# Computer Systems Organization

**Topic 3 Contd.**

Based on chapter 3 from Computer Systems by Randal E. Bryant and David R. O'Hallaron

# Mechanisms in Procedures

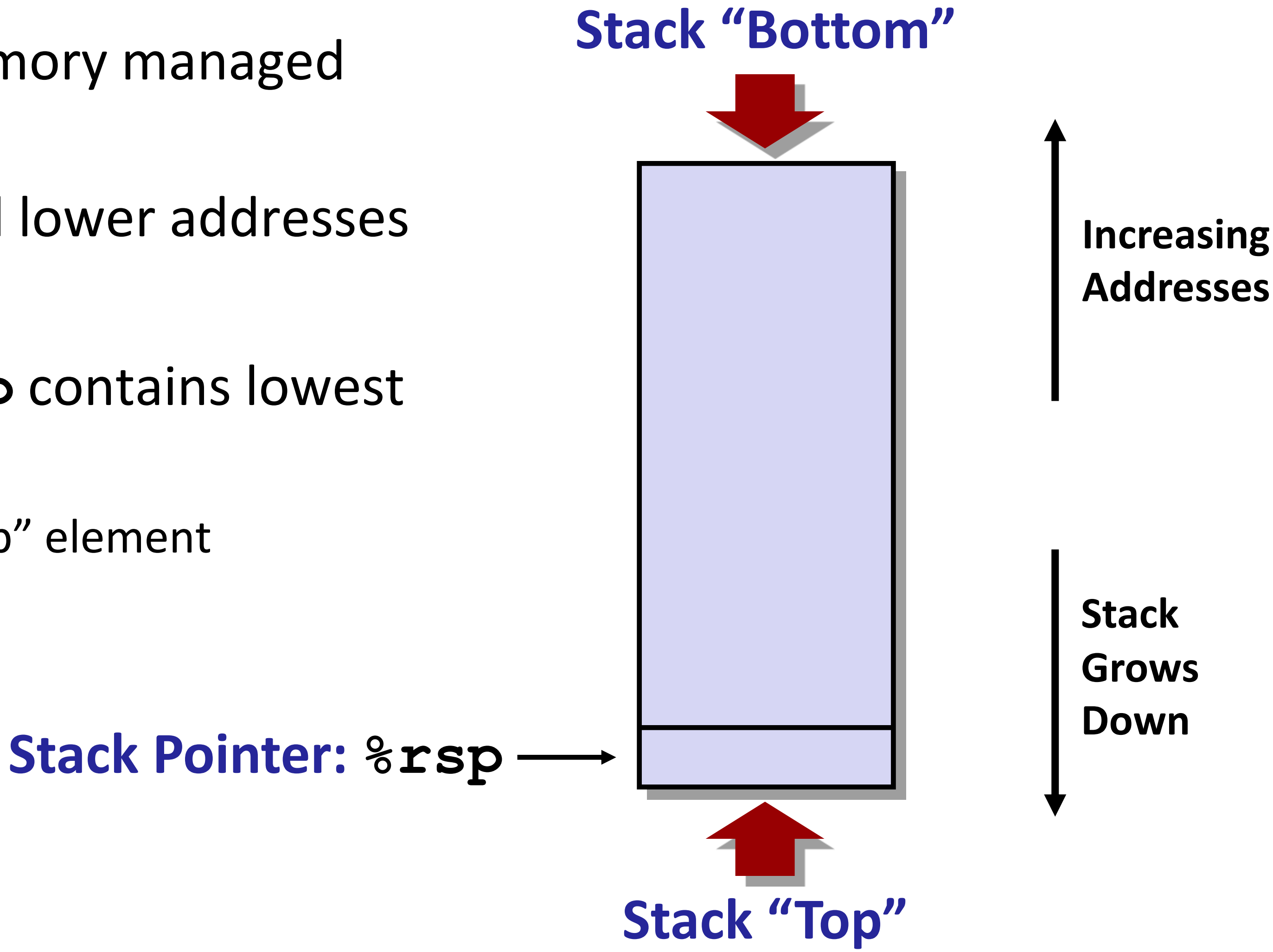
- **Passing control**
  - To beginning of procedure code
  - Back to return point
- **Passing data**
  - Procedure arguments
  - Return value
- **Memory management**
  - Allocate during procedure execution
  - Deallocate upon return
- Mechanisms all implemented with machine instructions

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

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

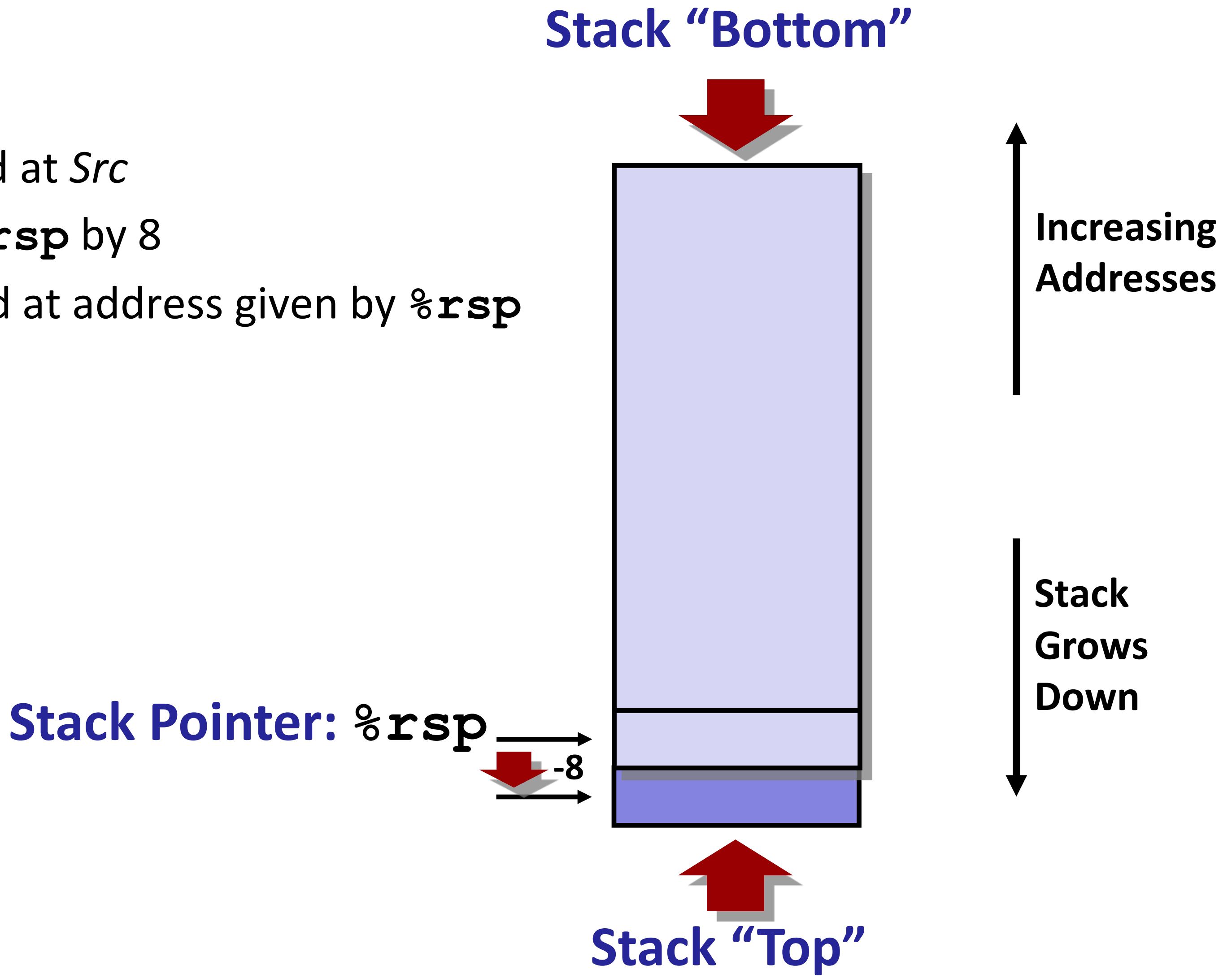
# x86-64 Stack

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



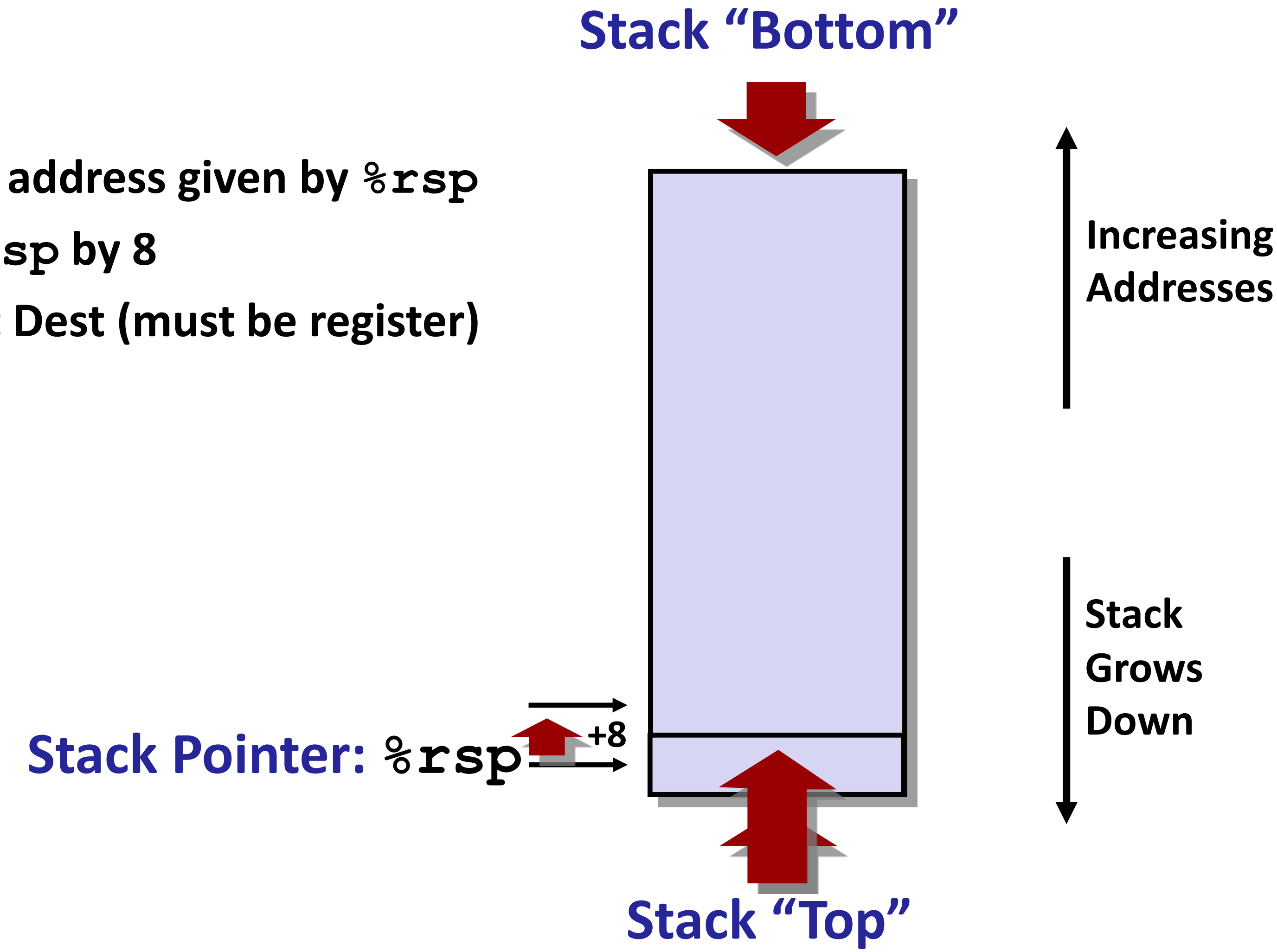
# x86-64 Stack: Push

- **`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)



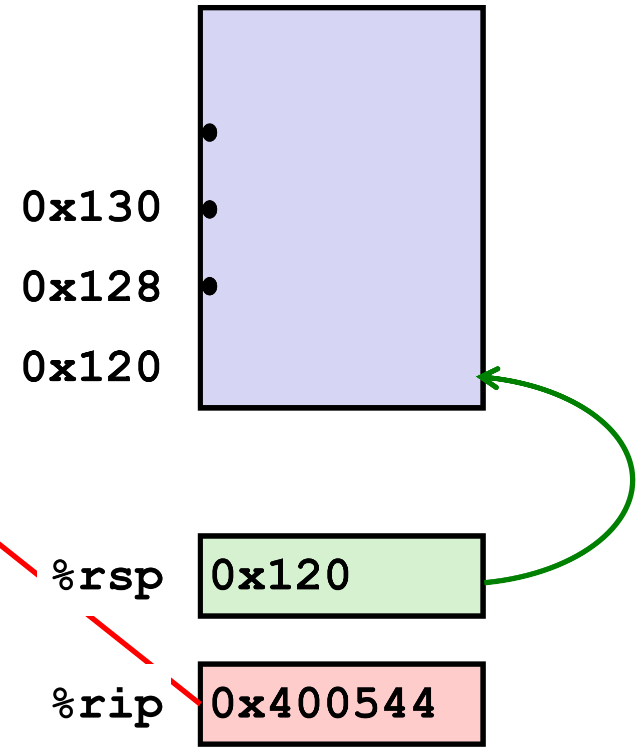
# Procedure Control Flow

- 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)
  •
  •
```

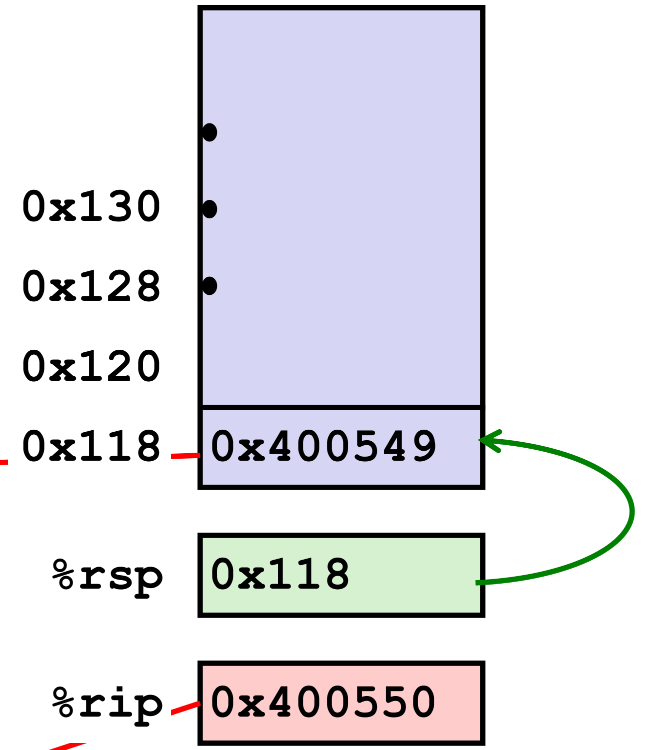
```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  retq
```




%rsp stack pointer
%rip program counter

# Control Flow Example #2

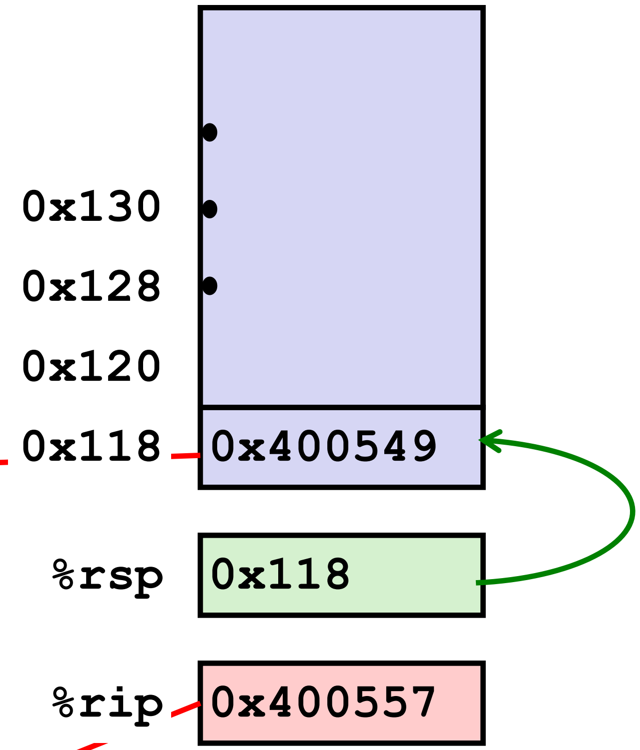
```
0000000000400540 <multstore>:
  •
  •
  400544: callq  400550 <mult2>
  400549: mov    %rax,(%rbx)
  •
  •
```




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

# Control Flow Example #3

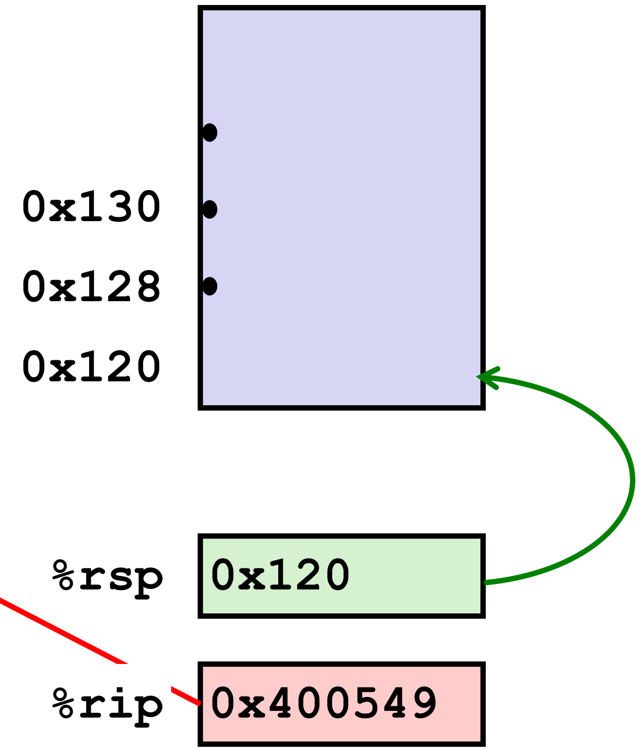
```
0000000000400540 <multstore>:
  •
  •
  400544: callq  400550 <mult2>
  400549: mov    %rax,(%rbx)
  •
  •
```




```
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
```

# Procedure Data Flow

## Registers

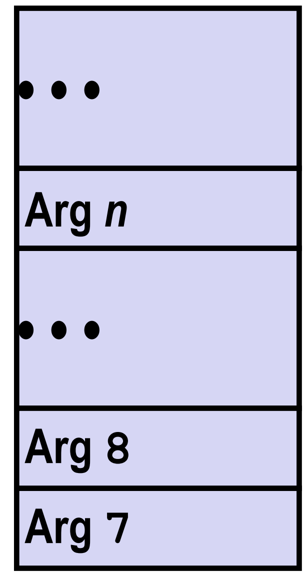
- **First 6 arguments**

| |
|---|
| `%rdi` |
| `%rsi` |
| `%rdx` |
| `%rcx` |
| `%r8` |
| `%r9` |

- **Return value**

| |
|---|
| `%rax` |

## Stack




- **Only allocate stack space when needed**

Registers **`%rbx`**, **`%rbp`** and **`%r12-r15`** are **`callee-save registers`**, meaning that they are saved across function calls.

# Data Flow Examples (Disassembled code)

```
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
```

# Stack-Based Languages

- **Languages that support recursion**
  - e.g., C, Pascal, 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 based model**
  - 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

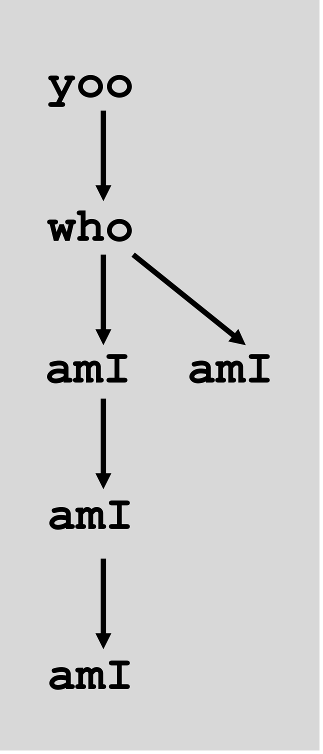
```
yoo(…)
{
  •
  •
  who();
  •
  •
}
```

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

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

Procedure `amI()` is recursive

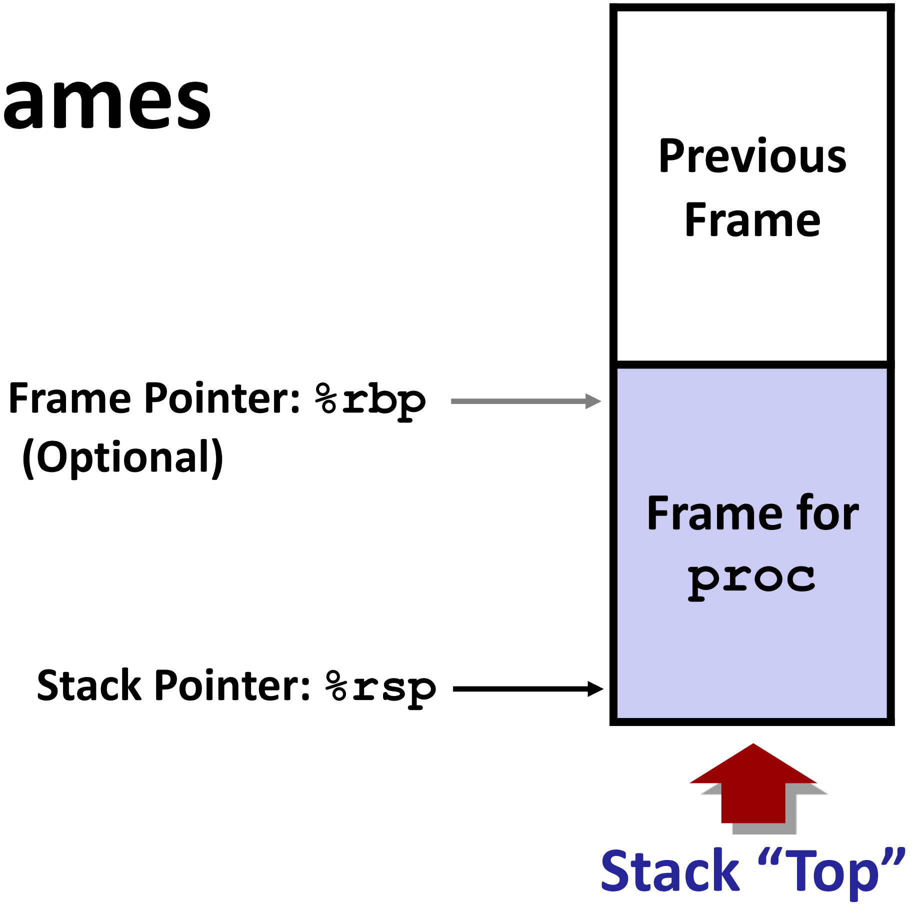
**Example Call Chain**



# Stack Frames

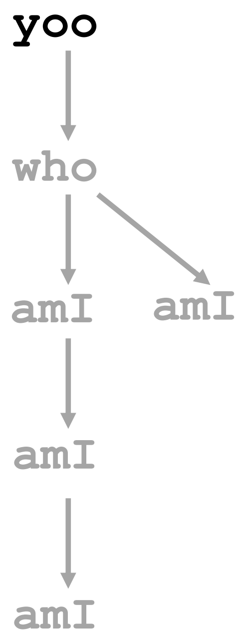
- **Contents**
  - Return information
  - Local storage (if needed)
  - Temporary space (if needed)

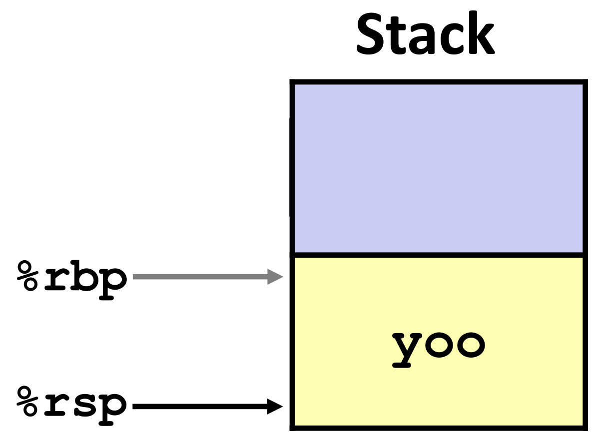
- **Management**
  - Space allocated when enter procedure
    - “Set-up” code
    - Includes push by **`call`** instruction
  - Deallocated when return
    - “Finish” code
    - Includes pop by **`ret`** instruction



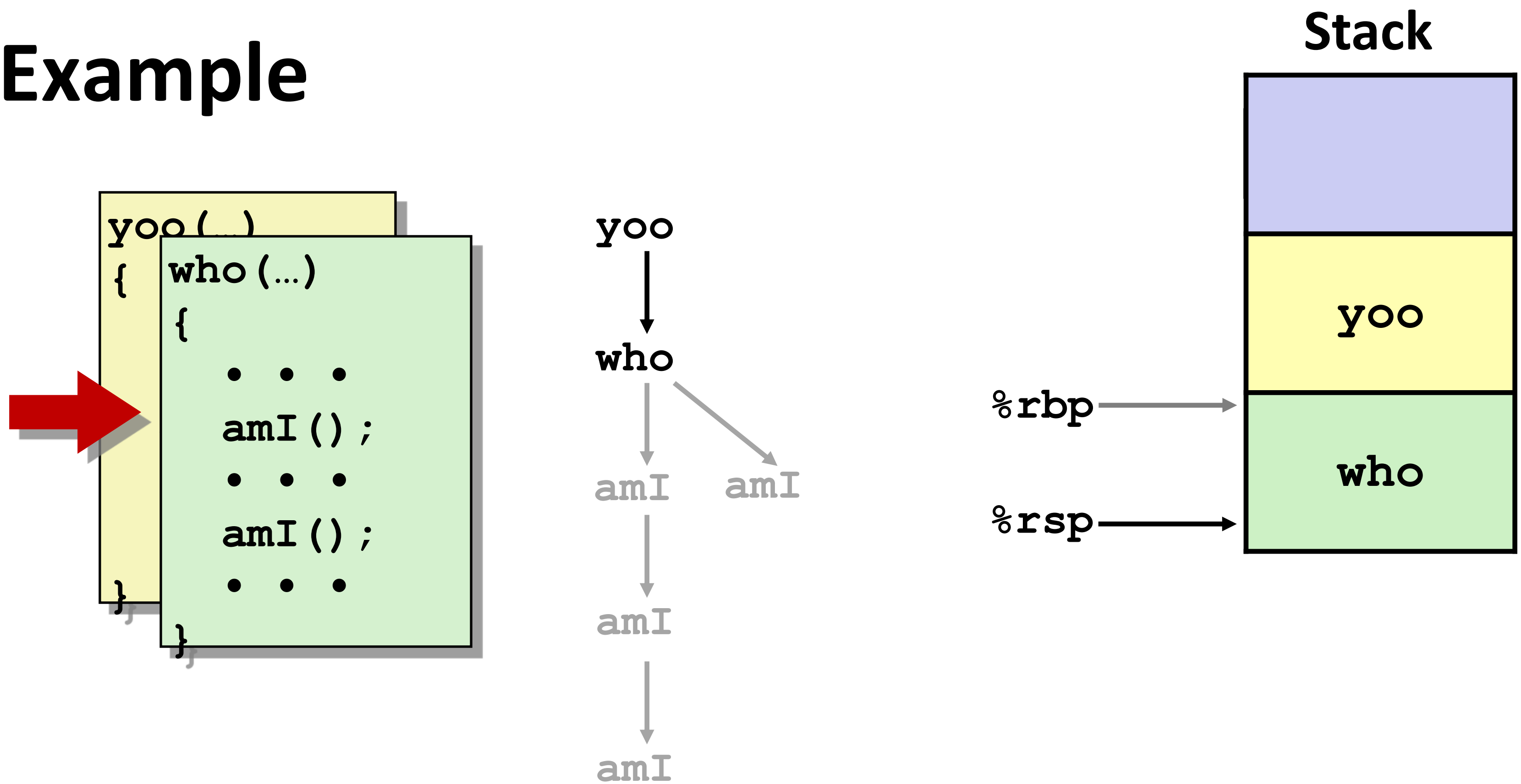
# Example

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

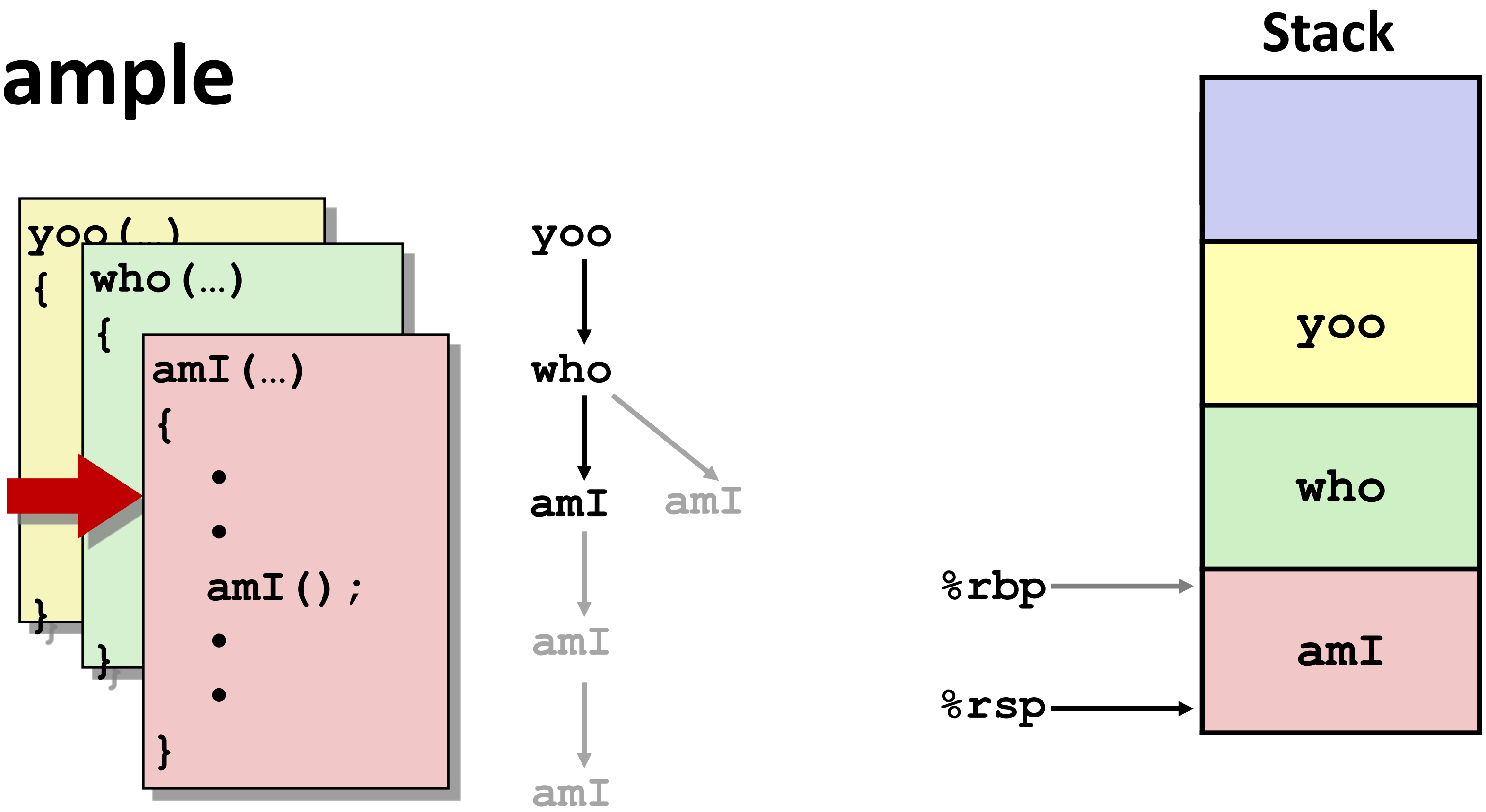


# Example

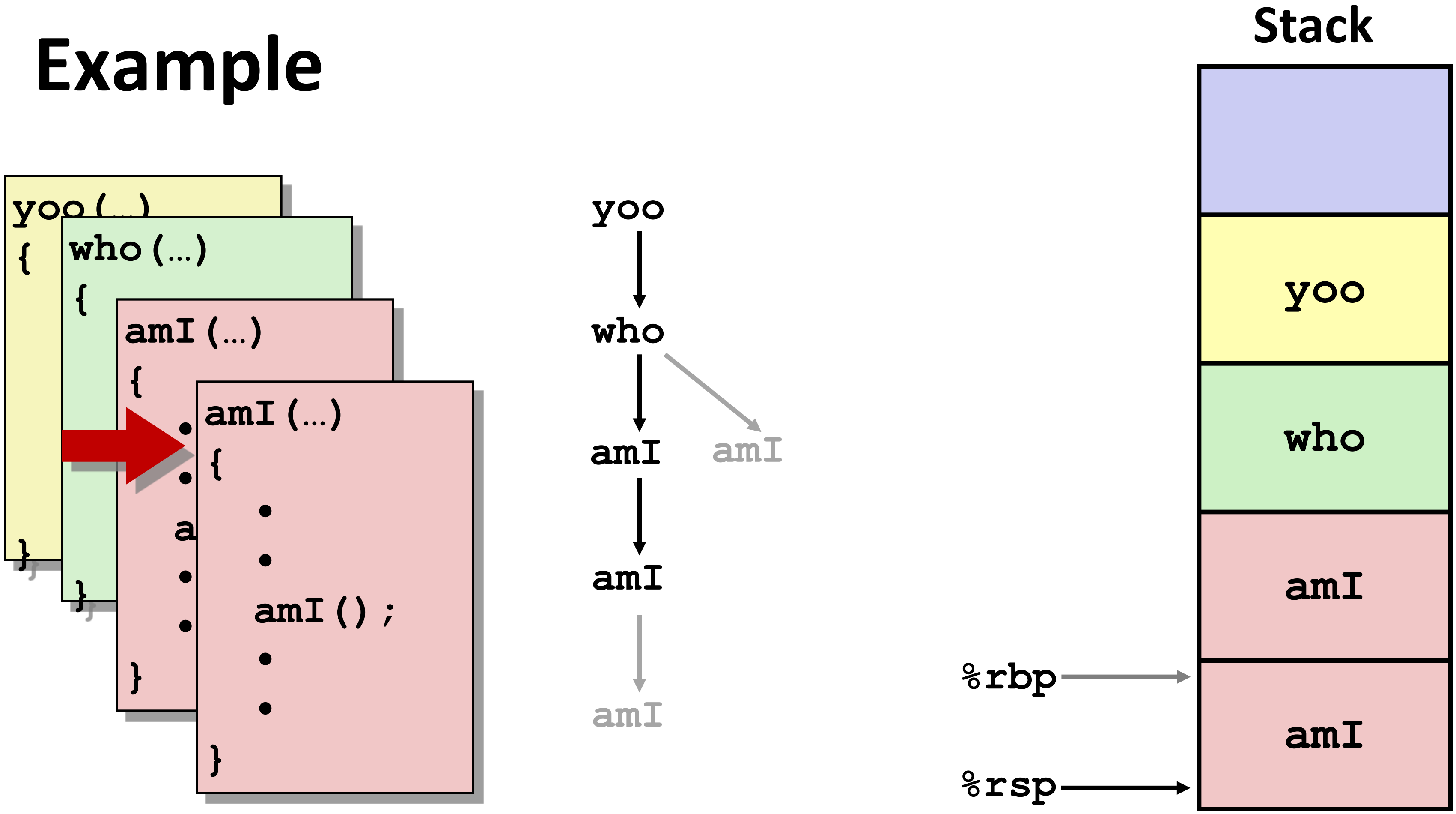


# Example

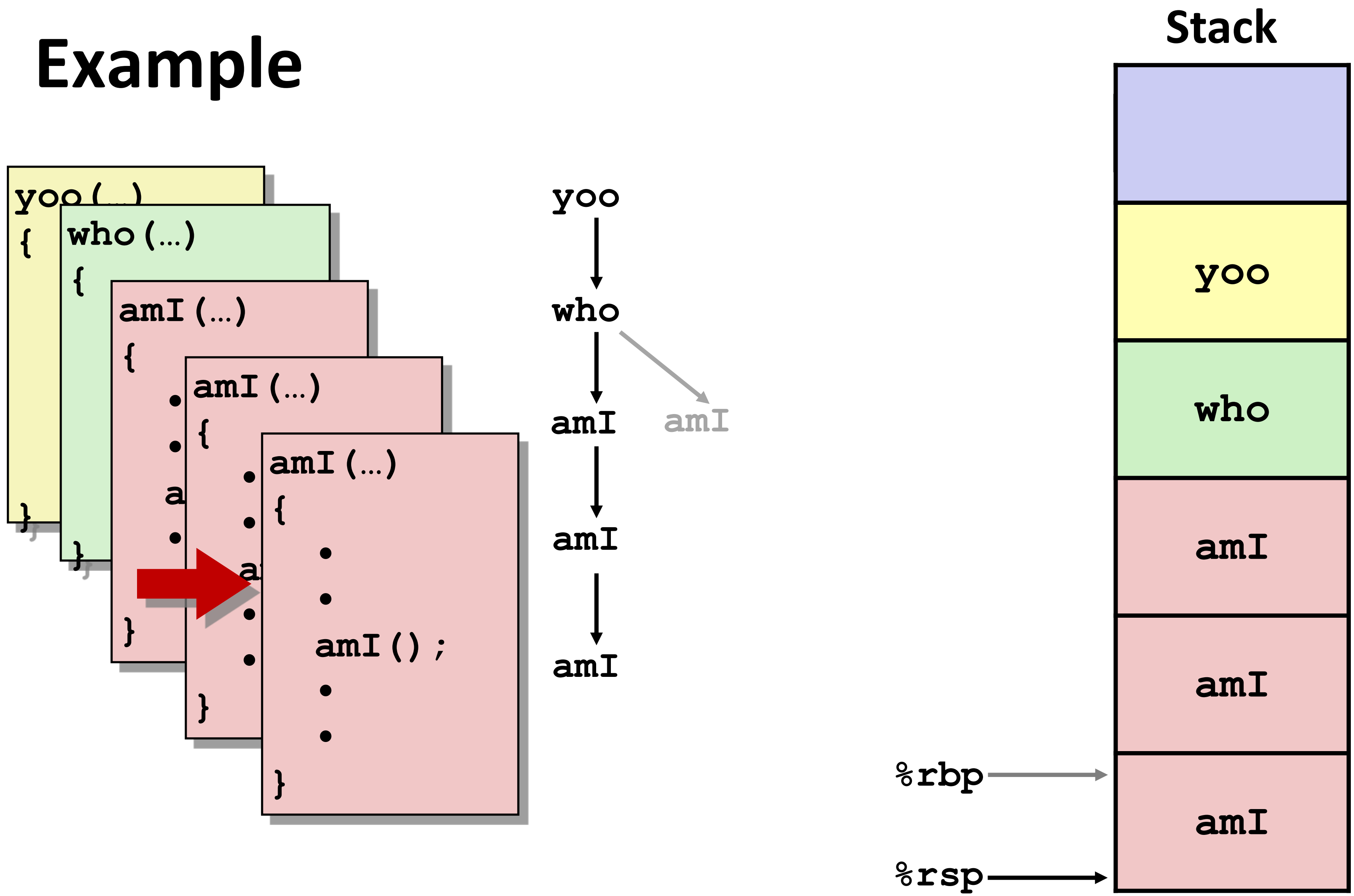


yoo(…)
{
}
who(…)
{
}
amI(…)
{
•
•
amI();
•
•
}
yoo
who
amI
amI
amI
amI
Stack
yoo
who
amI
%rbp
%rsp

# Example

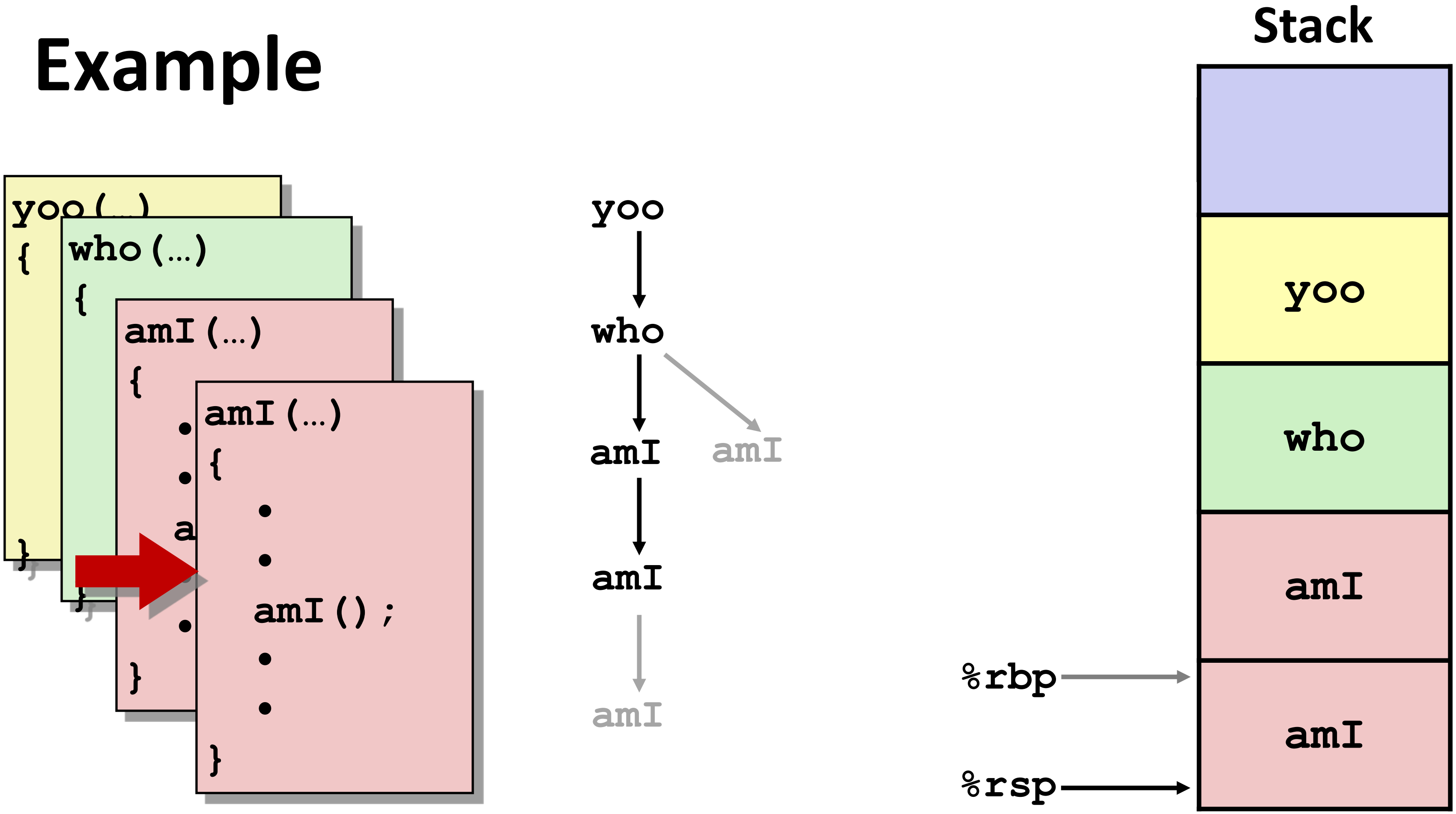

yoo(…)
{
}
who(…)
{
}
amI(…)
{
•
•
a
•
•
}
amI(…)
{
•
•
amI();
•
•
}
yoo
who
amI
amI
amI
amI
Stack
yoo
who
amI
amI
%rbp
%rsp

# Example



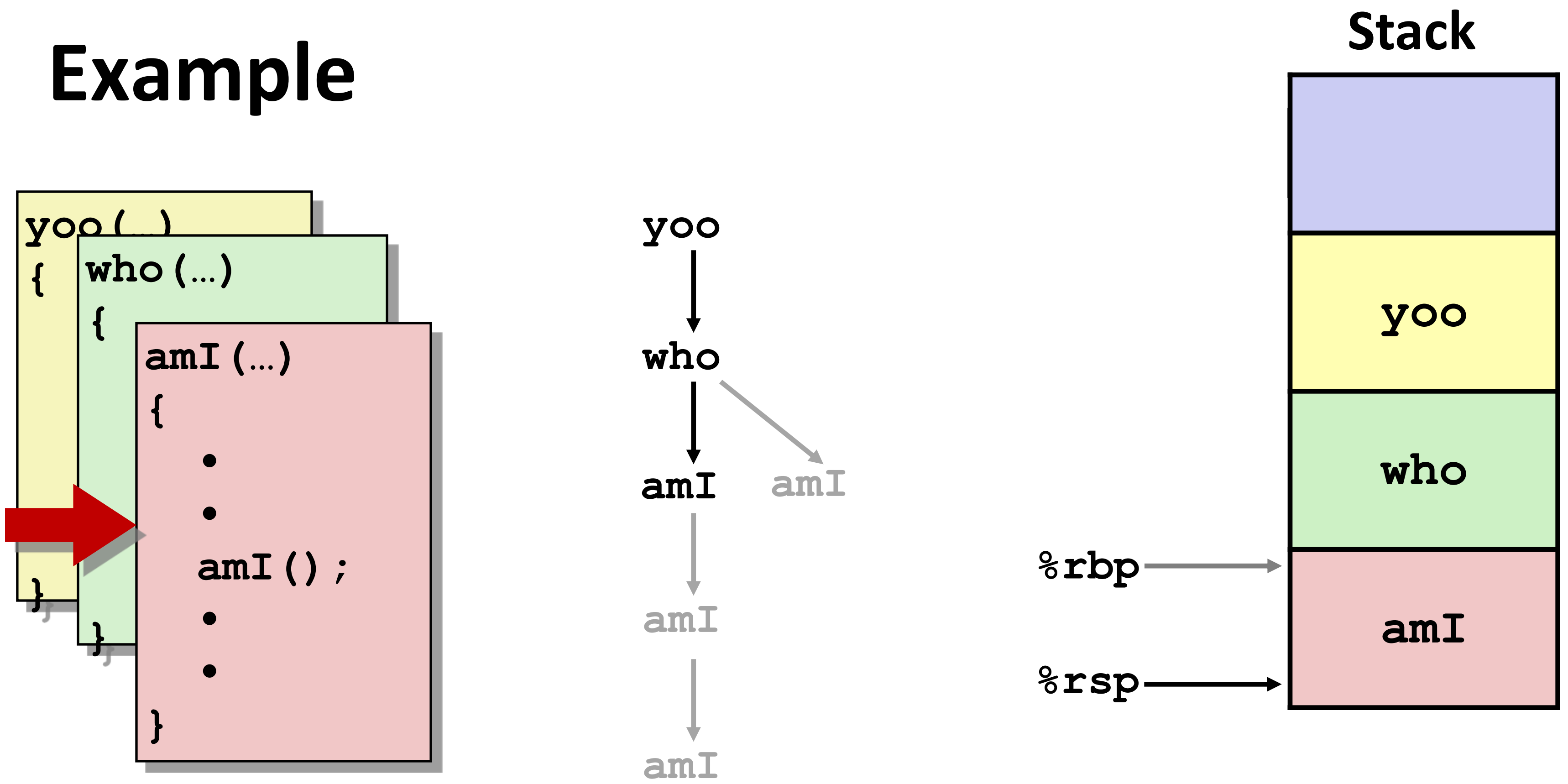
yoo(…)
{
}
who(…)
{
}
amI(…)
{
}
amI(…)
{
}
amI(…)
{
•
•
amI();
•
•
}
yoo
who
amI
amI
amI
amI
Stack
yoo
who
amI
amI
amI
%rbp
%rsp

# Example



yoo(…)
{
}
who(…)
{
}
amI(…)
{
}
amI(…)
{
amI();
}
yoo
who
amI
amI
amI
amI
Stack
yoo
who
amI
amI
%rbp
%rsp

# Example

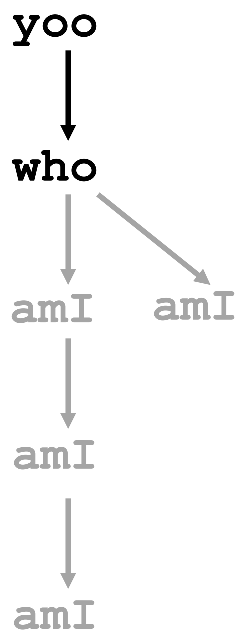


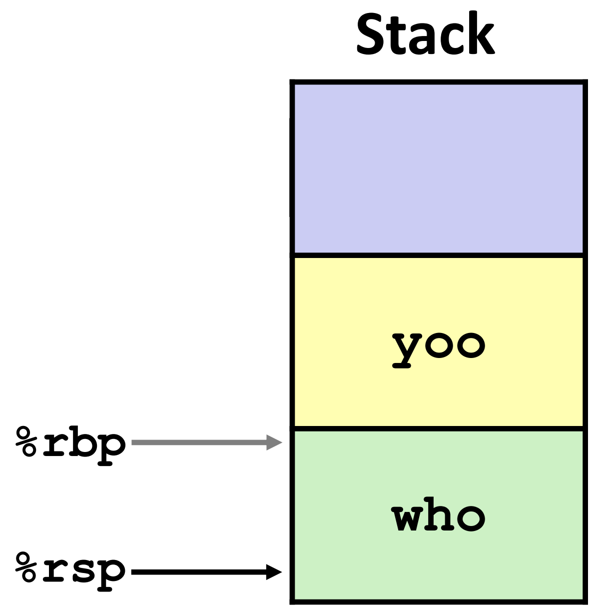
yoo(…)
{
}
who(…)
{
}
amI(…)
{
•
•
amI();
•
•
}
yoo
who
amI
amI
amI
amI
Stack
yoo
who
amI
%rbp
%rsp

# Example

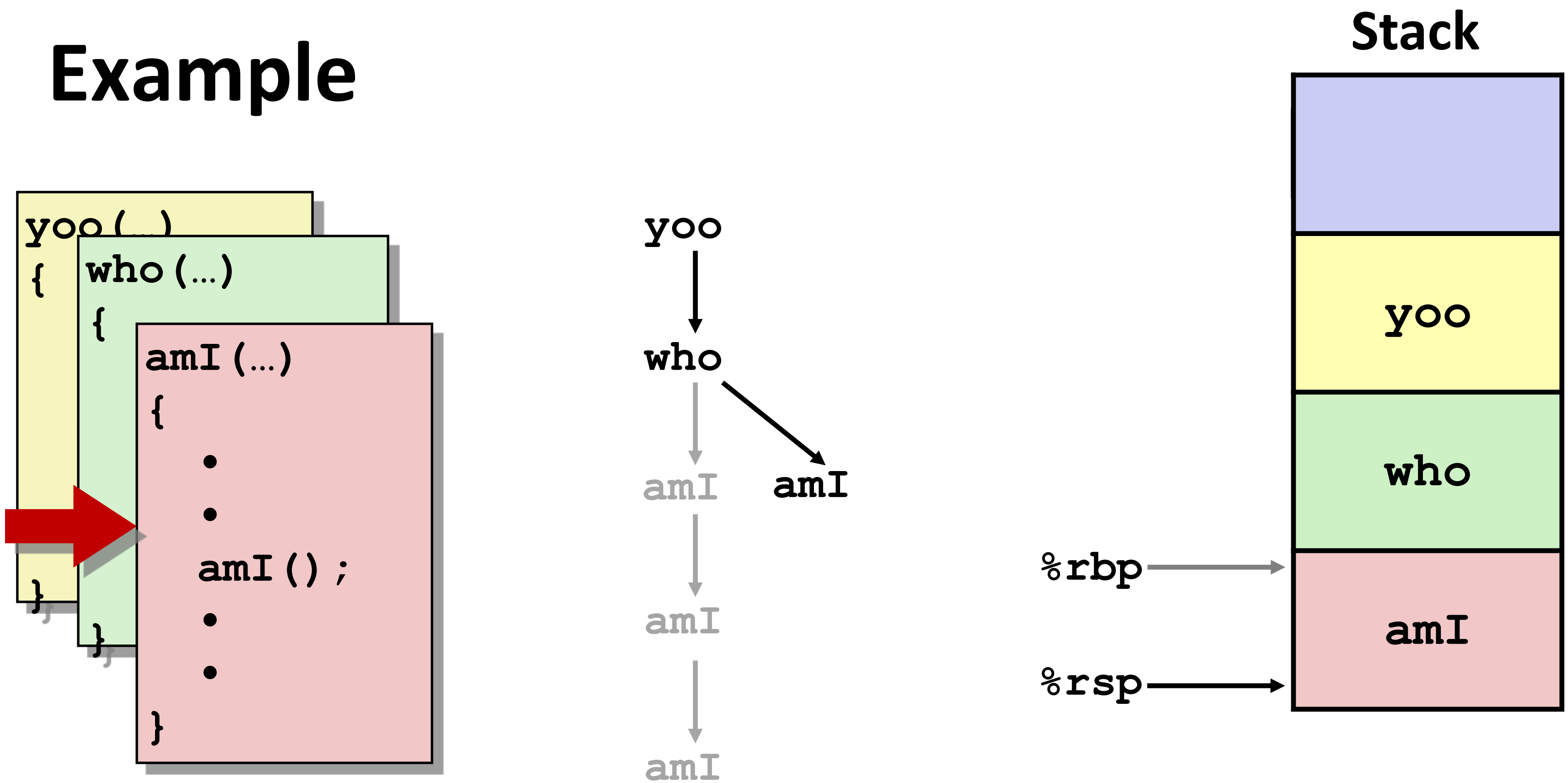
```
yoo(…)
{
}
```

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



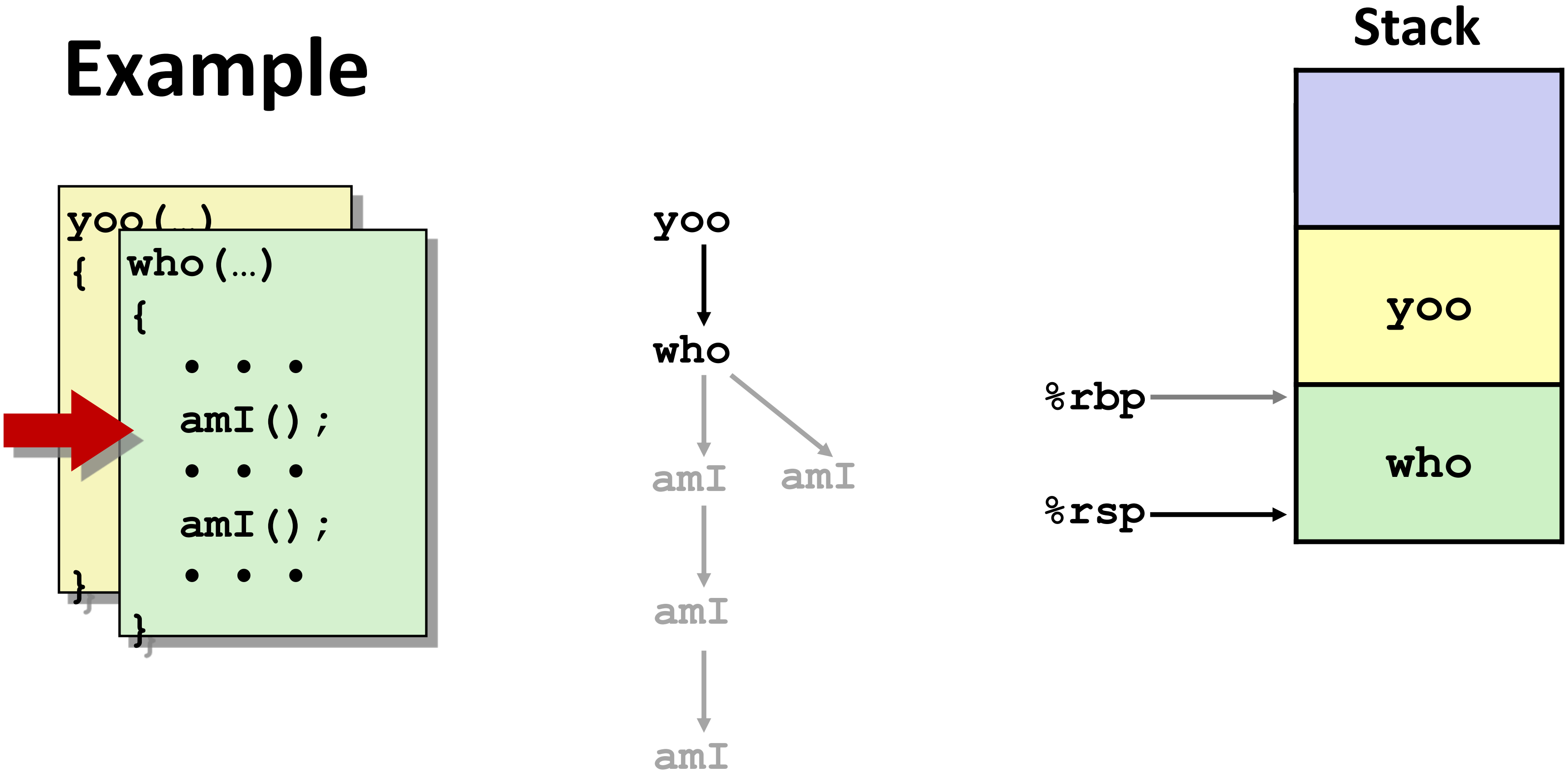



# Example



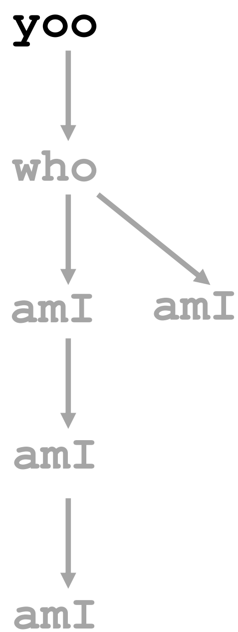
yoo(…)
{
who(…)
{
amI(…)
{
•
•
amI();
•
•
}
}
}
yoo
who
amI
amI
amI
amI
Stack
yoo
who
amI
%rbp
%rsp

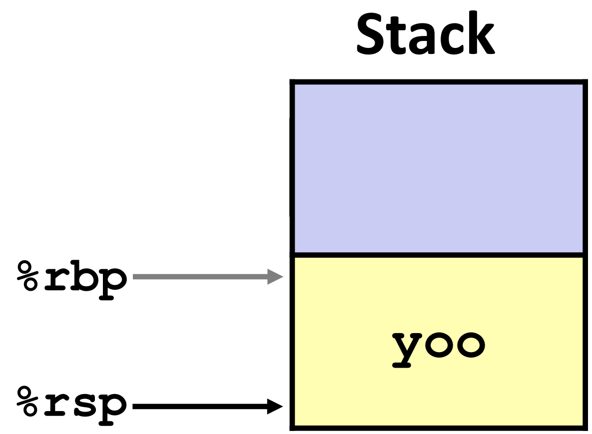
# Example



# Example

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

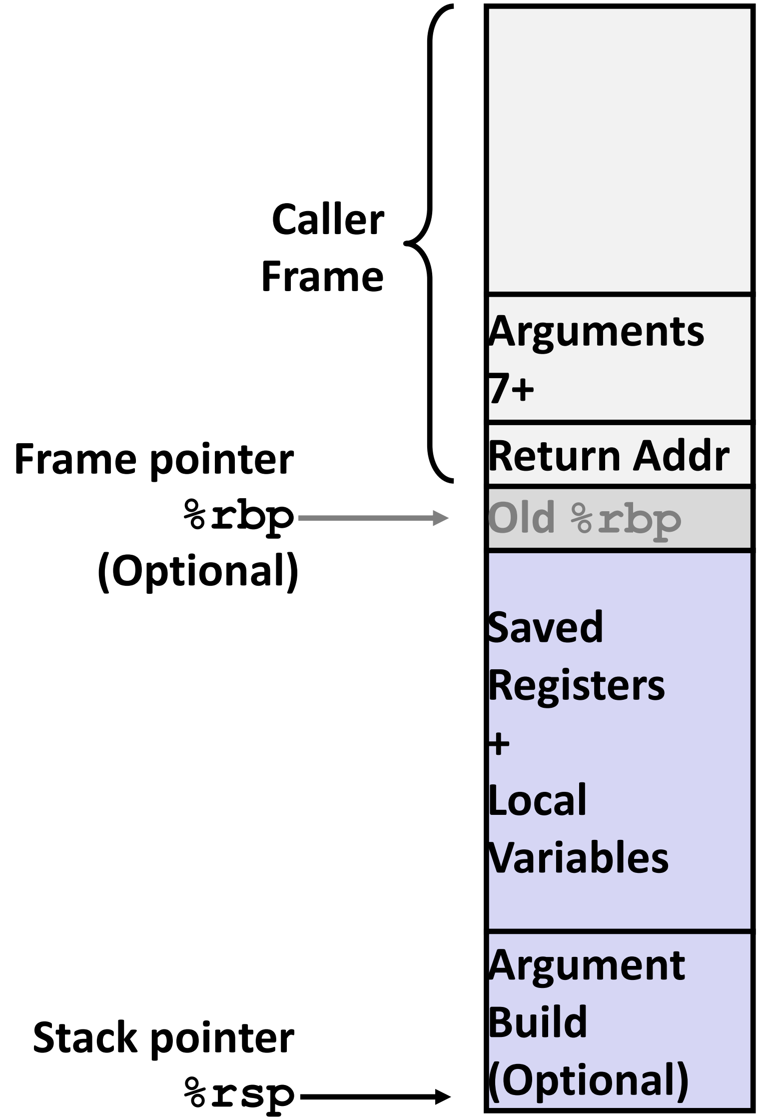


# x86-64 Stack Frame

- **Current Stack Frame ("Top" to Bottom)**
  - "Argument build:"
    Parameters for function about to call
  - 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;
}
```

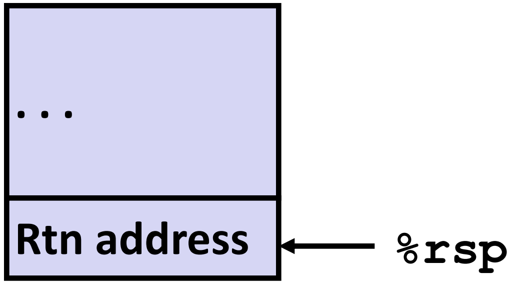
```
incr:
  movq    (%rdi), %rax
  addq    %rax, %rsi
  movq    %rsi, (%rdi)
  ret
```

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

# Example: Calling `incr` #1

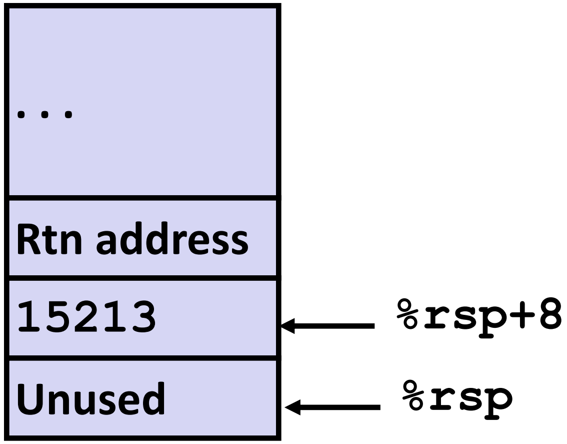
```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

**Initial Stack Structure**




```
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**



# 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**

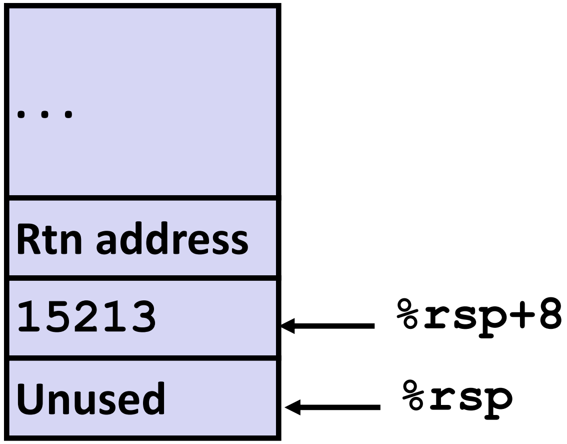
Registers **`%rbx`**, **`%rbp`** and **`%r12-r15`** are **`callee-save registers`**, meaning that they are saved across function calls.

# 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**




| 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**

| Stack | |
|---|---|
| ... | |
| Rtn address | |
| 18213 | ← `%rsp+8` |
| Unused | ← `%rsp` |

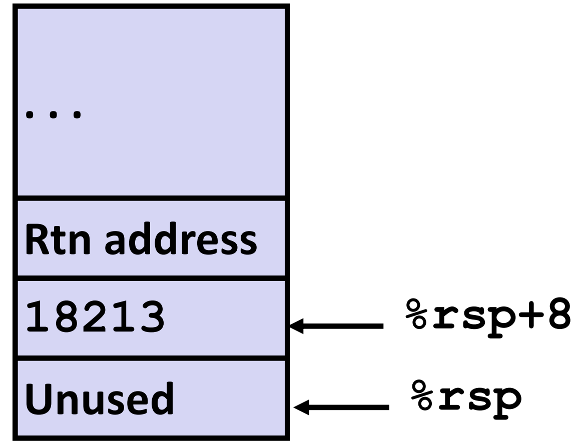
| Register | Use(s) |
|---|---|
| `%rdi` | `&v1` |
| `%rsi` | `3000` |

# Example: Calling `incr` #4

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

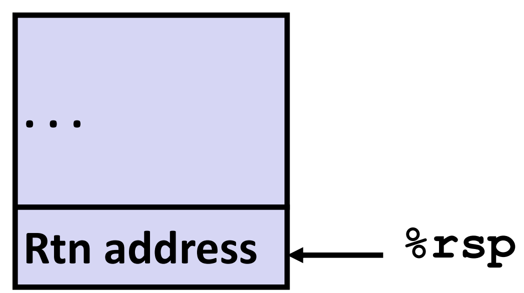
```
call_incr:
  subq    $16, %rsp
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

### Stack Structure




| Register | Use(s) |
|---|---|
| `%rax` | Return value |

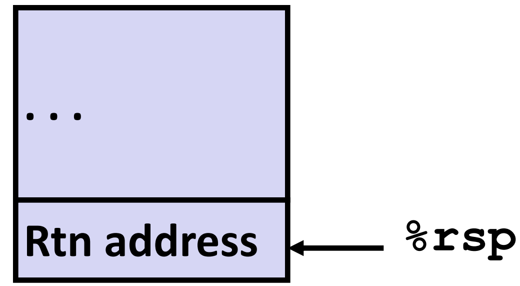
### Updated Stack Structure



# Example: Calling `incr` #5

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

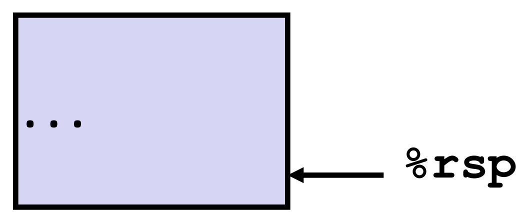
### Updated Stack Structure




```
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
```

| Register | Use(s) |
|---|---|
| `%rax` | Return value |

### Final Stack Structure



# Figure 3.31 Example of procedure definition and call

•

# Procedure Data Flow

## Registers

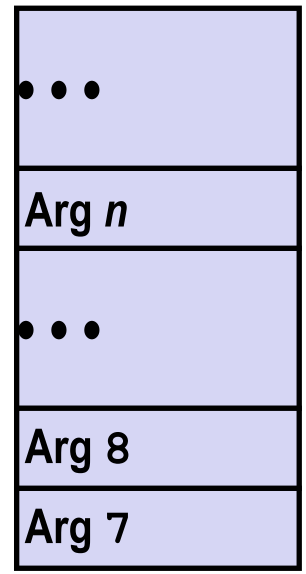
- **First 6 arguments**

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

- **Return value**

`%rax`

## Stack




- **Only allocate stack space when needed**

Registers **`%rbx`**, **`%rbp`** and **`%r12-r15`** are **`callee-save registers`**, meaning that they are saved across function calls.

# Register Saving Conventions

- Set of registers act as a single resource shared by all procedures
- When a **caller** procedure calls another procedure (called **callee**), the callee does not overwrite some register values
- X86-64 adopts a uniform set of conventions for register usage that must be respected by all procedures
- When a procedure **P** calls procedure **Q**, **Q** must preserve the values of **callee-saved** registers.
  - This is so they have same values when Q returns to P as they did when Q was called.
  - Q preserves a register value by not changing it at all or by pushing the original value on the stack, altering it and then popping the old value from the stack before returning.
  - Pushing of register values has the effect of creating the portion of stack frame labeled "Saved registers"

# Register Saving Conventions

- All other registers except for the stack pointer %rsp are classified as **caller-saved** registers
  - Can be modified by any function
  - The calling function P has to first save the data before it makes a call to another function Q

# 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
```

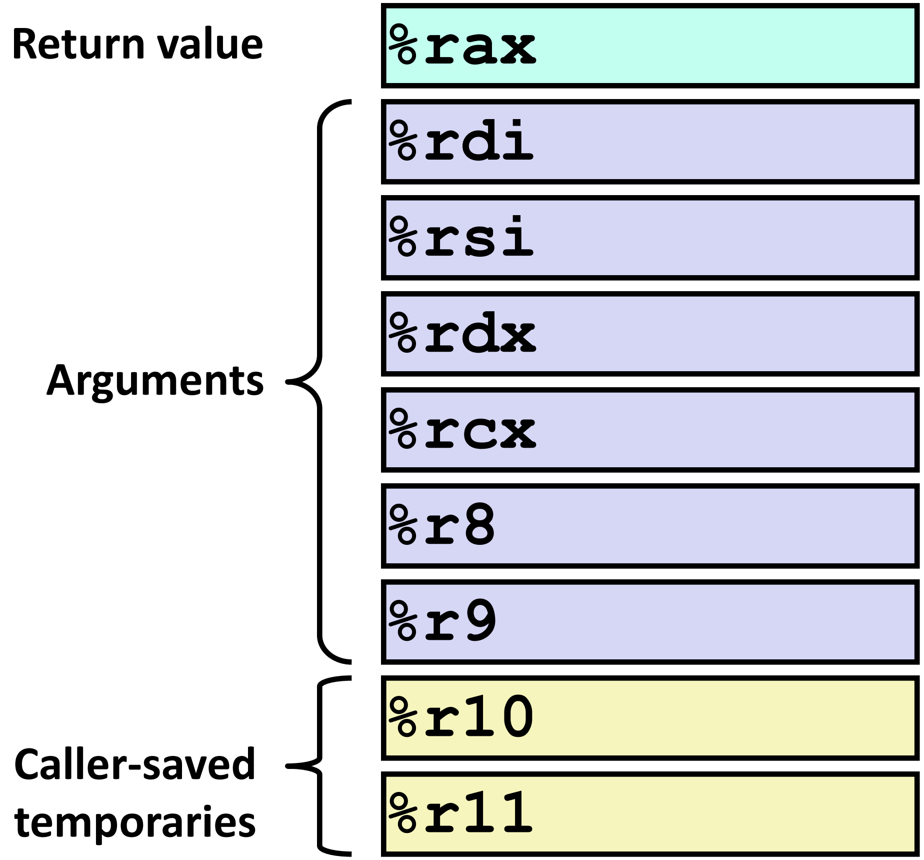
  - Contents of register `%rdx` overwritten by `who`
  - This could be trouble - need some coordination

# 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

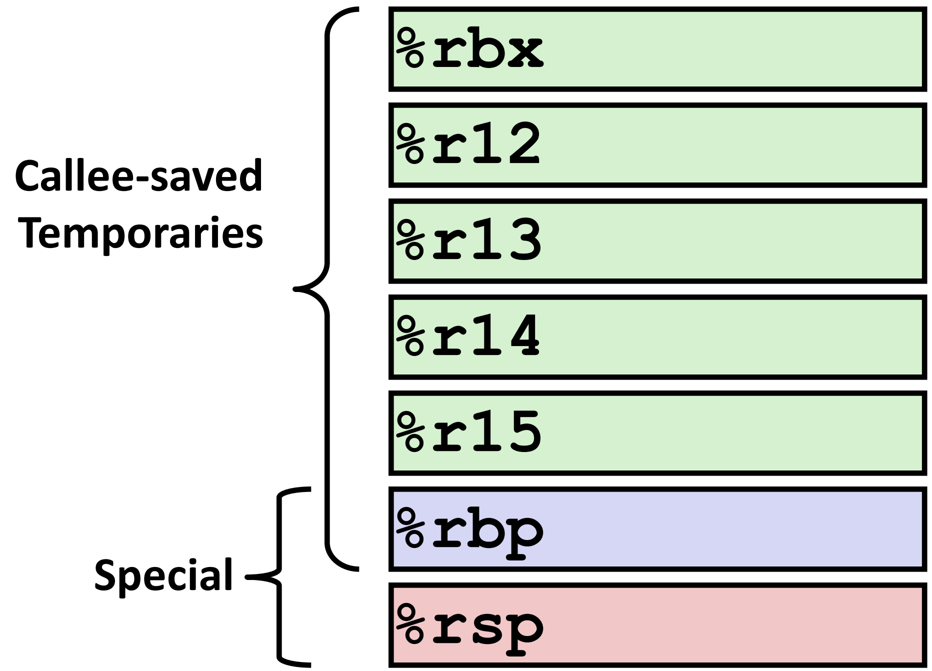
# x86-64 Register Usage

- `%rax`
  - Return value
  - 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 Register Usage

- **`%rbx`, `%r12`, `%r13`, `%r14`, `%r15`**
  - 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

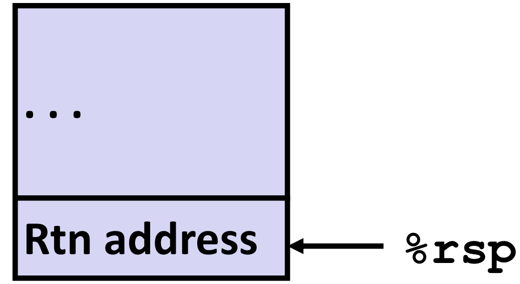


# Callee-Saved Example #1

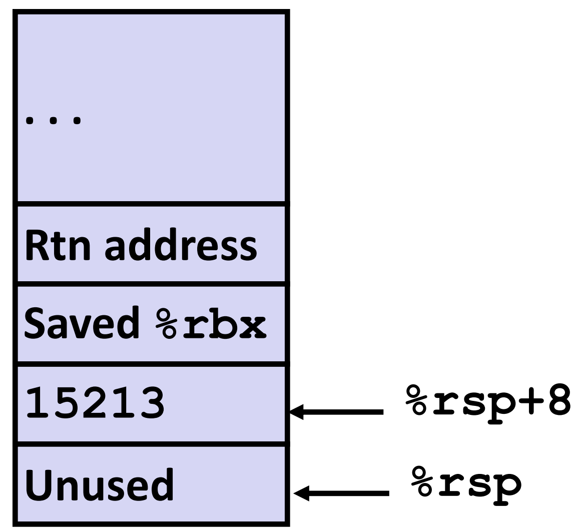
```
long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```

```
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

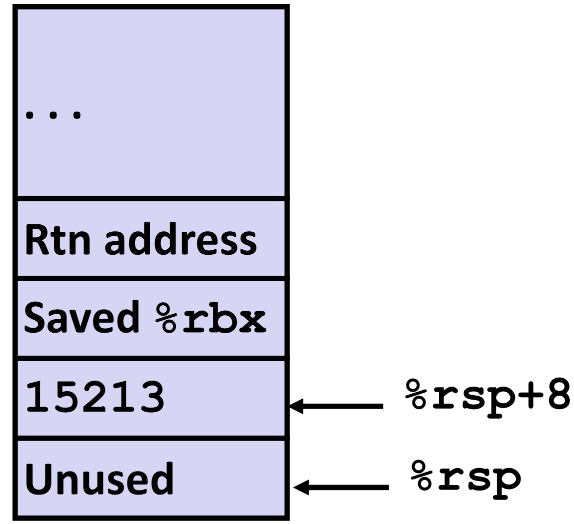


# Callee-Saved Example #2

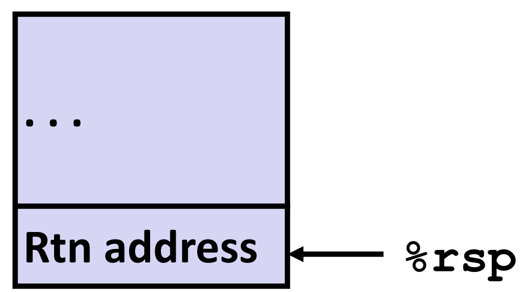
```
long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```

```
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**



# In summary

- **Call** instruction pushes the return address onto the stack and transfers control to a procedure.
- **Ret** instruction pops the return address off the stack and returns control to that location.

# Recursive Function

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1)
           + pcount_r(x >> 1);
}
```

```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  andl    $1, %ebx
  shrq    $1, %rdi
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep; ret
```

# Recursive Function Terminal Case

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1)
           + pcount_r(x >> 1);
}
```

```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  andl    $1, %ebx
  shrq    $1, %rdi
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep; ret
```

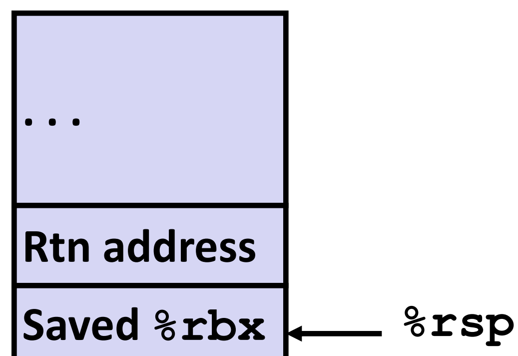
| Register | Use(s) | Type |
|---|---|---|
| `%rdi` | `x` | Argument |
| `%rax` | Return value | Return value |

# Recursive Function Register Save

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1)
           + pcount_r(x >> 1);
}
```

```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  andl    $1, %ebx
  shrq    $1, %rdi
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep; ret
```

| Register | Use(s) | Type |
|---|---|---|
| %rdi | x | Argument |



# Recursive Function Call Setup

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1)
           + pcount_r(x >> 1);
}
```

```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  andl    $1, %ebx
  shrq    $1, %rdi
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep; ret
```

| Register | Use(s) | Type |
|---|---|---|
| `%rdi` | `x >> 1` | Rec. argument |
| `%rbx` | `x & 1` | Callee-saved |

# Recursive Function Call

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1)
           + pcount_r(x >> 1);
}
```

```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  andl    $1, %ebx
  shrq    $1, %rdi
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep; ret
```

| Register | Use(s) | Type |
|---|---|---|
| `%rbx` | `x & 1` | Callee-saved |
| `%rax` | Recursive call return value | |

# Recursive Function Result

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1)
           + pcount_r(x >> 1);
}
```

```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  andl    $1, %ebx
  shrq    $1, %rdi
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep; ret
```

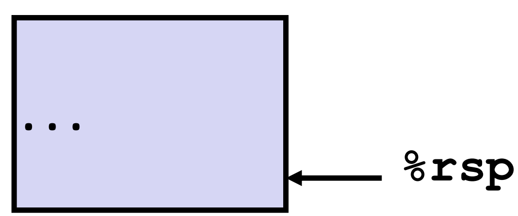
| Register | Use(s) | Type |
|---|---|---|
| `%rbx` | `x & 1` | Callee-saved |
| `%rax` | Return value | |

# Recursive Function Completion

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1)
           + pcount_r(x >> 1);
}
```

```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  andl    $1, %ebx
  shrq    $1, %rdi
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep; ret
```

| Register | Use(s) | Type |
|---|---|---|
| `%rax` | Return value | Return value |



# Observations About Recursion

- Handled Without Special Consideration
  - Stack frames mean that each function call has private storage
    - Saved registers & local variables
    - Saved return pointer
  - Register saving conventions prevent one function call from corrupting another's data
  - Stack discipline follows call / return pattern
    - If P calls Q, then Q returns before P
    - Last-In, First-Out
- Also works for mutual recursion
  - P calls Q; Q calls P