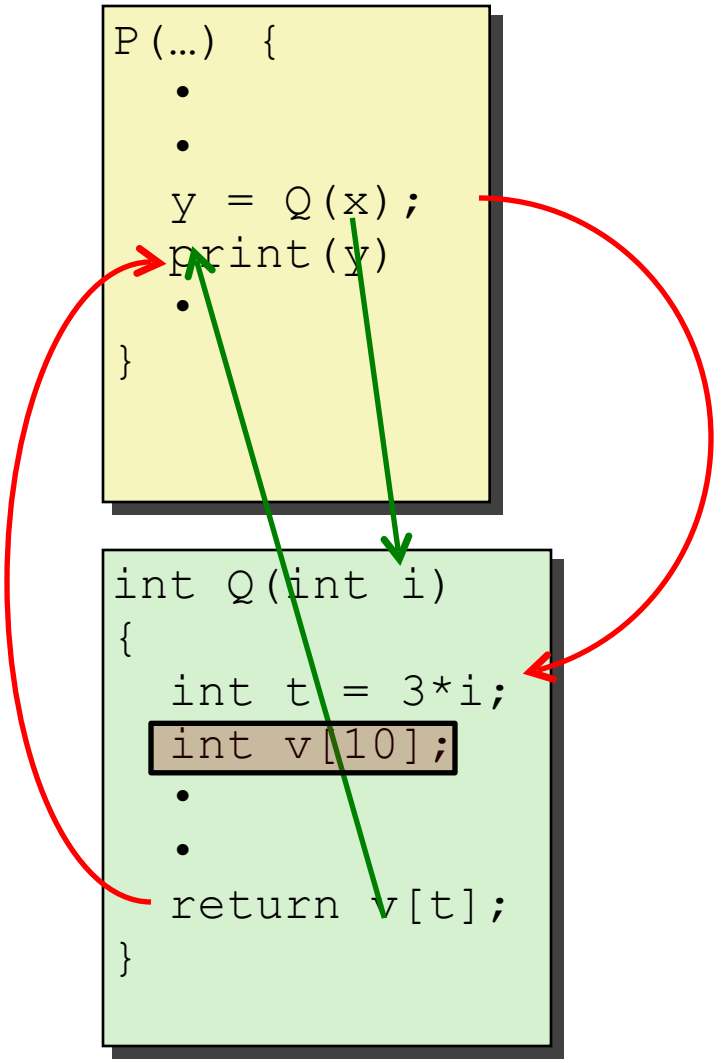
+

Procedures

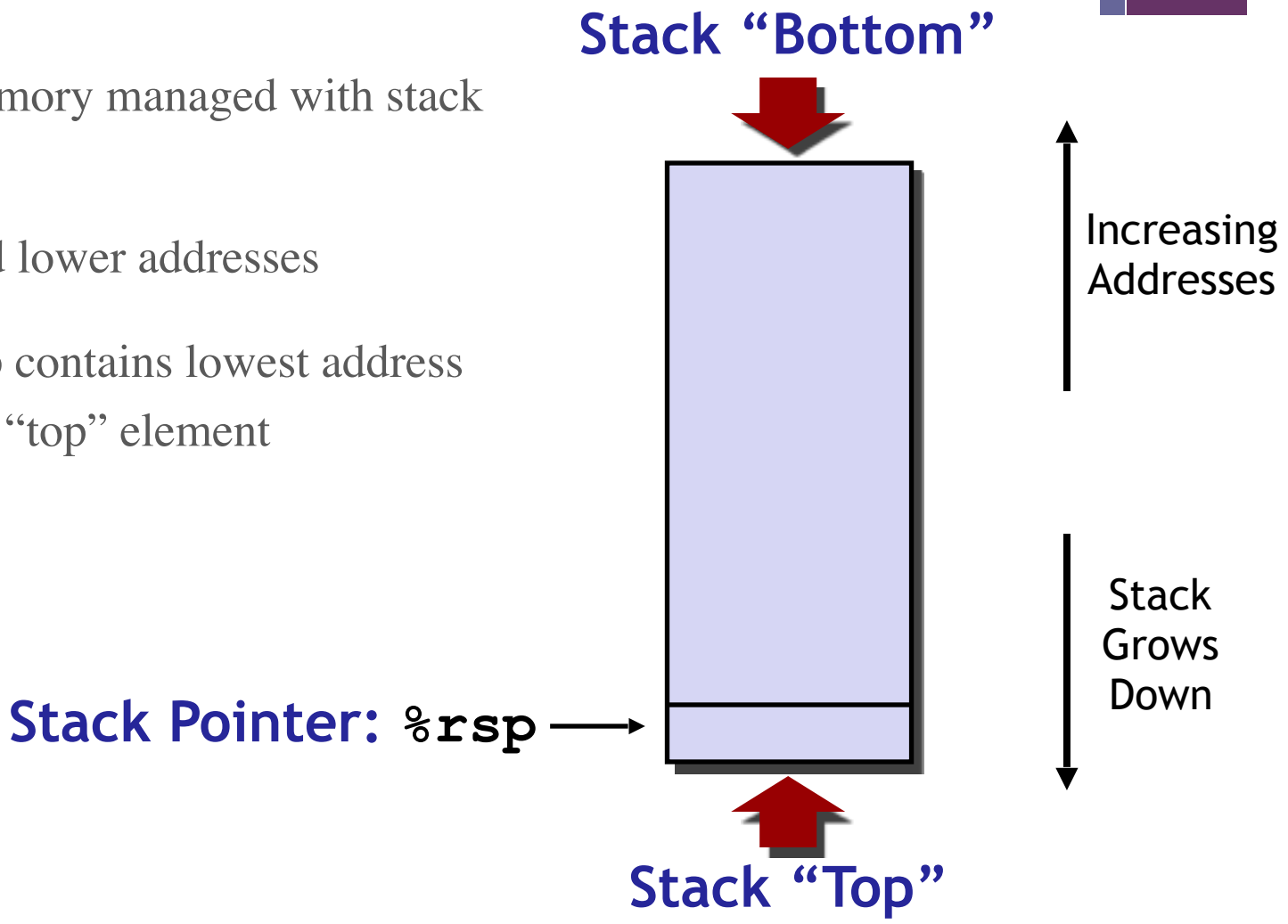
Mechanisms in Procedures

- **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
- **Mechanisms all implemented with machine instructions**
- **x86-64 implementation of a procedure uses only those mechanisms required**



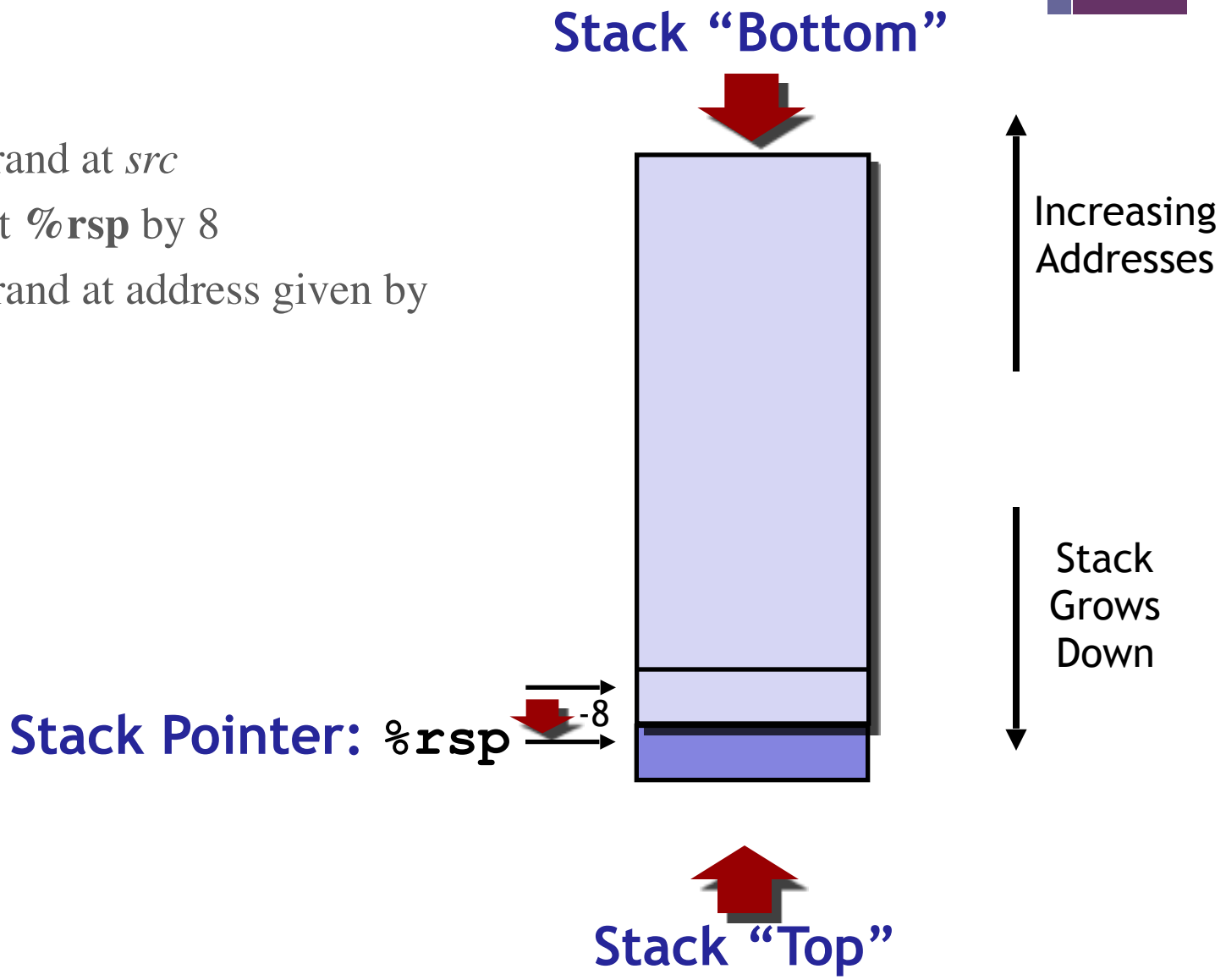
x86-64 Stack

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



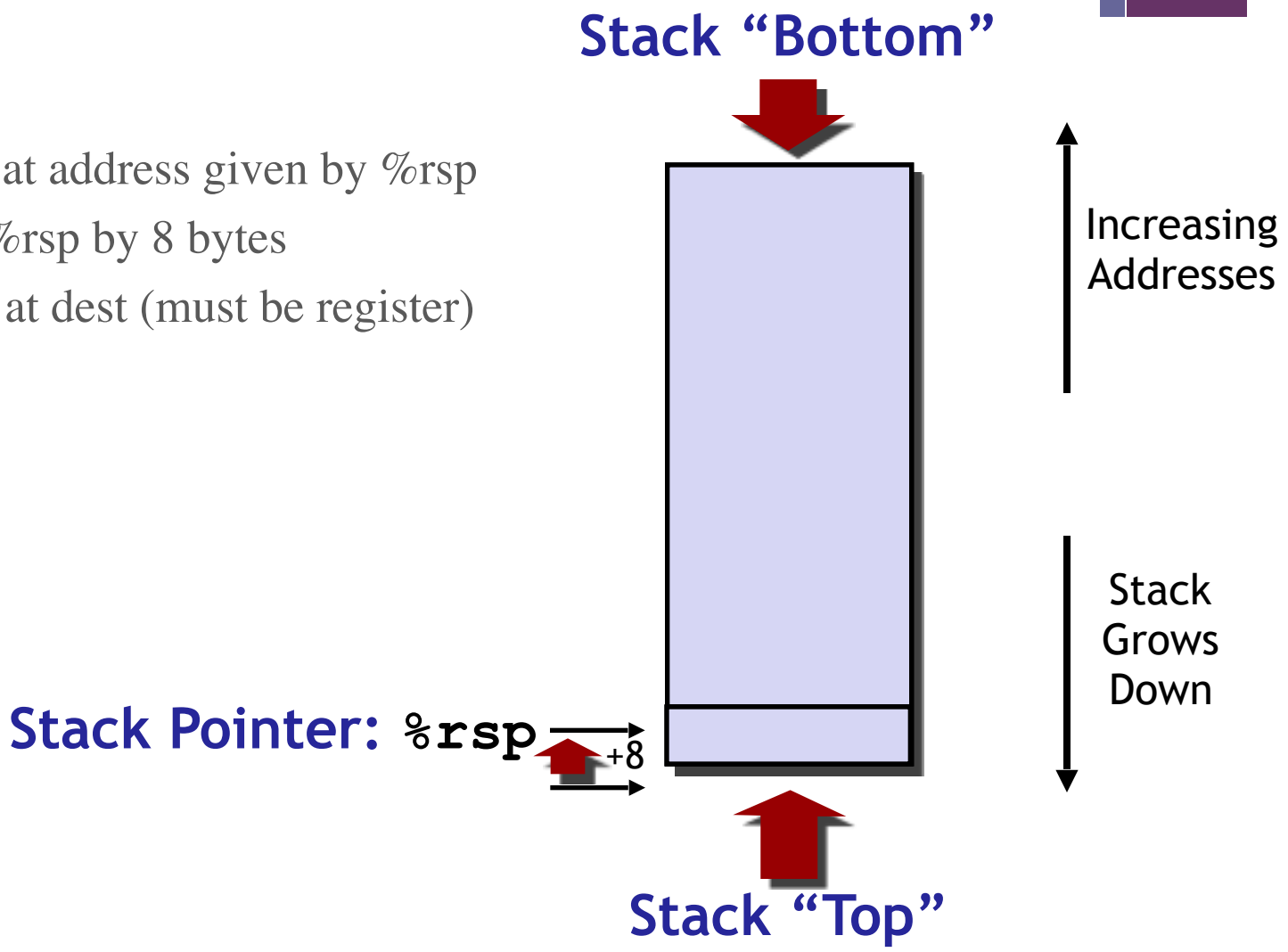
x86-64 Stack: Push

- **`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 bytes
  - Store value at dest (must be register)



+

Passing Control

Code Examples

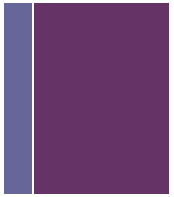
```
void multstore (long x, long y, long *dest){
    long t = mult2(x, y);
    *dest = t;
}
```

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

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

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

Procedure Control Flow

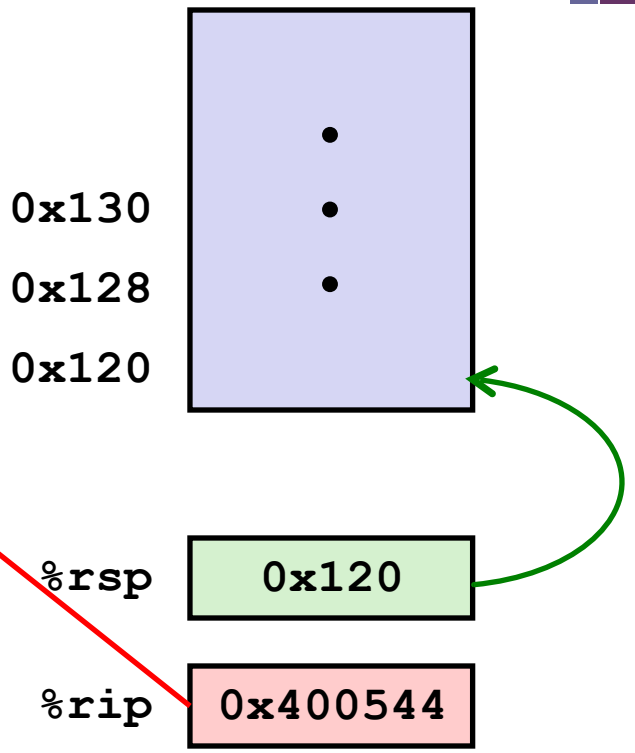


- Use stack to support procedure call and return
- **Procedure `call` with label**
  - Pushes ***return address*** on stack
    - Address of the next instruction right after call
  - Jumps to ***label***
- **Procedure return: `ret`**
  - Pops ***return address*** from stack
  - Jumps to ***return address***

Control Flow Example #1

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

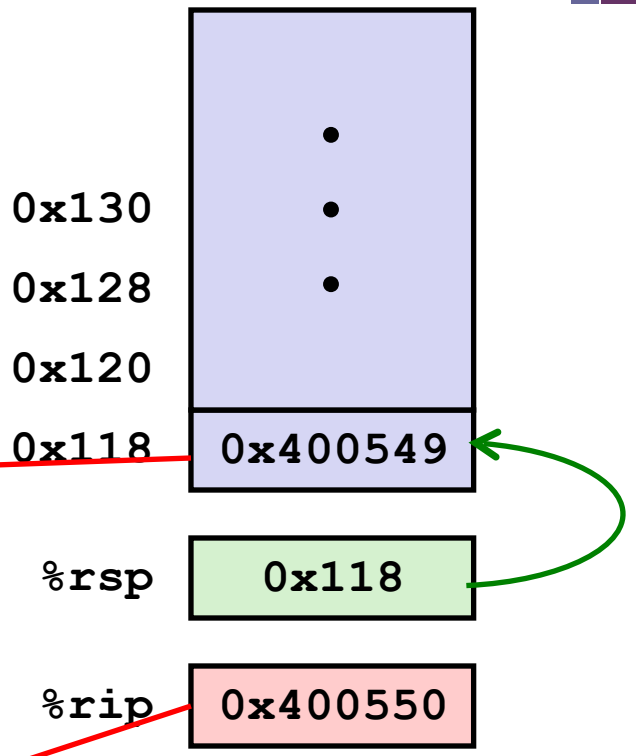
```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  ret
```



Control Flow Example #2

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

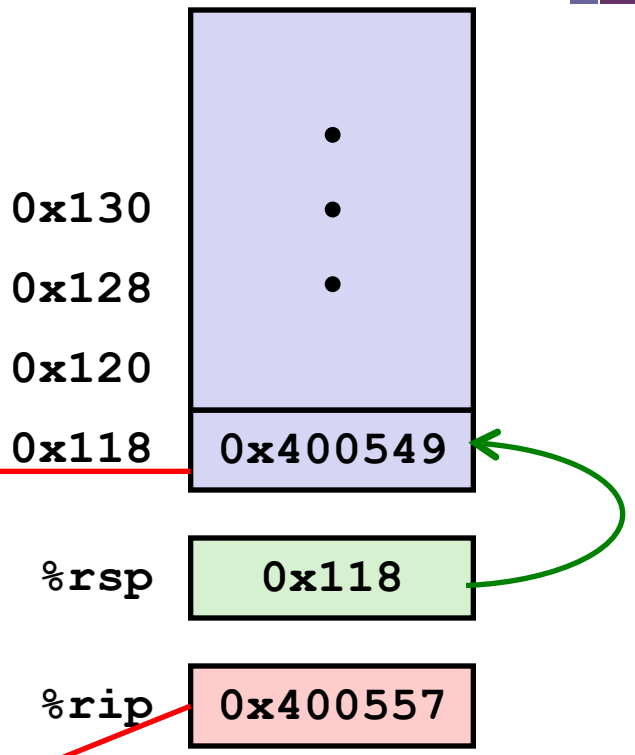
```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  ret
```



Control Flow Example #3

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

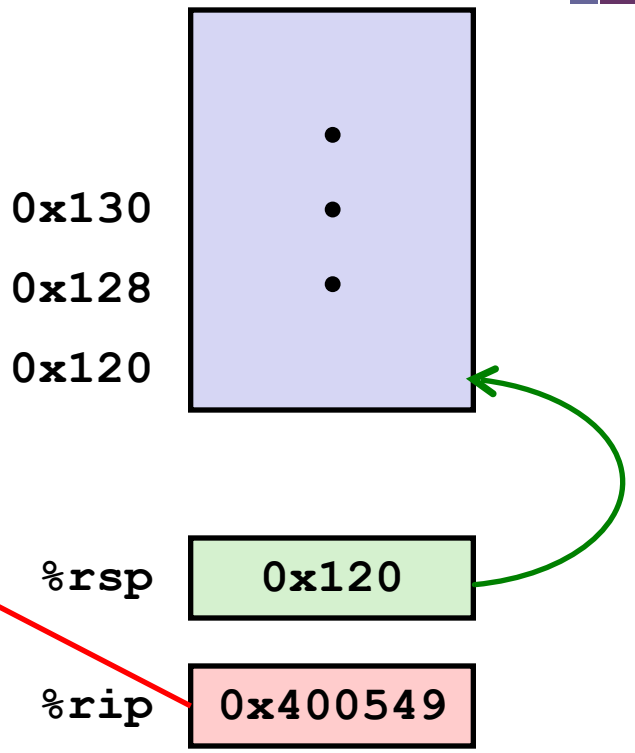
```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  ret
```



Control Flow Example #4

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

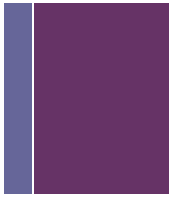
```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  ret
```



+

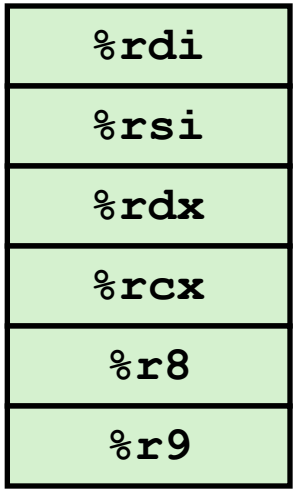
Passing Data

Procedure Data Flow

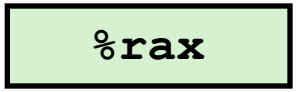


Registers

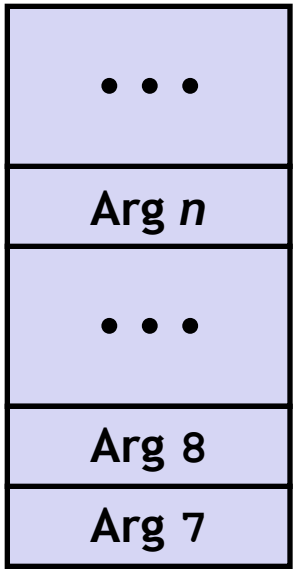
First 6 arguments



Return value



Stack



- **Only allocate stack space when needed**

Data Flow Examples

```
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: mov    %rdx,%rbx          # Save dest
  400544: callq  400550 <mult2>     # mult2(x,y)
  # t in %rax
  400549: mov    %rax,(%rbx)        # Save at dest
  • • •
```

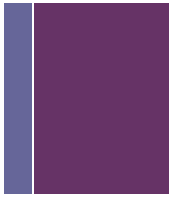
```
long mult2(long a, long b){
  long s = a * b;
  return s;
}
```

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

+

Managing Local Data

Stack Frames



- **Functions have "instantiations"**
  - Every function call is a distinct execution with distinct data.
  - Need some place to store state of each instantiation
    - Arguments
    - Local variables
    - Return pointer (next instruction in caller)
- **Stack allocated in *frames***
  - State for single procedure instantiation
  - Moreover, an allocation of memory holding all the data for some function call.
- **Recursion**
  - Supported by this idea of *instantiation* and *stack discipline*.

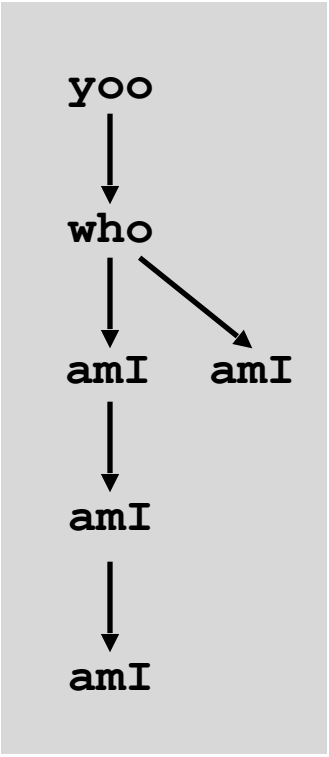
Call Chain Example

```
yoo(…)
{
   •
   •
   who();
   •
   •
}
```

```
who(…)
{
   • • •
   amI();
   • • •
   amI();
   • • •
}
```

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

Example Call Chain

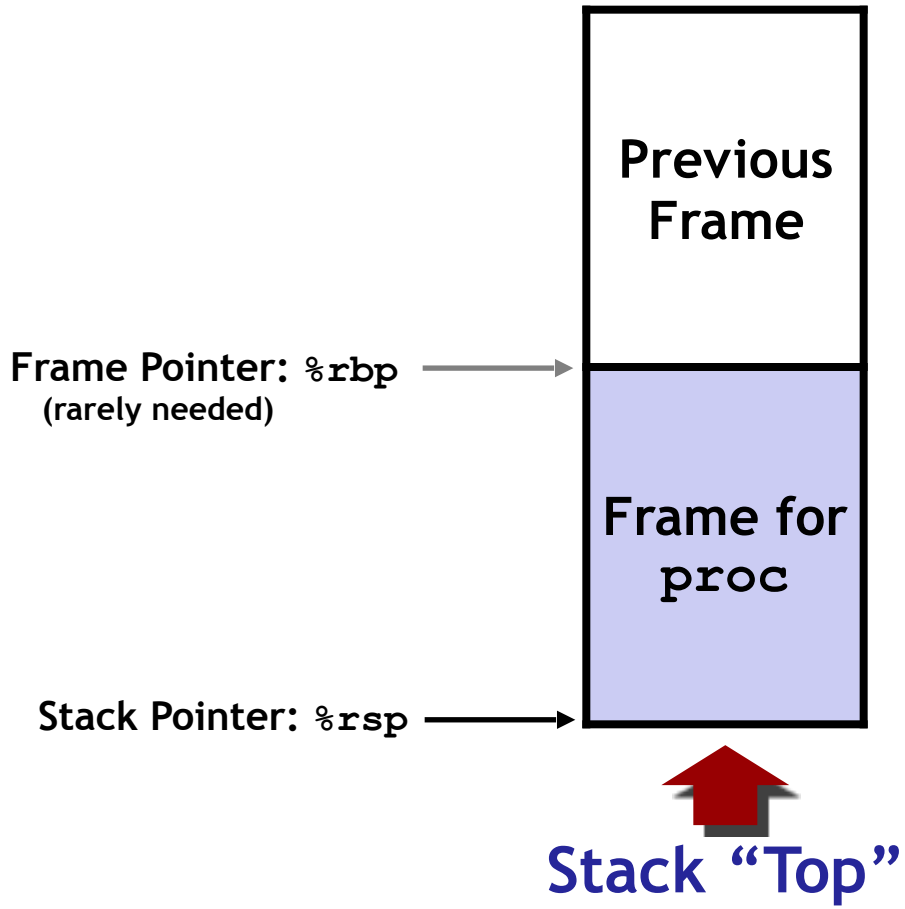


Procedure `amI()` is recursive

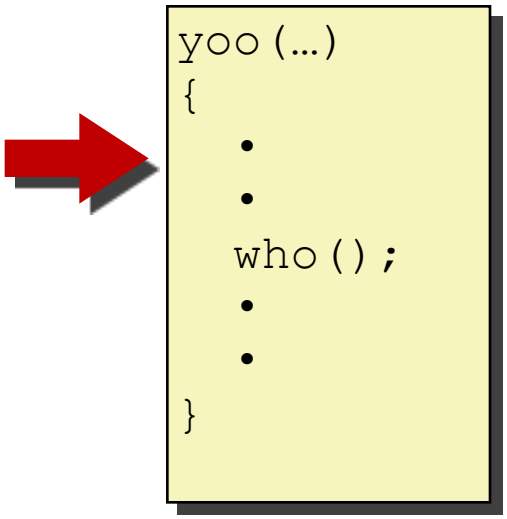
Stack Frames

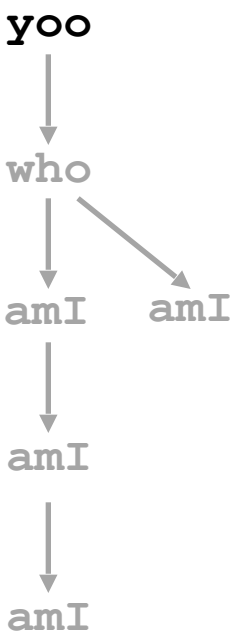
- **Contents**
  - Return information
  - Local storage (if needed)

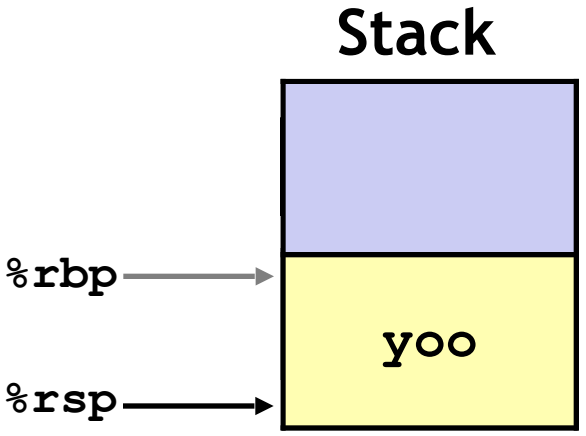
- **Management**
  - Space allocated on procedure call
    - push by **`call`** instruction
  - Space deallocated on return
    - pop by **`ret`** instruction



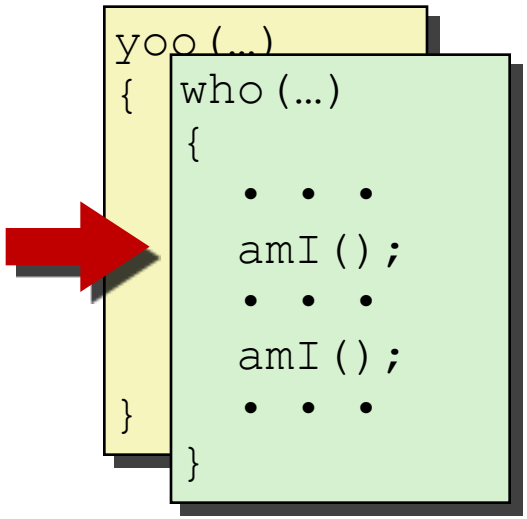
Example

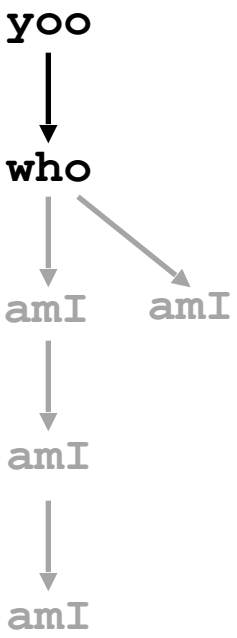


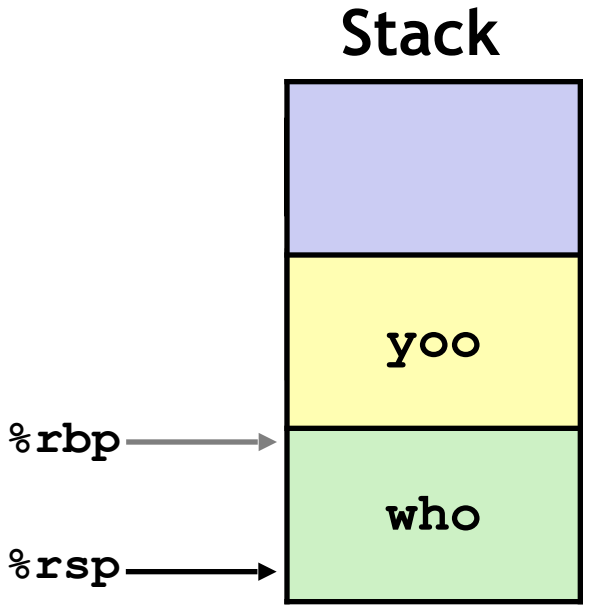




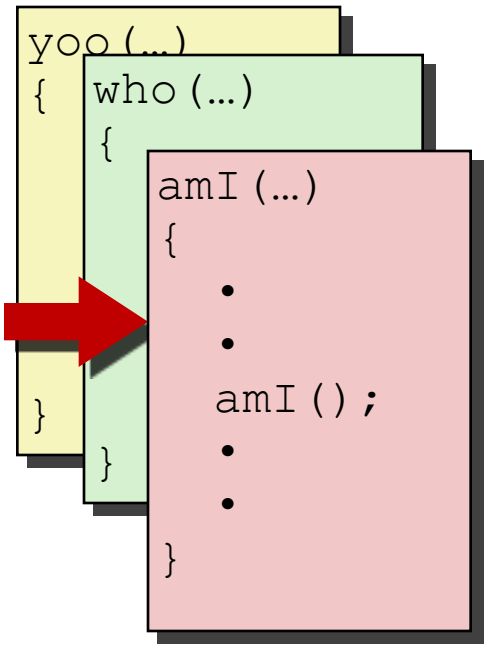
Example

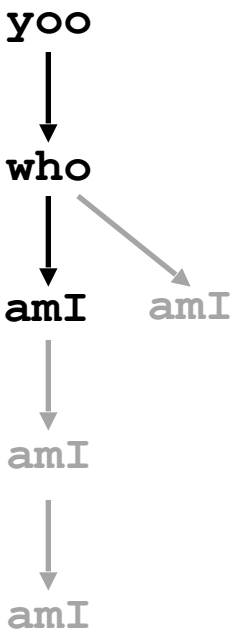


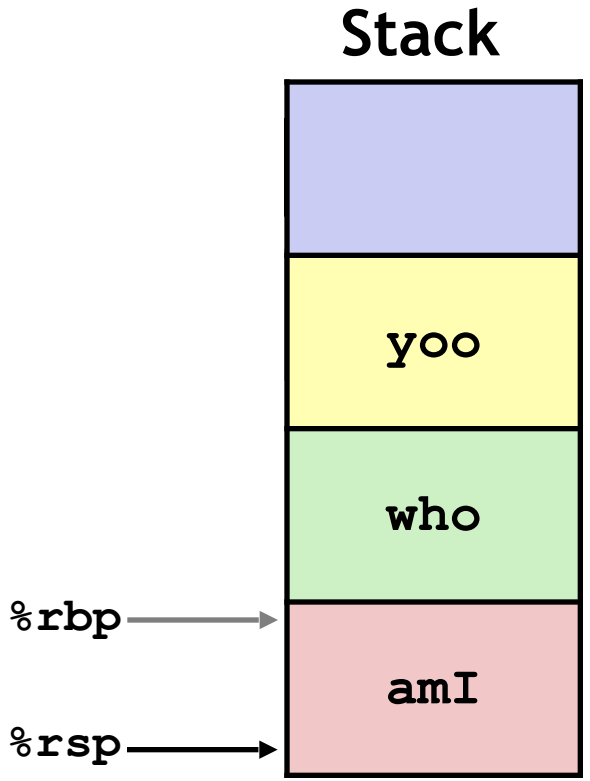




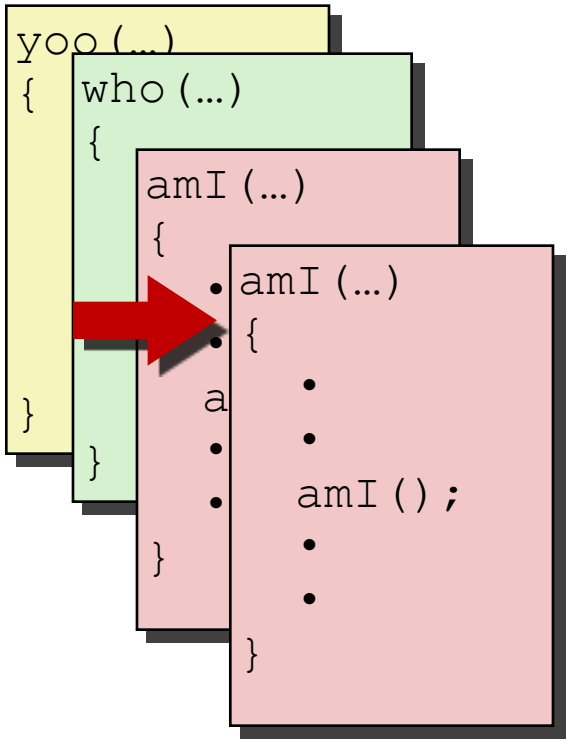
Example

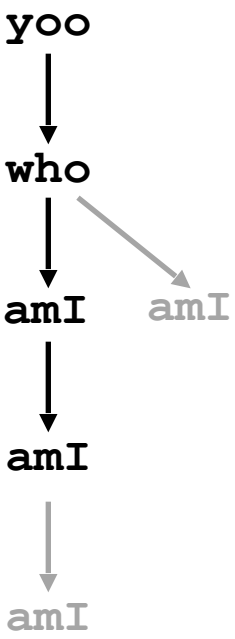

yoo(…)
{
who(…)
{
amI(…)
{
•
•
amI();
•
•
}
}
}

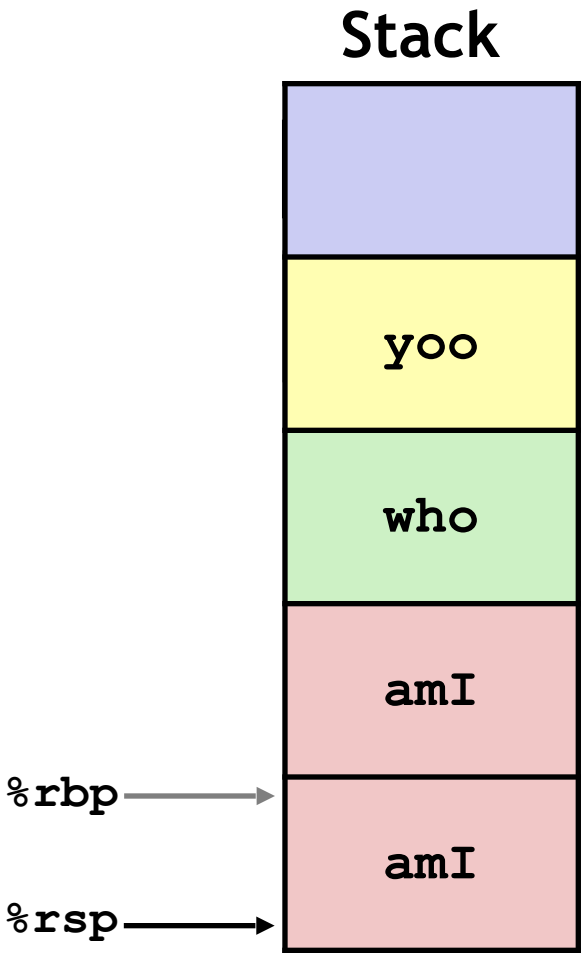

yoo
who
amI
amI
amI
amI


Stack
yoo
who
amI
%rbp
%rsp

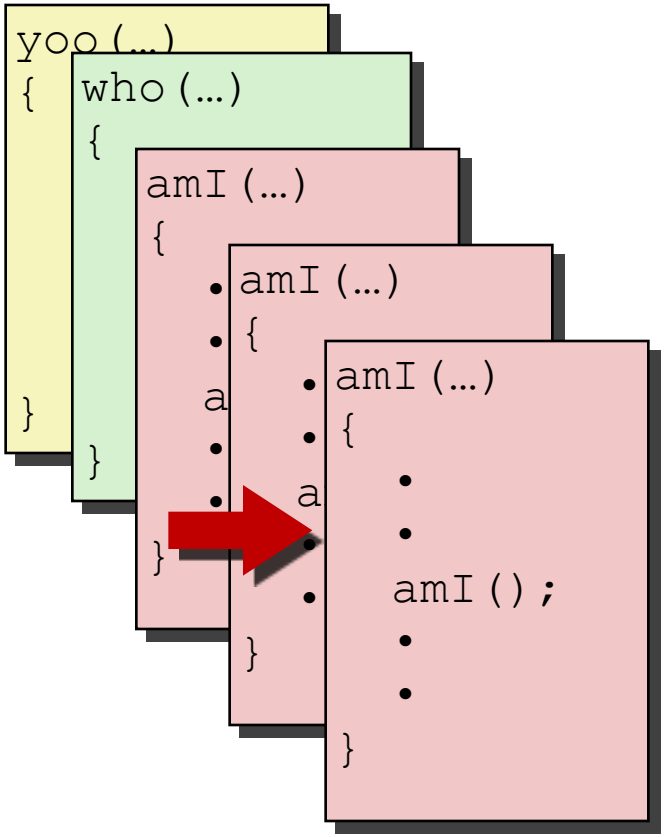
Example

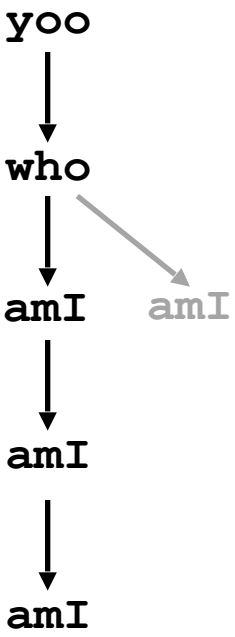

yoo(…)
{
who(…)
{
amI(…)
{
amI(…)
{
a
amI();
}
}
}

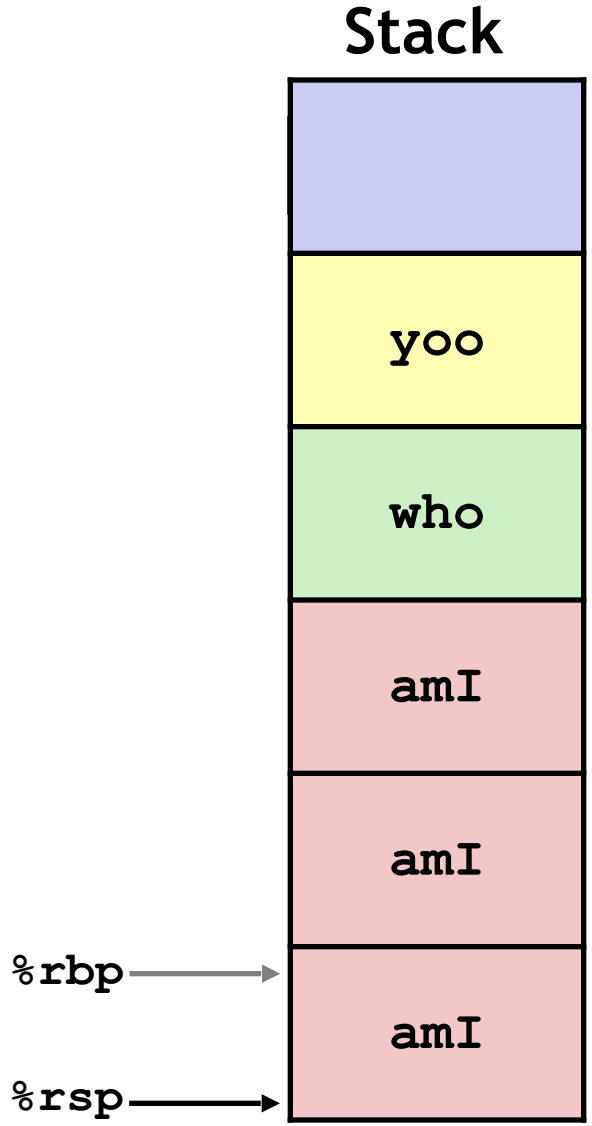

yoo
who
amI
amI
amI
amI


Stack
yoo
who
amI
amI
%rbp
%rsp

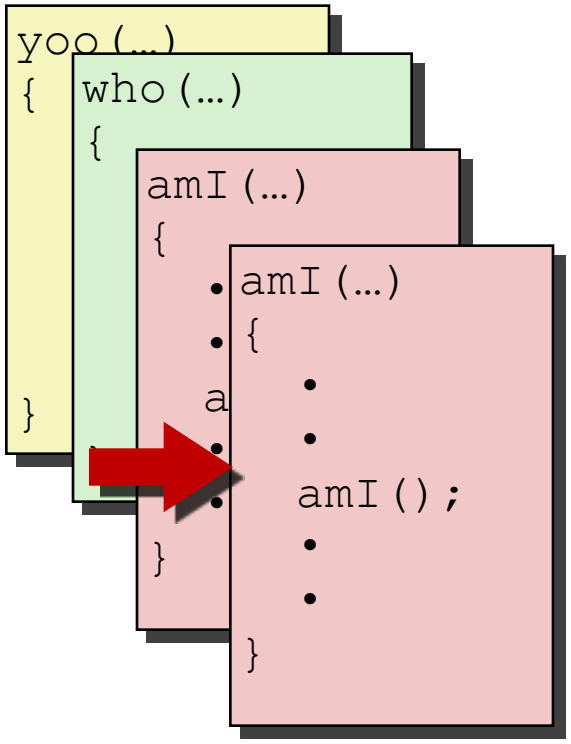
Example

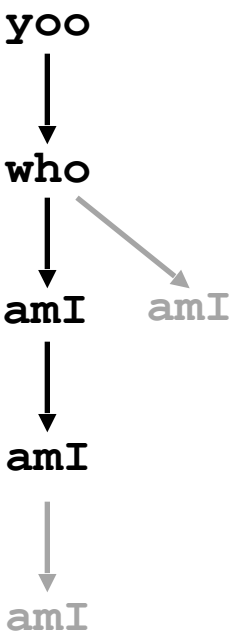

yoo (…)
{
who (…)
{
amI (…)
{
amI (…)
{
amI (…)
{
amI();
}

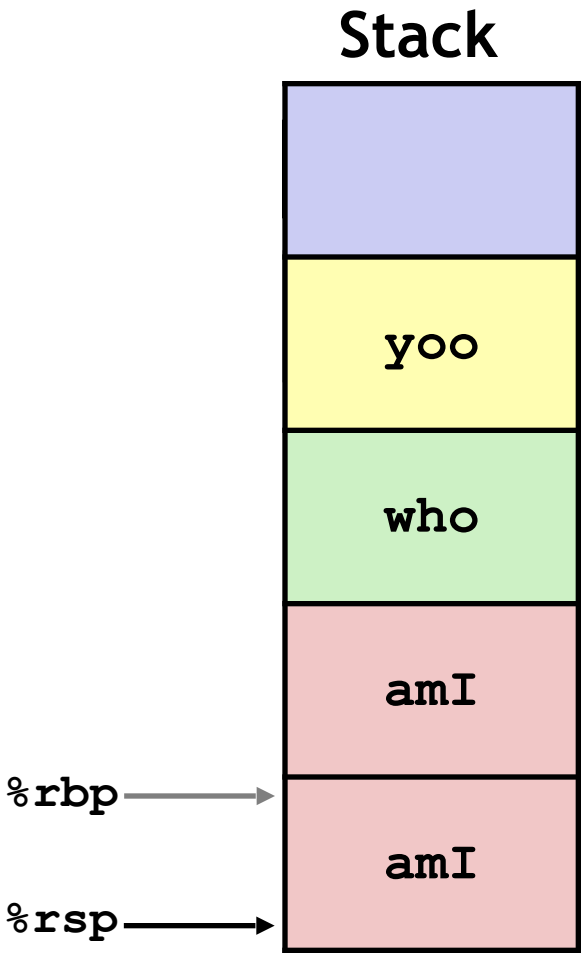

yoo
who
amI
amI
amI
amI


Stack
yoo
who
amI
amI
amI
%rbp
%rsp

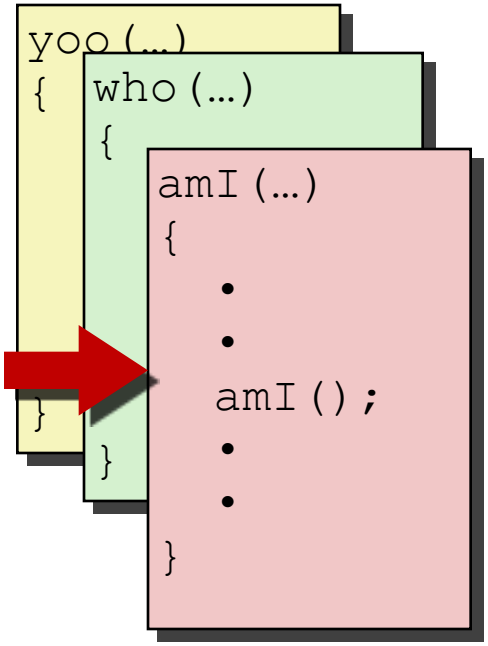
Example

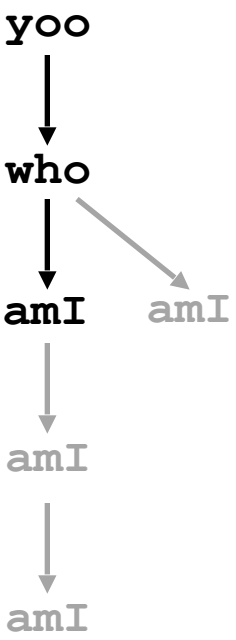

yoo(…)
{
who(…)
{
amI(…)
{
amI(…)
{
amI();
}

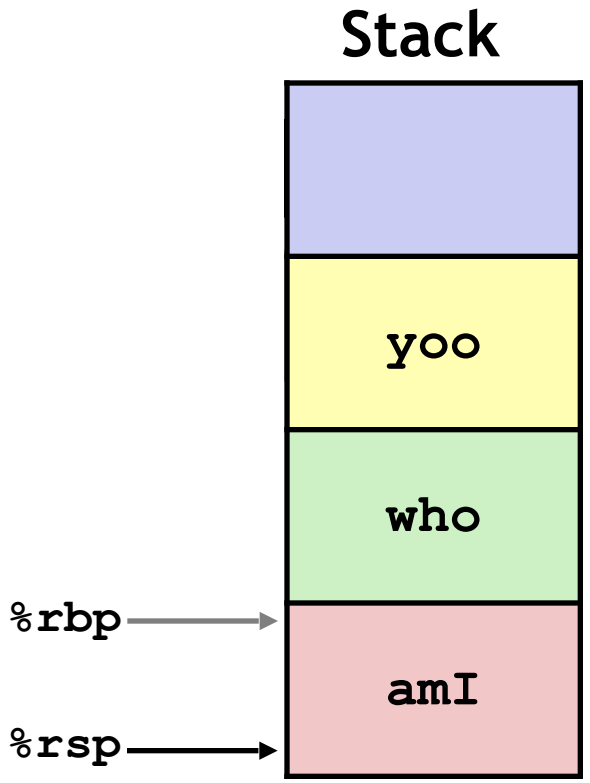

yoo
who
amI
amI
amI
amI


Stack
yoo
who
amI
amI
%rbp
%rsp

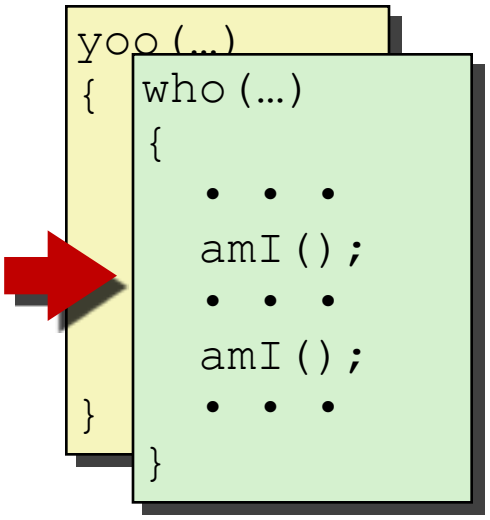
Example

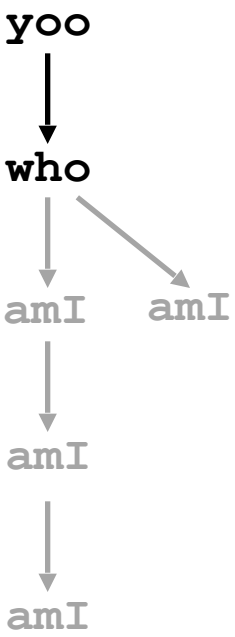

yoo(…)
{
who(…)
{
amI(…)
{
•
•
amI();
•
•
}
}
}

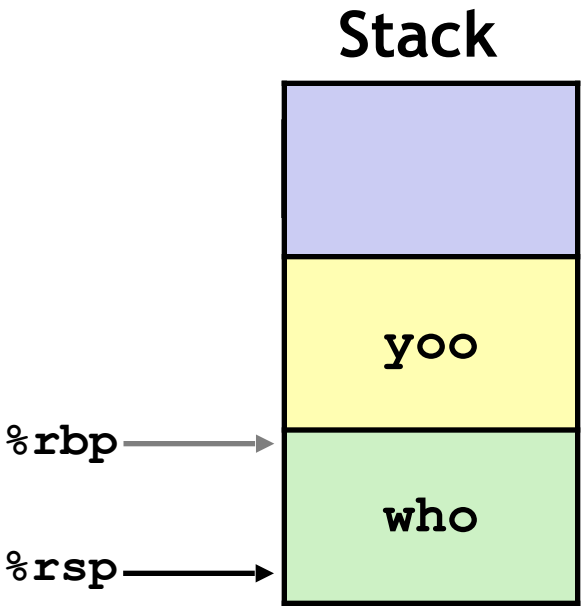

yoo
who
amI
amI
amI
amI


Stack
yoo
who
amI
%rbp
%rsp

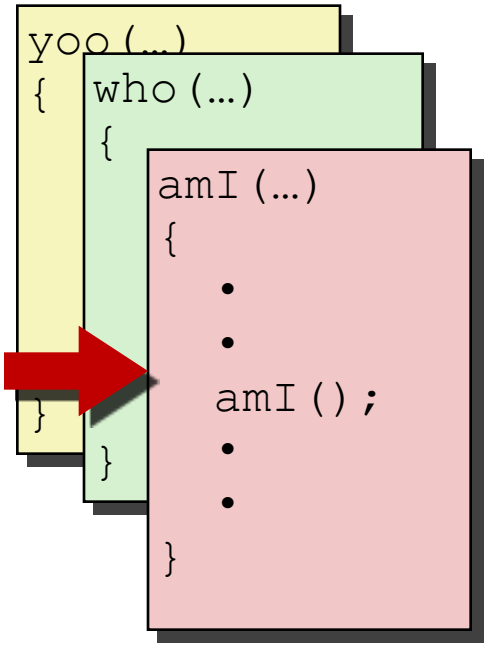
Example

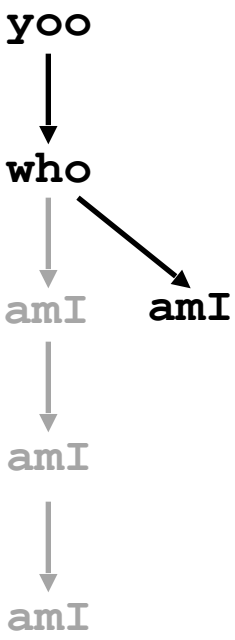


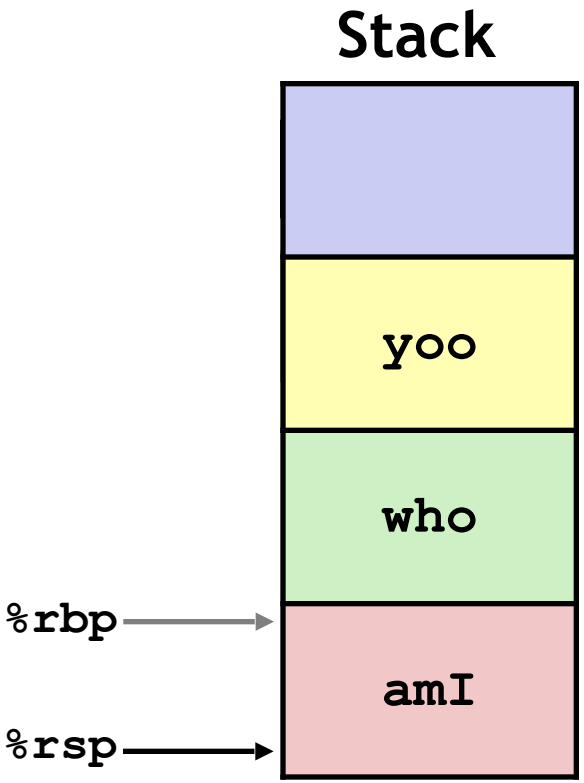




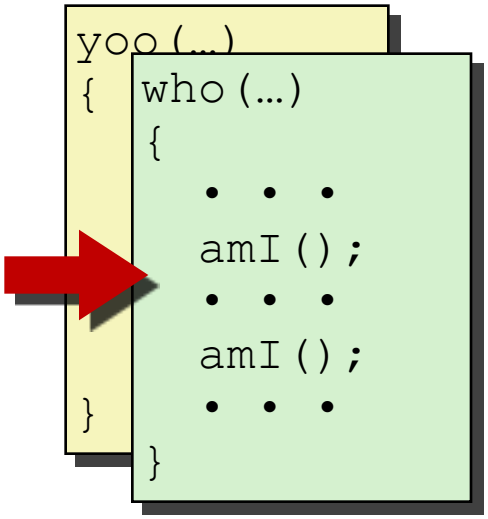
Example

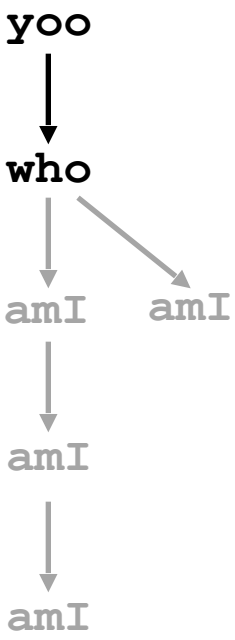

yoo(…)
{
who(…)
{
amI(…)
{
•
•
amI();
•
•
}
}
}

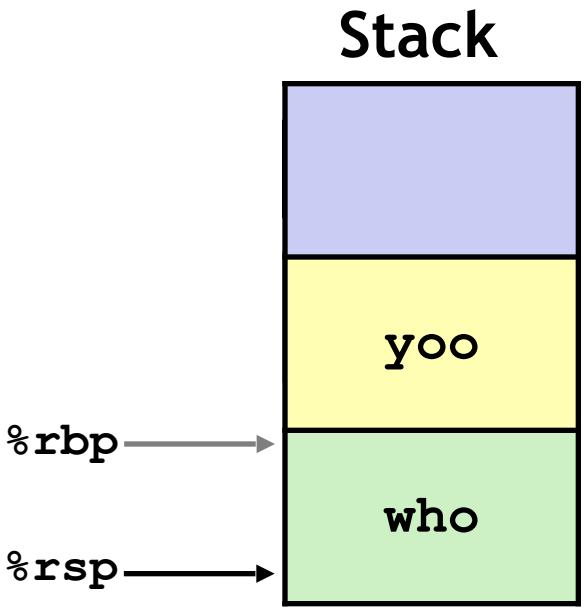

yoo
who
amI
amI
amI
amI


Stack
yoo
who
amI
%rbp
%rsp

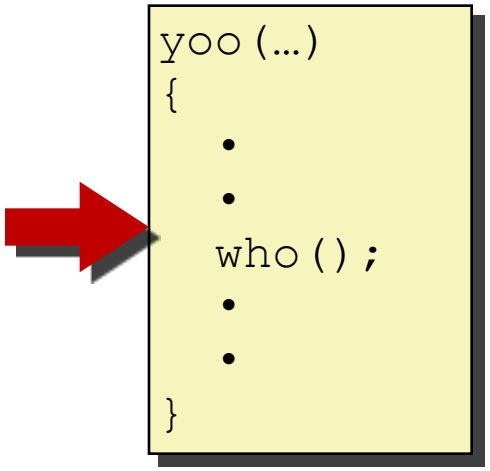
Example

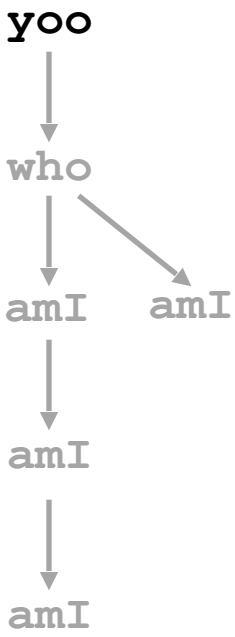


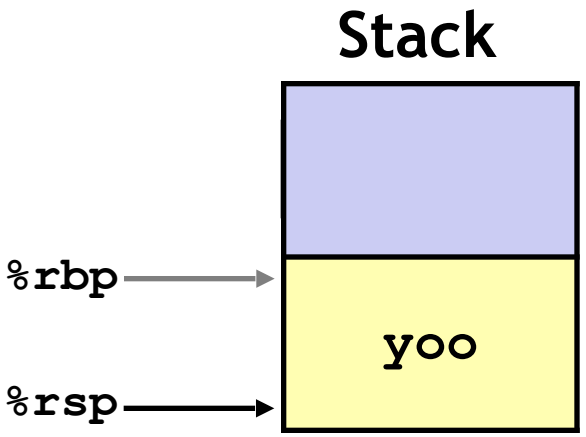




Example







+

Managing Local Data

Register Saving Conventions

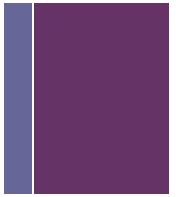
- When procedure yoo calls who:
  - `yoo()` is the ***caller***
  - `who()` is the ***callee***
- Can register be used for temporary storage?

```
yoo:
    • • •
    movq $15213, %rdx
    call who
    addq %rdx, %rax
    • • •
    ret
```

```
who:
    • • •
    subq $18213, %rdx
    • • •
    ret
```

  - Contents of register **%rdx** overwritten by **who**
  - Machine-Level programmer needs to solve for this.

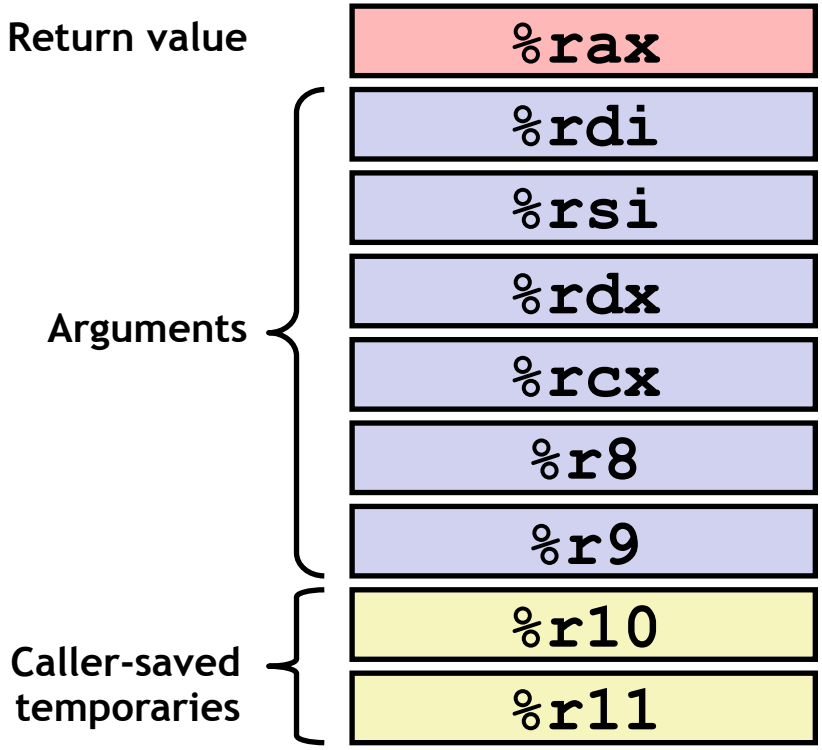
Register Saving Conventions



- When procedure yoo calls who:
  - `yoo()` is the ***caller***
  - `who()` is the ***callee***
- Can register be used for temporary storage?
- **Conventions**
  - *"Caller Saved"*
    - Caller saves temporary values in its frame before the call
  - *"Callee Saved"*
    - Callee saves temporary values in its frame before using
    - Callee restores them before returning to caller

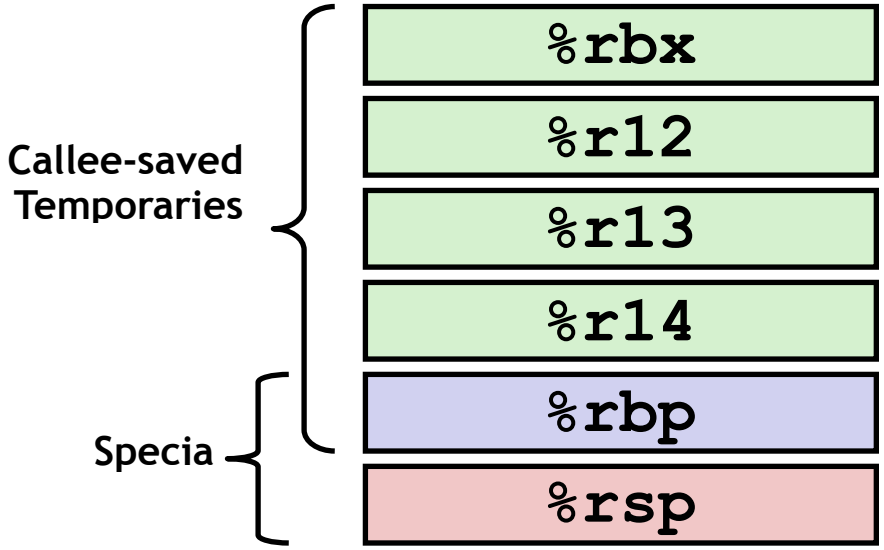
x86-64 Linux Caller-saved Registers

- `%rax`
  - Return value
  - Also caller-saved
  - Can be modified by procedure
- `%rdi, ..., %r9`
  - Arguments
  - Also caller-saved
  - Can be modified by procedure
- `%r10, %r11`
  - Caller-saved
  - Can be modified by procedure



x86-64 Linux Callee-saved Registers

- `%rbx, %r12, %r13, %r14`
  - Callee-saved
  - Callee must save & restore
- `%rbp`
  - Callee-saved
  - Callee must save & restore
  - May be used as frame pointer
- `%rsp`
  - Special form of callee save
  - Restored to original value upon exit from procedure



Callee-Saved Example

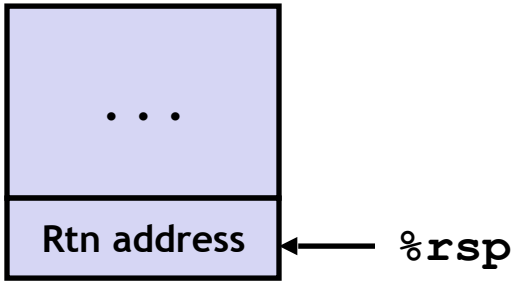
```
long incr(long *p, long val) {
    long x = *p;
    long y = x + val;
    *p = y;
    return x;
}
```

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

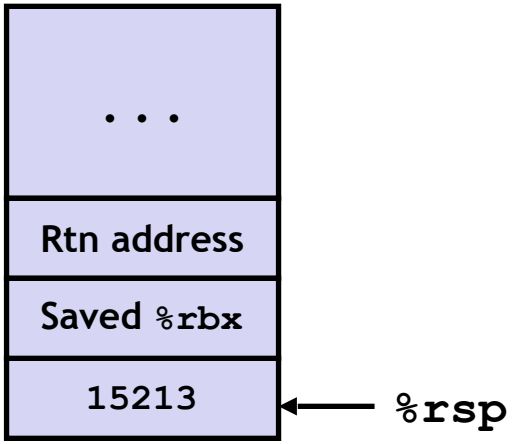
```
call_incr:
  pushq   %rbx            # callee-saved
  subq    $8, %rsp        # allocate
  movq    %rdi, %rbx      # caller-saved
  movq    $15213, (%rsp)  # "push"
  movq    $3000, %rsi
  leaq    (%rsp), %rdi
  call    incr
  addq    %rbx, %rax
  addq    $8, %rsp
  popq    %rbx
  ret
```

call_incr's stack

Initial Stack



Resulting Stack



Callee-Saved Example *con't*

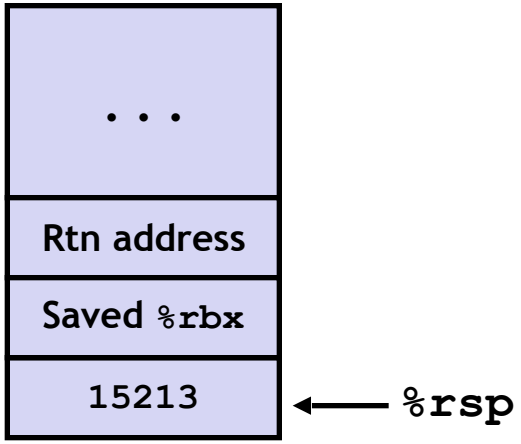
```
long incr(long *p, long val) {
    long x = *p;
    long y = x + val;
    *p = y;
    return x;
}
```

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

```
call_incr:
  pushq   %rbx            # callee-saved
  subq    $8, %rsp        # allocate
  movq    %rdi, %rbx      # caller-saved
  movq    $15213, (%rsp)  # "push"
  movq    $3000, %rsi
  leaq    (%rsp), %rdi
  call    incr
  addq    %rbx, %rax
  addq    $8, %rsp        # deallocate
  popq    %rbx            # restore %rbx
  ret
```

call_incr's stack

Resulting Stack



Pre-return Stack