Introduction to Computer Organization

DIS 1A – Week 3

Slides modified from UT WANG

Agenda

- All about functions !
- Arrays
- Struct and Union
- Midterm tips

Exercise: Fun with arithmetic

(a) C code

```
int arith(int x,
          int y,
          int z)
{
    int t1 = x+y;
    int t2 = z*48;
    int t3 = t1 & 0xFFFF;
    int t4 = t2 * t3;
    return t4;
}
```

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

```
int arith(int x,
          int y,
          int z)
{
    int t1 = x+y;
    int t2 = z*48;
    int t3 = t1 & 0xFFFF;
    int t4 = t2 * t3;
    return t4;
}
```

(b) Assembly code

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

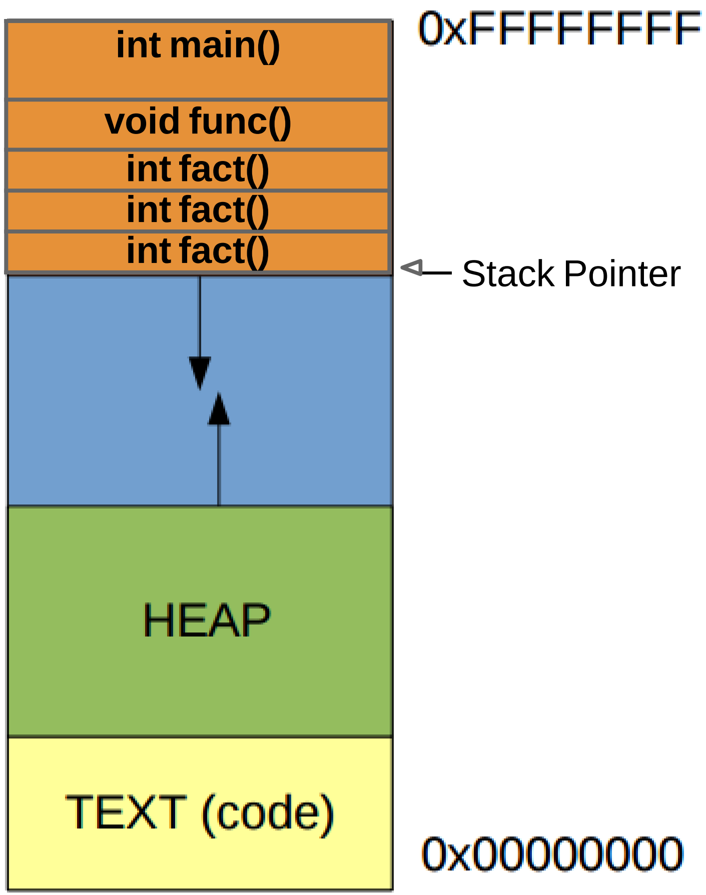
-All of a function's local variables are stored between %ebp and %esp

%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

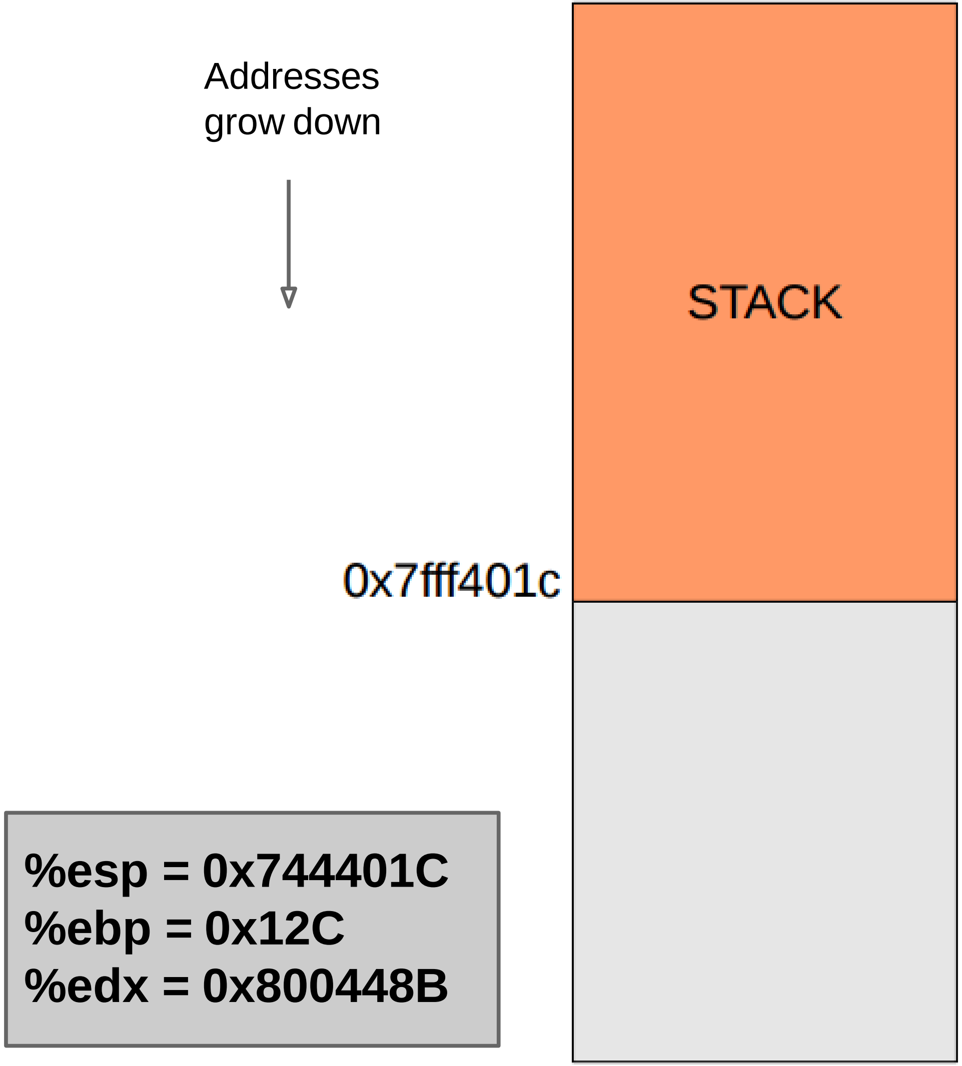
pushl, popl are convenience commands. Could simply use subl, movl,addl, etc.

The Stack

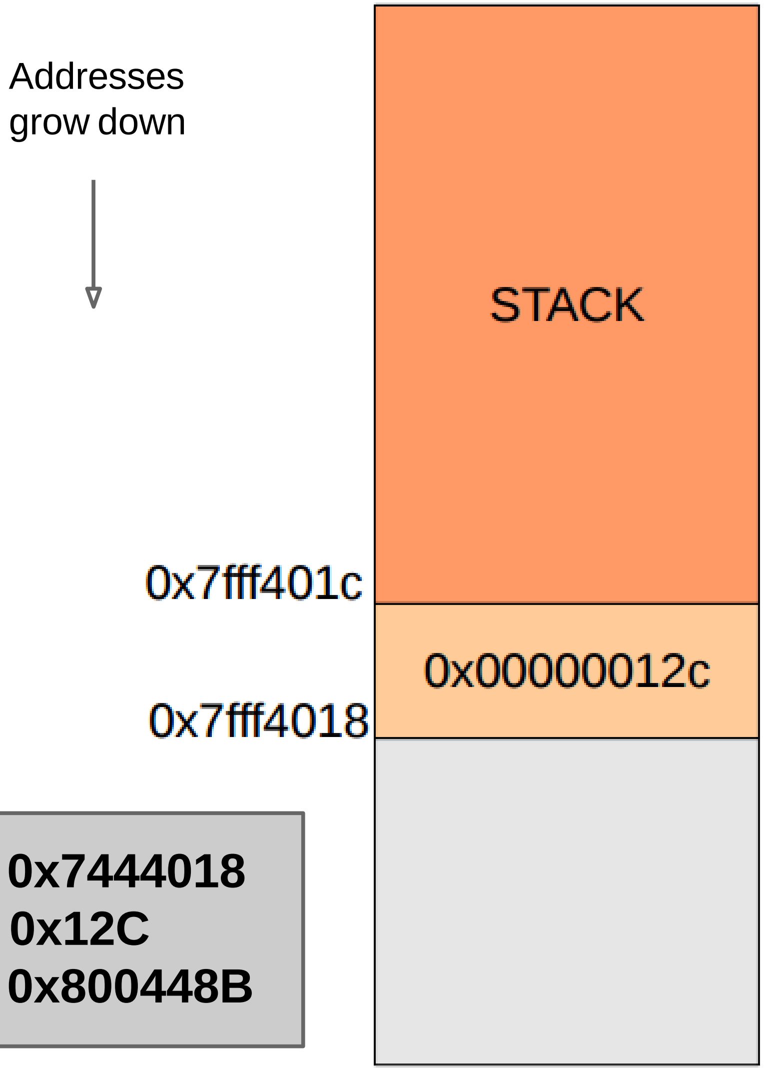
Consider the following stack.

What happens when I do:

    pushl %ebp



The Stack

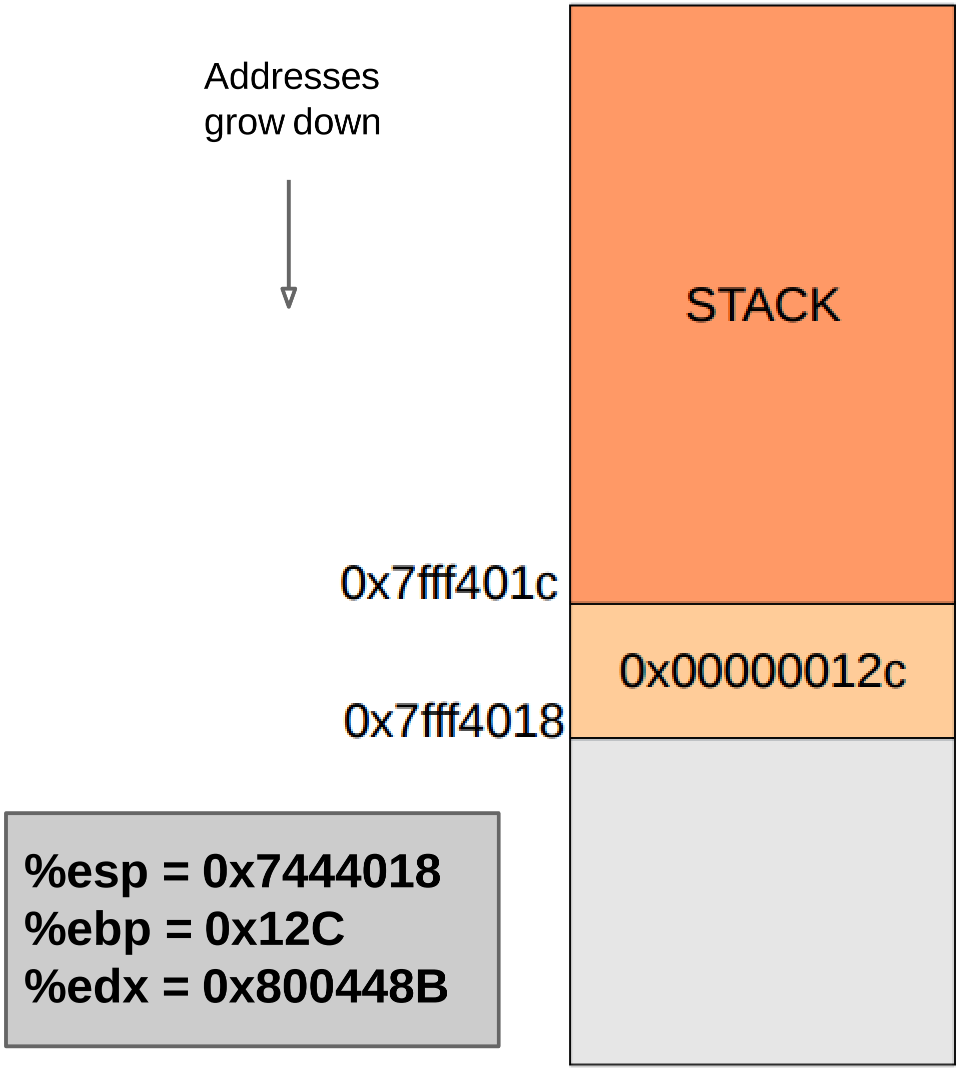


%esp = 0x7444018
%ebp = 0x12C
%edx = 0x800448B

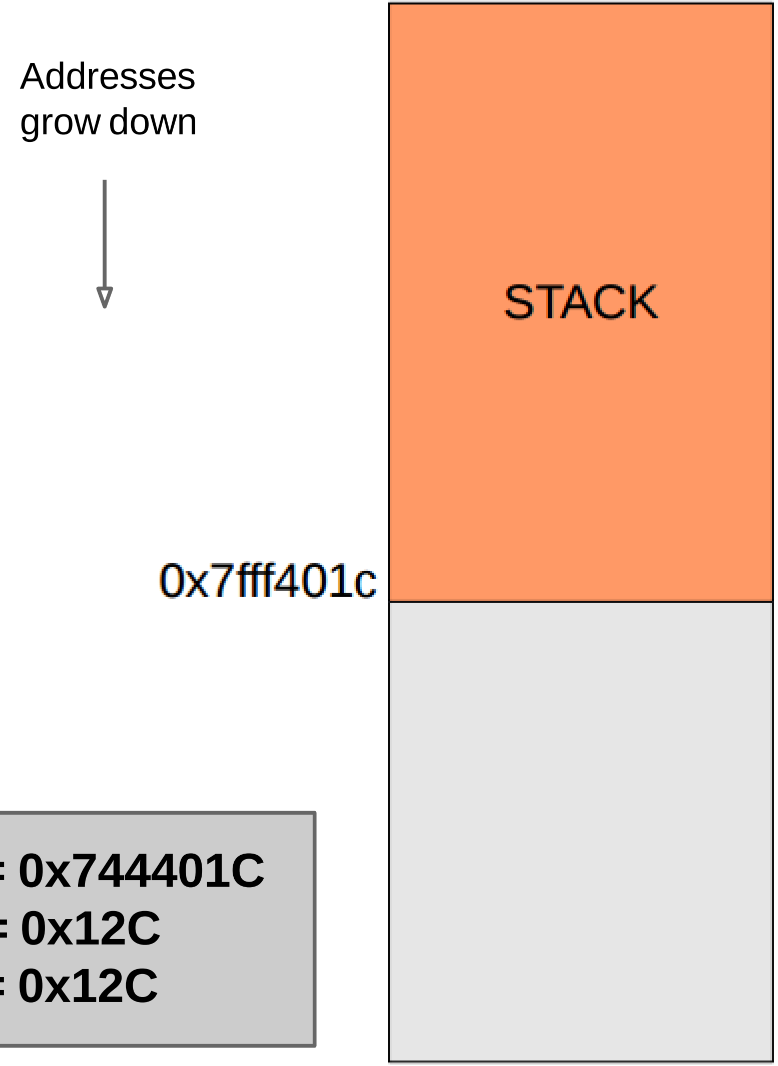
The Stack

What happens when I do:

popl %edx



The Stack

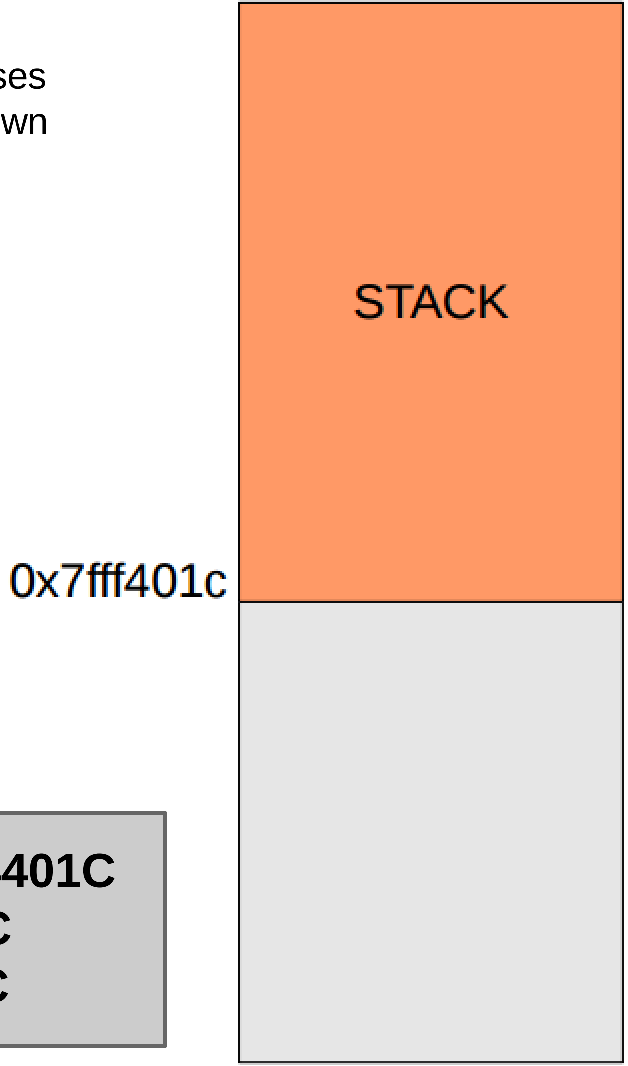


%esp = 0x744401C
%ebp = 0x12C
%edx = 0x12C

The Stack

- How do we create frame abstractions for procedures?

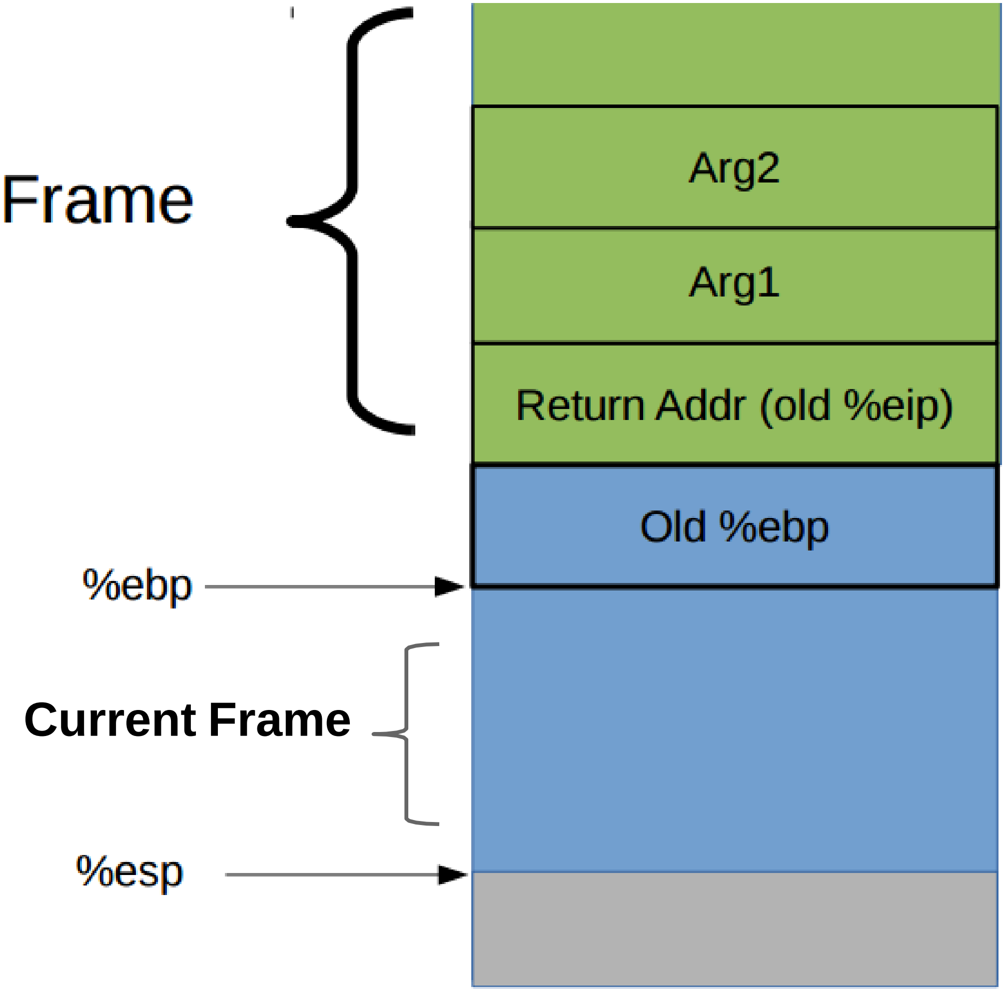
Addresses grow down



%esp = 0x744401C
%ebp = 0x12C
%edx = 0x12C

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
- **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

- ret
  - popl %eip

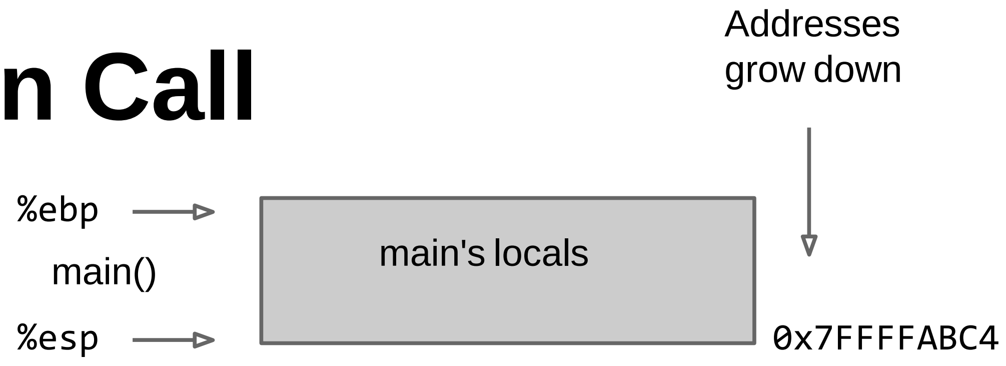
Example: Function Call

```
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

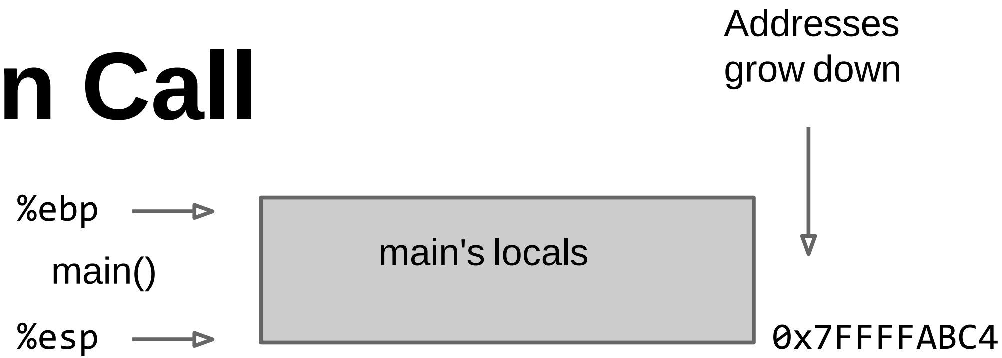
```
int f() {
    int x = 42;
    int foo = g(x);
    return foo + 5;
}
int g(int num) {
    return num+10;
}
```



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

Example: Function Call

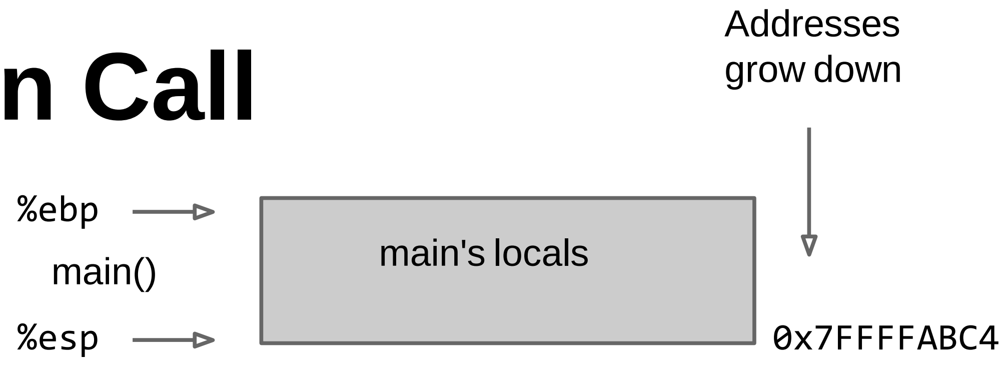
```
int f() {
    int x = 42;
    int foo = g(x);
    return foo + 5;
}
int g(int num) {
    return num+10;
}
```



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

Example: Function Call

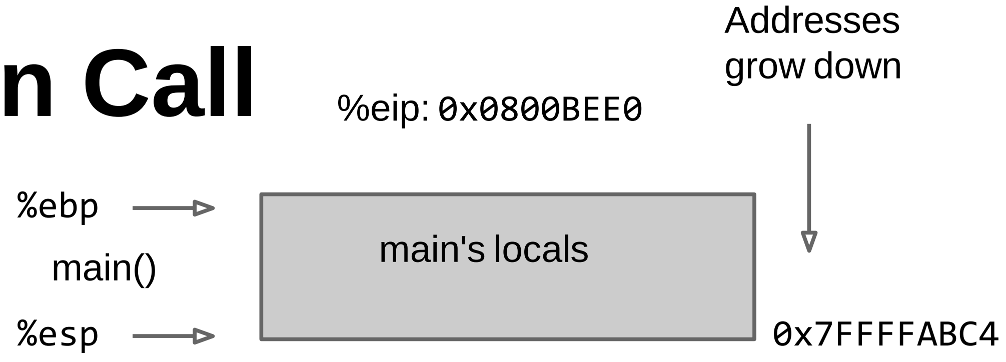
```
int f() {
    int x = 42;
    int foo = g(x);
    return foo + 5;
}
int g(int num) {
    return num+10;
}
```



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

No arguments!

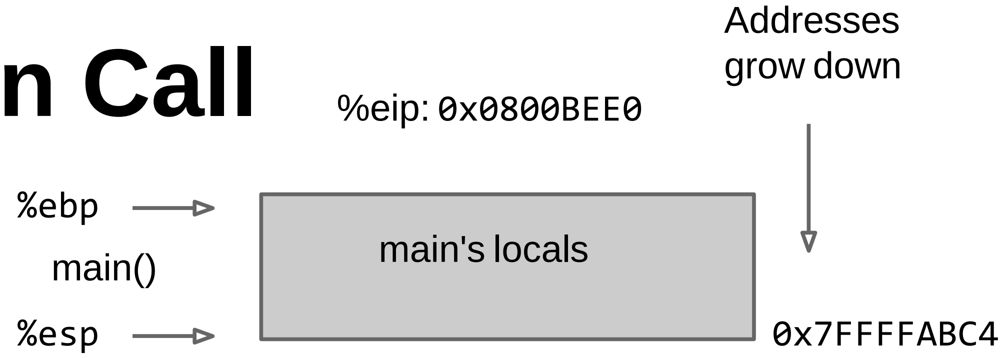
Example: Function Call




```
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




```c
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"

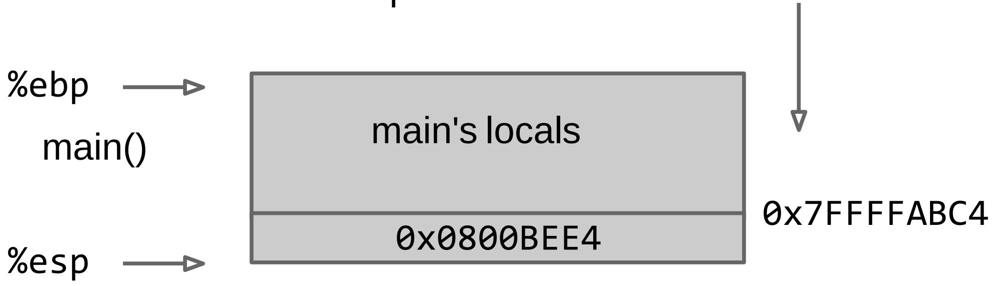
```
call f
```

Example: Function Call

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

Addresses grow down

%eip: `0x0801F00D`



%ebp = `0x7FFFFACE`

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

`pushl %ebp`

Example: Function Call

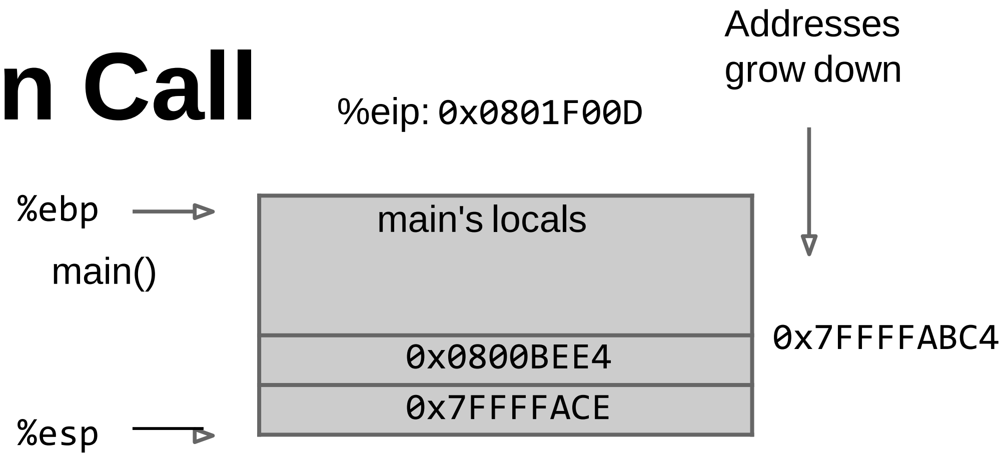
```
int f() {
    int x = 42;
    int foo = g(x);
    return foo + 5;
}
int g(int num) {
    return num+10;
}
```

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;
}
```



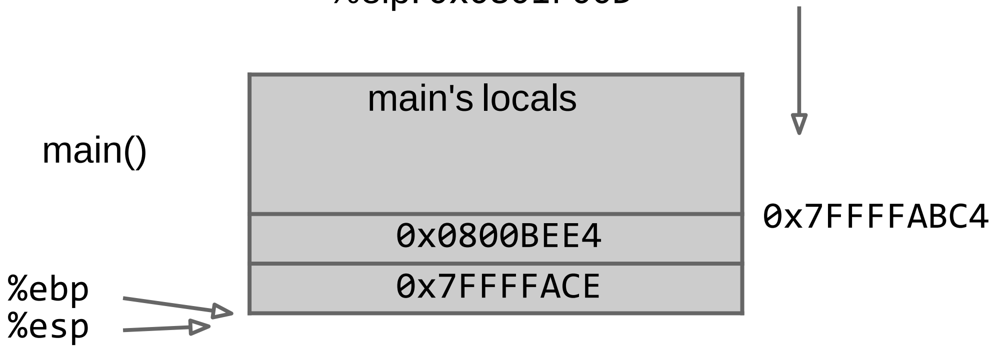
f() -- setup stack frame
1. Save caller's %ebp
2. Update %ebp to point to *my* frame base.

```
movl %esp %ebp
```

Example: Function Call

Addresses grow down

%eip: `0x0801F00D`



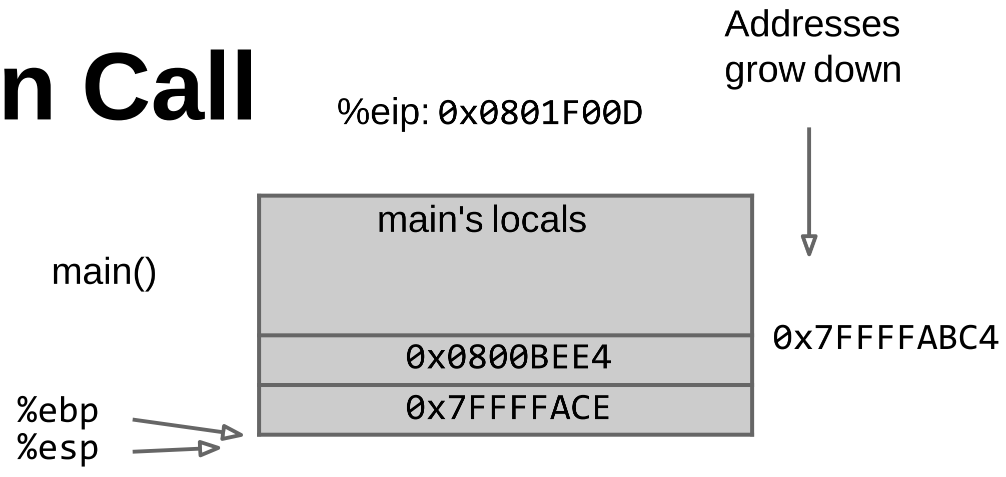

```
int f() {
    int x = 42;
    int foo = g(x);
    return foo + 5;
}
int g(int num) {
    return num+10;
}
```

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;
}
```



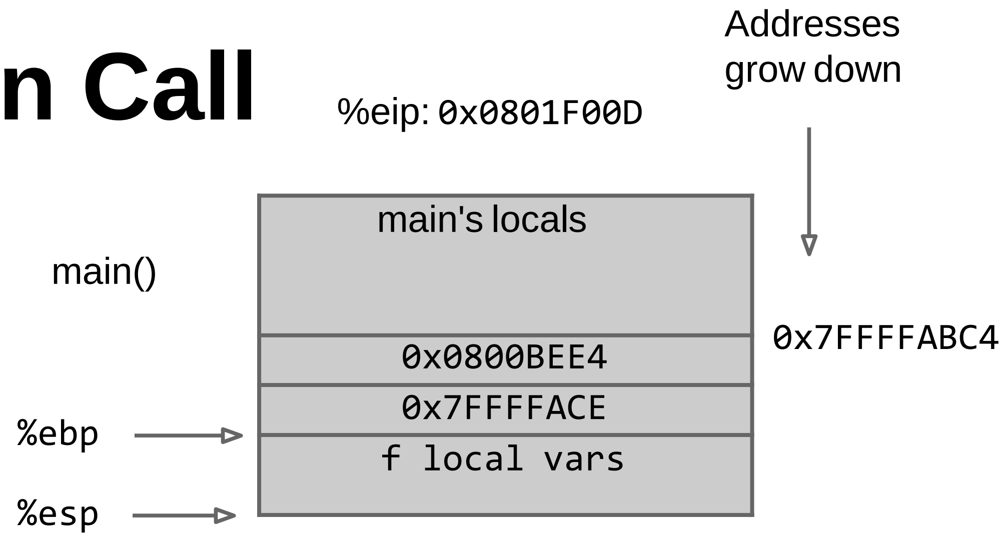
f() -- setup stack frame
1. Save caller's %ebp
2. Update %ebp to point to *my* frame base.
3. Allocate space for local variables

```
subl $8 %esp
```

Example: Function Call

%eip: `0x0801F00D`

```c
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

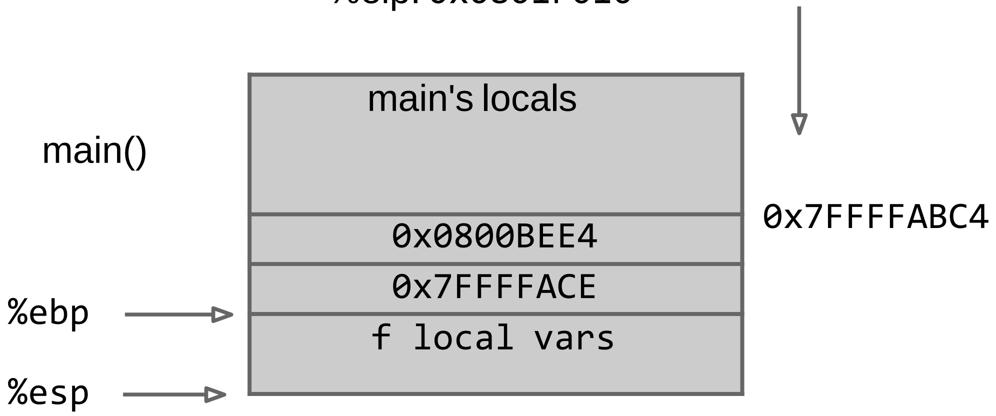
`subl $8 %esp`

Example: Function Call

Addresses grow down

%eip: `0x0801F010`

```
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

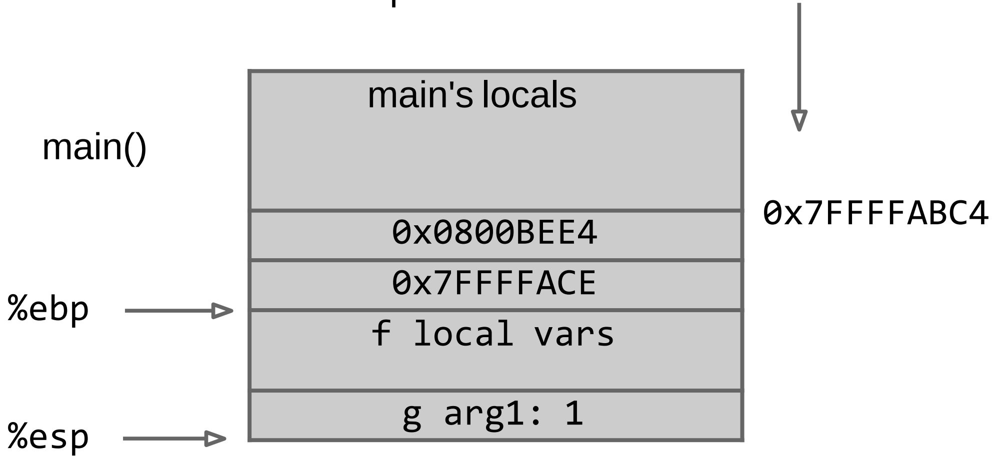
```
pushl $eax
```

Example: Function Call

Addresses grow down

%eip: `0x0801F010`

```c
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

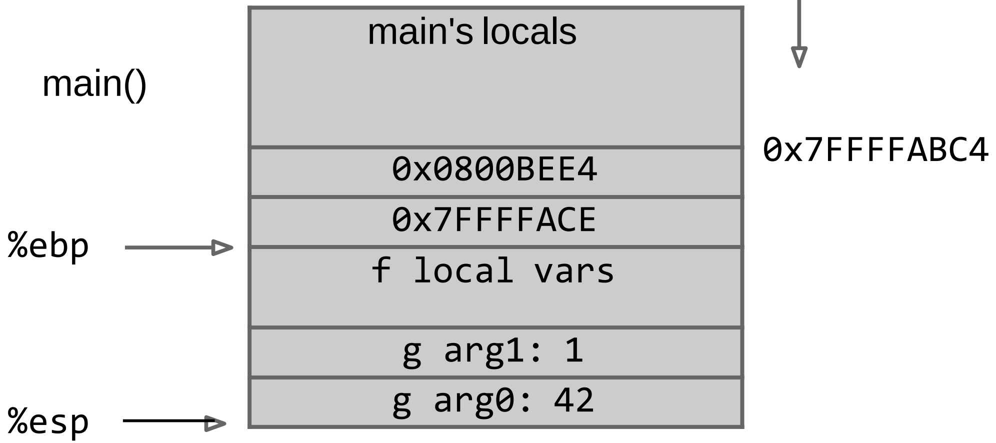
```
pushl 1
pushl $eax
```

Example: Function Call

%eip: `0x0801F010`

Addresses grow down

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



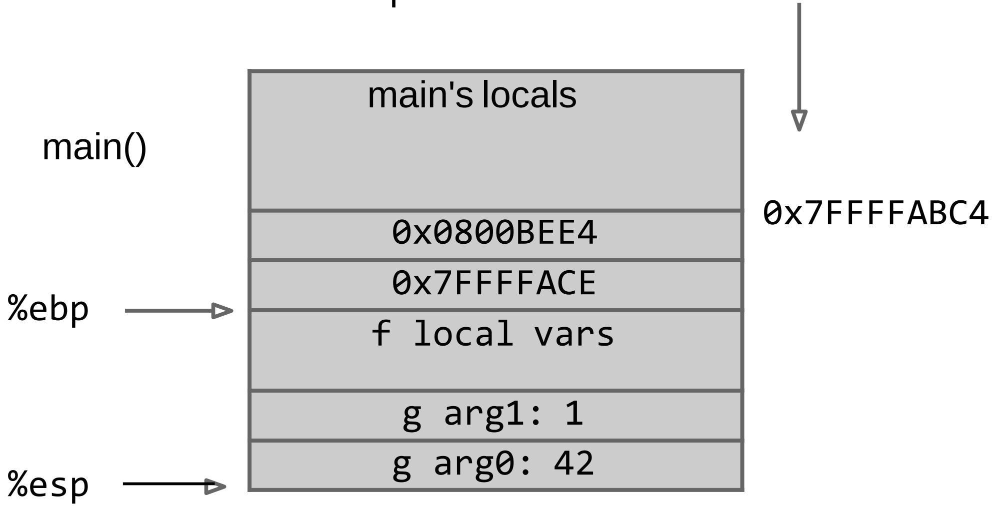
f() executes its code, and is about to call g().
1. Prepare arguments to g

```
pushl 1
pushl $eax
```

Example: Function Call

%eip: `0x0801F010`

Addresses grow down




```
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).

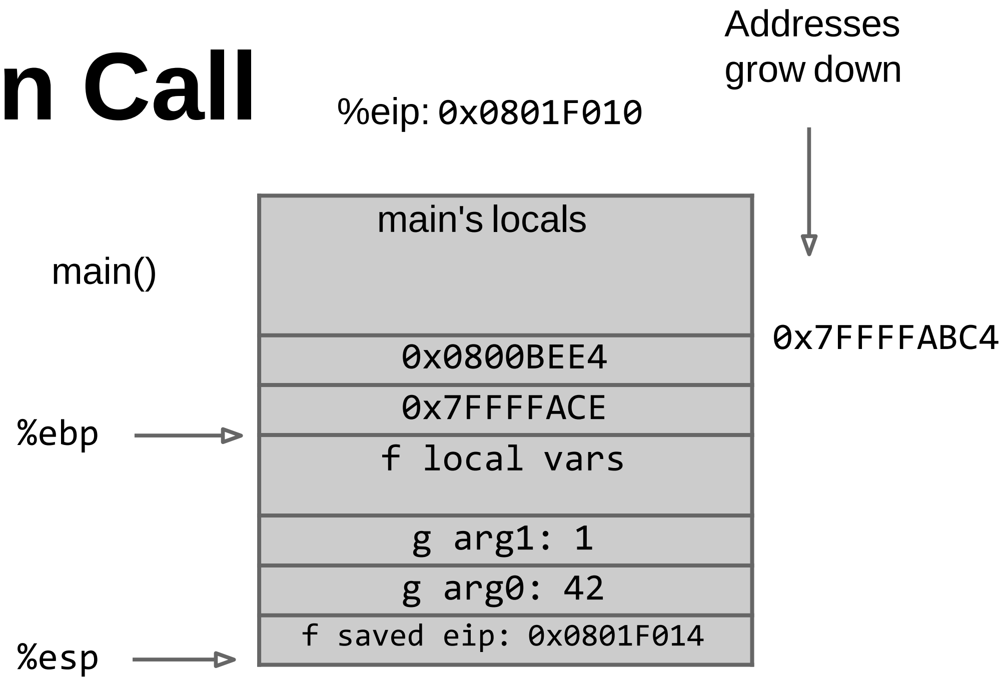
```
call g
```

Example: Function Call

%eip: `0x0801F010`

```
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().

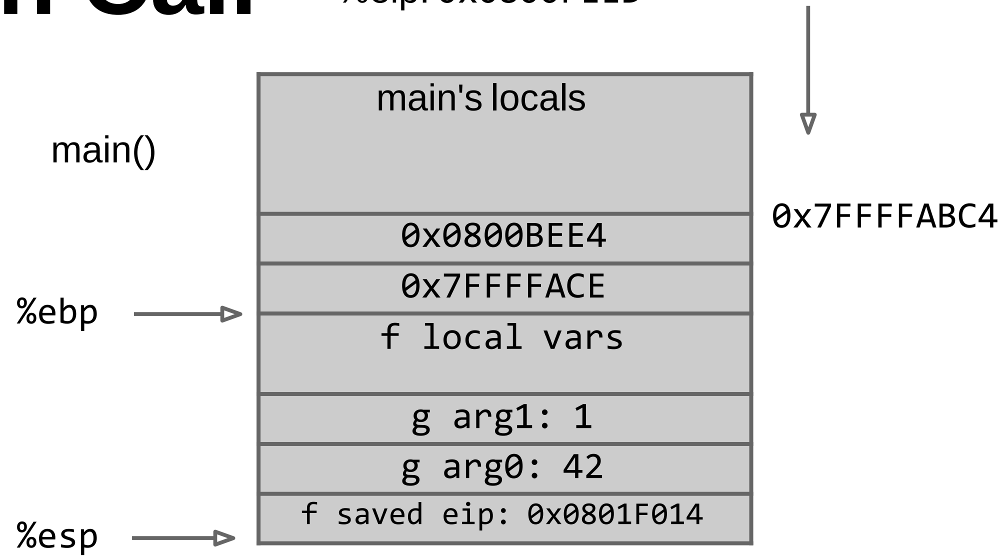
1. Prepare arguments to g
2. Call g (saves %eip).

`call g`

Example: Function Call

%eip: `0x0800FEED`

Addresses grow down




```
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`

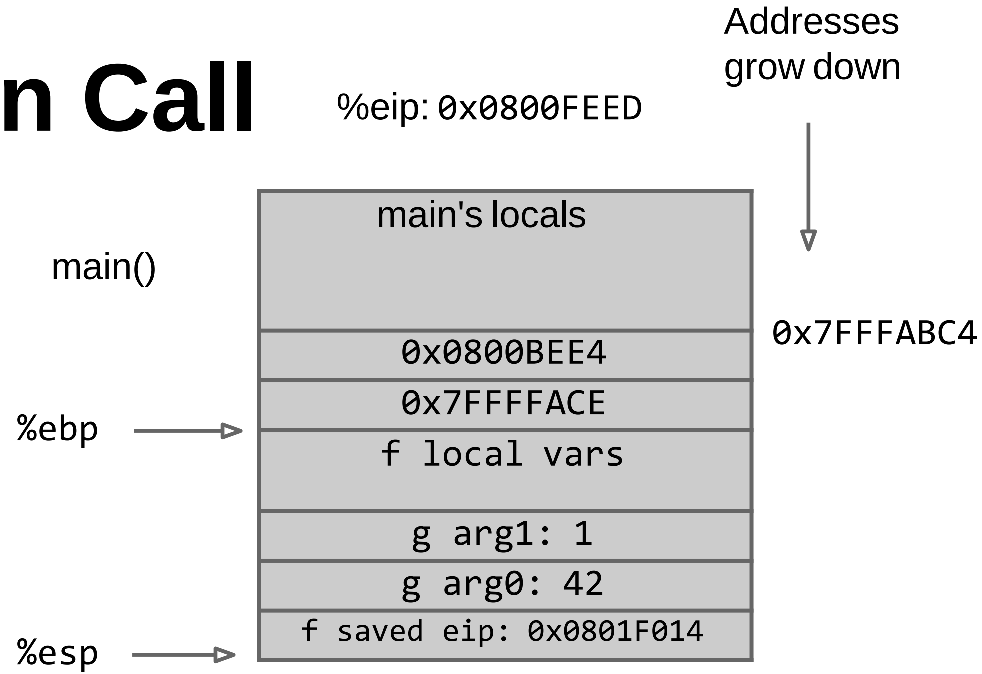
f() executes its code, and is about to call g().

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

%eip: `0x0800FEED`

Addresses grow down

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

main()

main's locals	0x7FFFABC4
0x0800BEE4	
0x7FFFFACE	
f local vars	← %ebp
g arg1: 1	
g arg0: 42	
f saved eip: 0x0801F014	
f saved ebp: 0x7FFFABDC	← %esp

g must setup its stack frame.
1. Save caller's %ebp.

`pushl %ebp`

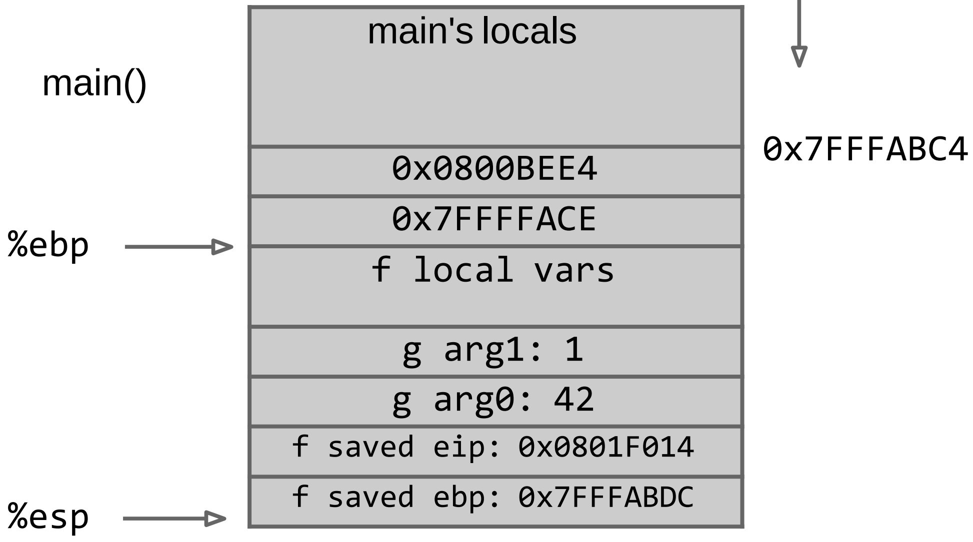
Example: Function Call

%eip: `0x0800FEED`

Addresses grow down

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

main()



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

%eip: `0x0800FEED`

Addresses grow down

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

main()

main's locals

0x7FFFABC4

`0x0800BEE4`

`0x7FFFFACE`

`f local vars`

`g arg1: 1`

`g arg0: 42`

`f saved eip: 0x0801F014`

`f saved ebp: 0x7FFFABDC`

%ebp

%esp

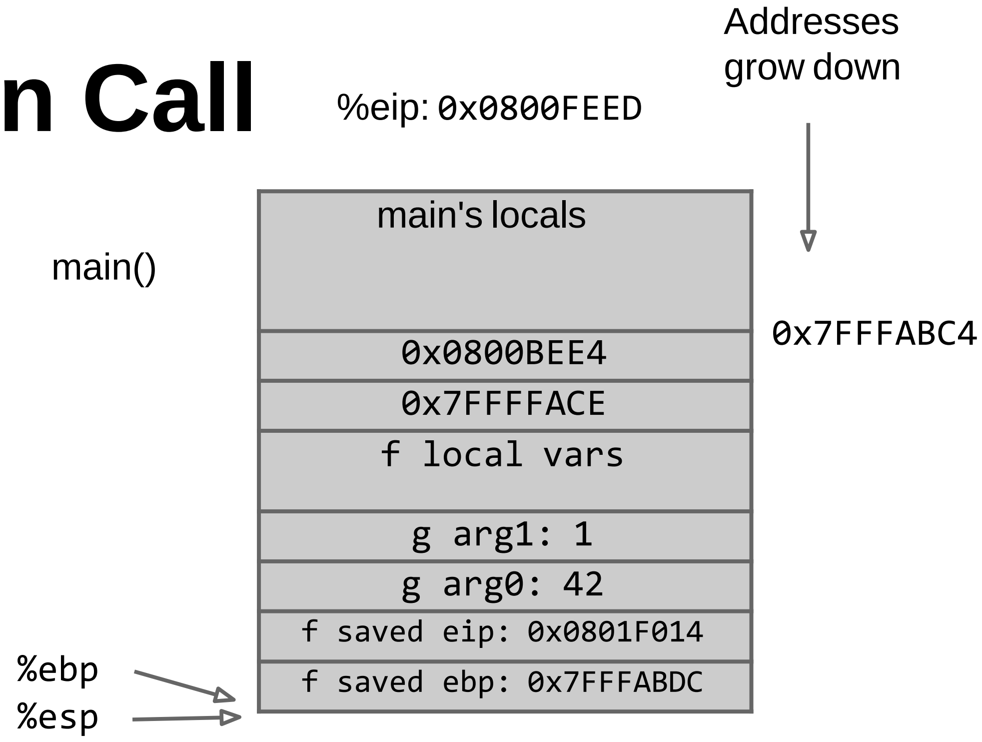
g must setup its stack frame.

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

%eip: `0x0800FEED`

Addresses grow down

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

main()

main's locals
0x0800BEE4
0x7FFFFACE
f local vars
g arg1: 1
g arg0: 42
f saved eip: 0x0801F014
f saved ebp: 0x7FFFABDC

0x7FFFABC4

%ebp

%esp

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.

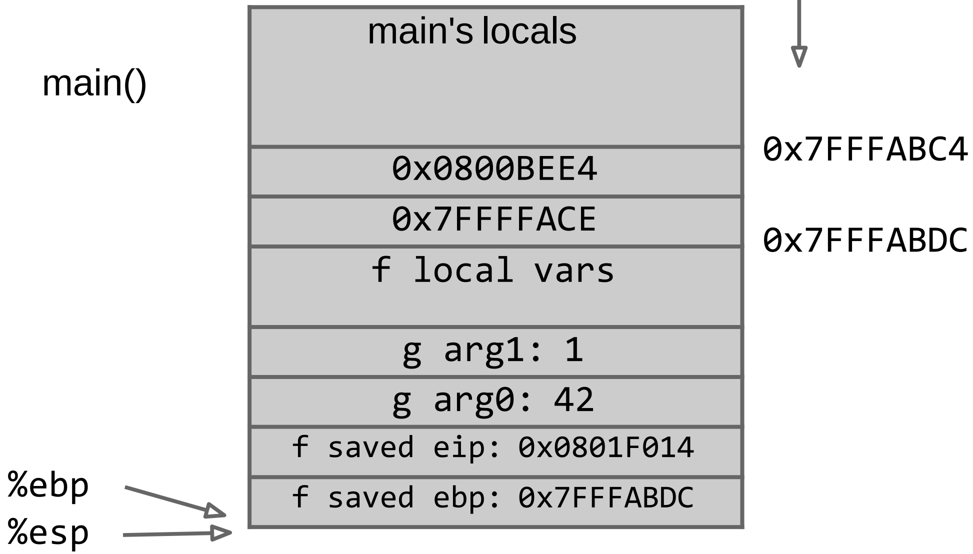
No local vars!

Example: Function Call

Addresses grow down

%eip: `0x0800FEED`

```
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

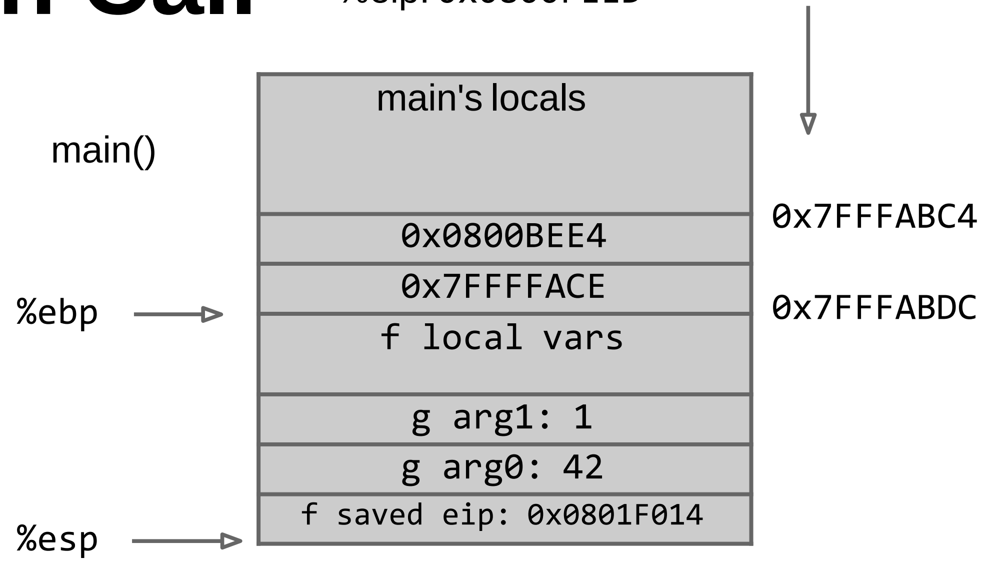
`popl %ebp`

Example: Function Call

%eip: `0x0800FEED`

Addresses grow down

```
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

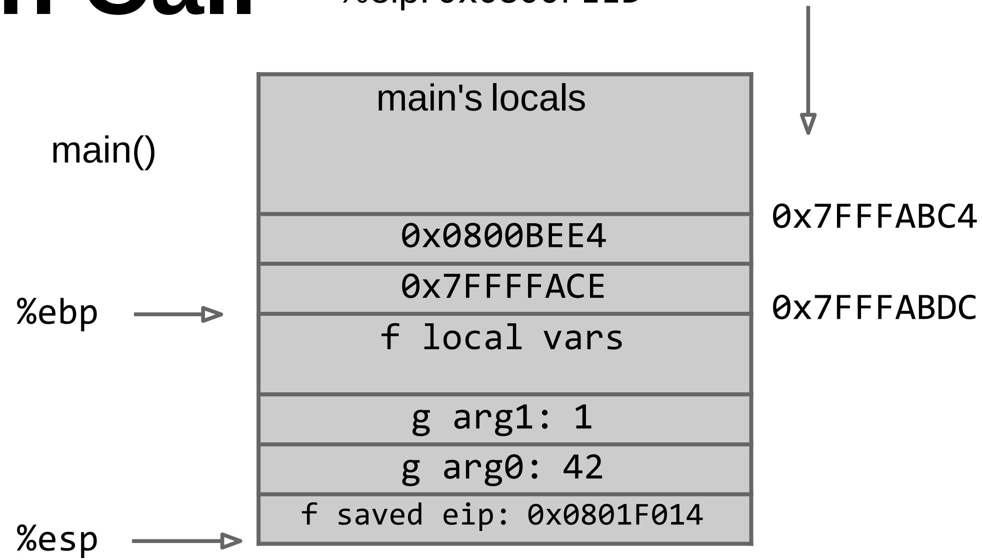
`popl %ebp`

Example: Function Call

Addresses grow down

%eip: `0x0800FEED`

```
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
```

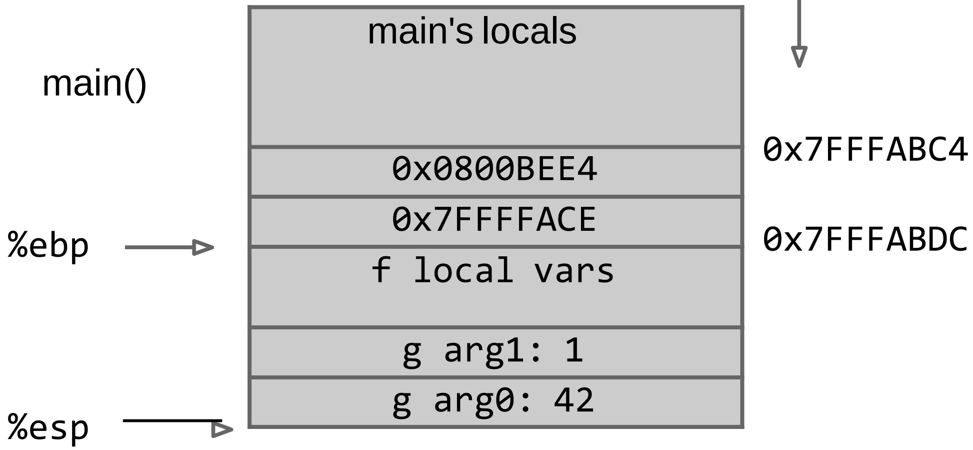
Example: Function Call

f in control now!

%eip: `0x0801F014`

Addresses grow down

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



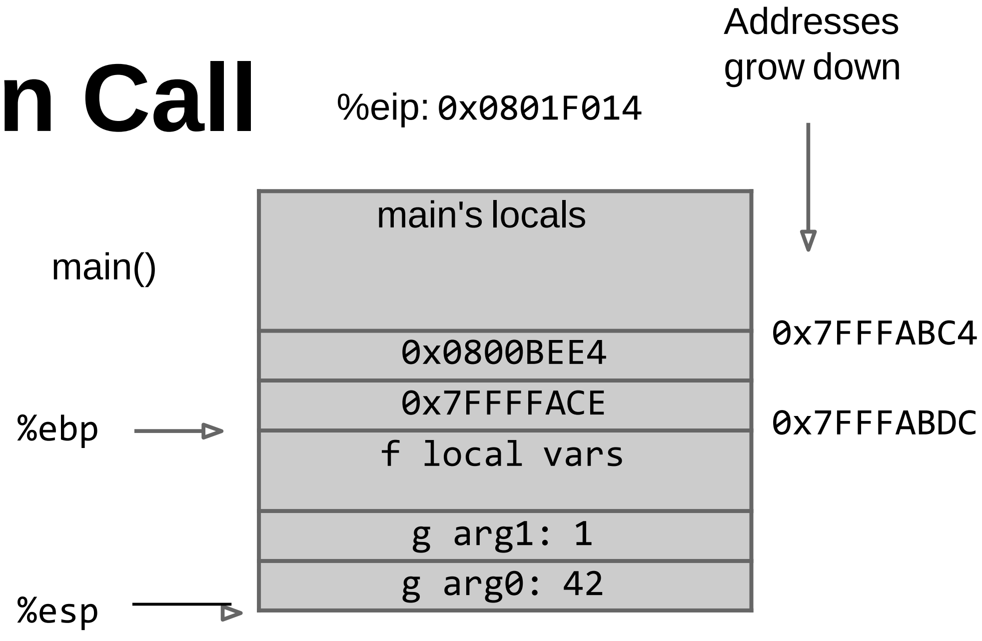
g() finishes, now must return.
1. Set %esp to caller's original %esp.
2. Set %ebp to caller's original %ebp.
3. Return to caller.

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;
}
```



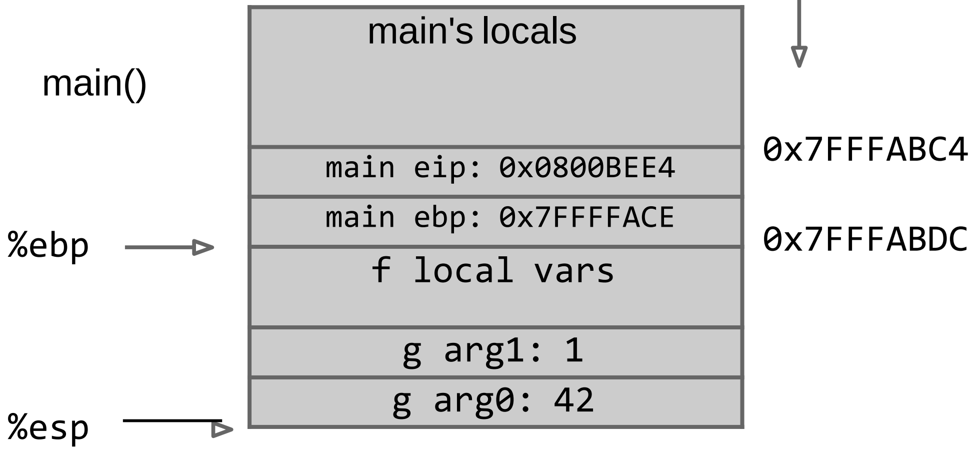
f() resumes executing.

Example: Function Call

Addresses grow down

%eip: `0x0801F014`

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



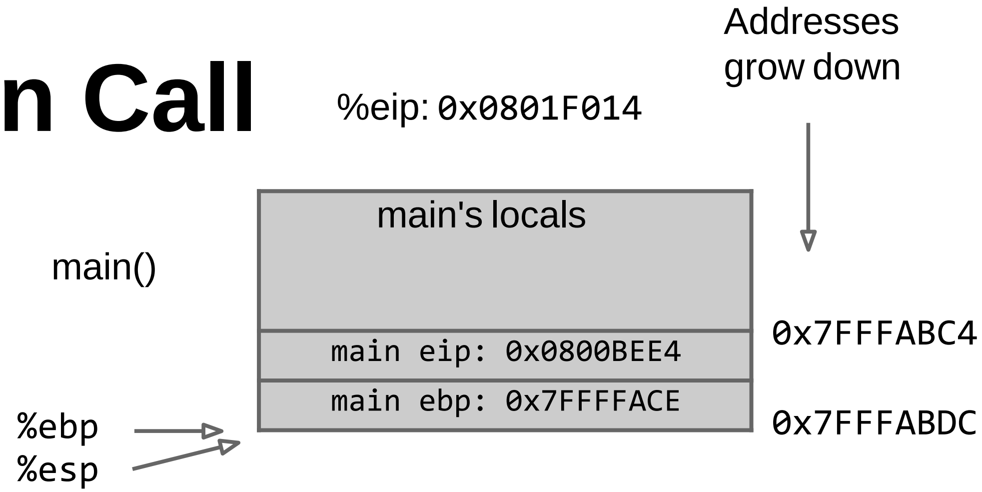
f() is ready to return.
1. Update %esp to caller's %esp.

`movl %ebp %esp`

Example: Function Call

%eip: `0x0801F014`

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



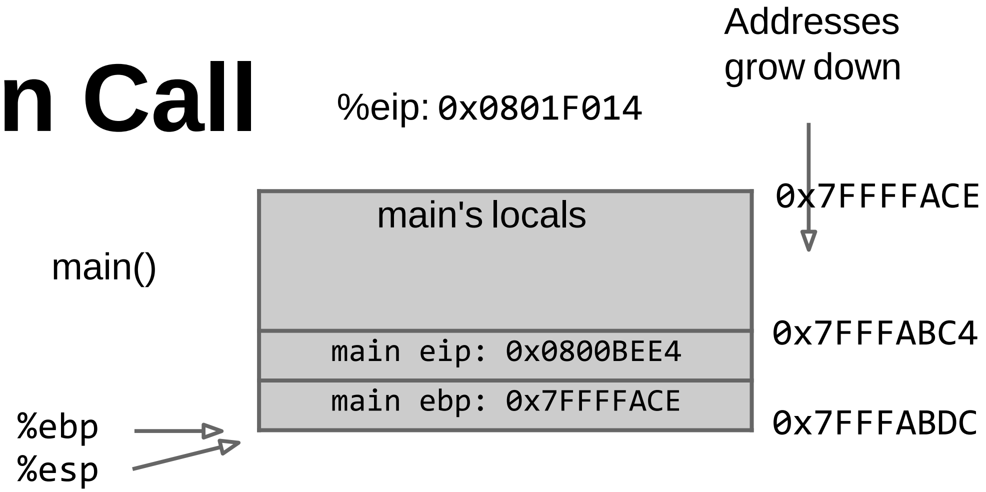
f() is ready to return.
1. Update %esp to caller's %esp.

```
movl %ebp %esp
```

Example: Function Call

%eip: `0x0801F014`

```
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`

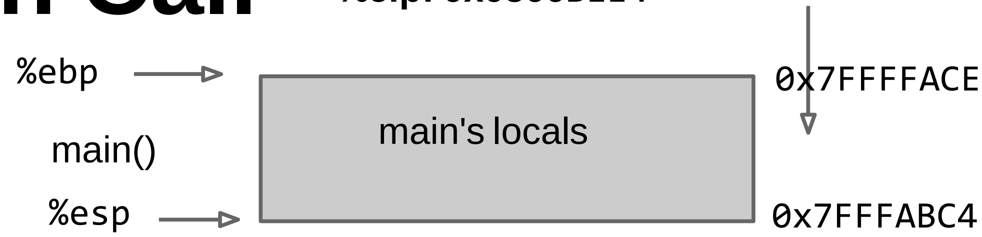
Example: Function Call

main() in control!

%eip: `0x0800BEE4`

Addresses grow down

```
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.

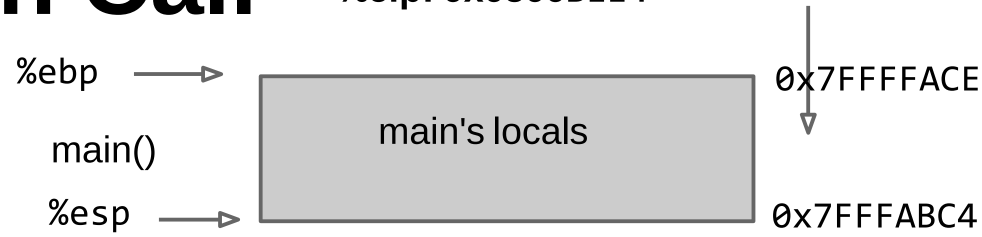
`ret`

Example: Function Call

main() in control!

%eip: `0x0800BEE4`

Addresses grow down




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

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
```

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.

x86-64 Stack Convention

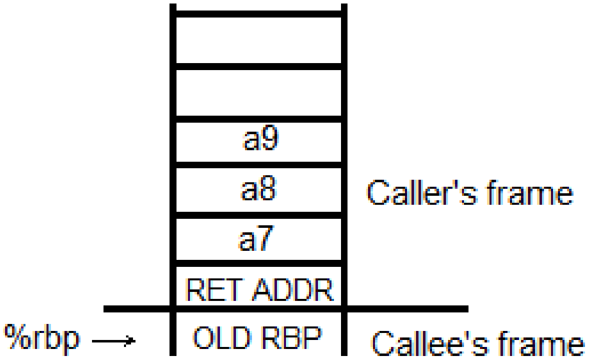
- What would the stack look like if we passed
- more than 6 arguments into the function?
-

Ex:

int func(long a1, long a2, long a3, long a4, long a5, long a6, long a7, long a8, long a9)

x86-64 Stack Convention

- int func (long a1, long a2, long a3,
  long a4, long a5, long a6,
  long a7, long a8, long a9)
- The first six arguments will be stored in registers.
- The following arguments will be placed the the stack by the caller.
- The earlier the argument in the list, the closer to %rbp.
- In this example, a7 is accessible to the callee via 0x10(%rbp), a8 is accessible via 0x18(%rbp), etc.



Compiling at Home

Try creating assembly output yourself!

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

Use x86 (ie 32-bit mode)

Compile to assembly

Do least amount of optimizations

Save generated assembly to: code.s

Input C source file. Your code here!

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!

Structs

```
struct s {
  char c1;
  int i;
  char c2;
  int j;
};
```

- What's the problem with this struct?

Structs

```
struct s {
  char c1;
  int i;
  char c2;
  int j;
};
```

- Say an instance of the struct begins at 0x10. Then c1 is at address 0x10. However, 'i' cannot be at address 0x11 (it needs to be 4-aligned). As a result, we need 3 bytes of padding.

Structs

- This is a waste of space! There will be 3 bytes of padding after c1 and 3 bytes of padding after c2, meaning that this struct will take up 16 bytes when really it only needs 10.

Structs

- Two common struct ordering guidelines (which could be at odds):
  1. Place the most commonly used data type first.
  2. Place the elements in descending order of size (ie largest first)
- Why?

Structs

- 1.
- Memory references are expensive (ex. (%eax))... but memory references with an offset are more expensive (ex. 8(%eax))
- Chances are, you'll be referring to the struct by a pointer to the beginning of the struct, which means that dereferencing the pointer without an offset will point to the first element.

Structs

- 2.
- If the elements with larger sizes are first, that means there will be less of a need for padding.
- For example, consider struct s, except with the first two elements swapped:

```
struct s {
  int i;
  char c1;
  char c2;
  int j;
};
```

Structs

- 2.
- ```
  struct s {
    int i;
    char c1;
    char c2;
    int j;
  };
  ```
- Now, we need 2 bytes of padding between c2 and j for a total of 12 bytes.
  ```

Structs

- Because each internal element must follow their own alignment rules, the alignment of the struct must be equal to the strictest of the elements within a struct.
- But wait...

Structs

- Consider:

```
struct s {
  char  c;
  int i;
};
```

- Because int i is aligned by 4, instances of struct s must be aligned by 4.
- There must also be 3 bytes of padding between c and I, meaning a total size of 8.

Structs

- Thus, a possible placement of (struct s s1) where s1.c = 0xFF and s1.i = 0x33221100 is the following:

Address:	0x10	0x11	0x12	0x13	0x14	0x15	0x16	0x17
Value:	0XFF	0xXX	0xXX	0xXX	0x00	0x11	0x22	0x33

- Where s begins at 0x10.
- This is how we meet the alignment requirements of each individual item

Structs

- But wait... what if you had the following code:

```
struct T
{
  struct s foo;
  char c;
} t;
```

- If t began at 0x10:
  - t.foo.i : 0x10 → 0x13
  - t.foo.c : 0x14
- But wait, t.c doesn't have to start at 0x18. It can be at 0x15 and all of the rules will be followed, right?

Structs

- But does it? If you try this:

```
int main()
{
  struct T test;
  printf("%p\n", &test.foo.i);
  printf("%p\n", &test.foo.c);
  printf("%p\n", &test.c);
}
```

- Output:
- 0x7ffdca140b40
- 0x7ffdca140b44
- 0x7ffdca140b48

- Nope, looks like sizeof(s) is really 8 bytes.

Unions

- Like structs except all of the values begin at the same address.
- union s {
-   short s;
-   char c;
- };
- This means that in a union that contains several values, only one of them is likely to be meaningful and assigning one term a value will trample other terms.

Unions

- union s {
-   short s;
-   char c;
- };
- union s foo;
- Say foo begins at 0x10.
- foo.s will be located in addresses 0x10 and 0x11
- foo.c will be located in address 0x10.

Unions

- union s {
-   short s;
-   char c;
- };
- union s foo;
- foo.s = 0xFFFF;
- foo.c = 0;
- printf(“%hx\n”, foo.s) => FF00

Midterm

- On Feb 7
- Covers the lecture material and relevant topics from book
- Covers the assignments and labs