CSE 2421

X86-64Assembly Language – Odds & Ends

Today

- **Procedures**

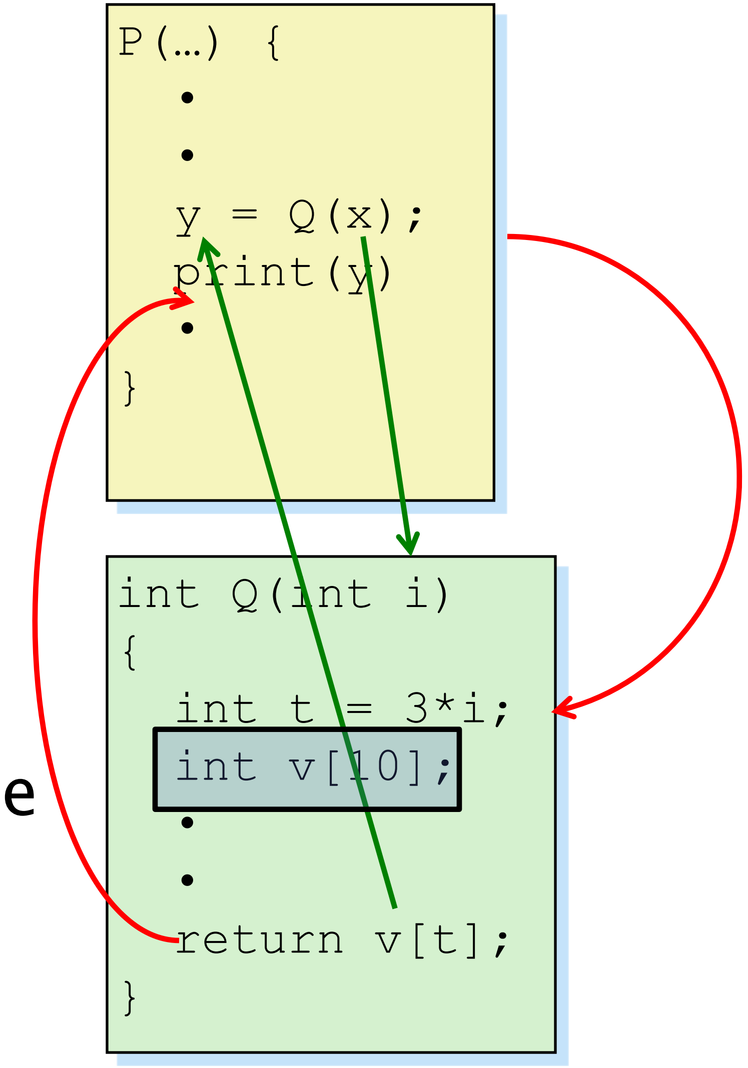
  Required Reading: *Computer Systems: A Programmer's Perspective, 3rd Edition*

  Chapter 3, Section 3.7 through 3.7.5 (inclusive)

  - Stack Structure
  - Calling Conventions
    - Passing control
    - Passing data
    - Managing local data
  - Procedure Summary

Mechanisms in Procedures

- Passing control
  - To beginning of procedure code
  - Back to return point
- Passing data
  - Procedure arguments
  - Return value
- Memory management
  - Allocate during procedure execution
  - Deallocate upon return
- Mechanisms all implemented with machine instructions
- x86-64 implementation of a procedure uses only those mechanisms required



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

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

Today

- **Procedures**

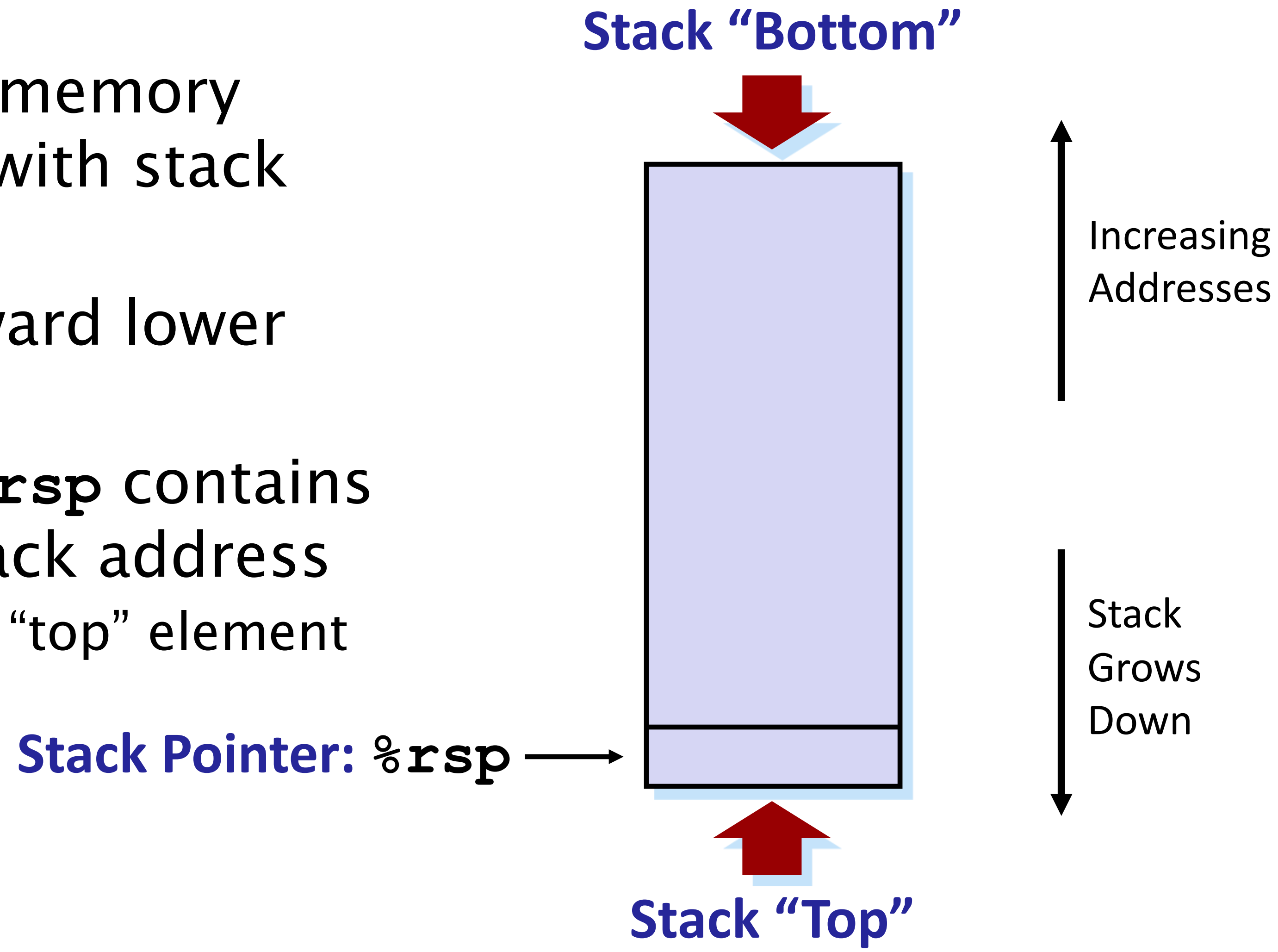
  Required Reading: *Computer Systems: A Programmer's Perspective, 3rd Edition*

  Chapter 3, Section 3.7 through 3.7.5 (inclusive)

  - Stack Structures
  - Calling Conventions
    - Passing control
    - Passing data
    - Managing local data
  - Procedure Summary

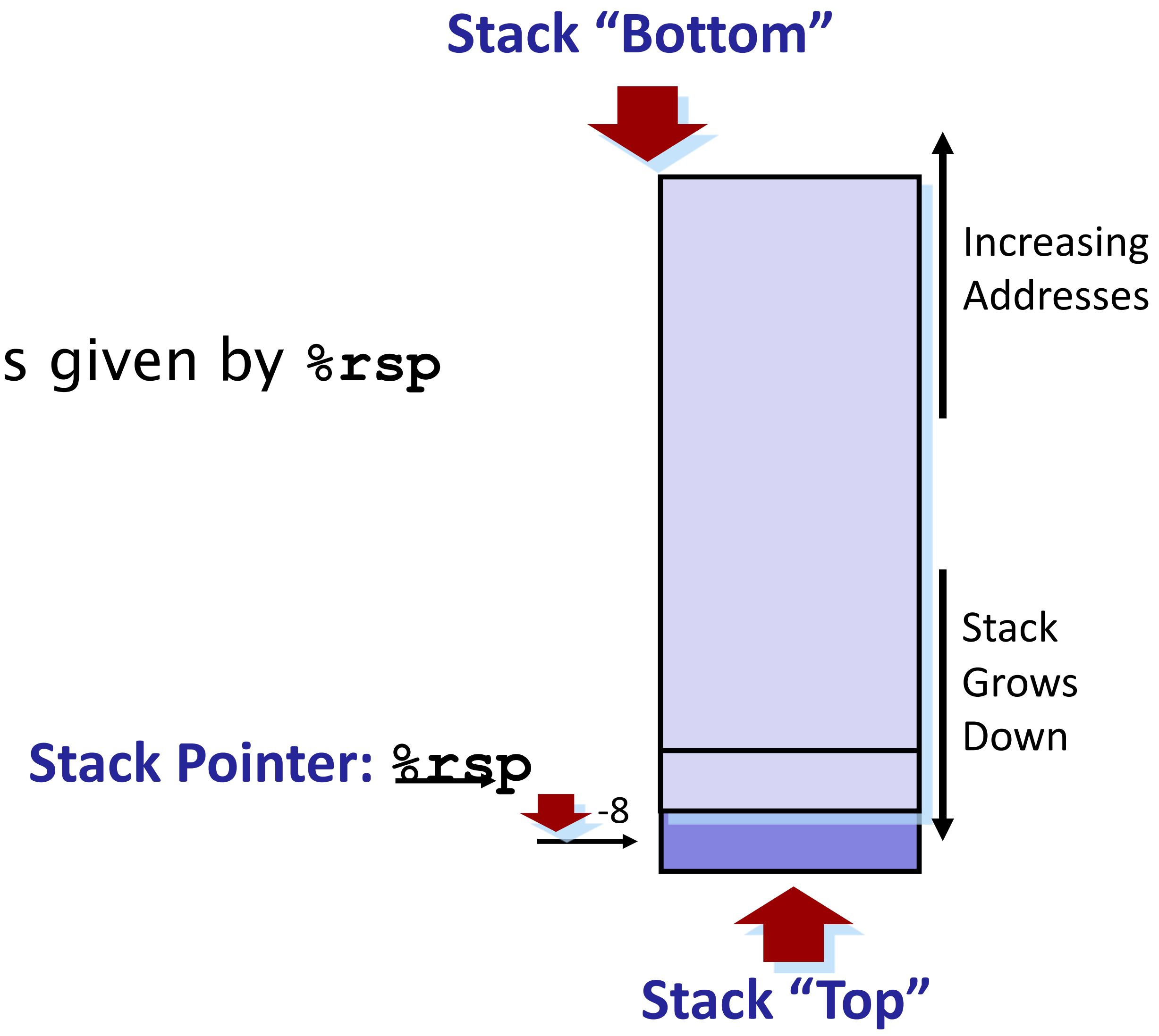
x86-64 Stack

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



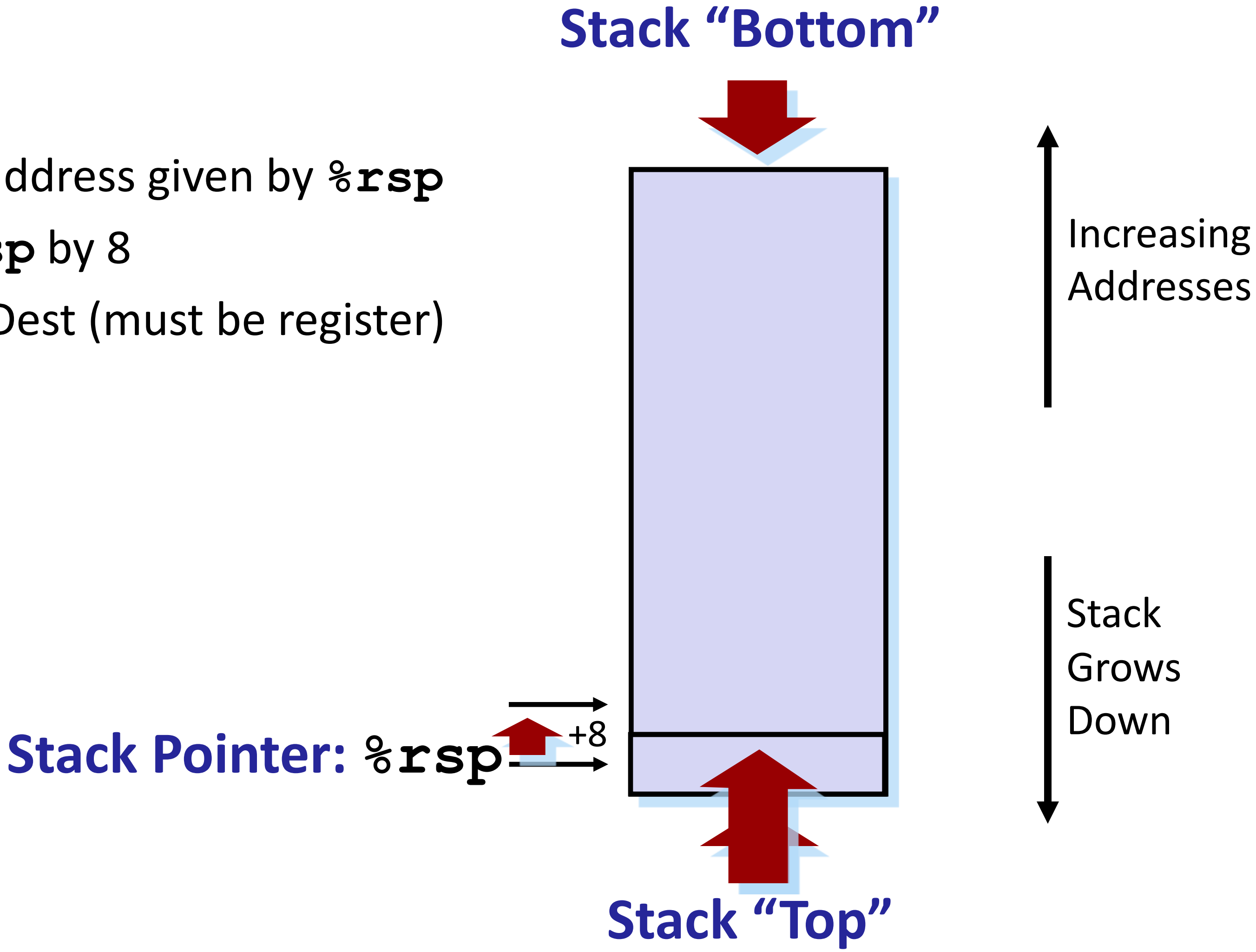
x86-64 Stack: Push

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



x86-64 Stack: Pop

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



Today

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

Code Examples

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

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

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

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

Procedure Control Flow

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

Stack Alignment requirement

- In X86-64 the ABI (Application Binary Interface) requires the stack address (the address in %rsp) to be 16-byte aligned *prior to any call*.
- This will always be true at the beginning of main in a program you write (the loader ensures it).
- For any function in your program which calls any other function, you must maintain 16-byte stack alignment.
- This means that you must always, when you do pushes or pops to or from the stack, adjust %rsp if the total number of bytes pushed/popped is not a multiple of 16.
- Technically, this is only necessary for programs which use SSE instructions (which do operations on 16 byte float registers), and we will not use these, so we can remove this requirement.
- Example on next slide

Stack alignment example

- Function code

```
push %rbp          #8 bytes pushed (stack is now 16B aligned)
movq %rsp, %rbp        #set function’s frame pointer
```

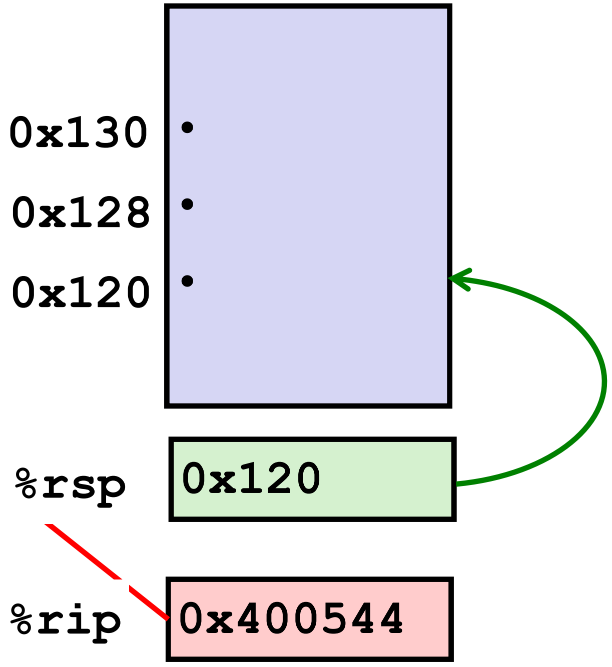
- If you want to put some parameters in registers, and call another function without pushing anything else onto the stack, you need to subtract 8 bytes from %rsp, to keep it 16-byte aligned:

```
sub $8, %rsp           #subtract 8 bytes from rsp to keep
                       #address 16-byte aligned
```

Control Flow Example #1

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

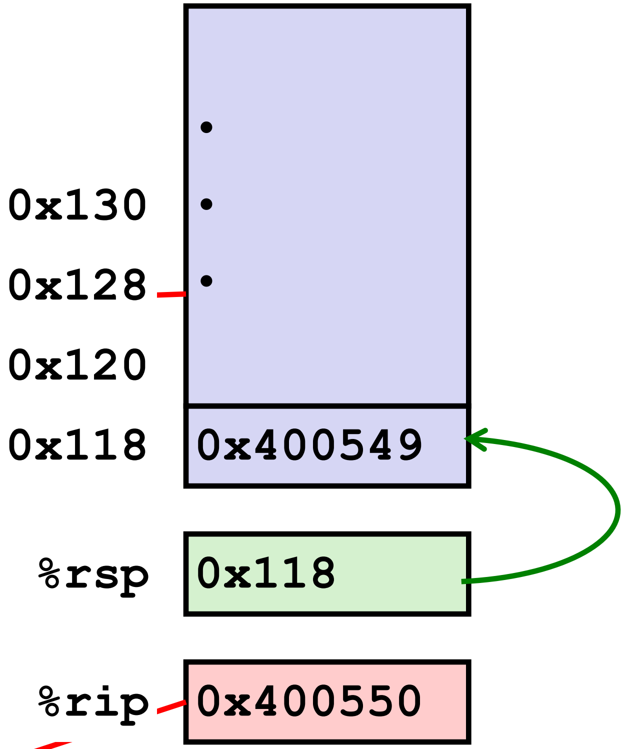
```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  retq
```



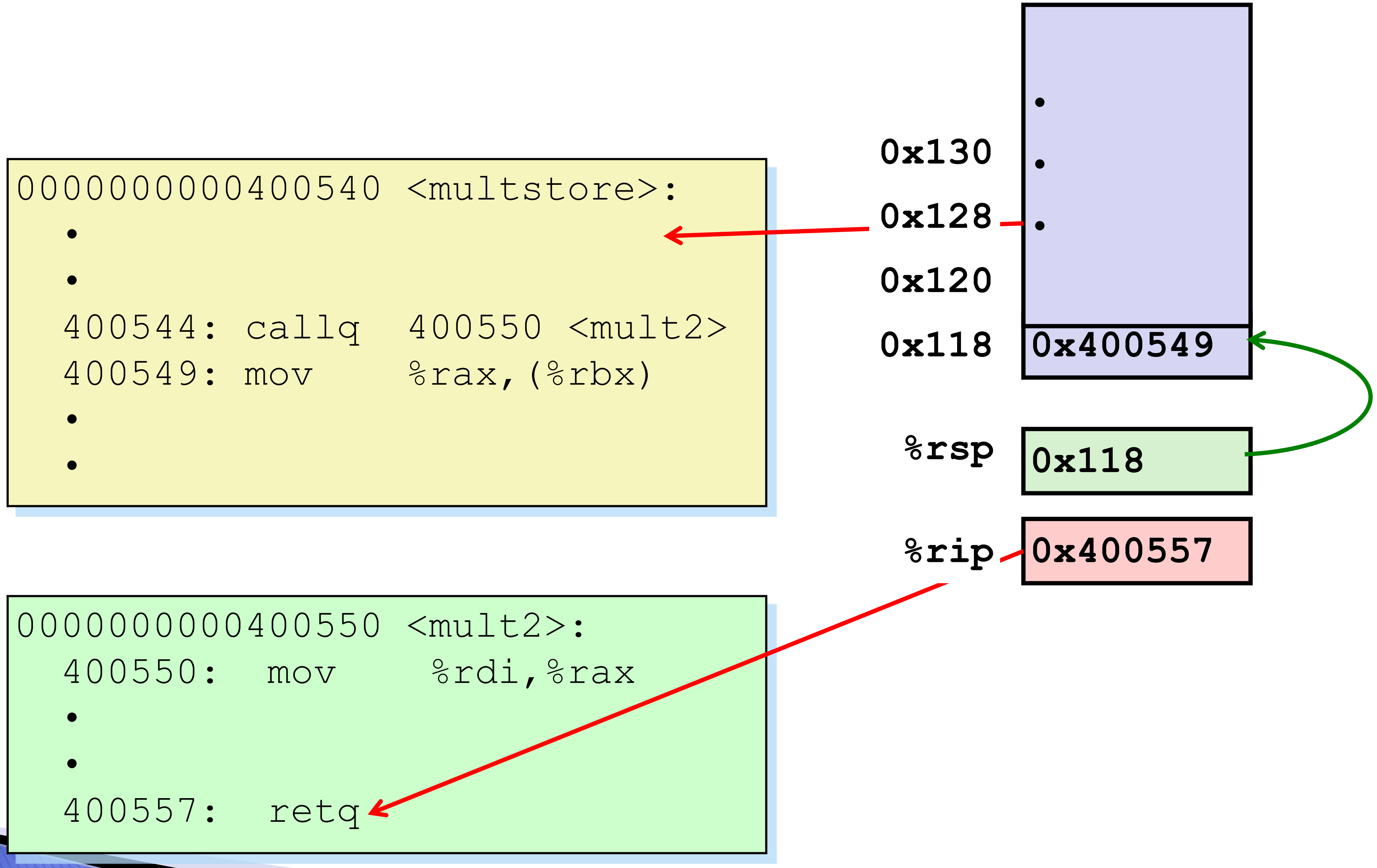
Control Flow Example #2

```
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
```

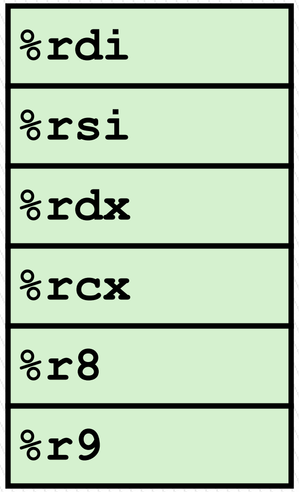
Today

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

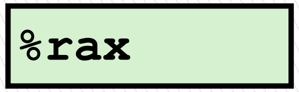
Procedure Data Flow

Registers

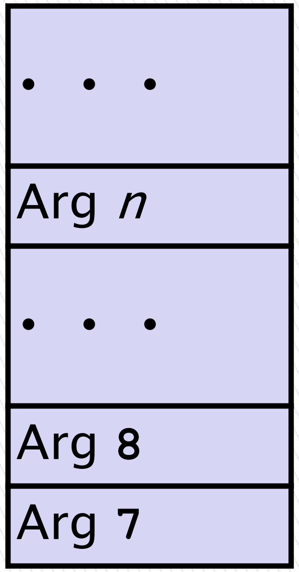
- First 6 arguments



- Return value



Stack



- Only allocate stack space when needed
- Push arg 7 last

Today

- Procedures
  - Stack Structure
  - Calling Conventions
    - Passing control
    - Passing data
    - Managing local data
  - Procedure Summary

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 a given procedure is needed for a limited time
    - From when it's called to when it returns
  - Callee returns before caller does
- Stack allocated in ***Frames***
  - state for single procedure instantiation

Call Chain Example

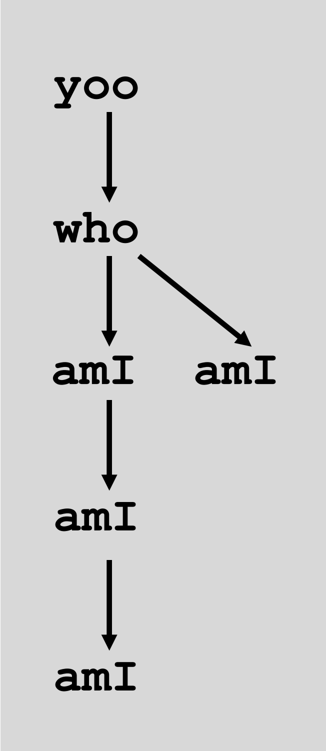
```
yoo(…)
{
  •
  •
  who();
  •
  •
}
```

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

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

Procedure **amI()** is recursive

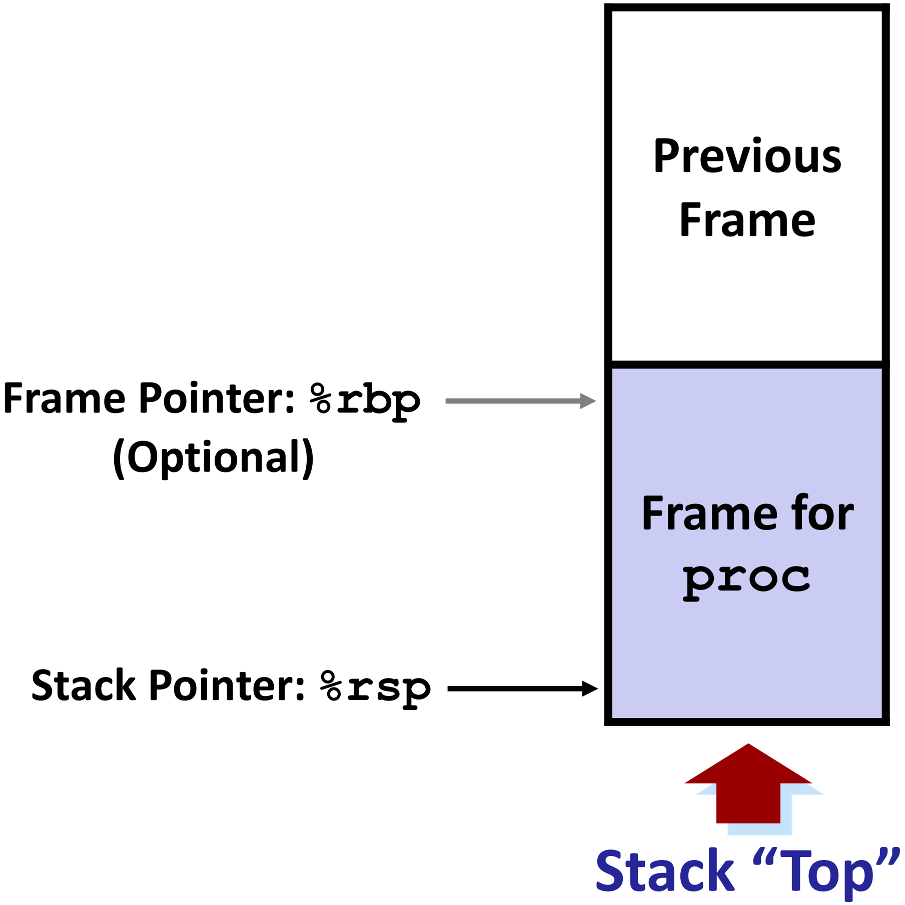
Example Call Chain



Stack Frames

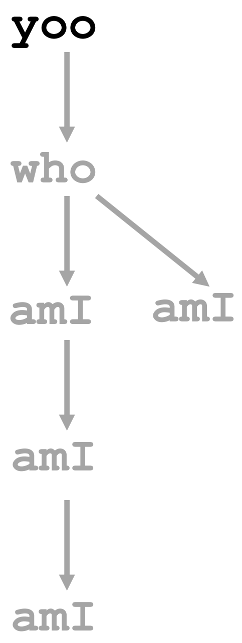
- Contents
  - Return information
  - Local storage (if needed)
  - Temporary space (if needed)

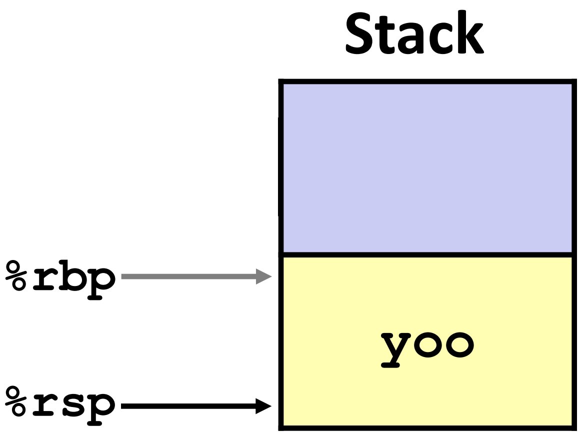
- Management
  - Space allocated when enter procedure
    - “Set-up” code
    - Includes push by `call` instruction
  - Deallocated when return
    - “Finish” code
    - Includes pop by `ret` instruction



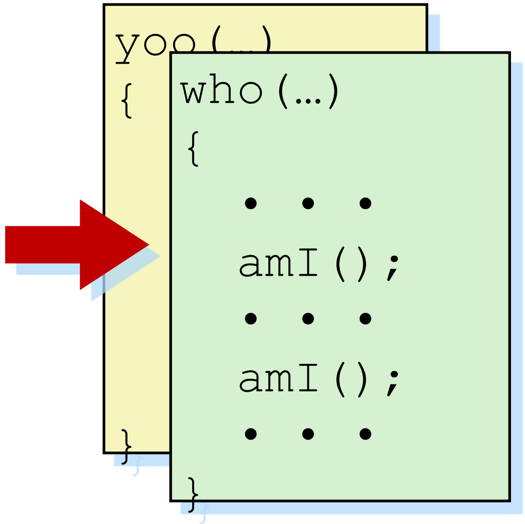
Example

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



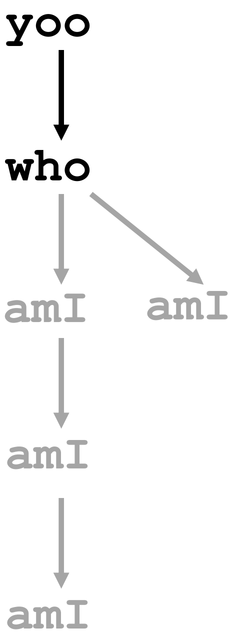


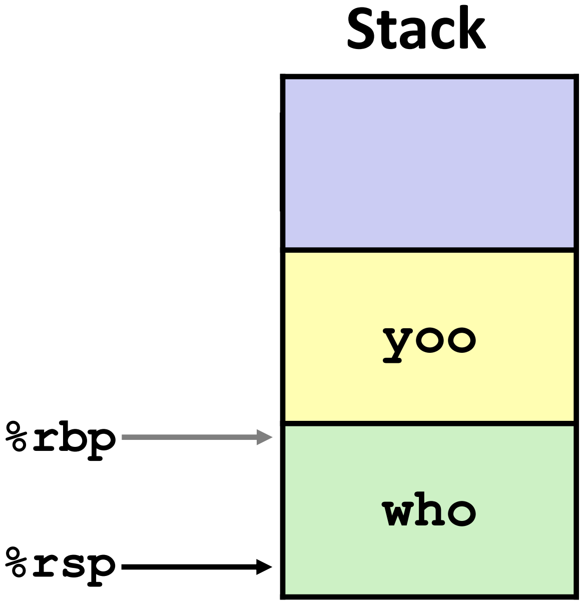
Example



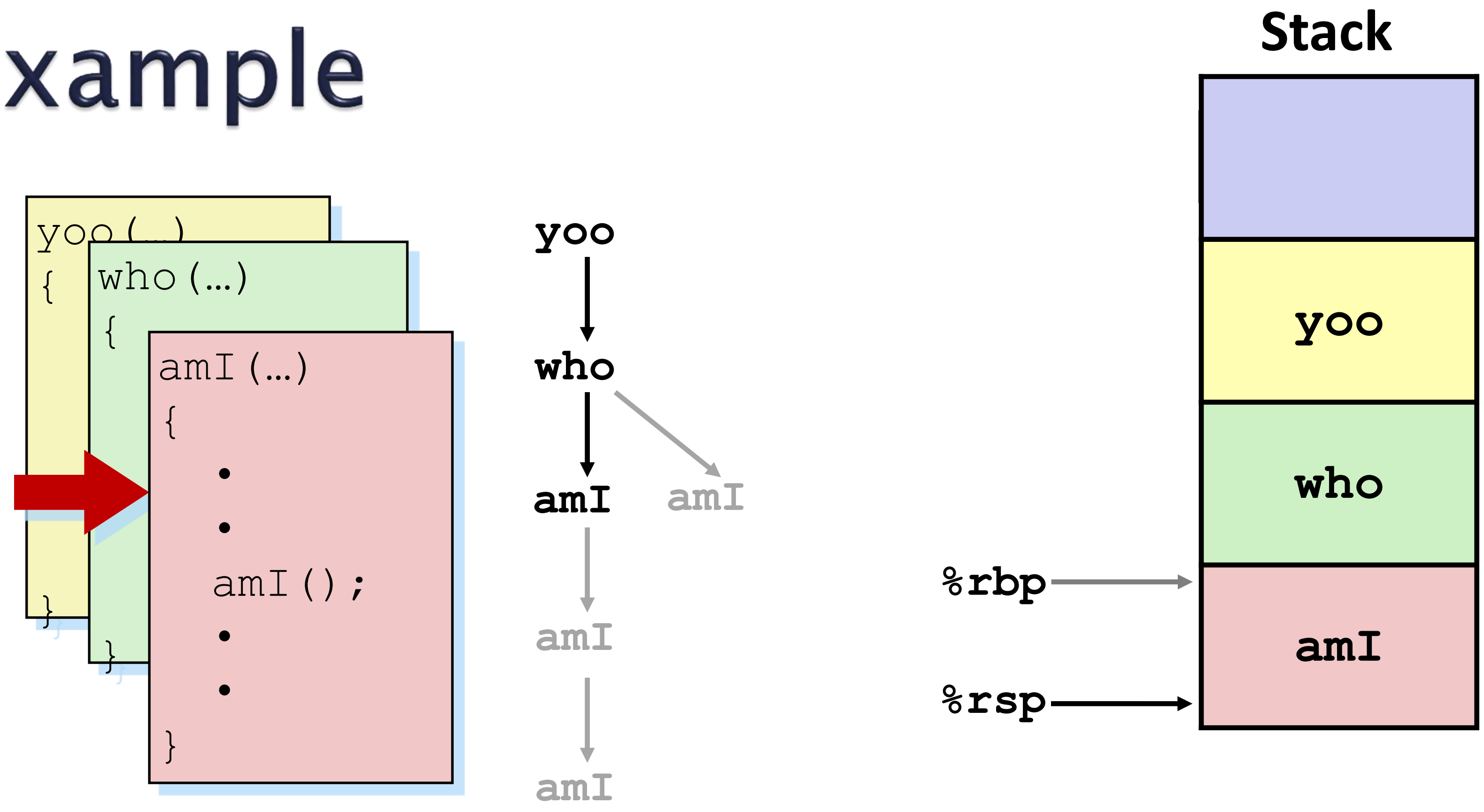
```
yoo(…)
{
}
```

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





Example

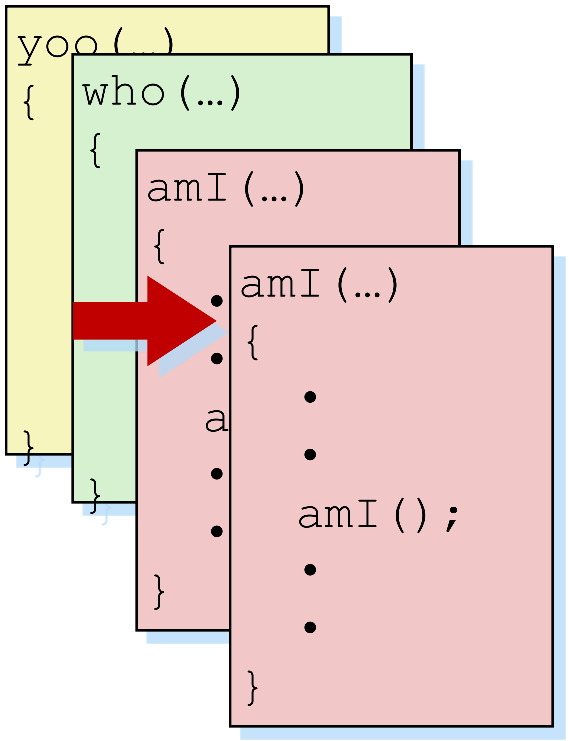


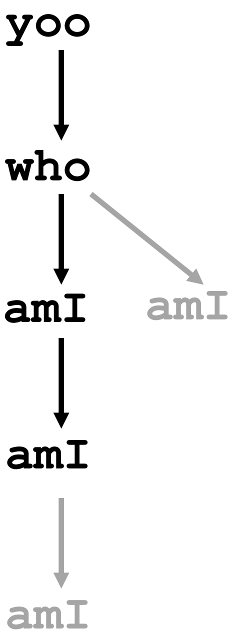
```
yoo(…)
{



}
```

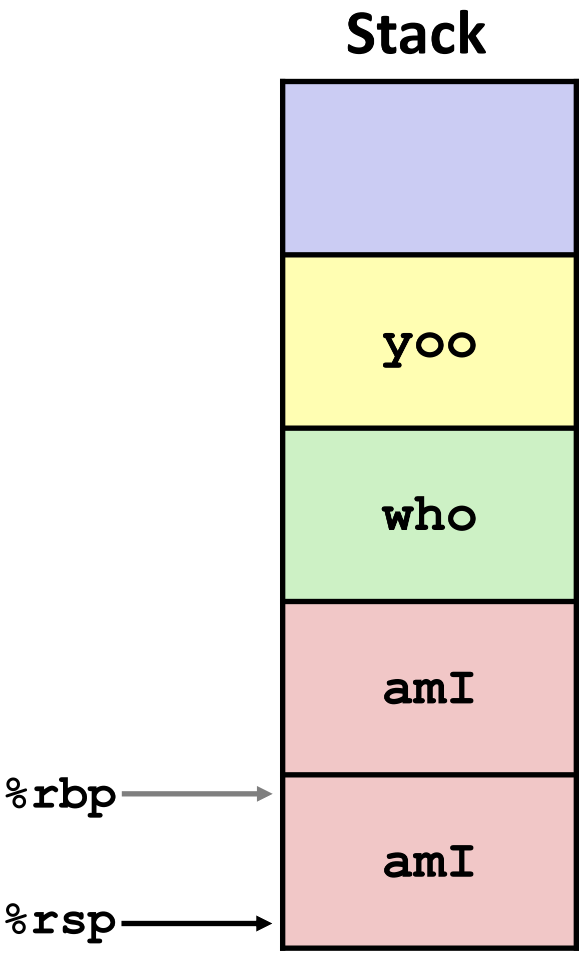
```
who(…)
{



}
```

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

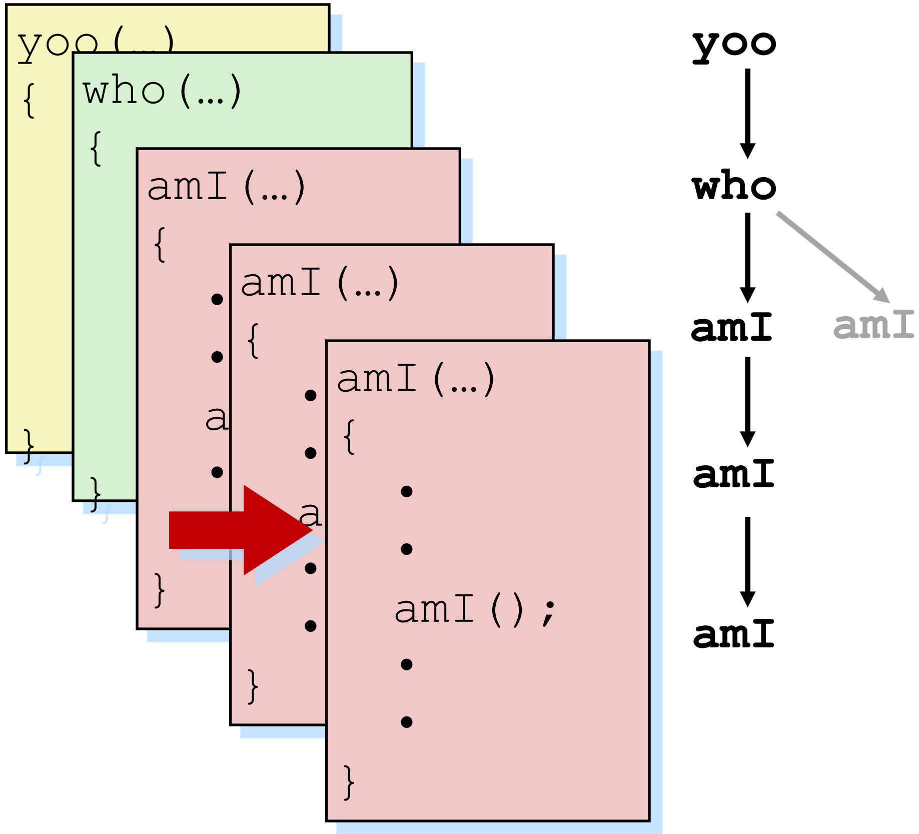
Example


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

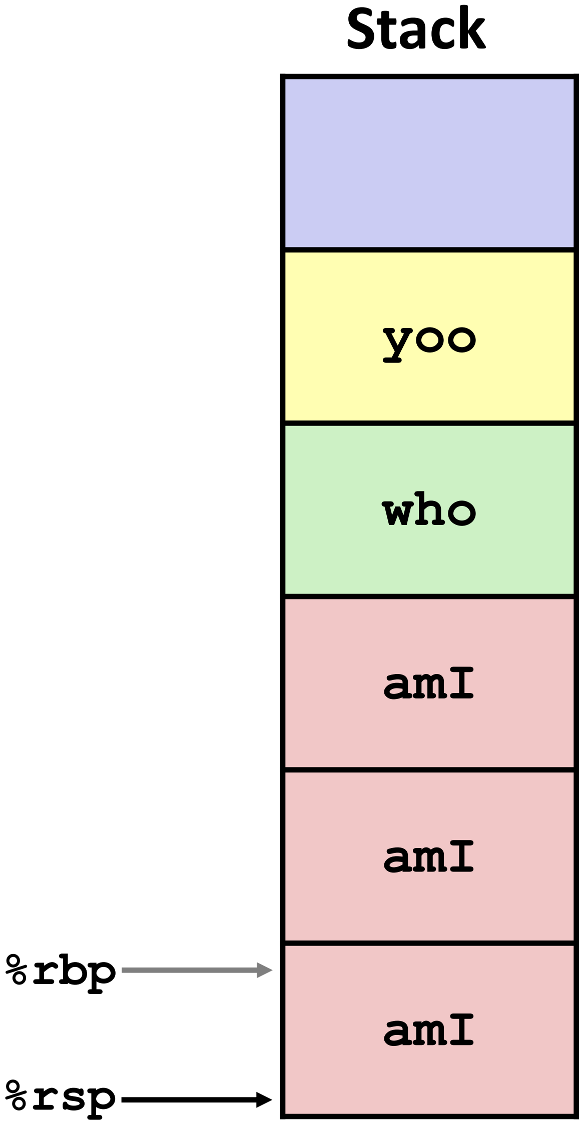

yoo
who
amI
amI
amI
amI


Stack
yoo
who
amI
amI
%rbp
%rsp

Example

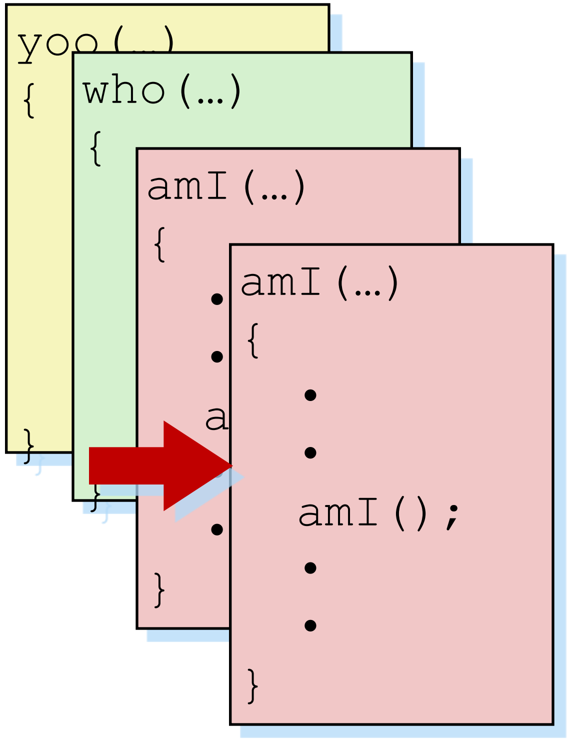


yoo (…)
{
}
who (…)
{
}
amI (…)
{
•
•
a
•
}
amI (…)
{
•
•
a
•
•
}
amI (…)
{
•
•
amI ();
•
•
}
yoo
who
amI
amI
amI
amI

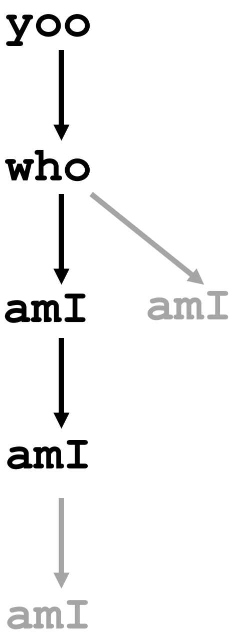


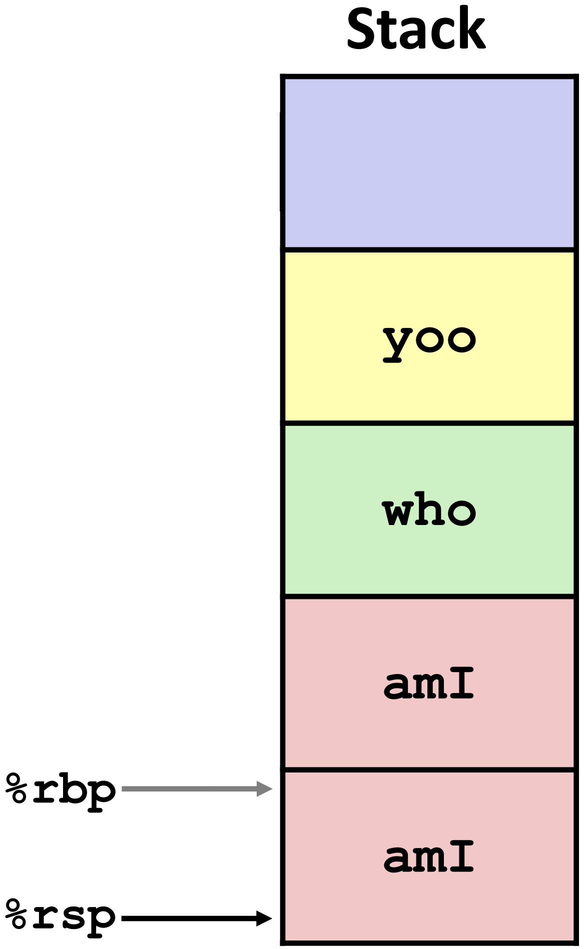
Stack
yoo
who
amI
amI
amI
%rbp
%rsp

Example

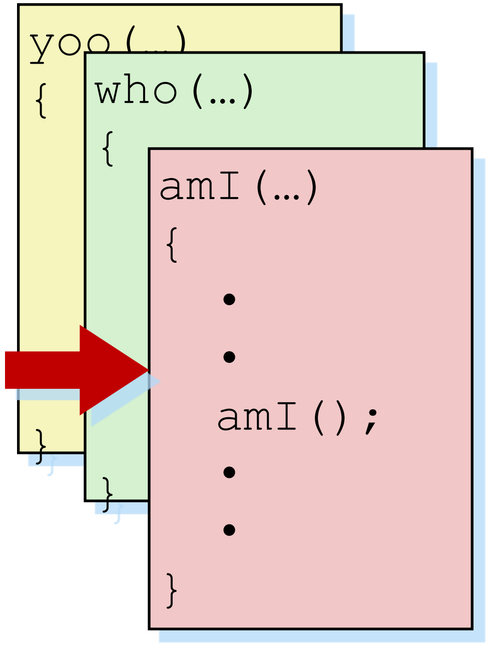


```
amI(…)
{
    •
    •
    amI();
    •
    •
}
```





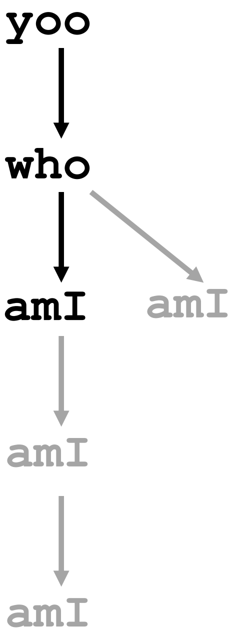
Example

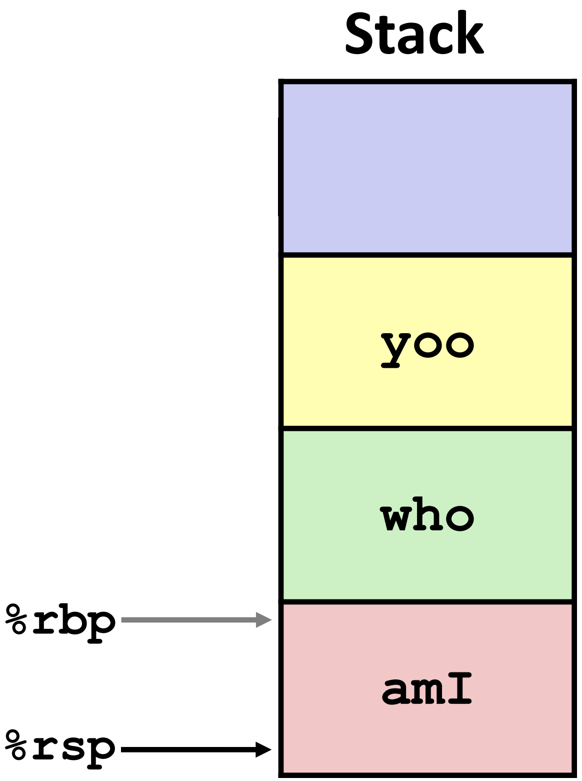


```
yoo(…)
{

}
```

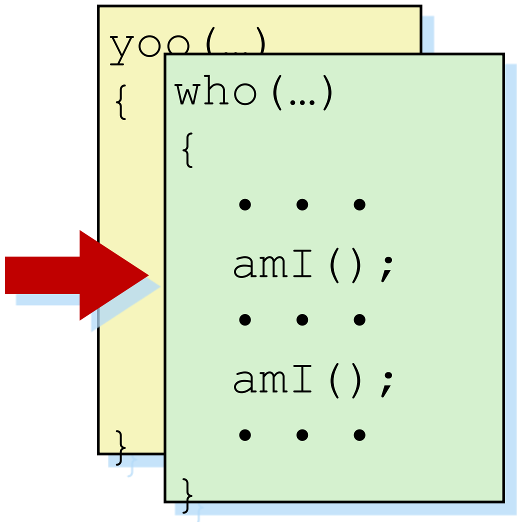
```
who(…)
{

}
```

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



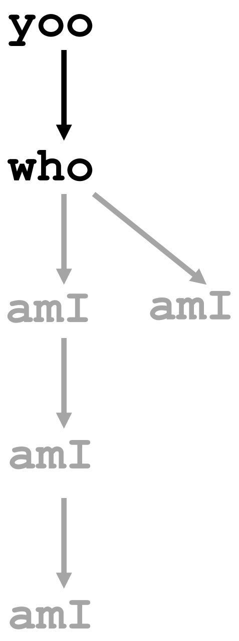


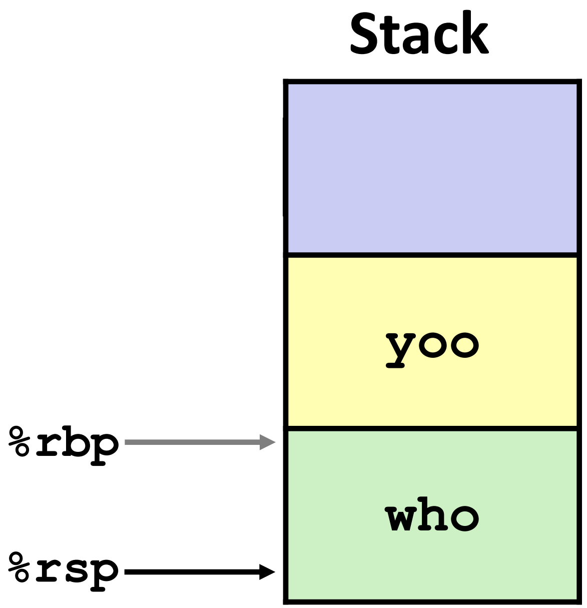
Example



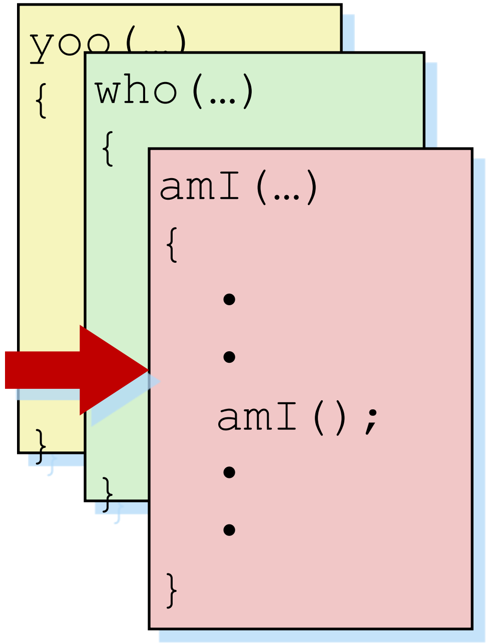
```
yoo(…)
{



}
```

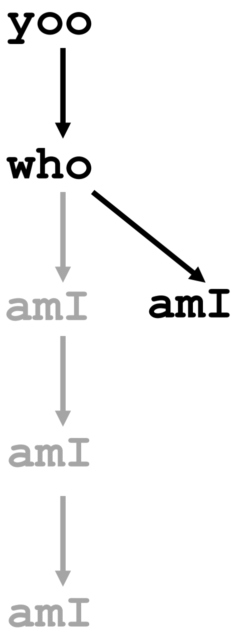
```
who(…)
{
  • • •
  amI();
  • • •
  amI();
  • • •
}
```

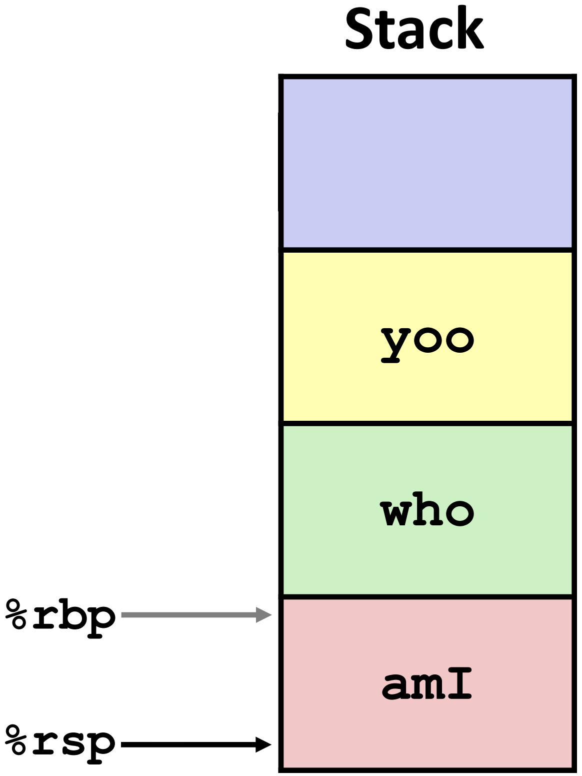




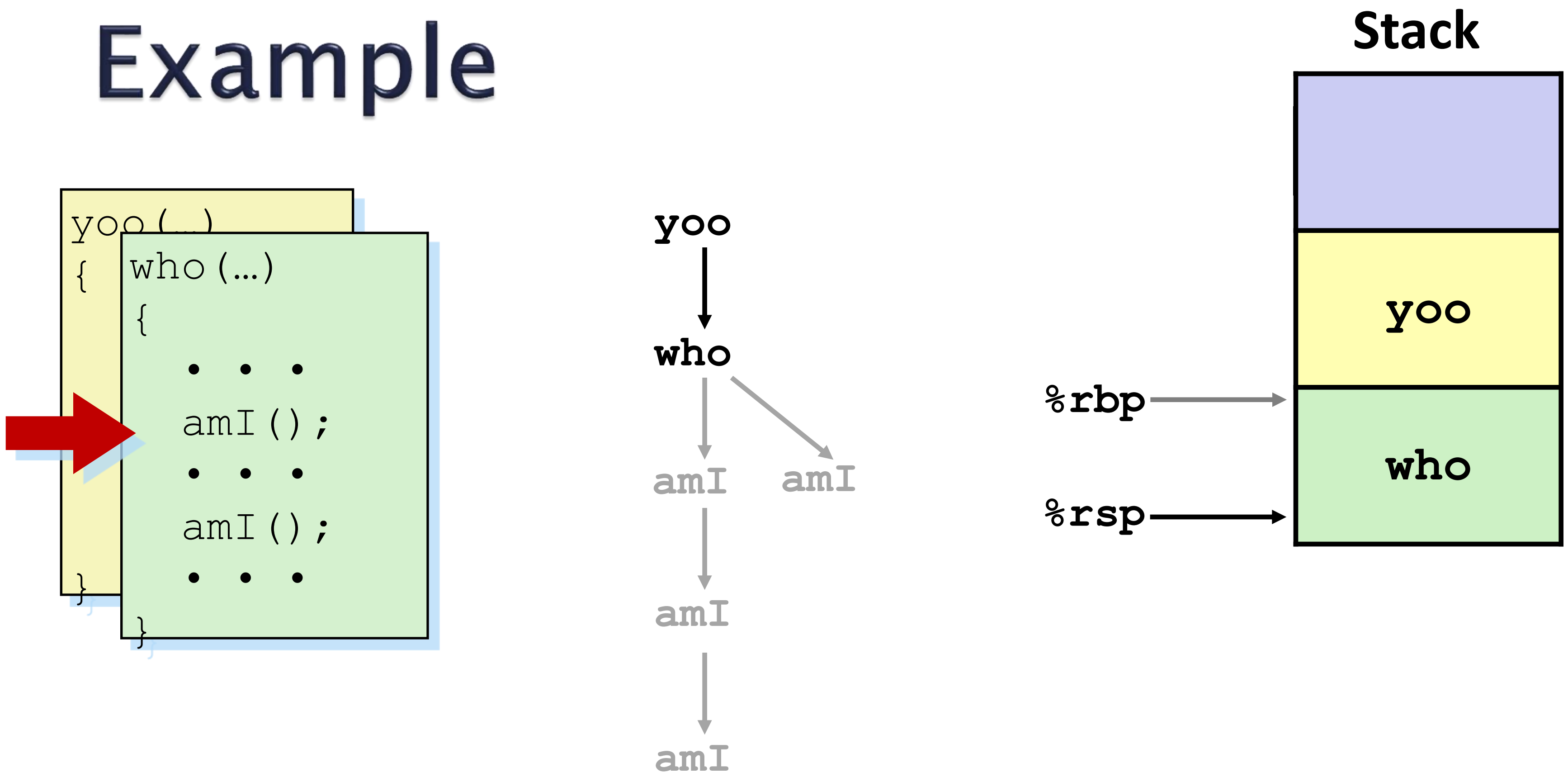
Example


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


yoo
who
amI
amI
amI
amI


Stack
yoo
who
amI
%rbp
%rsp

Example

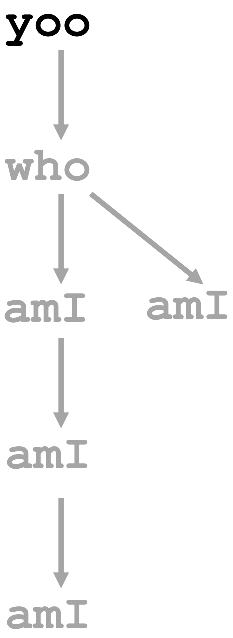


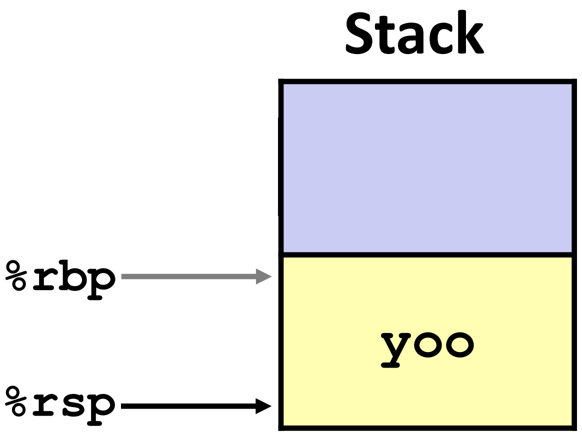
```
yoo(…)
{

}
```

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

Example

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

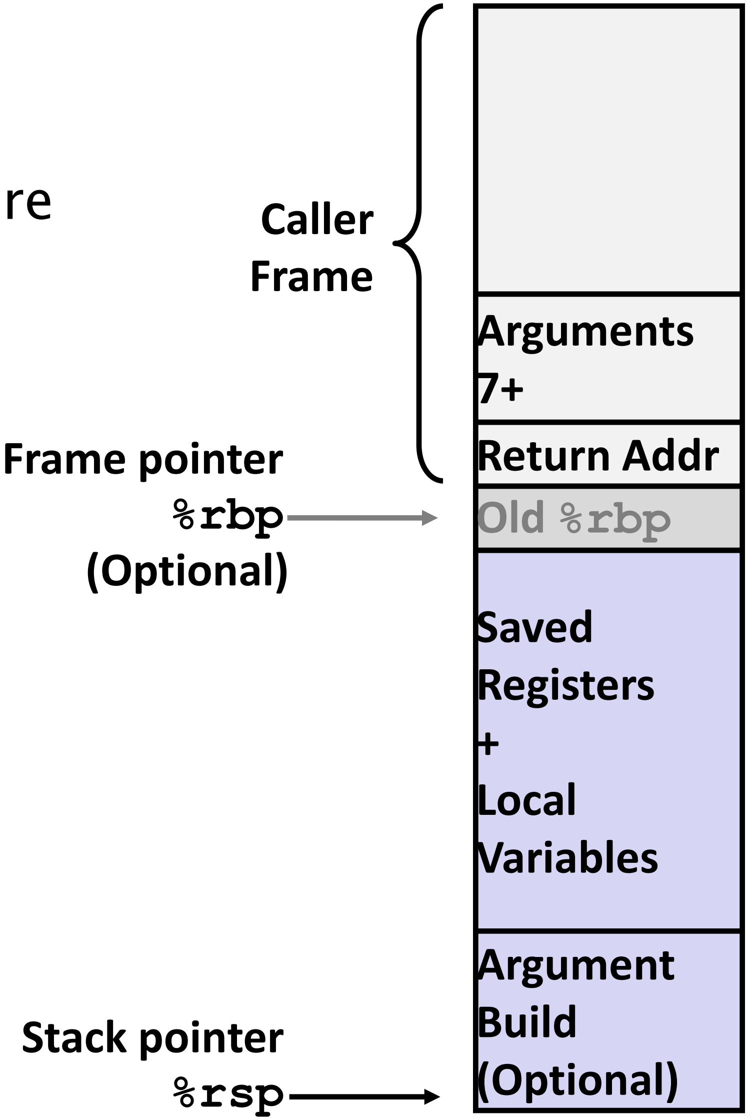




x86-64/Linux Stack Frame

- Current Stack Frame (“Top” to Bottom)
  - “Argument build:”
    Parameters for the function about to call, if more than 6
  - Local variables
    If so many, can’t keep them in registers
  - Saved register context
  - Old frame pointer

- Caller Stack Frame
  - Return address
    - Pushed by call instruction
  - Arguments for this call (more than 6)

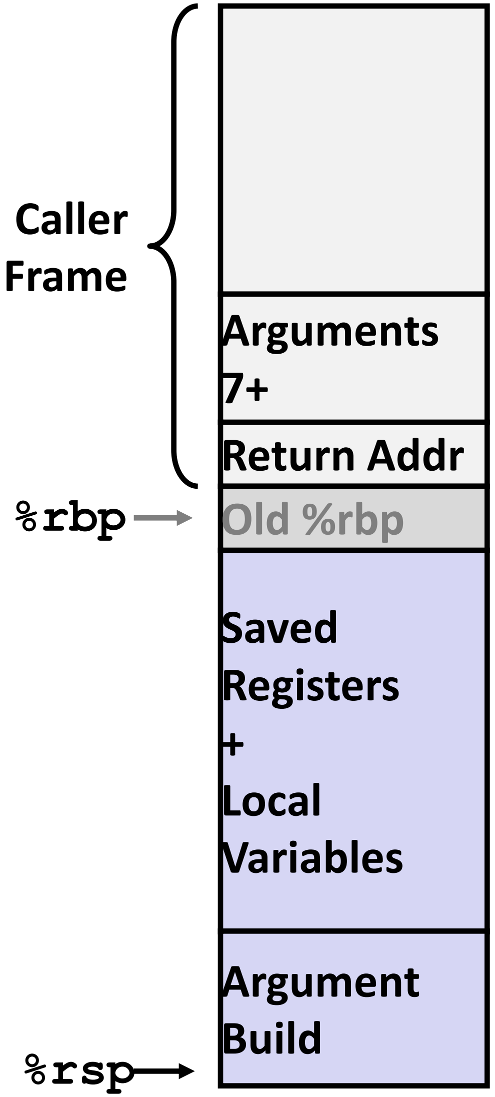

Caller Frame
Arguments 7+
Return Addr
Frame pointer %rbp (Optional)
Old %rbp
Saved Registers + Local Variables
Argument Build (Optional)
Stack pointer %rsp

Today

- Procedures
    - Stack Structure
    - Calling Conventions
        - Passing control
        - Passing data
        - Managing local data
    - Procedure Summary

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



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