CSCI-UA.0201

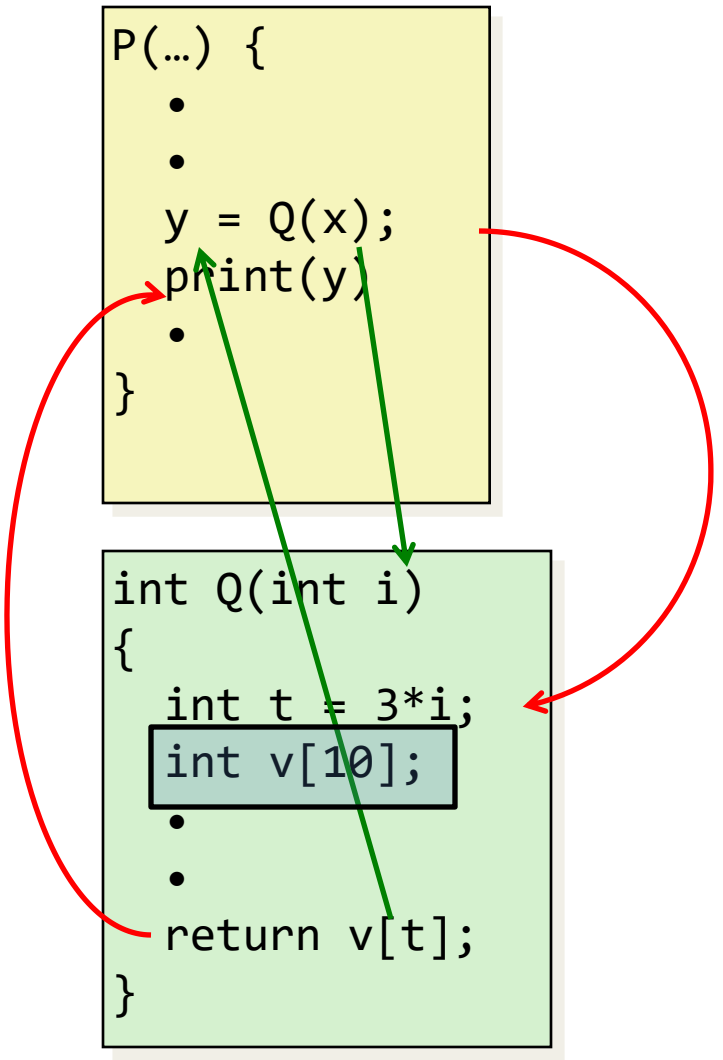
Computer Systems Organization

Machine Level – Function Calls

Thomas Wies
wies@cs.nyu.edu
https://cs.nyu.edu/wies

Suppose P calls Q

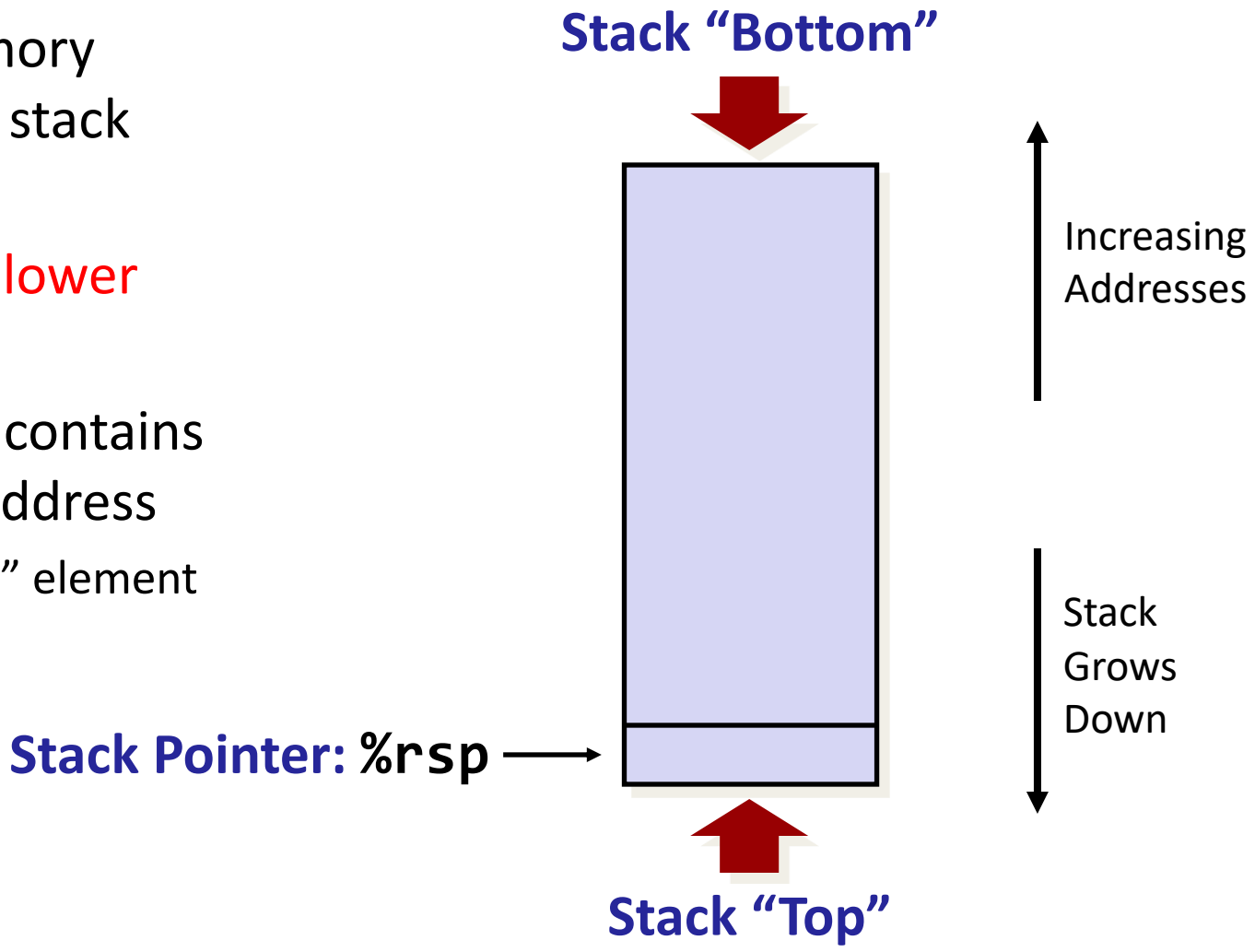
- **Passing control**
  - To beginning of procedure code
  - Back to return point
- **Passing data**
  - Procedure arguments
  - Return value
- **Memory management**
  - Allocate during procedure execution
  - Deallocate upon return



A quick glimpse at how stack works…

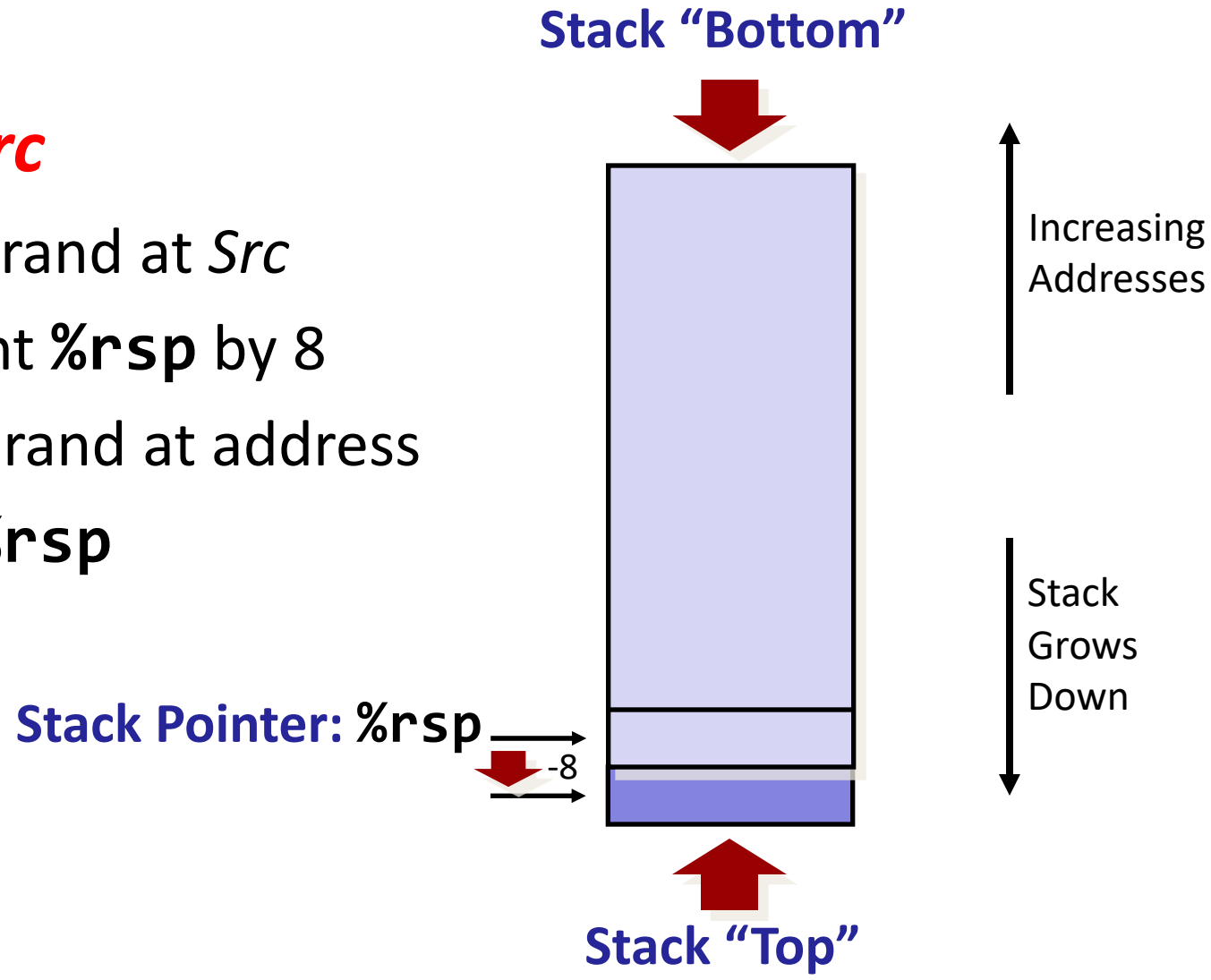
x86-64 Stack

- Region of memory managed with stack discipline
- Grows toward lower addresses
- Register **`%rsp`** contains lowest stack address
  - address of "top" element



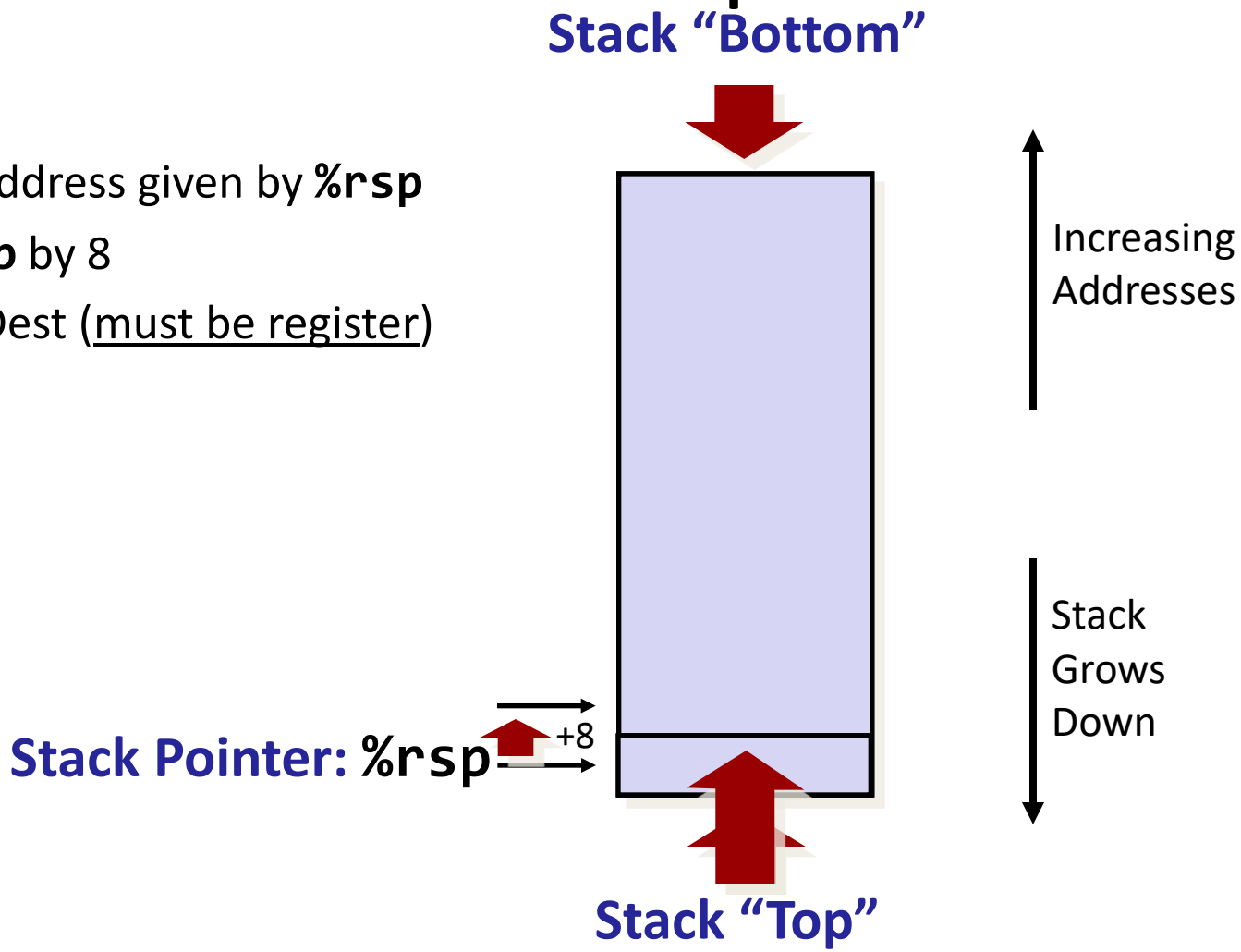
x86-64 Stack: Push

- **`pushq` *Src***
  - Fetch operand at *Src*
  - Decrement **`%rsp`** by 8
  - Write operand at address given by **`%rsp`**



x86-64 Stack: Pop

- **popq** ***Dest***
  - Read value at address given by **`%rsp`**
  - Increment **`%rsp`** by 8
  - Store value at Dest (must be register)



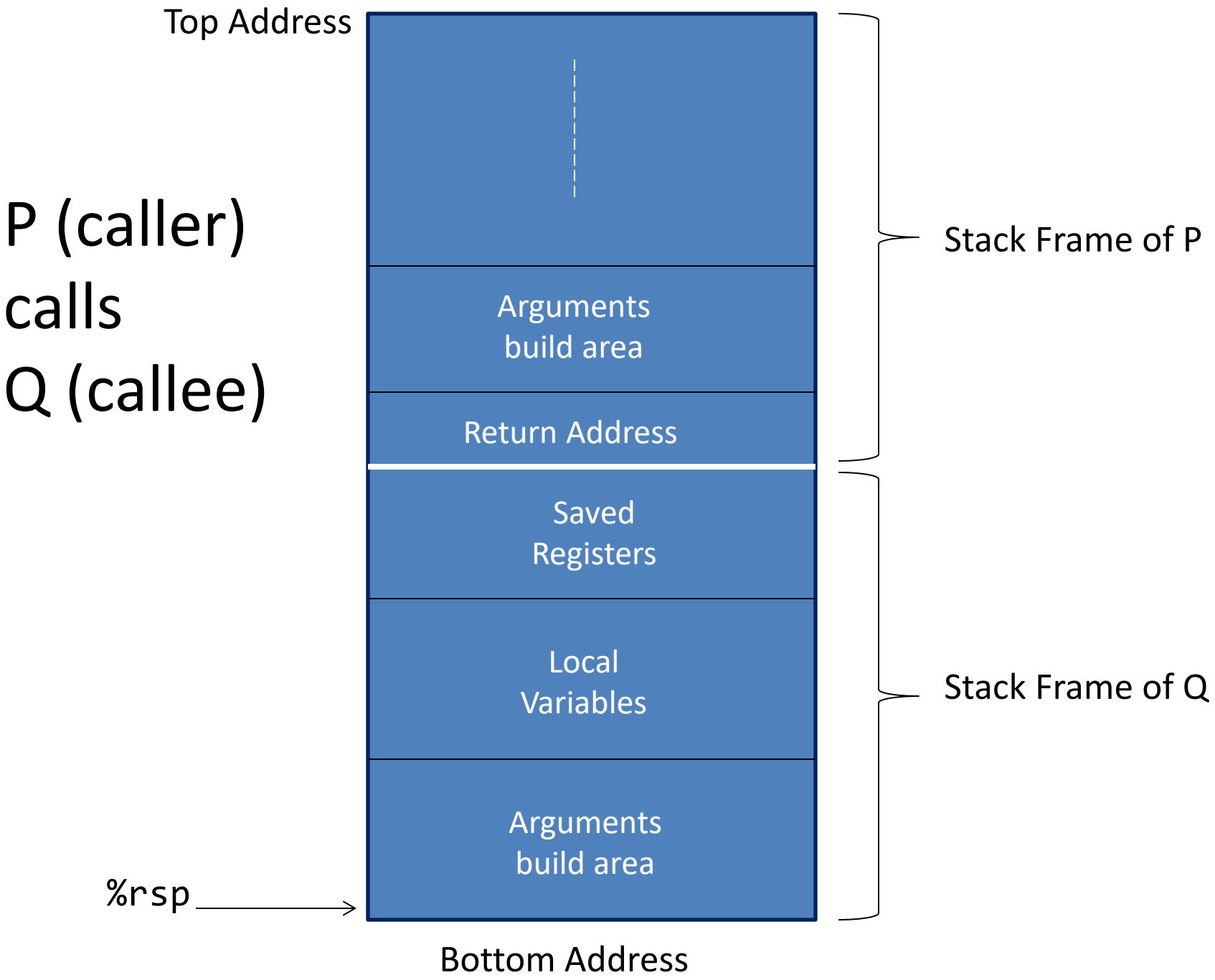
Examples:

```
void multstore
 (long x, long y, long *dest) {
    long t = mult2(x, y);
    *dest = t;
}
```

```
0000000000400540 <multstore>:
  400540: push   %rbx         # Save %rbx
  400541: movq   %rdx,%rbx            # Save dest
  400544: callq  400550 <mult2>       # mult2(x,y)
  400549: movq   %rax,(%rbx) # Save at dest
  40054c: popq   %rbx         # Restore %rbx
  40054d: retq                # Return
```

```
long mult2
  (long a, long b)
{
  long s = a * b;
  return s;
}
```

```
0000000000400550 <mult2>:
  400550:  movq   %rdi,%rax       # a
  400553:  imul   %rsi,%rax       # a * b
  400557:  retq             # Return
```


Top Address
P (caller)
calls
Q (callee)
Arguments
build area
Return Address
Saved
Registers
Local
Variables
Arguments
build area
%rsp
Stack Frame of P
Stack Frame of Q
Bottom Address

When P calls Q

- P is suspended and control moves to Q.
- A stack frame is setup on top of the stack for Q
- That stack frame contains:
  - saved registers
  - local variables
  - arguments if Q is calling another function
- Some procedures may not need a stack frame (why?).

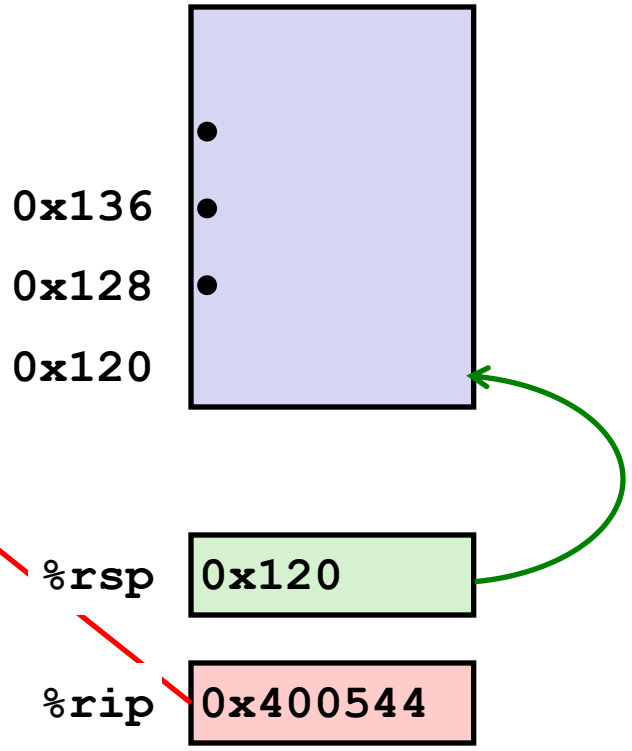
Procedure Control Flow

- Use stack to support procedure call and return
- Procedure call: **`call label`** **`[or call *op]`**
  - Push return address on stack
  - Jump to ***label***
- Return address:
  - Address of the next instruction right after call
- Procedure return: **`ret`**
  - Pop address from stack
  - Jump to address

Example

```
0000000000400540 <multstore>:
  •
  •
  400544: callq  400550 <mult2>
  400549: movq   %rax,(%rbx)
  •
  •
```

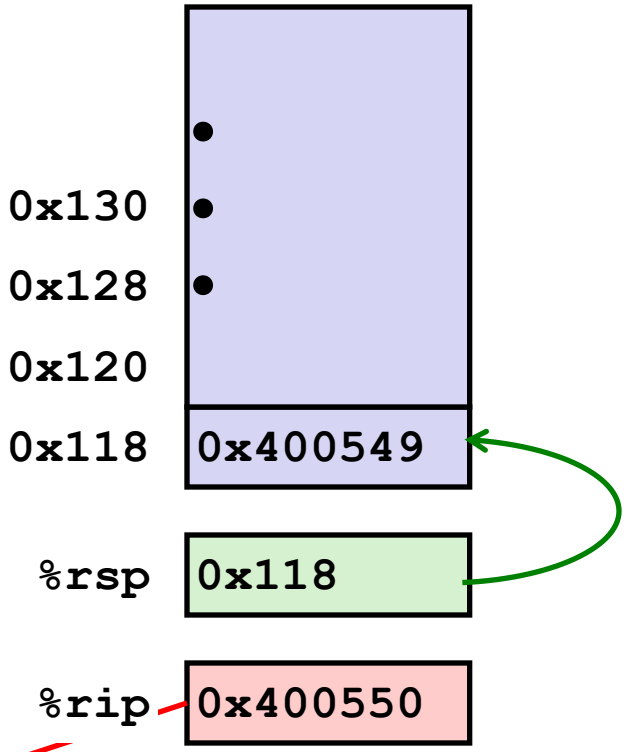
```
0000000000400550 <mult2>:
  400550:  movq   %rdi,%rax
  •
  •
  400557:  retq
```



Example

```
0000000000400540 <multstore>:
  •
  •
  400544: callq  400550 <mult2>
  400549: movq   %rax,(%rbx)
  •
  •
```

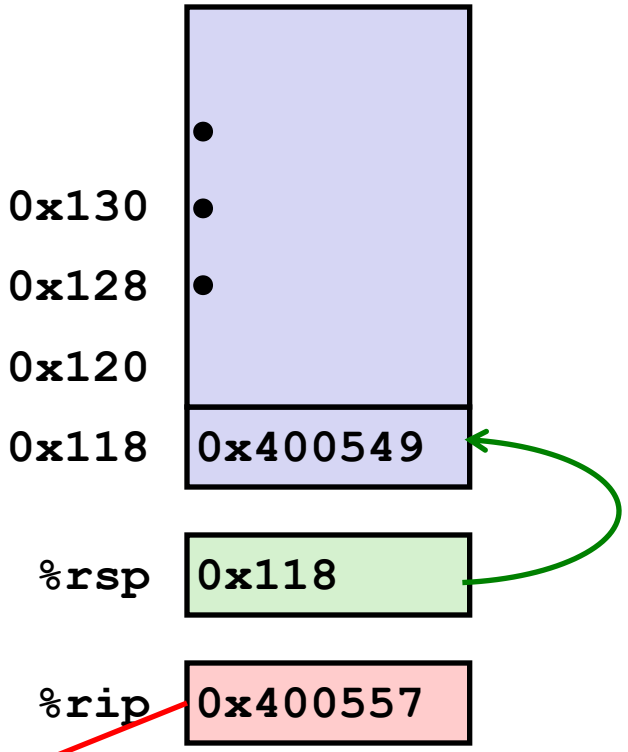
```
0000000000400550 <mult2>:
  400550:  movq   %rdi,%rax
  •
  •
  400557:  retq
```



Example

```
0000000000400540 <multstore>:
  •
  •
  400544: callq  400550 <mult2>
  400549: movq   %rax,(%rbx)
  •
  •
```

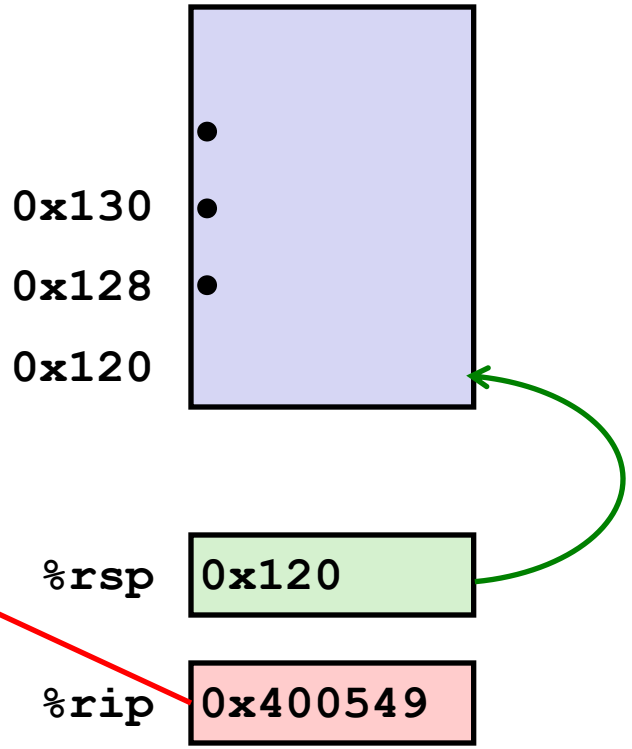
```
0000000000400550 <mult2>:
  400550:  movq   %rdi,%rax
  •
  •
  400557:  retq
```



Example

```
0000000000400540 <multstore>:
  •
  •
  400544: callq  400550 <mult2>
  400549: movq   %rax,(%rbx)
  •
  •
```

```
0000000000400550 <mult2>:
  400550:  movq   %rdi,%rax
  •
  •
  400557:  retq
```



Calling Conventions

Answers questions such as:

- Which arguments are passed in which registers?
- Which register holds the return value?
- Where can auxiliary arguments that don't fit into registers be found on the stack?
- Who is responsible for restoring which registers?
  - caller vs. callee-saved registers

Calling conventions are part of the **Application binary interface (ABI)**, which are typically OS-specific.

We focus on the ABI for x86-64 adhered to by most Unix-like operating systems.

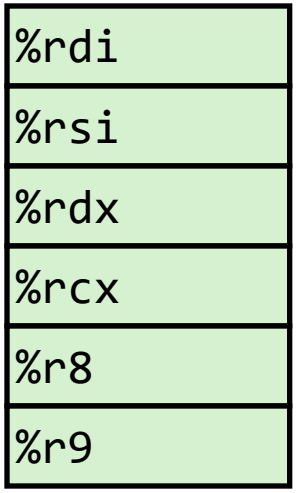
Calling Conventions

Register	Usage	Preserved across function calls
`%rax`	temporary register; with variable arguments passes information about the number of vector registers used; 1st return register	No
`%rbx`	callee-saved register	Yes
`%rcx`	used to pass 4th integer argument to functions	No
`%rdx`	used to pass 3rd argument to functions; 2nd return register	No
`%rsp`	stack pointer	Yes
`%rbp`	callee-saved register; optionally used as frame pointer	Yes
`%rsi`	used to pass 2nd argument to functions	No
`%rdi`	used to pass 1st argument to functions	No
`%r8`	used to pass 5th argument to functions	No
`%r9`	used to pass 6th argument to functions	No
`%r10`	temporary register, used for passing a function's static chain pointer	No
`%r11`	temporary register	No
`%r12-r14`	callee-saved registers	Yes
`%r15`	callee-saved register; optionally used as GOT base pointer	Yes

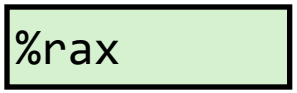
Procedure Data Flow

Registers

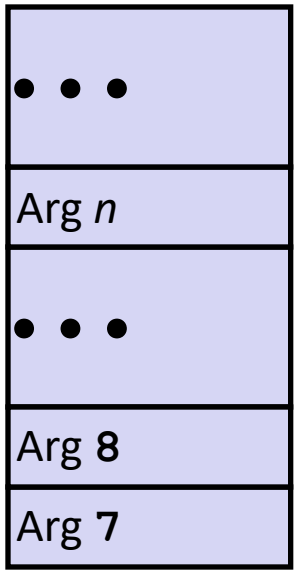
- First 6 arguments



- Return value



Stack



- Only allocate stack space when needed
- When passing parameters on the stack, all data sizes are rounded up to be multiple of eight.

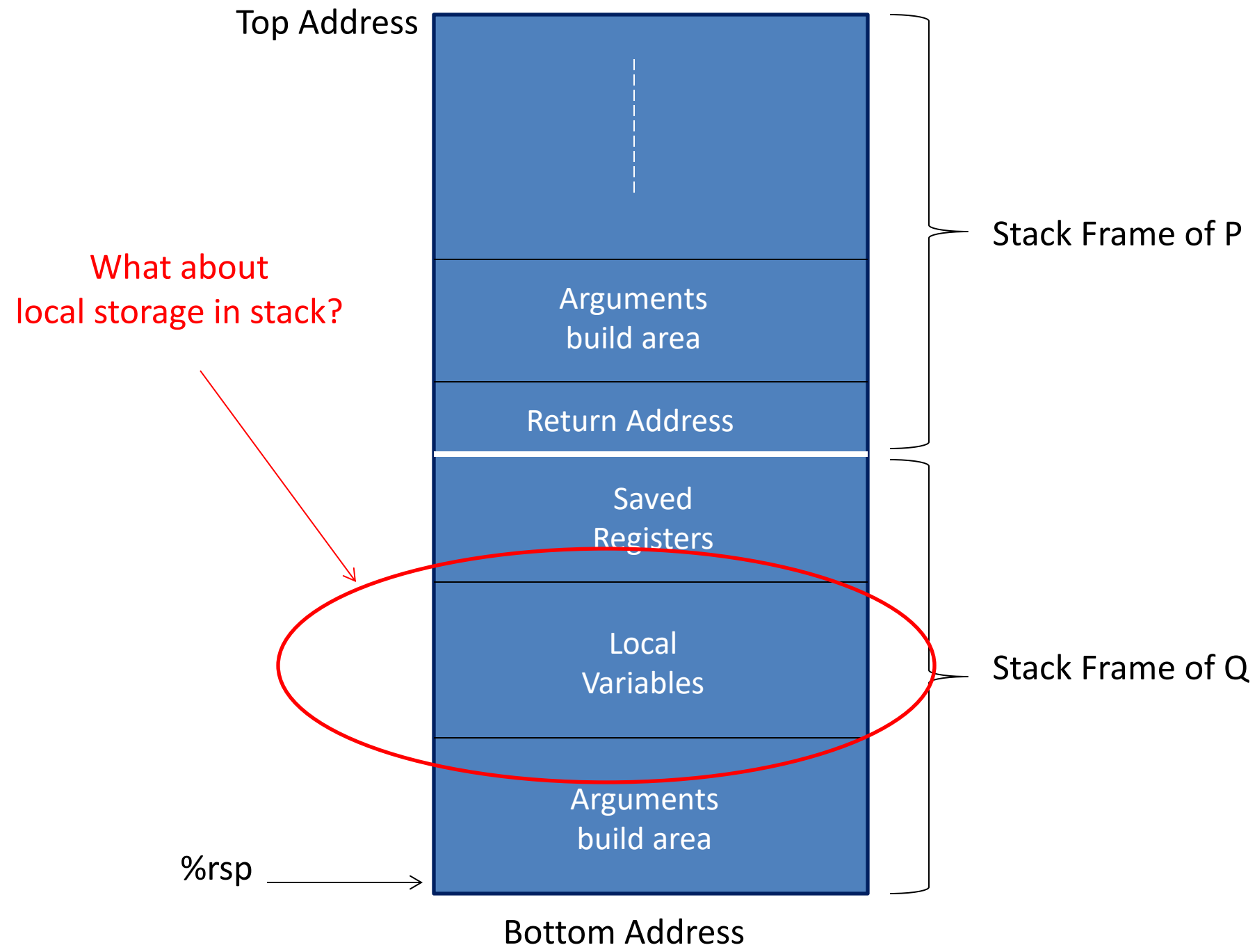
Example:
multstore calls mult2

```
void multstore
 (long x, long y, long *dest)
{
    long t = mult2(x, y);
    *dest = t;
}
```

```
0000000000400540 <multstore>:
  # x in %rdi, y in %rsi, dest in %rdx
  • • •
  400541: movq   %rdx,%rbx          # Save dest
  400544: callq  400550 <mult2>     # t = mult2(x,y)
  # t in %rax
  400549: mov    %rax,(%rbx) # Save at dest
  • • •
```

```
long mult2
  (long a, long b)
{
  long s = a * b;
  return s;
}
```

```
0000000000400550 <mult2>:
  # a in %rdi, b in %rsi
  400550:  movq   %rdi,%rax  # s = a
  400553:  imul   %rsi,%rax  # s = s * b
  # s in %rax
  400557:  retq              # return s
```


Top Address
What about
local storage in stack?
Arguments
build area
Return Address
Saved
Registers
Local
Variables
Arguments
build area
%rsp
Stack Frame of P
Stack Frame of Q
Bottom Address

When is local storage needed?

- Not enough registers
- A variable in high-level language is passed by reference (“&” in C) so it needs to have an address!
- Arrays, structures, …

Example: `incr`

```
long incr(long *p, long val) {
    long x = *p;
    long y = x + val;
    *p = y;
    return x;
}
```

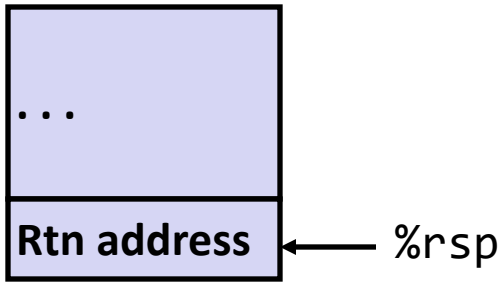
```
incr:
  movq    (%rdi), %rax
  addq    %rax, %rsi
  movq    %rsi, (%rdi)
  ret
```

Register	Use(s)
%rdi	Argument **p**
%rsi	Argument **val**, **y**
%rax	**x**, Return value

Example: Calling `incr`

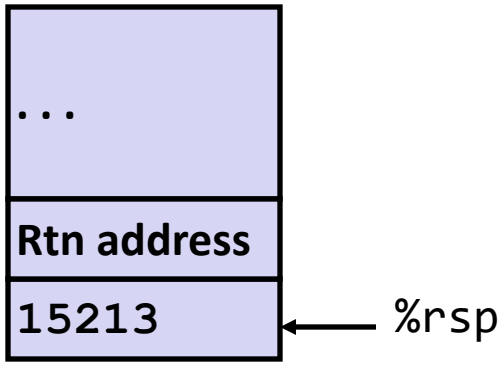
```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

Initial Stack Structure




```
call_incr:
  subq    $8, %rsp
  movq    $15213,(%rsp)
  movq    $3000, %rsi
  leaq    (%rsp), %rdi
  call    incr
  addq    (%rsp), %rax
  addq    $8, %rsp
  ret
```

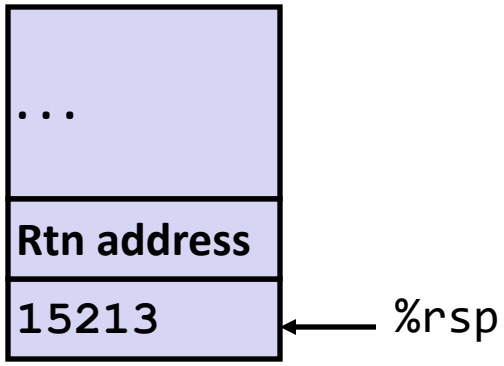
Resulting Stack Structure



Example: Calling `incr`

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

Stack Structure



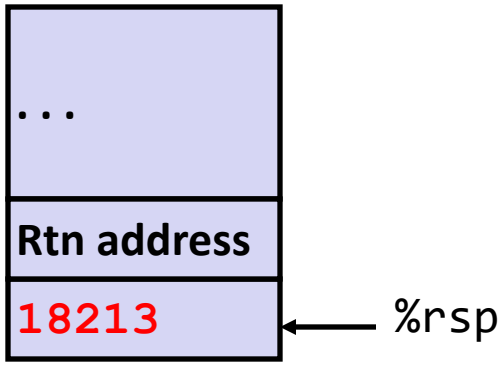

```
call_incr:
  subq    $8, %rsp
  movq    $15213,(%rsp)
  movq    $3000, %rsi
  leaq    (%rsp), %rdi
  call    incr
  addq    (%rsp), %rax
  addq    $8, %rsp
  ret
```

Register	Use(s)
%rdi	&v1
%rsi	3000

Example: Calling `incr`

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

Stack Structure



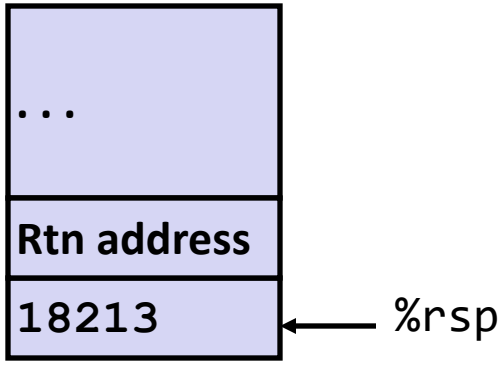

```
call_incr:
  subq    $8, %rsp
  movq    $15213,(%rsp)
  movq    $3000, %rsi
  leaq    (%rsp), %rdi
  call    incr
  addq    (%rsp), %rax
  addq    $8, %rsp
  ret
```

Register	Use(s)
%rdi	&v1
%rsi	3000

Example: Calling `incr`

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

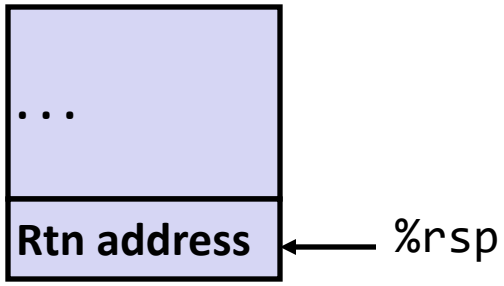
Stack Structure




```
call_incr:
  subq    $8, %rsp
  movq    $15213,(%rsp)
  movq    $3000, %rsi
  leaq    (%rsp), %rdi
  call    incr
  addq    (%rsp), %rax
  addq    $8, %rsp
  ret
```

Register	Use(s)
%rax	Return value

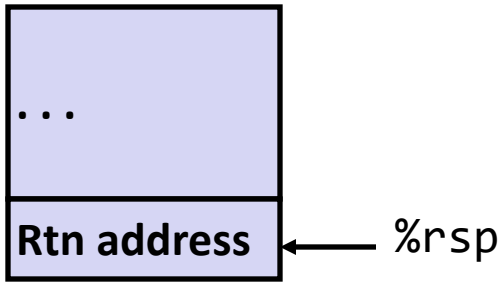
Updated Stack Structure



Example: Calling `incr`

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

Updated Stack Structure




```
call_incr:
  subq    $8, %rsp
  movq    $15213,(%rsp)
  movq    $3000, %rsi
  leaq    (%rsp), %rdi
  call    incr
  addq    (%rsp), %rax
  addq    $8, %rsp
  ret
```

Register	Use(s)
%rax	Return value

Final Stack Structure