

Machine-Level Programming III: Procedures

CSE251: System Programming
7th Lecture, Mar. 20, 2019

Instructors:

Hyungon Moon

Mechanisms in Procedures

■ Passing control

- To beginning of procedure code
- Back to return point

■ Passing data

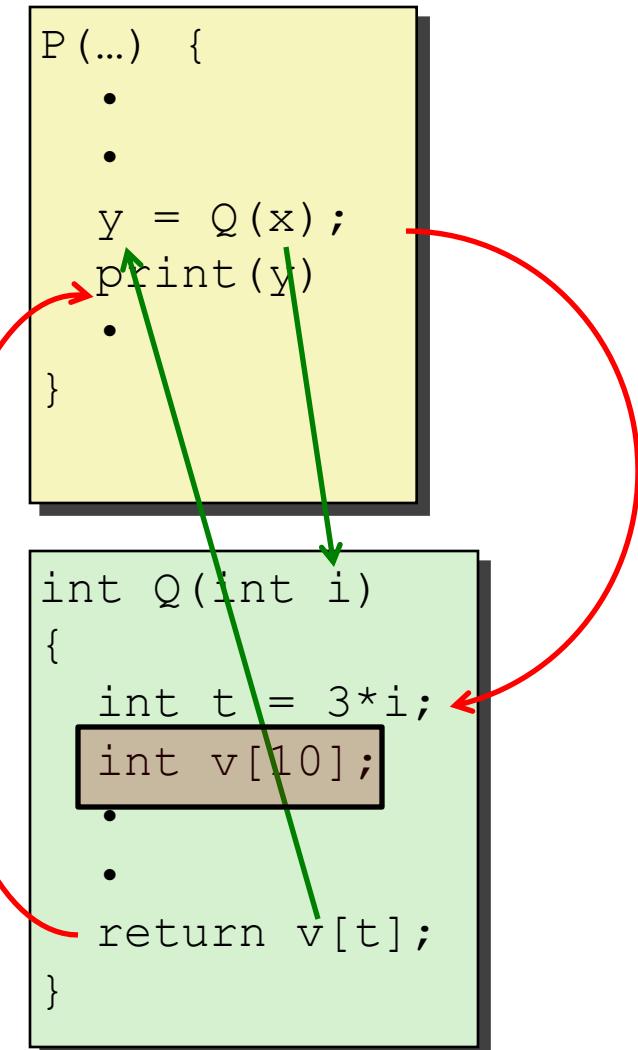
- Procedure arguments
- Return value

■ 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



Today

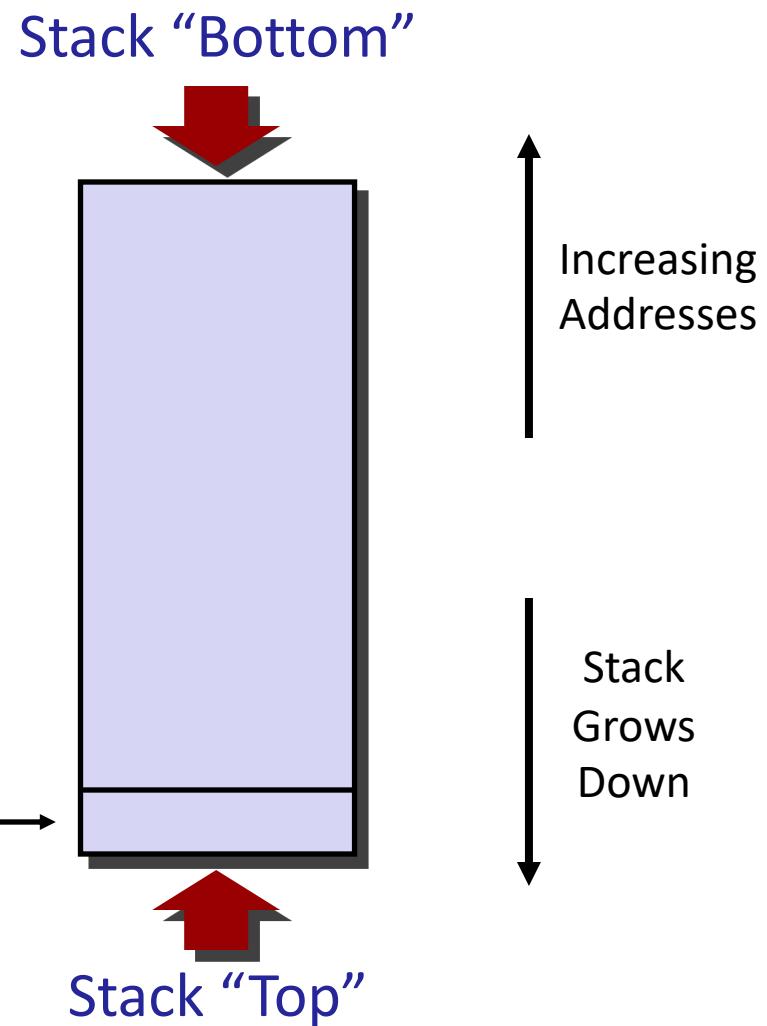
■ Procedures

- Stack Structure
- Calling Conventions
 - Passing control
 - Passing data
 - Managing local data
- Illustration of Recursion

x86-64 Stack

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

Stack Pointer: `%rsp` →



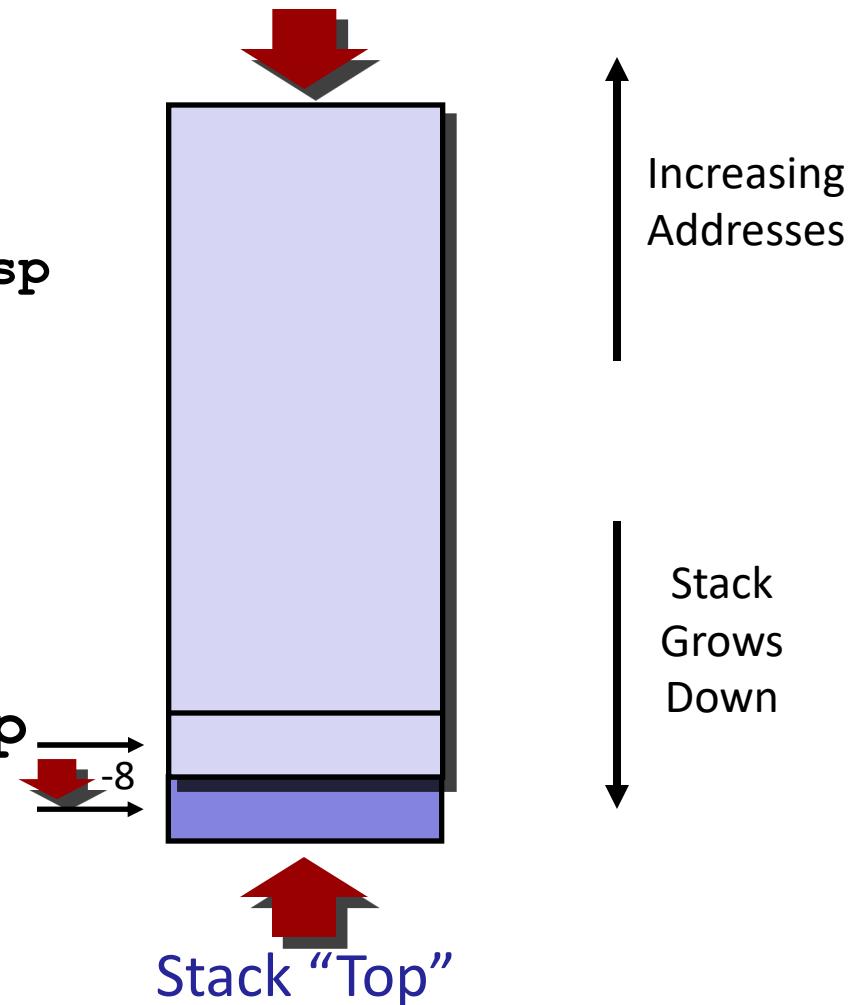
x86-64 Stack: Push

■ **pushq Src**

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

Stack Pointer: `%rsp`

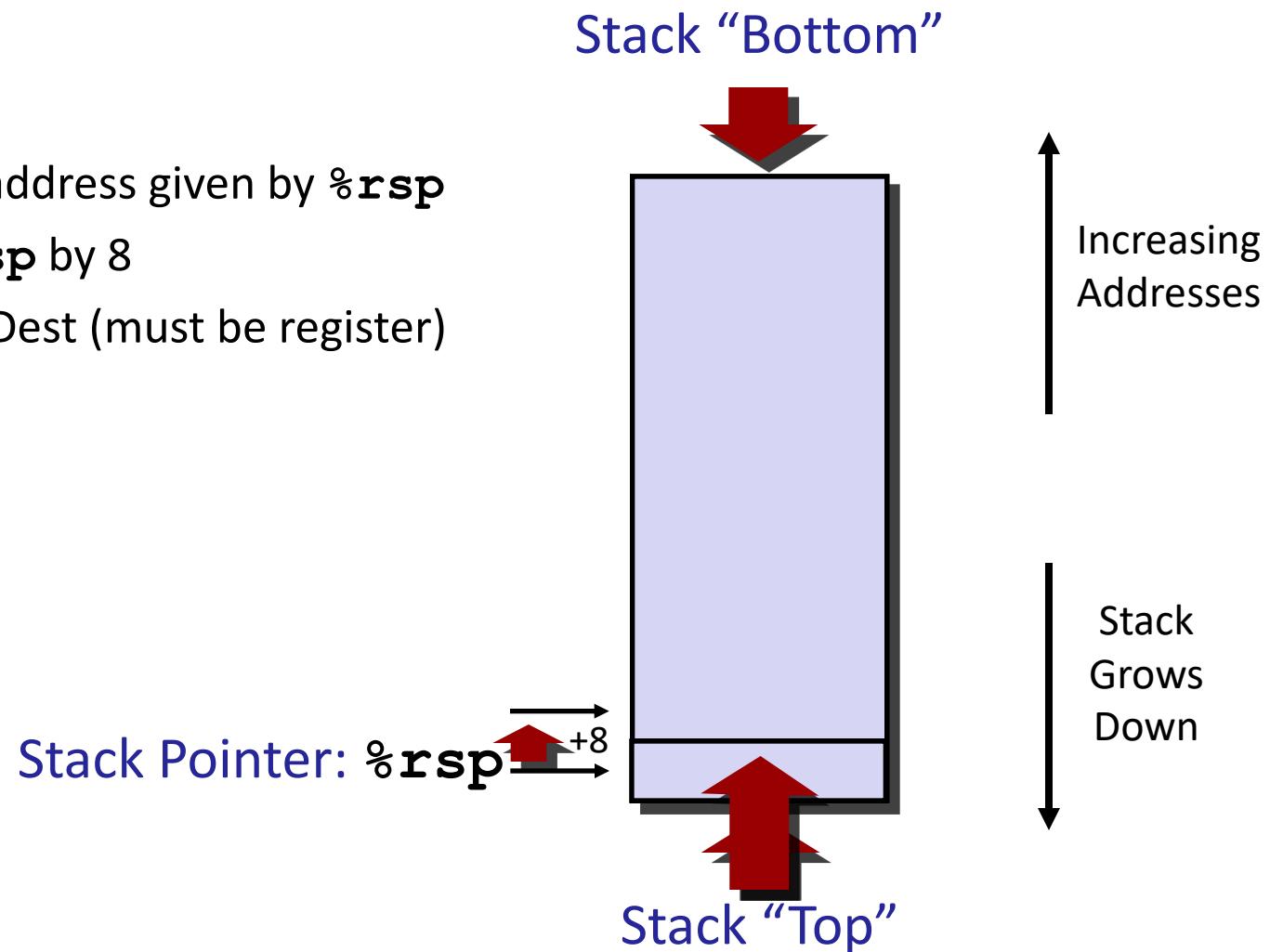
Stack “Bottom”



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
- Illustration of Recursion

Code Examples

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

```
000000000400540 <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;  
}
```

```
000000000400550 <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

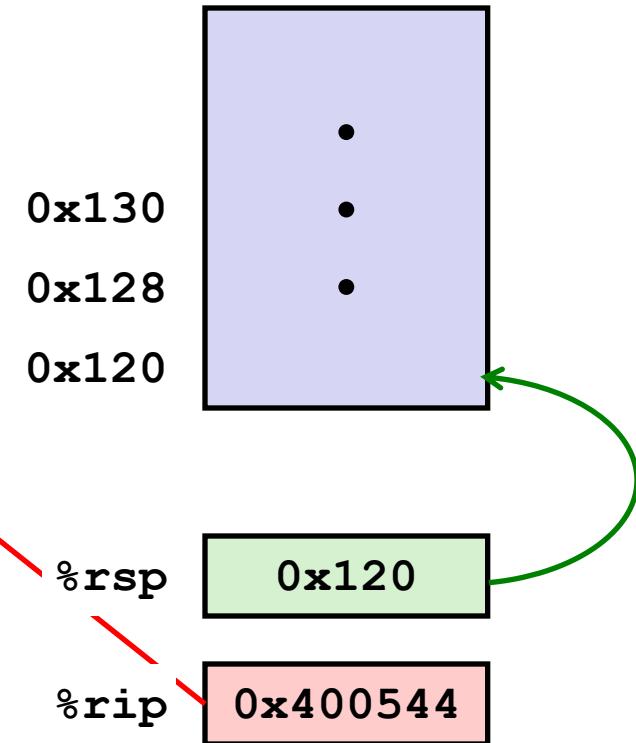
- **Procedure return: `ret`**

- Pop address from stack
 - Jump to address

Control Flow Example #1

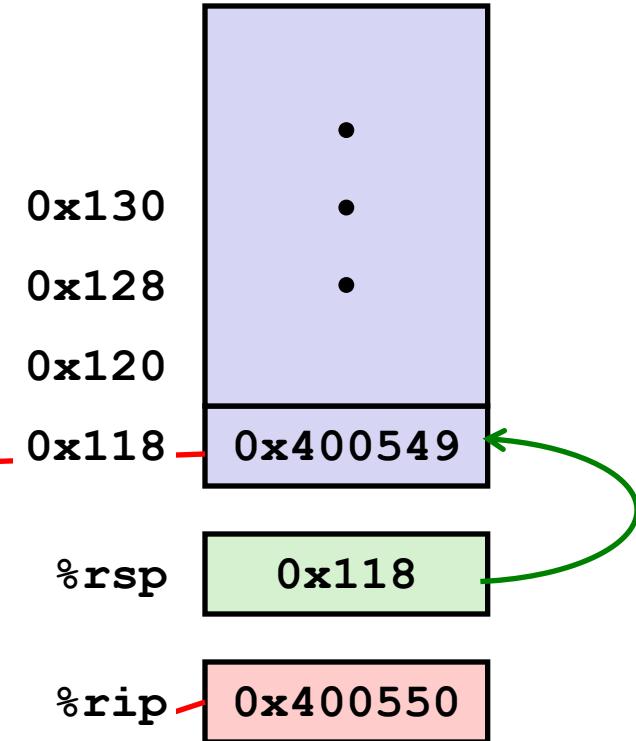
```
0000000000400540 <multstore>:  
.  
.  
400544: callq  400550 <mult2>  
400549: mov     %rax, (%rbx)  
.  
.
```

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



Control Flow Example #2

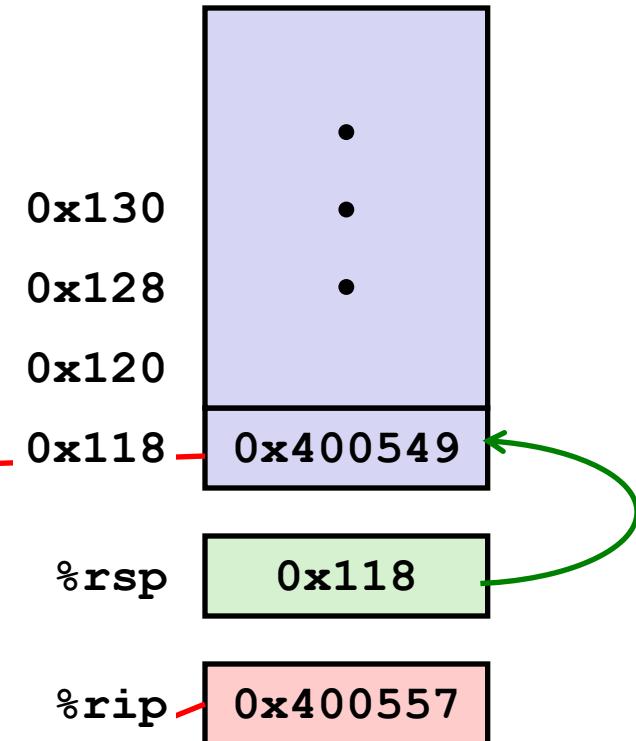
```
0000000000400540 <multstore>:  
.  
.  
400544: callq 400550 <mult2>  
400549: mov    %rax, (%rbx) ←
```



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

Control Flow Example #3

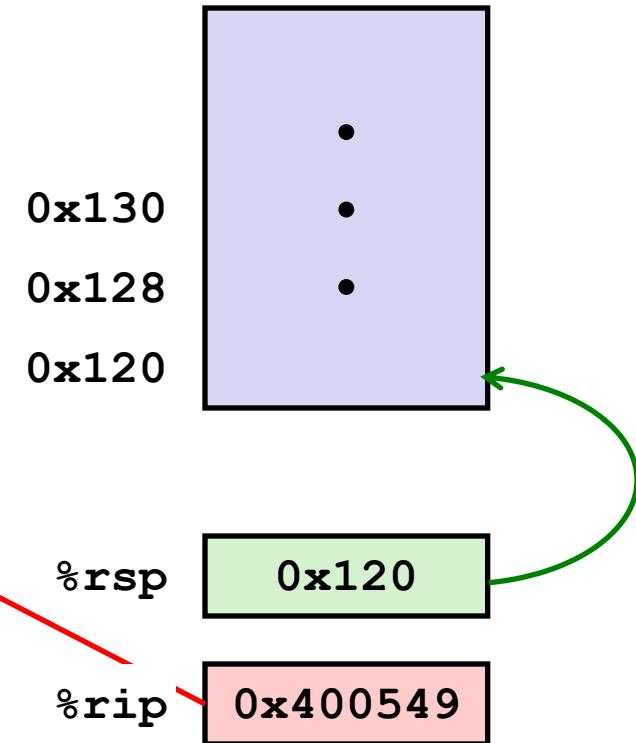
```
0000000000400540 <multstore>:  
.  
.  
400544: callq 400550 <mult2>  
400549: mov %rax, (%rbx) ←
```



```
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
```

Today

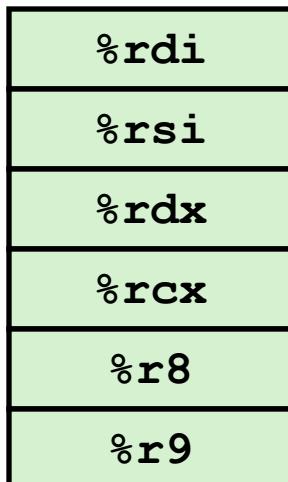
■ Procedures

- Stack Structure
- Calling Conventions
 - Passing control
 - Passing data
 - Managing local data
- Illustrations of Recursion & Pointers

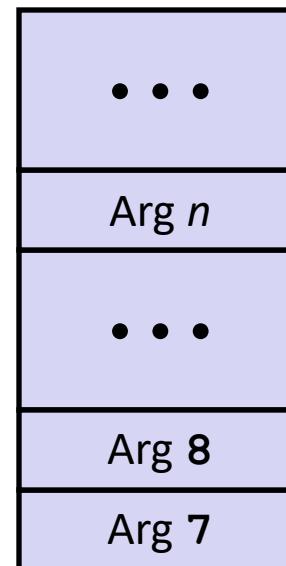
Procedure Data Flow

Registers

- First 6 arguments



Stack



- Return value



- 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
```

Today

■ Procedures

- Stack Structure
- Calling Conventions
 - Passing control
 - Passing data
 - Managing local data
- Illustration of Recursion

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 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

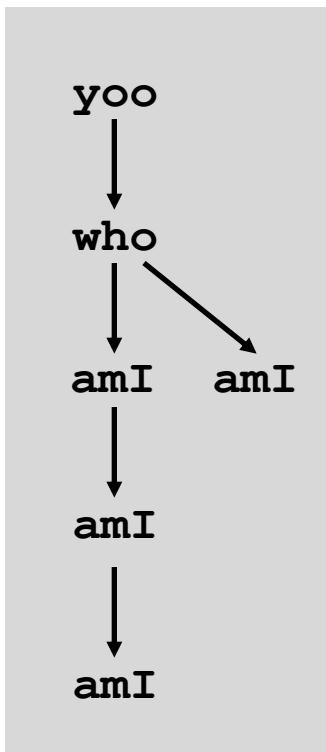
Call Chain Example

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

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

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

Example
Call Chain

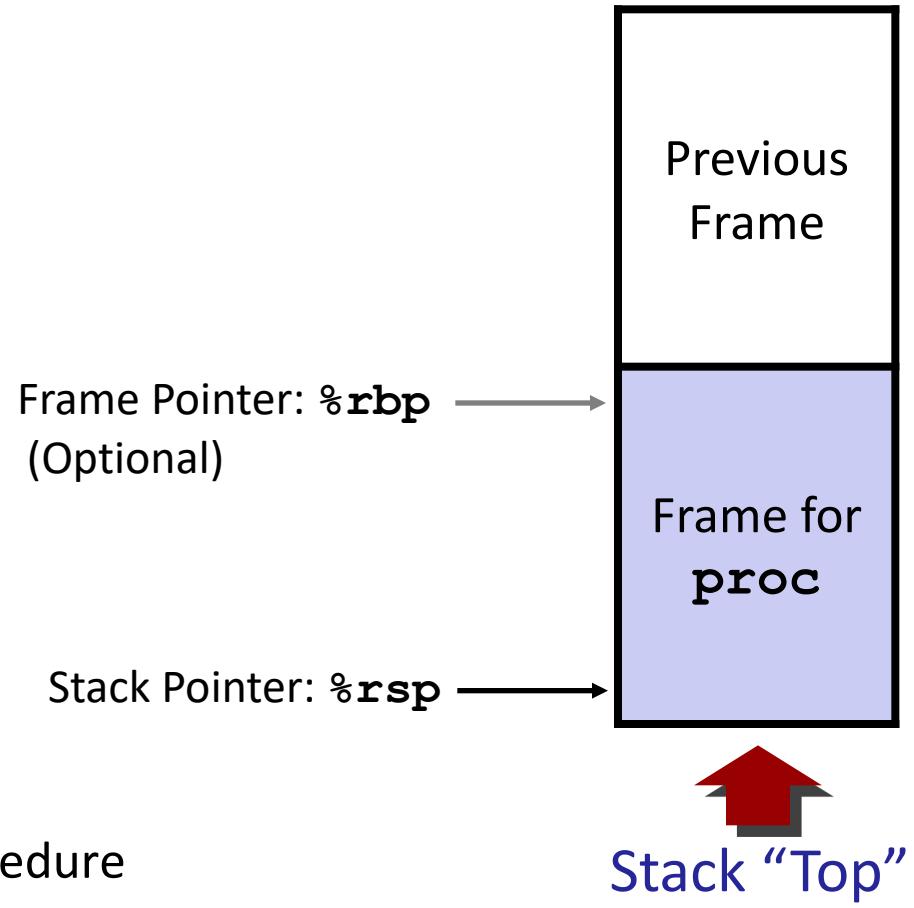


Procedure **amI ()** is recursive

Stack Frames

Contents

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

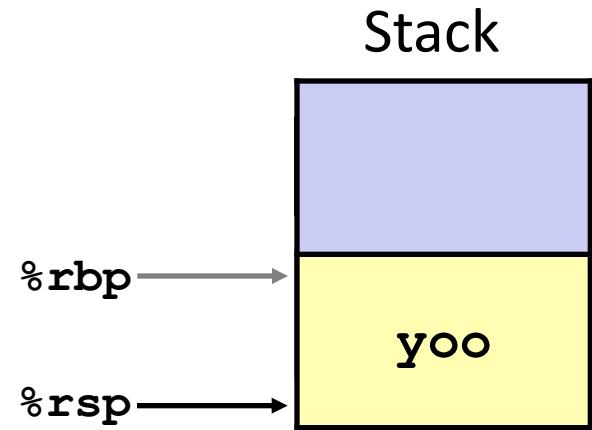
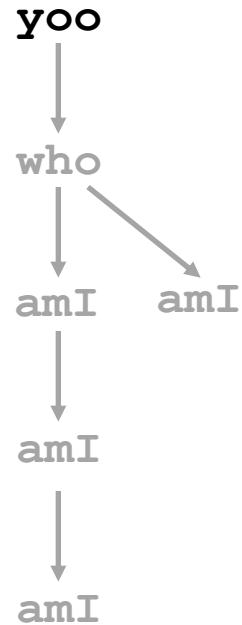


Management

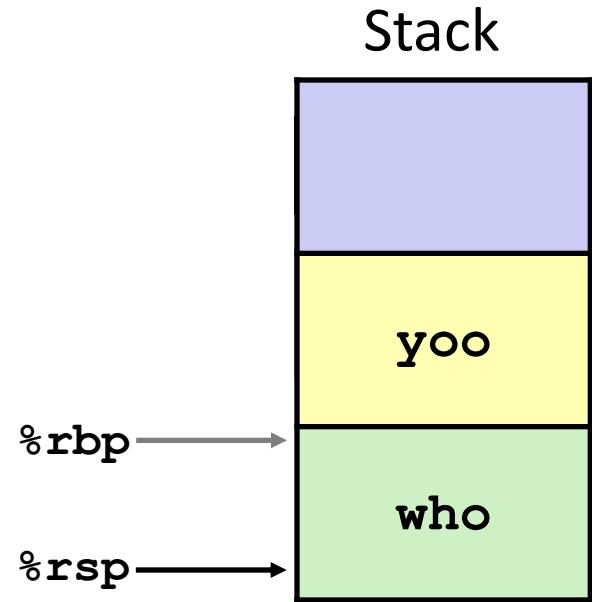
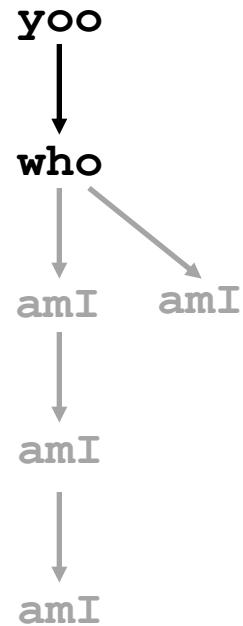
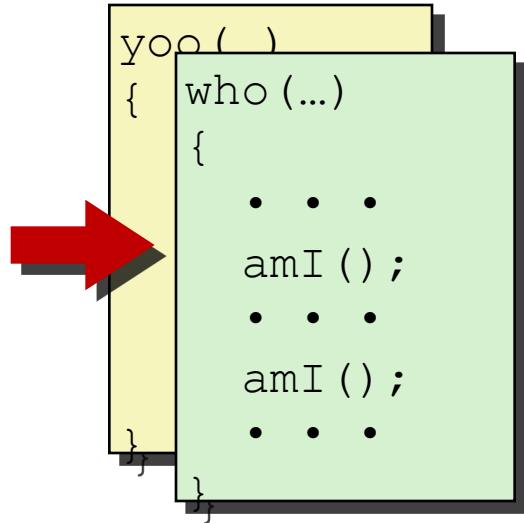
- Space allocated when enter procedure
 - “Set-up” code
 - Includes push by **call** instruction
- Deallocated when return
 - “Finish” code
 - Includes pop by **ret** instruction

Example

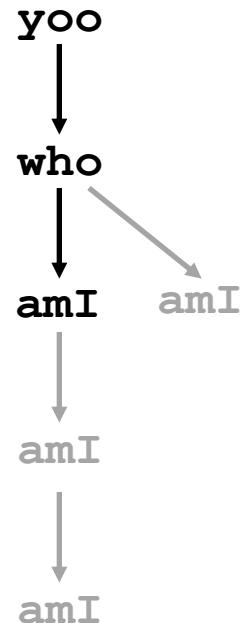
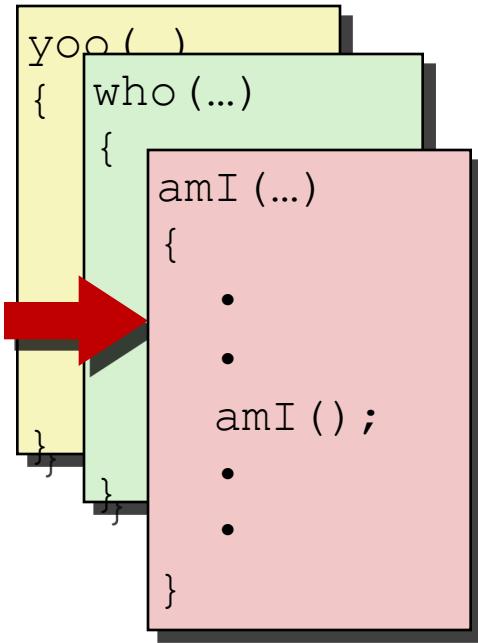
```
yoo (...)  
{  
    •  
    •  
    who () ;  
    •  
    •  
}
```



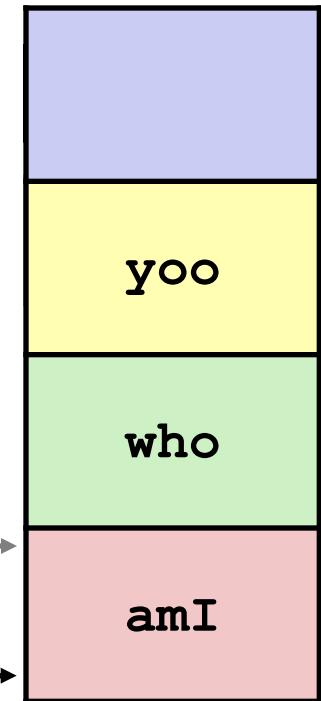
Example



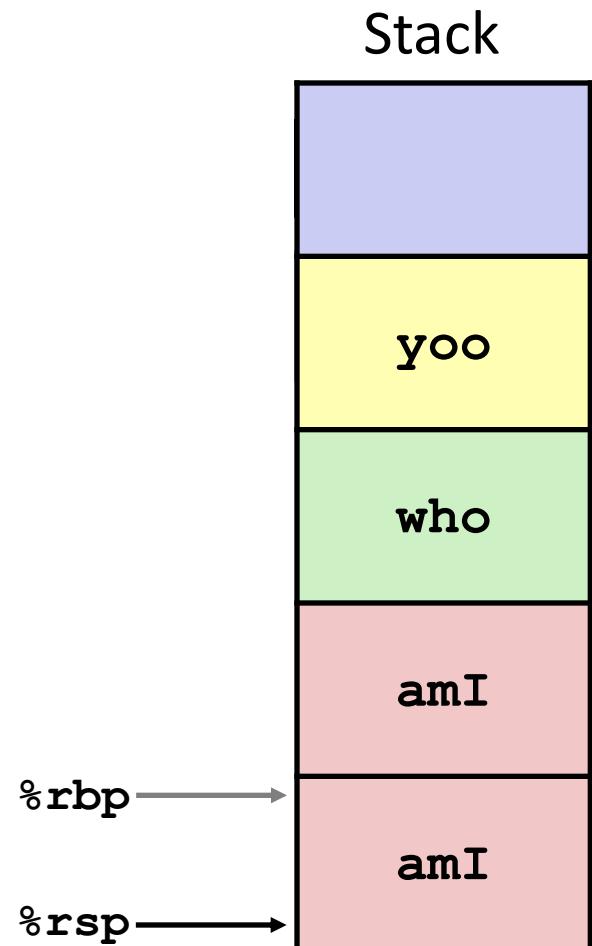
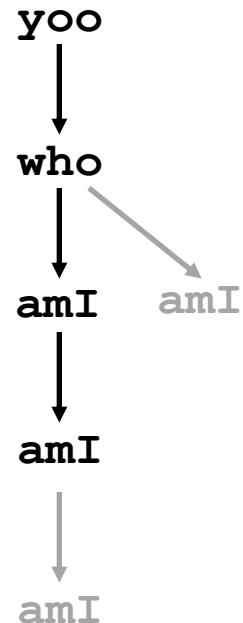
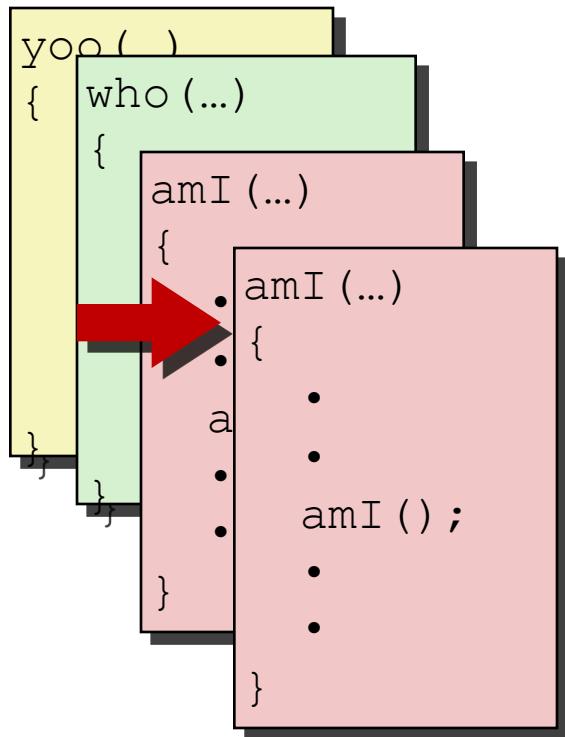
Example



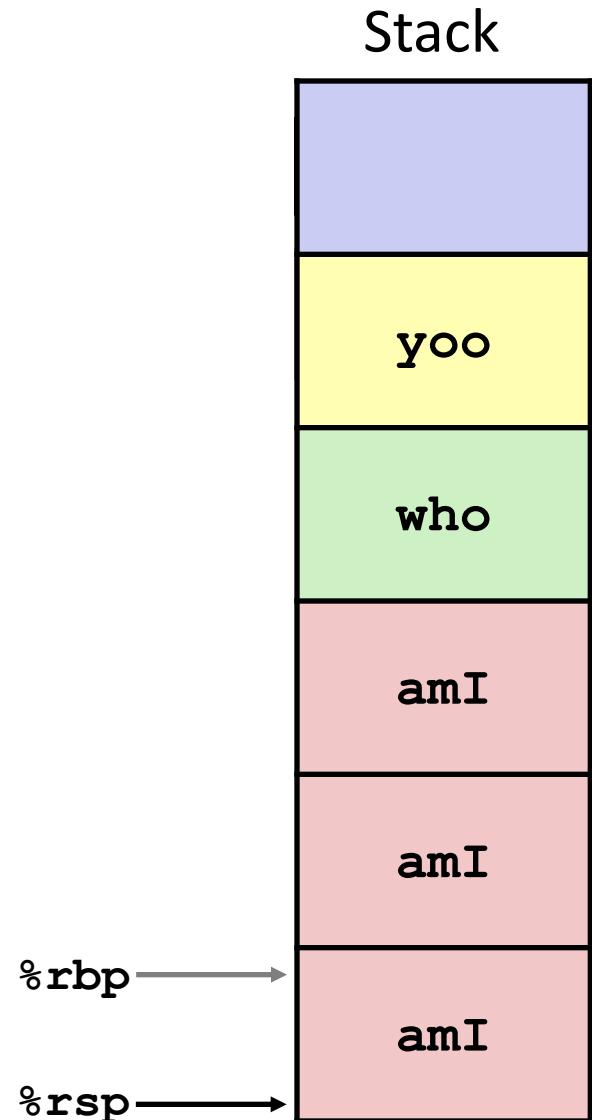
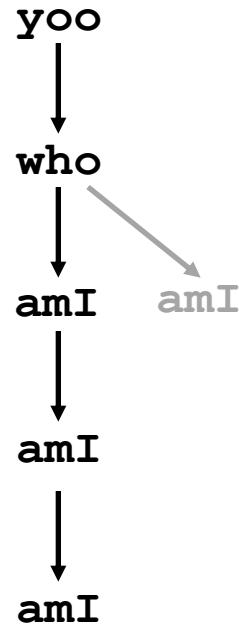
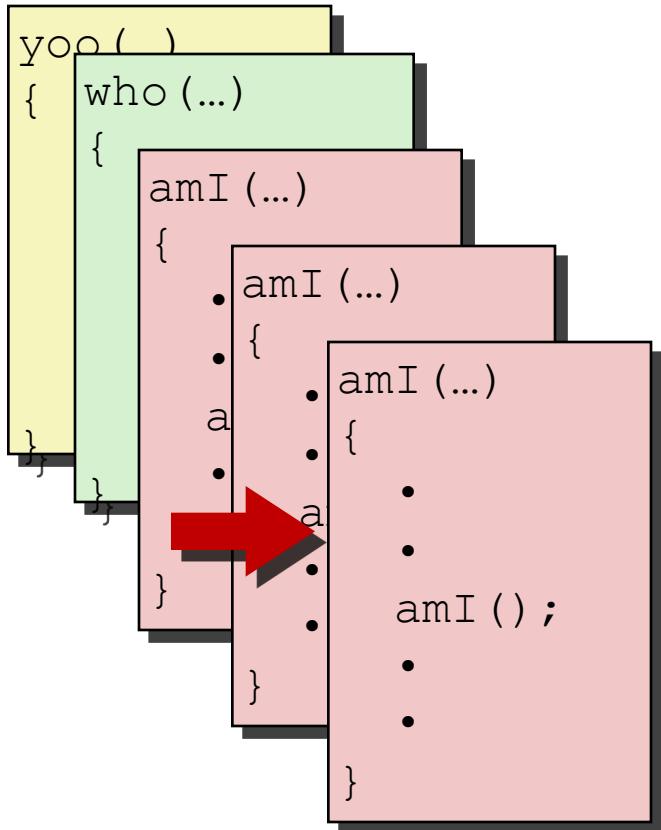
Stack



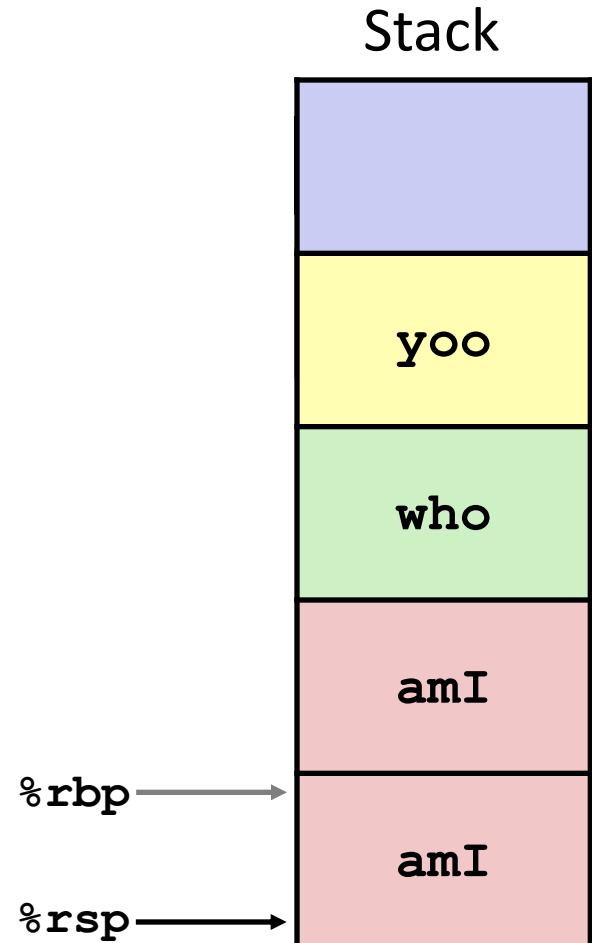
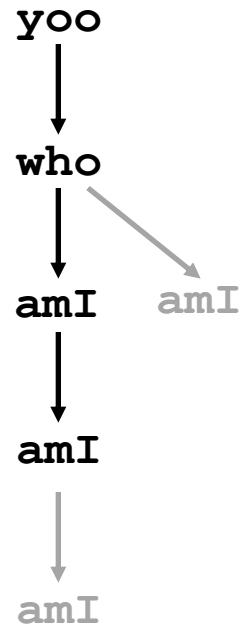
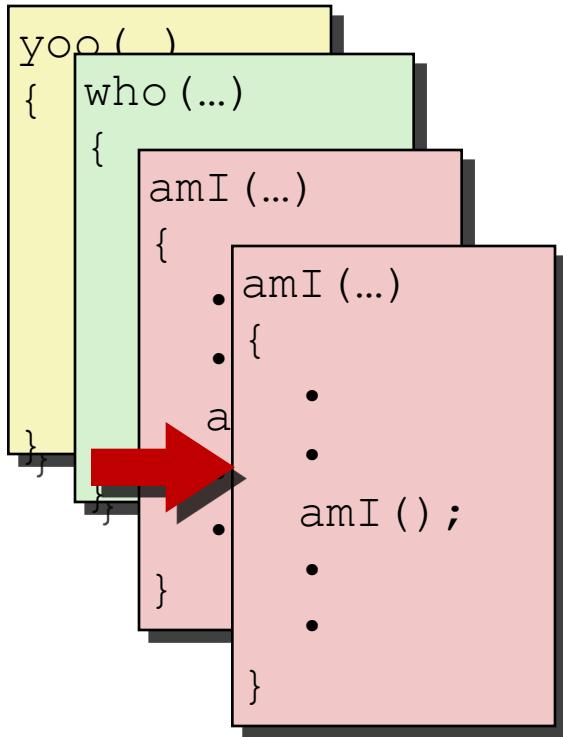
Example



Example

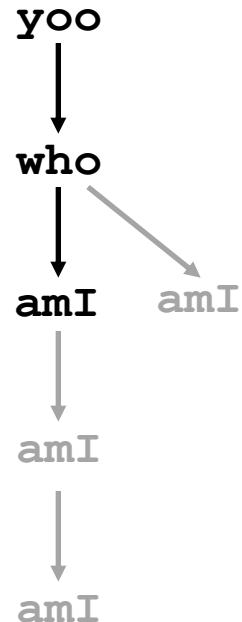


Example

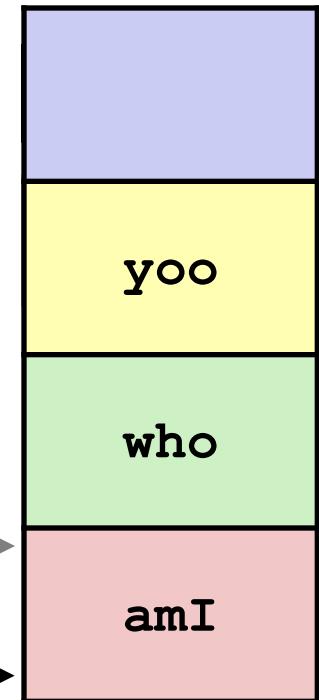


Example

The diagram illustrates nested function scopes using colored boxes and curly braces. A large yellow box at the top contains the text "yoo()". Inside it, a green box contains the text "who (...)". Inside the green box, a pink box contains the text "amI (...)" followed by a brace {}, a red arrow pointing right, and three dots. Below these, another pink box contains the text "amI () ;" followed by a brace {}, a red arrow pointing right, and three dots. The bottom-most pink box ends with a brace }.

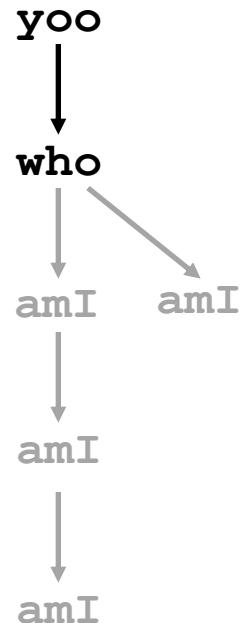


Stack

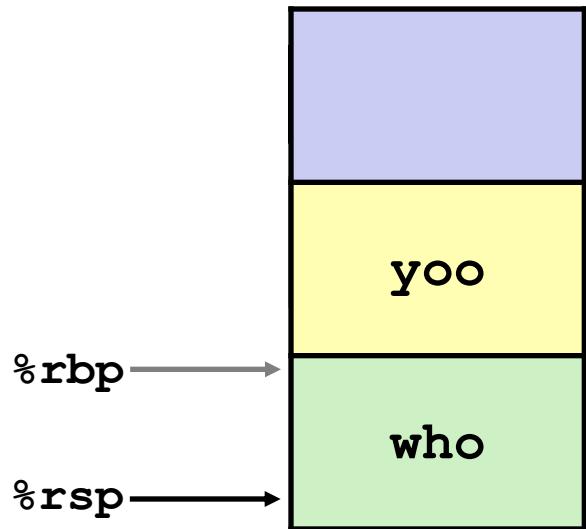


Example

```
yoo( )  
{    who( ... )  
{  
    . . .  
    amI( );  
    . . .  
    amI( );  
    . . .  
}
```



Stack

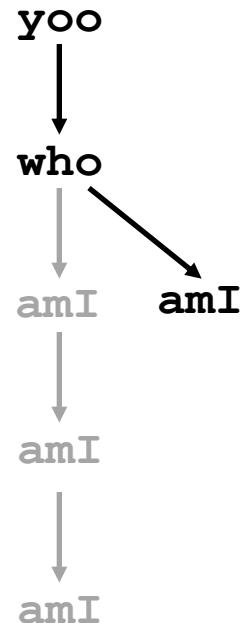


Example

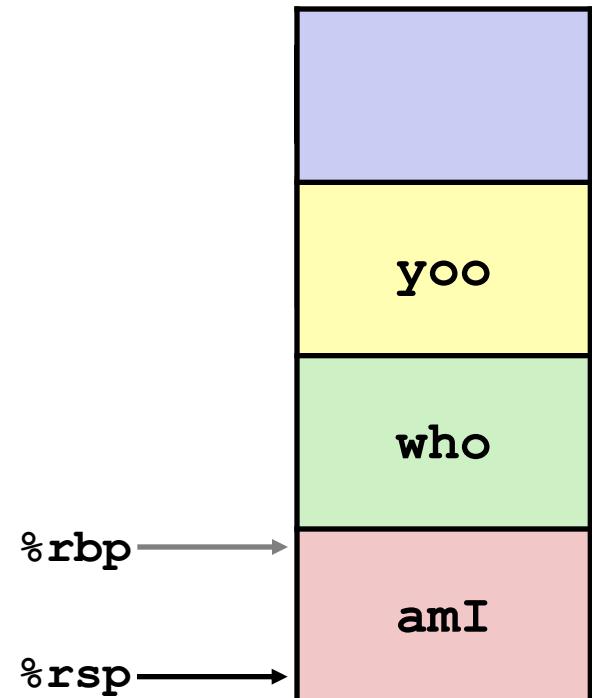
The diagram illustrates nested function scopes using colored boxes:

- A yellow box at the top contains the text "yoo()".
- A green box nested within it contains the text "who (...)".
- A pink box nested within the green one contains the text "amI (...)" followed by five black dots and a semicolon, representing a multi-line string.

A large red arrow points from the left towards the nested pink box, indicating the flow of execution or the scope of the innermost function. The boxes are separated by dark gray horizontal and vertical lines, and each box has a small black bracket at its bottom right corner.

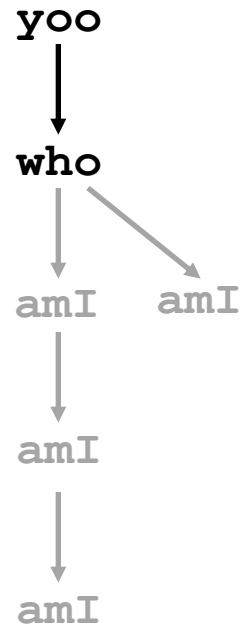


Stack

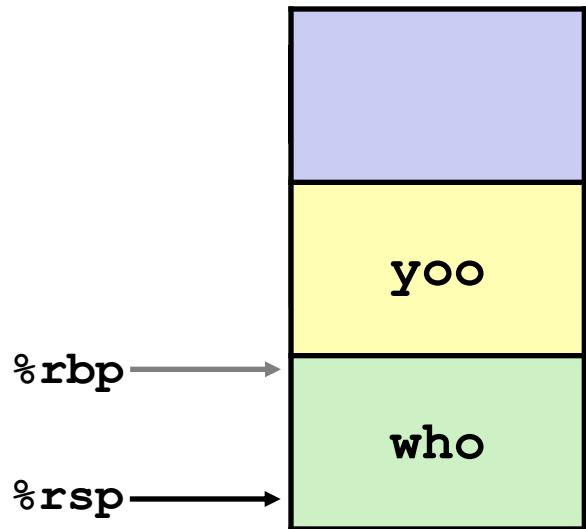


Example

```
yoo( )  
{   who( ... )  
{  
    . . .  
    amI( );  
    . . .  
    amI( );  
    . . .  
}
```

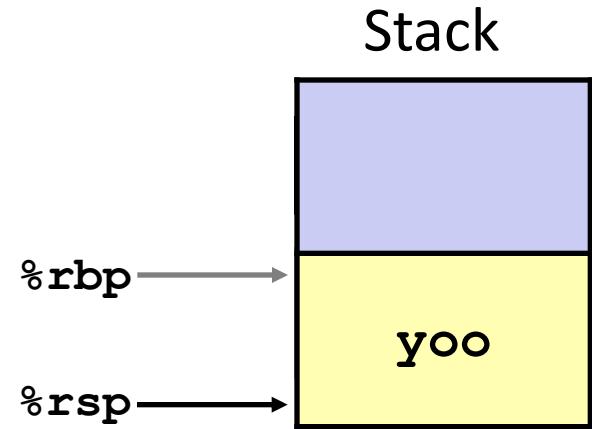
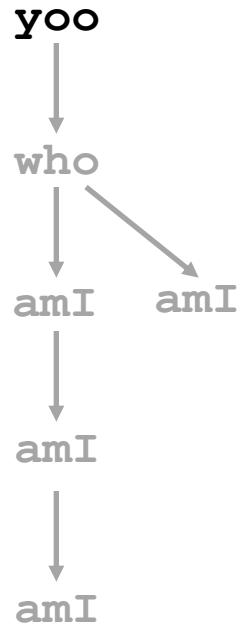


Stack



Example

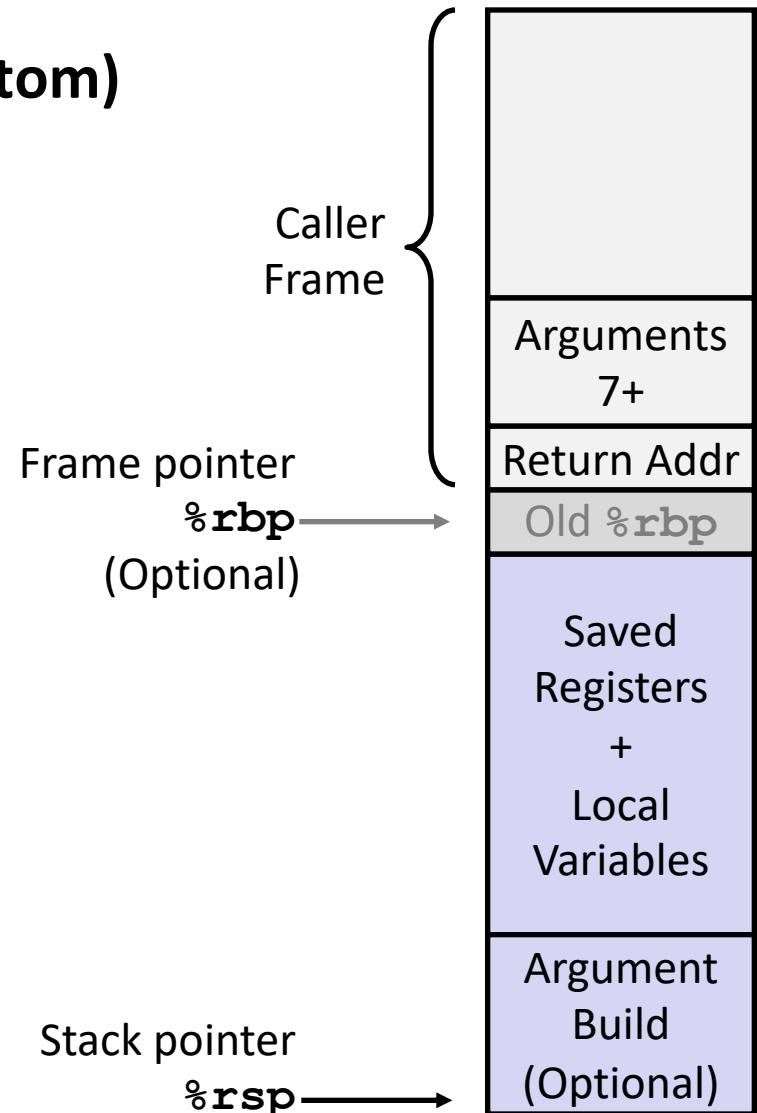
```
yoo (...)  
{  
    •  
    •  
    who () ;  
    •  
    •  
}
```



x86-64/Linux 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;  
}
```

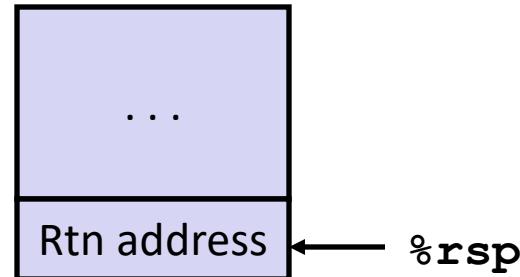
```
incr:  
    movq    (%rdi), %rax  
    addq    %rax, %rsi  
    movq    %rsi, (%rdi)  
    ret
```

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

Example: Calling `incr` #1

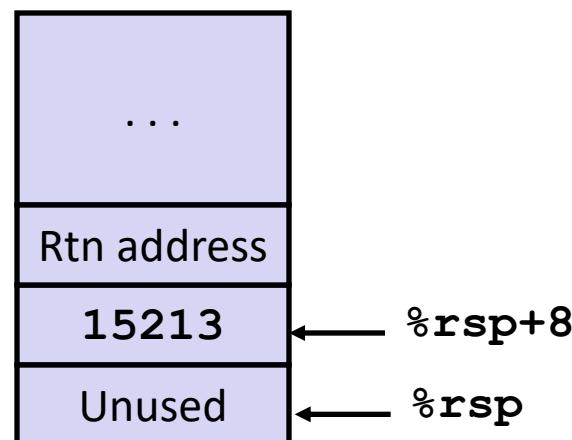
```
long call_incr() {  
    long v1 = 15213;  
    long v2 = incr(&v1, 3000);  
    return v1+v2;  
}
```

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

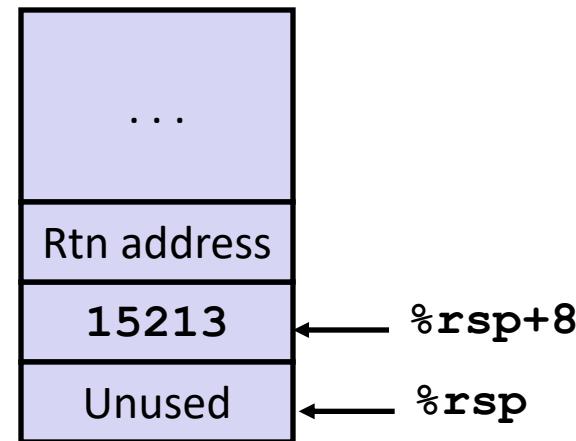


Example: Calling incr #2

```
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



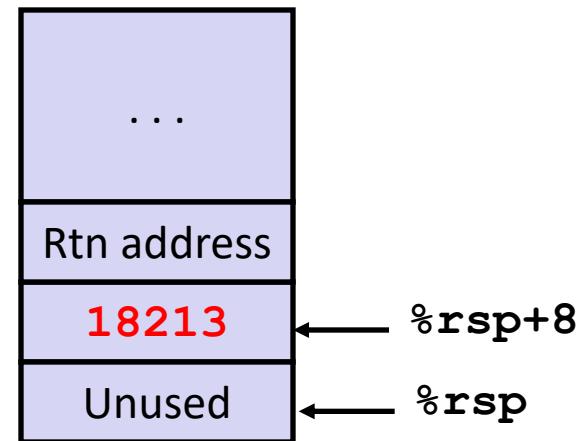
Register	Use(s)
%rdi	&v1
%rsi	3000

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

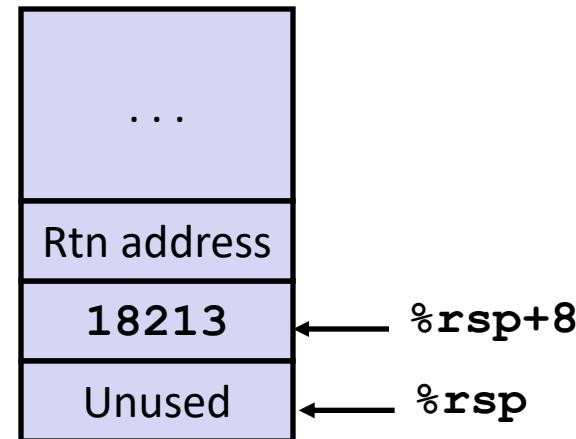


Register	Use(s)
%rdi	&v1
%rsi	3000

Example: Calling incr #4

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

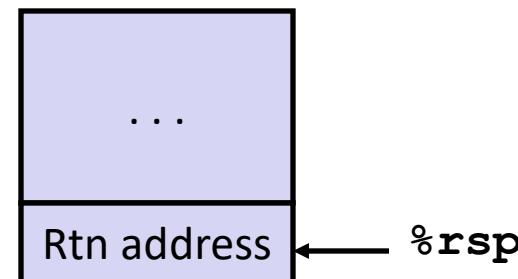
Stack Structure



```
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

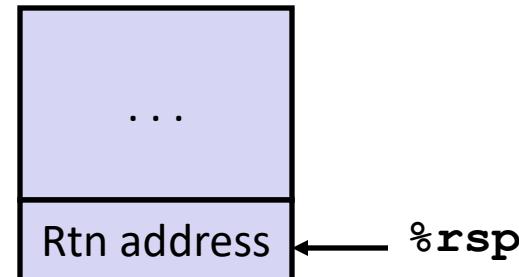
Updated Stack Structure



Example: Calling incr #5

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

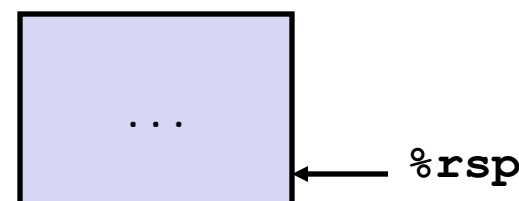
Updated Stack Structure



```
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



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 → something should be done!
 - 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 Linux Register Usage #1

■ **%rax**

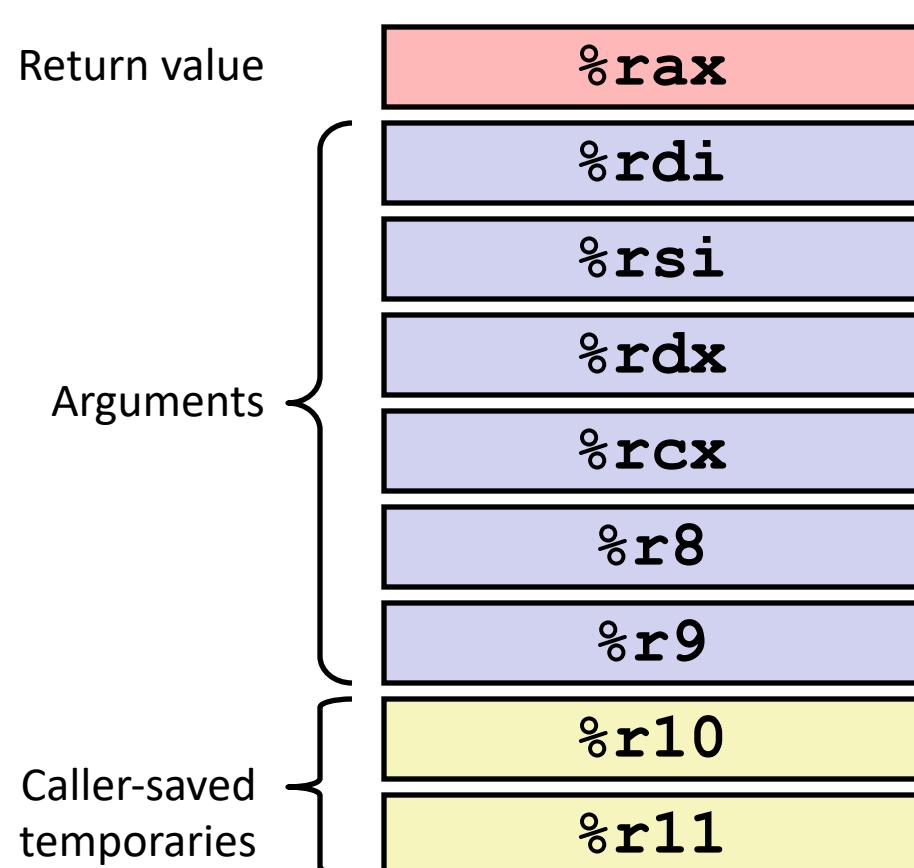
- Return value
- Also caller-saved
- Can be modified by procedure

■ **%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**

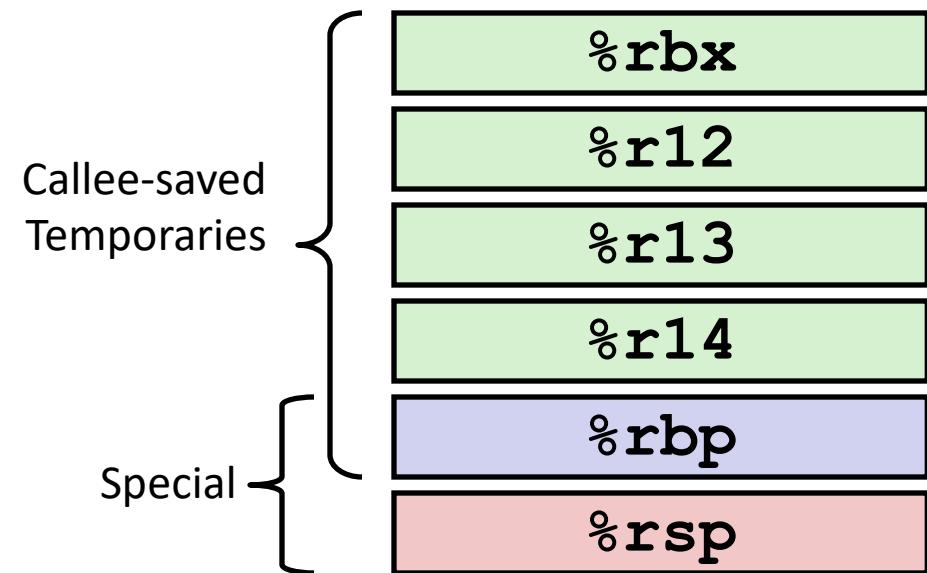
- Callee-saved
- Callee must save & restore

■ **%rbp**

- Callee-saved
- Callee must save & restore
- May be used as frame pointer
- Can mix & match

■ **%rsp**

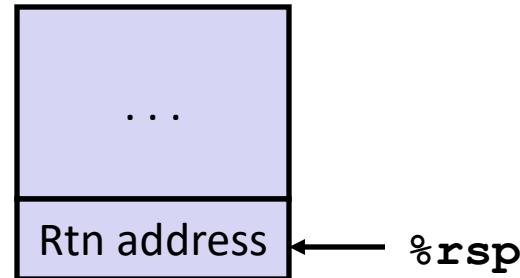
- Special form of callee save
- Restored to original value upon exit from procedure



Callee-Saved Example #1

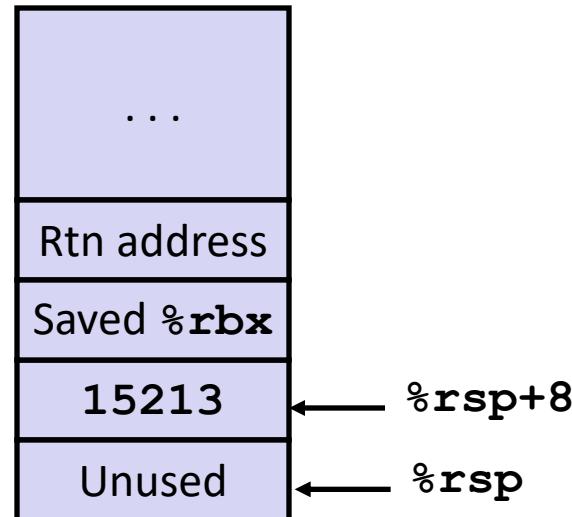
```
long call_incr2(long x) {  
    long v1 = 15213;  
    long v2 = incr(&v1, 3000);  
    return x+v2;  
}
```

Initial Stack Structure



```
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

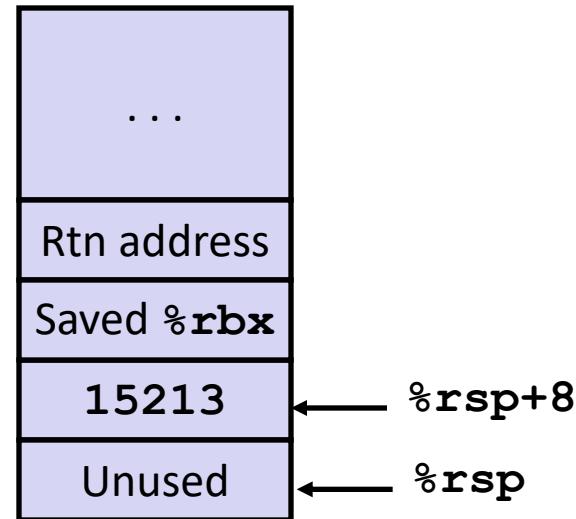


Callee-Saved Example #2

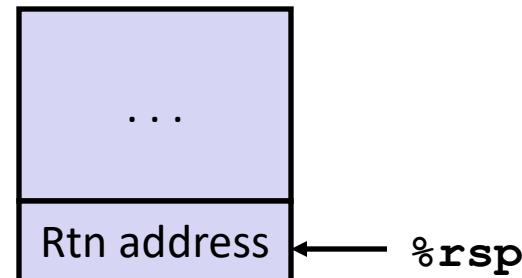
Resulting Stack Structure

```
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
```



Pre-return Stack Structure



Today

■ Procedures

- Stack Structure
- Calling Conventions
 - Passing control
 - Passing data
 - Managing local data
- Illustration of Recursion

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   %rdi # (by 1)
    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   %rdi # (by 1)
    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);
}
```

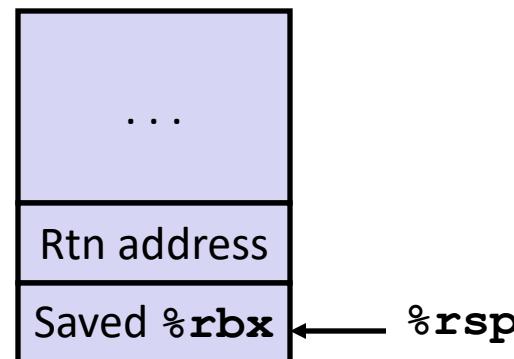
pcount_r:

```
    movl    $0, %eax
    testq   %rdi, %rdi
    je      .L6
    pushq   %rbx
    movq   %rdi, %rbx
    andl   $1, %ebx
    shrq   %rdi # (by 1)
    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    %rdi # (by 1)
    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    %rdi # (by 1)
    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	%rdi # (by 1)
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);
}
```

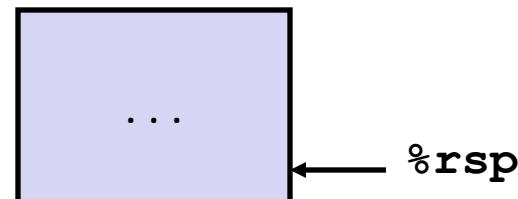
pcount_r:

```
    movl    $0, %eax
    testq   %rdi, %rdi
    je      .L6
    pushq   %rbx
    movq   %rdi, %rbx
    andl   $1, %ebx
    shrq   %rdi # (by 1)
    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
 - Unless the C code explicitly does so (e.g., buffer overflow in Lecture 9)
- 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

x86-64 Procedure Summary

■ Important Points

- Stack is the right data structure for procedure call / return
 - If P calls Q, then Q returns before P

■ Recursion (& mutual recursion) handled by normal calling conventions

- Can safely store values in local stack frame and in callee-saved registers
- Put function arguments at top of stack
- Result return in **%rax**

■ Pointers are addresses of values

- On stack or global

