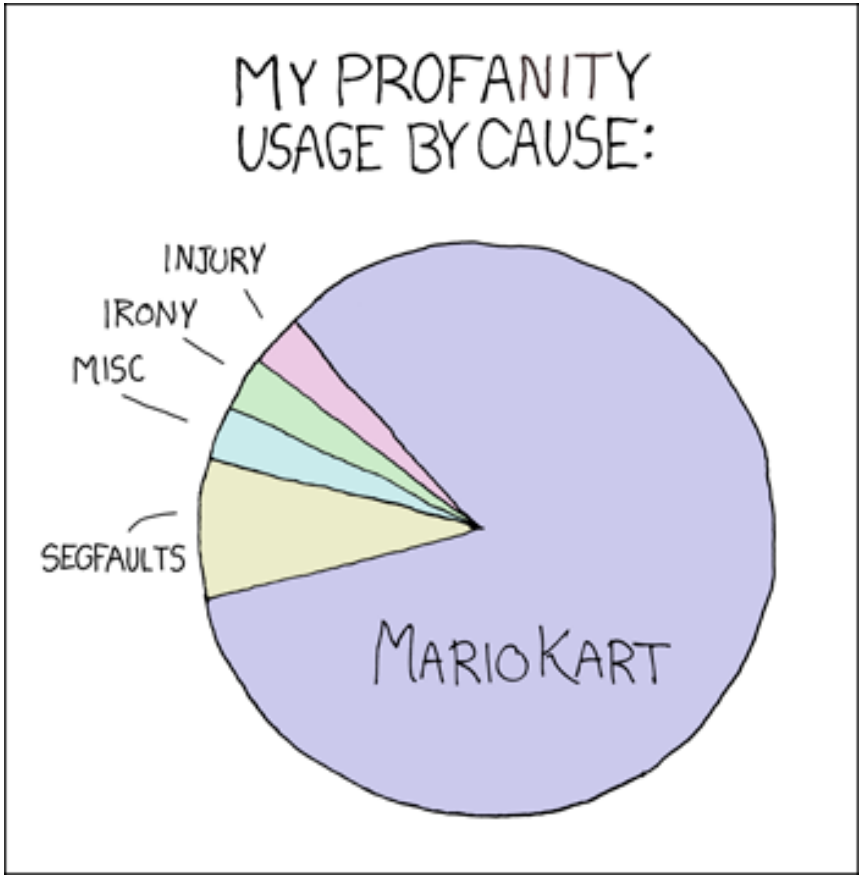
CSCI 2330


MY PROFANITY USAGE BY CAUSE:
INJURY
IRONY
MISC
SEGFAULTS
MARIOKART

TODO

- Assignment 3 assigned due October 28

Lab 2 graded

- `grade.txt` in `assignments/lab2/`
- You'll find feedback on any correctness issues

Lab 3 warning

- Don’t use AI!

Duff’s device

```
void duffs_device(char *to, char *from, unsigned long count) {
    if (count == 0) return;

    unsigned long n = count / 8 + (count % 8 != 0);

    switch (count % 8) {
    case 0: do { *to++ = *from++;
    case 7:      *to++ = *from++;
    case 6:      *to++ = *from++;
    case 5:      *to++ = *from++;
    case 4:      *to++ = *from++;
    case 3:      *to++ = *from++;
    case 2:      *to++ = *from++;
    case 1:      *to++ = *from++;
            } while (--n);
    }
}
```

Duff’s device

```
void duffs_device(char *to, char *from, unsigned long count) {
    unsigned long n = count / 8;
    switch (count % 8) {
    case 0:
        while (n > 0) {
            --n;
            *to++ = *from++;
    case 7: *to++ = *from++;
    case 6: *to++ = *from++;
    case 5: *to++ = *from++;
    case 4: *to++ = *from++;
    case 3: *to++ = *from++;
    case 2: *to++ = *from++;
    case 1: *to++ = *from++;
        }
    }
}
```

Next up: function calls

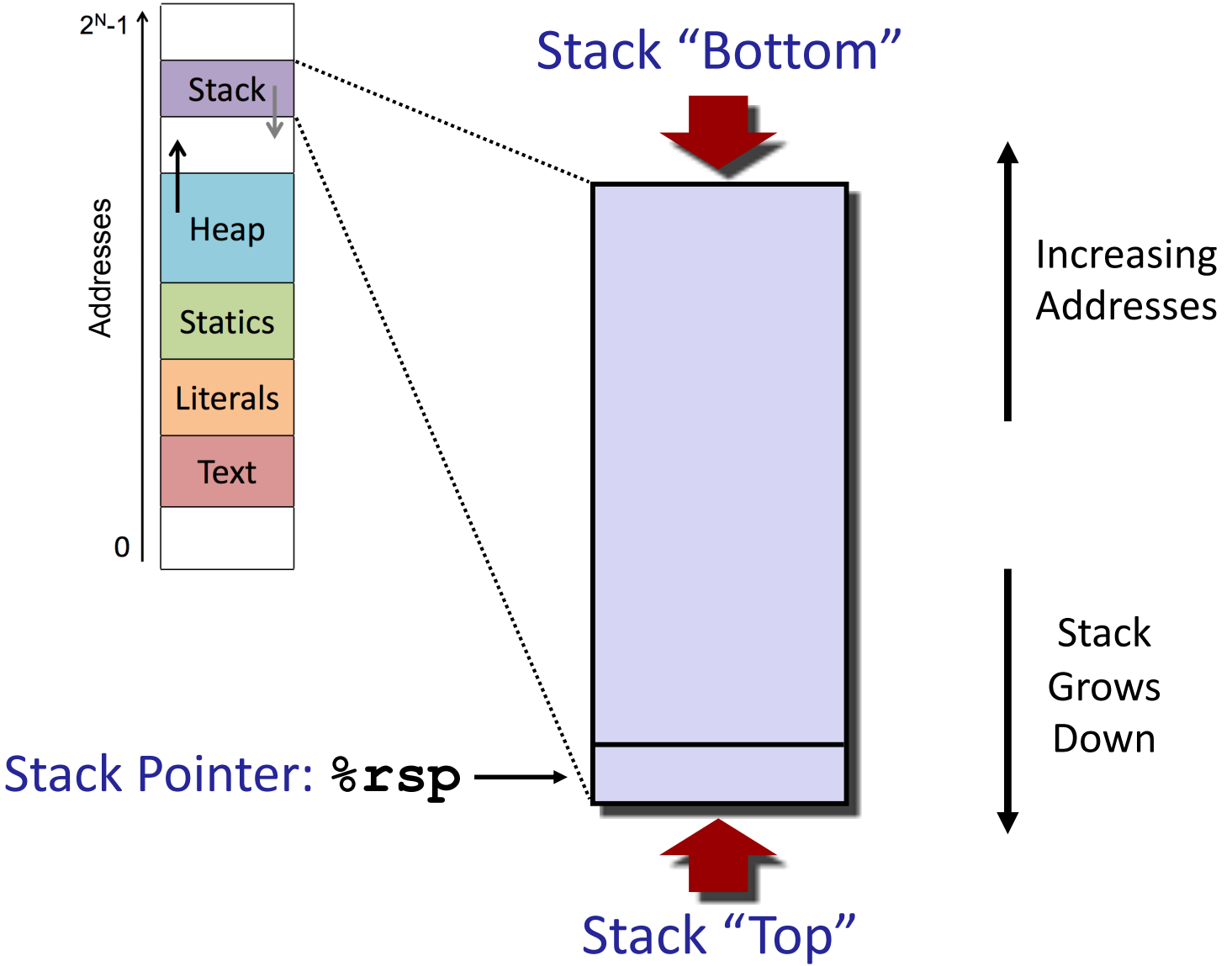
- We’ve seen a glimpse
- But how is the stack actually managed?
- How does a `ret` instruction “know” where to return? How does it know where it was called from?

Discuss three function call mechanisms

- Passing control
  - Jumping to start of procedure, back to return point
- Passing data
  - Arguments, return value (may not fit into registers)
- Memory management
  - Local variables (some may not fit into registers)
  - How do stack variables get allocated/deallocated?

Recall how stack operates

- Stack grows *down*
- *Top* of stack is at the *bottom*
- `%rsp` points to “top” of stack (the lowest valid address of stack)...


2N-1
Stack
Heap
Statics
Literals
Text
0
Addresses
Stack "Bottom"
Increasing Addresses
Stack Grows Down
Stack Pointer: %rsp
Stack "Top"

%rsp

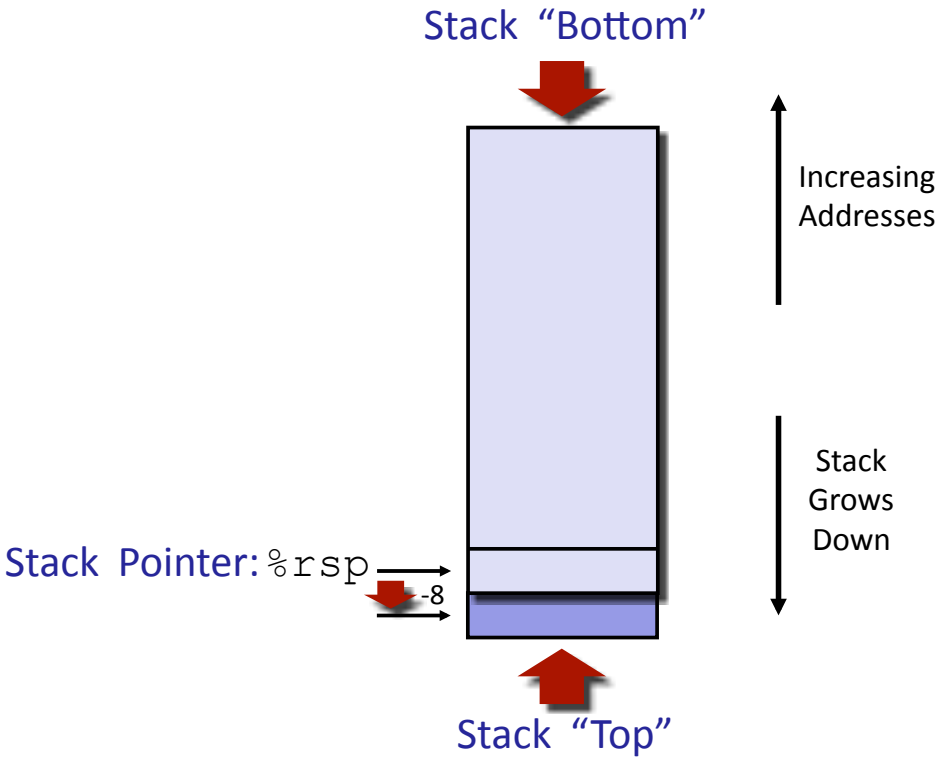
- `%rsp` points to “top” of stack (the lowest valid address of stack)
- `%rsp` has special instructions to manage its value
  - Although it is *also* a general purpose register
- If we want the stack to grow, how should we change the value of `%rsp`?
  - What about to shrink the stack?

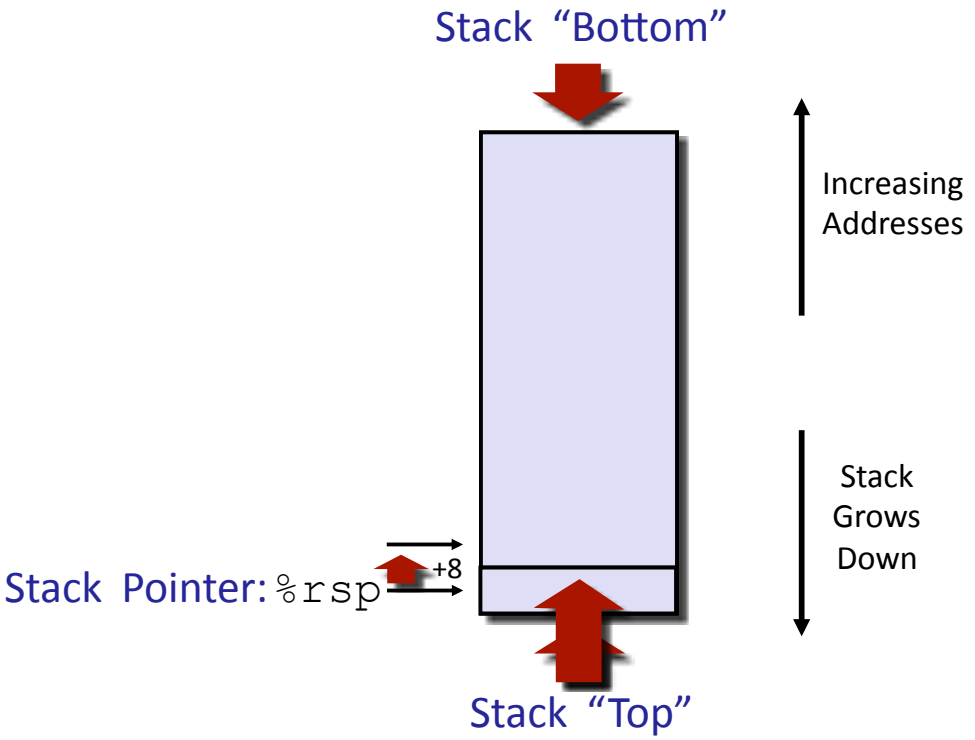
Stacks

- What are the basic operations of a stack (as a data structure)?

Stacks

- What are the basic operations of a stack (as a data structure)?
  - Push and pop!
- Let’s look at two new instructions…
  - `pushq <source operand>`
  - `popq <destination operand>`





`pushq Src`

1. Decrement **`%rsp`** by 8
2. Write **Src** to **`(%rsp)`**

`popq Dest`

1. Read **`(%rsp)`** to **Dest**
2. Increment **`%rsp`** by 8

We don’t *need* these instructions

- `pushq %rax`
  - `subq $8, %rsp` `# Make room on the stack`
  - `movq %rax, (%rsp)` `# Store value on stack`
- `popq %rax`
  - `movq (%rsp), %rax` `# Read value from stack`
  - `addq $8, %rsp` `# Remove it from stack`

Which of these are valid instructions?

- `pushq %rax`
- `pushq 4(%rax, %rdx, 2)`
- `pushq $10`
- `popq %rax`
- `popq 4(%rax, %rdx, 2)`
- `popq $10`

Which of these are valid instructions?

- `pushq %rax` `# Valid`
- `pushq 4(%rax, %rdx, 2)` `# Valid`
- `pushq $10` `# Valid`
- `popq %rax` `# Valid`
- `popq 4(%rax, %rdx, 2)` `# Valid`
- `popq $10` `# Invalid`

Let’s revisit two other instructions

- What instructions do we use to call functions and return from them?

Let’s revisit two other instructions

- What instructions do we use to call functions and return from them?
  - `callq` and `ret`
- Both of these are unconditional jumps
- `callq <address>` jumps to function address
- `ret` jumps to just after original call site
  - But it doesn’t take an operand. How does it know where to jump if we don’t tell it!?

Calling and returning

- We couldn’t specify the return address in `ret` even if we wanted to
- How would the function “know” where it is called from?
- What if it is called from five different places in a program? How does it know where to return?
- In the following example, where do we want to jump after executing `ret`…

```
long x = add(3, 5);
doSomething(x);
```

```
long add(long x, long y) {
  return x + y;
}
```

```
movq $3, %rdi
movq $5, %rsi
callq add
movq %rax, %rdi
callq doSomething
```

push retaddr

return address

```
add:
  movq %rdi, %rax
  addq %rsi, %rax
  retq
```

pop retaddr & jump

Calling and returning

- We want to return to the *return address*
- The instruction after the calling `callq`
- Every *function call* has a return address
  - NOT: every function
- Where could we keep track of return addresses?

Calling and returning

- We want to return to the *return address*
- The instruction after the calling `callq`
- Every *function call* has a return address
  - NOT: every function
- Where could we keep track of return addresses?
  - ~~Registers?~~ Not enough of them! Stack!

Calling and returning

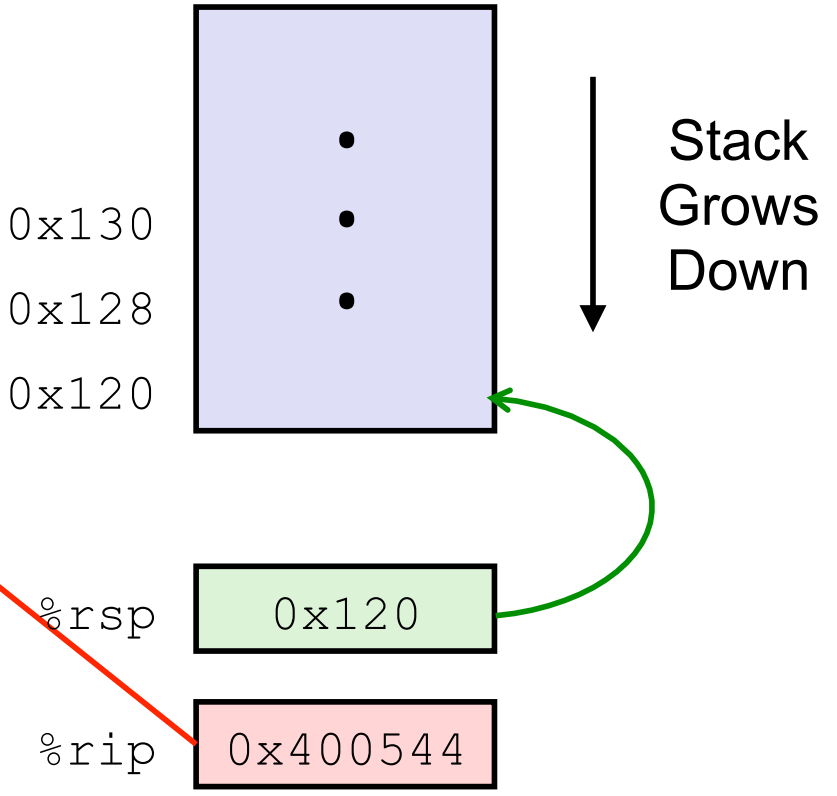
- `callq` pushes the return address to the stack
- `ret` pops it and jumps to it

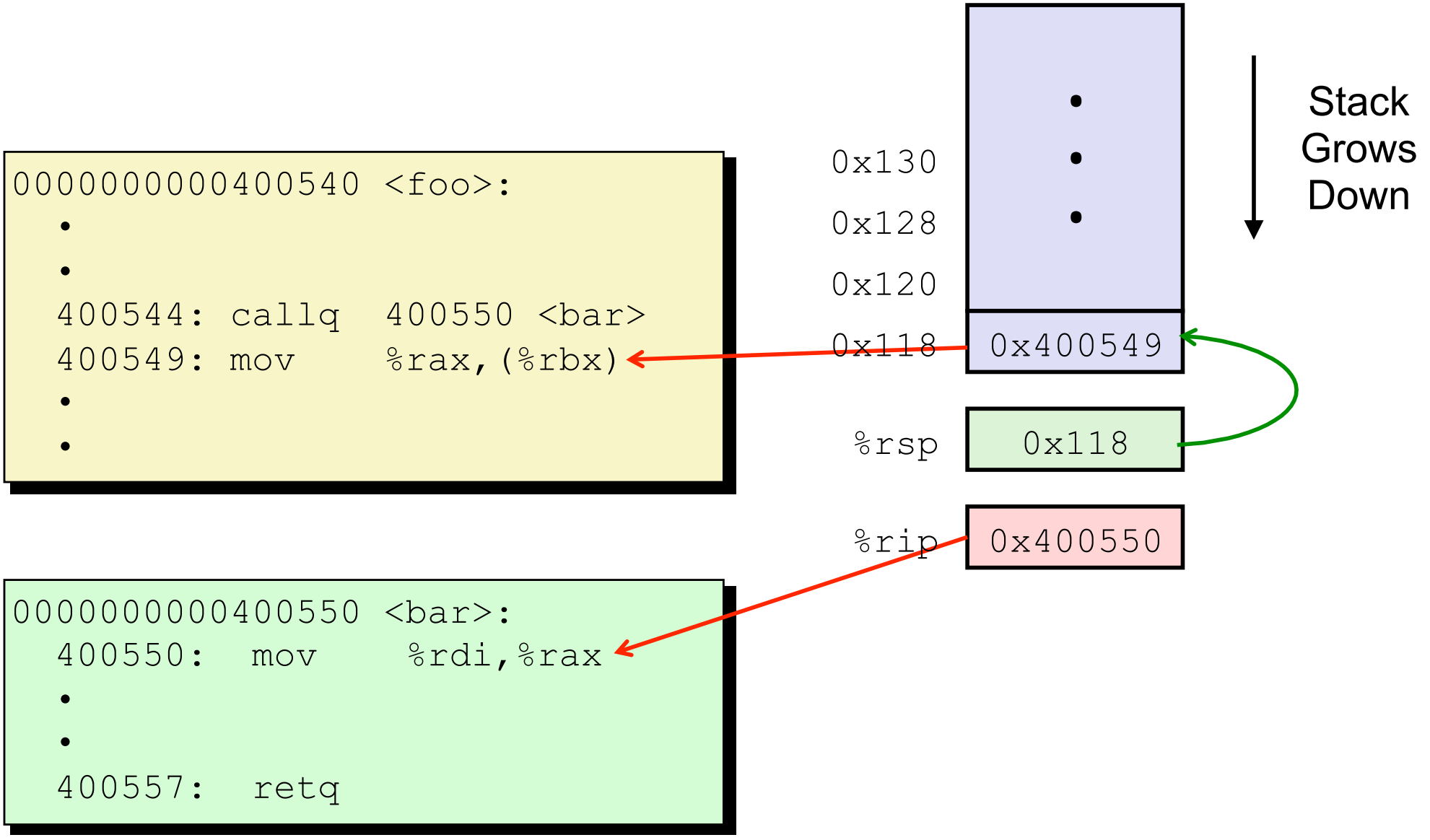
Calling and returning

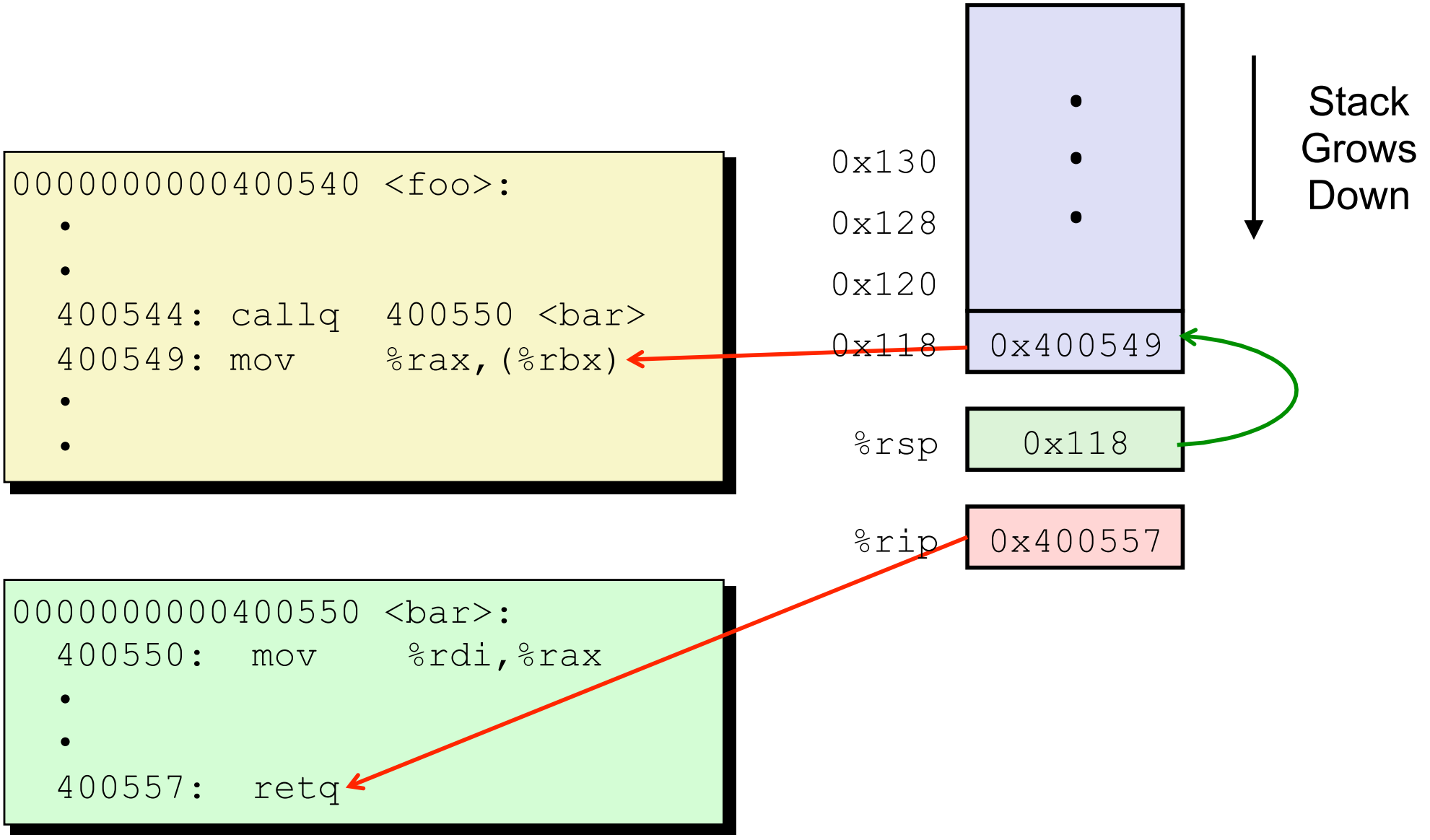
- `callq`:
  - `subq $8, %rsp`
  - `movq %rip + length_of_callq_instruction, (%rsp)`
  - `jmp target`
- `ret`:
  - `jmp (%rsp)`
  - `addq $8, %rsp`
- Let’s walk through an example…

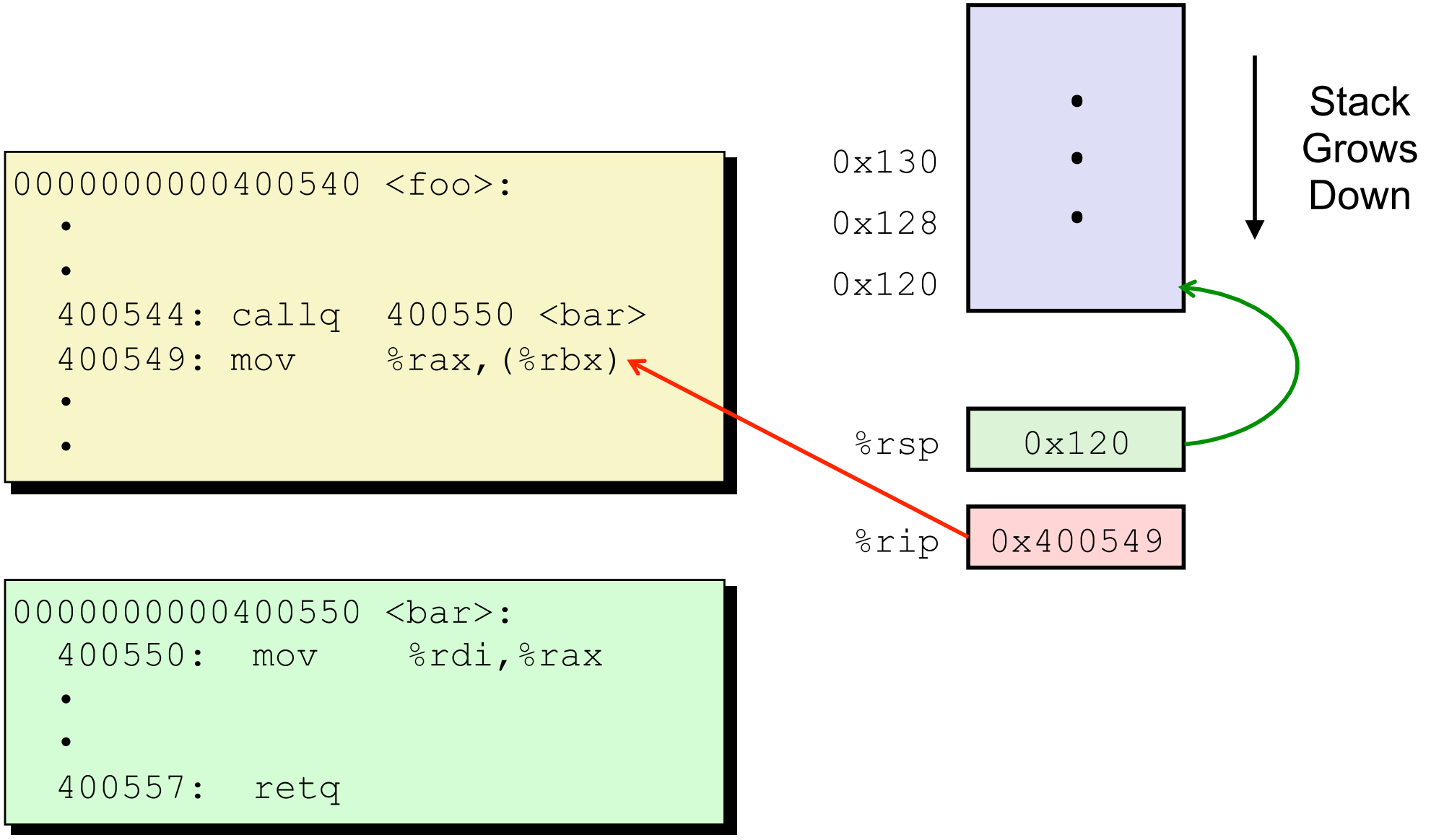
```
0000000000400540 <foo>:
   •
   •
   400544: callq  400550 <bar>
   400549: mov    %rax,(%rbx)
   •
   •
```

```
0000000000400550 <bar>:
   400550:  mov    %rdi,%rax
   •
   •
   400557:  retq
```



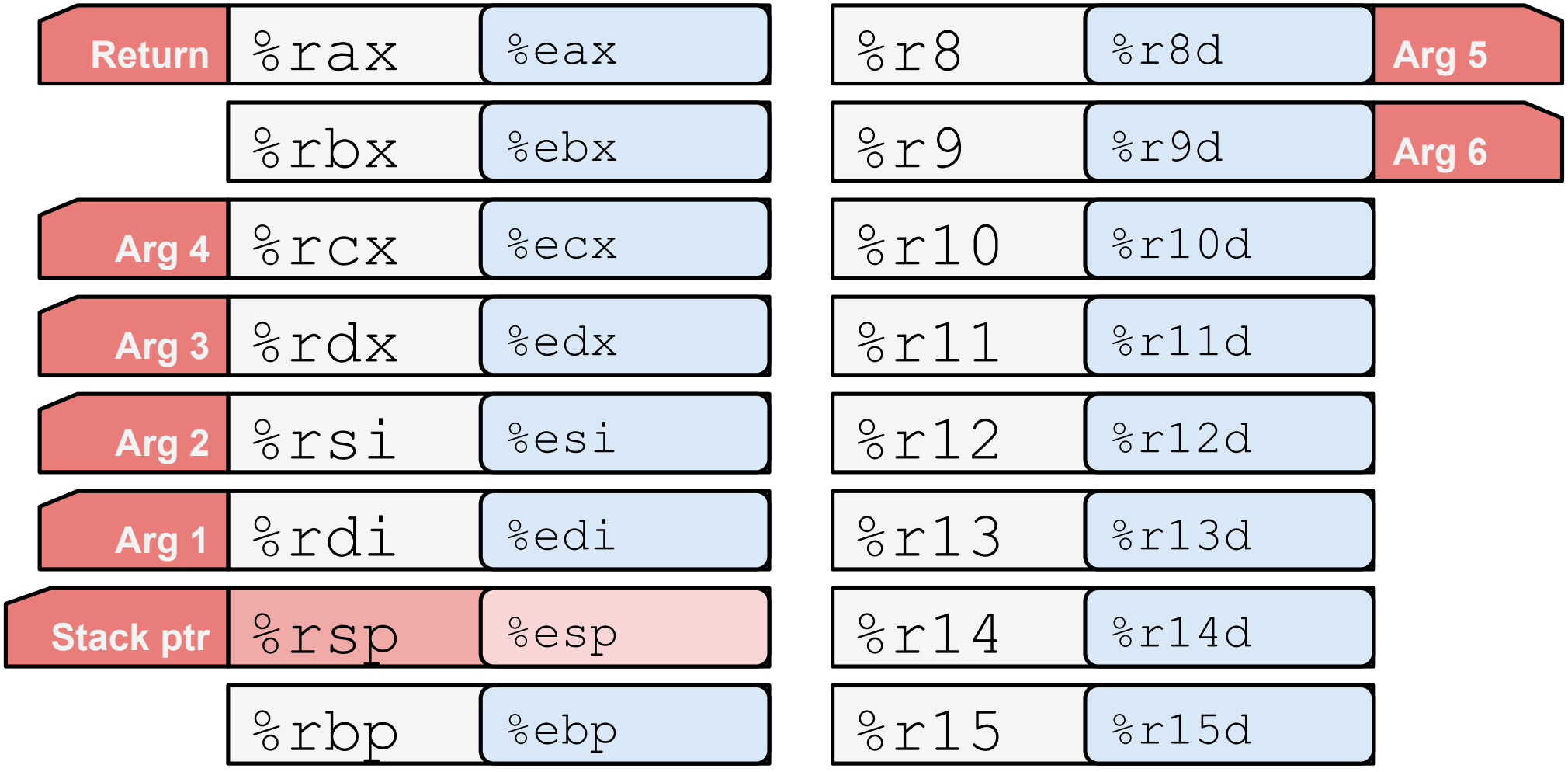

0000000000400540 <foo>:
•
•
400544: callq 400550 <bar>
400549: mov %rax,(%rbx)
•
•
•
0000000000400550 <bar>:
400550: mov %rdi,%rax
•
•
400557: retq
0x130
0x128
0x120
0x118
0x400549
%rsp
0x118
%rip
0x400550
Stack
Grows
Down


0000000000400540 <foo>:
•
•
400544: callq 400550 <bar>
400549: mov %rax,(%rbx)
•
•
•
0000000000400550 <bar>:
400550: mov %rdi,%rax
•
•
400557: retq
0x130
0x128
0x120
0x118
0x400549
%rsp
0x118
%rip
0x400557
Stack
Grows
Down


0000000000400540 <foo>:
•
•
400544: callq 400550 <bar>
400549: mov %rax,(%rbx)
•
•
•
0000000000400550 <bar>:
400550: mov %rdi,%rax
•
•
400557: retq
0x130
0x128
0x120
Stack Grows Down
%rsp
0x120
%rip
0x400549

Passing data

- Wait, I thought we already knew how to pass arguments to a function?
- And to return values from one?

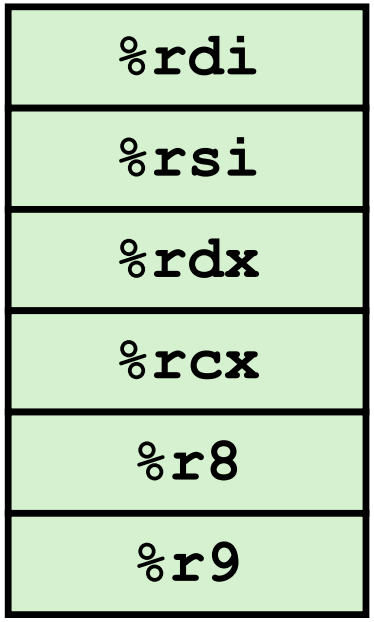

Return
%rax
%eax
%rbx
%ebx
Arg 4
%rcx
%ecx
Arg 3
%rdx
%edx
Arg 2
%rsi
%esi
Arg 1
%rdi
%edi
Stack ptr
%rsp
%esp
%rbp
%ebp
%r8
%r8d
Arg 5
%r9
%r9d
Arg 6
%r10
%r10d
%r11
%r11d
%r12
%r12d
%r13
%r13d
%r14
%r14d
%r15
%r15d

Problem

- What if we have more than six arguments?
- Now we have a problem…

Registers

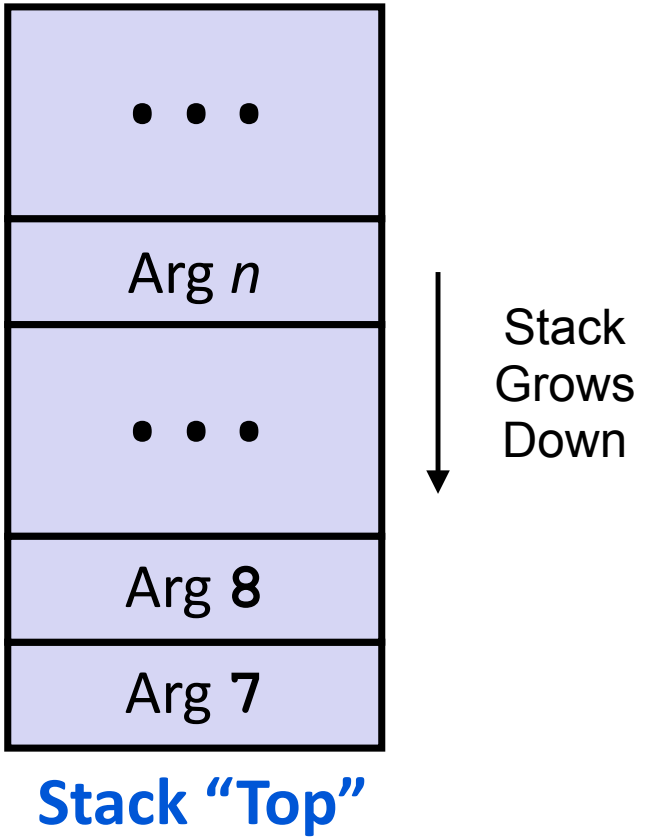
- First 6 arguments



- Return value



Stack



Push arguments in reverse order

- `f(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)`:

```
movq $1, %rdi
movq $2, %rsi
movq $3, %rdx
movq $4, %rcx
movq $5, %r8
movq $6, %r9
pushq $10
pushq $9
pushq $8
pushq $7
callq f
addq $32, %rsp
```

32-bit x86

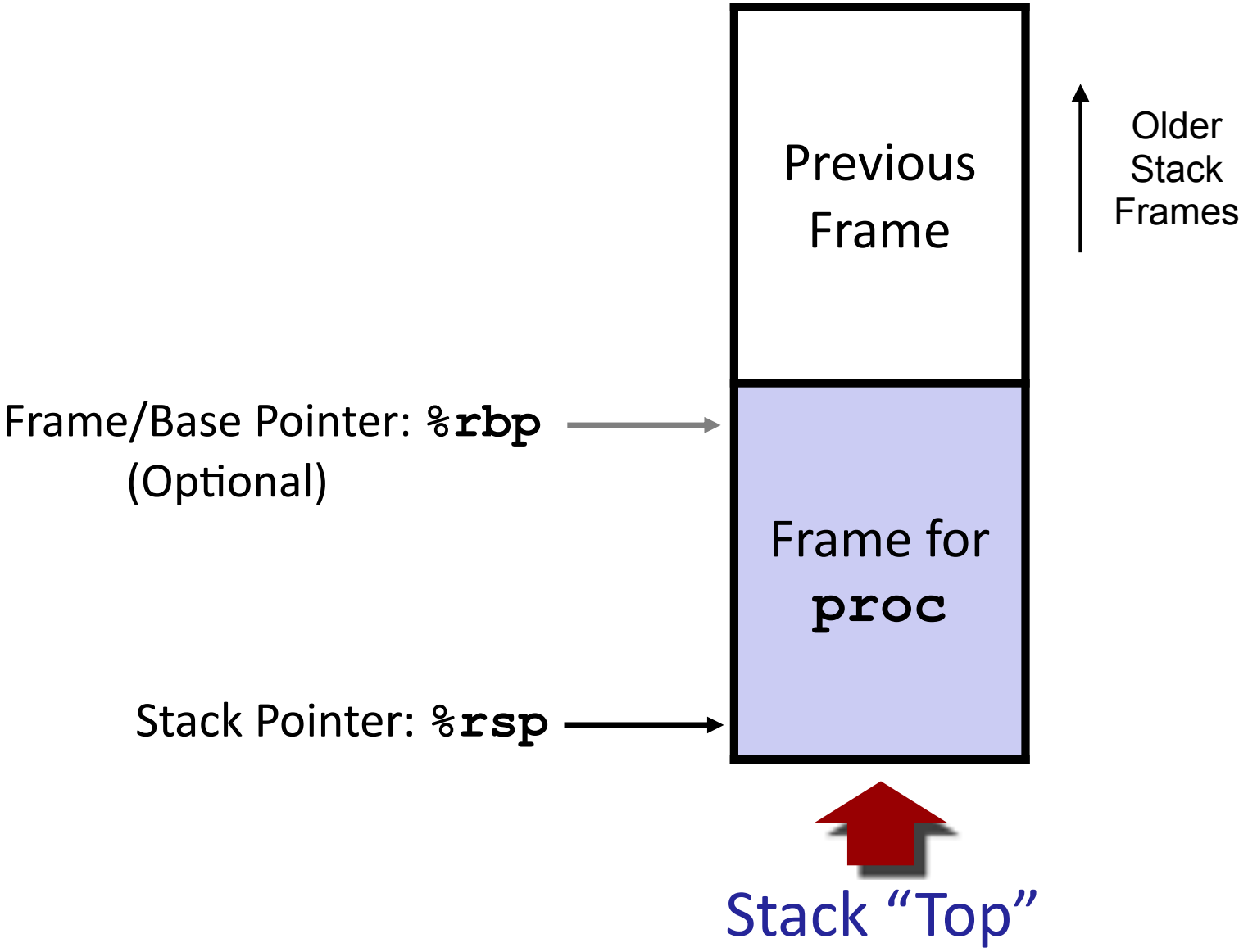
- 32-bit x86 passed all arguments via stack!
- Although there were other calling conventions that would use some registers (e.g., fastcall)
- x86-64 increase the number of registers
- New calling convention could afford to use some for arguments!

Big picture

- Where do they fit into this picture?
- Per function *call* we need to track:
  - Return address
  - Arguments (if more than 6)
  - Local variables
- We can’t track it per-function (e.g., recursion)
- Where do we keep it? *Stack frames*

Stack frames

- A stack frame is what we call the region of the stack belonging to a function call
- It contains the function’s local variables
  - When calling a function, the caller also puts arguments (past 6) and the return address in its own stack
- We can think of the entire stack as a sequence of stack frames…


Previous Frame
Older Stack Frames
Frame/Base Pointer: %rbp
(Optional)
Frame for proc
Stack Pointer: %rsp
Stack "Top"

Stack trace

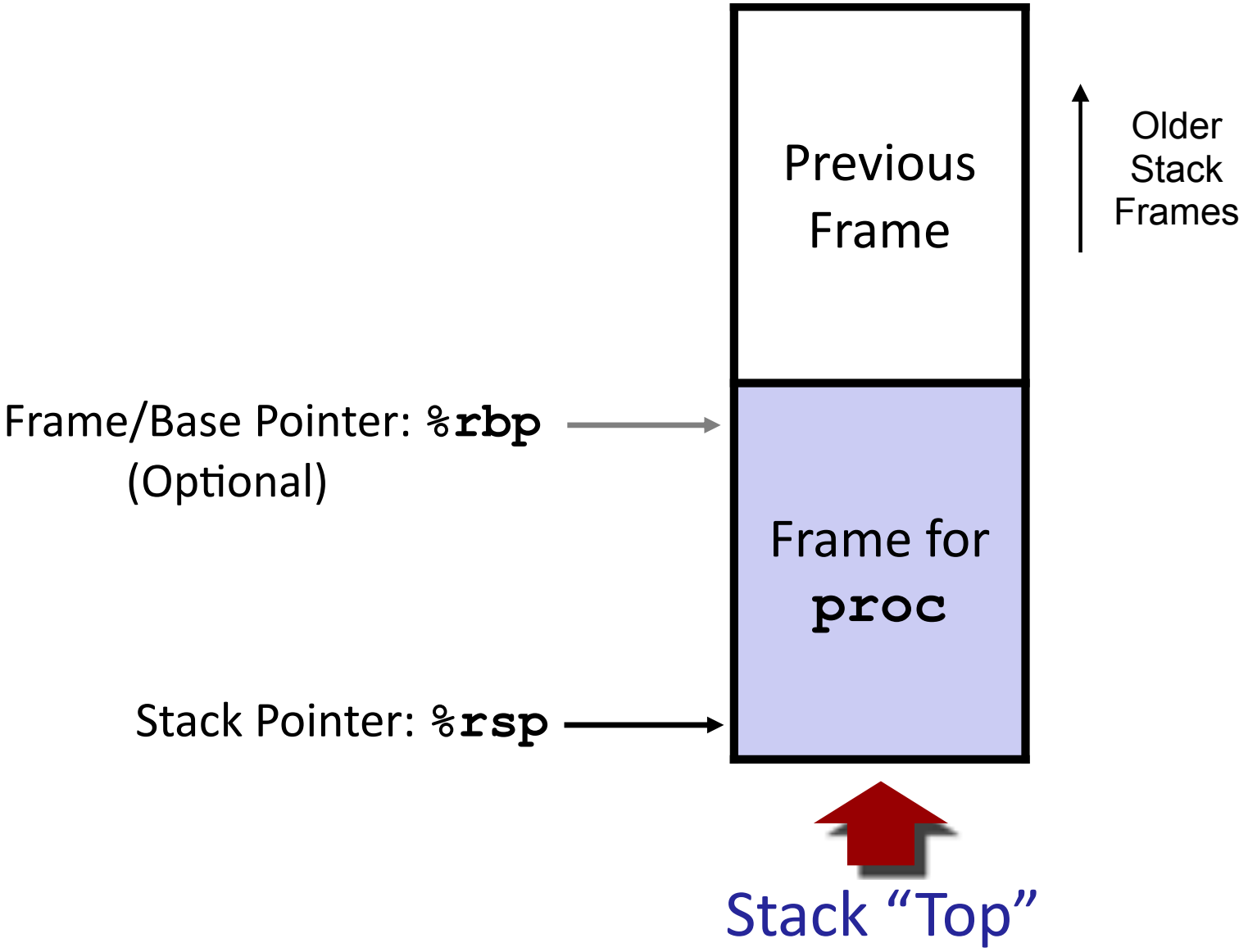
- Stack frames are what we see in Java when an exception is thrown
- In gdb, we can see the same via:
- `backtrace [bt]`

Accessing data in our stack frame

- We can access data using `%rsp`
- Since %rsp points to the “top” of the stack (lowest address) we will generally be using non-negative offsets
  - `8(%rsp)`
  - `4(%rsp)`
  - `0(%rsp)`

Problem: %rsp can change

- If we need to grow our stack frame, we might need to subtract from %rsp
- This makes keeping track of which offset to use difficult
- If `x` is at `4(%rsp)`, if we `subq -4, %rsp`, now it is at `8(%rsp)`!


Previous Frame
Older Stack Frames
Frame/Base Pointer: %rbp
(Optional)
Frame for proc
Stack Pointer: %rsp
Stack "Top"

Solutions

- Compilers can carefully keep track and try to always use the correct offset
- Or use %rbp as a “block pointer” pointing to the bottom (highest address) of your stack frame
- It doesn’t change when the stack grows because it is at the bottom
- If `x` is at -12(`%rbp`), if we `subq -4, %rsp`, it is *still* at -12(`%rbp`)!

Optimizing compilers

- Optimizing compilers prefer to carefully keep track to use the correct offset
- This frees up %rbp to use for anything else
- However, since debuggers often have difficulty reconstructing `%rsp`-based offsets, using `%rbp` can also make your program easier to debug

Example call chain

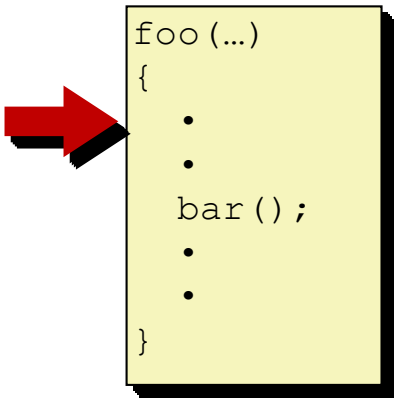
- Let’s look at an example of a call chain
- This one is interesting because it features a recursive function
- Let’s see how stack frames change as functions are called…

```
foo(…)
{
   •
   •
   bar();
   •
   •
}
```

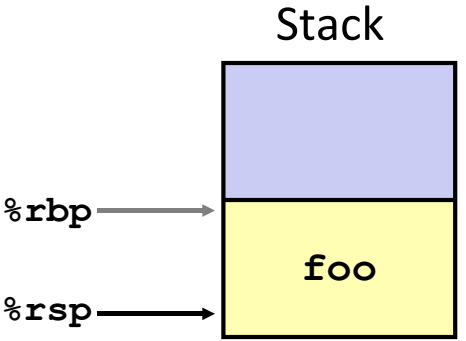
```
bar(…)
{
   • • •
   baz();
   • • •
   baz();
   • • •
}
```

```
baz(…)
{
   •
   •
   baz();
   •
   •
}
```

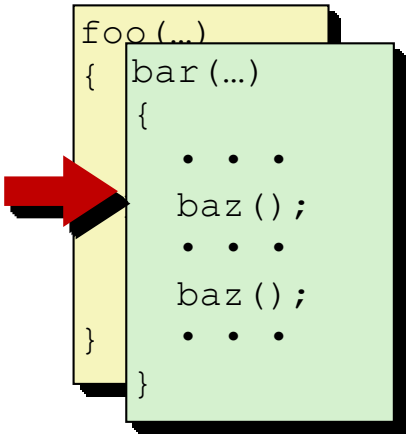
Procedure **`baz()`** is recursive



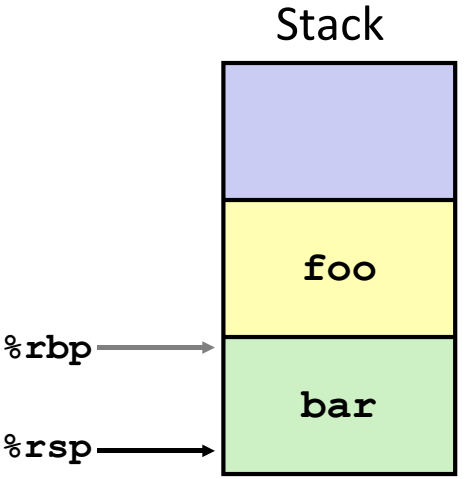
foo



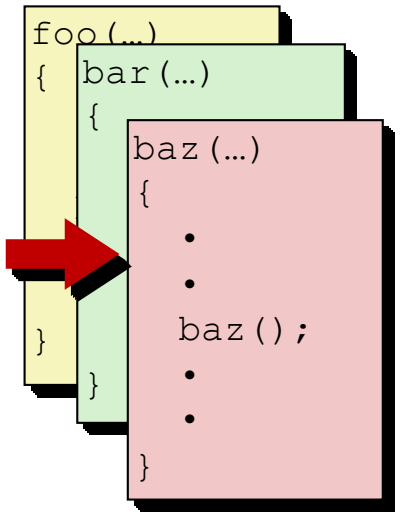
call **bar**



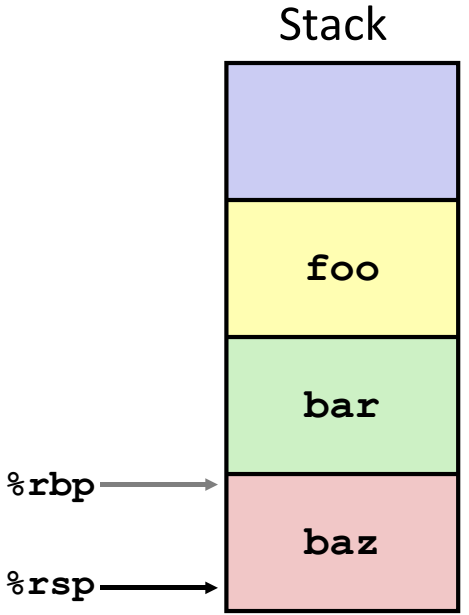




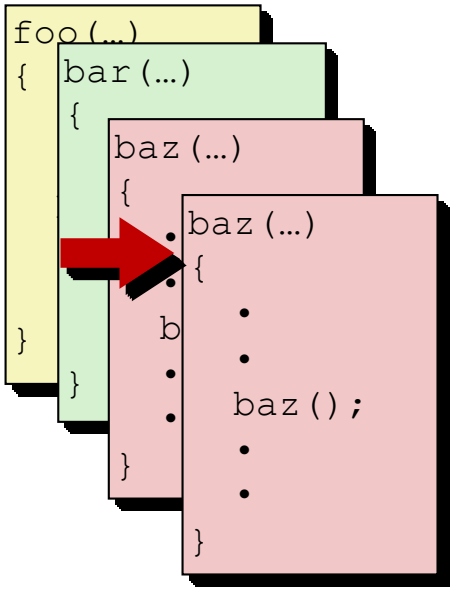
call **baz**



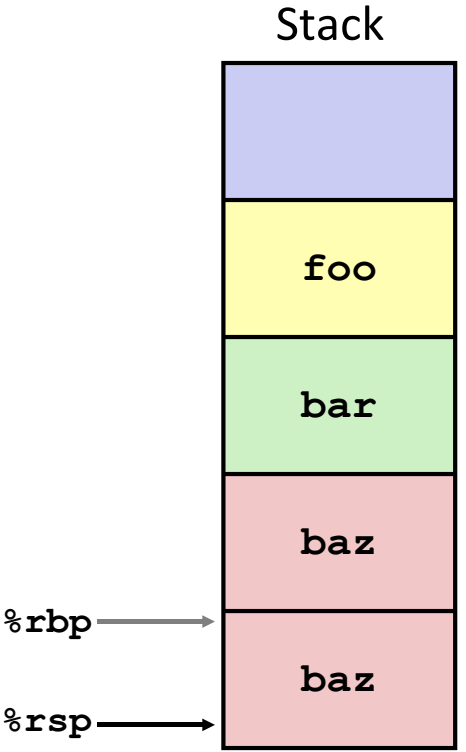




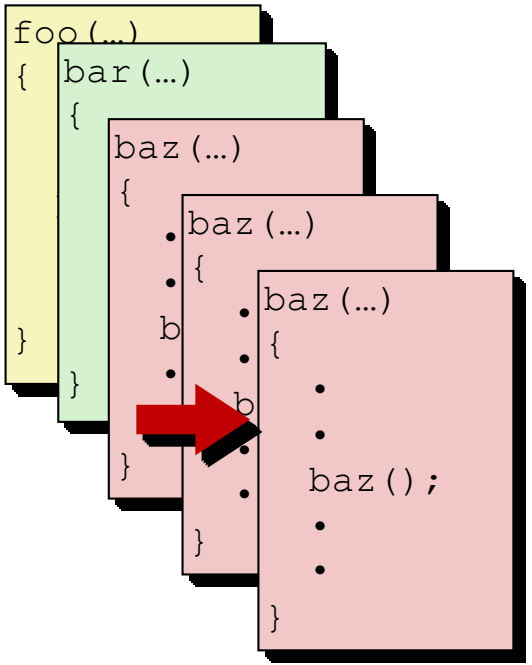
call **baz**
(recurse)





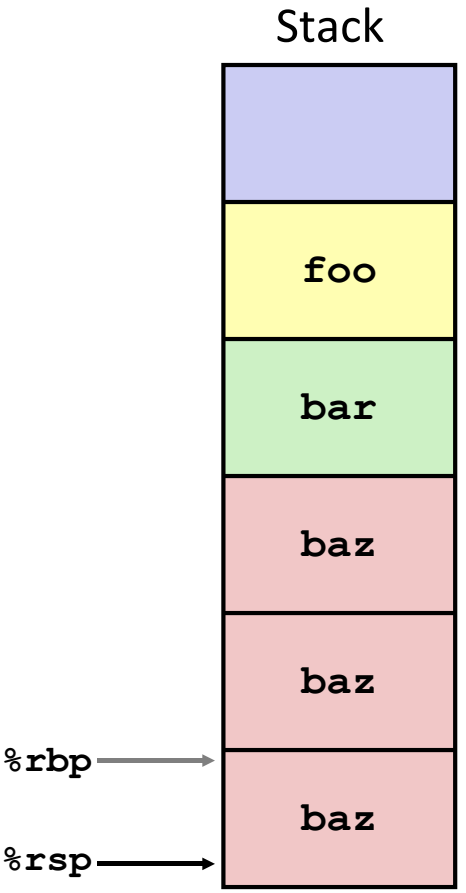


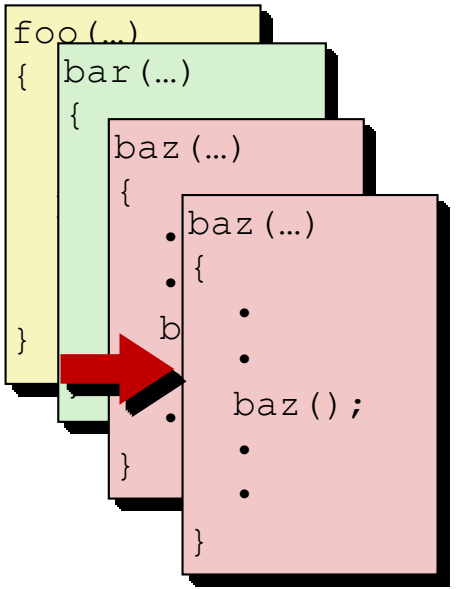
call **baz**
(recurse)



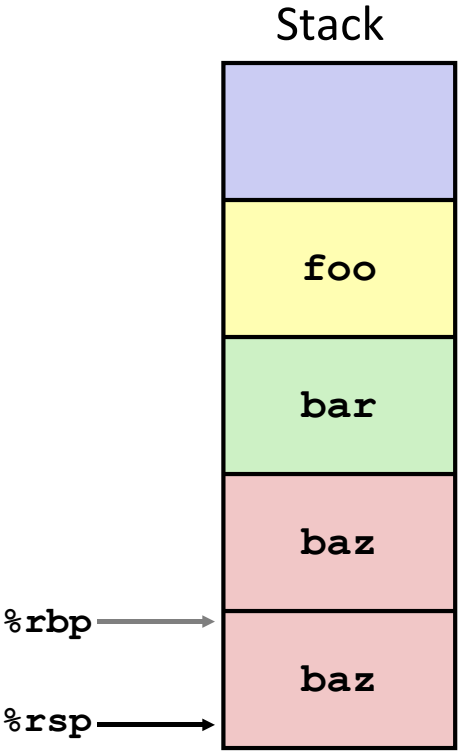
return
(base case)



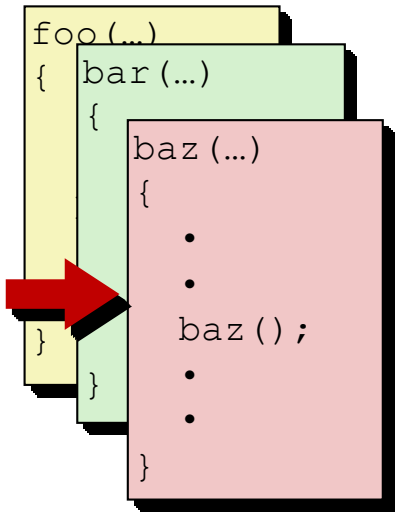




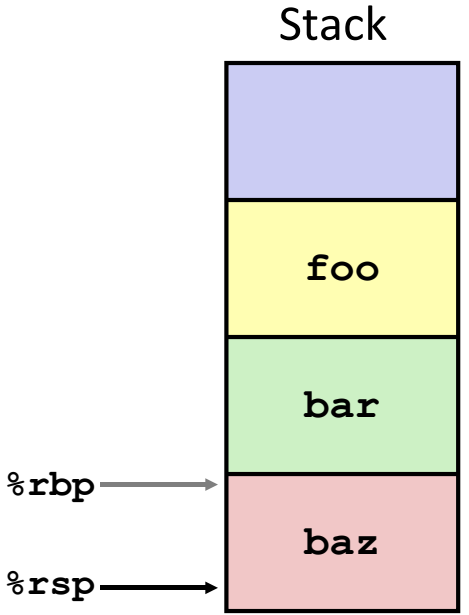




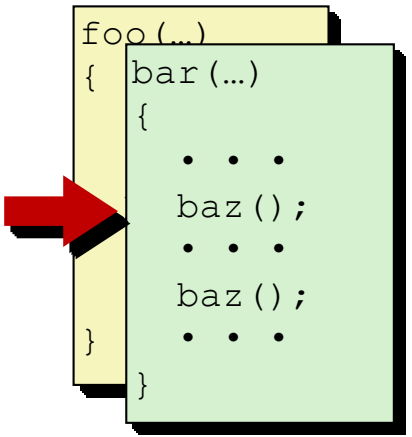
(unwinding recursion)



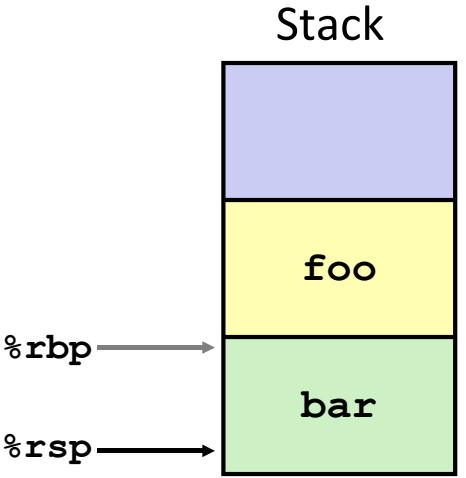




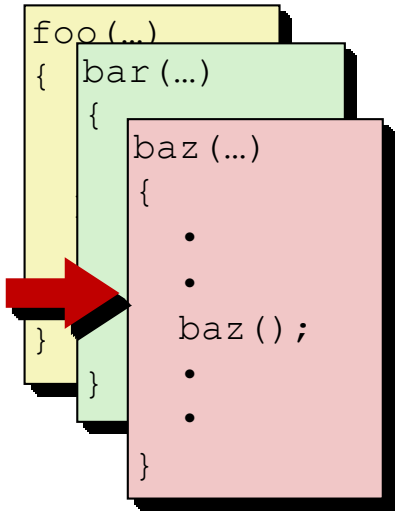
return
(unwinding recursion)

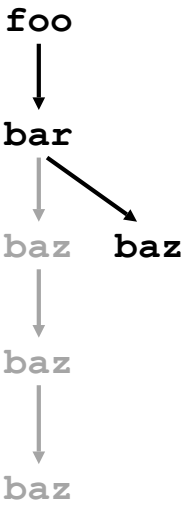


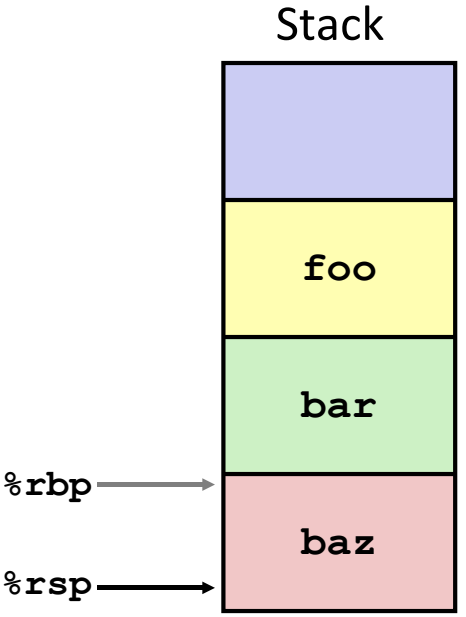




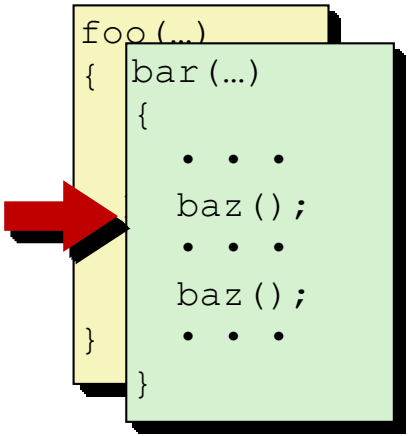
call **baz**

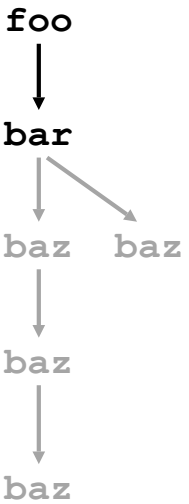


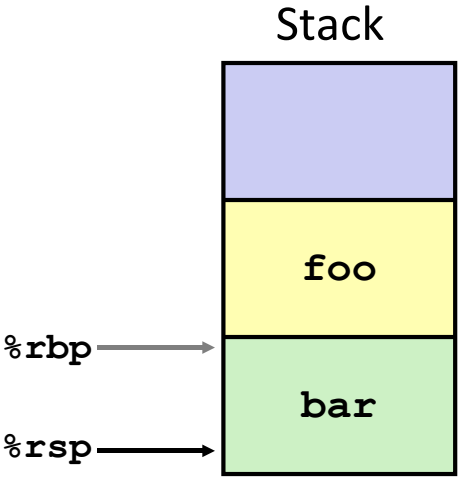




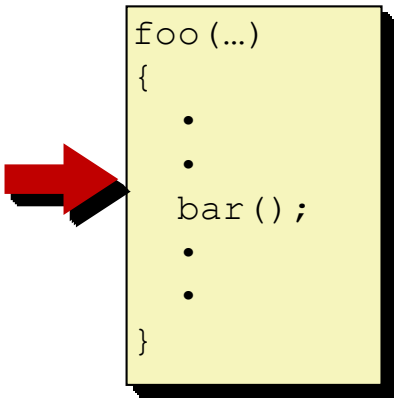
return
(base case)

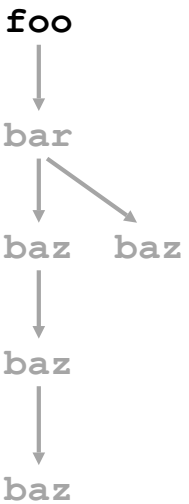


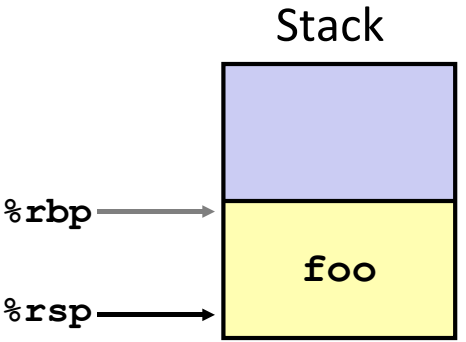




return



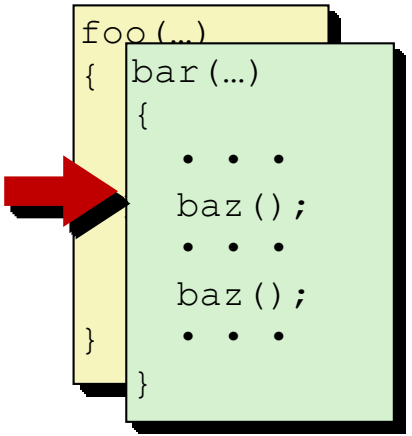




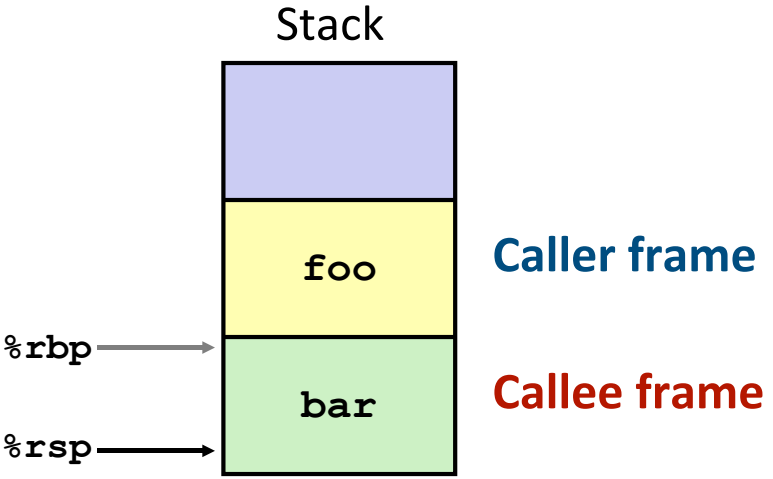
return

Data organization in stack frames

- First some terminology:
- The *caller* – the function making a call
- The *callee* – the function being called
- Note a callee can quickly also become a caller…

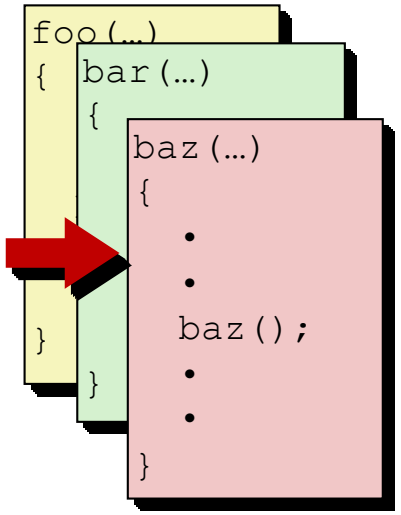




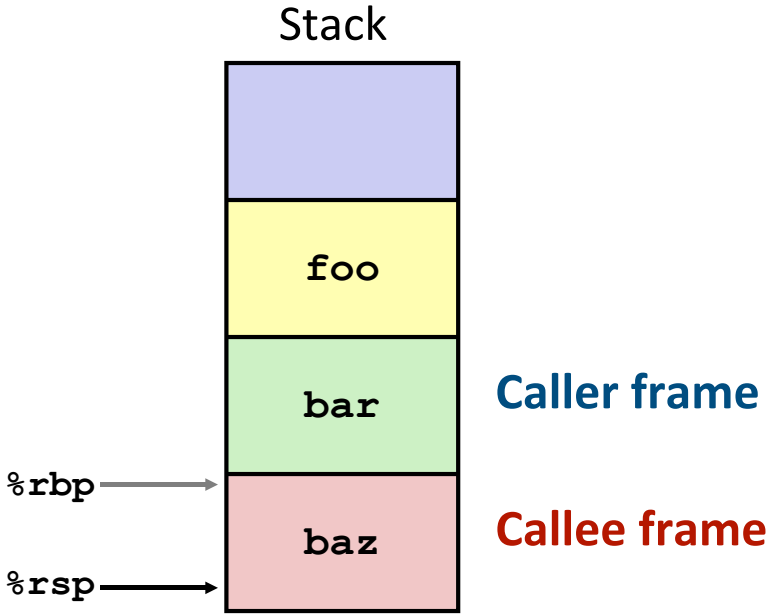


foo is the **caller**

bar is the **callee**





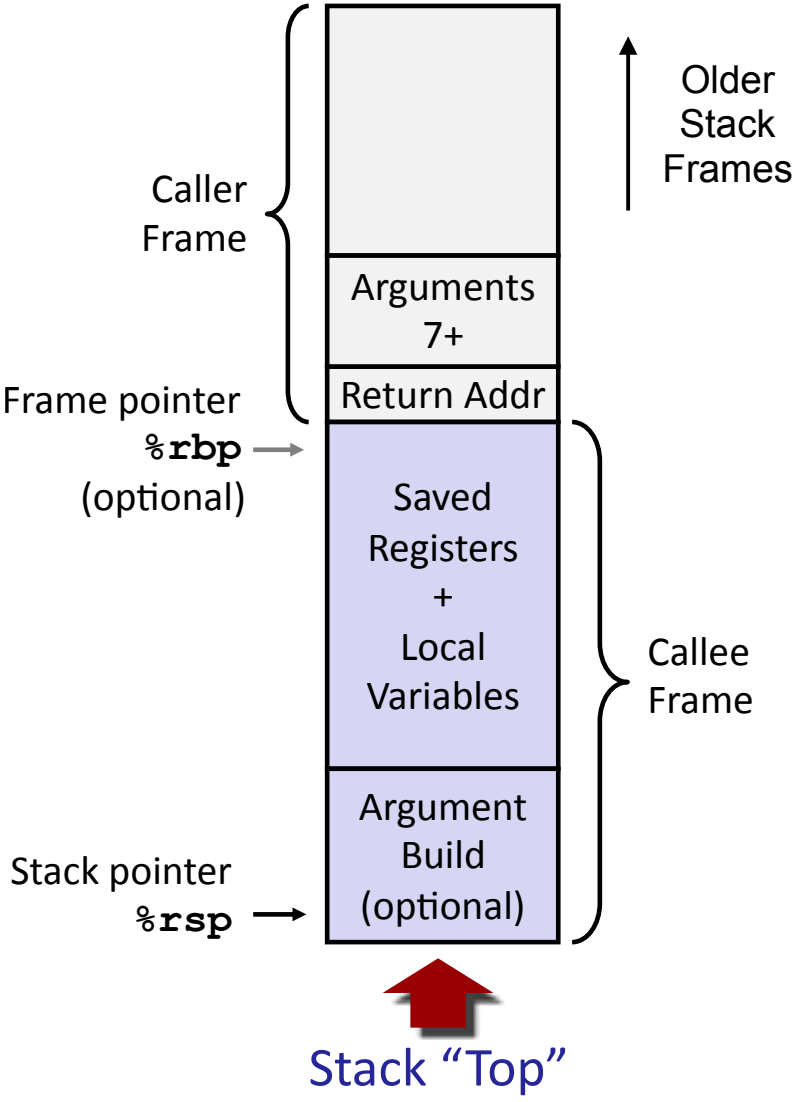


bar is the **caller**

baz is the **callee**

Data organization in stack frames

- Suppose we are in function A() about to call function B(). What do we first push onto the stack?
  - Any arguments past 6
  - The return address
- These are at the end of A's stack frame
- Once we jump to B's address, we begin B's stack frame…

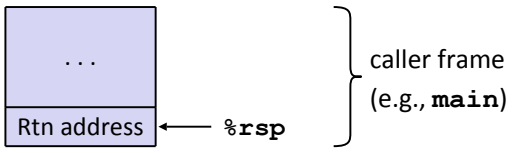

Older
Stack
Frames
Caller
Frame
Arguments
7+
Return Addr
Frame pointer
%rbp
(optional)
Saved
Registers
+
Local
Variables
Callee
Frame
Argument
Build
(optional)
Stack pointer
%rsp
Stack "Top"

Functions exercises

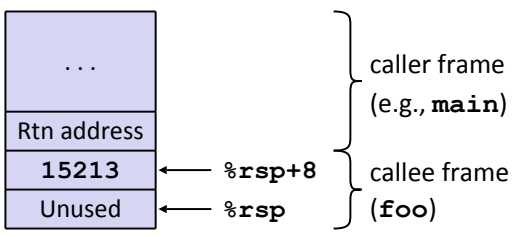
```
long foo() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

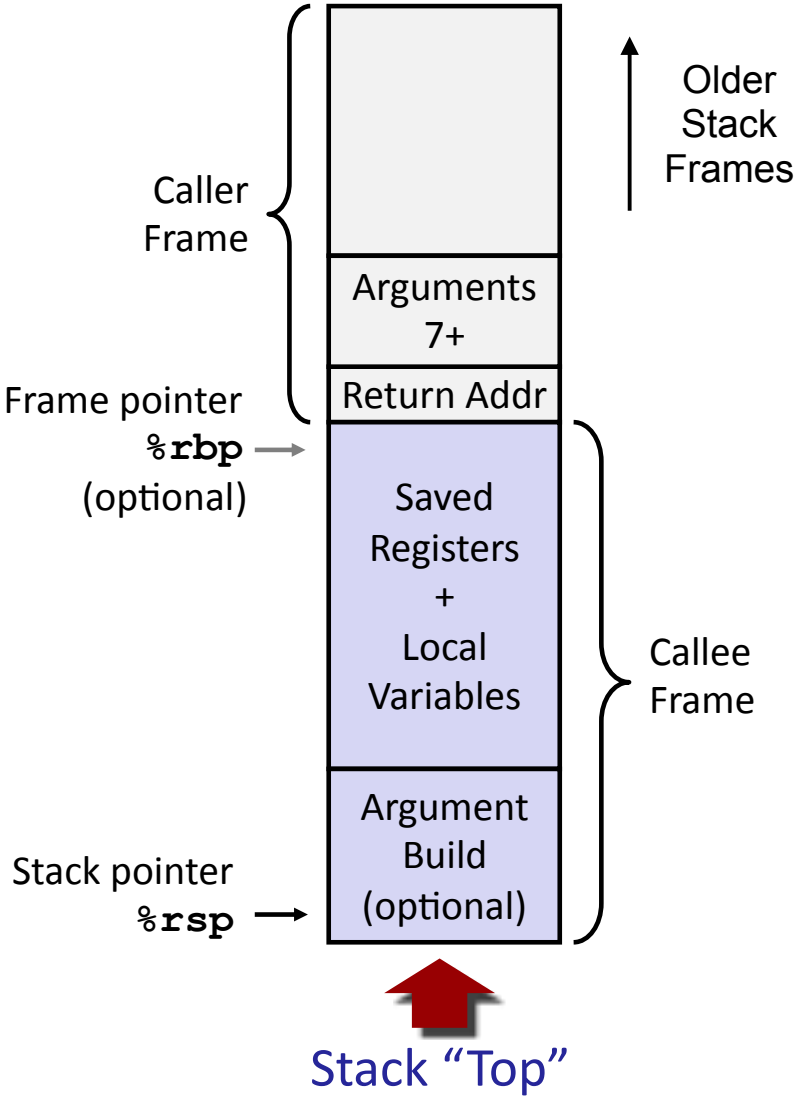
```
foo:
  subq    $16, %rsp
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

Initial Stack Structure



Resulting Stack Structure





TODO

- Assignment 3 assigned due October 28