Machine-Level Programming III: Procedures

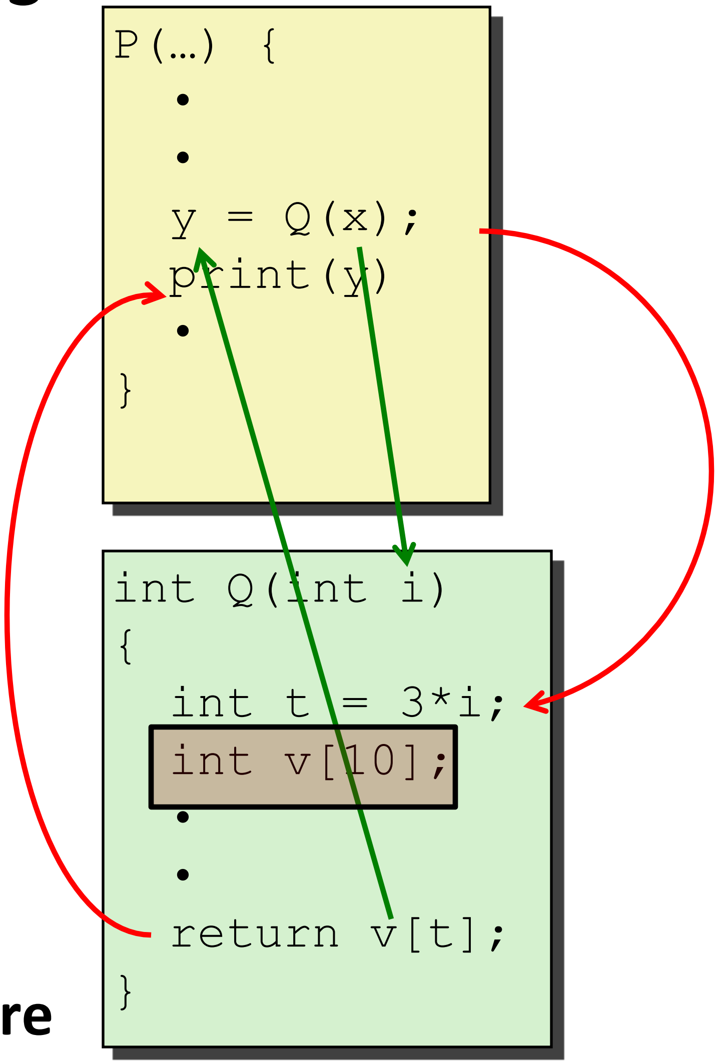
CS-392-A Systems Programming

Instructor:

Georgios Portokalidis – Stevens Institute of Technology

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**



```
P(…) {
   •
   •
   y = Q(x);
   print(y)
   •
}
```

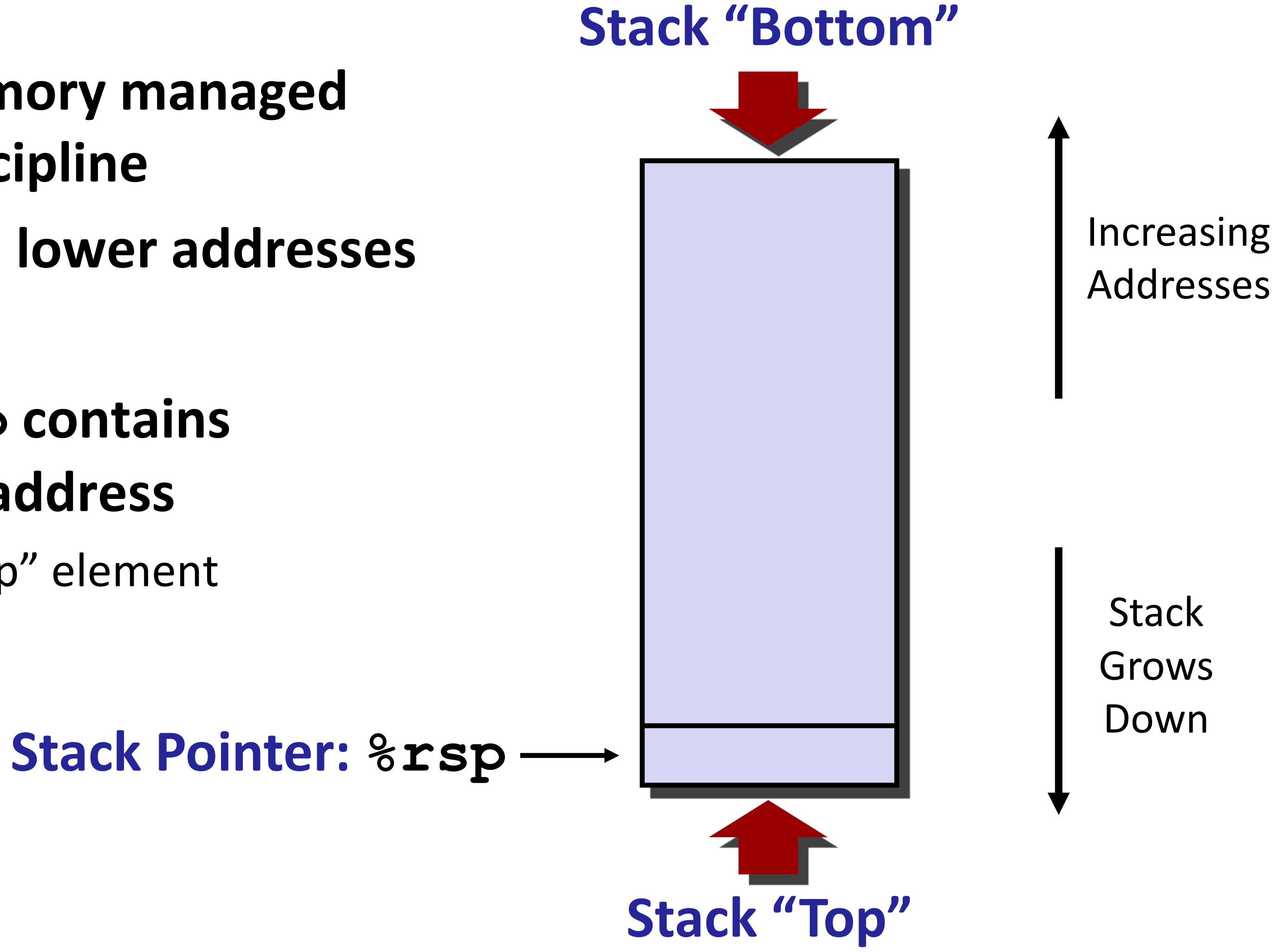
```
int Q(int i)
{
   int t = 3*i;
   int v[10];
   •
   •
   return v[t];
}
```

Today

- **Procedures**
  - **Stack Structure**
  - Calling Conventions
    - Passing control
    - Passing data
    - Managing local data

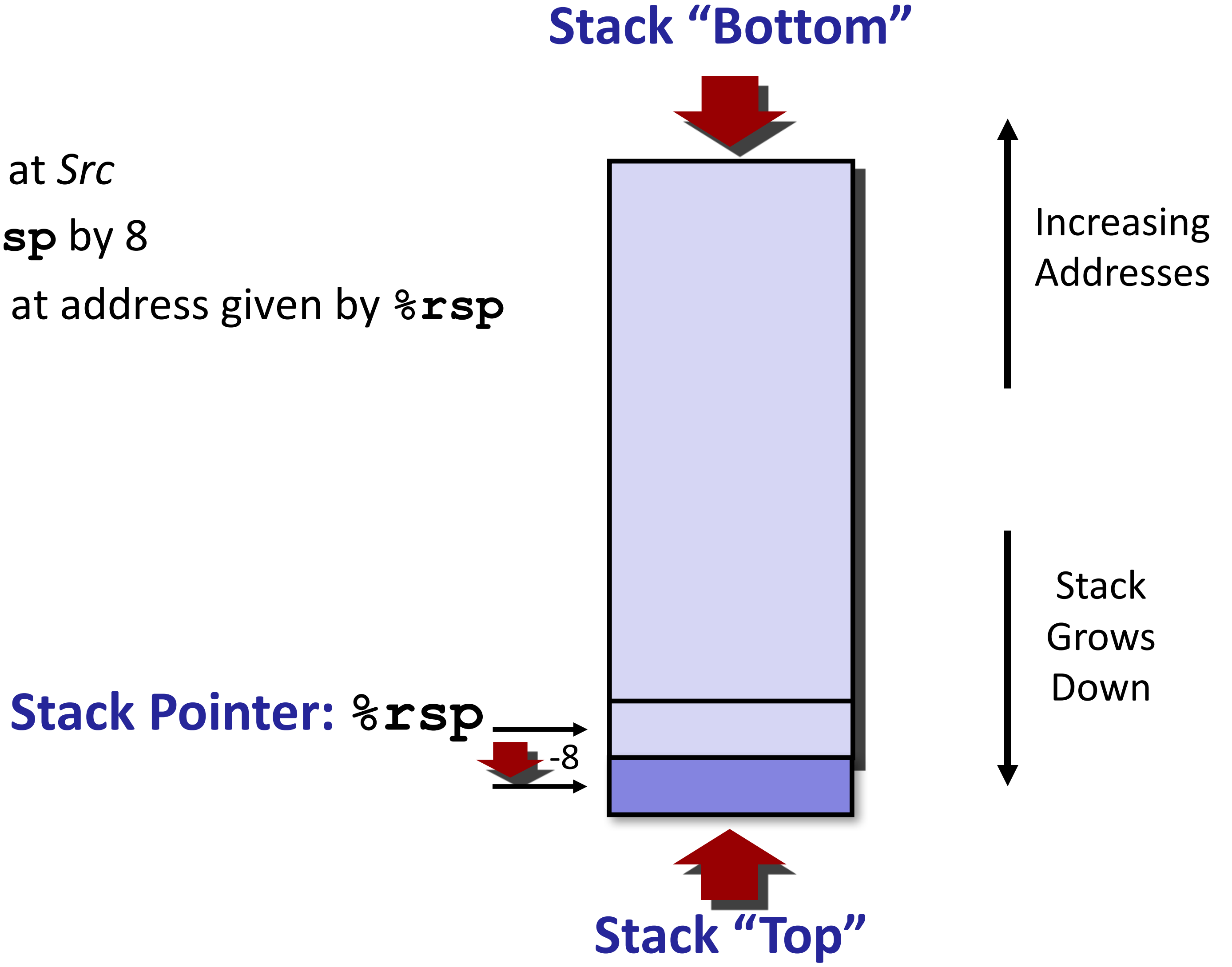
x86-64 Stack

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



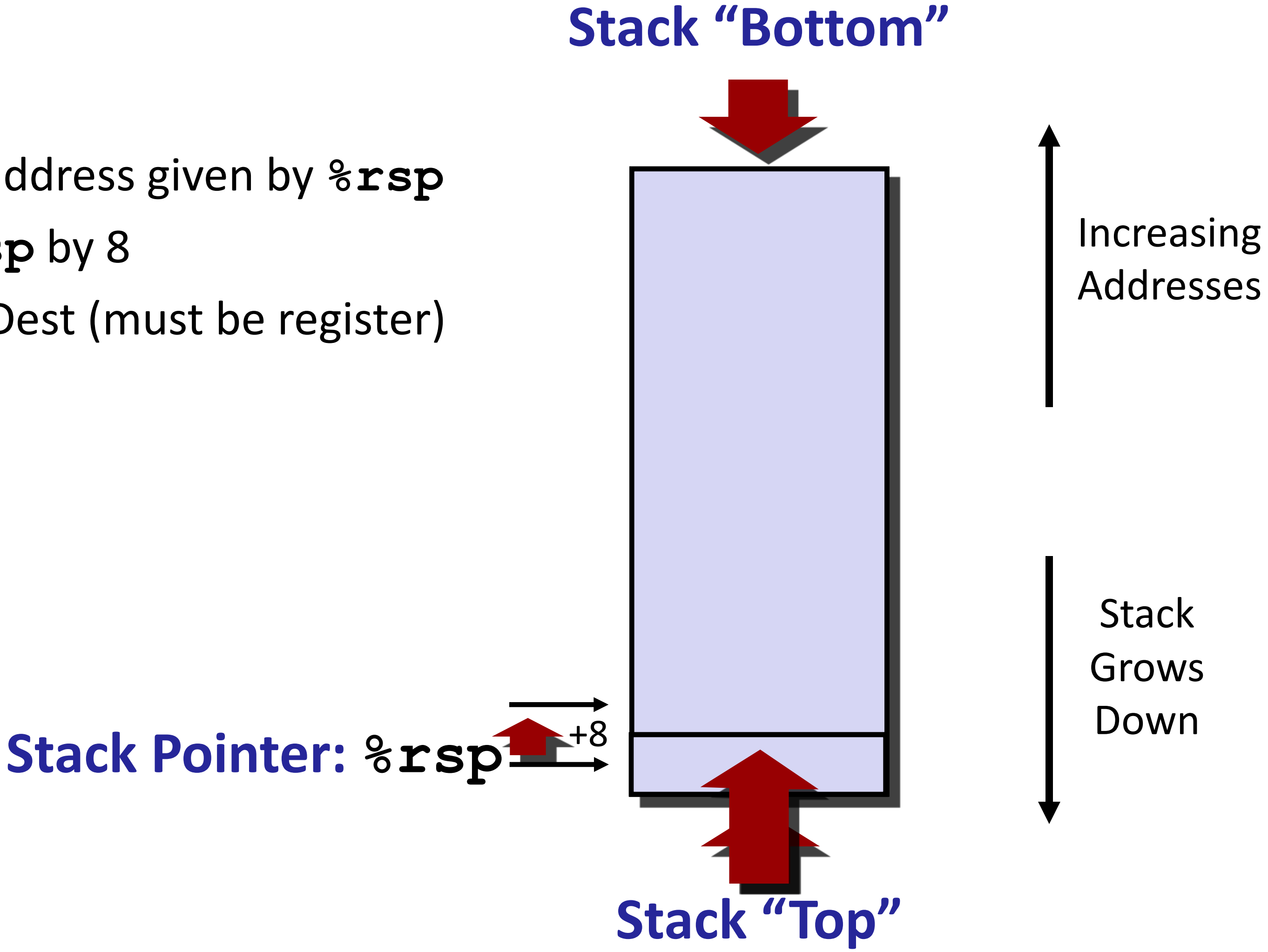
x86-64 Stack: Push

- **`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)



Today

- **Procedures**
  - Stack Structure
  - Calling Conventions
    - **Passing control**
    - Passing data
    - Managing local data

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
```

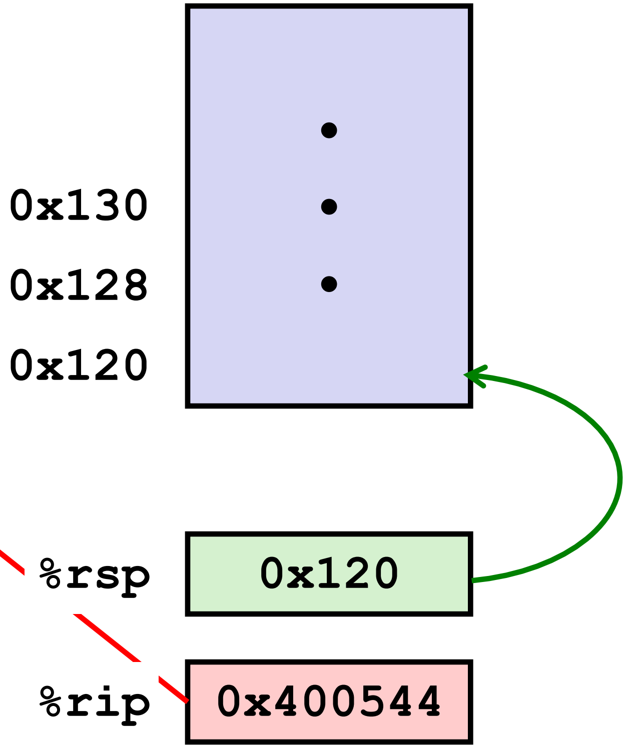
Procedure Control Flow

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

Control Flow Example #1

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

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



0x130

0x128

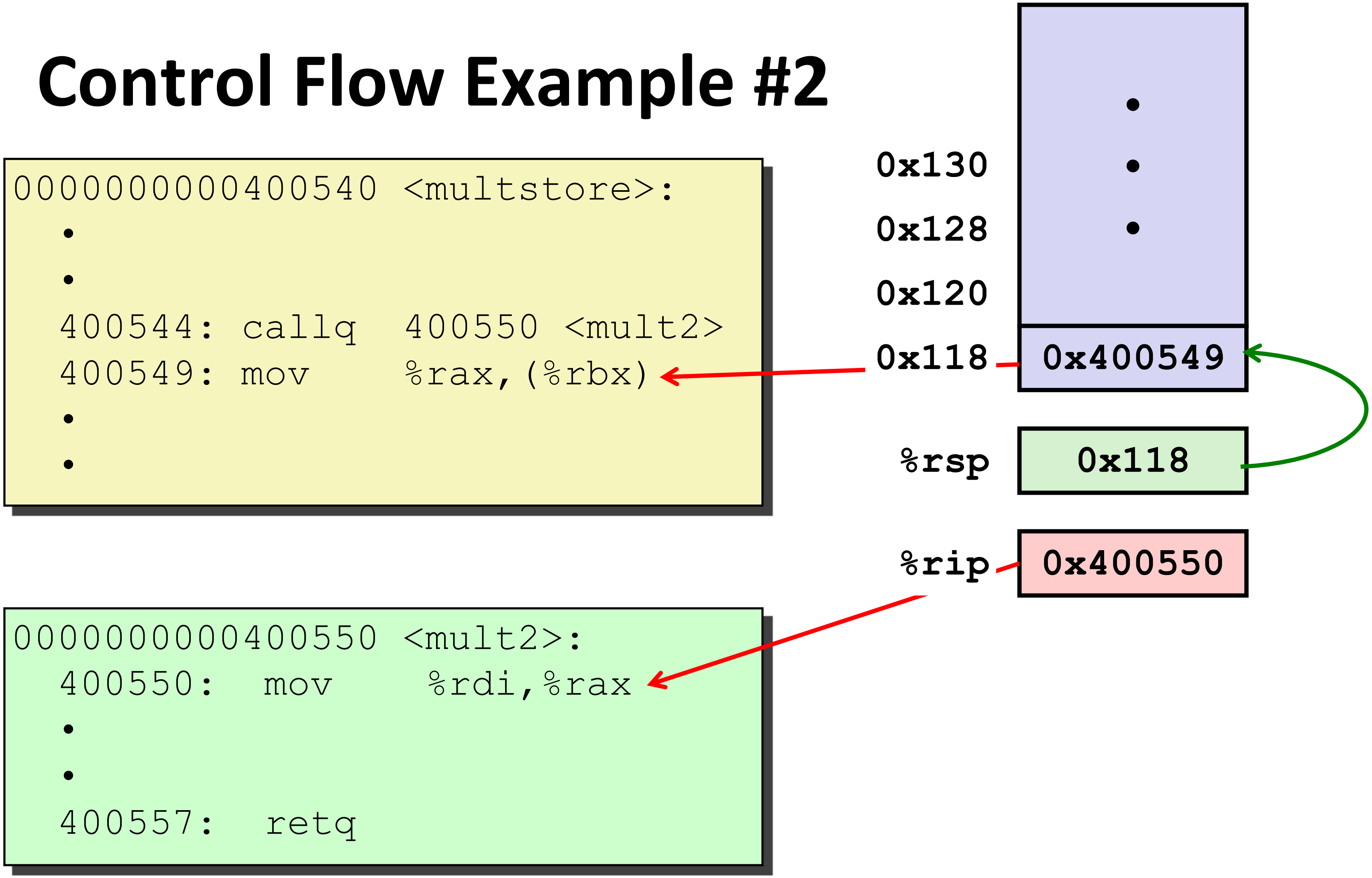
0x120

•
•
•

%rsp 0x120

%rip 0x400544

Control Flow Example #2



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

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

•
•
•

0x130

0x128

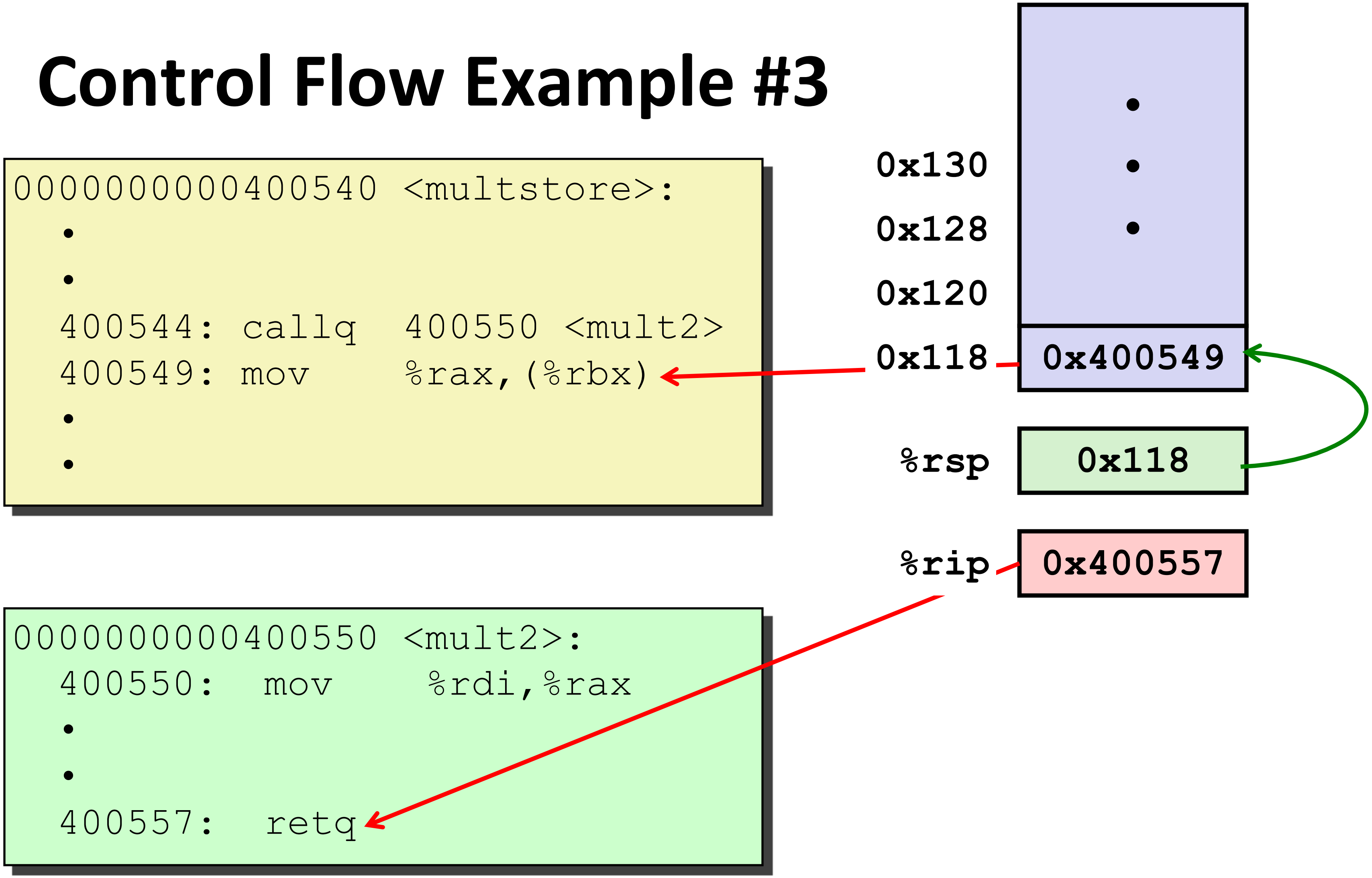
0x120

0x118 0x400549

%rsp 0x118

%rip 0x400550

Control Flow Example #3



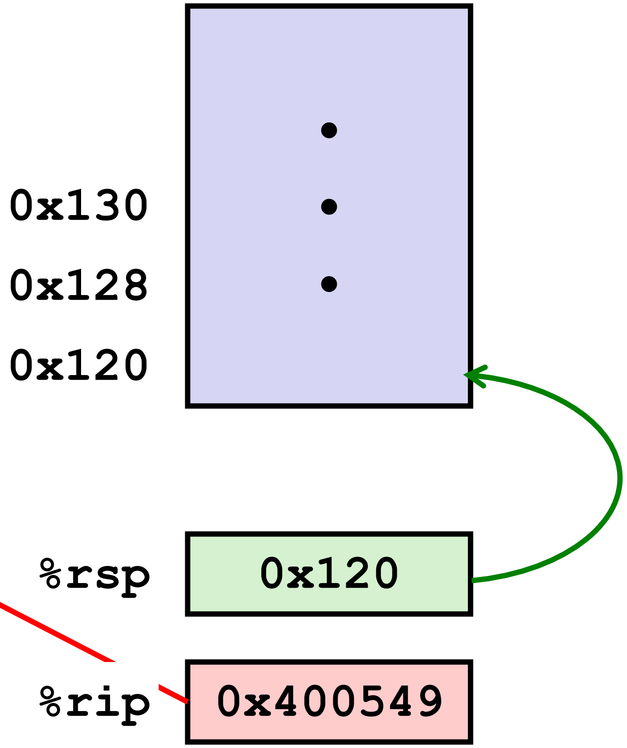
```
0000000000400540 <multstore>:
  •
  •
  400544: callq  400550 <mult2>
  400549: mov    %rax,(%rbx)
  •
  •
```

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

Control Flow Example #4

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

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



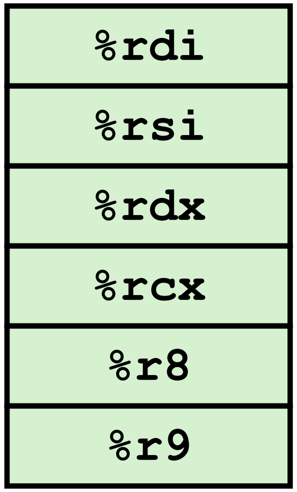
Today

- **Procedures**
  - Stack Structure
  - Calling Conventions
    - Passing control
    - **Passing data**
    - Managing local data

Procedure Data Flow

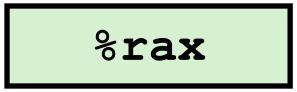
Registers

- First 6 arguments



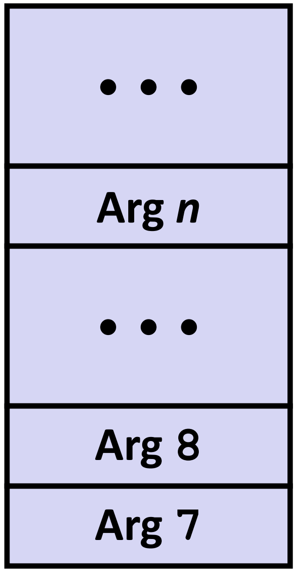
%rdi
%rsi
%rdx
%rcx
%r8
%r9

- Return value



%rax

Stack



• • •
Arg *n*
• • •
Arg 8
Arg 7

- 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:  retq                  # Return
```

Calling Conventions

- **What we saw is the System V AMD64 ABI calling convention**
  - Function arguments passed in: RDI, RSI, RDX, RCX, R8, R9, then the stack
  - Used on most UNIX systems (Linux, BSD, Solaris, OS X)
- **Other systems use different conventions**
  - Microsoft vectorcall
    - Introduced in Visual Studio 2013
    - Function arguments passed in: RCX, RDX, R8, R9, then the stack
    - Return value on RAX
  - Cdecl (C declaration)
    - Used on 32-bit Linux systems
    - Function arguments passed in the stack
    - Return value on EAX
  - Others https://en.wikipedia.org/wiki/X86_calling_convention

Today

- **Procedures**
  - Stack Structure
  - Calling Conventions
    - Passing control
    - Passing data
    - **Managing local data**

Stack-Based Languages

- **Languages that support recursion**
  - e.g., C, Python, Java
  - Code must be “*Reentrant*”
    - Multiple simultaneous instantiations of single procedure
  - Need some place to store state of each instantiation
    - Arguments
    - Local variables
    - Return pointer
- **Stack discipline**
  - State for given procedure needed for limited time
    - From when called to when return
  - Callee returns before caller does
- **Stack allocated in *Frames***
  - state for single procedure instantiation

Call Chain Example

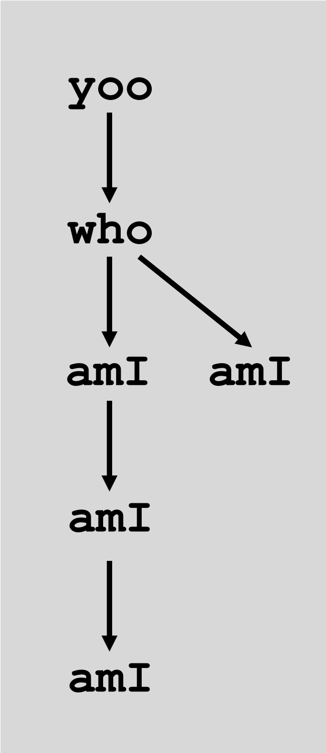
```
yoo(…)
{
   •
   •
   who();
   •
   •
}
```

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

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

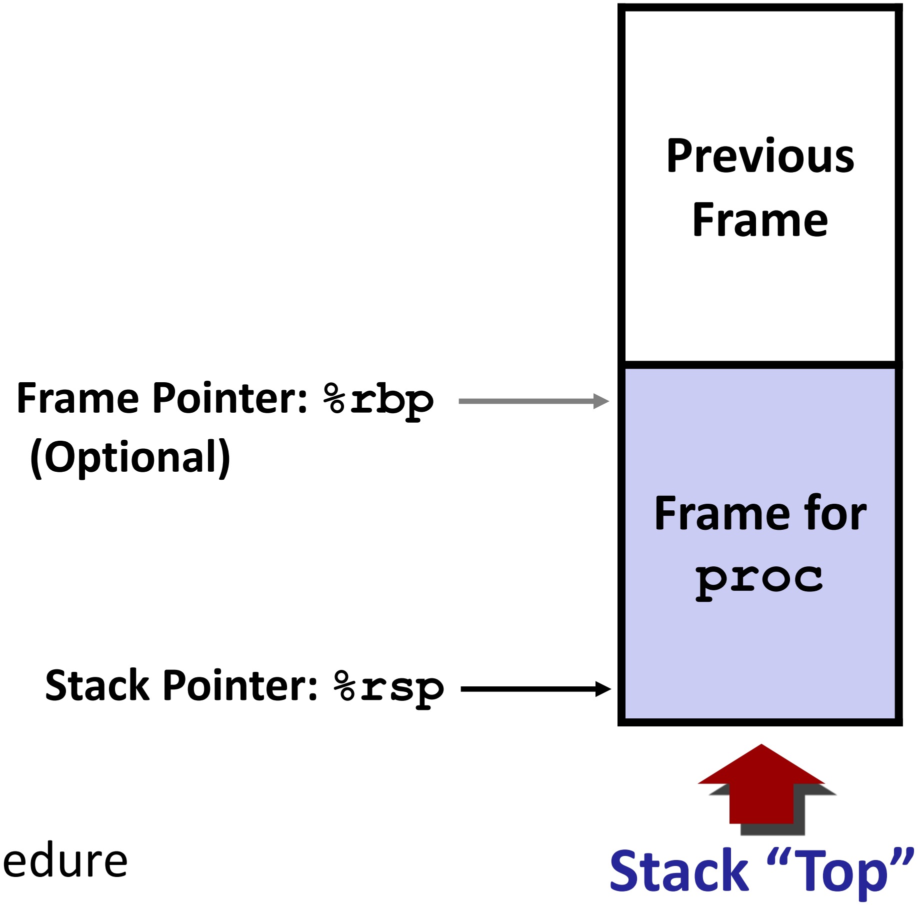
Procedure amI() is recursive

Example Call Chain



Stack Frames

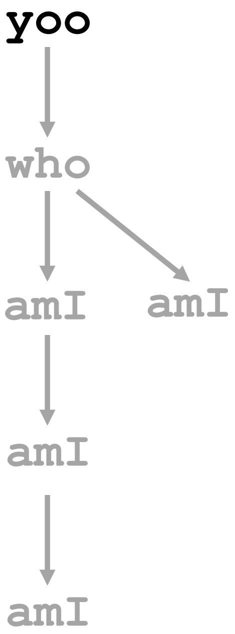
- **Contents**
  - Return information
  - Local storage (if needed)
  - Temporary space (if needed)

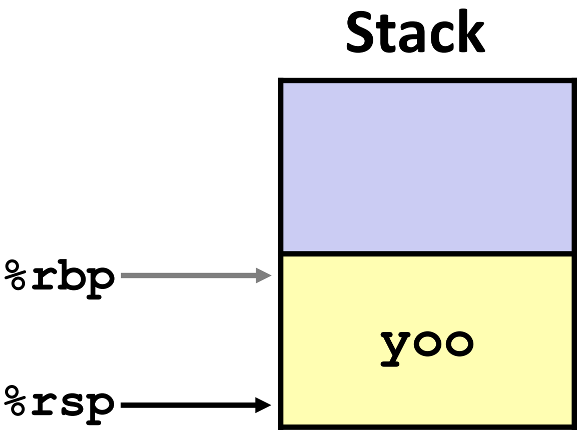


- **Management**
  - Space allocated when enter procedure
    - "Set-up" code
    - Includes push by **`call`** instruction
  - Deallocated when return
    - "Finish" code
    - Includes pop by **`ret`** instruction

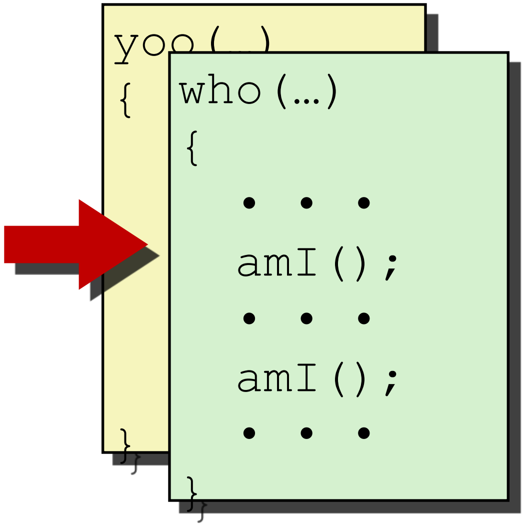
Example

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



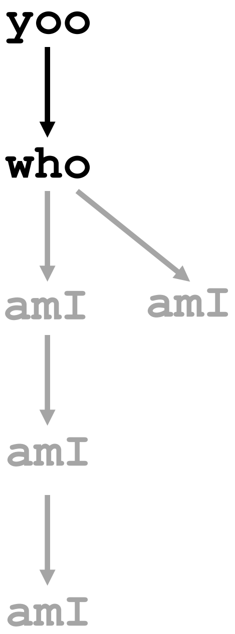


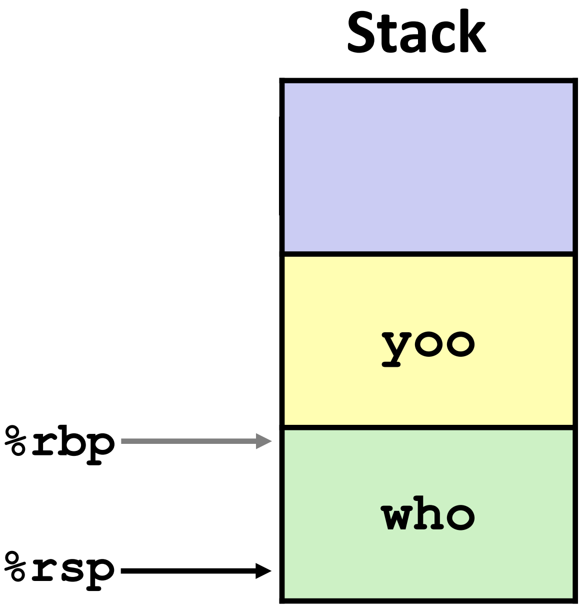
Example



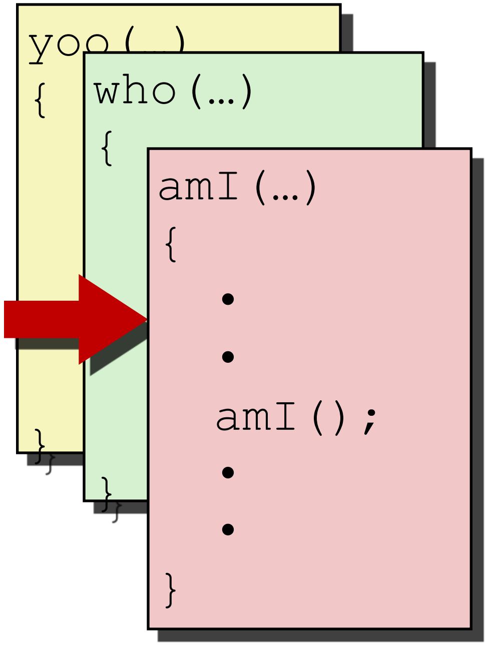
```
yoo(…)
{
  
  
}
```

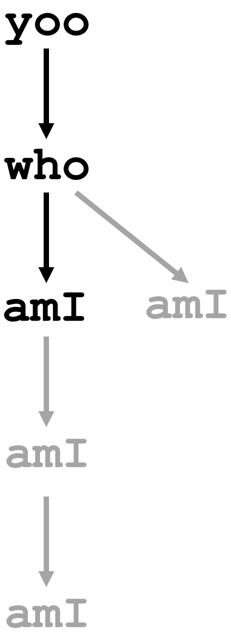
```
who(…)
{
   • • •
   amI();
   • • •
   amI();
   • • •
}
```

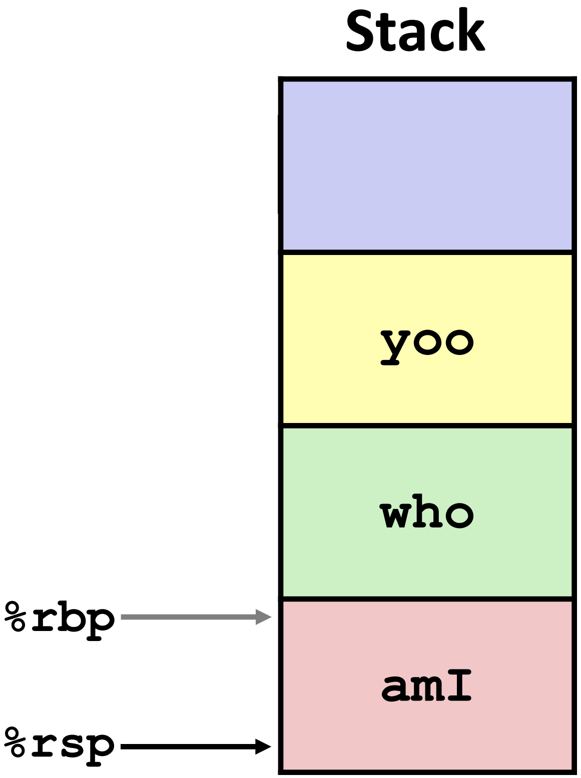




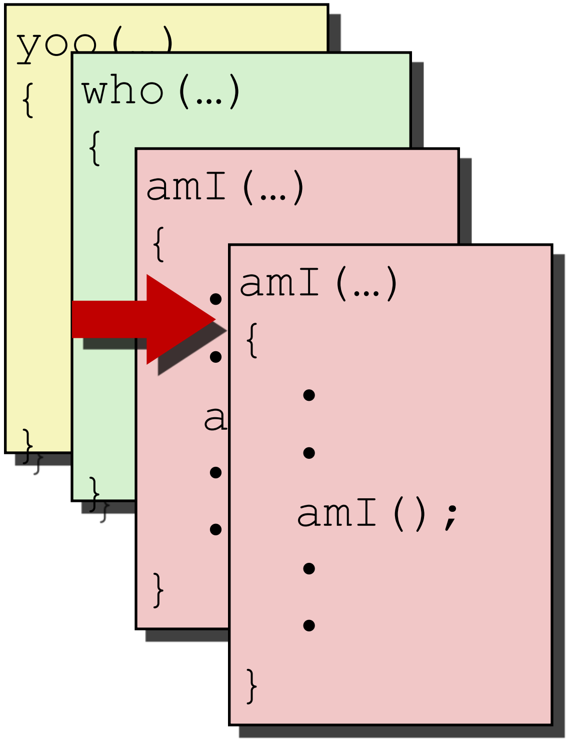
Example

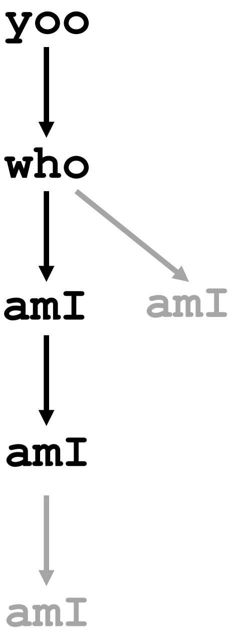

yoo(…)
{
}
who(…)
{
}
amI(…)
{
•
•
amI();
•
•
}

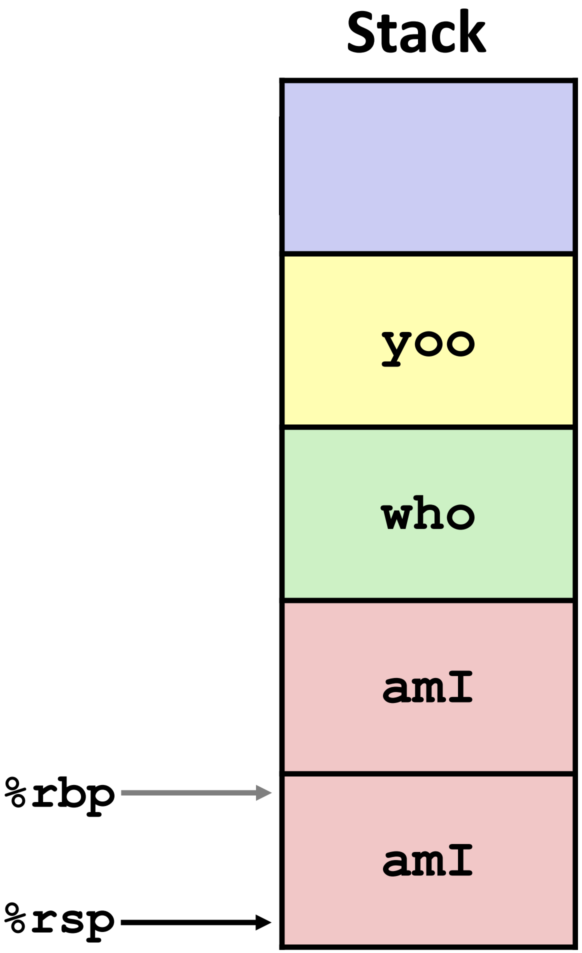

yoo
who
amI
amI
amI
amI


Stack
yoo
who
amI
%rbp
%rsp

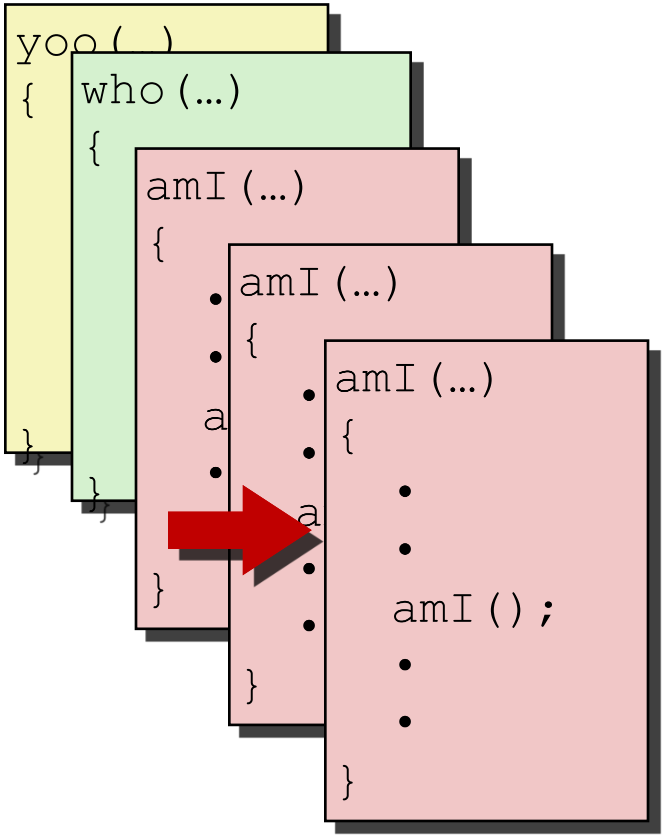
Example

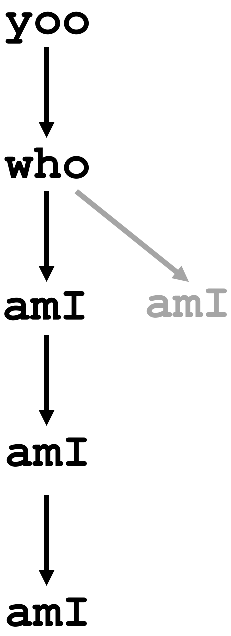

yoo (…)
{
}
who (…)
{
}
amI (…)
{
•
•
a
•
•
}
amI (…)
{
•
•
amI ();
•
•
}

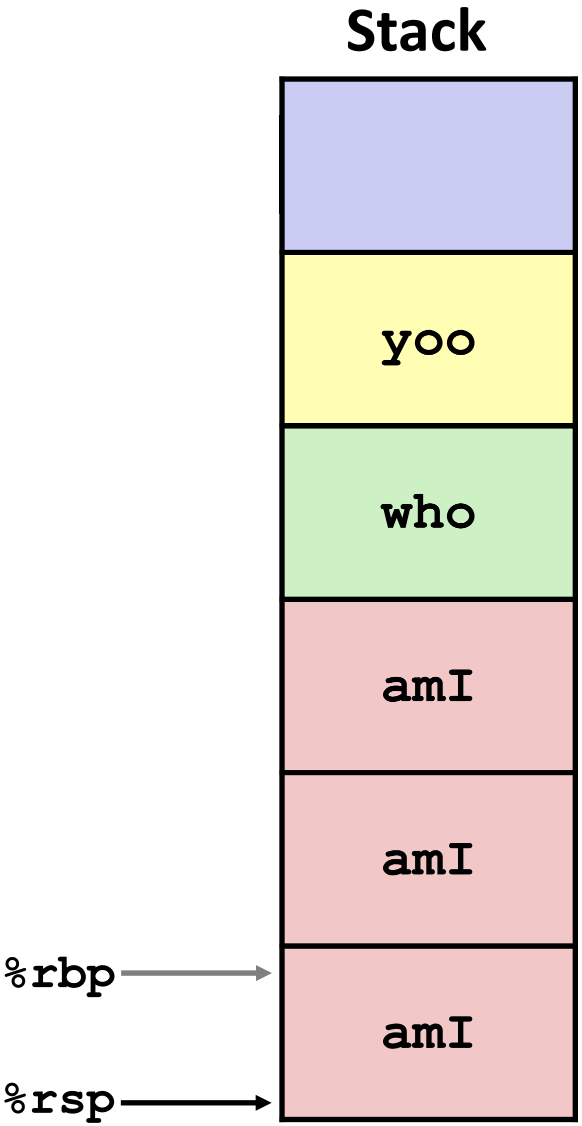

yoo
who
amI
amI
amI
amI


Stack
yoo
who
amI
amI
%rbp
%rsp

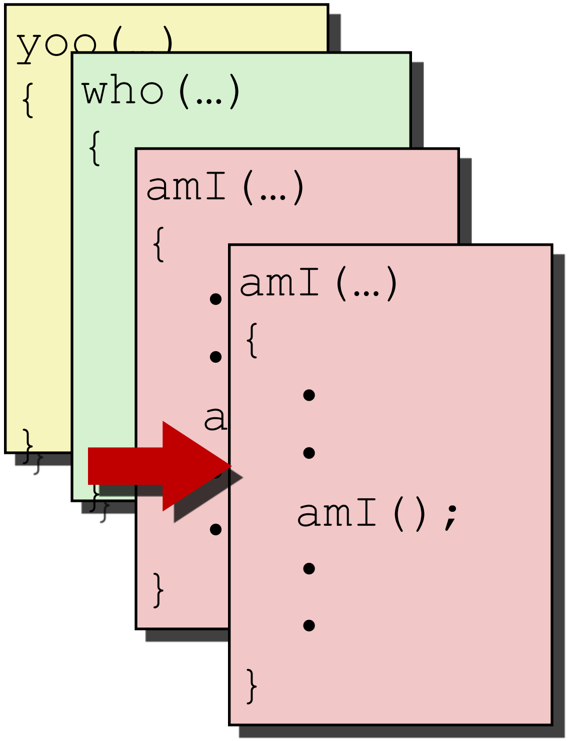
Example

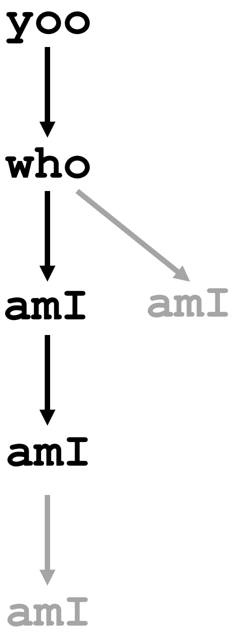

yoo(…)
{
}
who(…)
{
}
amI(…)
{
•
•
a
•
}
amI(…)
{
•
•
a
•
•
}
amI(…)
{
•
•
amI();
•
•
}

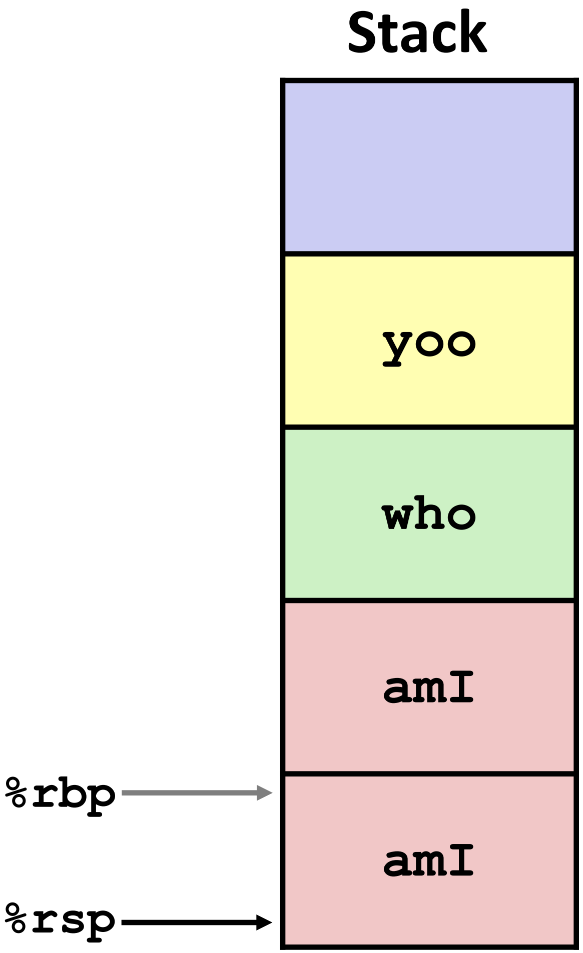

yoo
who
amI
amI
amI
amI


Stack
yoo
who
amI
amI
amI
%rbp
%rsp

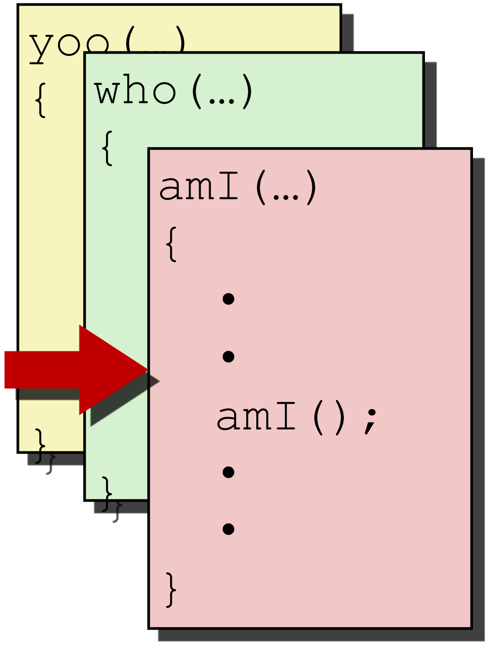
Example


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


yoo
who
amI
amI
amI
amI


Stack
yoo
who
amI
amI
%rbp
%rsp

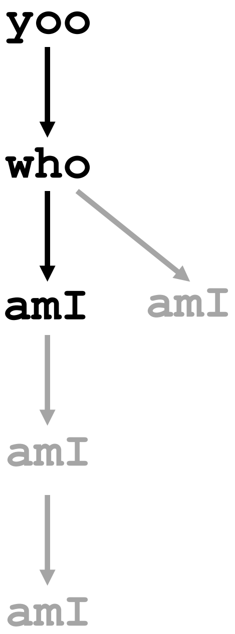
Example

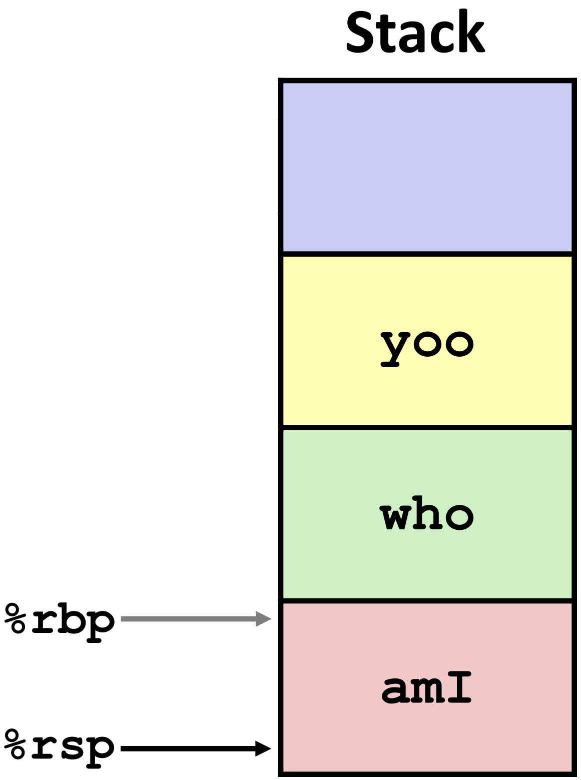


```
yoo(…)
{
  •
  •
  who();
  •
  •
}
```

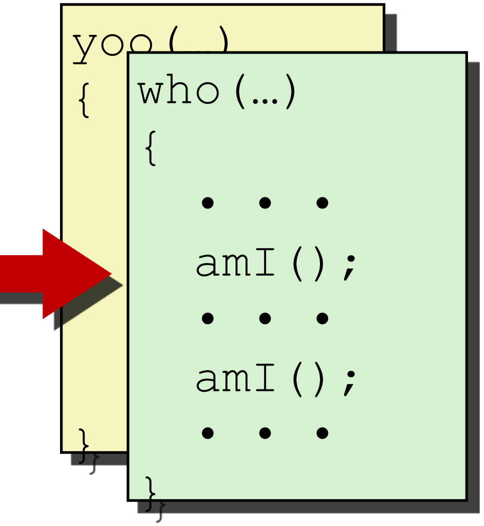
```
who(…)
{
  • • •
  amI();
  • • •
  amI();
  • • •
}
```

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



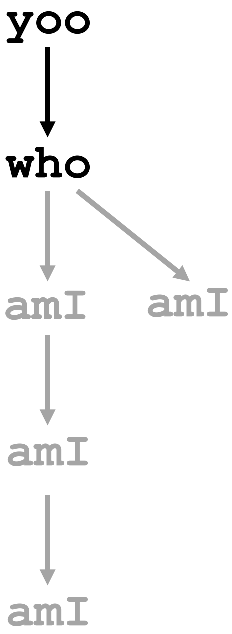


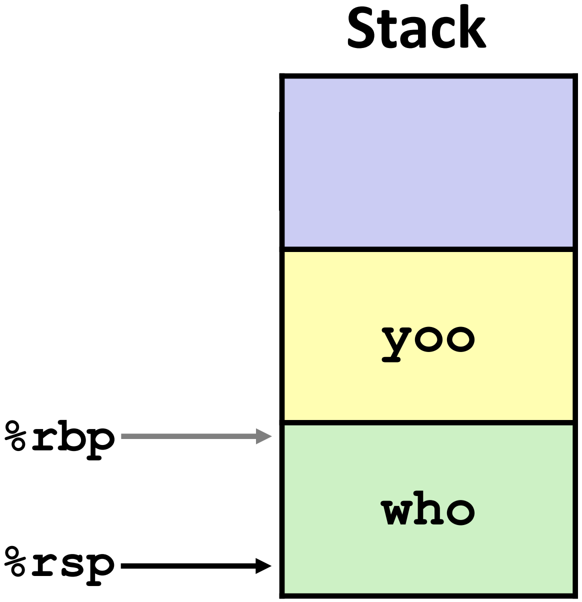
Example



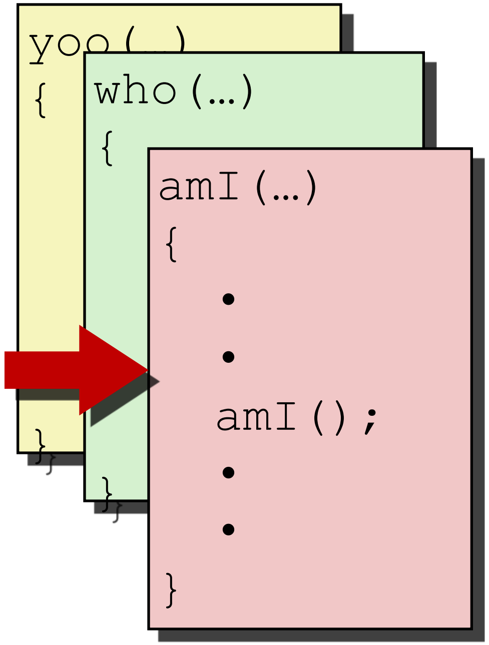
```
yoo(…)
{
}
```

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

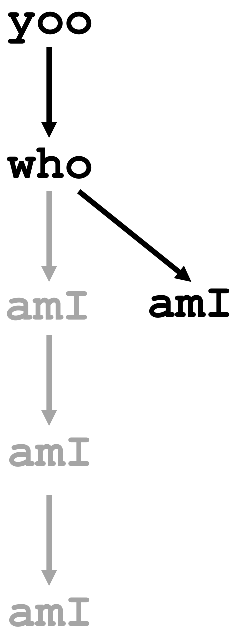


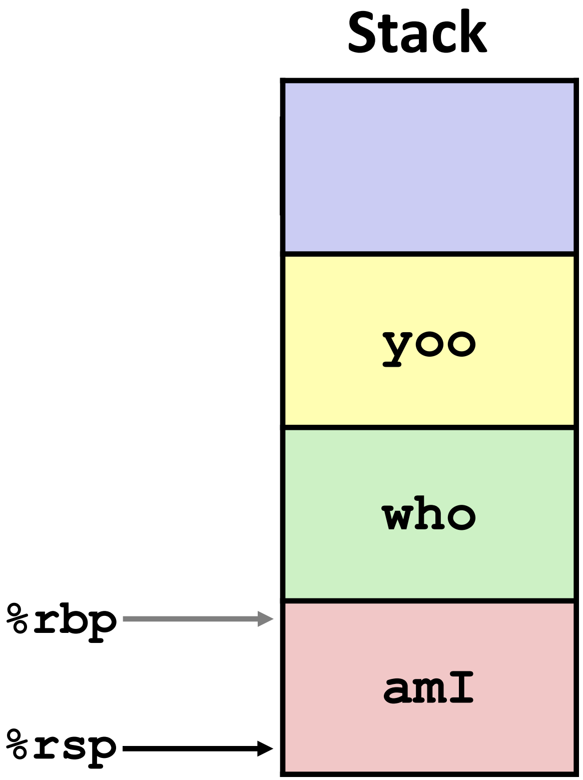


Example

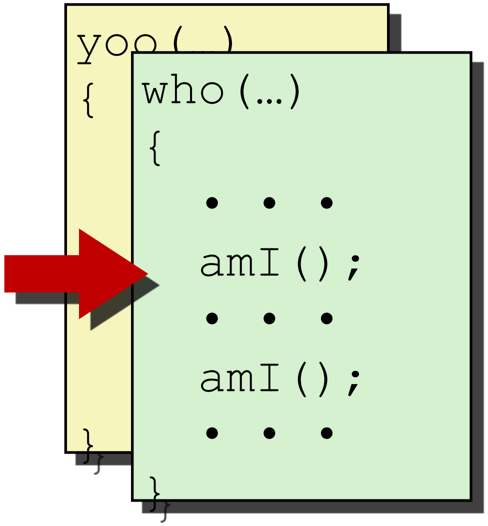


```
yoo(…)
{
  who(…)
  {
    amI(…)
    {
       •
       •
       amI();
       •
       •
    }
  }
}
```



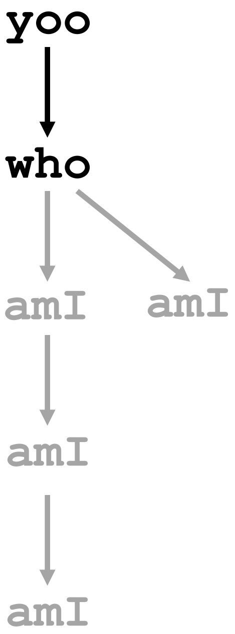


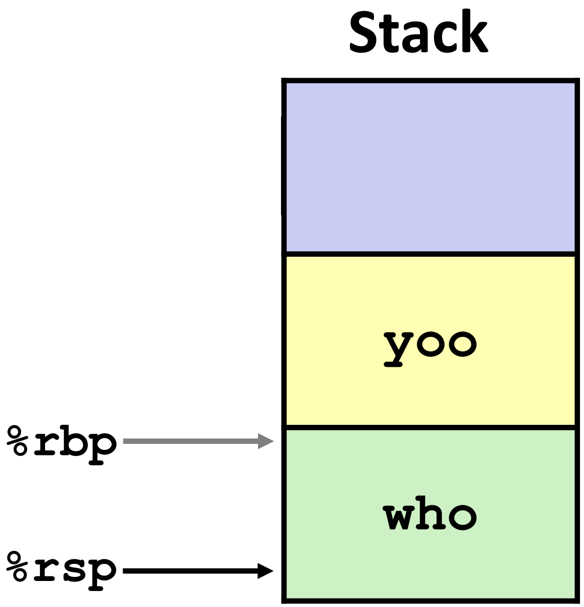
Example



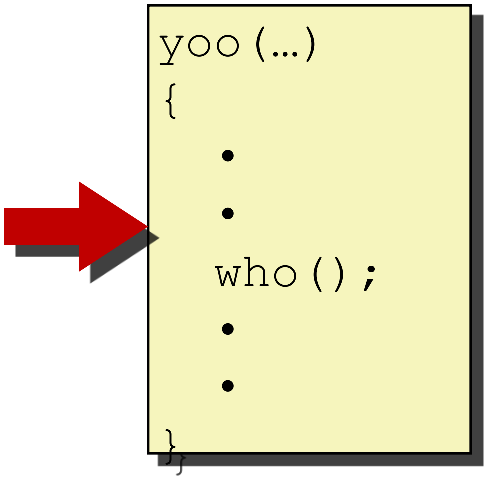
```
yoo(…)
{

}
```

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

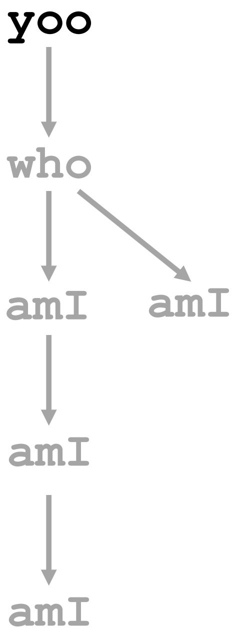


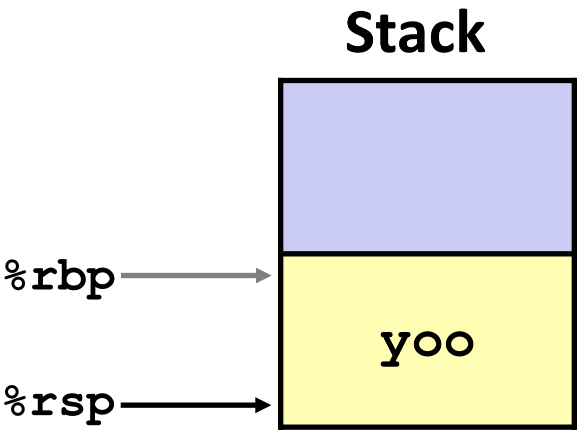


Example



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

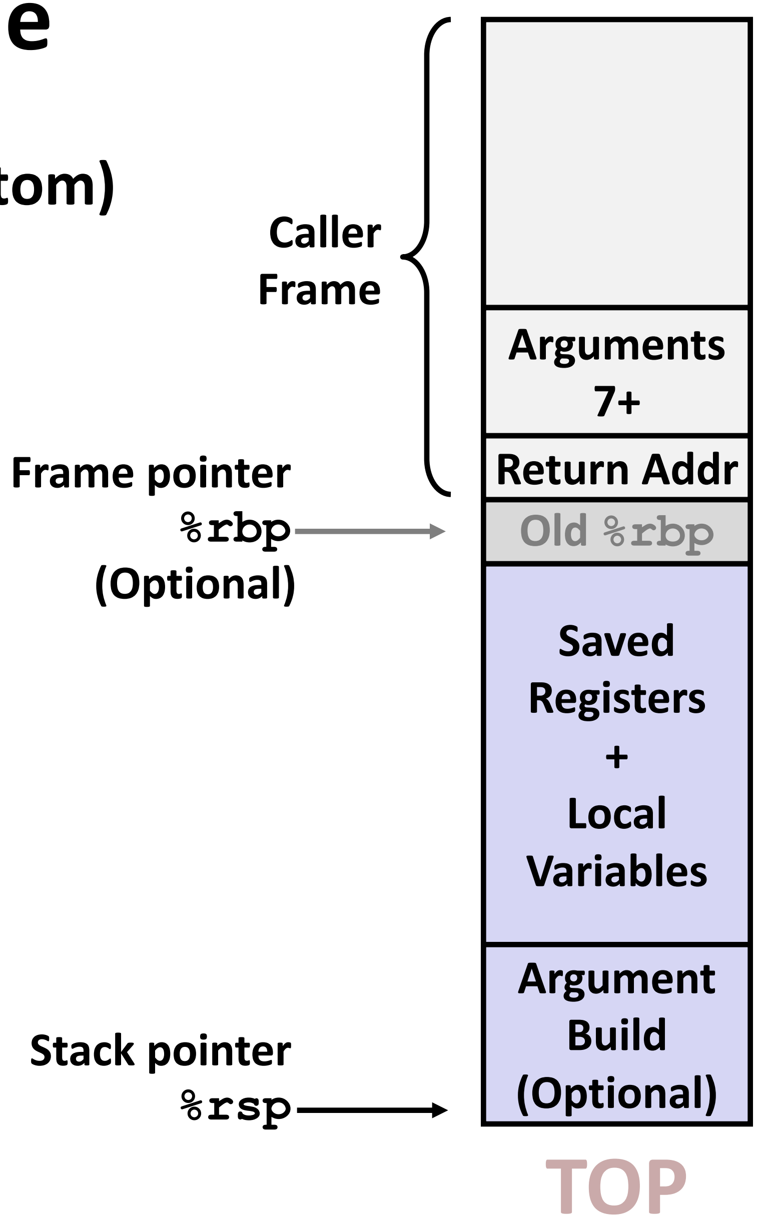




x86-64/Linux Stack Frame

- **Current Stack Frame ("Top" to Bottom)**
  - "Argument build:"
    Parameters for called function
  - Local variables
    If can't keep in registers
  - Saved register context
  - Old frame pointer (optional)

- **Caller Stack Frame**
  - Return address
    - Pushed by `call` instruction
  - Arguments for this call



Example: incr

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

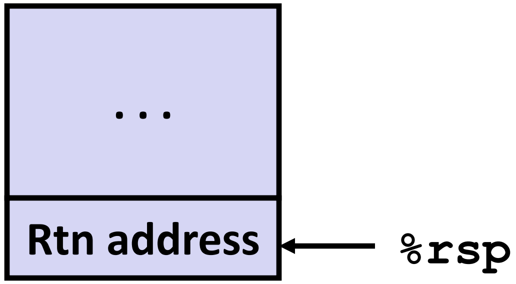
```
incr:
  movq    (%rdi), %rax
  addq    %rax, %rsi
  movq    %rsi, (%rdi)
  ret
```

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

Example: Calling incr #1

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

Initial Stack Structure

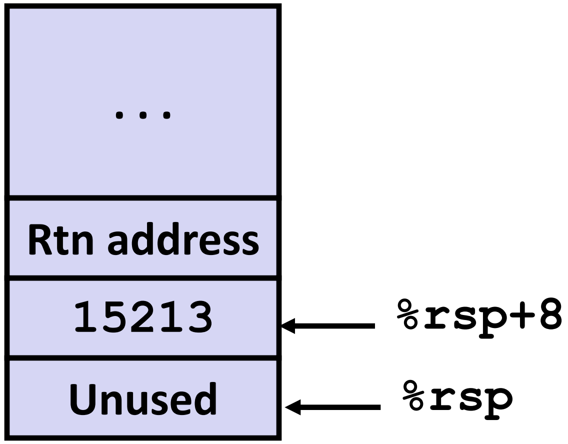

. . .

Rtn address

%rsp

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

Resulting Stack Structure


. . .

Rtn address

15213

Unused

%rsp+8

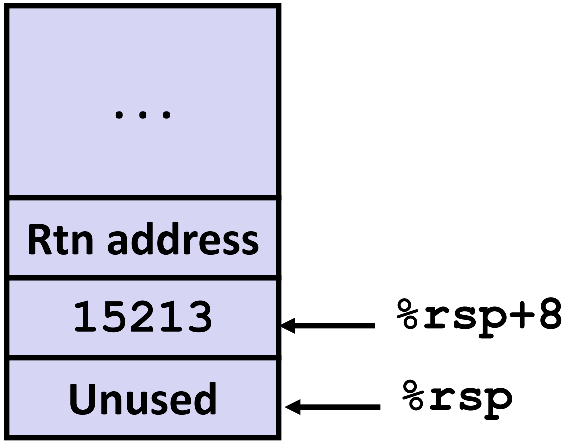
%rsp

Example: Calling incr #2

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

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

Stack Structure



. . .

Rtn address

15213

Unused

%rsp+8

%rsp

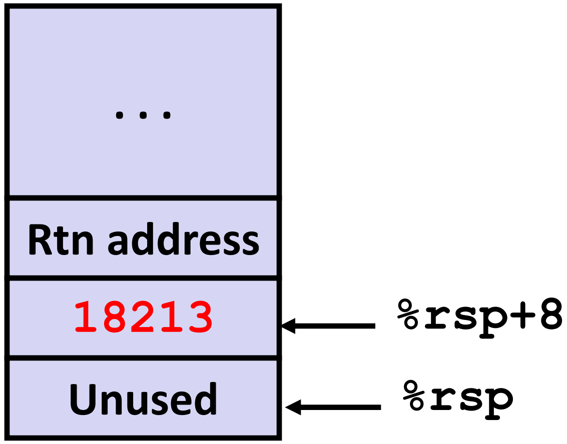
Register	Use(s)
%rdi	&v1
%rsi	3000

Example: Calling incr #3

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

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

Stack Structure



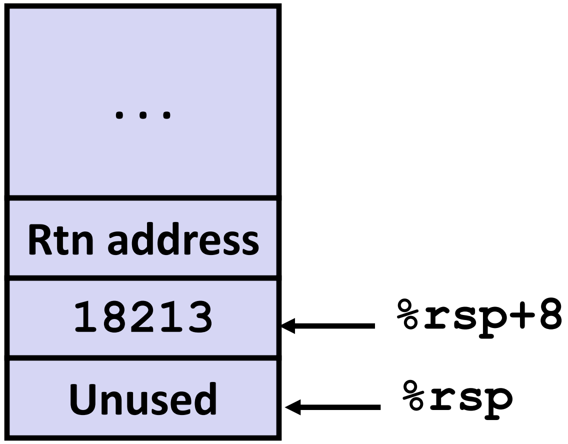
Register	Use(s)
%rdi	&v1
%rsi	3000

Example: Calling `incr` #4

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

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

Stack Structure



...

Rtn address

18213

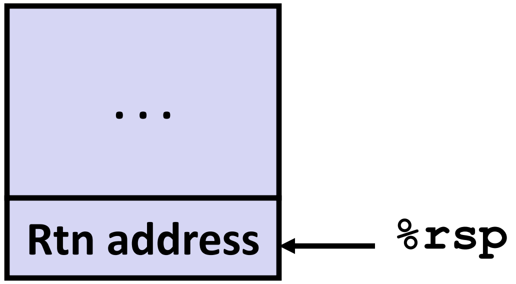
Unused

%rsp+8

%rsp

Register	Use(s)
%rax	Return value

Updated Stack Structure



...

Rtn address

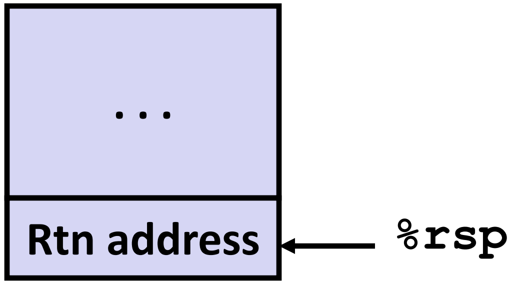
%rsp

Example: Calling `incr` #5

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

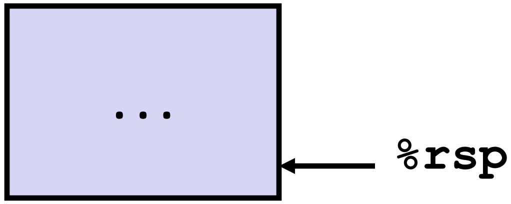
```
call_incr:
  subq    $16, %rsp
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

Updated Stack Structure



Register	Use(s)
`%rax`	Return value

Final Stack Structure



Register Saving Conventions

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

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

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

Register Saving Conventions

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

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

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

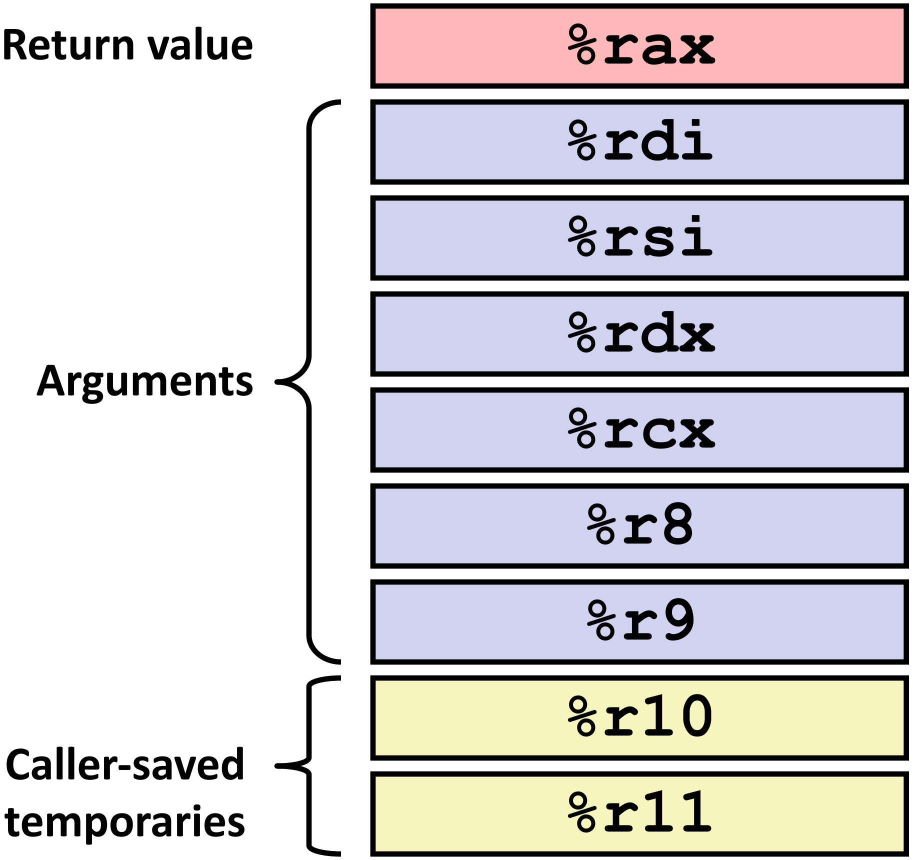
  - Contents of register `%rdx` overwritten by `who`
  - This could be trouble → we need more conventions

Register Saving Conventions

- **When procedure `yoo` calls `who`:**
  - `yoo` is the ***caller***
  - `who` is the ***callee***
- **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
- **Also part of the calling convention**

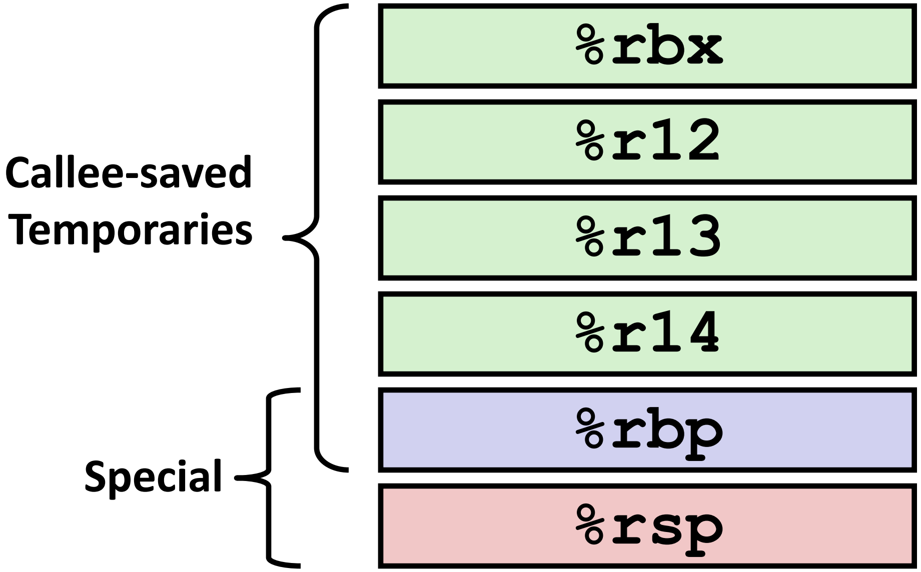
x86-64 Linux Register Usage #1

- **`%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 Register Usage #2

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

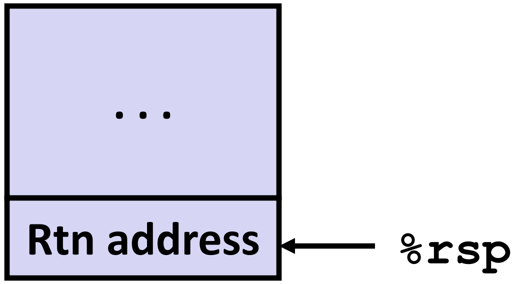


Callee-Saved Example #1

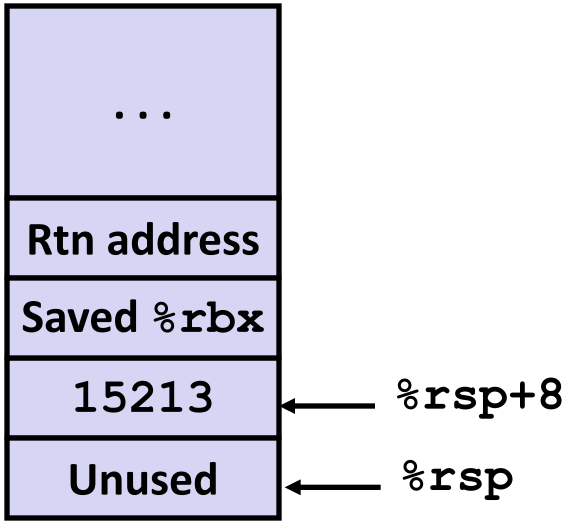
```
long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```

```
call_incr2:
  pushq    %rbx
  subq     $16, %rsp
  movq     %rdi, %rbx
  movq     $15213, 8(%rsp)
  movl     $3000, %esi
  leaq     8(%rsp), %rdi
  call     incr
  addq     %rbx, %rax
  addq     $16, %rsp
  popq     %rbx
  ret
```

Initial Stack Structure



Resulting Stack Structure

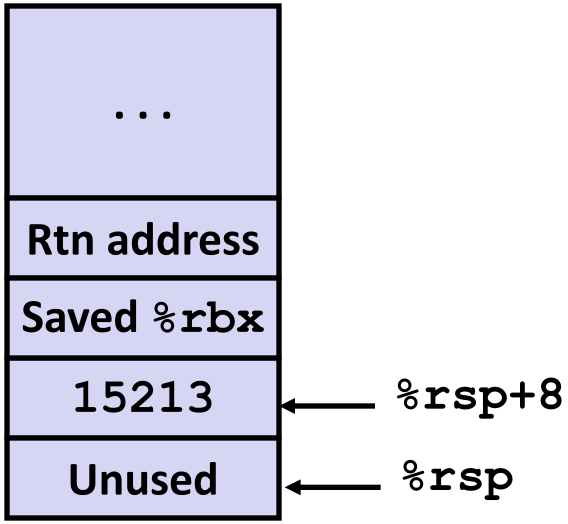


Callee-Saved Example #2

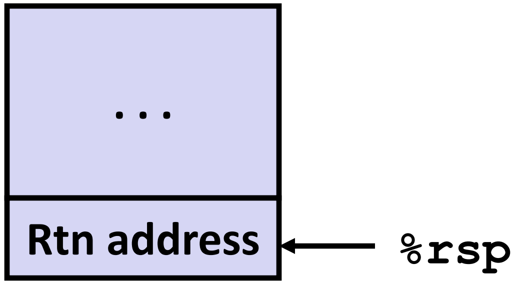
```
long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```

```
call_incr2:
  pushq   %rbx
  subq    $16, %rsp
  movq    %rdi, %rbx
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    %rbx, %rax
  addq    $16, %rsp
  popq    %rbx
  ret
```

Resulting Stack Structure



Pre-return Stack Structure



Some Observations

- **Stack frames provide each function call with private storage**
  - Saved registers & local variables
  - Saved return pointer

- **Register saving conventions prevent one function call from corrupting another’s data**
  - Unless the C code explicitly does so (e.g., buggy code)

- **Stack discipline follows call / return pattern**
  - If P calls Q, then Q returns before P
  - Last-In, First-Out

Reading

- **Book section 3.7**

- **https://en.wikipedia.org/wiki/X86_calling_convention**
  - What happens to non-integer arguments (floats, etc.)?
  - Stack alignment requirements
  - Allocating extra space for additional features