Machine-Level Programming III: Procedures

15-213/18-213: Introduction to Computer Systems
7th Lecture, June 4, 2019

Instructor:
Sol Boucher

```
long switch_eg
   (long x, long y, long z)
{
    long w = 1;
    switch(x) {
    case 1:
        w = y*z;
        break;
    case 2:
        w = y/z;
        /* Fall Through */
    case 3:
        w += z;
        break;
    case 5:
    case 6:
        w -= z;
        break;
    default:
        w = 2;
    }
    return w;
}
```

Switch Statement Example

- **Multiple case labels**
  - Here: 5 & 6
- **Fall through cases**
  - Here: 2
- **Missing cases**
  - Here: 4

Jump Table Structure

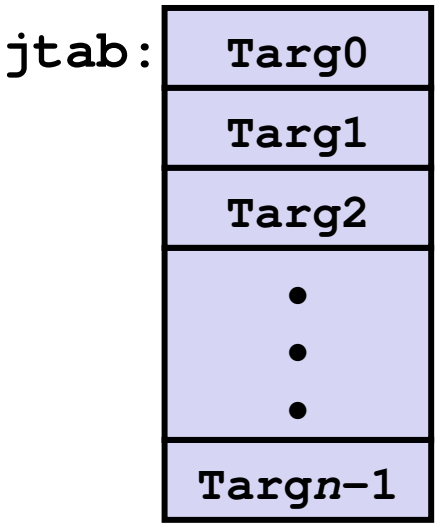
Switch Form

```
switch(x) {
  case val_0:
    Block 0
  case val_1:
    Block 1
    • • •
  case val_n-1:
    Block n-1
}
```

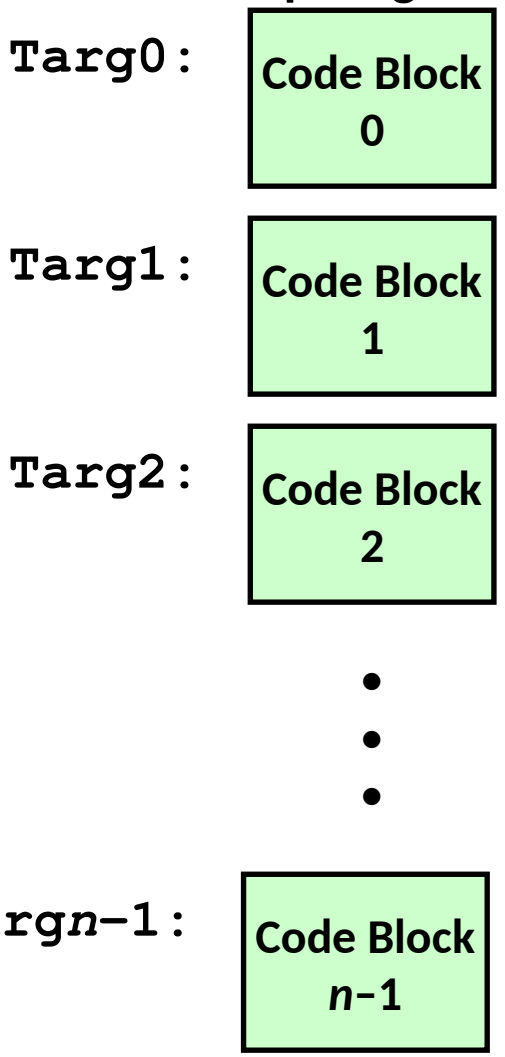
Translation (Extended C)

```
goto *JTab[x];
```

Jump Table



Jump Targets



Switch Statement Example

```
long switch_eg(long x, long y, long z)
{
    long w = 1;
    switch(x) {
      . . .
    }
    return w;
}
```

Setup:

```
switch_eg:
    movq    %rdx, %rcx
    cmpq    $6, %rdi    # x:6
    ja      .L8
    jmp     *.L4(,%rdi,8)
```

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rdx	Argument z
%rax	Return value

What range of values takes default?

Note that `w` not initialized here

Switch Statement Example

```
long switch_eg(long x, long y, long z)
{
    long w = 1;
    switch(x) {
      . . .
    }
    return w;
}
```

Jump table

```
.section .rodata
    .align 8
.L4:
    .quad    .L8 # x = 0
    .quad    .L3 # x = 1
    .quad    .L5 # x = 2
    .quad    .L9 # x = 3
    .quad    .L8 # x = 4
    .quad    .L7 # x = 5
    .quad    .L7 # x = 6
```

Setup:

```
switch_eg:
    movq    %rdx, %rcx
    cmpq    $6, %rdi        # x:6
    ja      .L8             # Use default
    jmp     *.L4(,%rdi,8)   # goto *JTab[x]
```

Indirect jump

Assembly Setup Explanation

Table Structure

- Each target requires 8 bytes
- Base address at `.L4`

Jump table

```
.section .rodata
    .align 8
.L4:
    .quad    .L8 # x = 0
    .quad    .L3 # x = 1
    .quad    .L5 # x = 2
    .quad    .L9 # x = 3
    .quad    .L8 # x = 4
    .quad    .L7 # x = 5
    .quad    .L7 # x = 6
```

Jumping

- Direct: `jmp .L8`
- Jump target is denoted by label `.L8`

- Indirect: `jmp *.L4(,%rdi,8)`
- Start of jump table: `.L4`
- Must scale by factor of 8 (addresses are 8 bytes)
- Fetch target from effective Address `.L4 + x*8`
  - Only for $0 \le x \le 6$

Jump Table

Jump table

```
.section .rodata
    .align 8
.L4:
    .quad    .L8  # x = 0
    .quad    .L3  # x = 1
    .quad    .L5  # x = 2
    .quad    .L9  # x = 3
    .quad    .L8  # x = 4
    .quad    .L7  # x = 5
    .quad    .L7  # x = 6
```

```
    switch(x) {
    case 1:       // .L3
        w = y*z;
        break;
    case 2:       // .L5
        w = y/z;
        /* Fall Through */
    case 3:       // .L9
        w += z;
        break;
    case 5:
    case 6:       // .L7
        w -= z;
        break;
    default:      // .L8
        w = 2;
    }
```

Code Blocks (x == 1)

```
  switch(x) {
  case 1:   // .L3
        w = y*z;
        break;
    . . .
}
```

```
.L3:
   movq    %rsi, %rax  # y
   imulq   %rdx, %rax  # y*z
   ret
```

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rdx	Argument z
%rax	Return value

Handling Fall-Through

```
long w = 1;
. . .
switch(x) {
. . .
case 2:
    w = y/z;
    /* Fall Through */
case 3:
    w += z;
    break;
. . .
}
```

```
case 2:
    w = y/z;
    goto merge;
```

```
case 3:
    w = 1;
merge:
    w += z;
```

Code Blocks (x == 2, x == 3)

```
    long w = 1;
. . .
    switch(x) {
. . .
    case 2:
        w = y/z;
        /* Fall Through */
    case 3:
        w += z;
        break;
. . .
    }
```

```
.L5:                        # Case 2
   movq     %rsi, %rax
   cqto
   idivq    %rcx            #  y/z
   jmp      .L6             #  goto merge
.L9:                        # Case 3
   movl     $1, %eax        #  w = 1
.L6:                        # merge:
   addq     %rcx, %rax      #  w += z
   ret
```

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rdx	Argument z
%rax	Return value

Code Blocks (x == 5, x == 6, default)

```
  switch(x) {
    . . .
    case 5:  // .L7
    case 6:  // .L7
        w -= z;
        break;
    default: // .L8
        w = 2;
}
```

```
.L7:                    # Case 5,6
  movl  $1, %eax        #  w = 1
  subq  %rdx, %rax      #  w -= z
  ret
.L8:                    # Default:
  movl  $2, %eax        #  2
  ret
```

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rdx	Argument z
%rax	Return value

Finding Jump Table in Binary

```
 00000000004005e0 <switch_eg>:
4005e0:       48 89 d1                mov    %rdx,%rcx
4005e3:       48 83 ff 06             cmp    $0x6,%rdi
4005e7:       77 2b                   ja     400614 <switch_eg+0x34>
4005e9:       ff 24 fd f0 07 40 00    jmpq   *0x4007f0(,%rdi,8)
4005f0:       48 89 f0                mov    %rsi,%rax
4005f3:       48 0f af c2             imul   %rdx,%rax
4005f7:       c3                      retq
4005f8:       48 89 f0                mov    %rsi,%rax
4005fb:       48 99                   cqto
4005fd:       48 f7 f9                idiv   %rcx
400600:       eb 05                   jmp    400607 <switch_eg+0x27>
400602:       b8 01 00 00 00          mov    $0x1,%eax
400607:       48 01 c8                add    %rcx,%rax
40060a:       c3                      retq
40060b:       b8 01 00 00 00          mov    $0x1,%eax
400610:       48 29 d0                sub    %rdx,%rax
400613:       c3                      retq
400614:       b8 02 00 00 00          mov    $0x2,%eax
400619:       c3                      retq
```

Finding Jump Table in Binary (cont.)

```
 00000000004005e0 <switch_eg>:
. . .
4005e9:       ff 24 fd f0 07 40 00    jmpq   *0x4007f0(,%rdi,8)
. . .
```

```
 % gdb switch
(gdb) x /8xg 0x4007f0
0x4007f0:       0x0000000000400614      0x00000000004005f0
0x400800:       0x00000000004005f8      0x0000000000400602
0x400810:       0x0000000000400614      0x000000000040060b
0x400820:       0x000000000040060b      0x2c646c25203d2078
(gdb)
```

Finding Jump Table in Binary (cont.)

```
 % gdb switch
(gdb) x /8xg 0x4007f0
0x4007f0:       0x0000000000400614      0x00000000004005f0
0x400800:       0x00000000004005f8      0x0000000000400602
0x400810:       0x0000000000400614      0x000000000040060b
0x400820:       0x000000000040060b      0x2c646c25203d2078
```

```
  . . .
  4005f0:       48 89 f0                mov    %rsi,%rax
  4005f3:       48 0f af c2             imul   %rdx,%rax
  4005f7:       c3                      retq
  4005f8:       48 89 f0                mov    %rsi,%rax
  4005fb:       48 99                   cqto
  4005fd:       48 f7 f9                idiv   %rcx
  400600:       eb 05                   jmp    400607 <switch_eg+0x27>
  400602:       b8 01 00 00 00          mov    $0x1,%eax
  400607:       48 01 c8                add    %rcx,%rax
  40060a:       c3                      retq
  40060b:       b8 01 00 00 00          mov    $0x1,%eax
  400610:       48 29 d0                sub    %rdx,%rax
  400613:       c3                      retq
  400614:       b8 02 00 00 00          mov    $0x2,%eax
  400619:       c3                      retq
```

Today

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

Mechanisms in Procedures

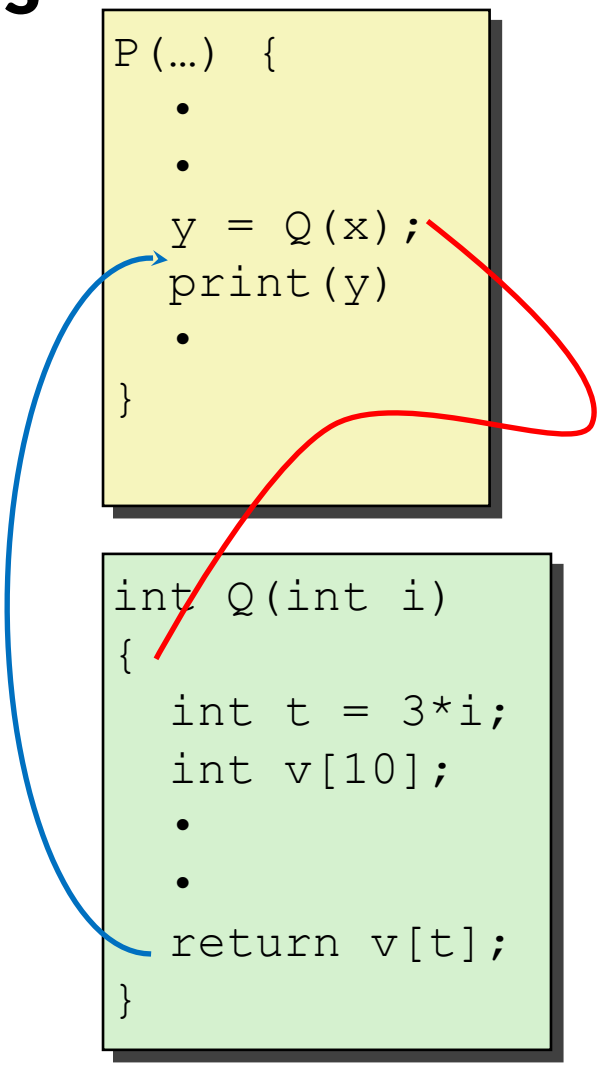
- **Passing control**
  - To beginning of procedure code
  - Back to return point
- **Passing data**
  - Procedure arguments
  - Return value
- **Memory management**
  - Allocate during procedure execution
  - Deallocate upon return
- **Mechanisms all implemented with machine instructions**

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

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

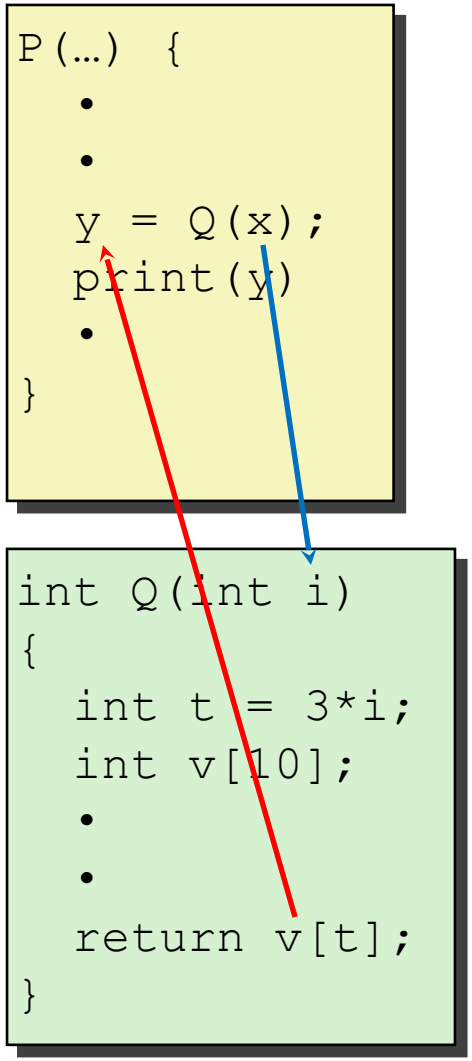
Mechanisms in Procedures

- **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**



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**



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];
}
```

Mechanisms in Procedures

Machine instructions implement the mechanisms, but the choices are determined by designers. These choices make up the **Application Binary Interface (ABI)**.

- Deallocate upon return

- **Mechanisms all implemented with machine instructions**

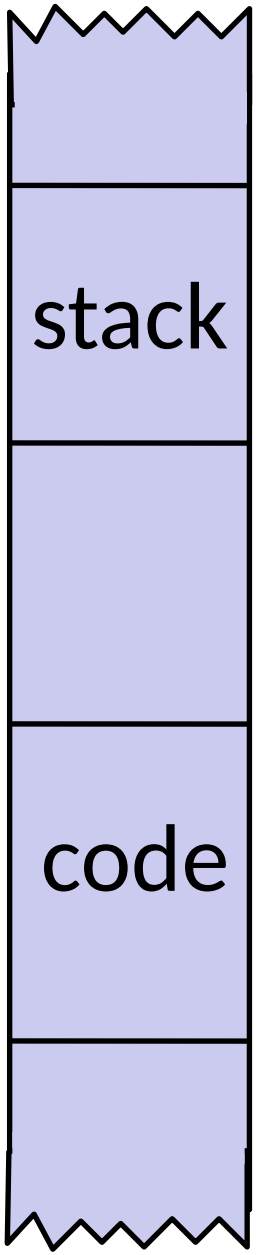
```
P( ) {
  int v[10];
  •
  •
  return v[t];
}
```

Today

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

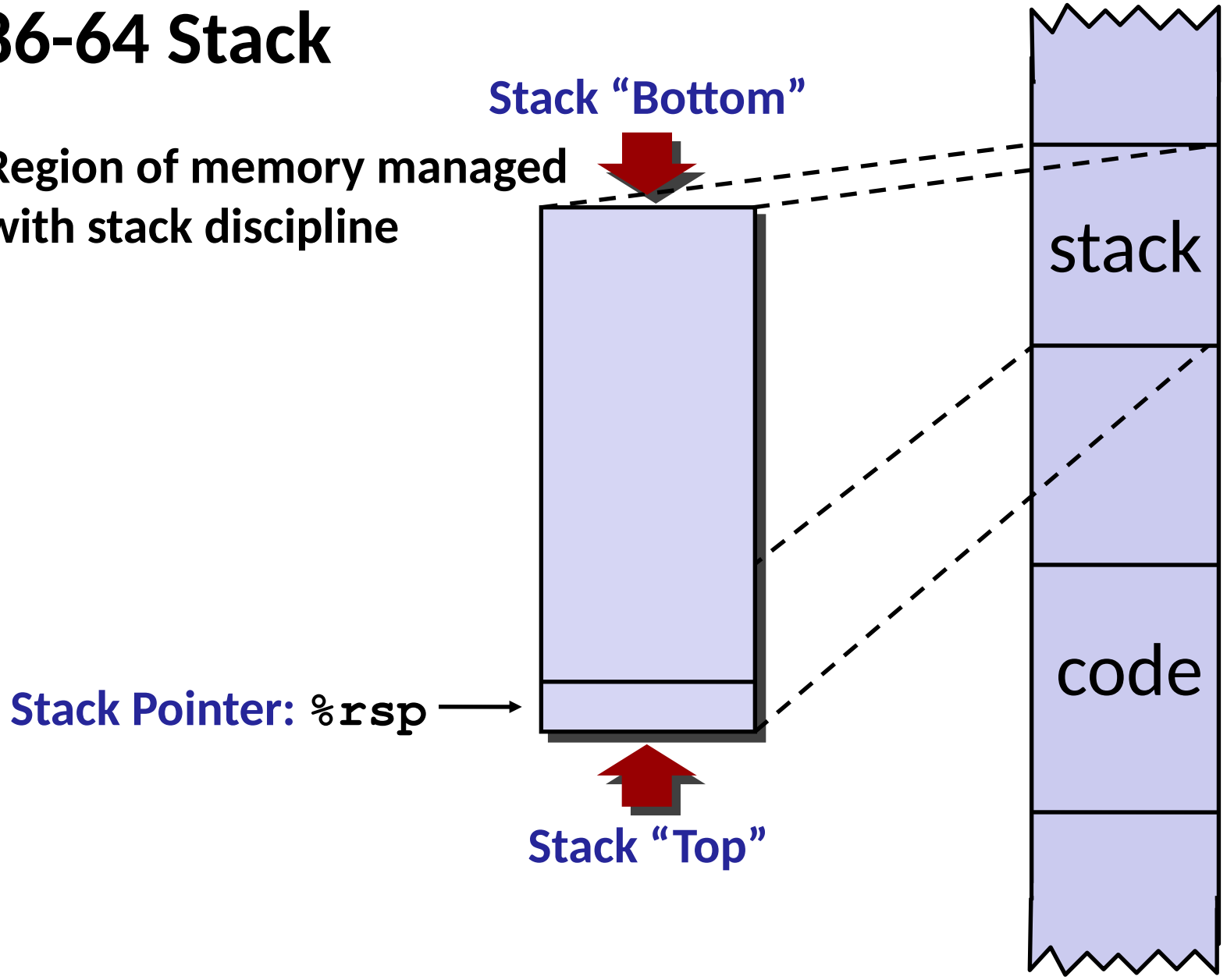
x86-64 Stack

- **Region of memory managed with stack discipline**
  - Memory viewed as array of bytes.
  - Different regions have different purposes.
  - (Like ABI, a policy decision)



x86-64 Stack

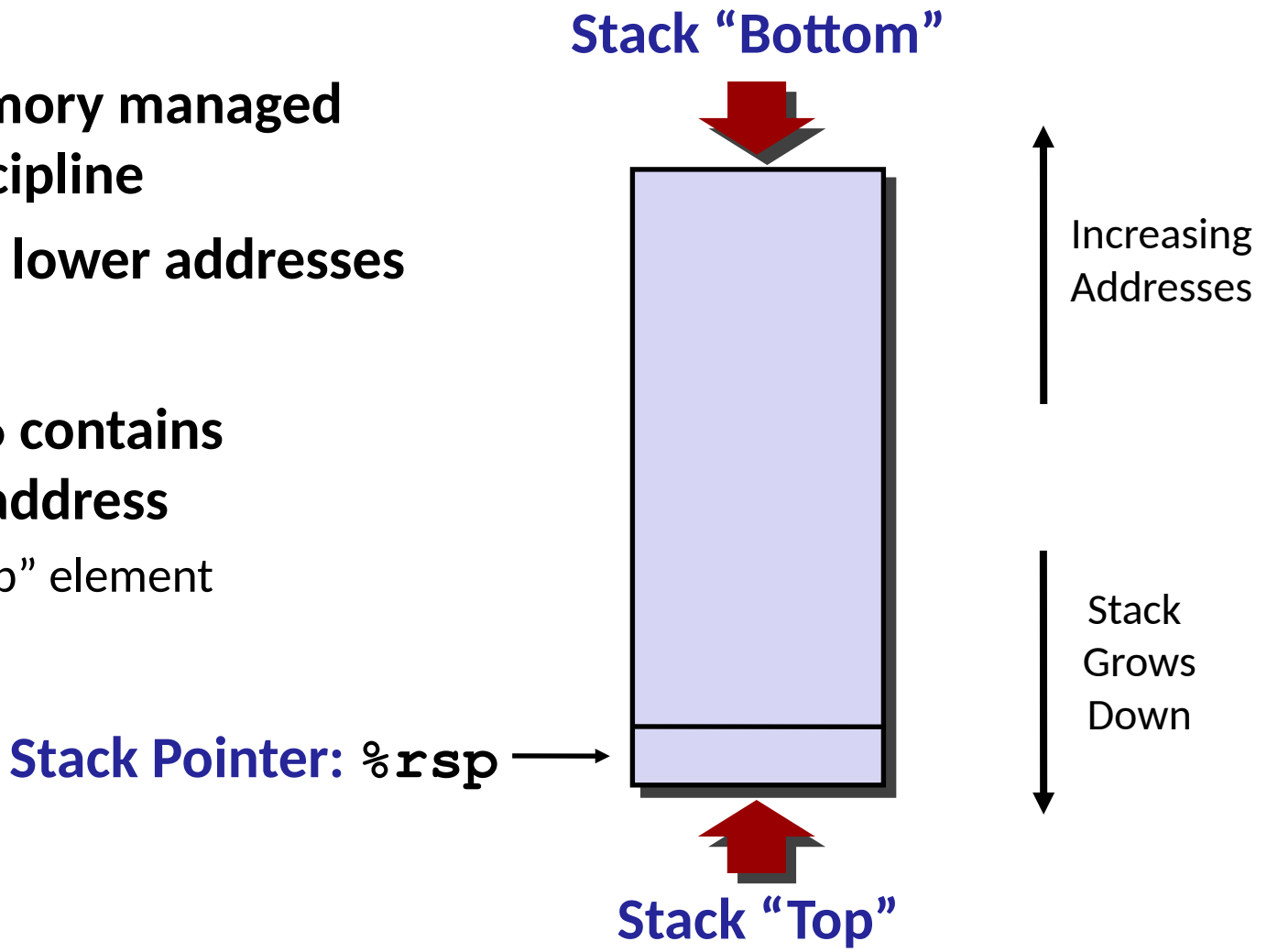
- Region of memory managed with stack discipline



x86-64 Stack

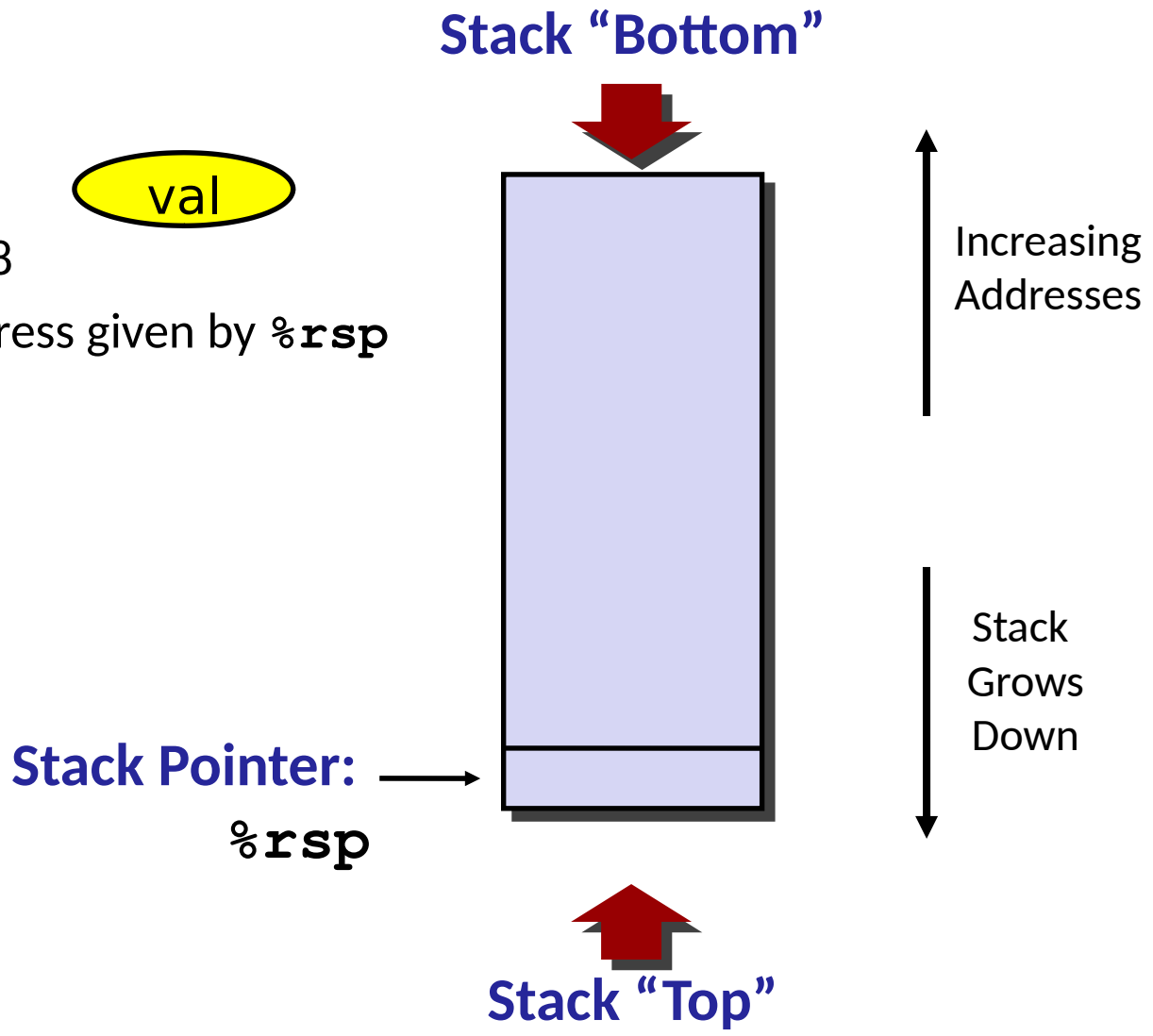
- Region of memory managed with stack discipline
- Grows toward lower addresses

- Register `%rsp` contains lowest stack address
  - address of "top" element



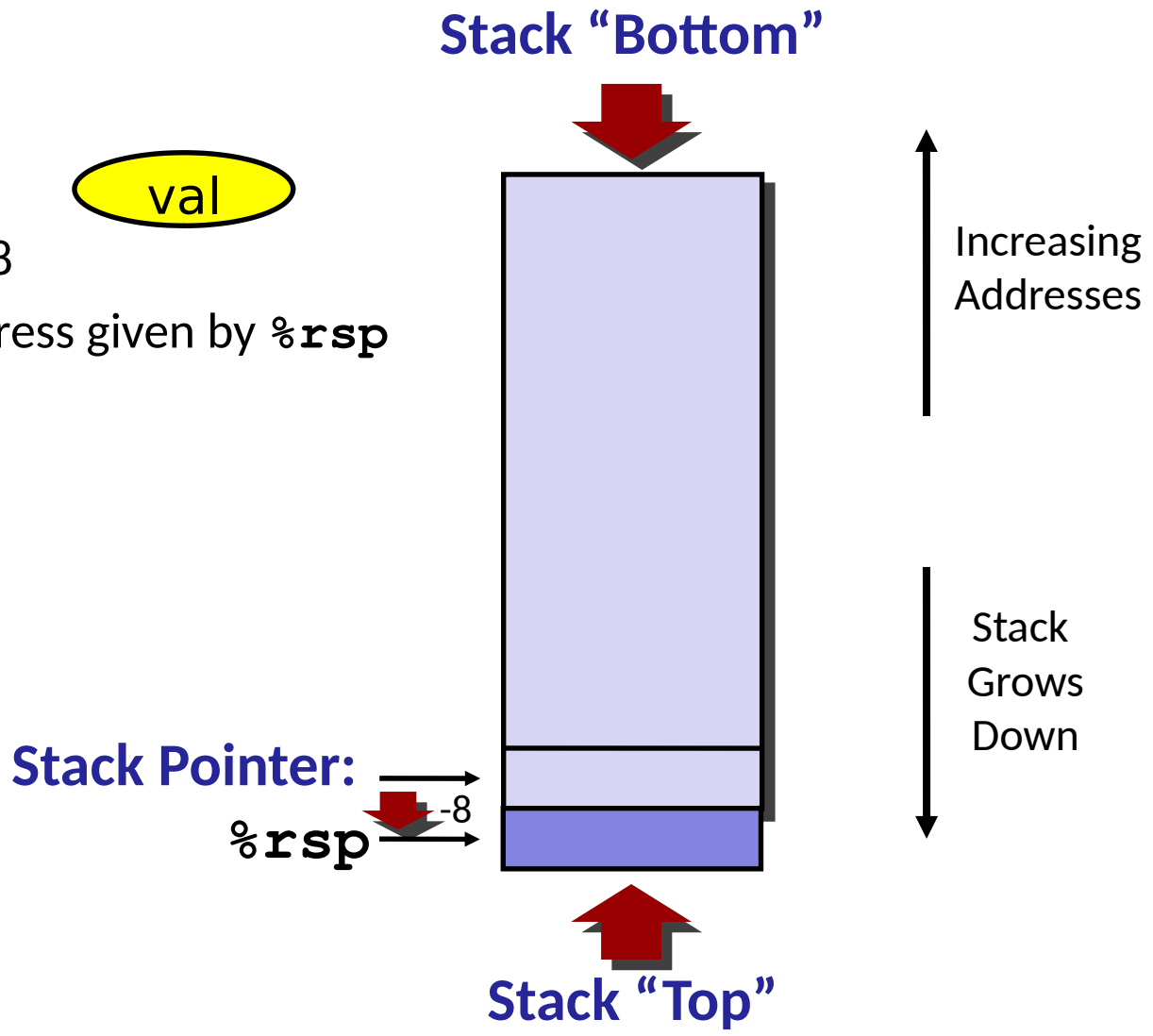
x86-64 Stack: Push

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



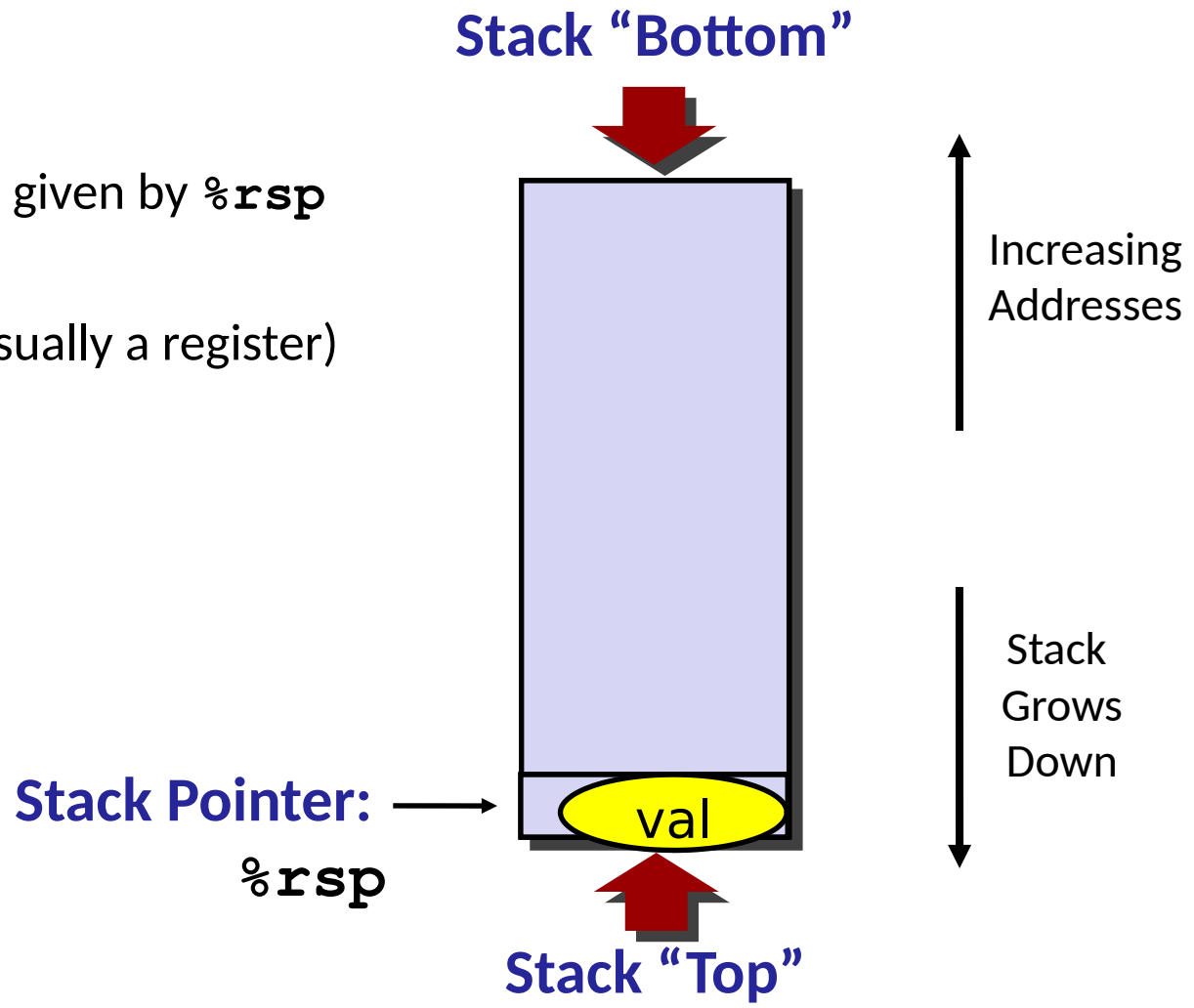
x86-64 Stack: Push

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



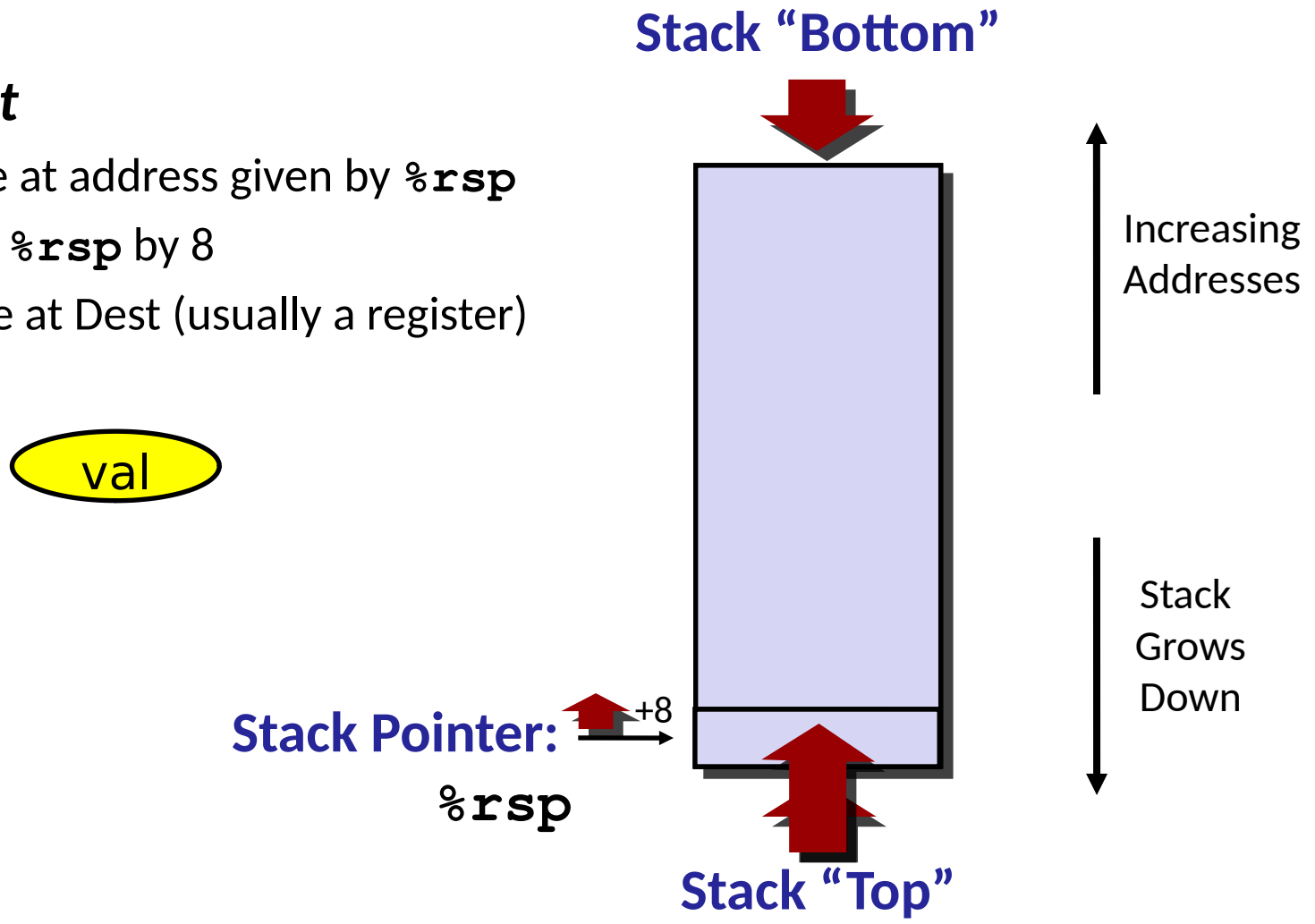
x86-64 Stack: Pop

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



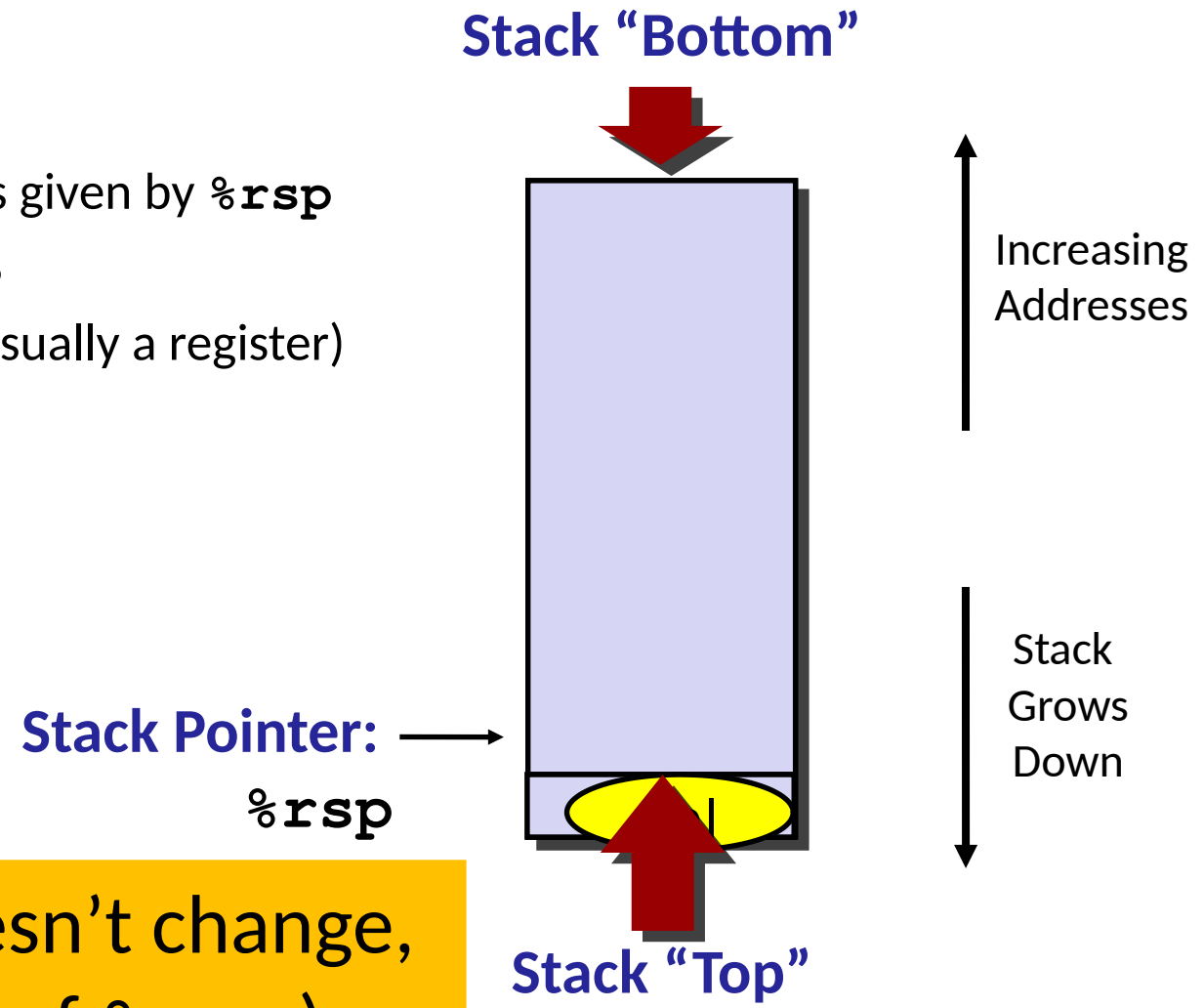
x86-64 Stack: Pop

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



x86-64 Stack: Pop

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



(The memory doesn't change, only the value of `%rsp`)

Today

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

Activity: `r m` ONLY!

Code Examples

```
void returnOne(void)
{
    return abs(-1);
}
```

```
0000000000016e9 <returnOne>:
  16e9: movl  $0xffffffff, %edi
  16ee: callq 0x1240 <abs>
  16f3: retq
```

```
int abs(int num)
{
  return num >= 0 ? num : -num;
}
```

Branches are expensive…

…but the compiler is a datalab-level hacker!

```
0000000000001240 <abs>:
  1240: movl %edi, %edx
  1242: sarl $0x1f, %edx # either 0 or -1
  1245: movl %edi, %eax
  1247: xorl %edx, %eax # num or -num-1
  1249: subl %edx, %eax # ANS-0 or ANS-(-1)
  124b: retq
```

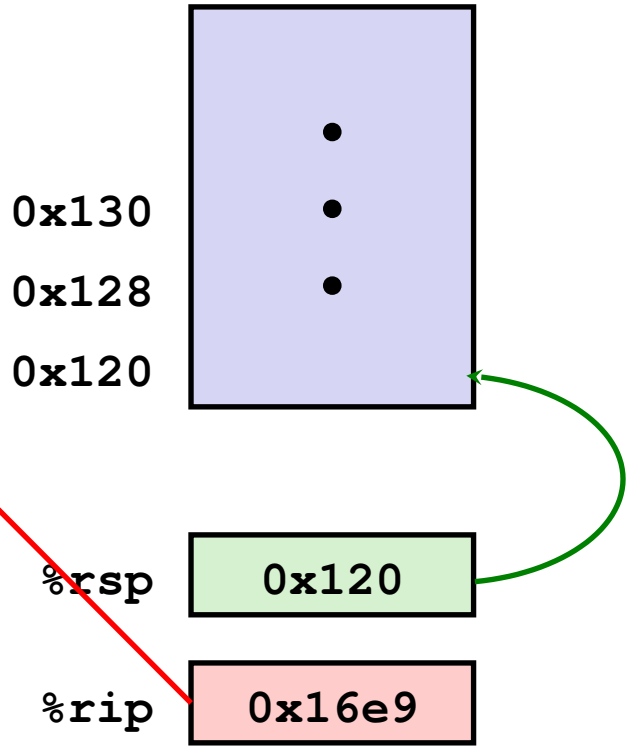
Procedure Control Flow

- **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 #0

```
00000000000016e9 <returnOne>:
  16e9: movl  $0xffffffff, %edi
  16ee: callq 0x1240 <abs>
  16f3: retq
```

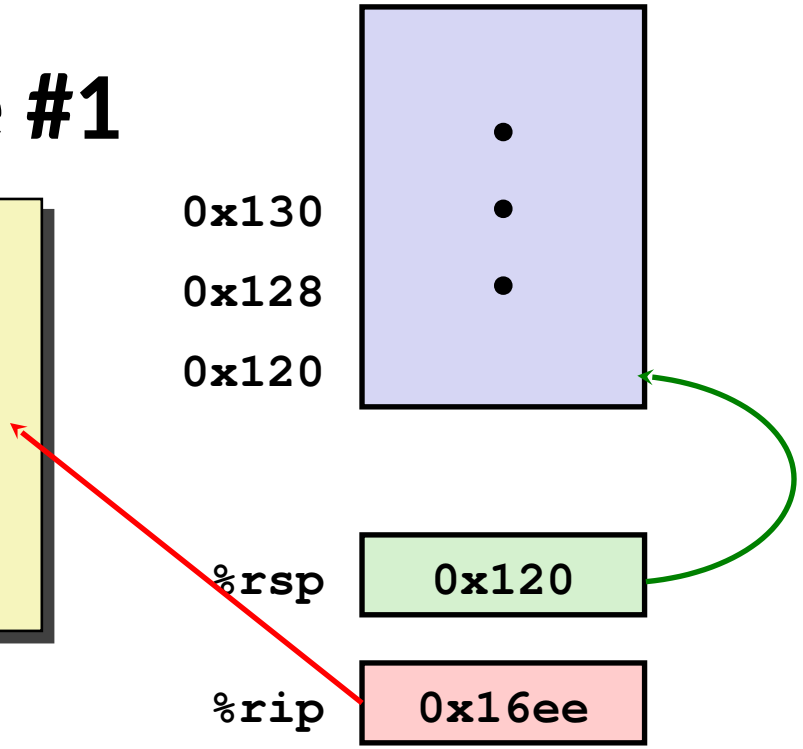
```
0000000000001240 <abs>:
  1240: movl %edi, %edx
  •
  •
  124b: retq
```



Control Flow Example #1

```
00000000000016e9 <returnOne>:
  16e9: movl  $0xffffffff, %edi
  16ee: callq 0x1240 <abs>
  16f3: retq
```

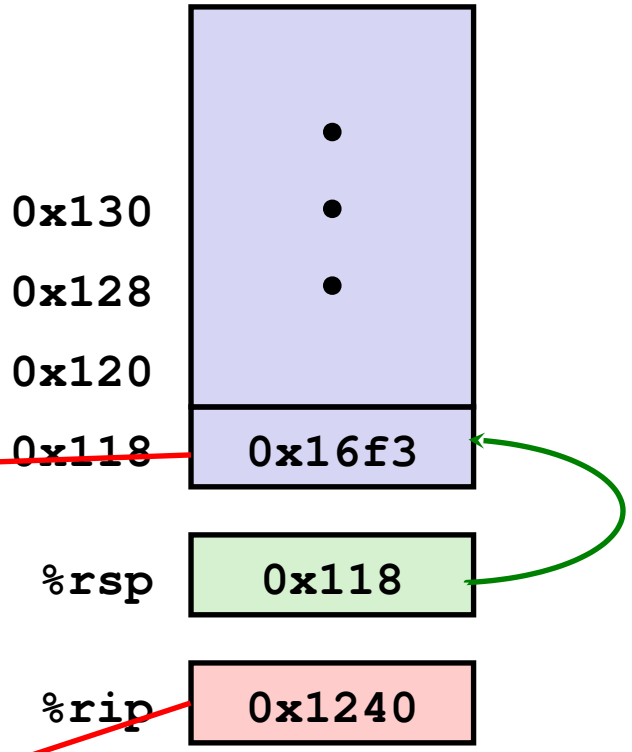
```
0000000000001240 <abs>:
  1240: movl %edi, %edx
  •
  •
  124b: retq
```



Control Flow Example #2

```
00000000000016e9 <returnOne>:
  16e9: movl  $0xffffffff, %edi
  16ee: callq 0x1240 <abs>
  16f3: retq
```

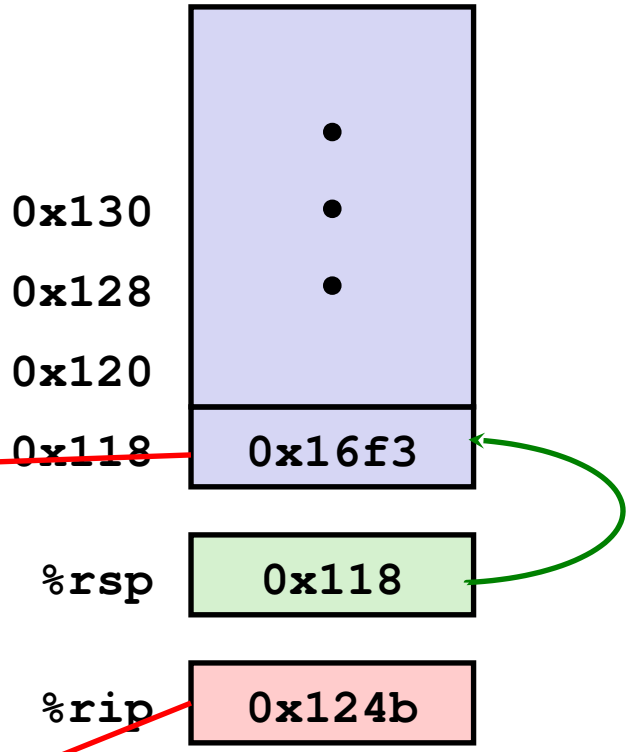
```
0000000000001240 <abs>:
  1240: movl %edi, %edx
  •
  •
  124b: retq
```



Control Flow Example #3

```
00000000000016e9 <returnOne>:
  16e9: movl  $0xffffffff, %edi
  16ee: callq 0x1240 <abs>
  16f3: retq
```

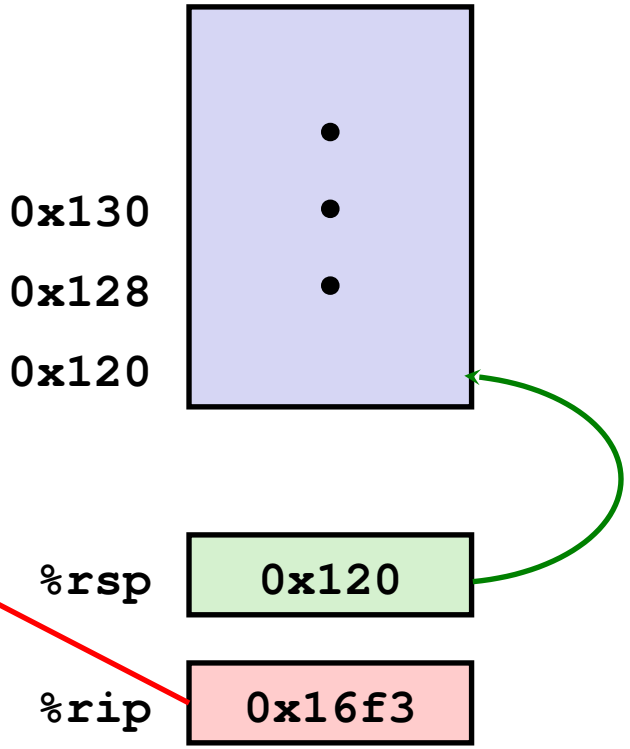
```
0000000000001240 <abs>:
  1240: movl %edi, %edx
  •
  •
  124b: retq
```



Control Flow Example #4

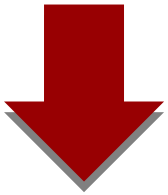
```
00000000000016e9 <returnOne>:
  16e9: movl  $0xffffffff, %edi
  16ee: callq 0x1240 <abs>
  16f3: retq
```

```
0000000000001240 <abs>:
  1240: movl %edi, %edx
  •
  •
  124b: retq
```



Tail-Call Optimization

```
00000000000016e9 <returnOne>:
  16e9: movl   $0xffffffff, %edi
  16ee: callq 0x1240 <abs>
  16f3: retq
```



```
00000000000016e9 <returnOne>:
  16e9: movl   $0xffffffff, %edi
  16ee: jmpq   0x1240 <abs>
```

Today

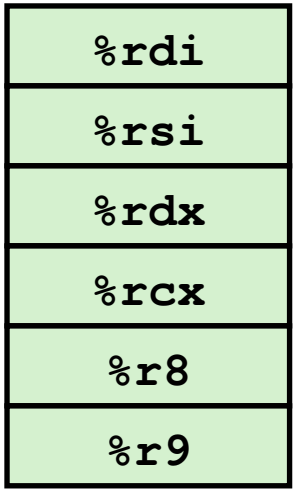
- **Procedures**
  - Stack Structure
  - **Calling Conventions**
    - Passing control
    - **Passing data**
    - Managing local data
  - Illustrations of Recursion & Pointers

Activity: `r z` ONLY!

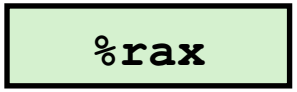
Procedure Data Flow

Registers

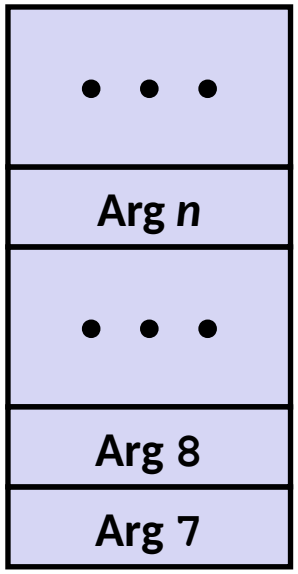
- First 6 arguments



- Return value



Stack



- Only allocate stack space when needed

Data Flow Example

```
void seeMoreArgs(void) {
  printf("%d %d %d %d %d %d %d %d", 2, 3, 4, 5, 6, 7, 8);
}
```

```
000000000000118a <seeMoreArgs>:
   118a <+0>:     sub    $0x8,%rsp
   118e <+4>:     pushq  $0x8
   1190 <+6>:     pushq  $0x7
   1192 <+8>:     mov    $0x6,%r9d
   1198 <+14>:    mov    $0x5,%r8d
   119e <+20>:    mov    $0x4,%ecx
   11a3 <+25>:    mov    $0x3,%edx
   11a8 <+30>:    mov    $0x2,%esi
   11ad <+35>:    lea    0xe50(%rip),%rdi
   11b4 <+42>:    mov    $0x0,%eax
   11b9 <+47>:    callq  0x1050 <printf@plt>
   11be <+52>:    add    $0x18,%rsp
   11c2 <+56>:    retq
```

Data Flow Example

```
int seeMoreArgs(void) {
  return printf("%d %d %d %d %d %d %d %d", 2, 3, 4, 5, 6, 7, 8);
}
```

```
000000000000118a <seeMoreArgs>:
   118a <+0>:     sub    $0x8,%rsp
   118e <+4>:     pushq  $0x8
   1190 <+6>:     pushq  $0x7
   1192 <+8>:     mov    $0x6,%r9d
   1198 <+14>:    mov    $0x5,%r8d
   119e <+20>:    mov    $0x4,%ecx
   11a3 <+25>:    mov    $0x3,%edx
   11a8 <+30>:    mov    $0x2,%esi
   11ad <+35>:    lea    0xe50(%rip),%rdi
   11b4 <+42>:    mov    $0x0,%eax
   11b9 <+47>:    callq  0x1050 <printf@plt>
   11be <+52>:    add    $0x18,%rsp
   11c2 <+56>:    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

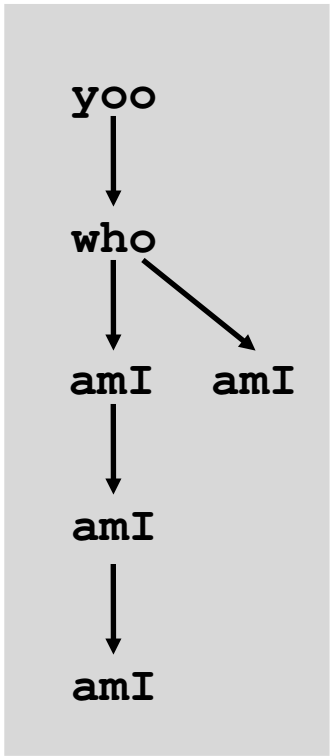
Call Chain Example

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

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

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

Example Call Chain

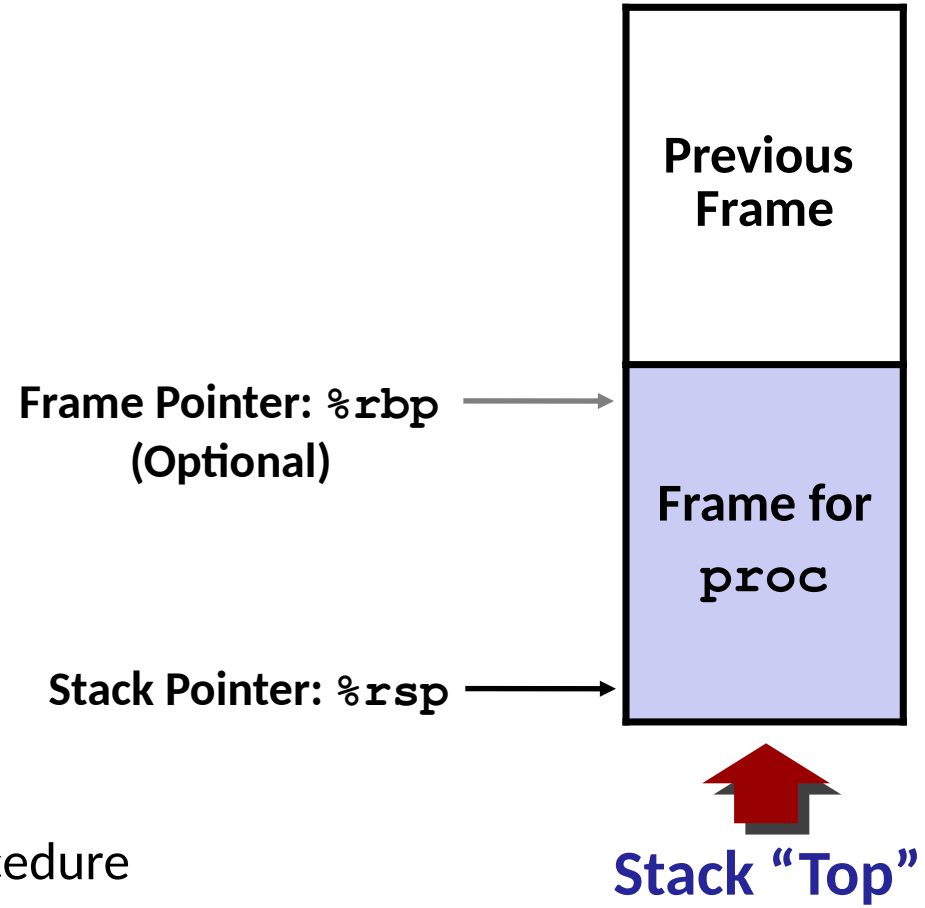


Procedure `amI()` is recursive

Stack Frames

Contents

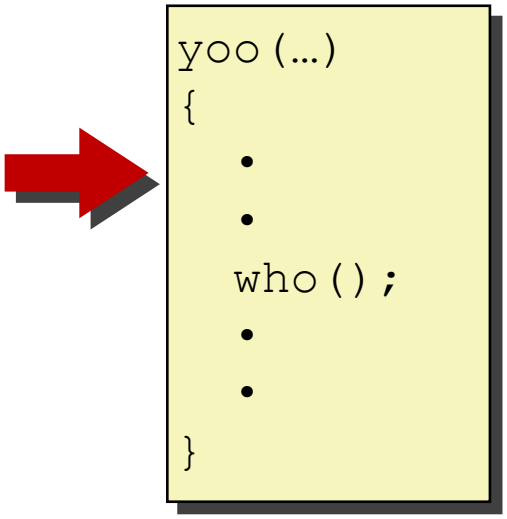
- Return information
- Local storage (if needed)
- Temporary space (if needed)

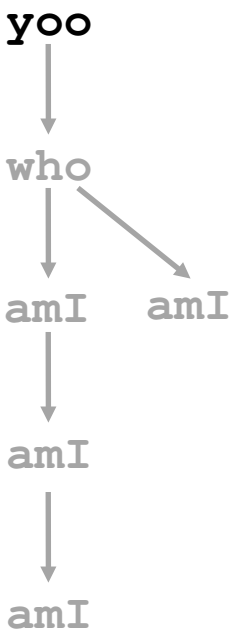


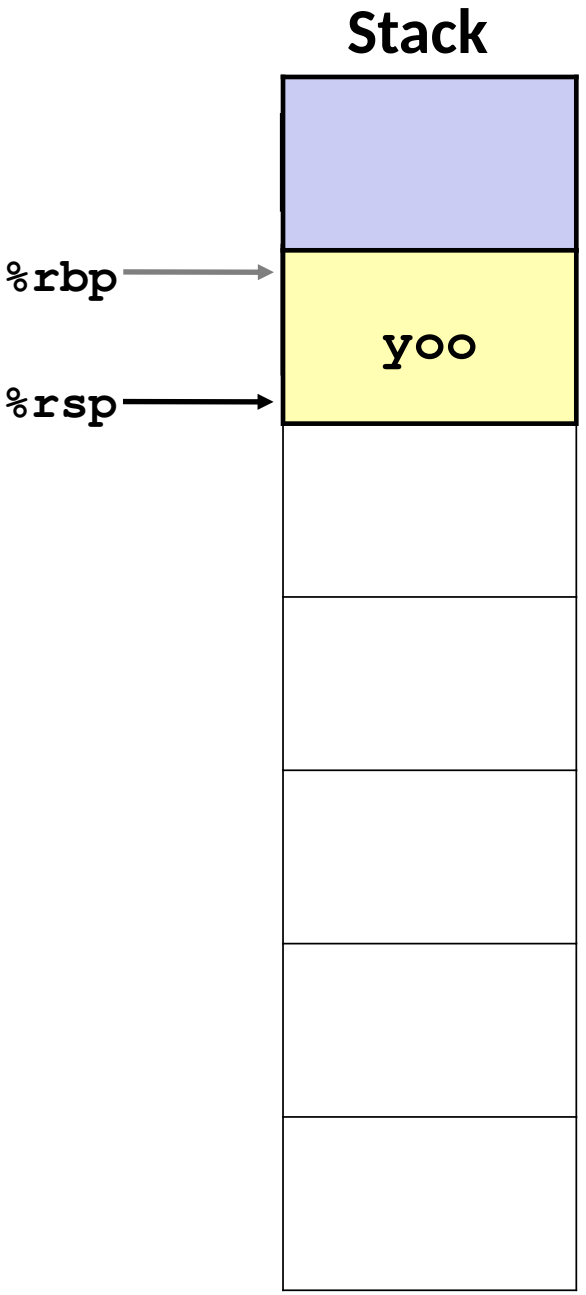
Management

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

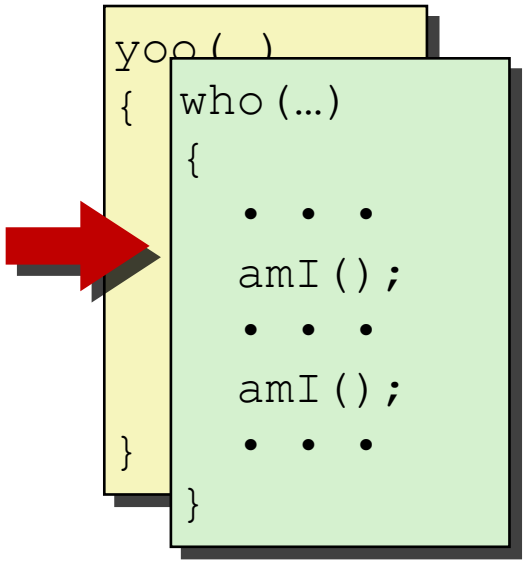
Example

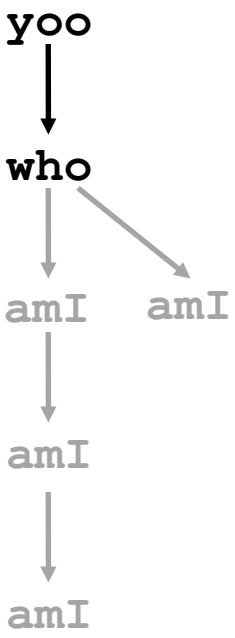


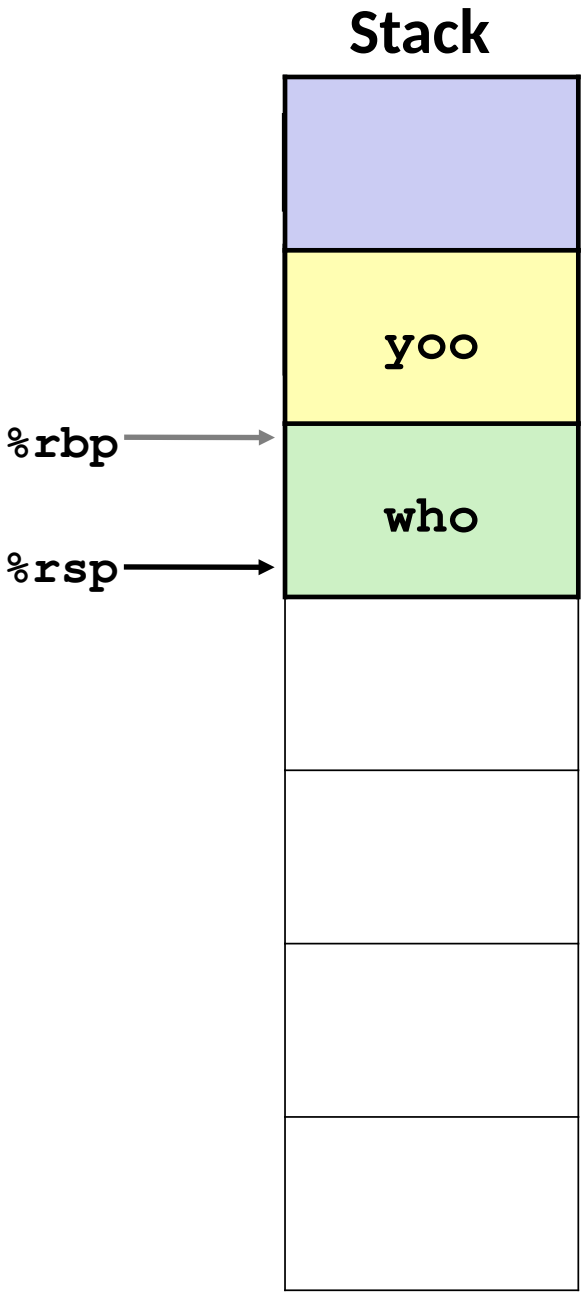




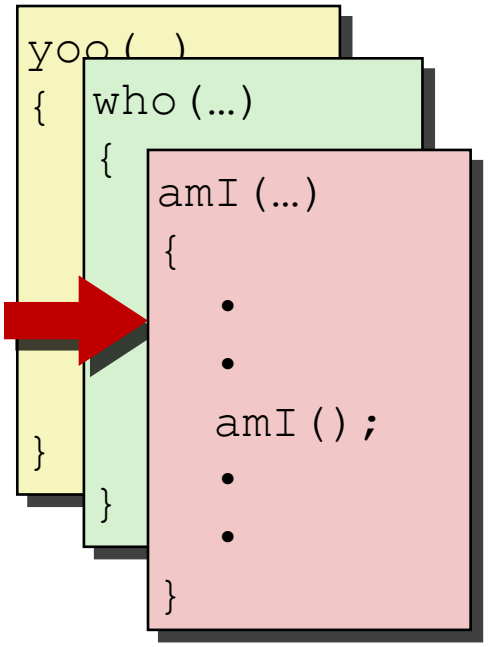
Example

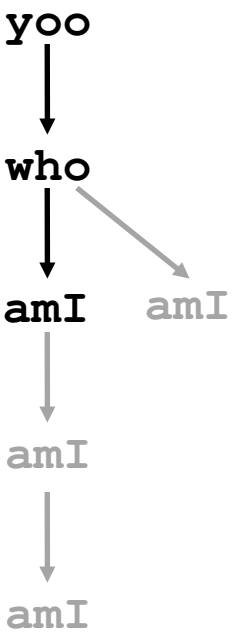


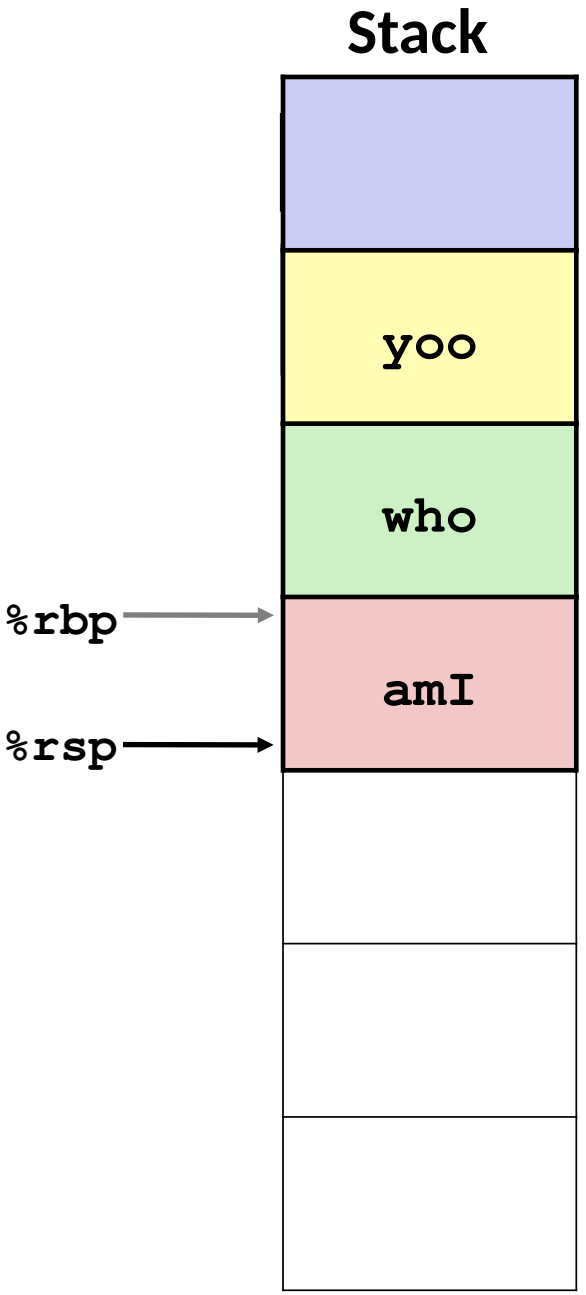




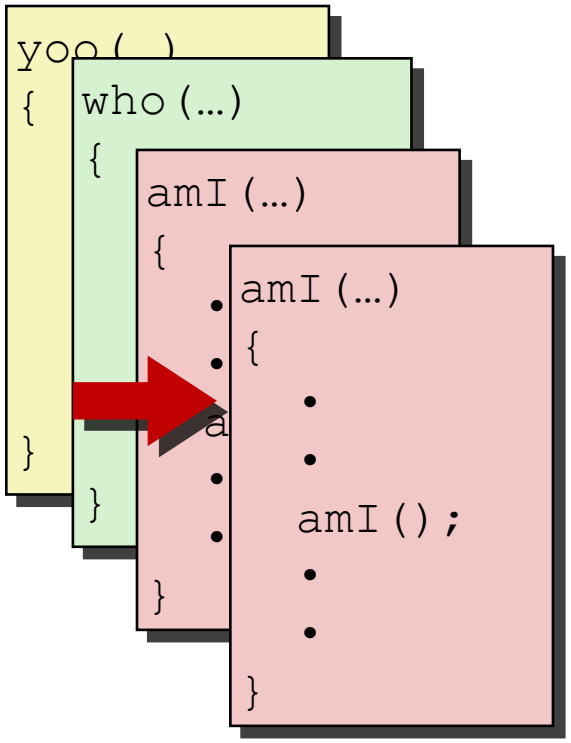
Example

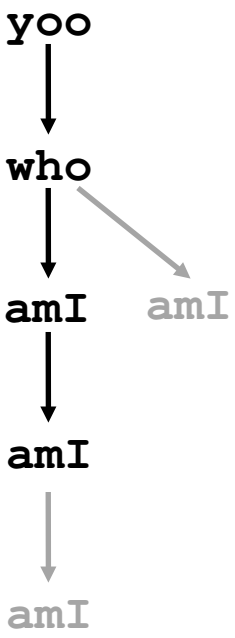

yoo( )
{
who(…)
{
amI(…)
{
•
•
amI();
•
•
}
}
}

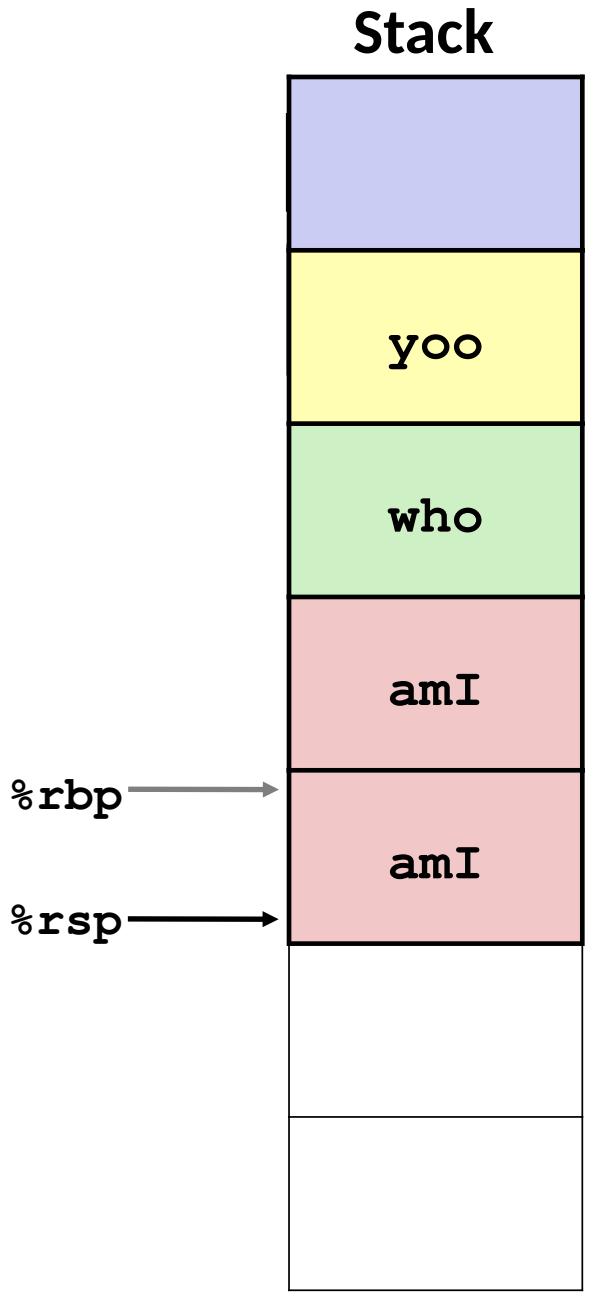

yoo
who
amI
amI
amI
amI


Stack
yoo
who
amI
%rbp
%rsp

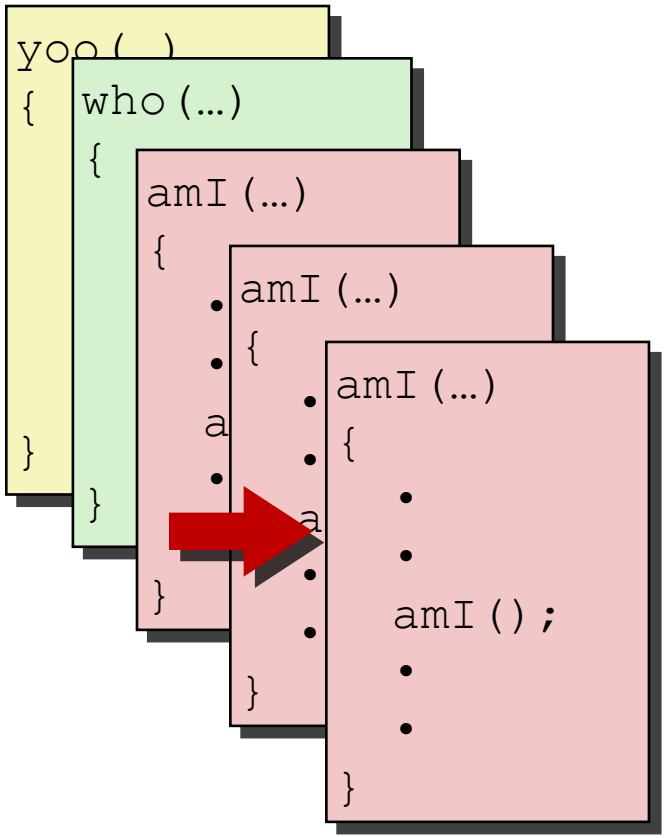
Example

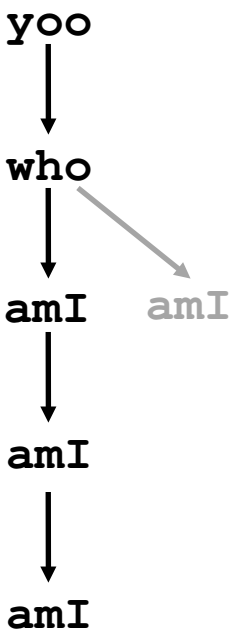

yoo( )
{
who(…)
{
amI(…)
{
amI(…)
{
amI();
}
}
}
}

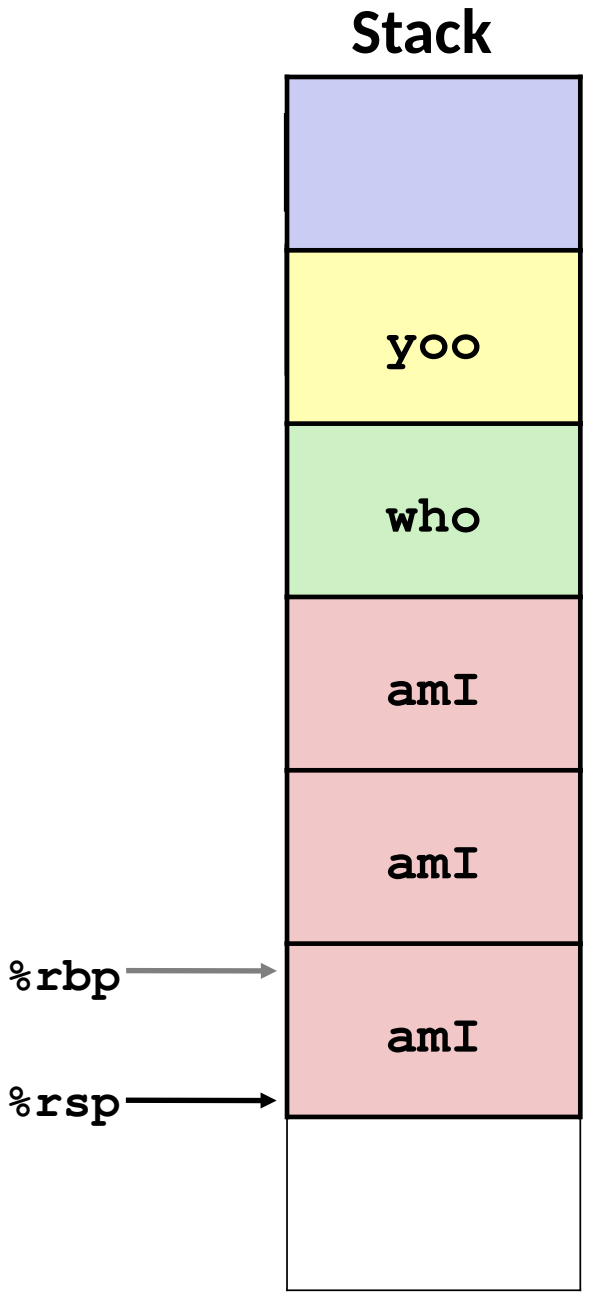

yoo
who
amI
amI
amI
amI


Stack
yoo
who
amI
amI
amI
%rbp
%rsp

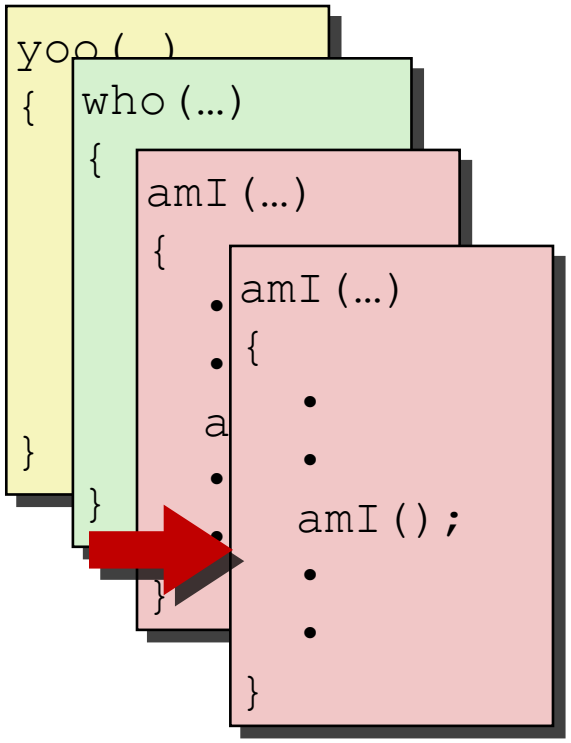
Example

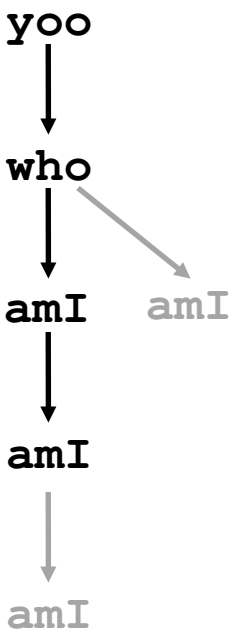

yoo( )
{
who(…)
{
amI(…)
{
amI(…)
{
amI(…)
{
amI();
}
}
}
}
}

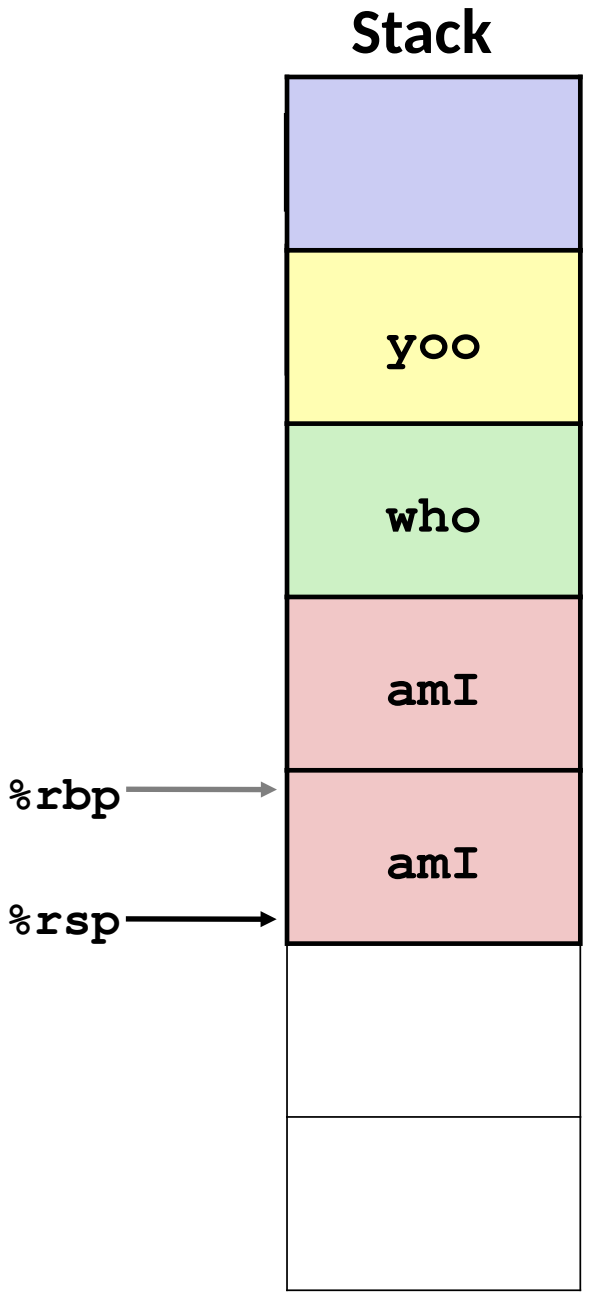

yoo
who
amI
amI
amI
amI


Stack
yoo
who
amI
amI
amI
%rbp
%rsp

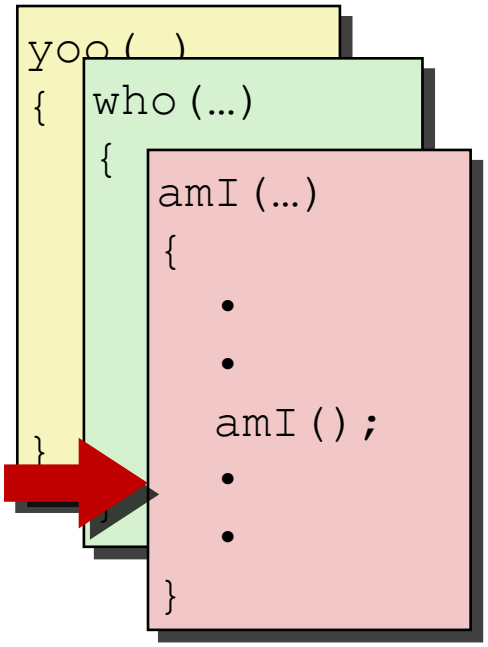
Example

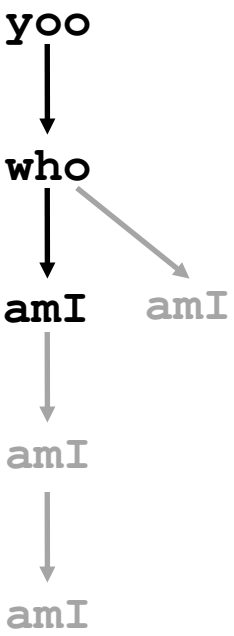

yoo( )
{
who(…)
{
amI(…)
{
amI(…)
{
amI();
}
}
}

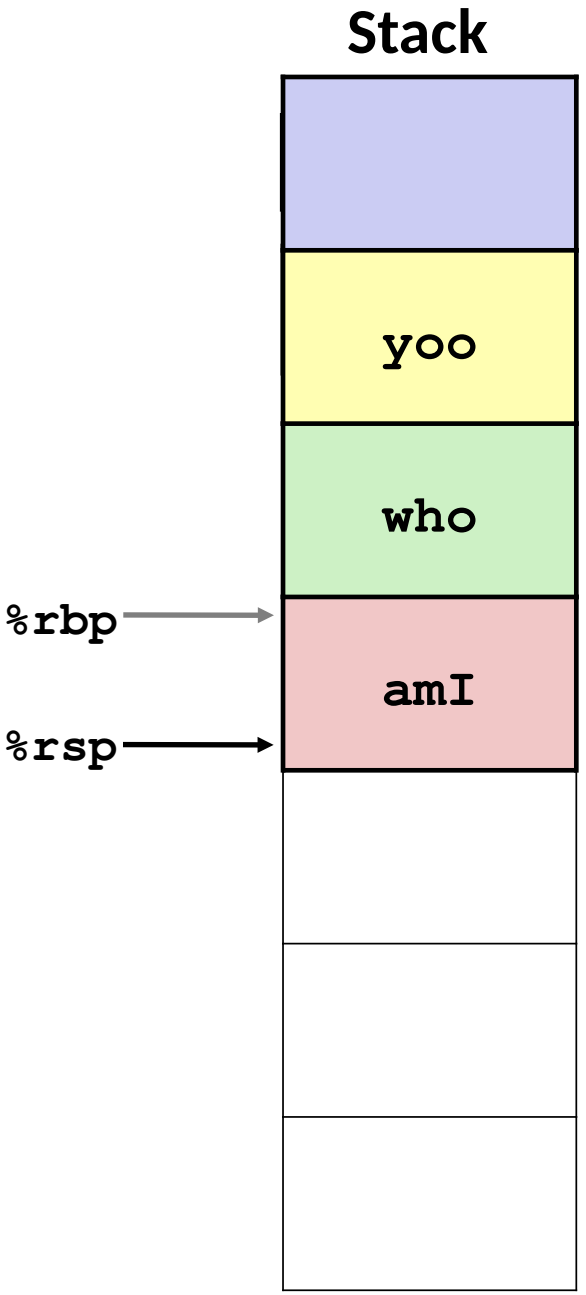

yoo
who
amI
amI
amI
amI


Stack
yoo
who
amI
amI
%rbp
%rsp

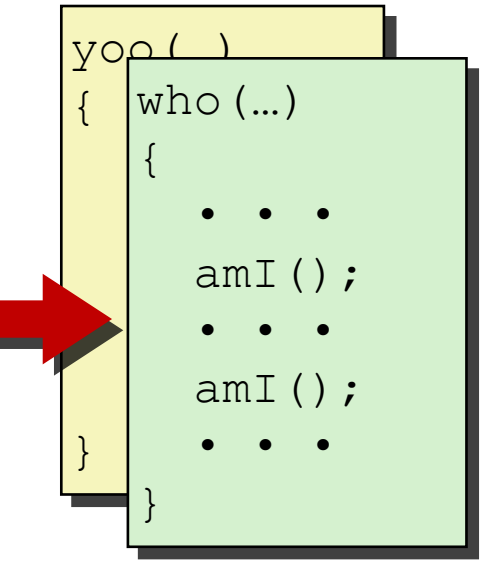
Example

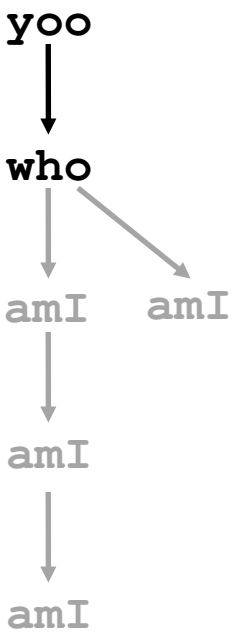

yoo( )
{
who(…)
{
amI(…)
{
•
•
amI();
•
•
}

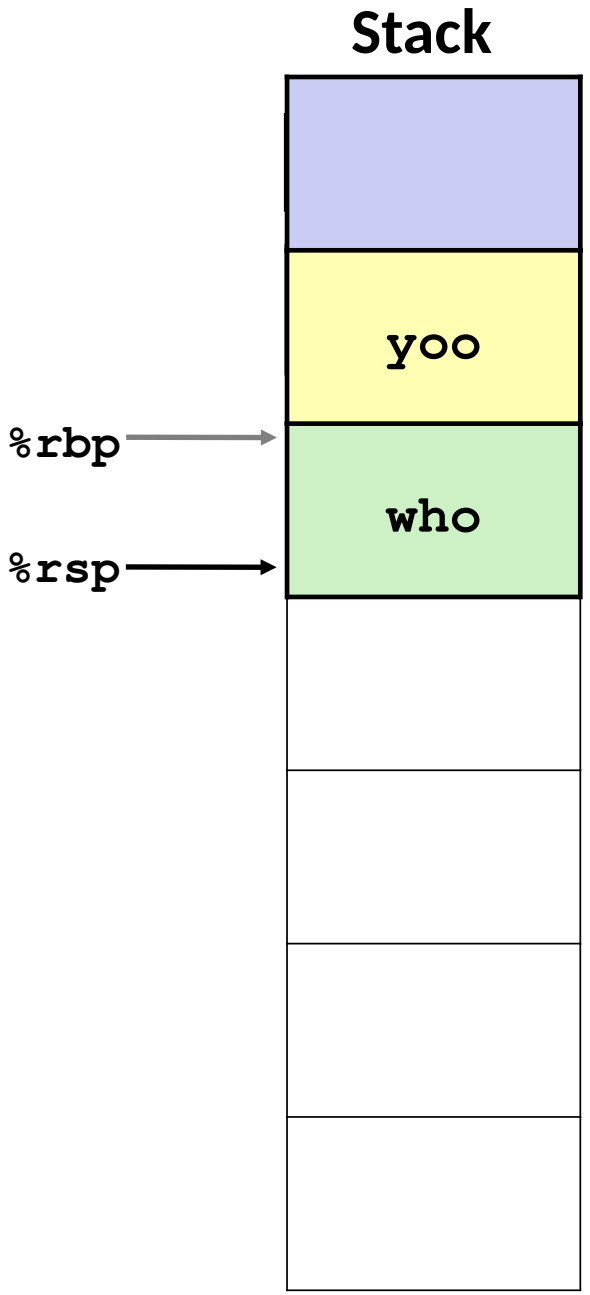

yoo
who
amI
amI
amI
amI


Stack
yoo
who
amI
%rbp
%rsp

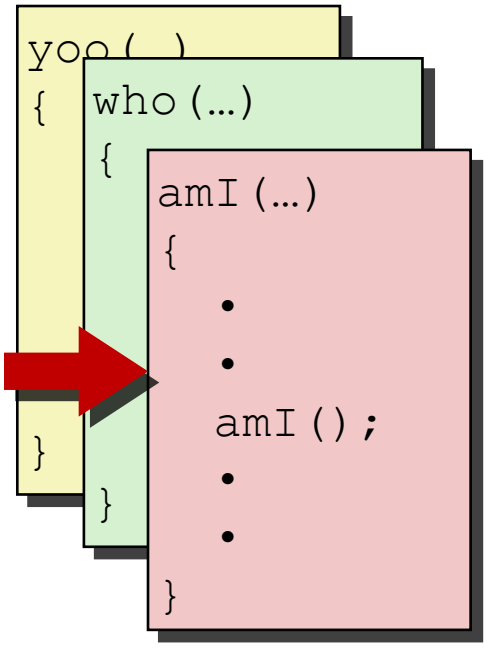
Example

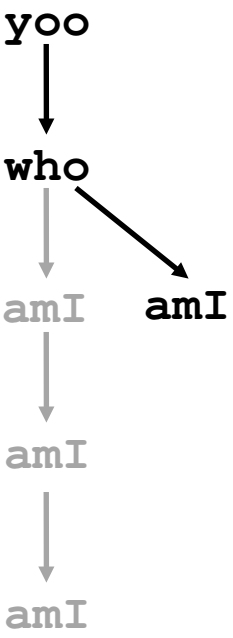


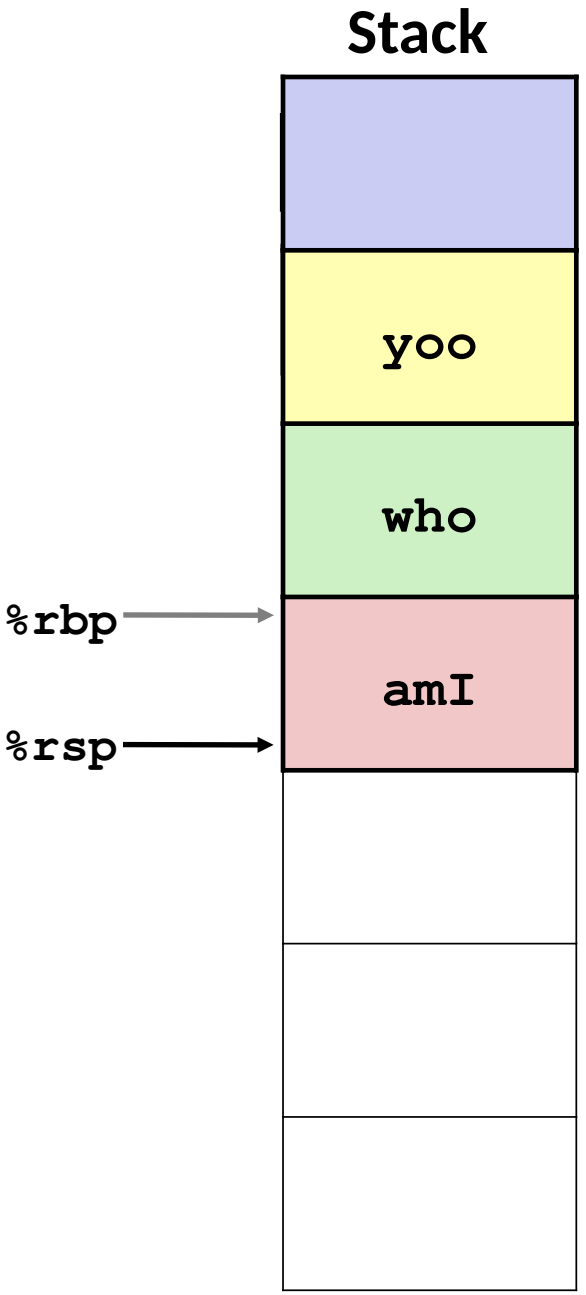




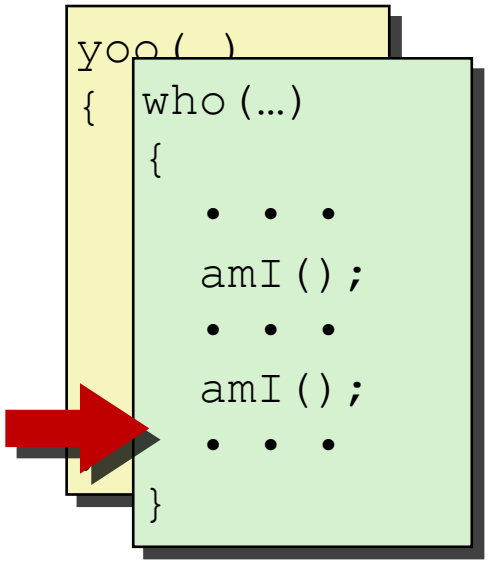
Example

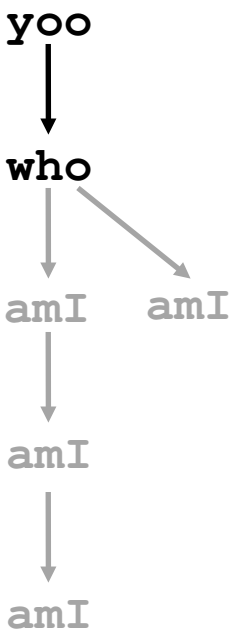

yoo( )
{
who(…)
{
amI(…)
{
•
•
amI();
•
•
}
}
}

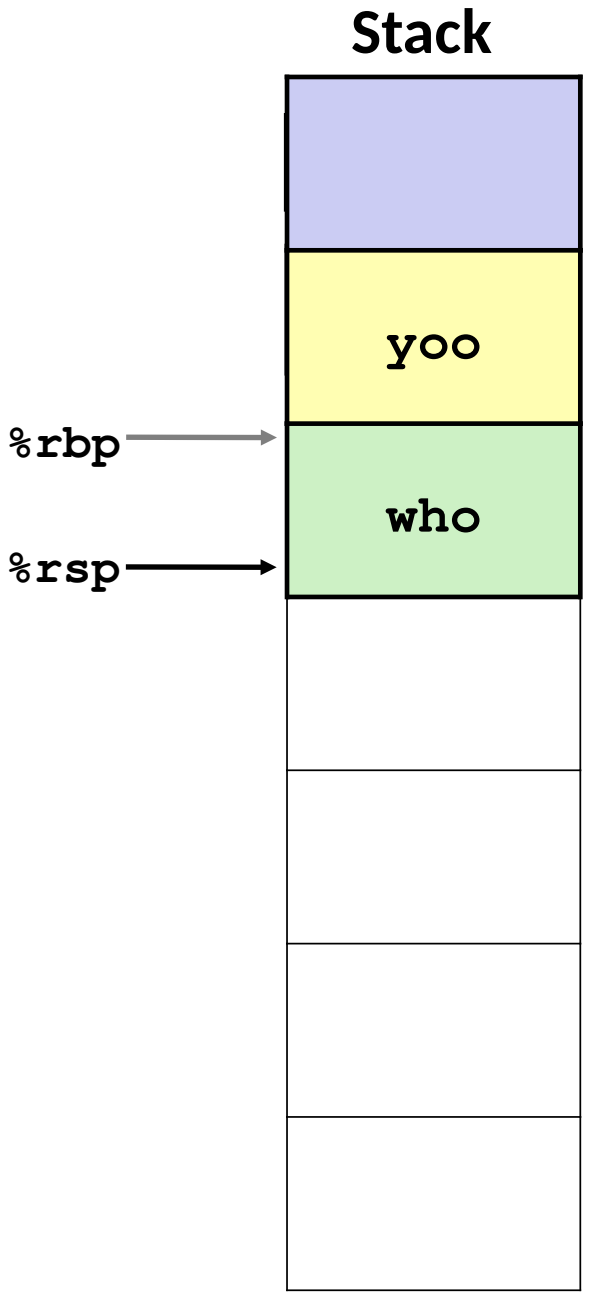

yoo
who
amI
amI
amI
amI


Stack
yoo
who
amI
%rbp
%rsp

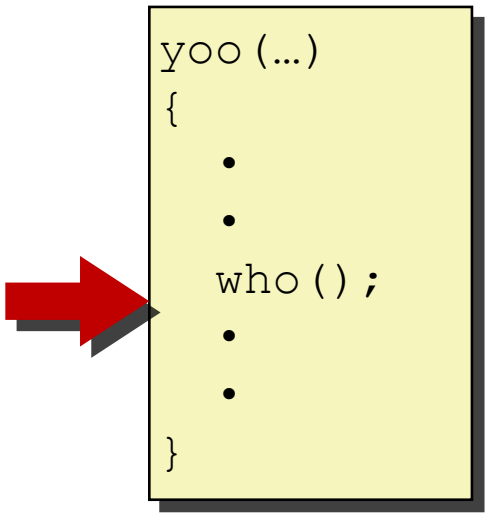
Example

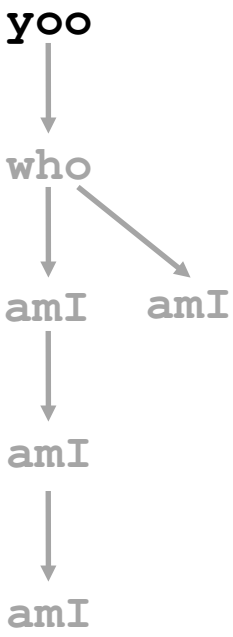


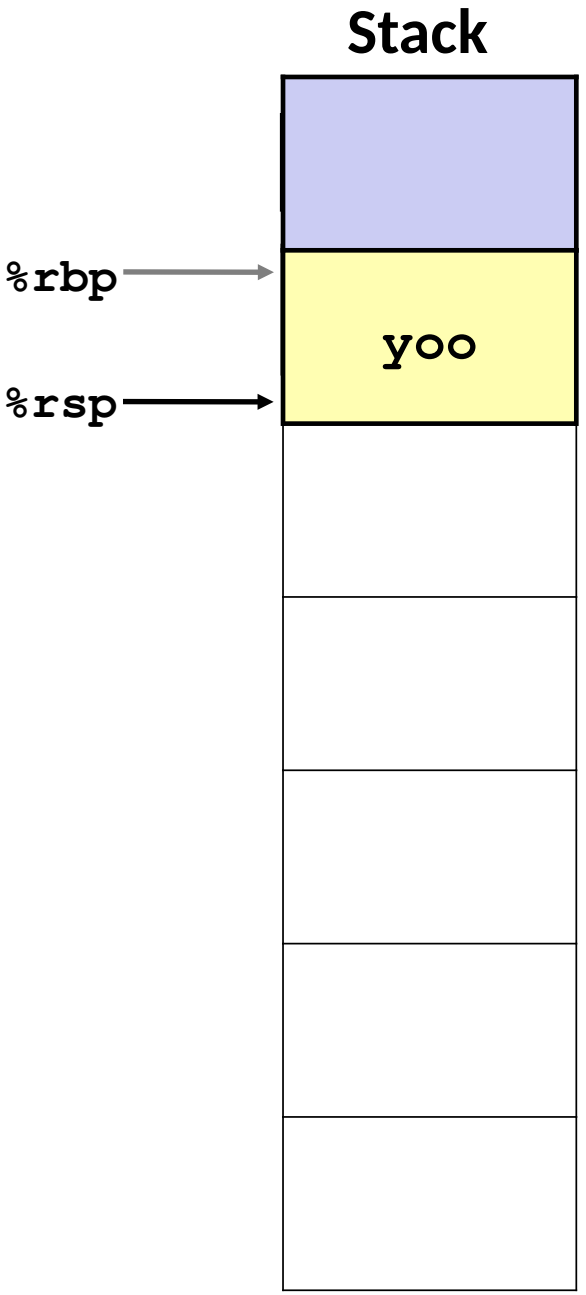




Example







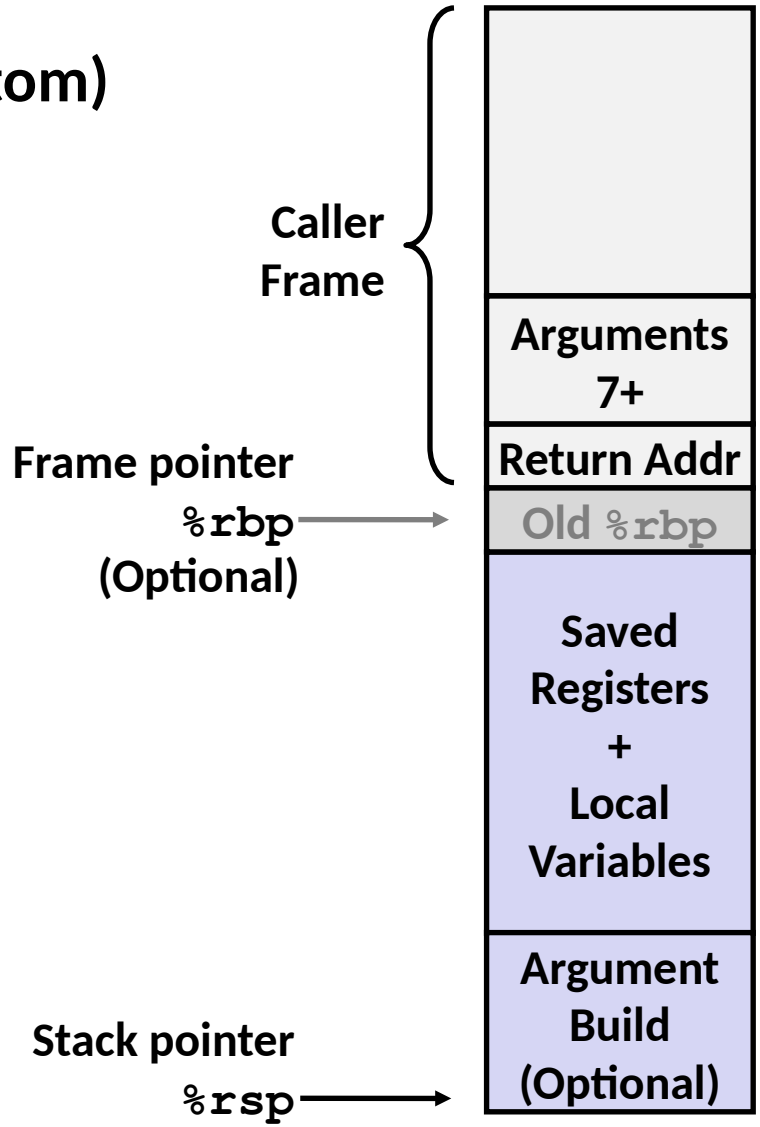
x86-64/Linux Stack Frame

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



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
```

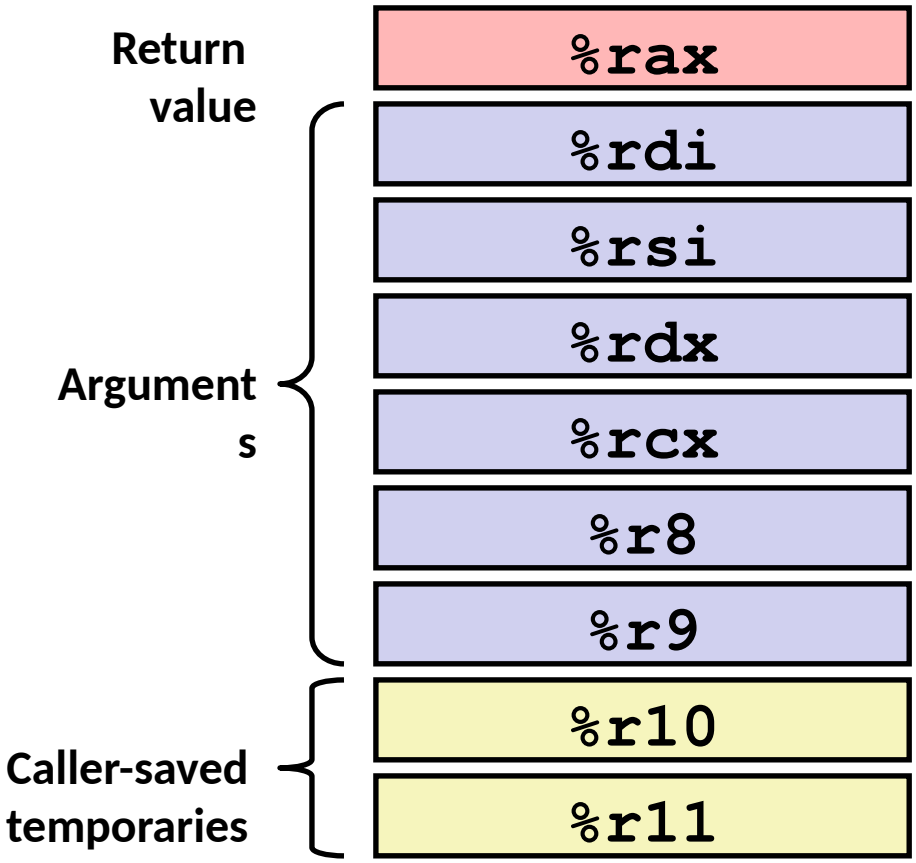
  - Contents of register `%rdx` overwritten by `who`
  - This could be trouble → something should be done!
    - Need some coordination

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

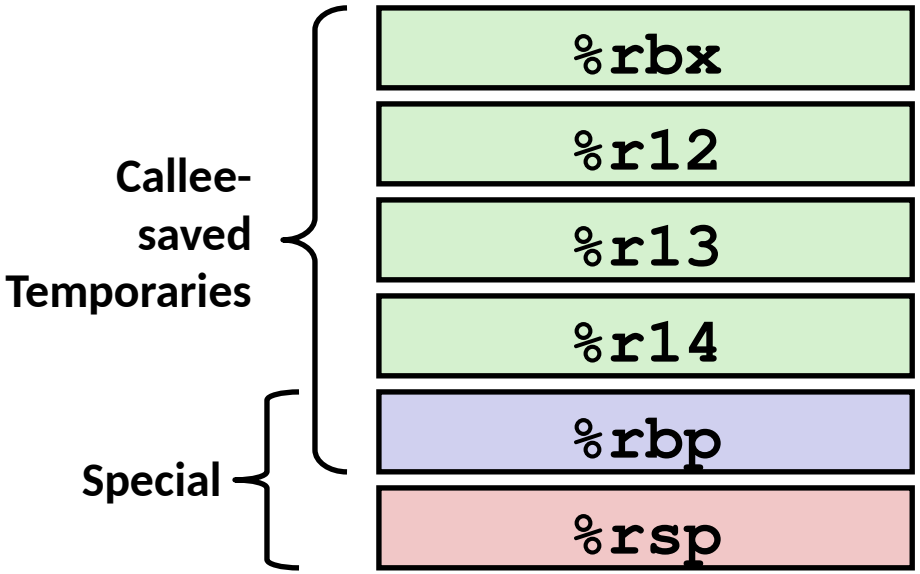
x86-64 Linux Register Usage #1

- `%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



Activity: ALL of `act7` and quiz

Callee-Saved Registers Example

```
void mult4(int a, int b, int c, int d) { return (a*b)*(c*d); }
```

```
0000000000001428 <mult4>:
   1428 <+0>:     push   %rbx
   1429 <+1>:     push   %r12
   142b <+3>:     push   %r13
   142d <+5>:     mov    %edx,%r12d
   1430 <+8>:     mov    %ecx,%r13d
   1433 <+11>:    callq  0x1422 <mult2>
   1438 <+16>:    mov    %r12d,%edi
   143b <+19>:    mov    %r13d,%esi
   143e <+22>:    mov    %eax,%ebx
   1440 <+24>:    callq  0x1422 <mult2>
   1445 <+29>:    mov    %eax,%edi
   1447 <+31>:    mov    %ebx,%esi
   1449 <+33>:    callq  0x1422 <mult2>
   144e <+38>:    pop    %r13
   1450 <+40>:    pop    %r12
   1452 <+42>:    pop    %rbx
   1453 <+43>:    retq
```

Callee-Saved Registers Example

```
void mult4(int a, int b, int c, int d) { return (a*b)*(c*d); }
```

```
0000000000001428 <mult4>:
   1428 <+0>:     push   %rbx
   1429 <+1>:     push   %r12
   142b <+3>:     push   %r13
   142d <+5>:     mov    %edx,%r12d
   1430 <+8>:     mov    %ecx,%r13d
   1433 <+11>:    callq  0x1422 <mult2>
   1438 <+16>:    mov    %r12d,%edi
   143b <+19>:    mov    %r13d,%esi
   143e <+22>:    mov    %eax,%ebx
   1440 <+24>:    callq  0x1422 <mult2>
   1445 <+29>:    mov    %eax,%edi
   1447 <+31>:    mov    %ebx,%esi
   1449 <+33>:    callq  0x1422 <mult2>
   144e <+38>:    pop    %r13
   1450 <+40>:    pop    %r12
   1452 <+42>:    pop    %rbx
   1453 <+43>:    retq
```

Local Array Example

```
void getV(int i) {
  int x[4];
  return getValue(x, i);
}
```

```
0000000000001346 <getV>:
   1346 <+0>:     sub    $0x18,%rsp
   134a <+4>:     mov    %edi,%esi
   134c <+6>:     mov    %rsp,%rdi
   134f <+9>:     callq  0x145c <getValue>
   1354 <+14>:    add    $0x18,%rsp
   1358 <+18>:    retq
```

Local Array Example

```
void getV(int i) {
  int x[4];
  return getValue(x, i);
}
```

```
0000000000001346 <getV>:
   1346 <+0>:     sub    $0x18,%rsp
   134a <+4>:     mov    %edi,%esi
   134c <+6>:     mov    %rsp,%rdi
   134f <+9>:     callq  0x145c <getValue>
   1354 <+14>:    add    $0x18,%rsp
   1358 <+18>:    retq
```

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
  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
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep; ret
```

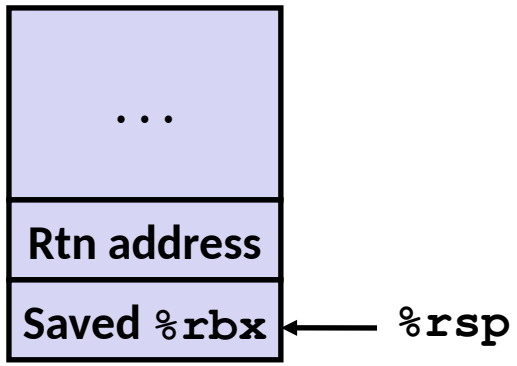
Register	Use(s)	Type
%rdi	x	Argument
%rax	Return value	Return value

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
  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
  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
  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
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep; ret
```

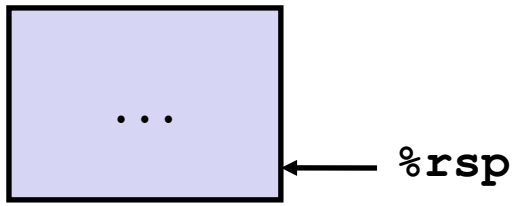
Register	Use(s)	Type
%rbx	x & 1	Callee-saved
%rax	Return value	

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
  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

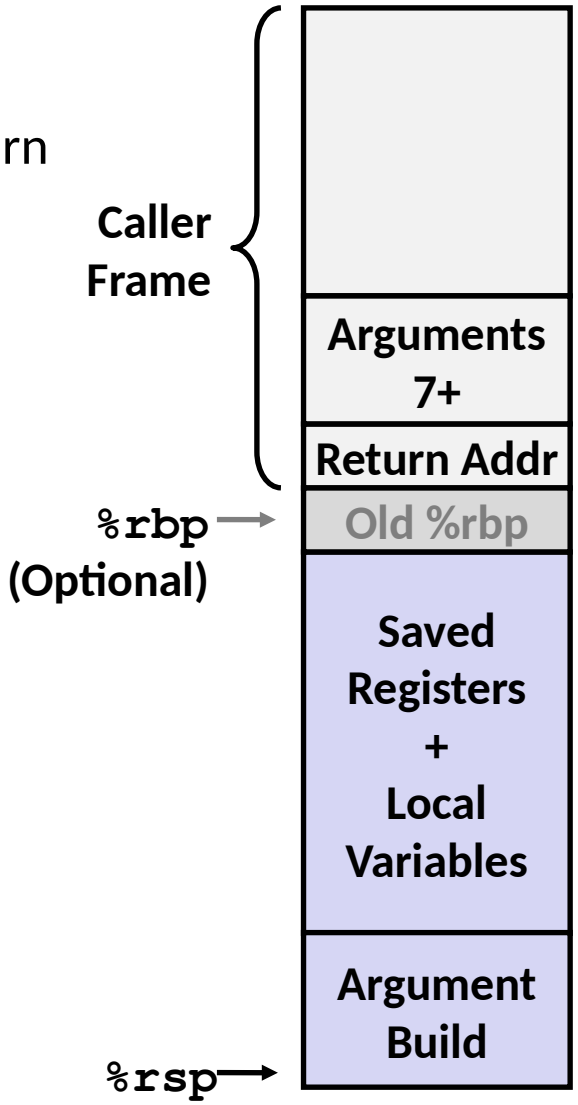
- Stack frames mean that each function call has private storage
  - Saved registers & local variables
  - Saved return pointer
- Register saving conventions prevent one function call from corrupting another's data
  - Unless the C code explicitly does so (e.g., buffer overflow in Lecture 9)
- Stack discipline 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**



Appendix

Example: `incr`

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

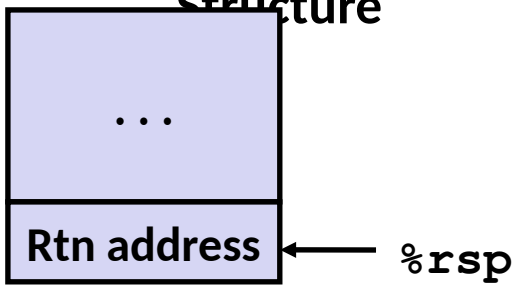
```
incr:
  movq    (%rdi), %rax
  addq    %rax, %rsi
  movq    %rsi, (%rdi)
  ret
```

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

Example: Calling `incr` #1

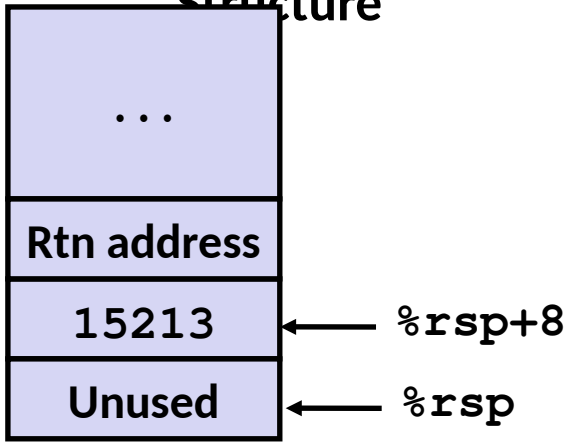
```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

Initial Stack Structure



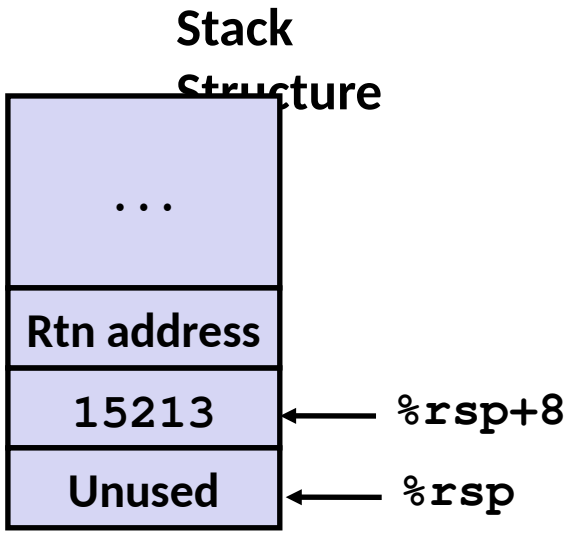

```
call_incr:
  subq    $16, %rsp
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

Resulting Stack Structure



Example: Calling `incr` #2

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

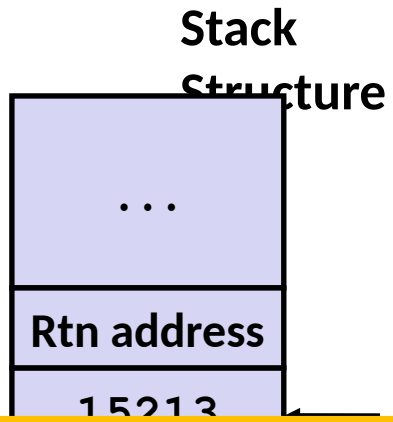



```
call_incr:
  subq    $16, %rsp
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

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

Example: Calling `incr` #2

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



Aside 1: **`movl $3000, %esi`**

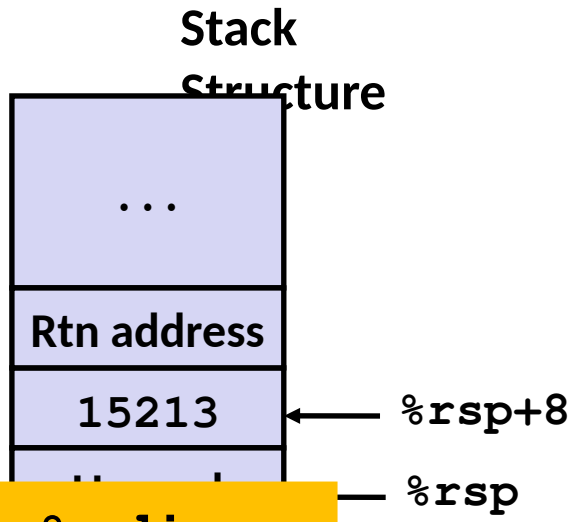
- Remember, movl -> %exx zeros out high order 32 bits.
- Why use movl instead of movq? 1 byte shorter.

```
ca
    movl    $3000, %esi
    leaq    8(%rsp), %rdi
    call    incr
    addq    8(%rsp), %rax
    addq    $16, %rsp
    ret
```

%rdi	&v1
%rsi	3000

Example: Calling `incr` #2

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



Aside 2: `leaq 8(%rsp), %rdi`

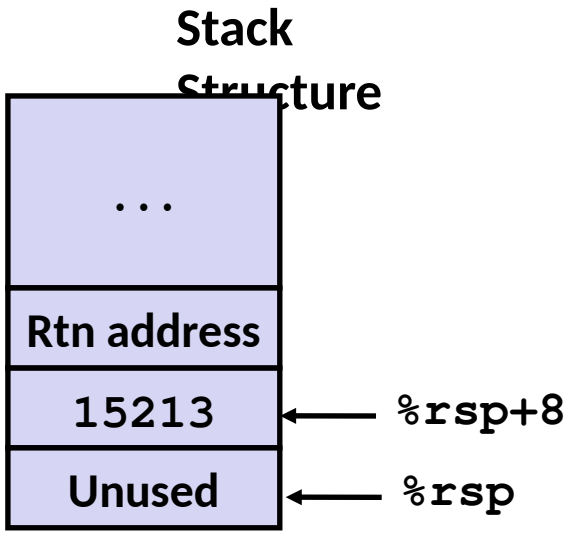
- Computes %rsp+8
- Actually used for what it is meant!

```
ca
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

se(s)
v1
%rsi	3000

Example: Calling `incr` #2

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

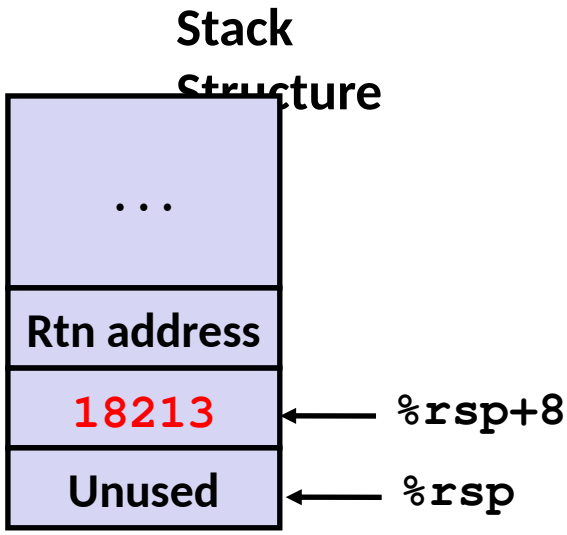



```
call_incr:
  subq    $16, %rsp
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

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;
}
```

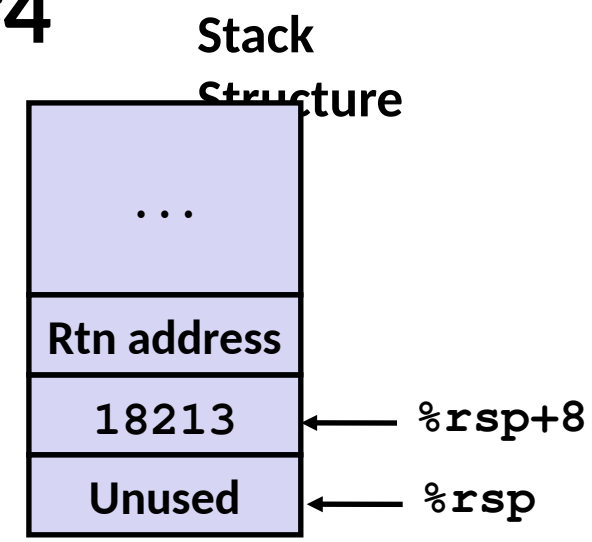



```
call_incr:
  subq    $16, %rsp
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

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;
}
```



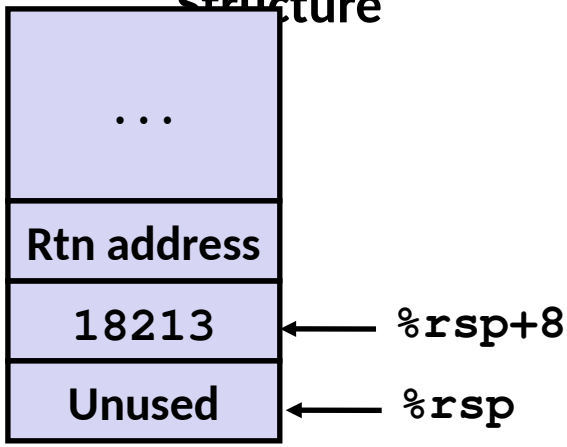

```
call_incr:
  subq    $16, %rsp
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

Register	Use(s)
%rax	Return value

Example: Calling `incr` #5a

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

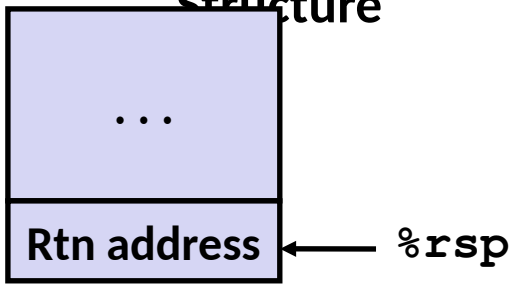
Stack Structure




```
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

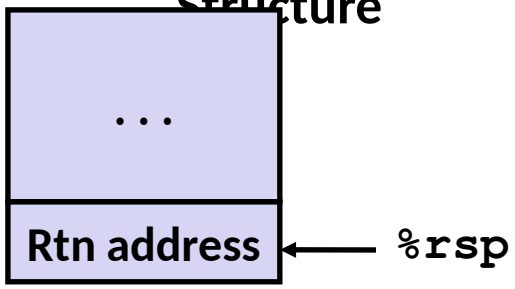
Updated Stack Structure



Example: Calling `incr` #5b

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

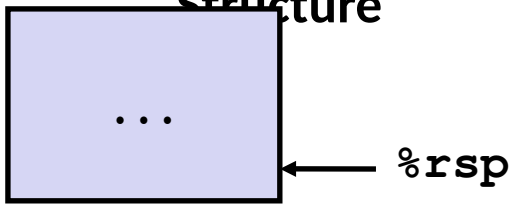
Updated Stack Structure




```
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

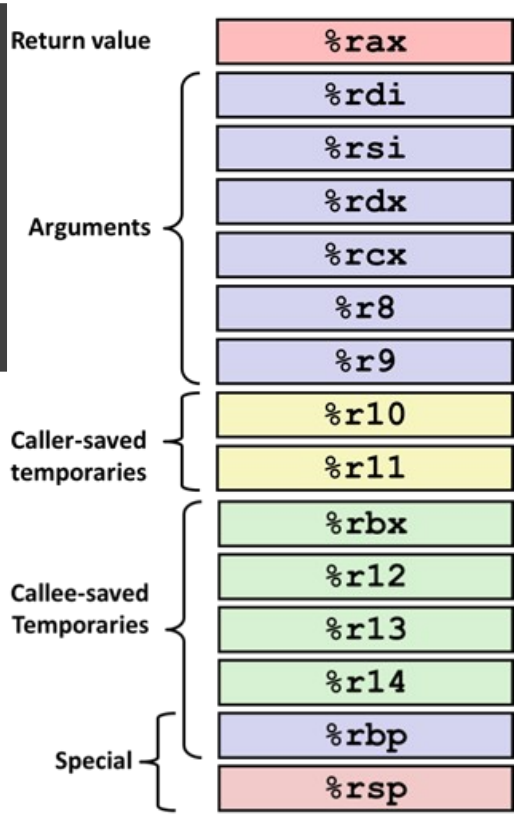
Final Stack Structure



Small Exercise

```
long add5(long b0, long b1, long b2, long b3, long b4) {
    return b0+b1+b2+b3+b4;
}

long add10(long a0, long a1, long a2, long a3, long a4, long a5,
    long a6, long a7, long a8, long a9) {
    return add5(a0, a1, a2, a3, a4)+
        add5(a5, a6, a7, a8, a9);
}
```



- Where are `a0`,…, `a9` passed?

  `rdi`, `rsi`, `rdx`, `rcx`, `r8`, `r9`, stack

- Where are `b0`,…, `b4` passed?

  `rdi`, `rsi`, `rdx`, `rcx`, `r8`

- Which registers do we need to save?

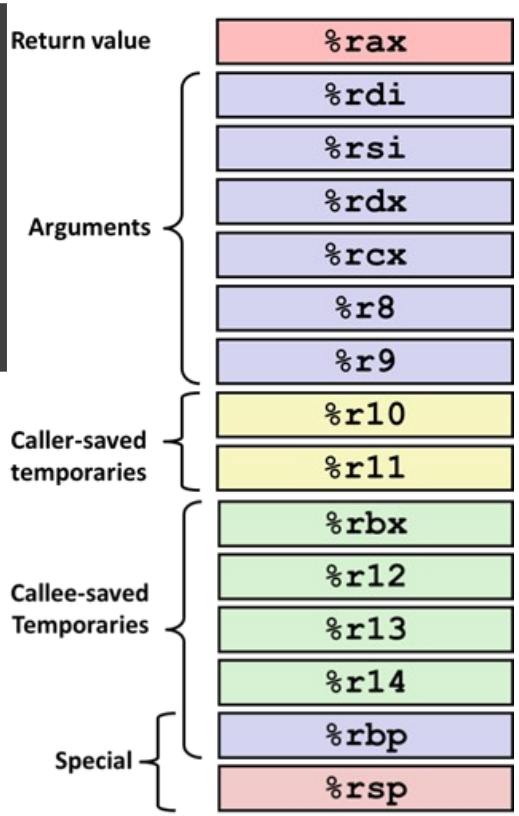
  Ill-posed question. Need assembly.

  `rbx`, `rbp`, `r9` (during first call to `add5`)

Small Exercise

```
long add5(long b0, long b1, long b2, long b3, long b4) {
    return b0+b1+b2+b3+b4;
}

long add10(long a0, long a1, long a2, long a3, long a4, long a5,
    long a6, long a7, long a8, long a9) {
    return add5(a0, a1, a2, a3, a4)+
        add5(a5, a6, a7, a8, a9);
}
```

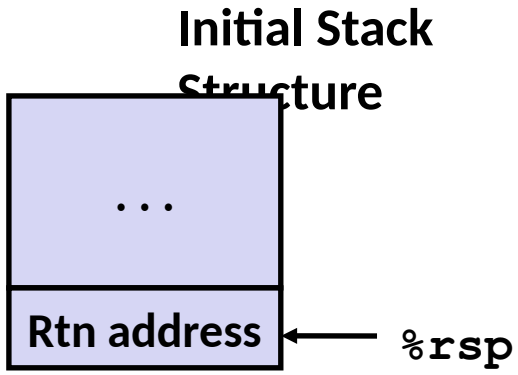



```
add10:
        pushq   %rbp
        pushq   %rbx
        movq    %r9, %rbp
        call    add5
        movq    %rax, %rbx
        movq    48(%rsp), %r8
        movq    40(%rsp), %rcx
        movq    32(%rsp), %rdx
        movq    24(%rsp), %rsi
        movq    %rbp, %rdi
        call    add5
        addq    %rbx, %rax
        popq    %rbx
        popq    %rbp
        ret
```

```
add5:
        addq    %rsi, %rdi
        addq    %rdi, %rdx
        addq    %rdx, %rcx
        leaq    (%rcx,%r8), %rax
        ret
```

Callee-Saved Example #1

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



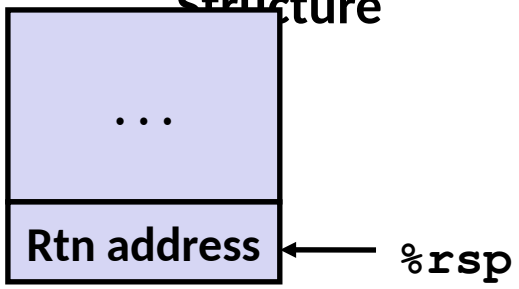
- X comes in register **`%rdi`**.
- We need **`%rdi`** for the call to incr.
- Where should be put x, so we can use it after the call to incr?

Callee-Saved Example #2

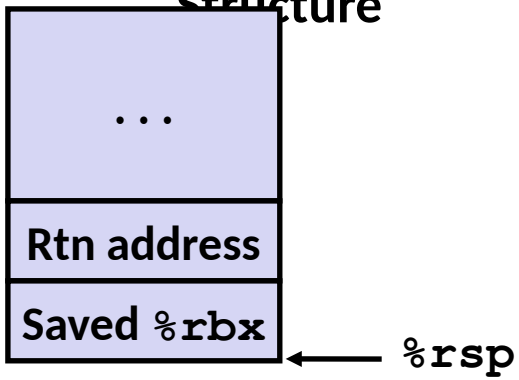
```
long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```

```
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
```

Initial Stack Structure



Resulting Stack Structure

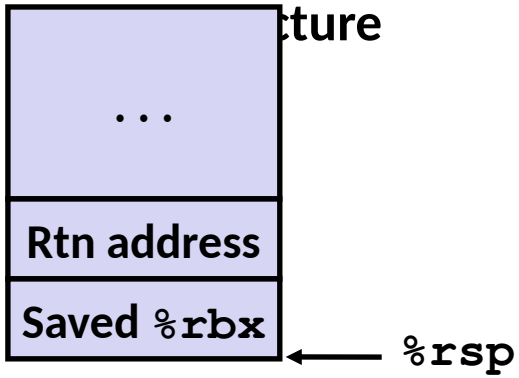


Callee-Saved Example #3

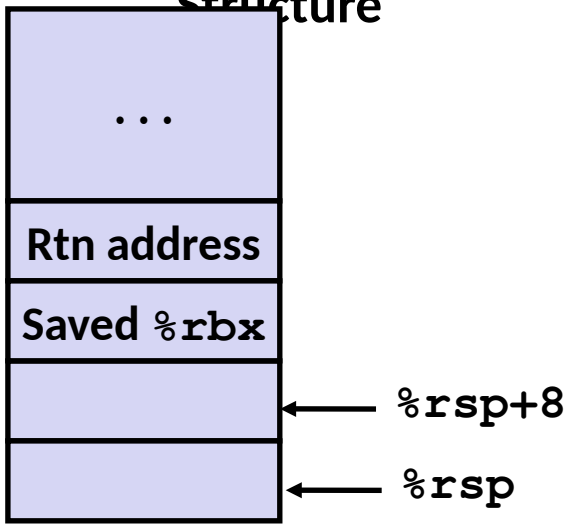
```
long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```

```
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

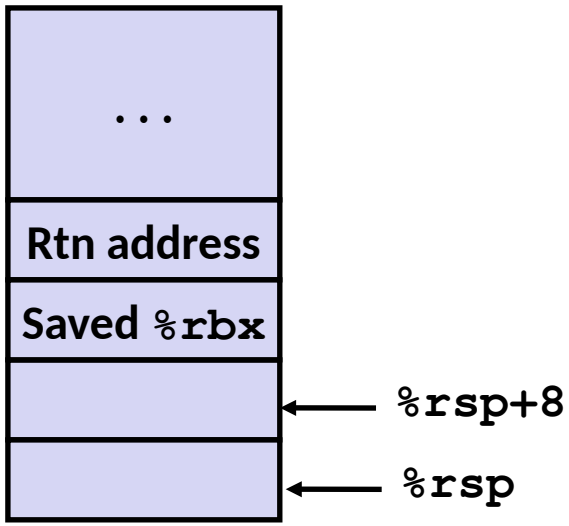


Callee-Saved Example #4

```
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
```

Stack Structure



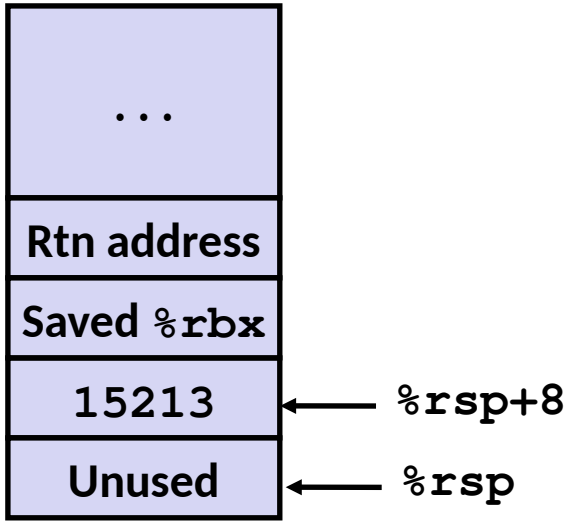
- X saved in **%rbx**.
- A callee saved register.

Callee-Saved Example #5

```
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
```

Stack Structure



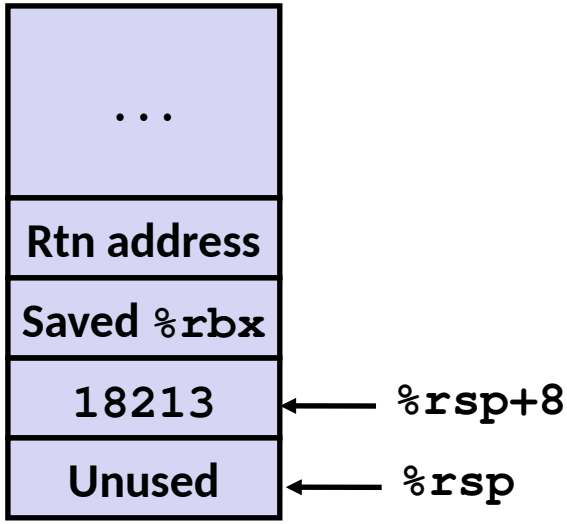
- X saved in **%rbx**.
- A callee saved register.

Callee-Saved Example #6

```
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
```

Stack Structure



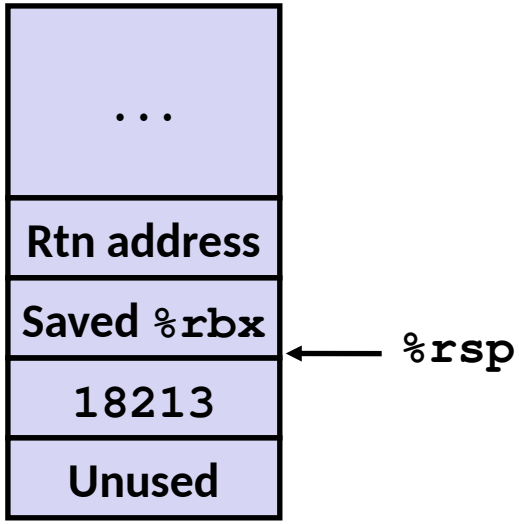
- X Is safe in **%rbx**
- Return result in **%rax**

Callee-Saved Example #7

```
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
```

Stack Structure



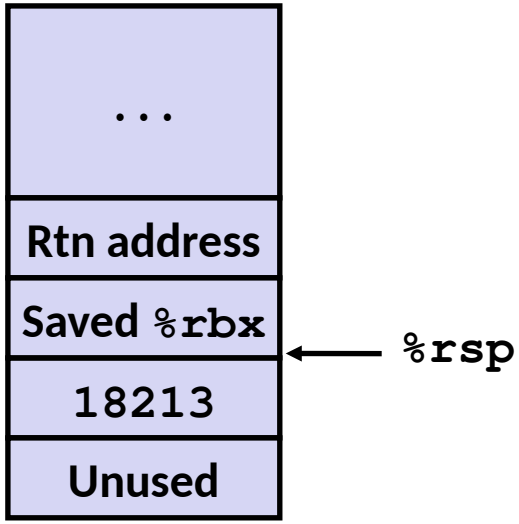
- Return result in **%rax**

Callee-Saved Example #8

```
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



final Stack Structure