arguments to f().

Example: Function Call

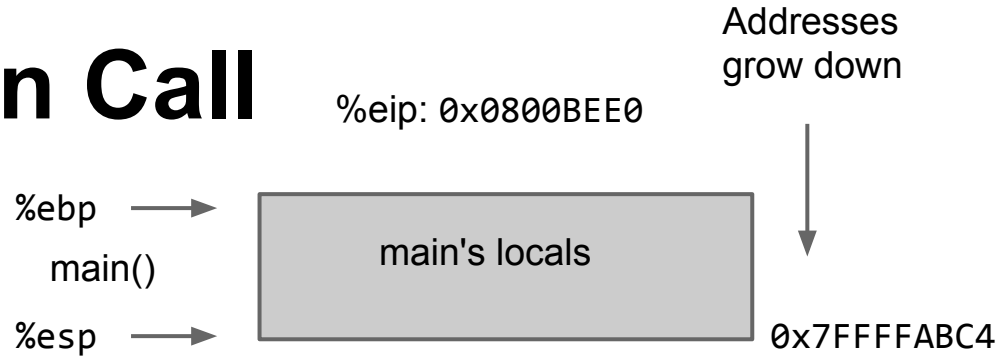
```
int f() {
    int x = 42;
    int foo = g(x);
    return foo + 5;
}
int g(int num) {
    return num+10;
}
```



In main(), about to call f().
1. Prepare arguments to f(). **No arguments!**

Example: Function Call

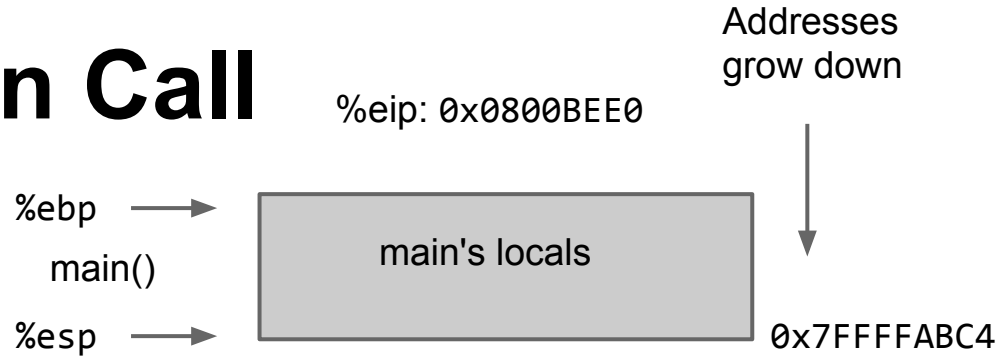
```
int f() {
    int x = 42;
    int foo = g(x);
    return foo + 5;
}
int g(int num) {
    return num+10;
}
```



In main(), about to call f().
1. Prepare arguments to f().
2. Call f()

Example: Function Call

```
int f() {
    int x = 42;
    int foo = g(x);
    return foo + 5;
}
int g(int num) {
    return num+10;
}
```



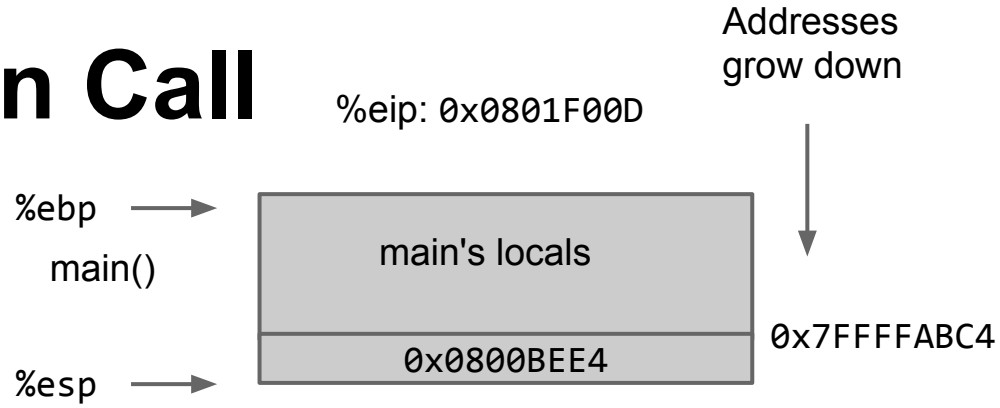
Assume instruction after call to g(x) is at location: `0x0800BEE4`
Assume first instruction of g is at location: `0x0800F00D`

In main(), about to call f().
1. Prepare arguments to f().
2. Call f()

```
call f
```

Example: Function Call

```
int f() {
    int x = 42;
    int foo = g(x);
    return foo + 5;
}
int g(int num) {
    return num+10;
}
```

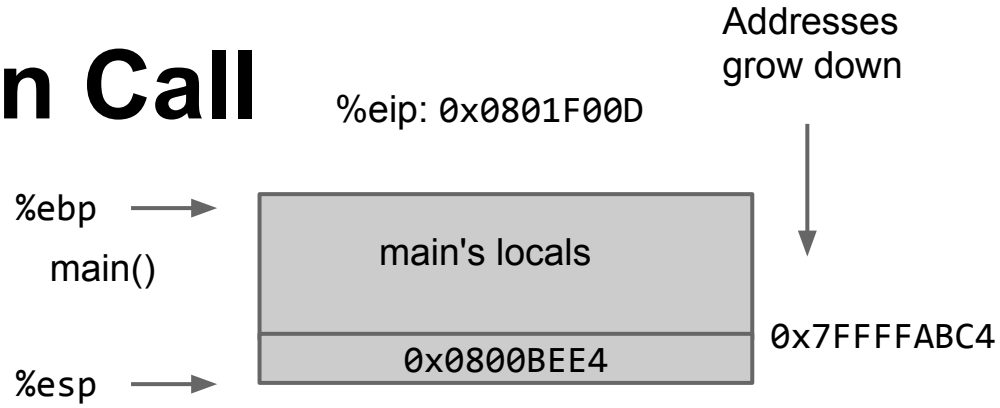


In main(), about to call f().
1. Prepare arguments to f().
2. Call f()

```
call f
```

Example: Function Call

```
int f() {
    int x = 42;
    int foo = g(x);
    return foo + 5;
}
int g(int num) {
    return num+10;
}
```



In main(), about to call f().
1. Prepare arguments to f().
2. Call f()
3. f() is now "in control"

```
call f
```

Example: Function Call

```
int f() {
    int x = 42;
    int foo = g(x);
    return foo + 5;
}
int g(int num) {
    return num+10;
}
```

%ebp = 0x7FFFFACE

f() -- setup stack frame
1. Save caller's %ebp

```
pushl %ebp
```

Example: Function Call

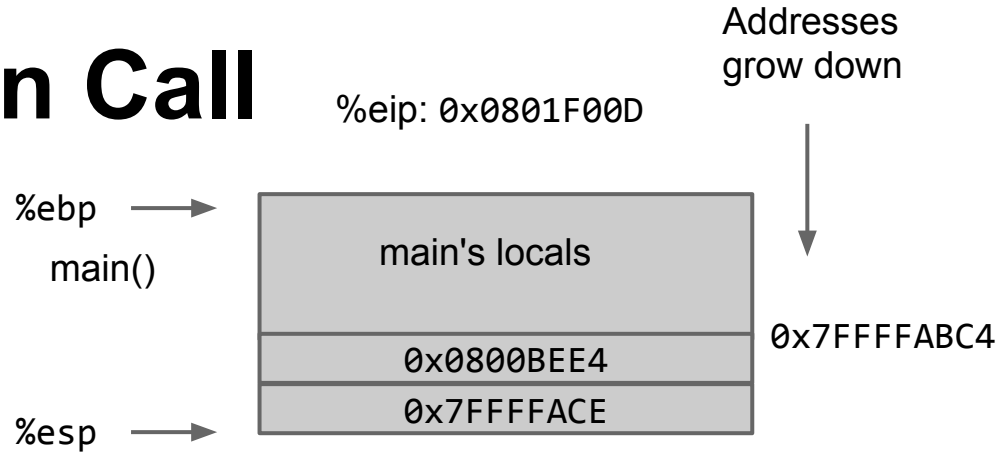
```
int f() {
    int x = 42;
    int foo = g(x);
    return foo + 5;
}
int g(int num) {
    return num+10;
}
```

f() -- setup stack frame
1. Save caller's %ebp

```
pushl %ebp
```

Example: Function Call

```
int f() {
    int x = 42;
    int foo = g(x);
    return foo + 5;
}
int g(int num) {
    return num+10;
}
```



f() -- setup stack frame
1. Save caller's %ebp
2. Update %ebp to point to *my* frame base.

```
movl %esp %ebp
```

Example: Function Call

```
int f() {
    int x = 42;
    int foo = g(x);
    return foo + 5;
}
int g(int num) {
    return num+10;
}
```

f() -- setup stack frame
1. Save caller's %ebp
2. Update %ebp to point to *my* frame base.

```
movl %esp %ebp
```

Example: Function Call

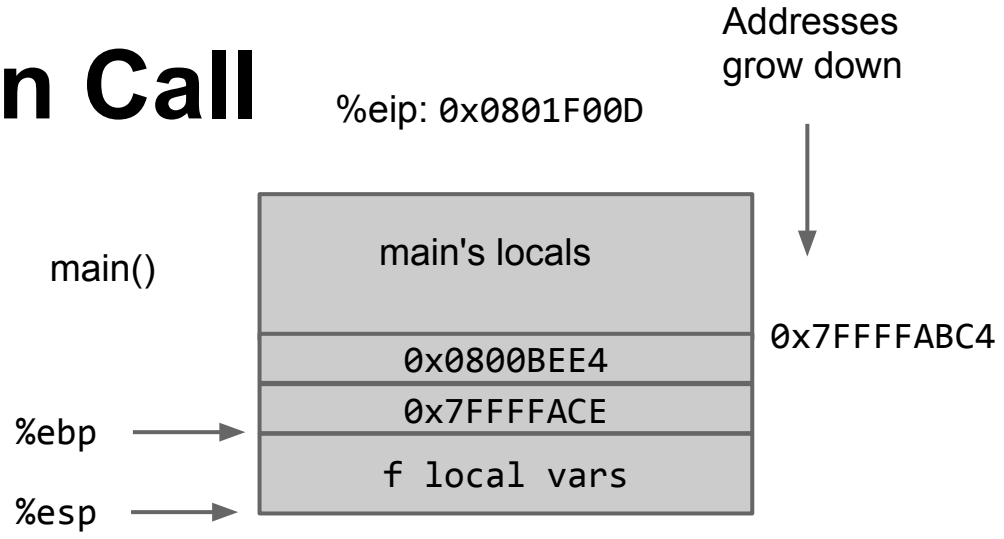
```
int f() {
    int x = 42;
    int foo = g(x);
    return foo + 5;
}
int g(int num) {
    return num+10;
}
```

f() -- setup stack frame
1. Save caller's %ebp
2. Update %ebp to point to *my* frame base.
3. Allocate space for local variables

```
subl $8 %esp
```

Example: Function Call

```
int f() {
    int x = 42;
    int foo = g(x);
    return foo + 5;
}
int g(int num) {
    return num+10;
}
```



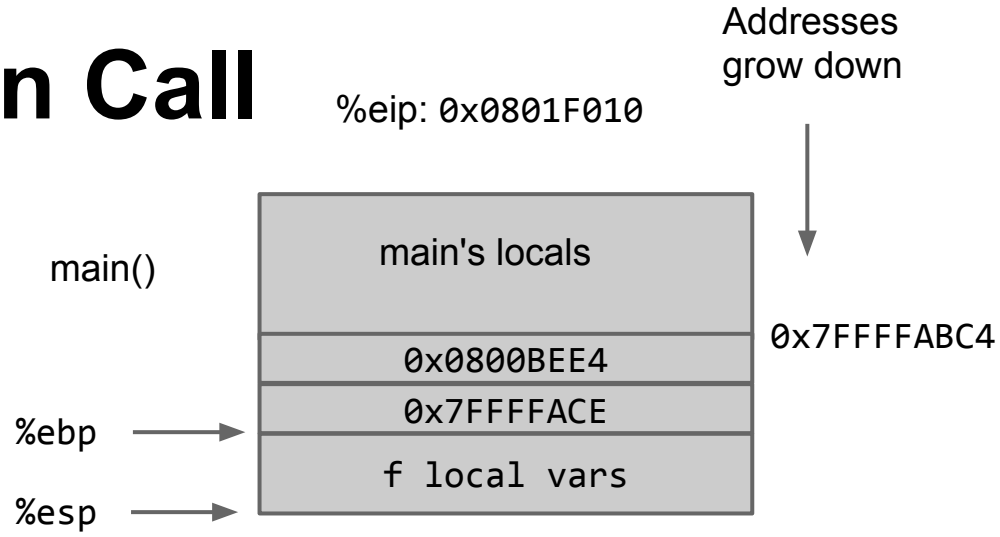
*Note: Depending on how clever the compiler is, it may not allocate stack space for *all* local vars if it can do it with registers.*

f() -- setup stack frame
1. Save caller's %ebp
2. Update %ebp to point to *my* frame base.
3. Allocate space for local variables

```
subl $8 %esp
```

Example: Function Call

```
int f() {
    int x = 42;
    int foo = g(x);
    return foo + 5;
}
int g(int num) {
    return num+10;
}
```

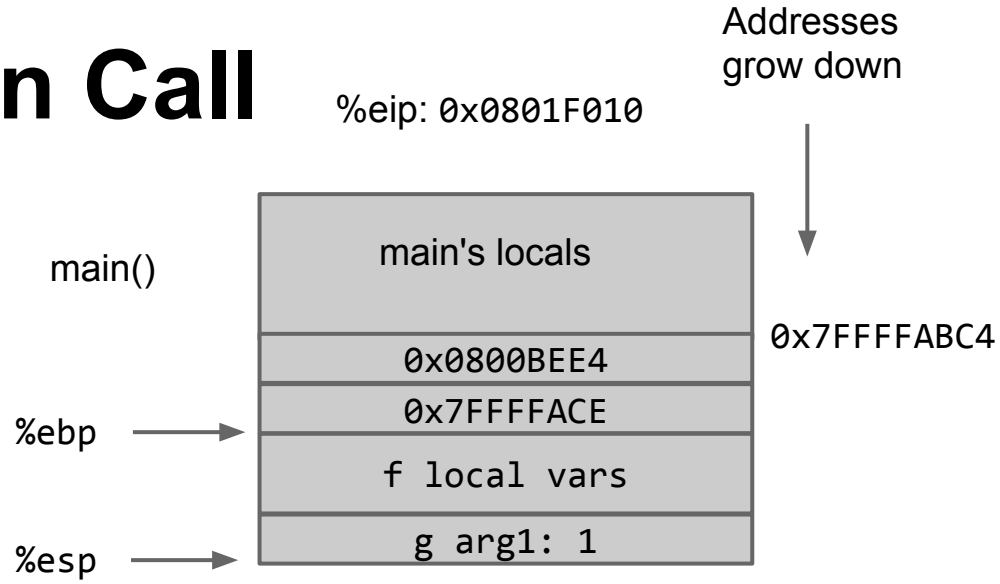


f() executes its code, and is about to call g().
1. Prepare arguments to g

```
pushl $eax
```

Example: Function Call

```
int f() {
    int x = 42;
    int foo = g(x,1);
    return foo + 5;
}
int g(int n, int a) {
    return num+10;
}
```

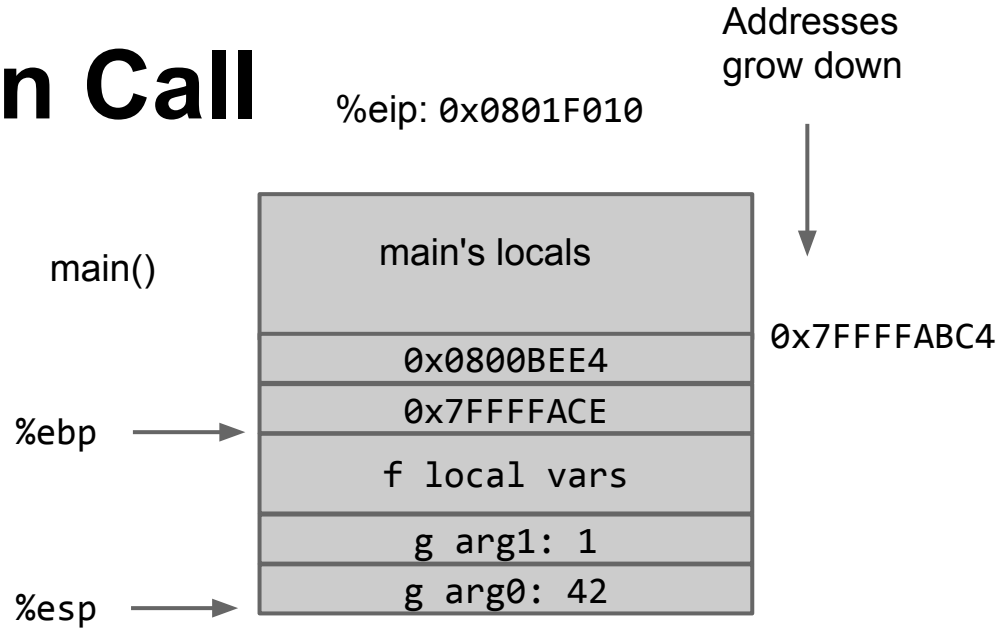


f() executes its code, and is about to call g().
1. Prepare arguments to g

```
pushl 1
pushl $eax
```

Example: Function Call

```
int f() {
    int x = 42;
    int foo = g(x,1);
    return foo + 5;
}
int g(int n, int a) {
    return num+10;
}
```



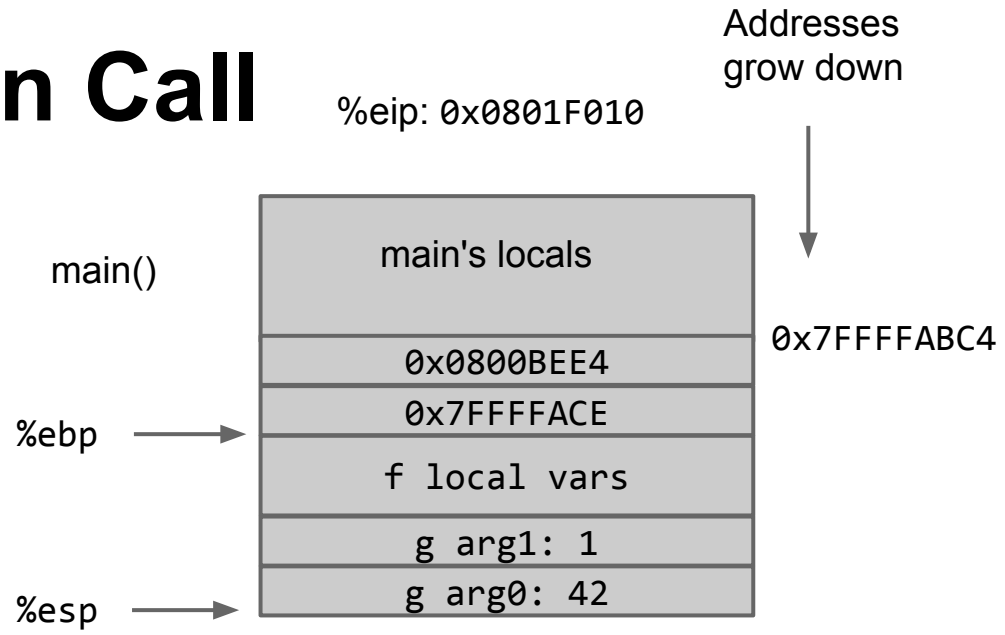
f() executes its code, and is about to call g().
1. Prepare arguments to g

```
pushl 1
pushl $eax
```

Example: Function Call

```
int f() {
    int x = 42;
    int foo = g(x,1);
    return foo + 5;
}
int g(int n, int a) {
    return num+10;
}
```

Assume f's next instruction is at `0x0801F014`



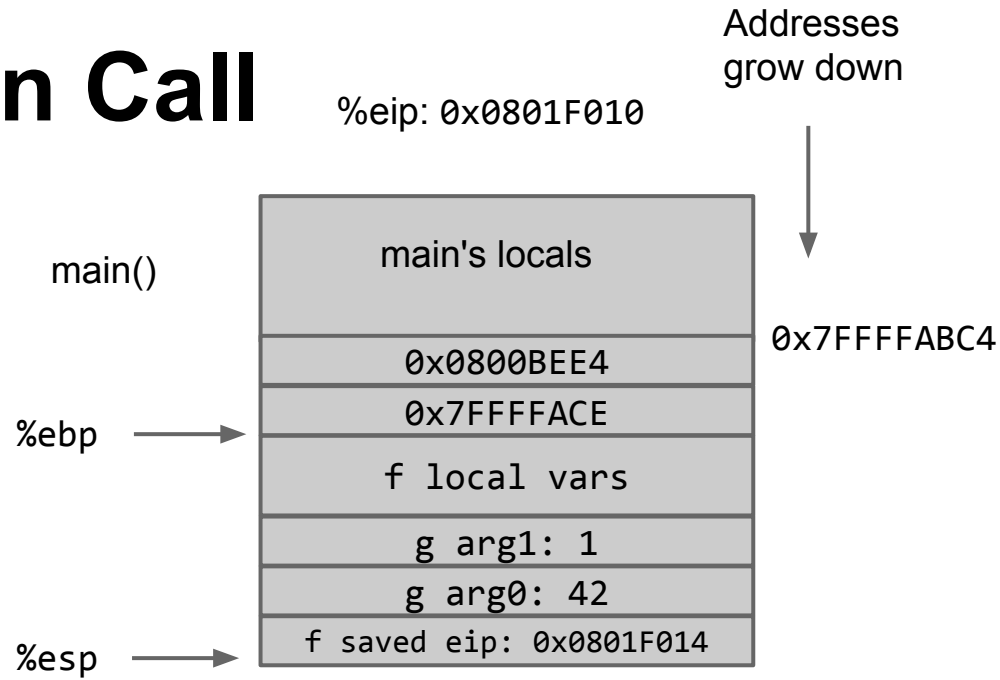
f() executes its code, and is about to call g().
1. Prepare arguments to g
2. Call g (saves %eip).

```
call g
```

Example: Function Call

```
int f() {
    int x = 42;
    int foo = g(x,1);
    return foo + 5;
}
int g(int n, int a) {
    return num+10;
}
```

Assume f's next instruction is at `0x0801F014`



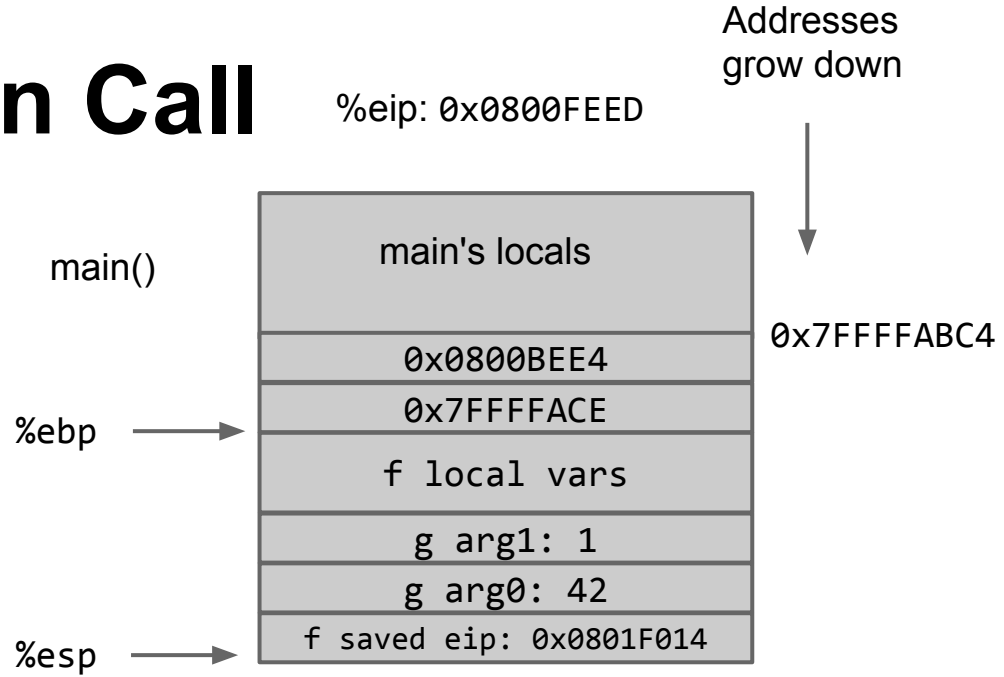
f() executes its code, and is about to call g().
1. Prepare arguments to g
2. Call g (saves %eip).

`call g`

Example: Function Call

```
int f() {
    int x = 42;
    int foo = g(x,1);
    return foo + 5;
}
int g(int n, int a) {
    return num+10;
}
```

Assume f's next instruction is at `0x0801F014`



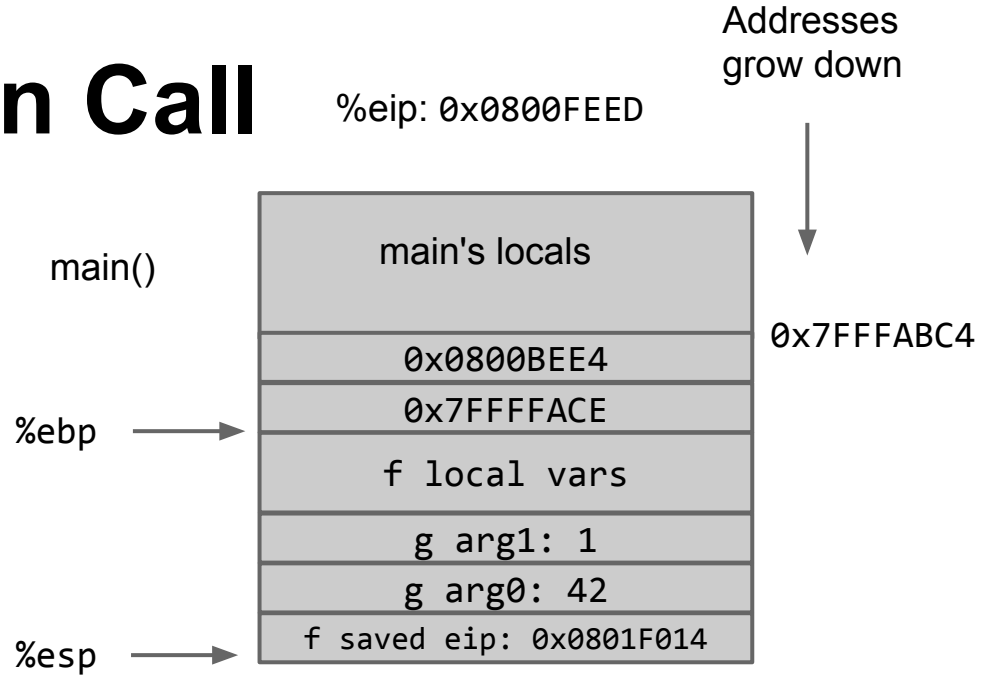
f() executes its code, and is about to call g().

1. Prepare arguments to g
2. Call g (saves %eip).
3. g is in control now!

`call g`

Example: Function Call

```
int f() {
    int x = 42;
    int foo = g(x,1);
    return foo + 5;
}
int g(int n, int a) {
    return num+10;
}
```



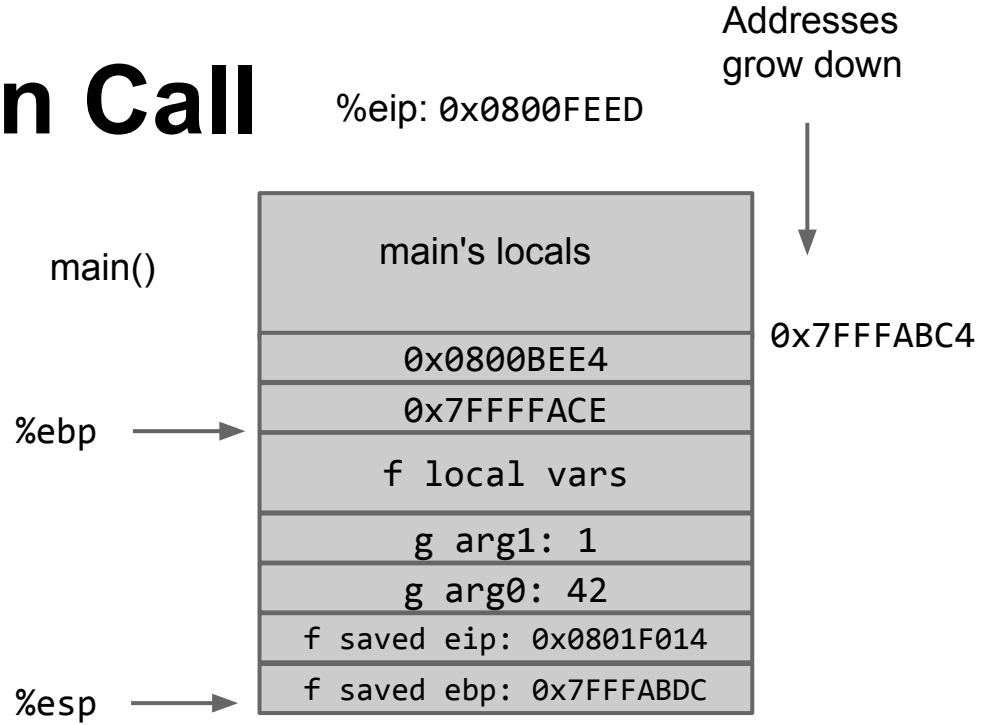
g must setup its stack frame.

1. Save caller's %ebp.

```
pushl %ebp
```

Example: Function Call

```
int f() {
    int x = 42;
    int foo = g(x,1);
    return foo + 5;
}
int g(int n, int a) {
    return num+10;
}
```



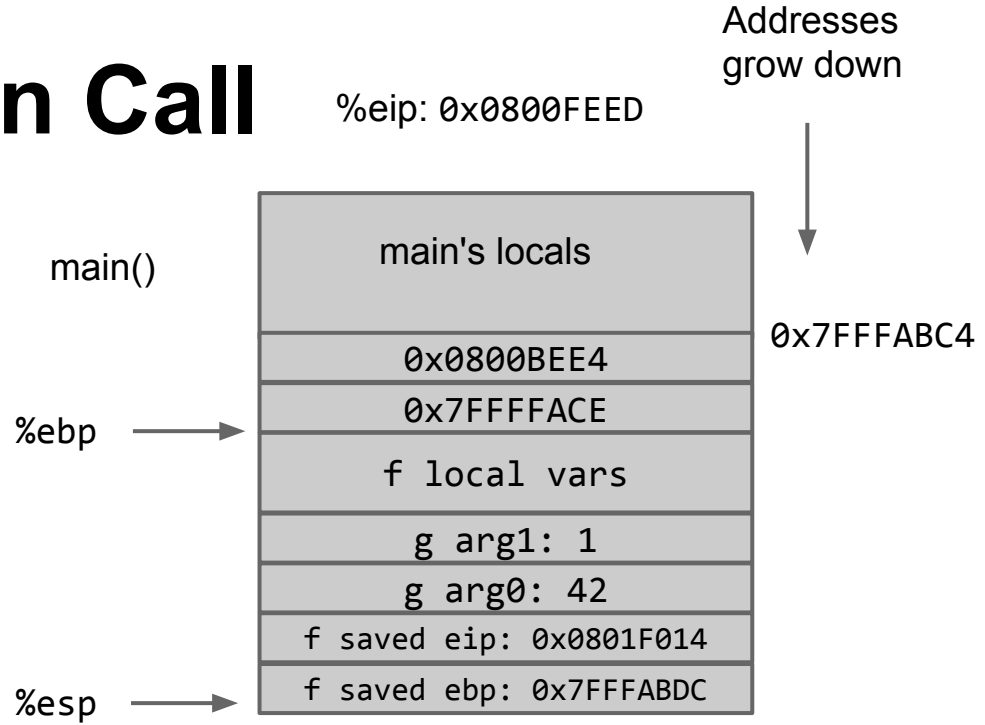
g must setup its stack frame.

1. Save caller's %ebp.

```
pushl %ebp
```

Example: Function Call

```
int f() {
    int x = 42;
    int foo = g(x,1);
    return foo + 5;
}
int g(int n, int a) {
    return num+10;
}
```

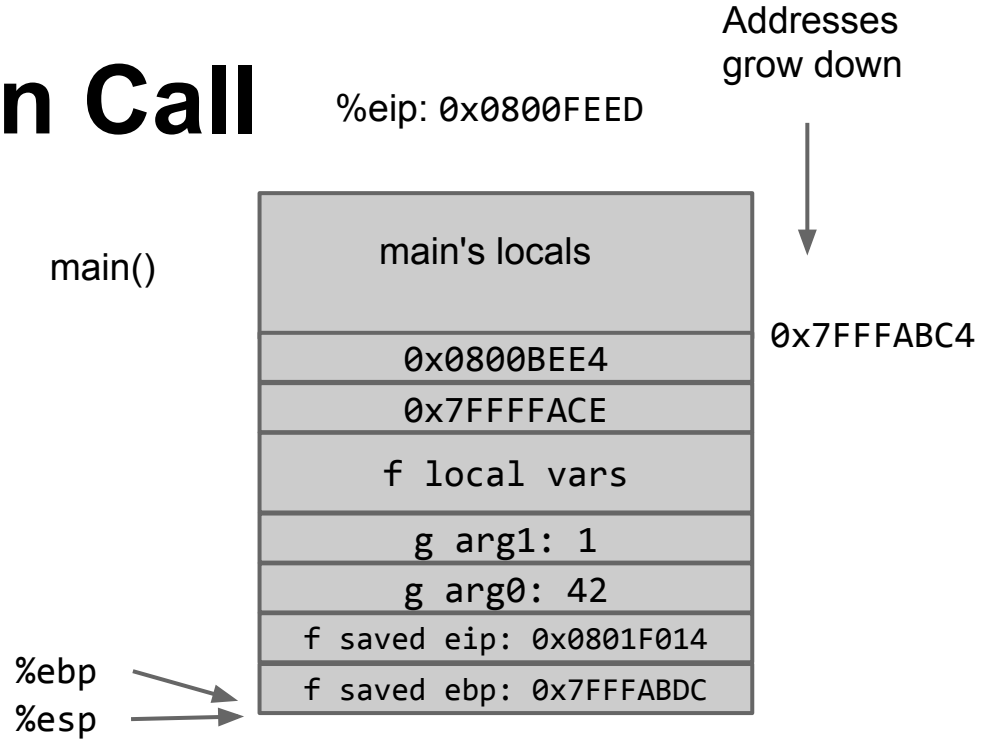


g must setup its stack frame.
1. Save caller's %ebp.
2. Update %ebp to point to *my* frame base.

```
movl %esp %ebp
```

Example: Function Call

```
int f() {
    int x = 42;
    int foo = g(x,1);
    return foo + 5;
}
int g(int n, int a) {
    return num+10;
}
```

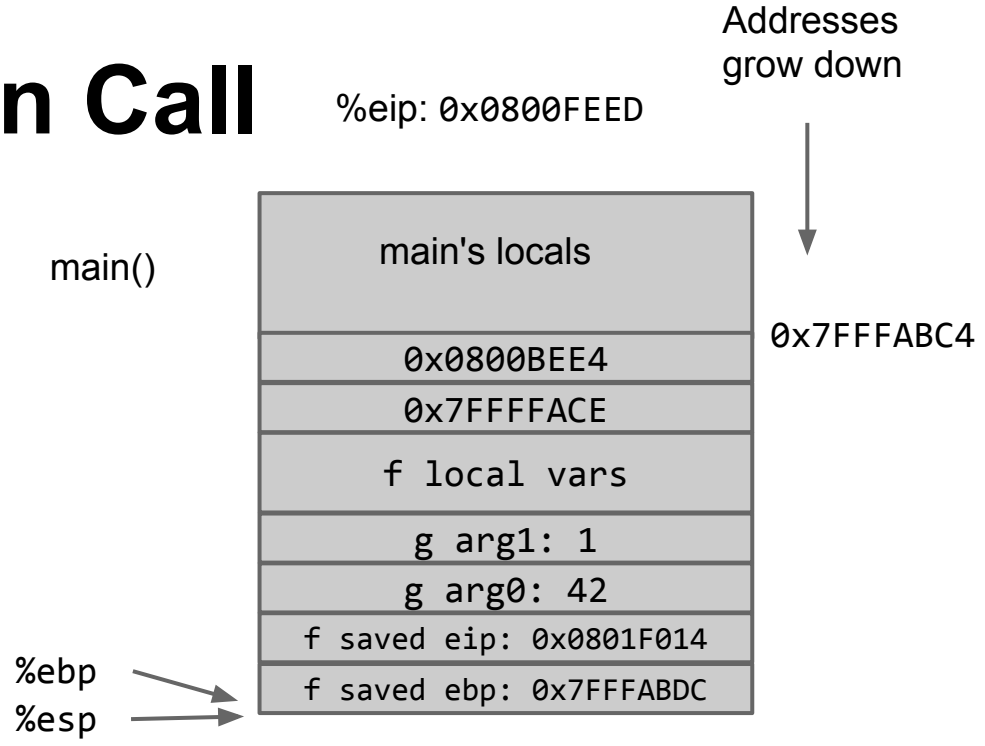


g must setup its stack frame.
1. Save caller's %ebp.
2. Update %ebp to point to *my* frame base.

```
movl %esp %ebp
```

Example: Function Call

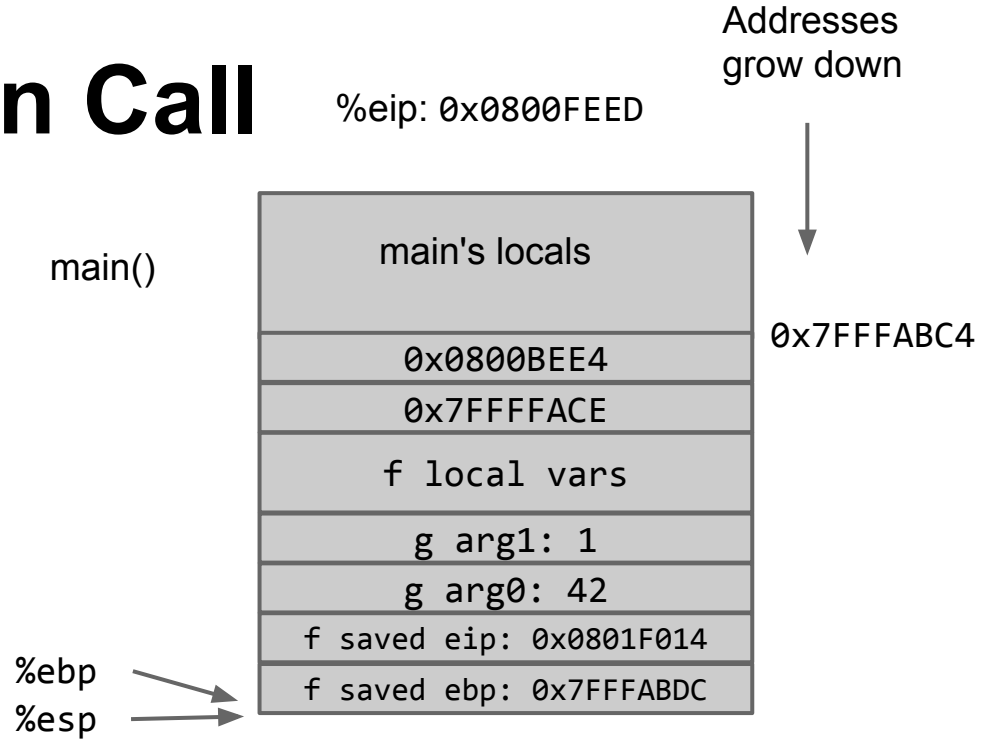
```
int f() {
    int x = 42;
    int foo = g(x,1);
    return foo + 5;
}
int g(int n, int a) {
    return num+10;
}
```



g must setup its stack frame.
1. Save caller's %ebp.
2. Update %ebp to point to *my* frame base.
3. Allocate space for local vars.

Example: Function Call

```
int f() {
    int x = 42;
    int foo = g(x,1);
    return foo + 5;
}
int g(int n, int a) {
    return n+10;
}
```

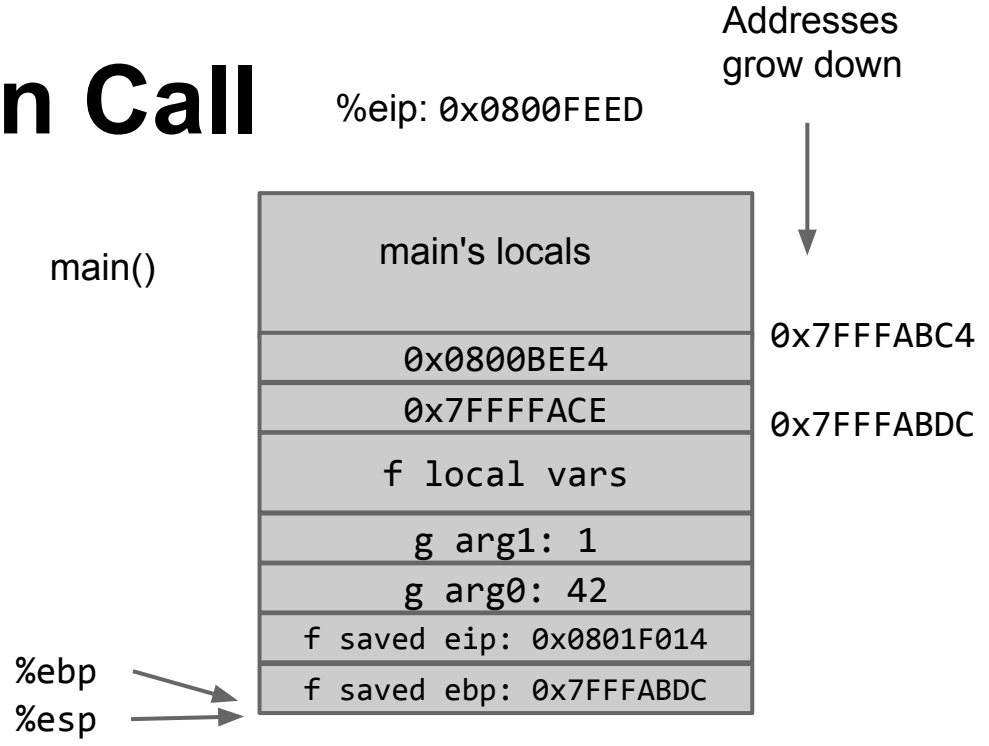


g must setup its stack frame.
1. Save caller's %ebp.
2. Update %ebp to point to *my* frame base.
3. Allocate space for local vars.

No local vars!

Example: Function Call

```
int f() {
    int x = 42;
    int foo = g(x,1);
    return foo + 5;
}
int g(int n, int a) {
    return n+10;
}
```



g() finishes, now must return.
1. Set %esp to caller's original %esp.
2. Set %ebp to caller's original %ebp.

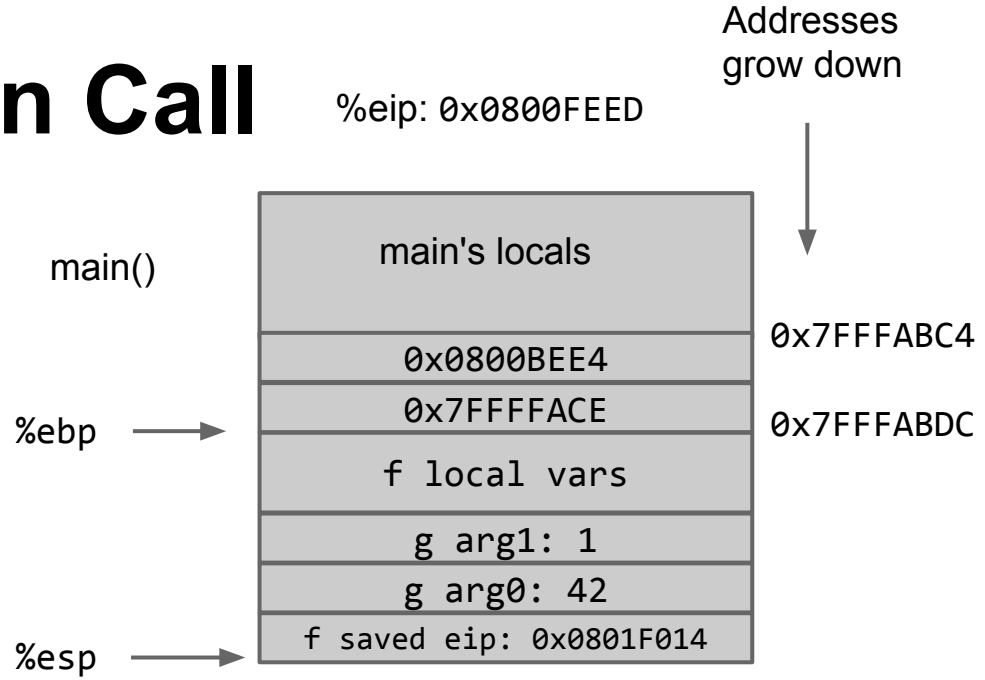
This is %ebp!

```
movl %ebp %esp

popl %ebp
```

Example: Function Call

```
int f() {
    int x = 42;
    int foo = g(x,1);
    return foo + 5;
}
int g(int n, int a) {
    return n+10;
}
```



g() finishes, now must return.
1. Set %esp to caller's original %esp.
2. Set %ebp to caller's original %ebp.

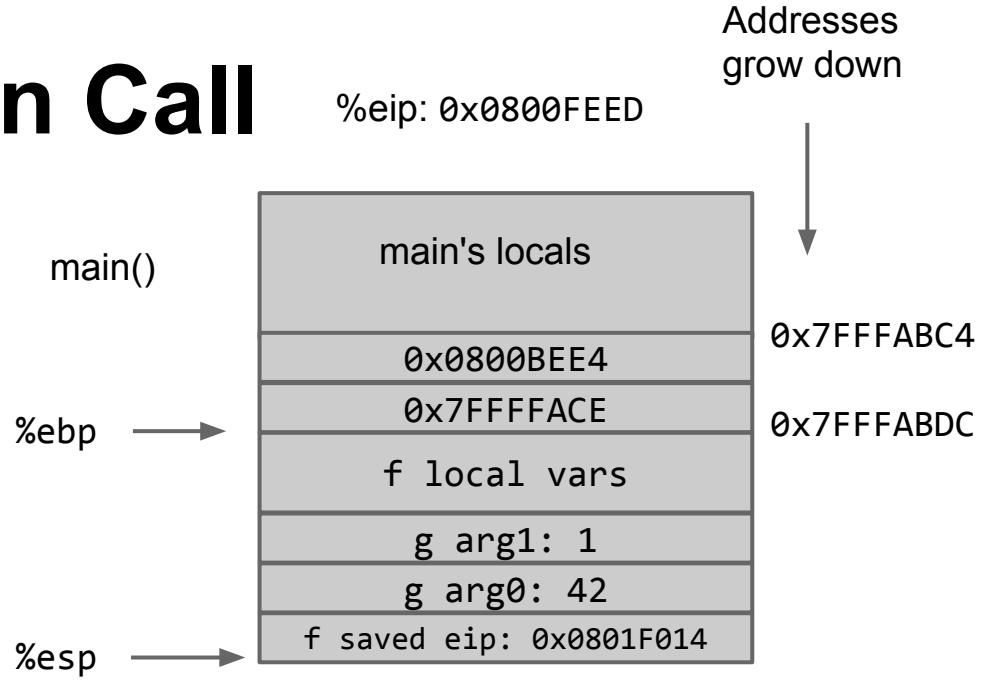
This is %ebp!

```
movl %ebp %esp

popl %ebp
```

Example: Function Call

```
int f() {
    int x = 42;
    int foo = g(x,1);
    return foo + 5;
}
int g(int n, int a) {
    return n+10;
}
```



g() finishes, now must return.
1. Set %esp to caller's original %esp.
2. Set %ebp to caller's original %ebp.
3. Return to caller.

Pops top of stack, places value into %eip.

```
ret
```

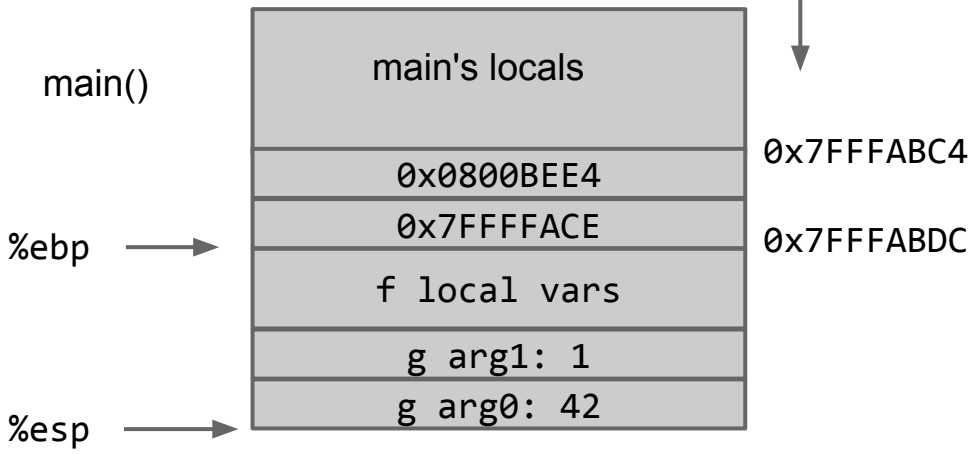
Example: Function Call

```
int f() {
    int x = 42;
    int foo = g(x,1);
    return foo + 5;
}
int g(int n, int a) {
    return n+10;
}
```

f in control now!

%eip: 0x0801F014

Addresses grow down



g() finishes, now must return.

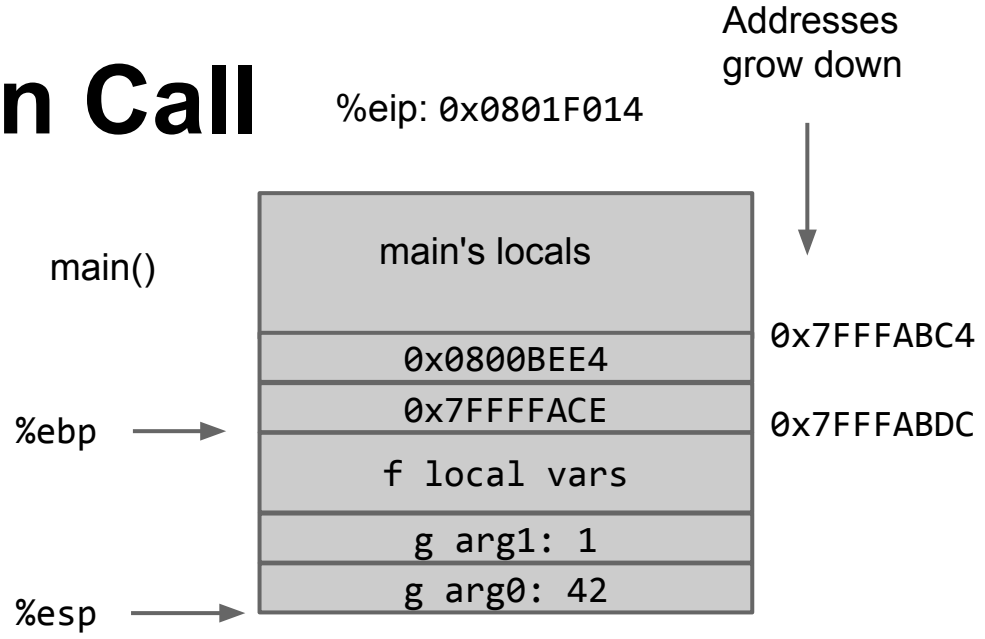
1. Set %esp to caller's original %esp.
2. Set %ebp to caller's original %ebp.
3. Return to caller.

Pops top of stack, places value into %eip.

```
ret
```

Example: Function Call

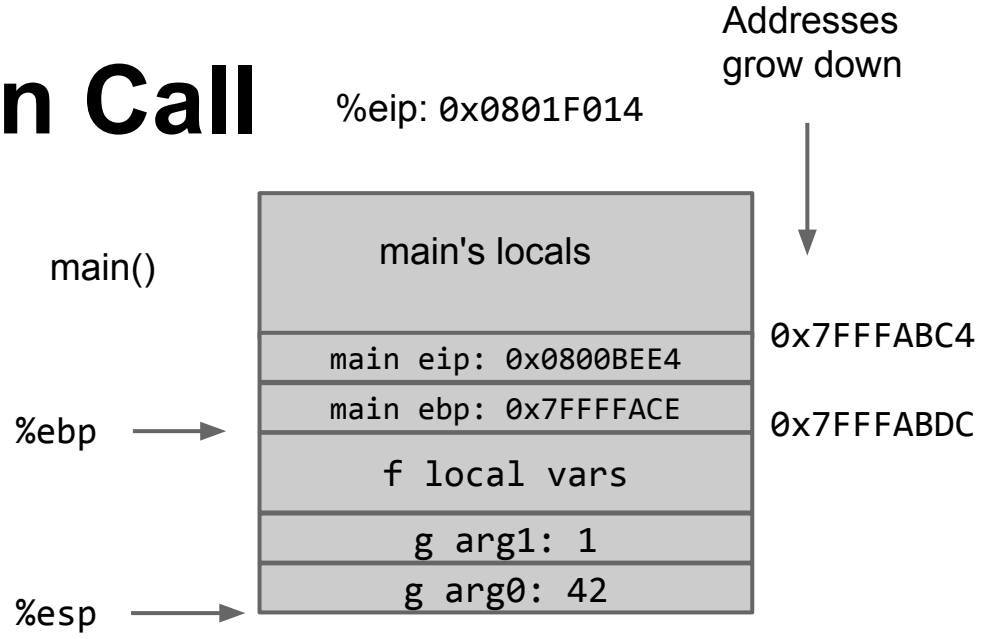
```
int f() {
    int x = 42;
    int foo = g(x,1);
    return foo + 5;
}
int g(int n, int a) {
    return n+10;
}
```



f() resumes executing.

Example: Function Call

```
int f() {
    int x = 42;
    int foo = g(x,1);
    return foo + 5;
}
int g(int n, int a) {
    return n+10;
}
```

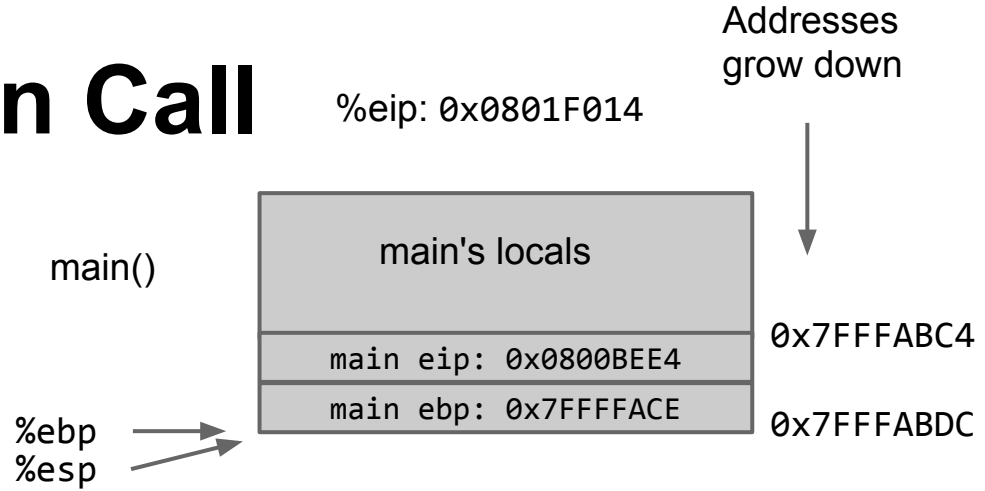


f() is ready to return.

1. Update %esp to caller's %esp. `movl %ebp %esp`

Example: Function Call

```
int f() {
    int x = 42;
    int foo = g(x,1);
    return foo + 5;
}
int g(int n, int a) {
    return n+10;
}
```

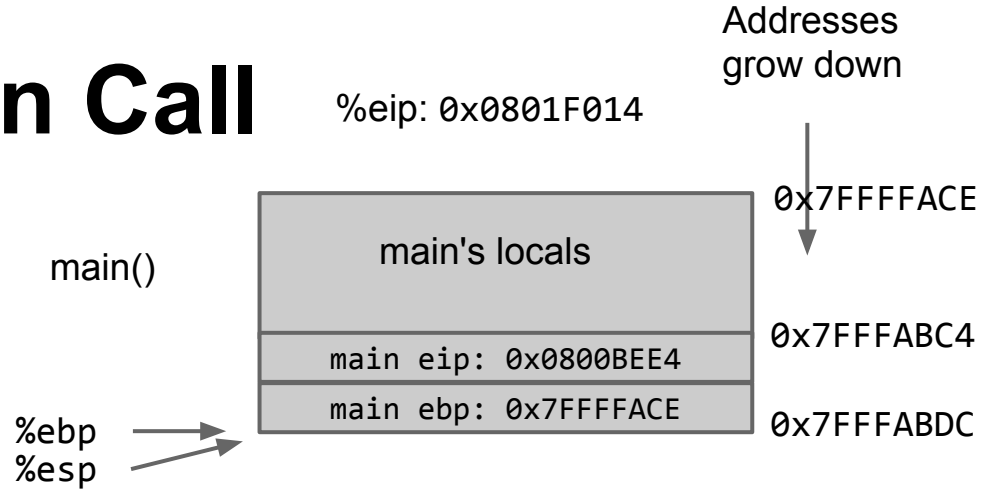


f() is ready to return.

1. Update %esp to caller's %esp. `movl %ebp %esp`

Example: Function Call

```
int f() {
    int x = 42;
    int foo = g(x,1);
    return foo + 5;
}
int g(int n, int a) {
    return n+10;
}
```



f() is ready to return.
1. Update %esp to caller's %esp.
2. Update %ebp to caller's %ebp.

```
popl %ebp
```

Example: Function Call

```
int f() {
    int x = 42;
    int foo = g(x,1);
    return foo + 5;
}
int g(int n, int a) {
    return n+10;
}
```

f() is ready to return.
1. Update %esp to caller's %esp.
2. Update %ebp to caller's %ebp.

```
popl %ebp
```

Example: Function Call

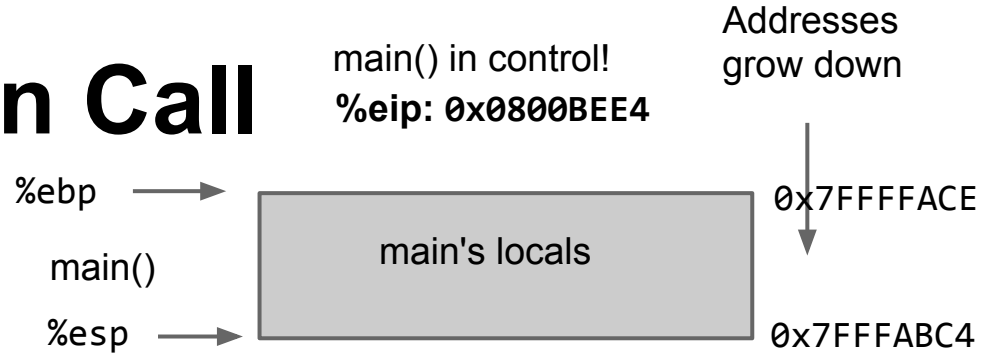
```
int f() {
    int x = 42;
    int foo = g(x,1);
    return foo + 5;
}
int g(int n, int a) {
    return n+10;
}
```

f() is ready to return.
1. Update %esp to caller's %esp.
2. Update %ebp to caller's %ebp.
3. Return control to caller.

ret

Example: Function Call

```
int f() {
    int x = 42;
    int foo = g(x,1);
    return foo + 5;
}
int g(int n, int a) {
    return n+10;
}
```



f() is ready to return.
1. Update %esp to caller's %esp.
2. Update %ebp to caller's %ebp.
3. Return control to caller.

ret

Example: Function Call

```
int f() {
    int x = 42;
    int foo = g(x,1);
    return foo + 5;
}
int g(int n, int a) {
    return n+10;
}
```

main() in control!
%eip: 0x0800BEE4

Addresses grow down

main() resumes executing where it left off, and finishes its awesome computation.

```
.main:
    ...
    call f
    # here now
    ...
```

Conditional Jumps

`je, jz` -- jump if equal/zero

`jne, jnz` -- jump if not-equal/not-zero

`jl, jle` -- jump if less than/less-than-or-equal

`jg, jge` -- jump if greater than/greater-than-or-equal

Several more jump types (ie overflow, sign, parity, etc.).

cmp

Use cmpl, testl to use the conditional jump!

```
cmpl %eax %edx
jge .L2
```

Jumps to .L2 if %eax <= %edx

testl

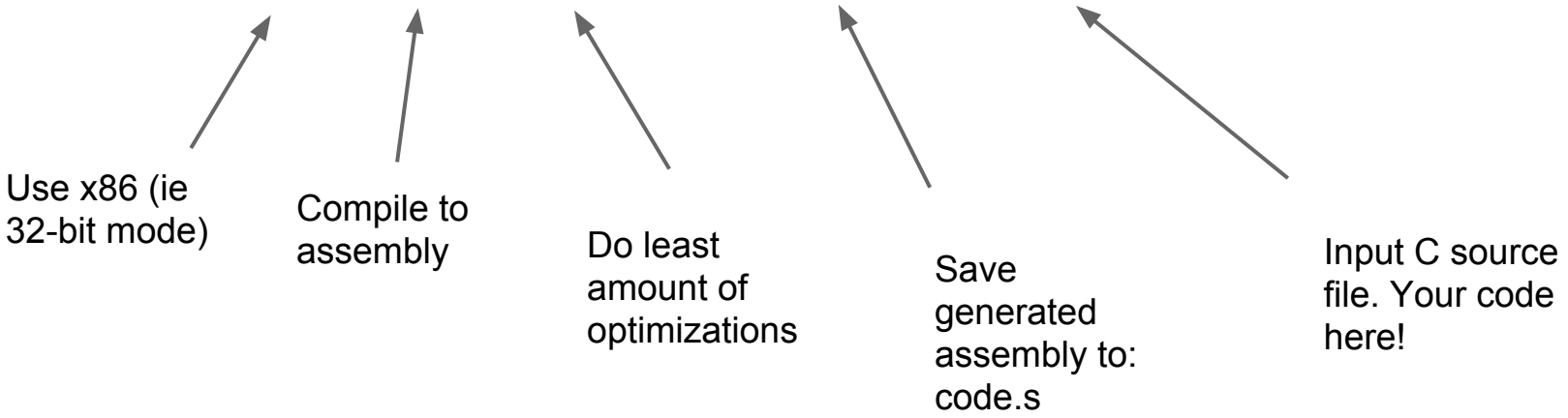
```
testl %eax, %eax
jz zeroLabel; jump if %eax is zero
js negLabel ; jump if EAX is negative
jns posLabel ; jump if EAX is positive
```

Quick way to check if a register is 0, negative, or positive.

Compiling at Home

Try creating assembly output yourself!

```
$ gcc -m32 -S -O0 -o code.s code.c
```



Compiling at Home

Add this flag to disable weird lines with .cfi_ junk:

```
$ gcc -m32 -S -O0 -fno-asynchronous-unwind-tables -o code.s code.c
```

Compiling at Home

Full pipeline to compile .c code -> .s -> executable.

```
# Compile: generates assembly from c code
gcc -S -m32 -O0 -fno-asynchronous-unwind-tables -o printint.s printint.c
# Assemble: generates object file from assembly
gcc -c -m32 -o printint.o printint.s
# Linker: generates executable from object file
gcc -m32 -o printint printint.o
```

Use last two commands to create executables of your own x86 code!

Midterm

- ~9 questions
- Open book
- Covers:
  - Integers: §1–§2.3, §2.5
  - Assembly: §3–§3.5, §3.13
  - Control: §3.6
  - Procedures: §3.7
  - Data structures: §3.8–§3.10
  - Pointers: §3.11, §3.12, §3.15
  - Lab 2 and HW3

The Bad News

- The midterm doesn’t exist yet
- You won’t find the answers in your book
  - “What if” questions
  - “No right answer” questions
    - “Explain your reasoning”
  - Multi-topic questions
- Essentially a timed take-home assignment (minus ability to google/stackoverflow)

Tips

- Practice problems from the book, labs
- Condense your notes to save time
- Think like the professor
  - Take note of tangents to the lecture material
  - Why would he choose one HW problem over another?
  - “Come up with more efficient calling conventions for performing multiple precision arithmetic”
  - “Count the number of transitions from 0 to 1 when counting the bits of an integer from right to left.”
- Bring a calculator

More Practice 

- Bit Manipulation:
  - 2.67, 2.73, 2.75
- Reverse Engineering Assembly:
  - 3.62, 3.63, 3.65, 3.66, 3.58
- Procedures x Data Structures:
  - 3.64