Functions and the Stack

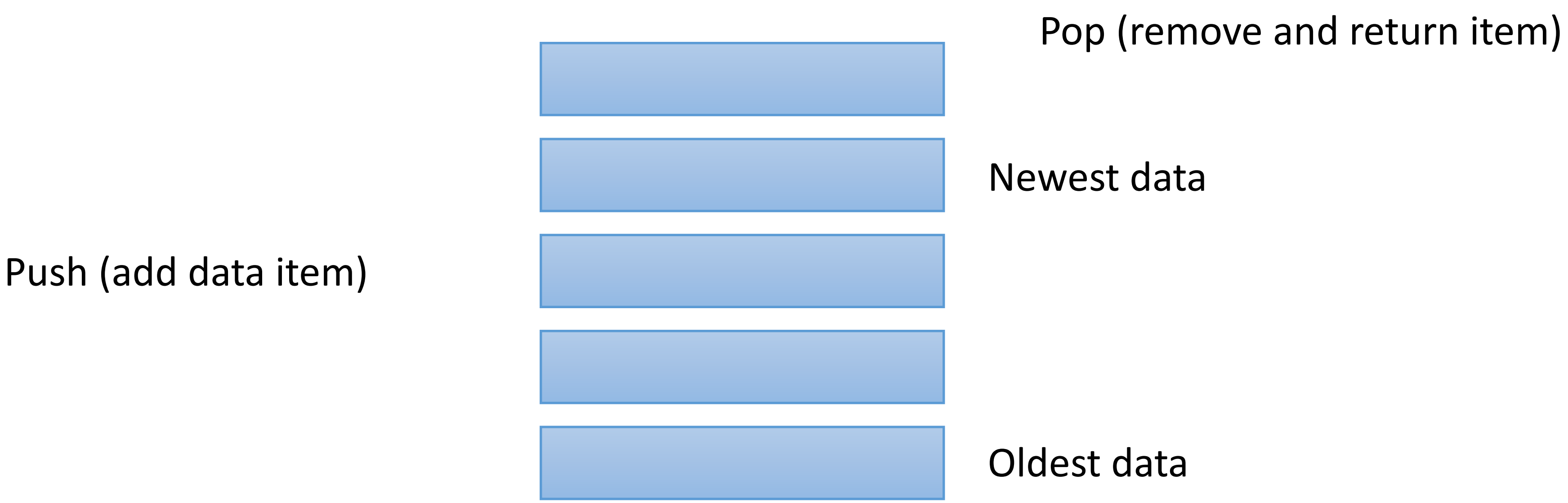
10/4/16

Overview

- Stack data structure, applied to memory
- Behavior of function calls
- Storage of function data, at IA32 level

“A” Stack

- A stack is a basic data structure
  - Last in, first out behavior (LIFO)
  - Two operations
    - Push (add item to top of stack)
    - Pop (remove item from top of stack)

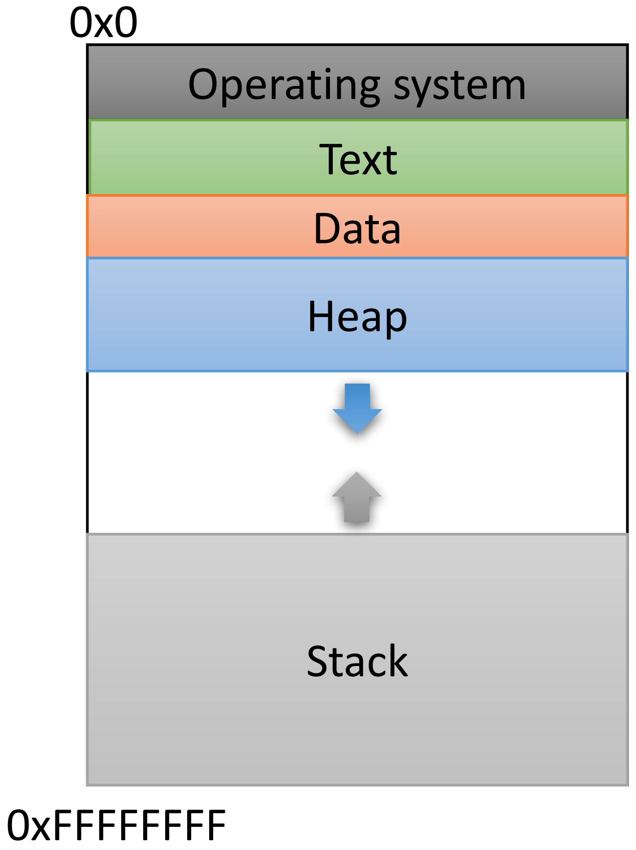


“The” Stack

- Apply stack data structure to memory
  - Store local (automatic) variables
  - Maintain state for functions (e.g., where to return)
- Organized into units called *frames*
  - One frame represents all of the information for one function.
  - Sometimes called *activation records*

Memory Model

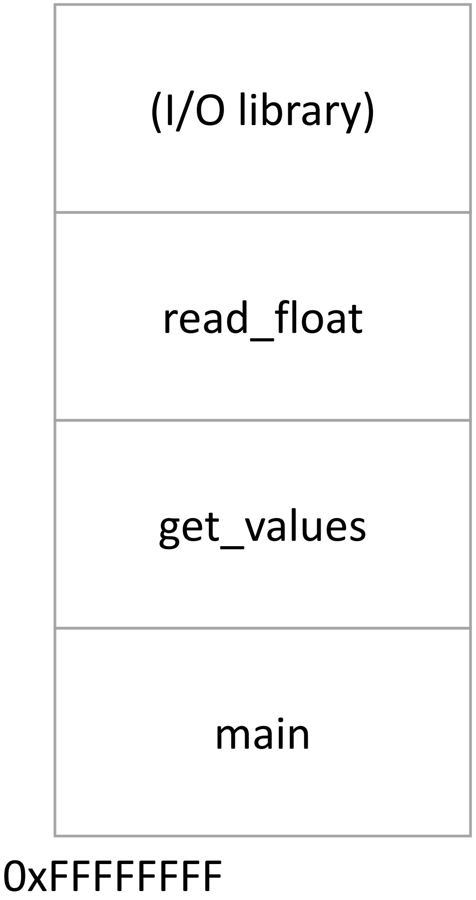
- Stack starts at the highest memory addresses, grows into lower addresses.



Stack Frames

- As functions get called,
  new frames added to stack.

- Example: Lab 4
  - main calls get_values()
  - get_values calls read_float()
  - read_float calls I/O library

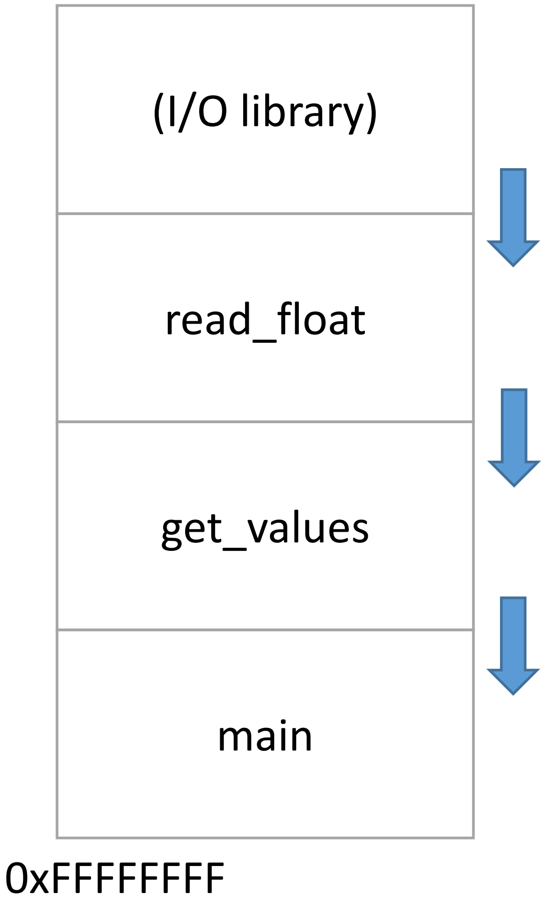


Stack Frames

- As functions return,
  frames removed from stack.

- Example: Lab 4
  - I/O library returns to read_float
  - read_float returns to get_values
  - get_values returns to main

All of this stack growing/shrinking happens automatically (from the programmer’s perspective).



What is responsible for creating and removing stack frames?

A. The user

B. The compiler

C. C library code

D. The operating system

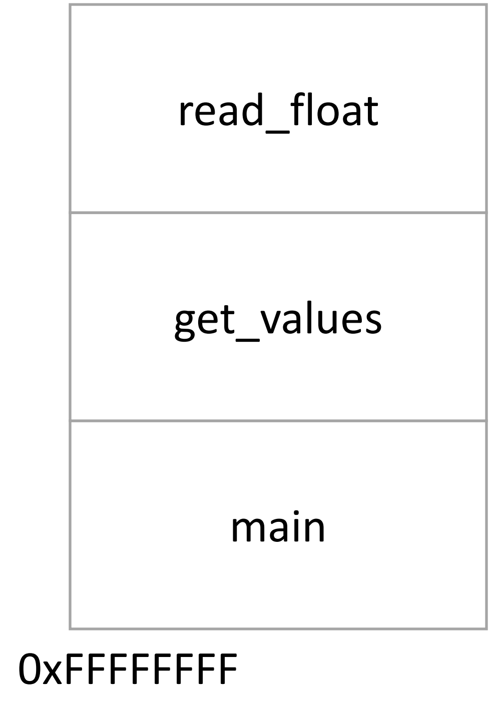
E. Something / someone else

Insight: EVERY function needs a stack frame. Creating / destroying a stack frame is a (mostly) generic procedure.

Stack Frame Contents

- What needs to be stored in a stack frame?
  - Alternatively: What *must* a function know / access?

  - Hint: At least 5 things

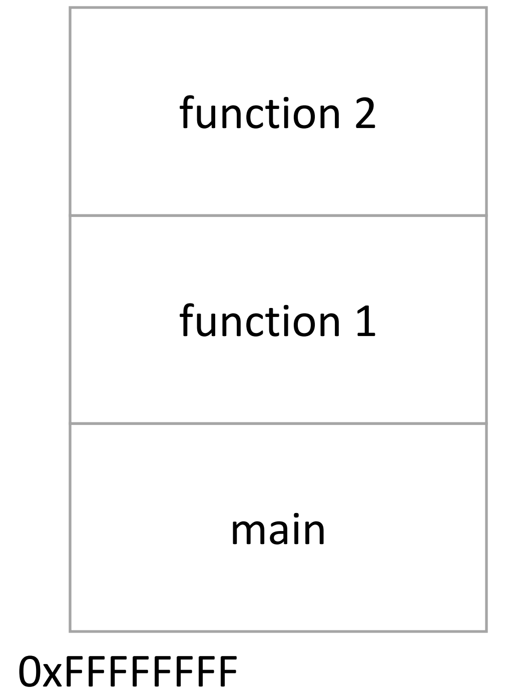


Stack Frame Contents

- What needs to be stored in a stack frame?
  - Alternatively: What *must* a function know?

- Local variables
- Previous stack frame base address
- Function arguments
- Return value
- Return address

- Saved registers
- Spilled temporaries



Local Variables

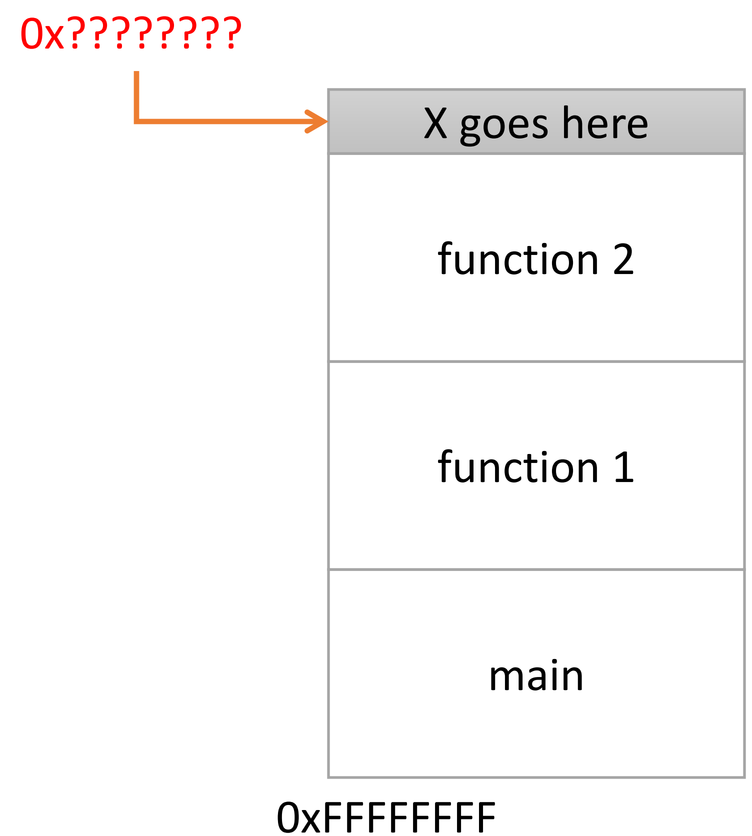
If the programmer says:

```
int x = 0;
```

Where should x be stored?

(Recall basic stack data structure)

Which memory address is that?



How should we determine the address to use for storing a new local variable?

A. The programmer specifies the variable location.

B. The CPU stores the location of the current stack frame.

C. The operating system keeps track of the top of the stack.

D. The compiler knows / determines where the local data for each function will be as it generates code.

E. The address is determined some other way.

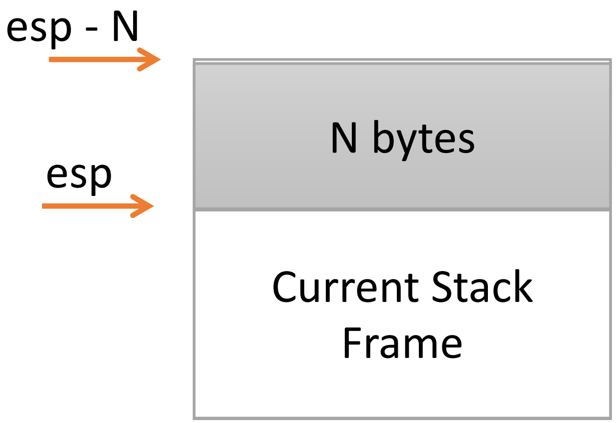
- Compile time (static)
  - Information that is known by analyzing your program
  - Independent of the machine and inputs
- Run time (dynamic)
  - Information that isn’t known until program is running
  - Depends on machine characteristics and user input

The Compiler Can…

- Determine how much space you need on the stack to store local variables.
- Insert IA32 instructions for you to set up the stack for function calls.
  - Create stack frames on function call
  - Restore stack to previous state on function return
- Perform type checking, etc.

Local Variables

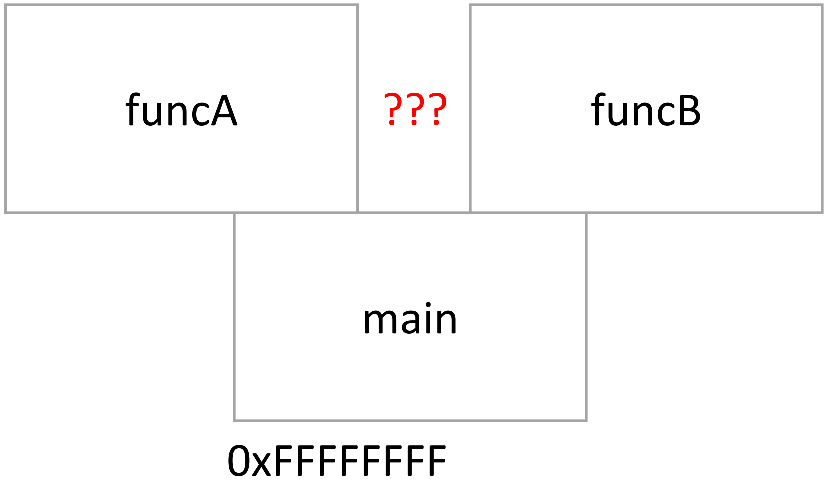
- Compiler can allocate N bytes on the stack by subtracting N from the “stack pointer”: %esp



The Compiler Can't…

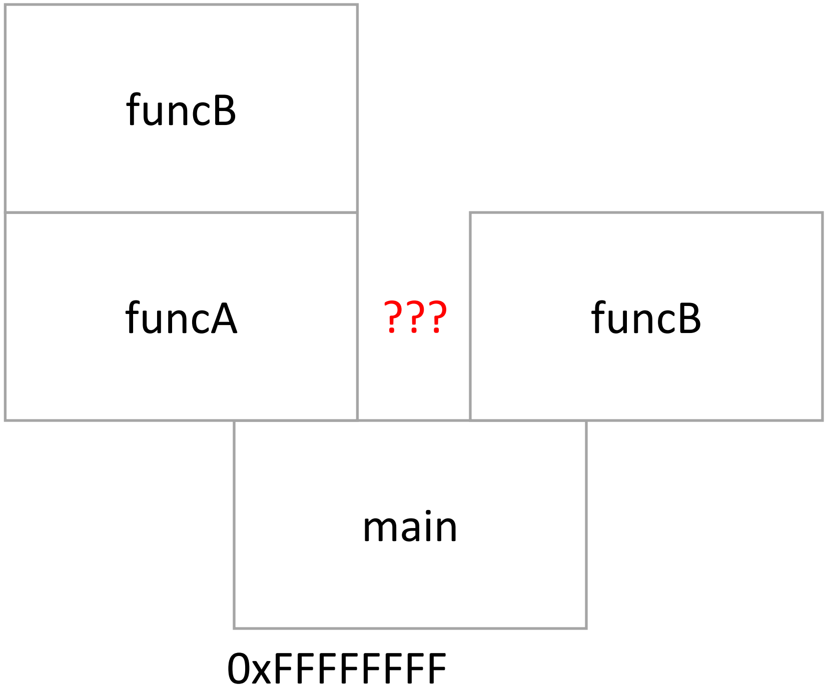
- Predict user input.

```
int main() {
  int x = get_user_input();
  if (x > 5) {
    funcA(x);
  } else {
    funcB();
  }
}
```



The Compiler Can't…

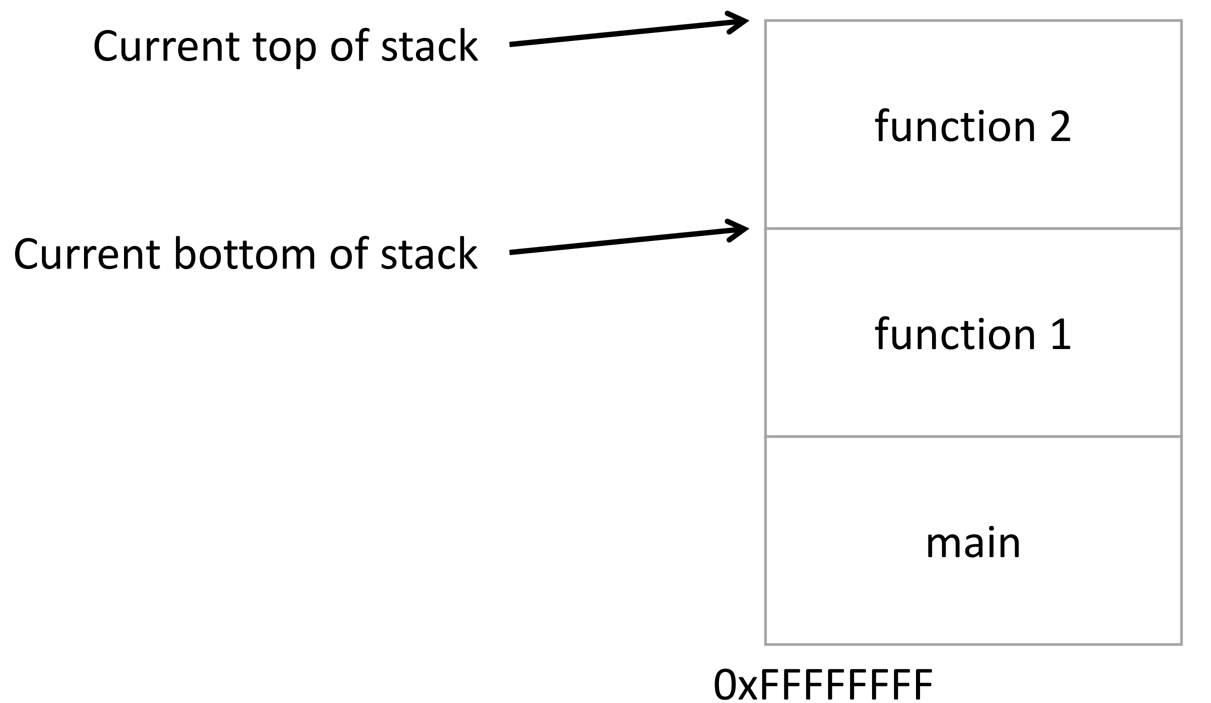
- Predict user input.
- Assume a function will always be at a certain address on the stack.



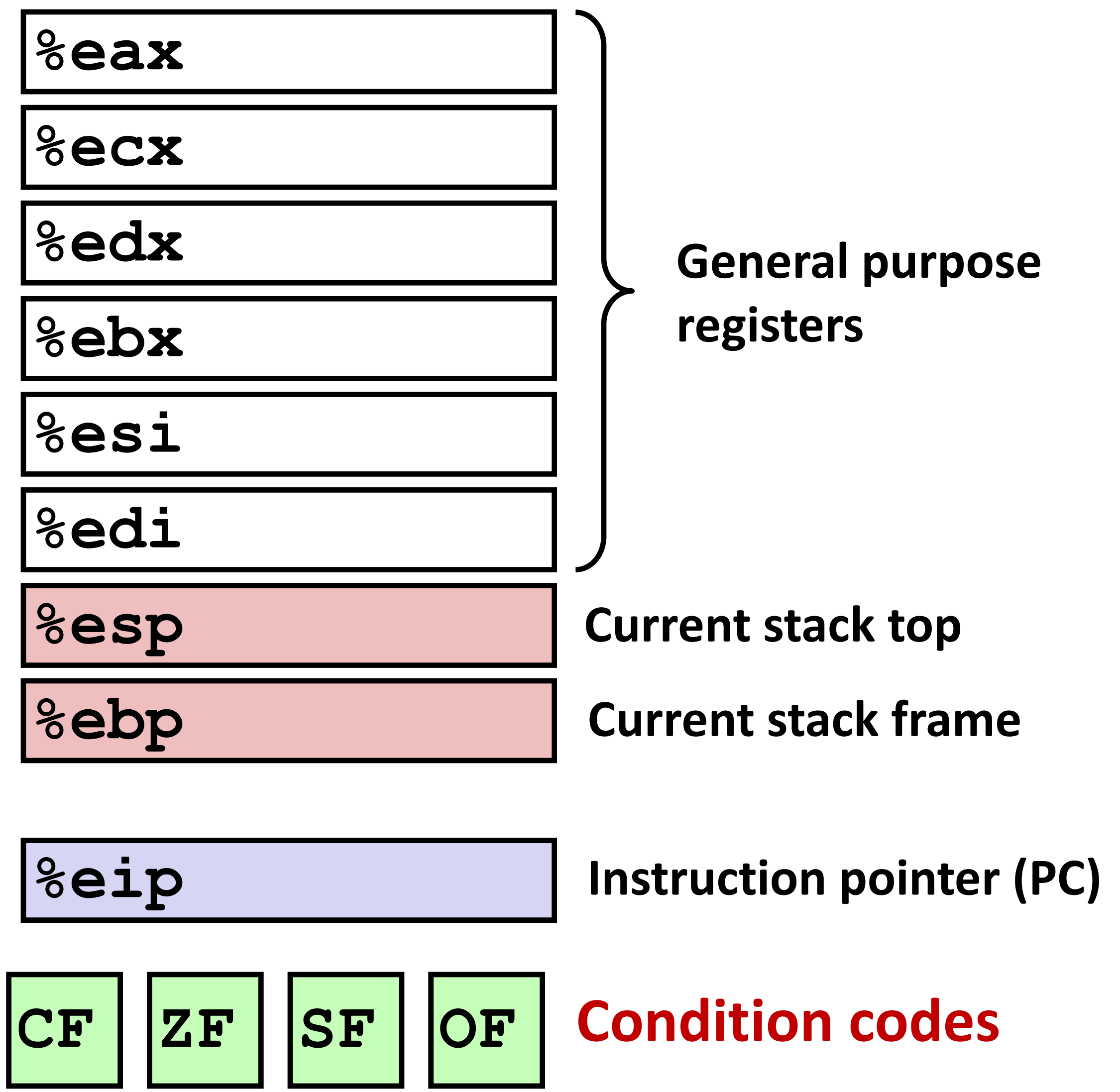
Alternative: create stack frames relative to the current (dynamic) state of the stack.

Stack Frame Location

- Where in memory is the current stack frame?



Recall: IA32 Registers

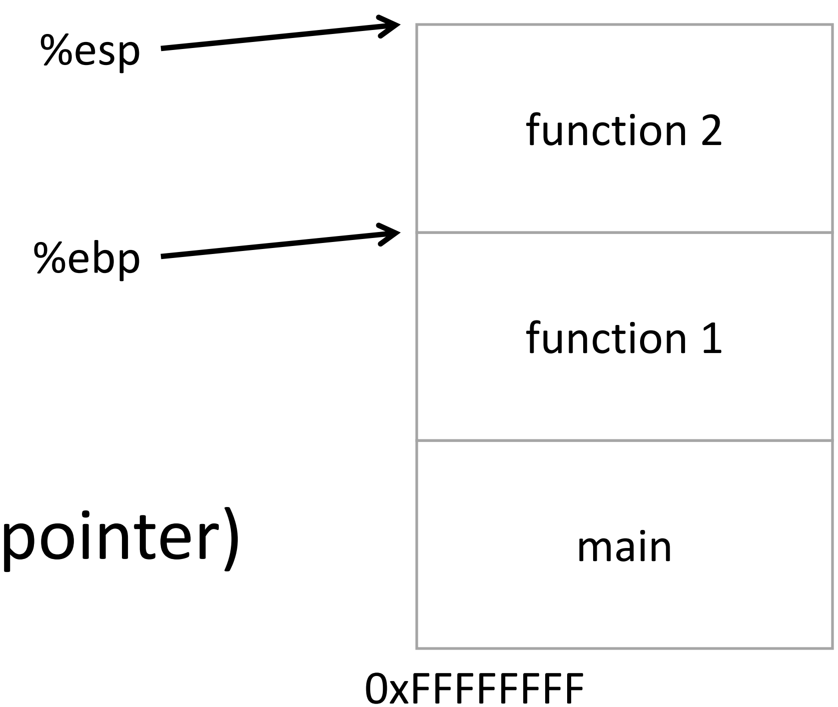


Stack Frame Location

- Where in memory is the current stack frame?

- Maintain invariant:
  - The current function's stack frame is always between the addresses stored in %esp and %ebp

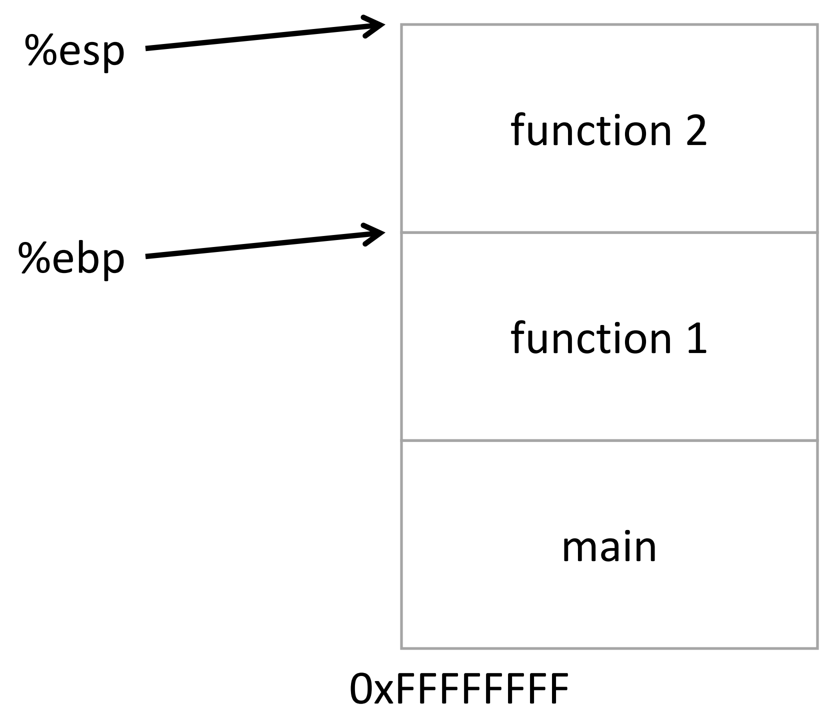
- %esp: stack pointer
- %ebp: frame pointer (base pointer)



Stack Frame Location

- Compiler ensures that this invariant holds.
  - We'll see how a bit later.

- This is why all local variables we've seen in IA32 are relative to %ebp or %esp!



How would we implement pushing x to the top of the stack in IA32?

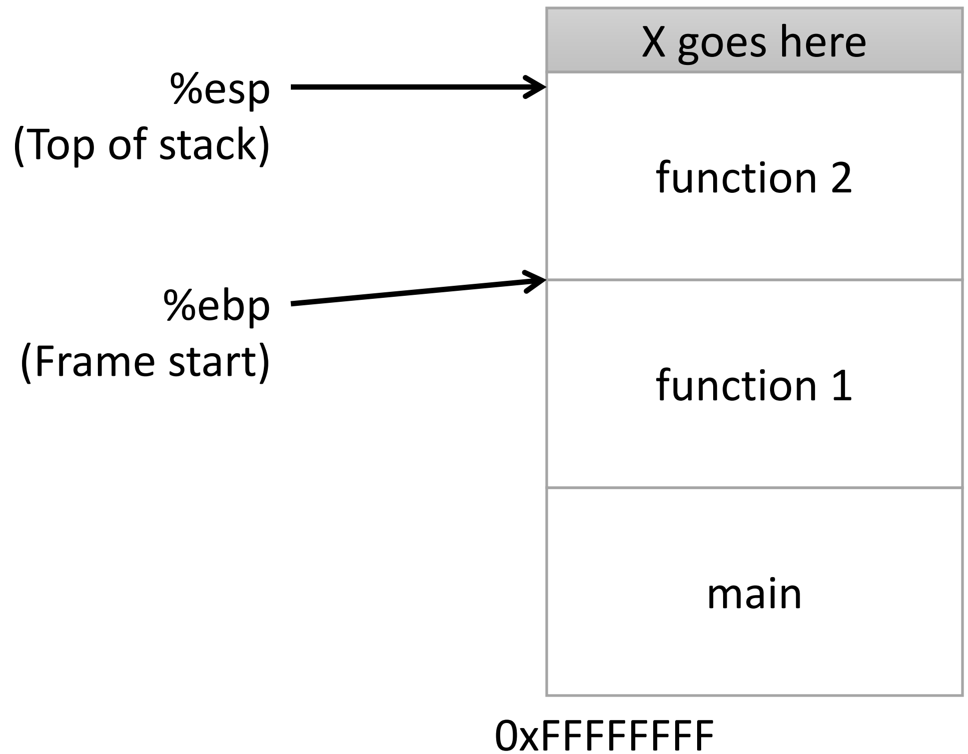
A. Increment %esp
   Store x at (%esp)

B. Store x at (%esp)
   Increment %esp

C. Decrement %esp
   Store x at (%esp)

D. Store x at (%esp)
   Decrement %esp

E. Copy %esp to %ebp
   Store x at (%ebp)



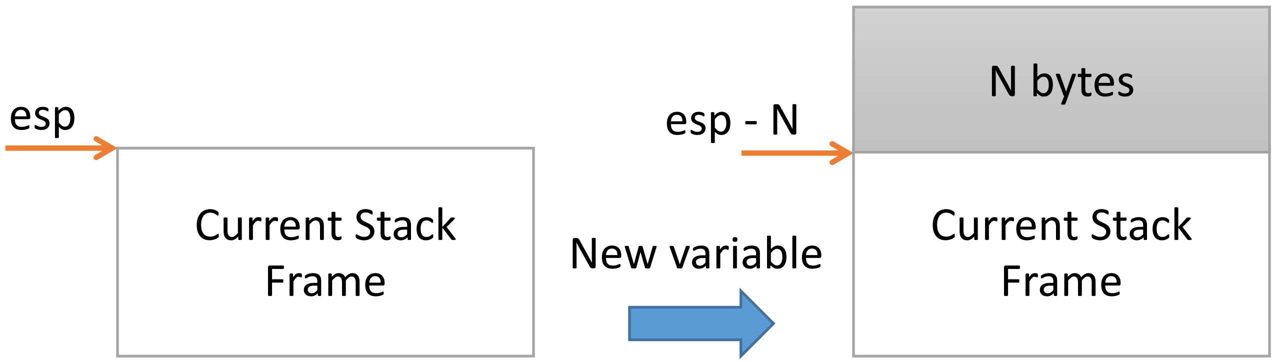
Push & Pop

- IA32 provides convenient instructions:
  - `pushl src`
    - Move stack pointer up by 4 bytes `subl $4, %esp`
    - Copy 'src' to current top of stack `movl src, (%esp)`
  - `popl dst`
    - Copy current top of stack to 'dst' `movl (%esp), dst`
    - Move stack pointer down 4 bytes `addl $4, %esp`

- src and dst are the contents of any register

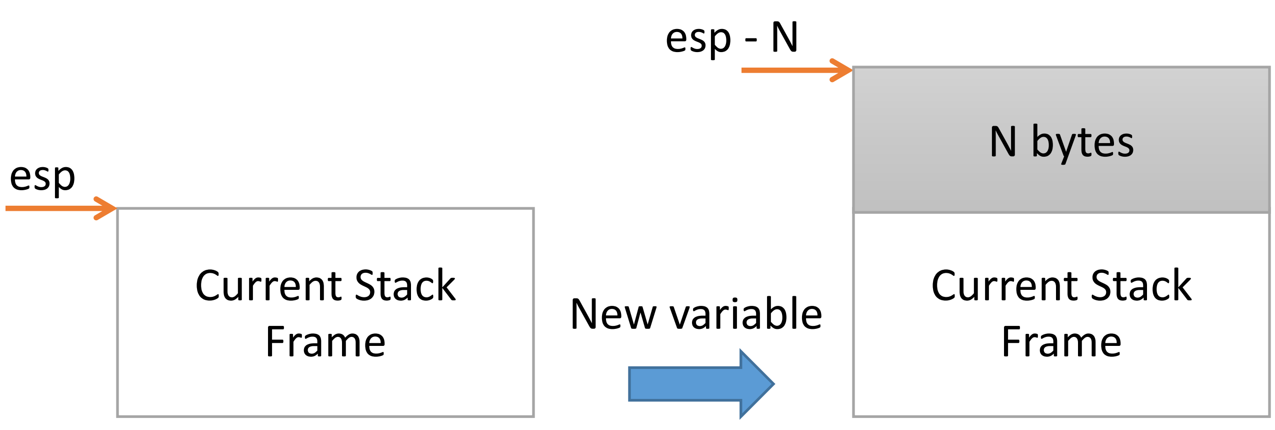
Local Variables

- More generally, we can make space on the stack for N bytes by subtracting N from %esp



Local Variables

- More generally, we can make space on the stack for N bytes by subtracting N from %esp
- When we're done, free the space by adding N back to %esp

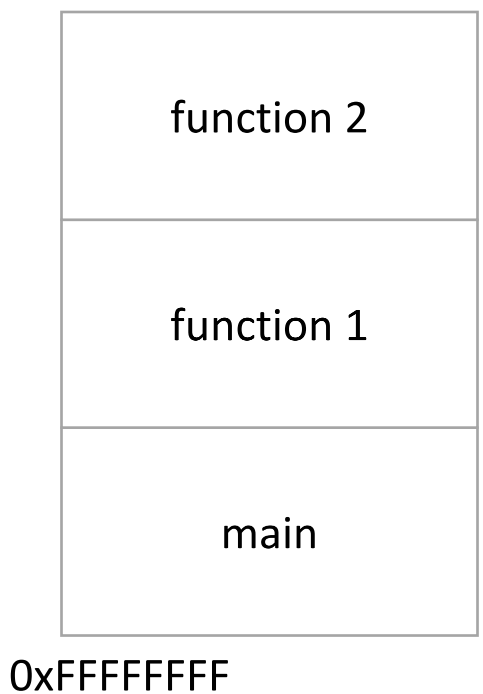


Stack Frame Contents

- What needs to be stored in a stack frame?
  - Alternatively: What *must* a function know?

- Local variables
- Previous stack frame base address
- Function arguments
- Return value
- Return address

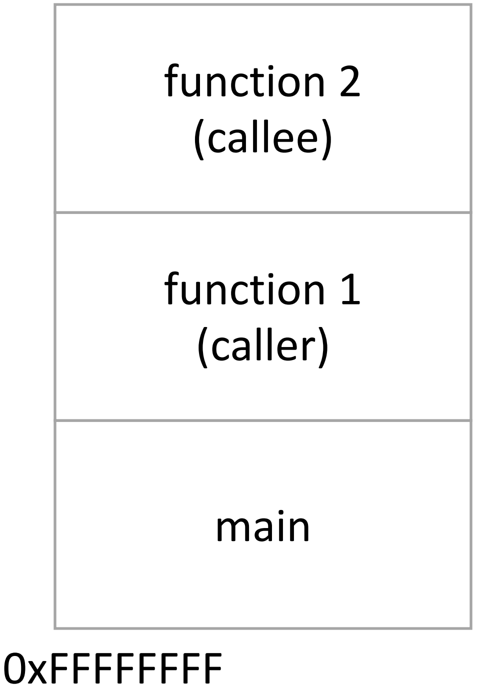
- Saved registers
- Spilled temporaries



Stack Frame Relationships

- If function 1 calls function 2:
  - function 1 is the caller
  - function 2 is the callee

- With respect to main:
  - main is the caller
  - function 1 is the callee



Where should we store all this stuff?

Previous stack frame base address
Function arguments
Return value
Return address

A. In registers
B. On the heap
C. In the caller's stack frame
D. In the callee's stack frame
E. Somewhere else

Calling Convention

- You could store this stuff wherever you want!
  - The hardware does NOT care.
  - What matters: everyone agrees on where to find the necessary data.

- Calling convention: agreed upon system for exchanging data between caller and callee

IA32 Calling Convention (gcc)

- In register %eax:
  - The return value
- In the callee's stack frame:
  - The caller's %ebp value (previous frame pointer)
- In the caller's frame (shared with callee):
  - Function arguments
  - Return address (saved PC value)

IA32 Calling Convention (gcc)

- In register %eax:
  - The return value

- In the callee's stack frame:
  - The caller's %ebp value (previous frame pointer)

- In the caller's frame (shared with callee):
  - Function arguments
  - Return address (saved PC value)

IA32 Calling Convention (gcc)

- In register %eax:
  - The return value

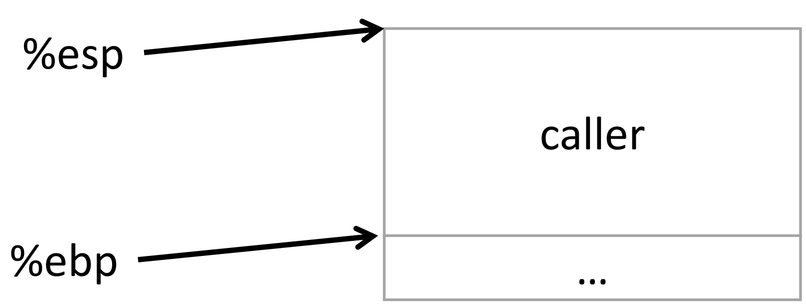
- In the callee's stack frame:
  - The caller's %ebp value (previous frame pointer)

- In the caller's frame (shared with callee):
  - Function arguments
  - Return address (saved PC value)

Frame Pointer

- Must maintain invariant:
  - The current function's stack frame is always between the addresses stored in %esp and %ebp

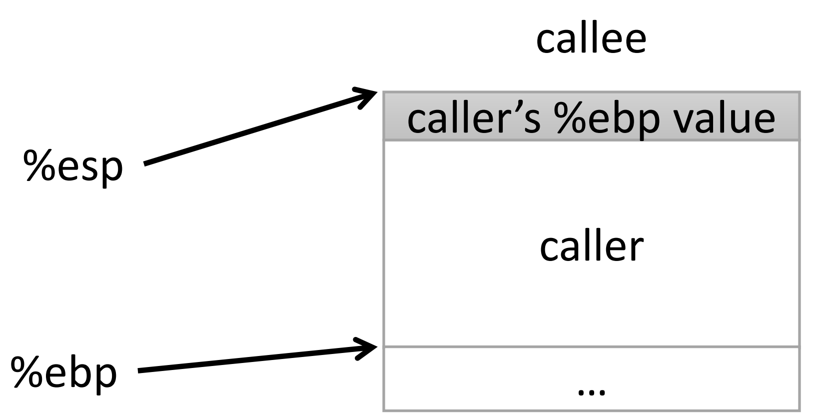
- Must adjust %esp, %ebp on call / return.



Frame Pointer

- Must maintain invariant:
  - The current function's stack frame is always between the addresses stored in %esp and %ebp

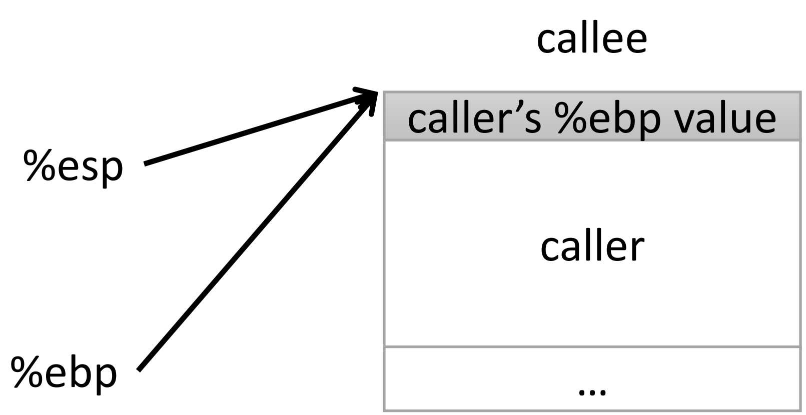
- Immediately upon calling a function:
  1. pushl %ebp



Frame Pointer

- Must maintain invariant:
  - The current function's stack frame is always between the addresses stored in %esp and %ebp

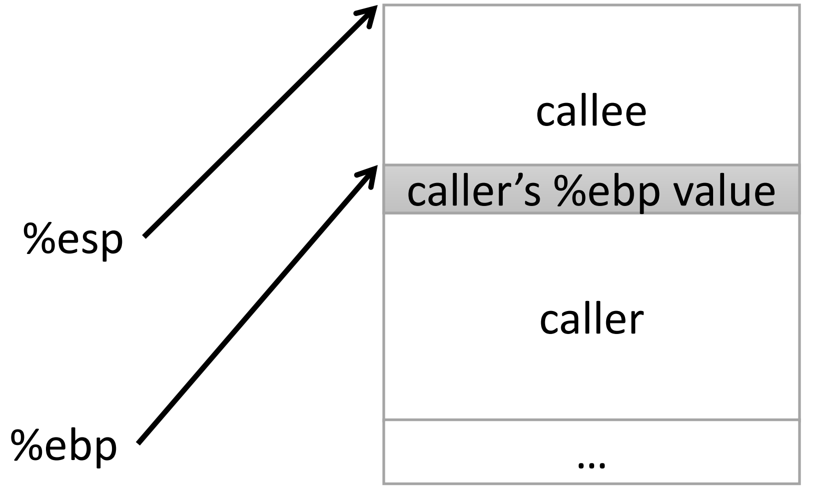
- Immediately upon calling a function:
  1. pushl %ebp
  2. Set %ebp = %esp



Frame Pointer

- Must maintain invariant:
  - The current function's stack frame is always between the addresses stored in %esp and %ebp

- Immediately upon calling a function:
  1. pushl %ebp
  2. Set %ebp = %esp
  3. Subtract N from %esp

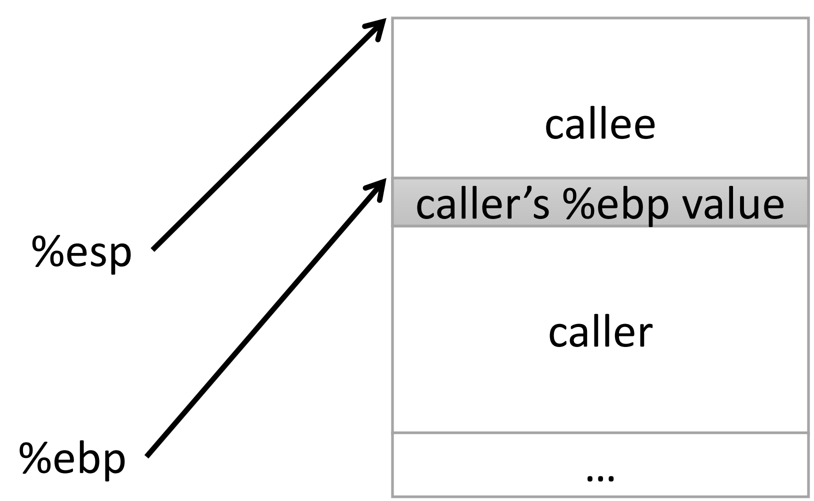


Callee can now execute.

Frame Pointer

- Must maintain invariant:
  - The current function's stack frame is always between the addresses stored in %esp and %ebp

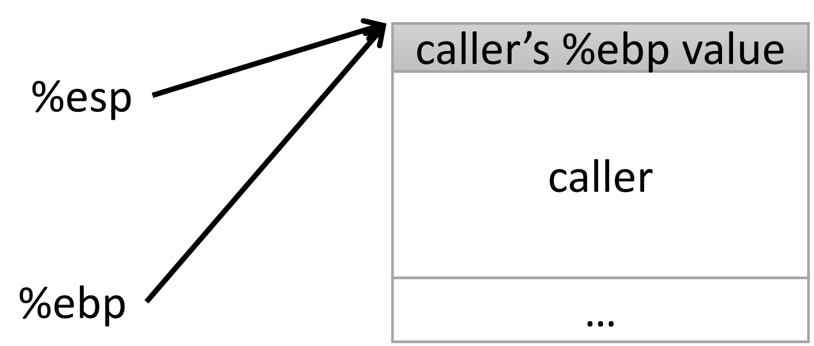
- To return, reverse this:



Frame Pointer

- Must maintain invariant:
  - The current function's stack frame is always between the addresses stored in %esp and %ebp

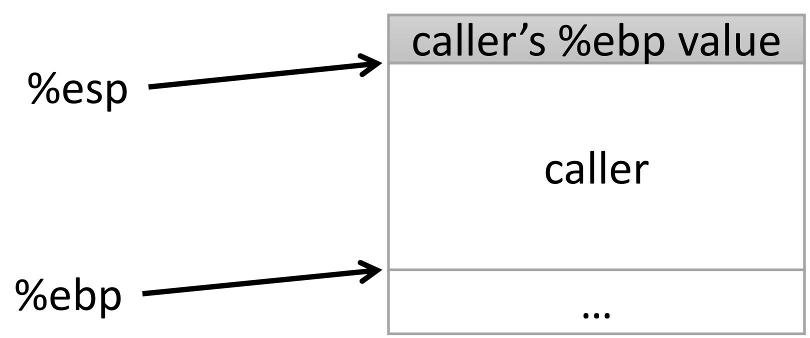
- To return, reverse this:
  1. set %esp = %ebp



Frame Pointer

- Must maintain invariant:
  - The current function's stack frame is always between the addresses stored in %esp and %ebp

- To return, reverse this:
  1. set %esp = %ebp
  2. popl %ebp

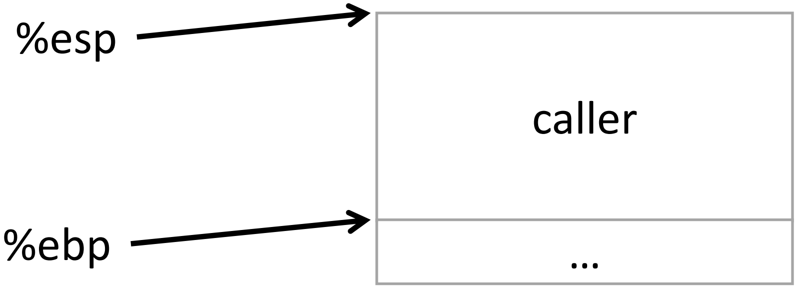


Frame Pointer

- Must maintain invariant:
  - The current function's stack frame is always between the addresses stored in %esp and %ebp

- To return, reverse this:
  1. set %esp = %ebp
  2. popl %ebp

IA32 has another convenience instruction for this: leave



Back to where we started.

Recall: Assembly While Loop

```
some_function:
    pushl %ebp
    movl %esp, %ebp

    # Your code here

    movl $10, %eax
    leave
    ret
```

Set up the stack frame for this function. (pushl %ebp, movl %esp, %ebp)

Store return value in %eax. (movl $10, %eax)

Restore caller's %esp, %ebp. (leave)

Lab 4: swap.s

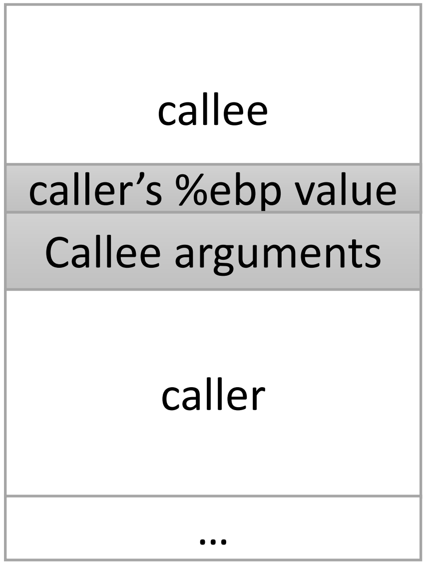
```
swap:
    pushl %ebp
    movl %esp, %ebp
    subl $16, %esp

    # Your code here

    leave
    ret
```

IA32 Calling Convention (gcc)

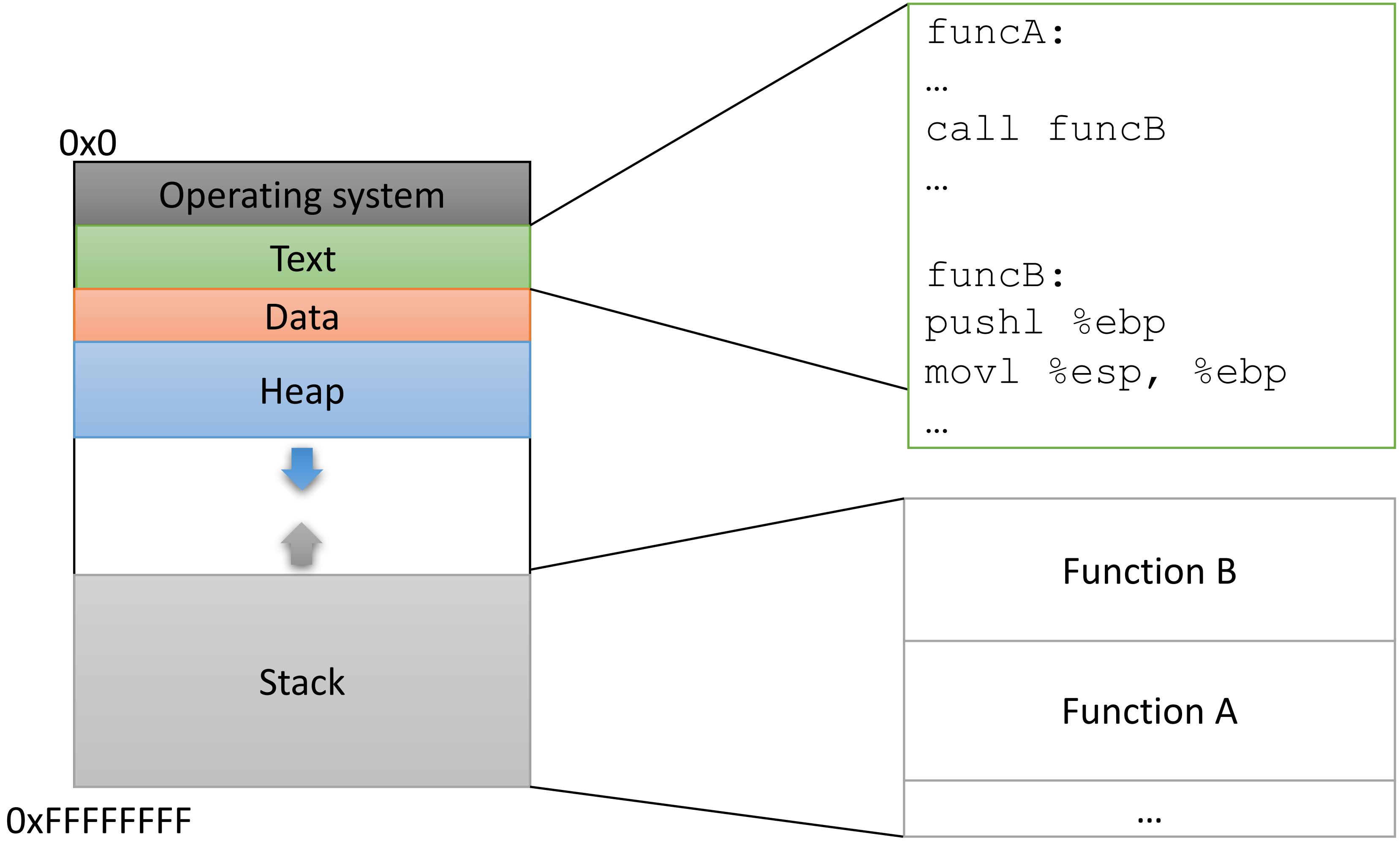
- In register %eax:
  - The return value
- In the callee’s stack frame:
  - The caller’s %ebp value (previous frame pointer)
- In the caller’s frame (shared with callee):
  - Function arguments
  - Return address (saved PC value)

Function Arguments

- Arguments are pushed onto the stack before the call instruction jumps into the callee.

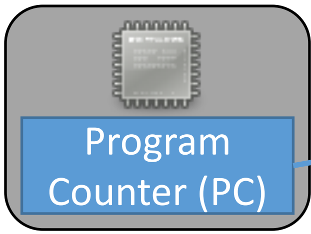


Instructions in Memory



Program Counter

Recall: PC stores the address of the next instruction.
(A pointer to the next instruction.)



What do we do now?

Follow PC, fetch instruction:

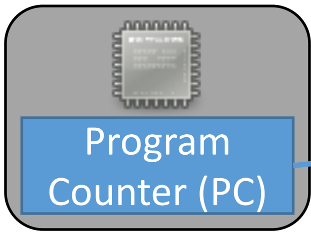
```
addl $5, %ecx
```

Text Memory Region

```
funcA:
addl $5, %ecx
movl %ecx, -4(%ebp)
…
call funcB
addl %eax, %ecx
…

funcB:
pushl %ebp
movl %esp, %ebp
…
movl $10, %eax
leave
ret
```

Program Counter

Recall: PC stores the address of the next instruction.
(A pointer to the next instruction.)



What do we do now?

Follow PC, fetch instruction:

`addl $5, %ecx`

Update PC to next instruction.

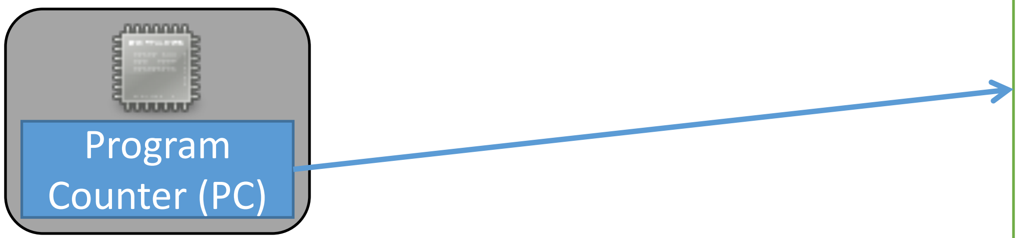
Execute the `addl`.

Text Memory Region

```
funcA:
addl $5, %ecx
movl %ecx, -4(%ebp)
…
call funcB
addl %eax, %ecx
…

funcB:
pushl %ebp
movl %esp, %ebp
…
movl $10, %eax
leave
ret
```

Program Counter

Recall: PC stores the address of the next instruction.
(A pointer to the next instruction.)



What do we do now?

Follow PC, fetch instruction:

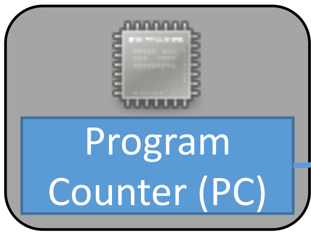
```
movl $ecx, -4(%ebp)
```

Text Memory Region

```
funcA:
addl $5, %ecx
movl %ecx, -4(%ebp)
…
call funcB
addl %eax, %ecx
…

funcB:
pushl %ebp
movl %esp, %ebp
…
movl $10, %eax
leave
ret
```

Program Counter

Recall: PC stores the address of the next instruction.
(A pointer to the next instruction.)



Text Memory Region

```
funcA:
addl $5, %ecx
movl %ecx, -4(%ebp)
…
call funcB
addl %eax, %ecx
…

funcB:
pushl %ebp
movl %esp, %ebp
…
movl $10, %eax
leave
ret
```

What do we do now?

Follow PC, fetch instruction:

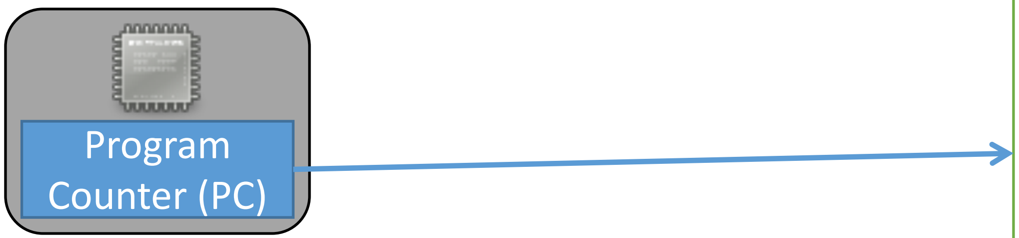
`movl $ecx, -4(%ebp)`

Update PC to next instruction.

Execute the `movl`.

Program Counter

Recall: PC stores the address of the next instruction.
(A pointer to the next instruction.)



Text Memory Region

```
funcA:
addl $5, %ecx
movl %ecx, -4(%ebp)
…
call funcB
addl %eax, %ecx
…

funcB:
pushl %ebp
movl %esp, %ebp
…
movl $10, %eax
leave
ret
```

What do we do now?

Keep executing in a straight line downwards like this until:

We hit a jump instruction.
We call a function.

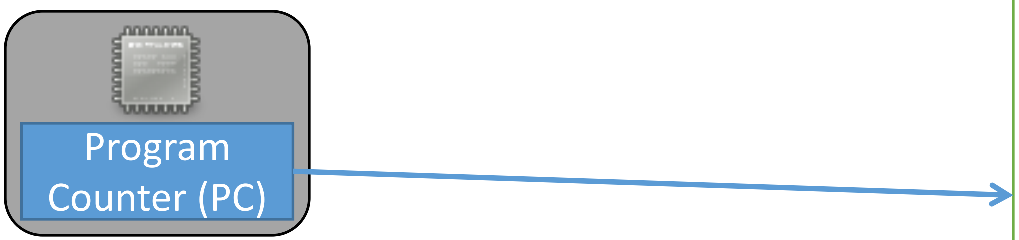
Changing the PC: Jump

- On a jump:
  - Check condition codes
  - Set PC to execute elsewhere (not next instruction)

- Do we ever need to go back to the instruction after the jump?

Maybe (and if so, we'd have a label to jump back to), but usually not.

Changing the PC: Functions

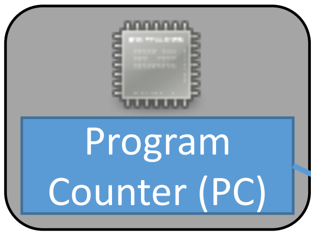


What we'd like this to do:

Text Memory Region

```
funcA:
addl $5, %ecx
movl %ecx, -4(%ebp)
…
call funcB
addl %eax, %ecx
…

funcB:
pushl %ebp
movl %esp, %ebp
…
movl $10, %eax
leave
ret
```

Changing the PC: Functions



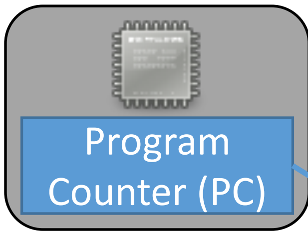
Text Memory Region

```
funcA:
addl $5, %ecx
movl %ecx, -4(%ebp)
…
call funcB
addl %eax, %ecx
…

funcB:
pushl %ebp
movl %esp, %ebp
…
movl $10, %eax
leave
ret
```

What we'd like this to do:

Set up function B's stack.

Changing the PC: Functions



Text Memory Region

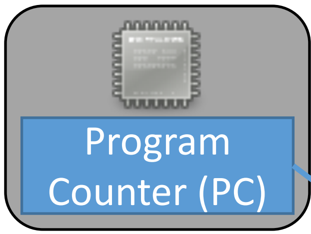
```
funcA:
addl $5, %ecx
movl %ecx, -4(%ebp)
…
call funcB
addl %eax, %ecx
…

funcB:
pushl %ebp
movl %esp, %ebp
…
movl $10, %eax
leave
ret
```

What we'd like this to do:

Set up function B's stack.

Execute the body of B, produce result (stored in %eax).

Changing the PC: Functions



What we'd like this to do:

Set up function B's stack.

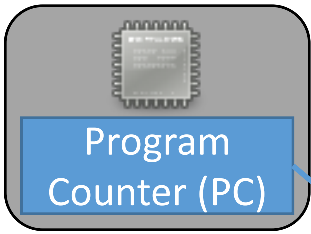
Execute the body of B, produce result (stored in %eax).

Restore function A's stack.

Text Memory Region

```
funcA:
addl $5, %ecx
movl %ecx, -4(%ebp)
…
call funcB
addl %eax, %ecx
…

funcB:
pushl %ebp
movl %esp, %ebp
…
movl $10, %eax
leave
ret
```

Changing the PC: Functions



Text Memory Region

```
funcA:
addl $5, %ecx
movl %ecx, -4(%ebp)
…
call funcB
addl %eax, %ecx
…

funcB:
pushl %ebp
movl %esp, %ebp
…
movl $10, %eax
leave
ret
```

What we'd like this to do:

Return:
Go back to what we were doing before funcB started.

Unlike jumping, we intend to go back!