Machine Level Programming: Procedures

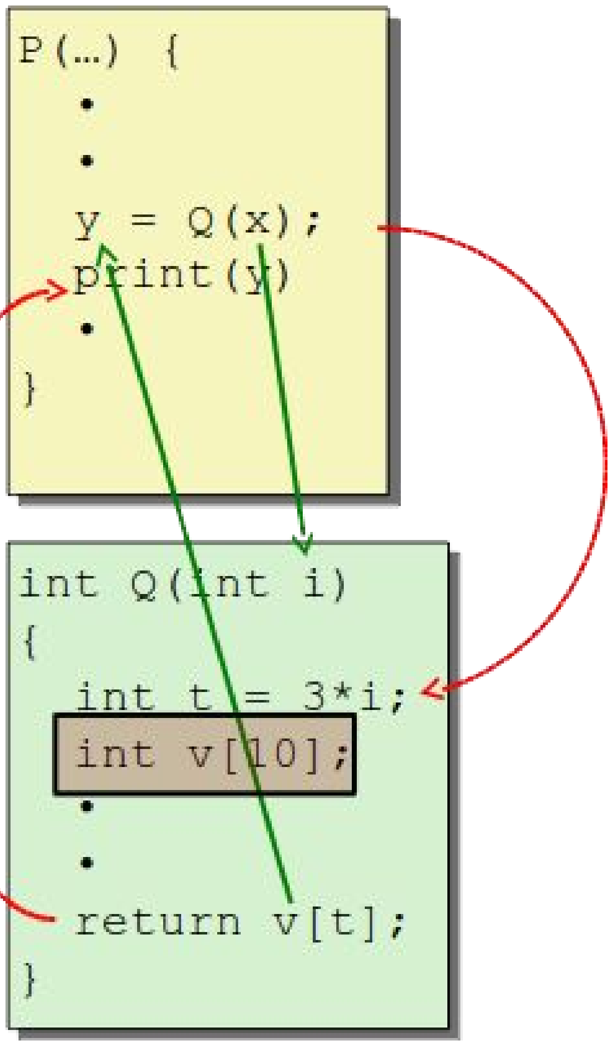
Computer Systems Organization (Spring 2017)
CSCI-UA 201, Section 3

Instructor: Joanna Klukowska

Slides adapted from
Randal E. Bryant and David R. O'Hallaron (CMU)
Mohamed Zahran (NYU)

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



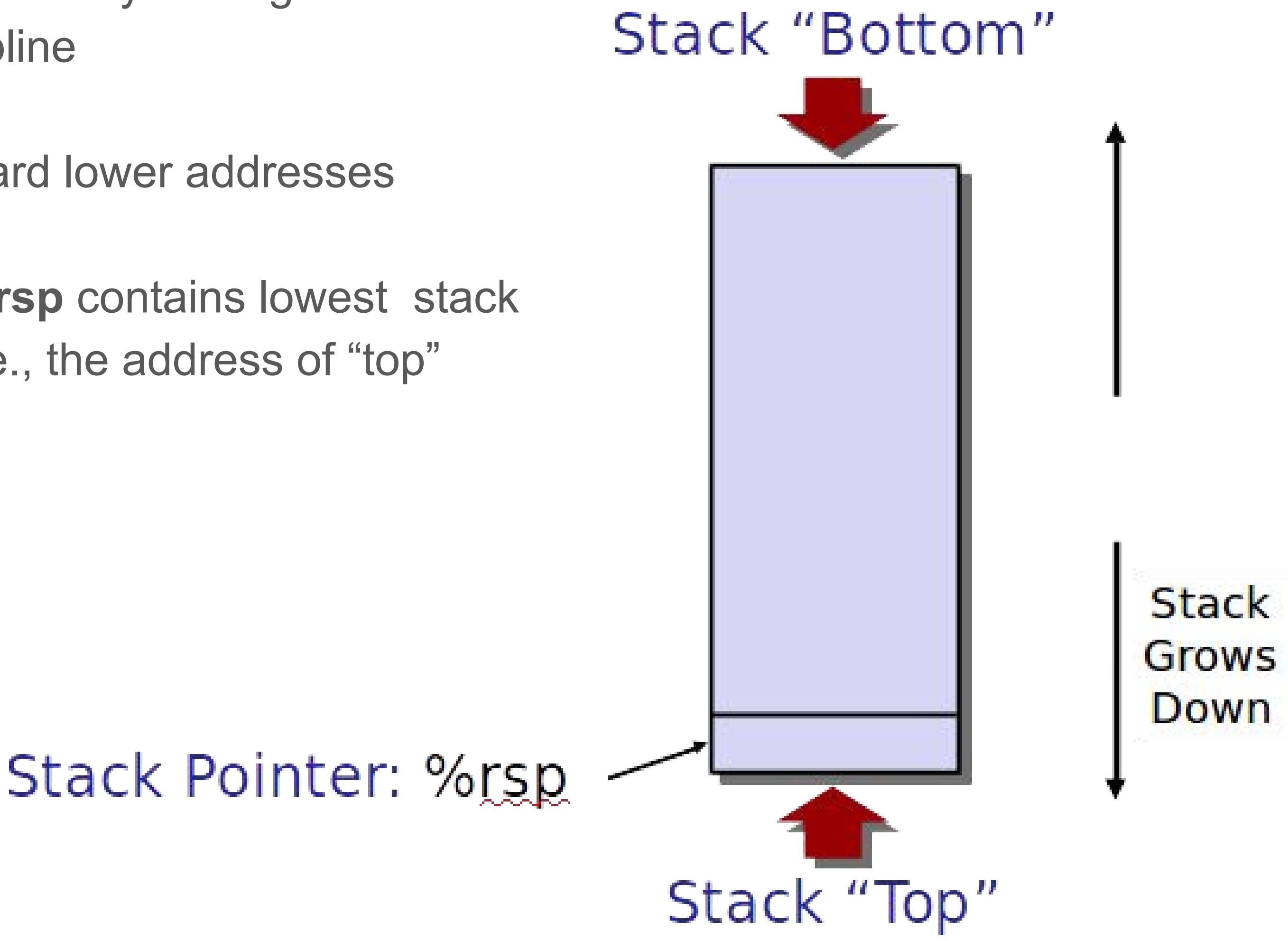
```
P(…) {
  •
  •
  y = Q(x);
  print(y)
  •
}
```

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

Stack Structure

x86-64 Stack

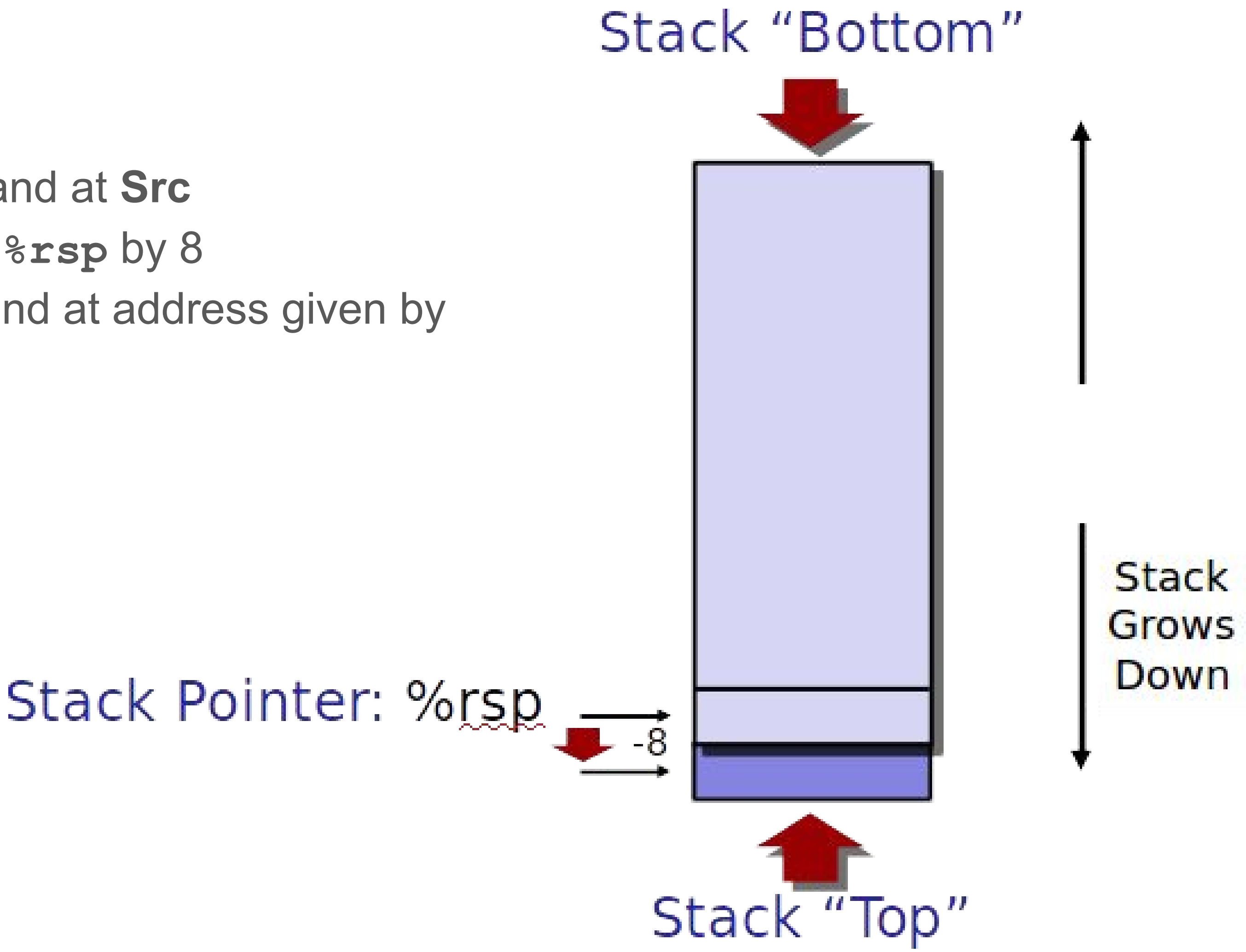
- Region of memory managed with stack discipline
- Grows toward lower addresses
- Register **%rsp** contains lowest stack address (i.e., the address of "top" element)



x86-64: **push**

`pushq Src`

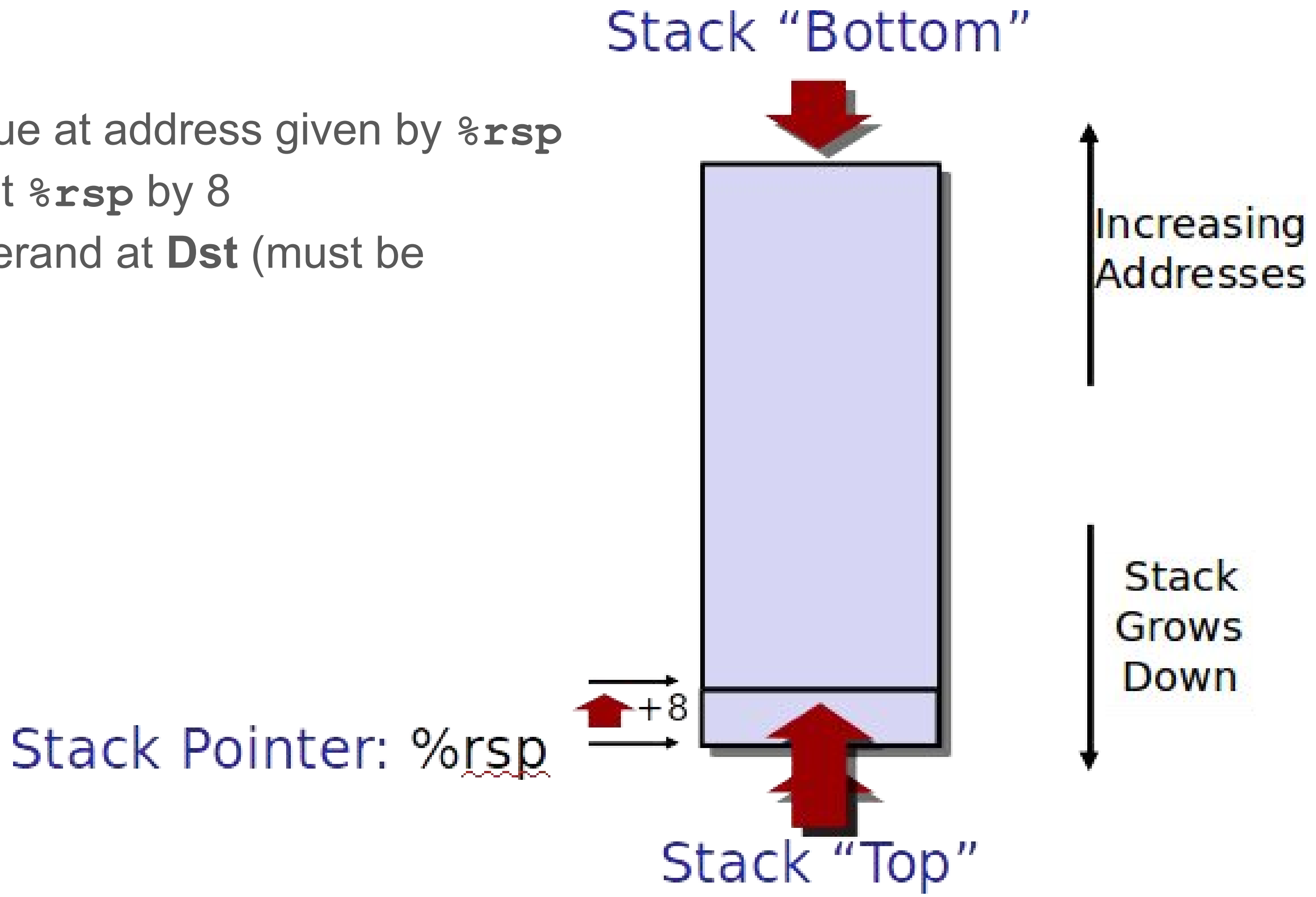
- Fetch operand at **Src**
- Decrement **`%rsp`** by 8
- Write operand at address given by **`%rsp`**



x86-64: **`pop`**

`popq Dst`

- Read value at address given by **`%rsp`**
- Increment **`%rsp`** by 8
- Fetch operand at **Dst** (must be register)



Passing Control

Procedure Control Flow - Code example

```
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
```

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

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

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

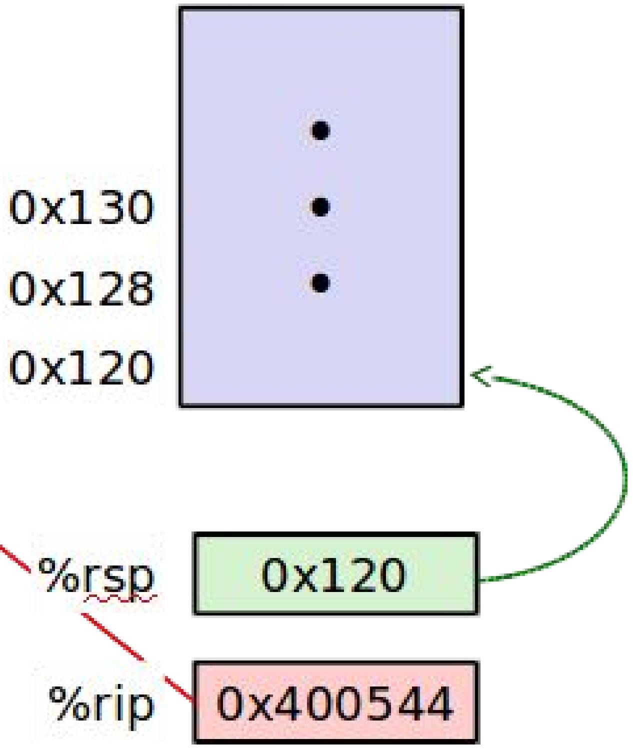
Procedure Control Flow

- 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

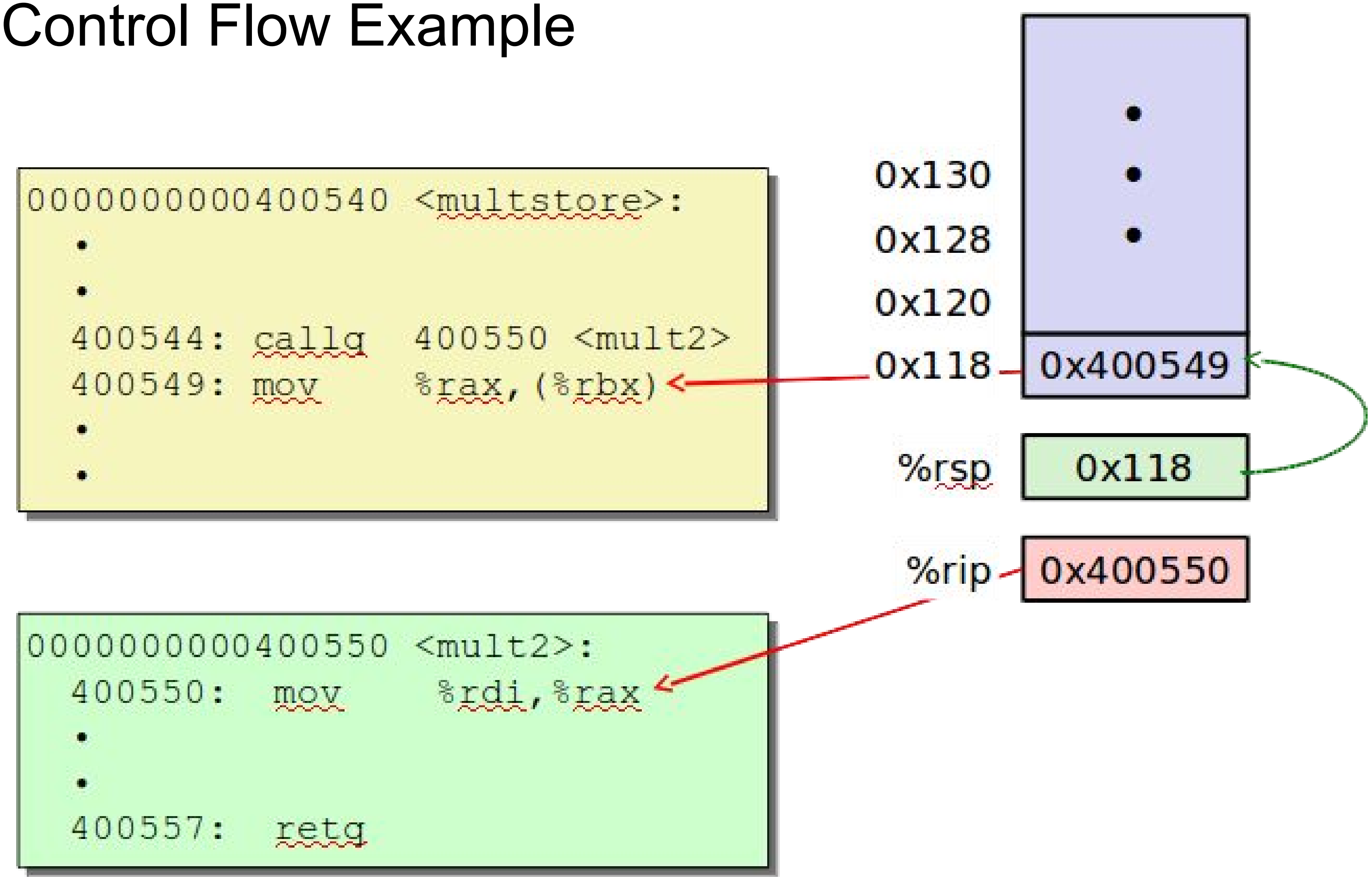
Control Flow Example

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

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



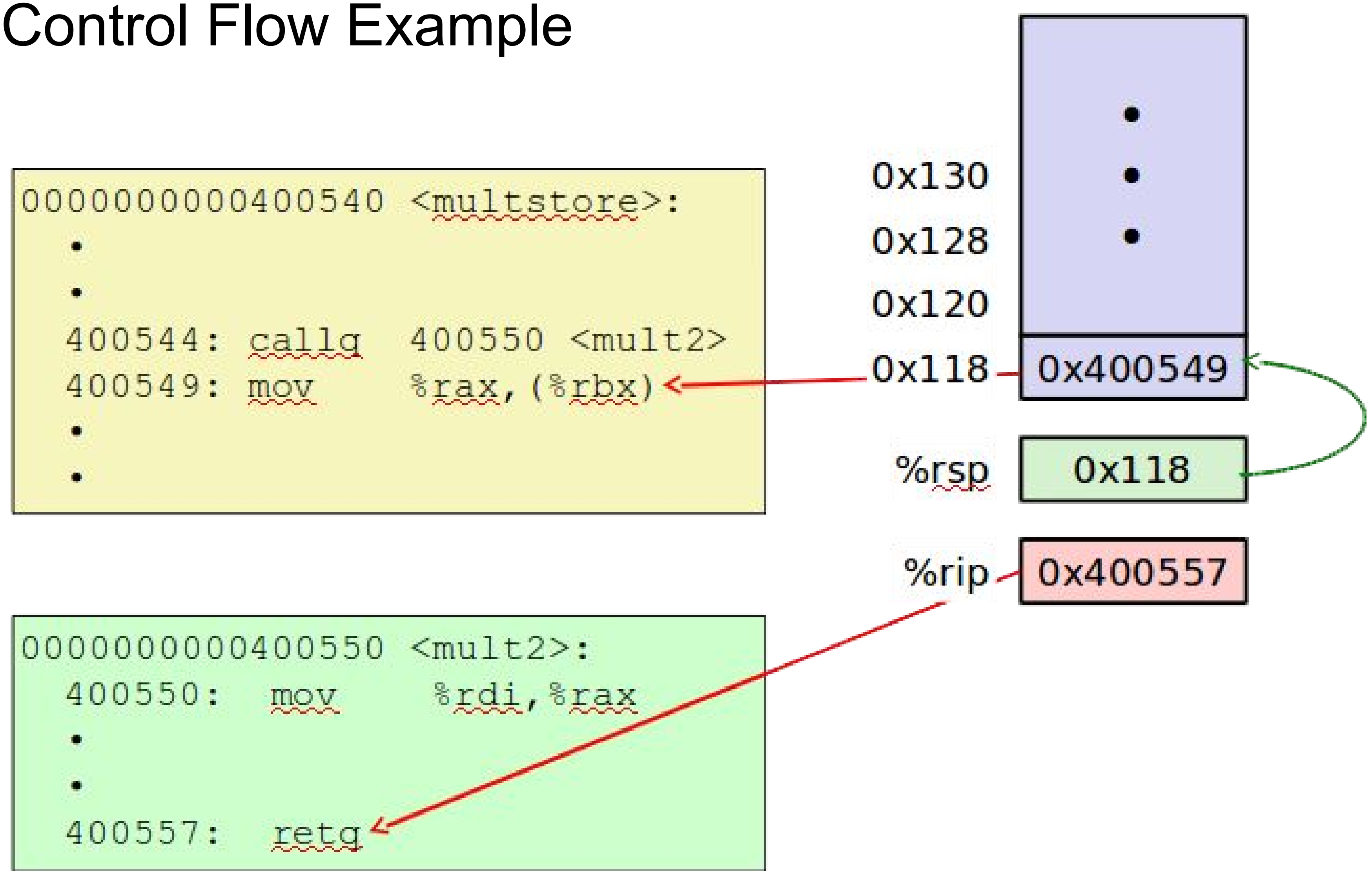
Control Flow Example



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

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

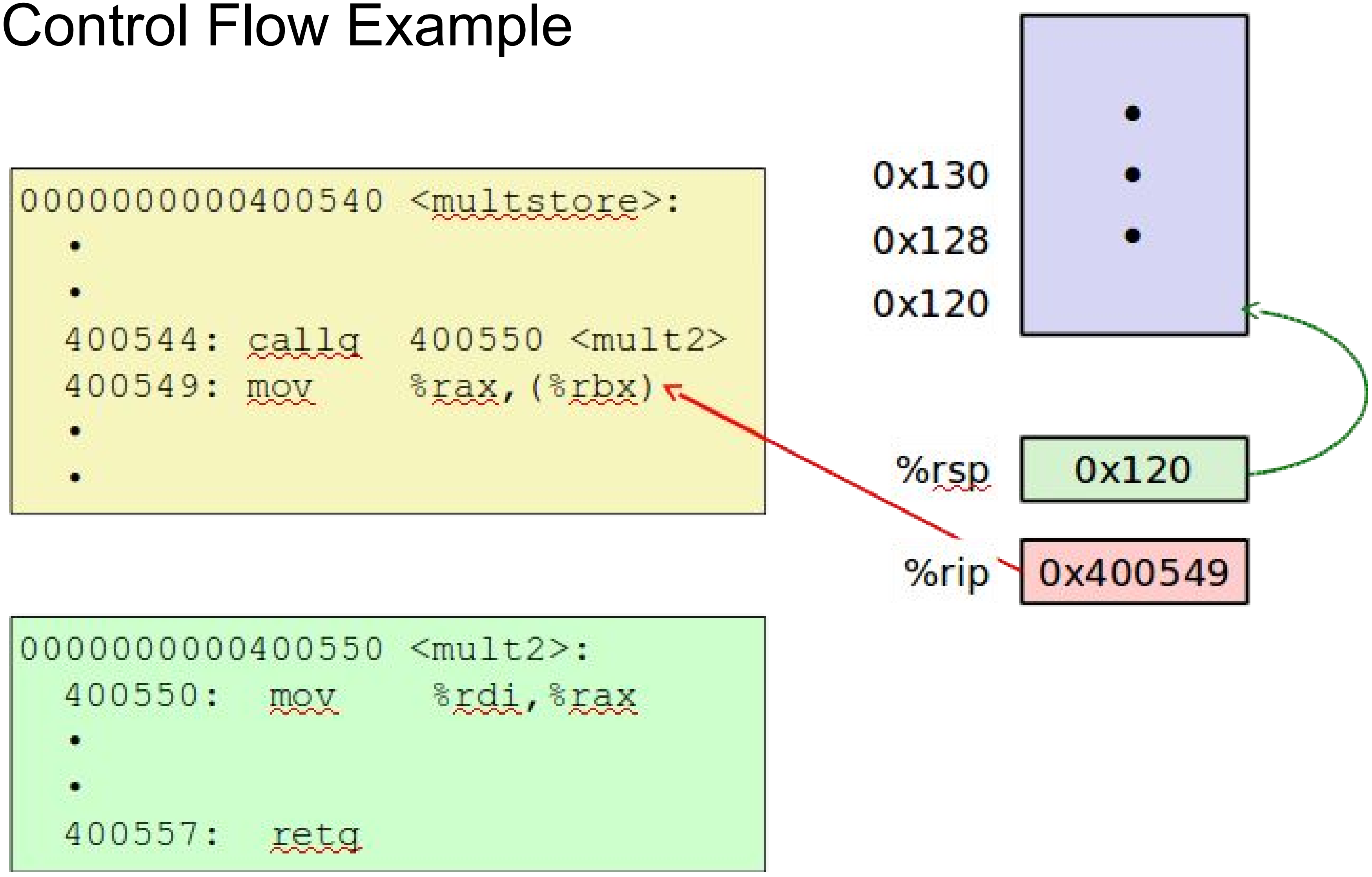
Control Flow Example



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

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

Control Flow Example



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

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

Passing Data

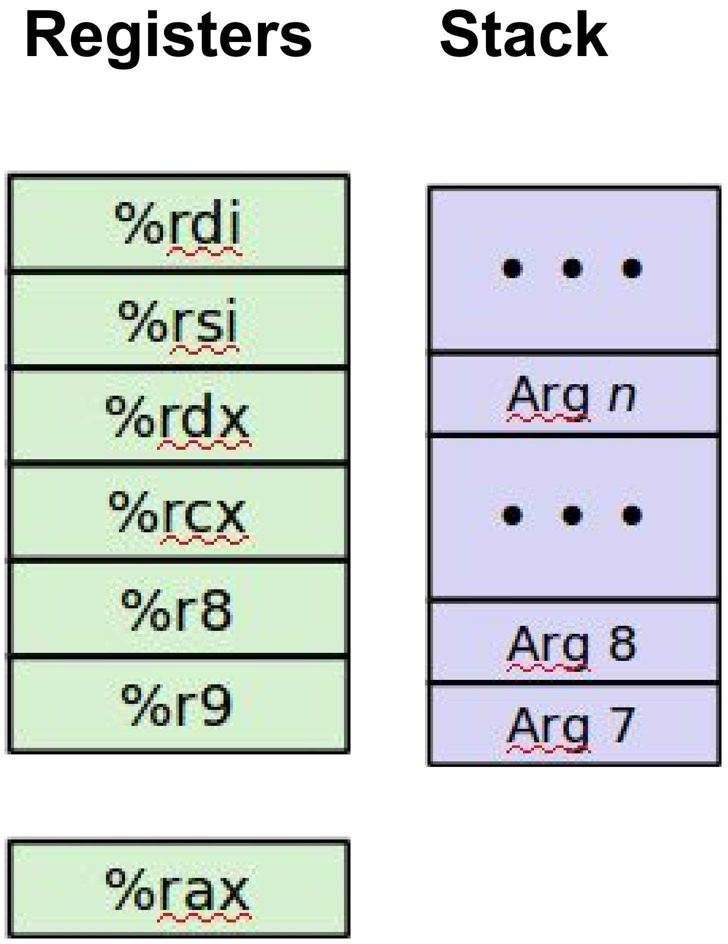
Passing arguments and returning values

Procedure arguments:

- Registers
  - First six integer/pointer arguments are placed in registers: `%rdi, %rsi, %rdx%, %rcx, %r8, %r9`
  - Note: you have to remember the order because that's how the arguments are mapped
- Stack
  - 7+ arguments (integer and pointer) saved on the stack
  - (in IA-32 all arguments were saved on the stack - accessing stack is slower than accessing the registers)

Return value:

- Register `%rax` is used to transfer a return value to the caller.



Example: Passing Data

```
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
  • • •
```

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

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

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

Local Data

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

Example: Function Call Chain

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

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

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

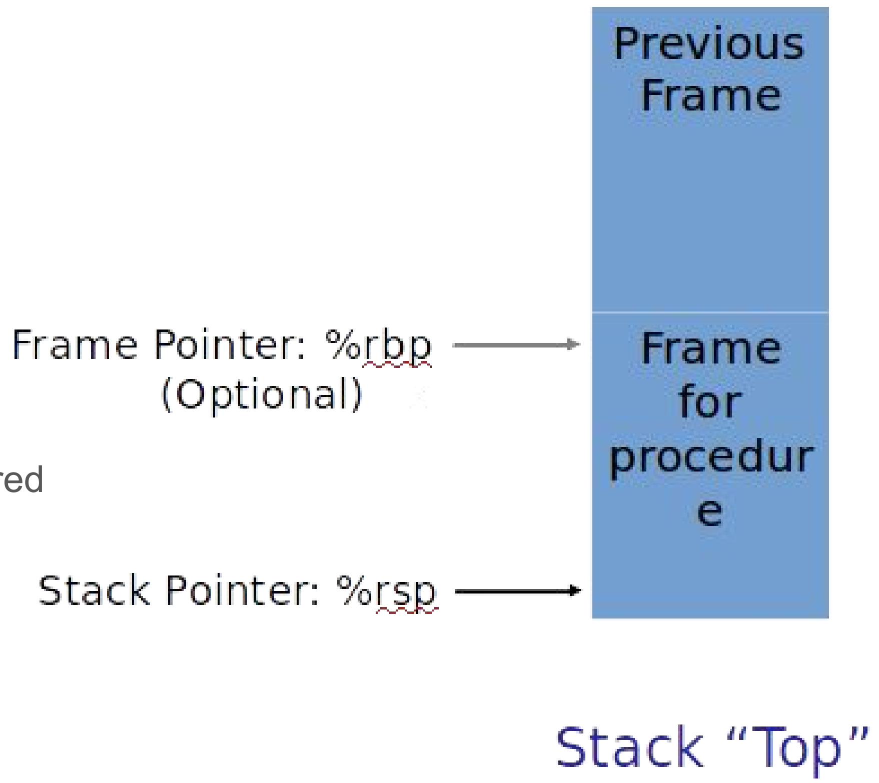
Example
Call Chain

$$\begin{tikzcd}
\text{yoo} \arrow[d] & \\
\text{who} \arrow[d] \arrow[dr] & \\
\text{amI} \arrow[d] & \text{amI} \\
\text{amI} \arrow[d] & \\
\text{amI} &
\end{tikzcd}$$

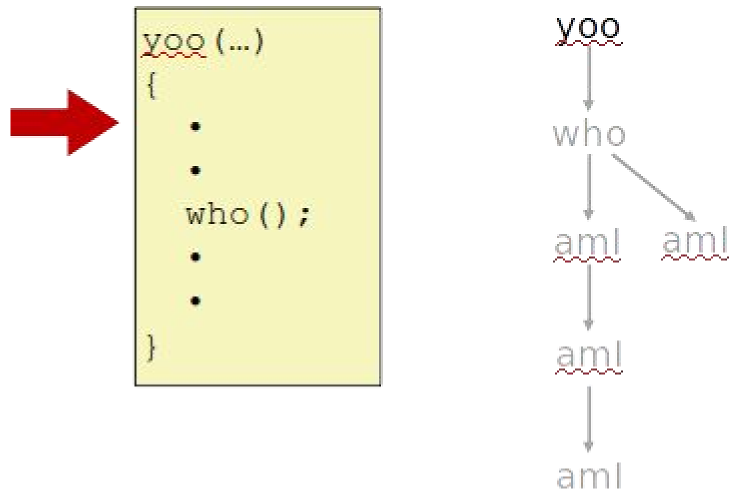
Procedure amI() is recursive

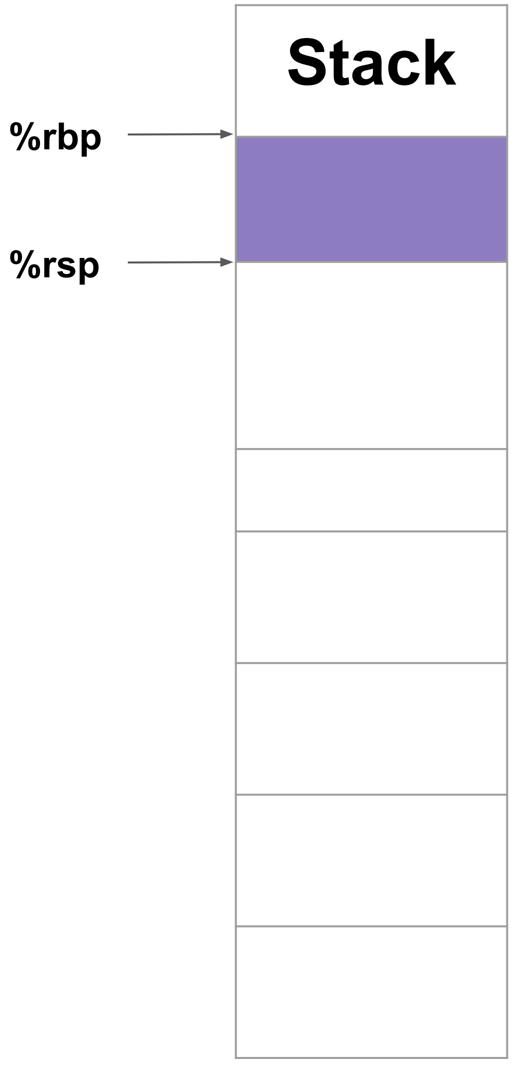
Stack Frames

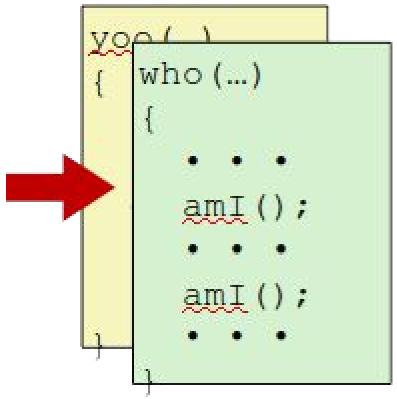
- Contents
  - Return information
  - Local storage (if needed)
  - Temporary space (if needed)
- Management
  - Space allocated when procedure is entered
    - "Set-up" code
    - Includes push by call instruction
  - Deallocated when return
    - "Finish" code
    - Includes pop by ret instruction



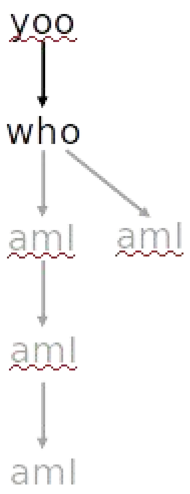
```
yoo(…)
{
    •
    •
    who();
    •
    •
}
```

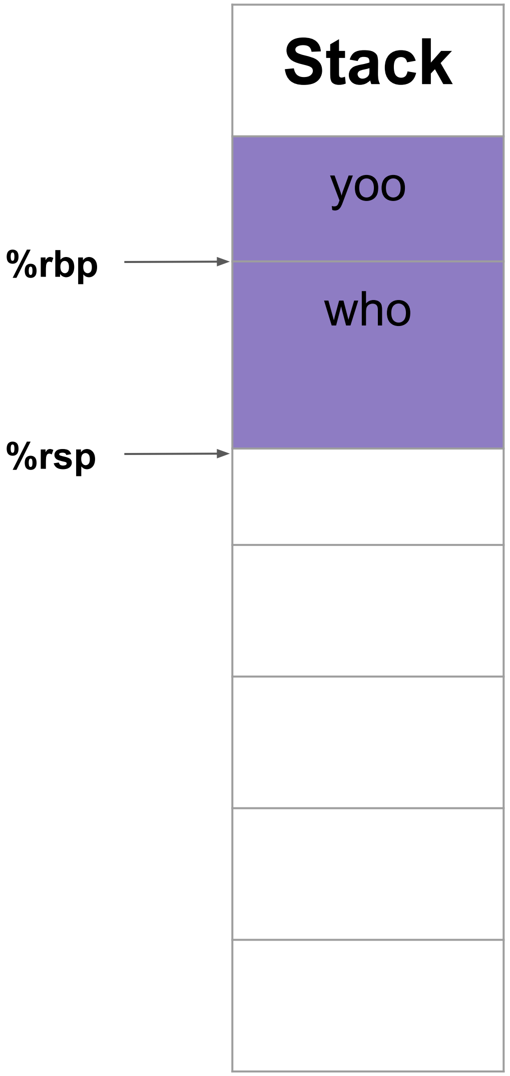


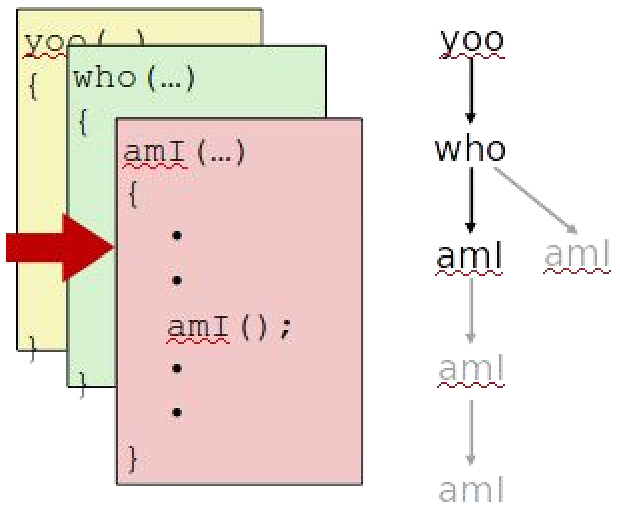


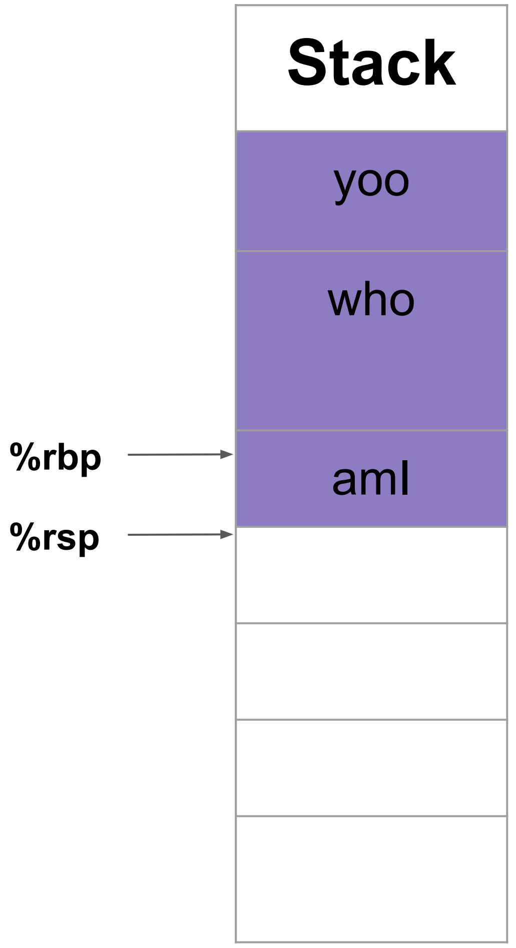


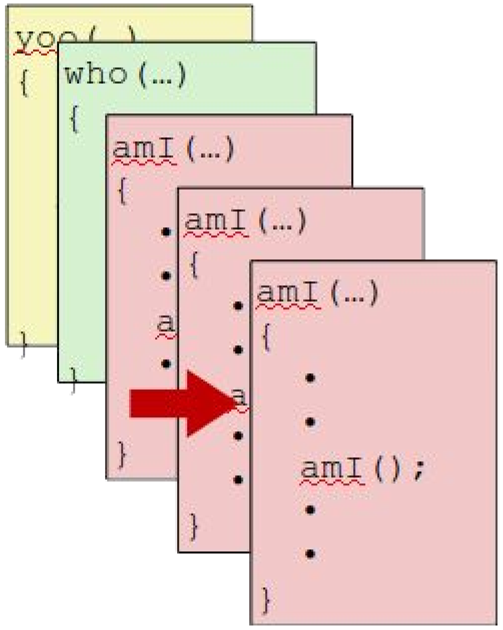
```
yoo(…)
{
  who(…)
  {
    • • •
    amI();
    • • •
    amI();
    • • •
  }
}
```

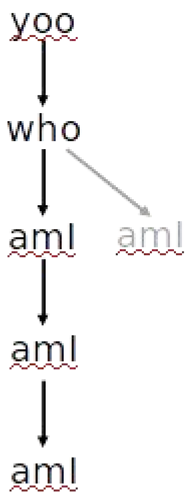


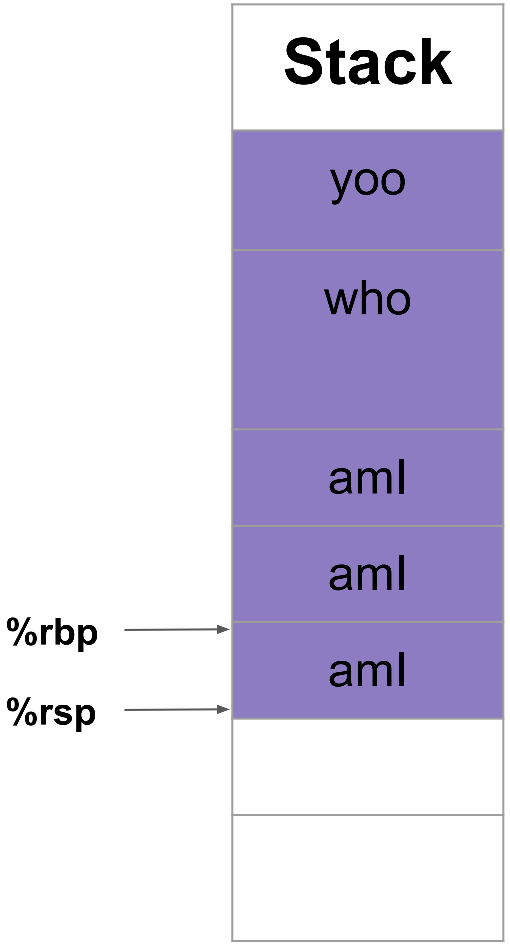


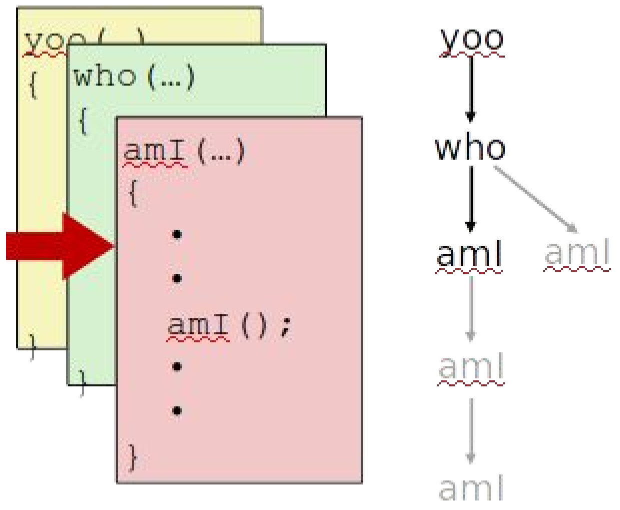

yoo(…)
{
}
who(…)
{
}
amI(…)
{
•
•
amI();
•
•
}
yoo
who
amI
amI
amI
amI

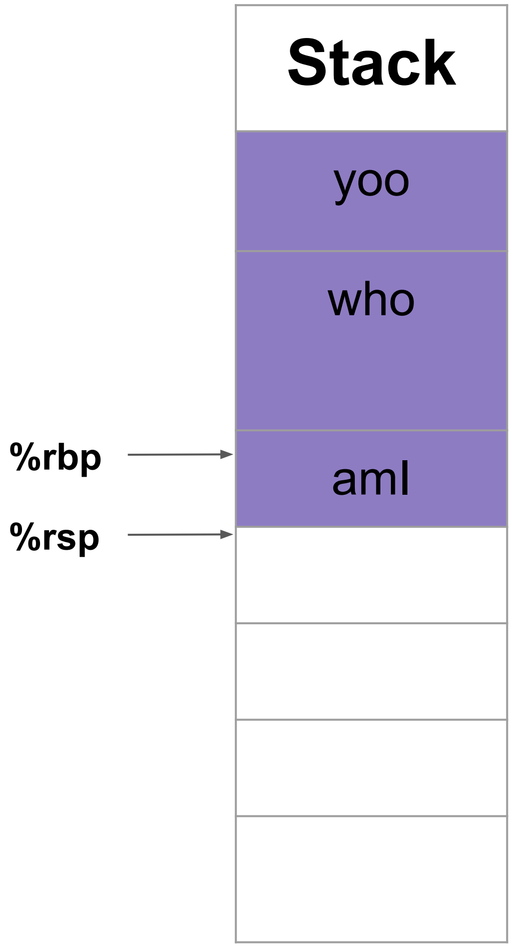

Stack
yoo
who
amI
%rbp
%rsp

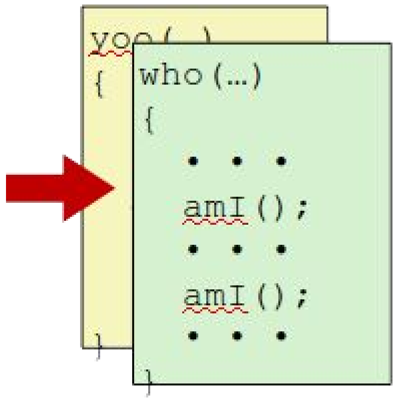

yoo(…)
{
}
who(…)
{
}
amI(…)
{
•
•
a
•
}
amI(…)
{
•
•
a
•
•
}
amI(…)
{
•
•
amI();
•
•
}


yoo
who
amI
amI
amI
amI

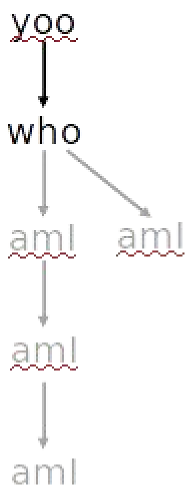

Stack
yoo
who
amI
amI
amI
%rbp
%rsp


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

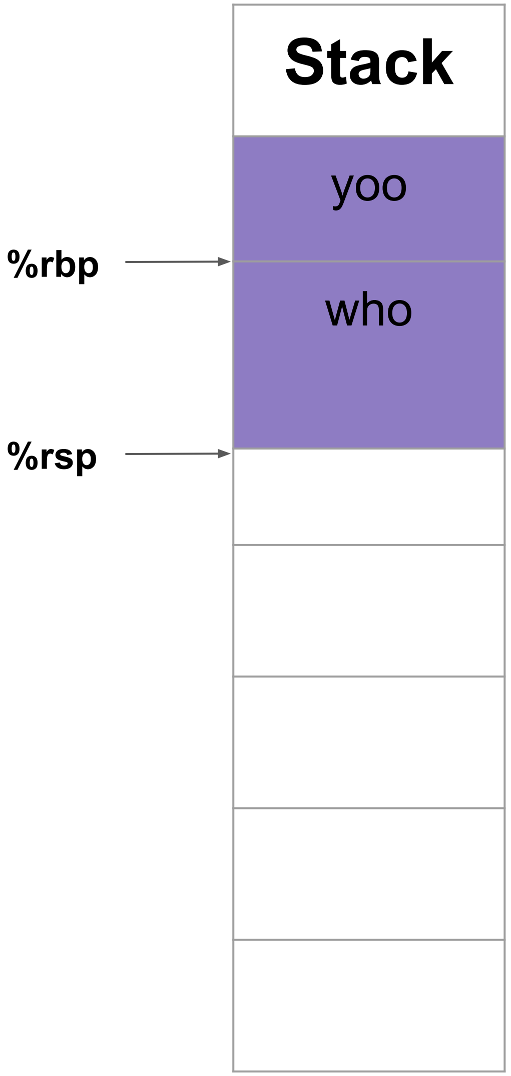

Stack
yoo
who
amI
%rbp
%rsp



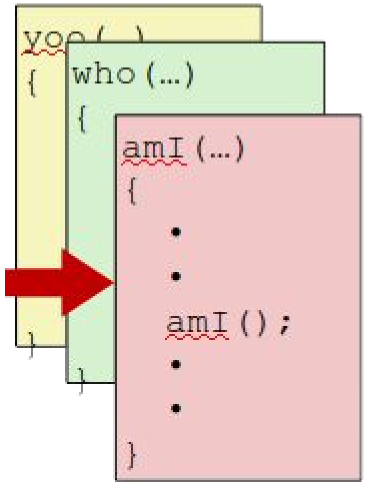
yoo(…)
{
}
who(…)
{
• • •
amI();
• • •
amI();
• • •
}

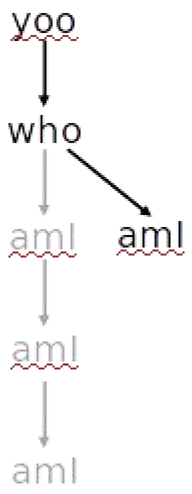


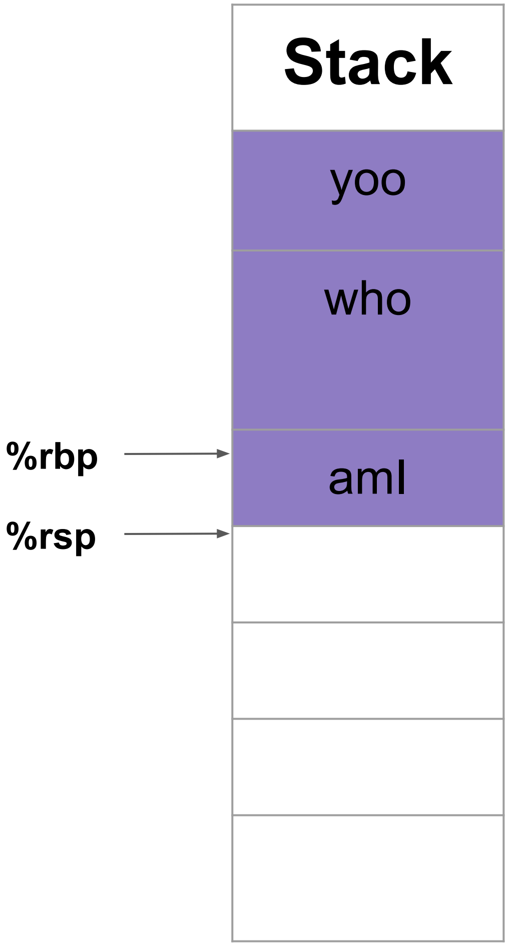
yoo
who
amI
amI
amI
amI



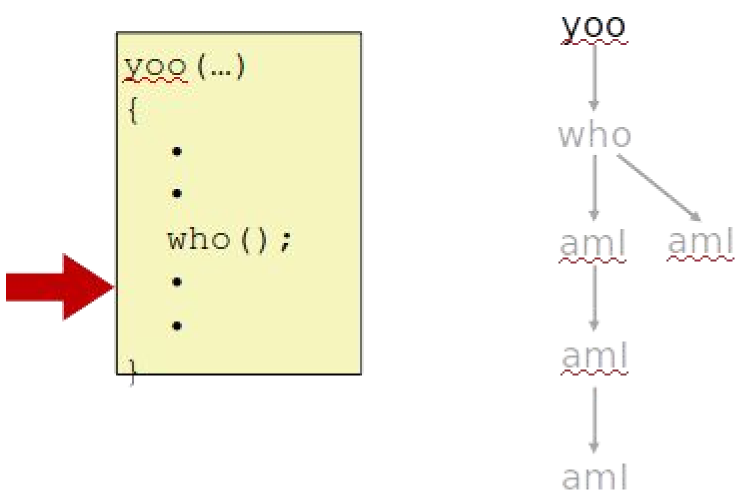
Stack
yoo
who
%rbp
%rsp

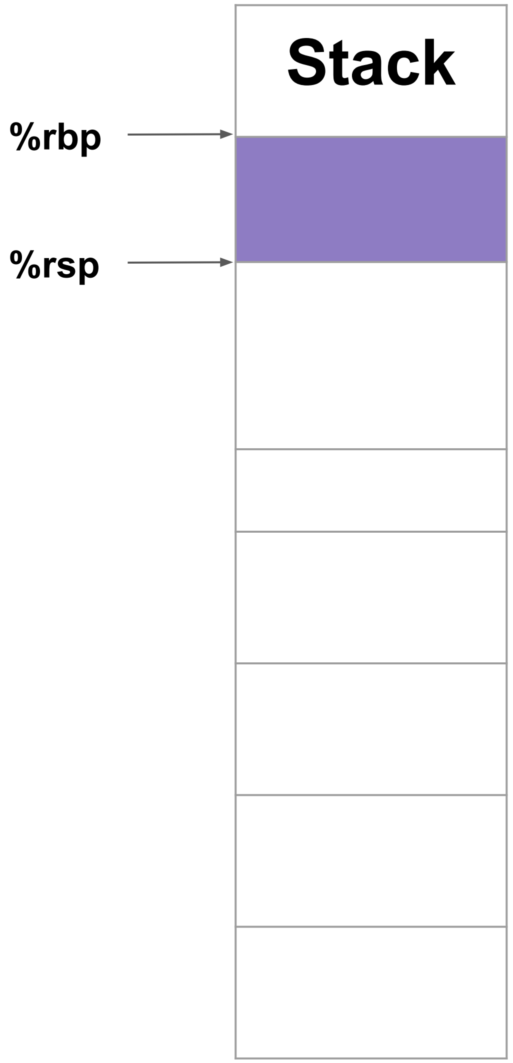

yoo(…)
{
}
who(…)
{
}
amI(…)
{
•
•
amI();
•
•
}


yoo
who
amI
amI
amI
amI


Stack
yoo
who
amI
%rbp
%rsp

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

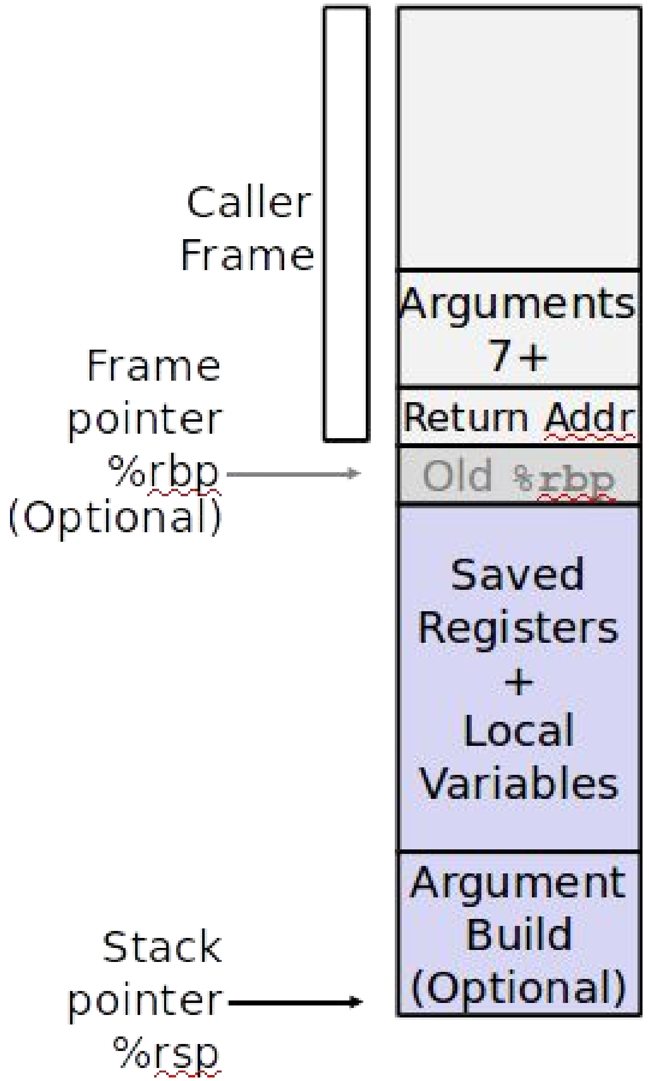




X86-64 Stack Frame

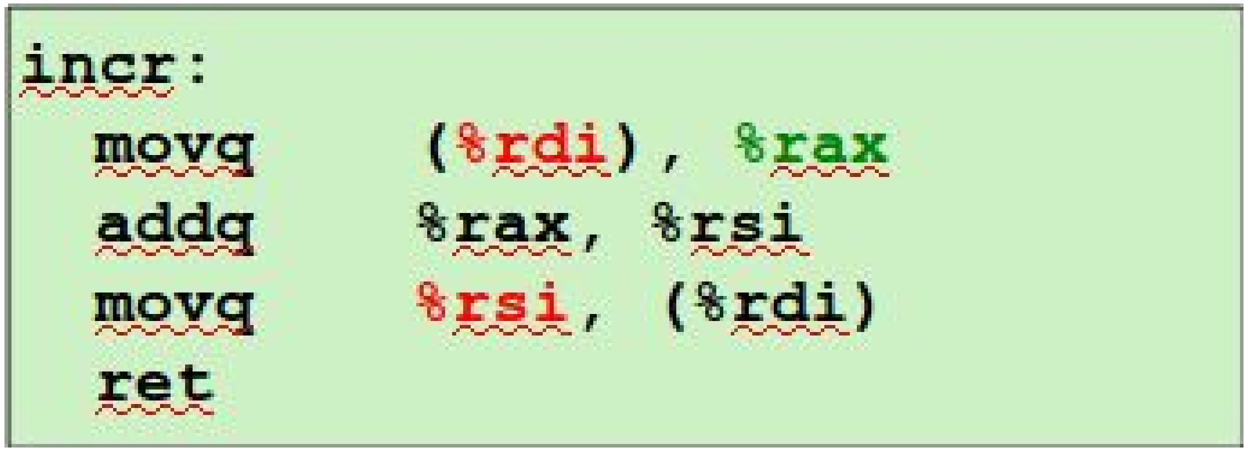
- Current Stack Frame ("Top" to Bottom)
  - "Argument build:"
    - Parameters for function about to call
    - Local variables
    - If can't keep in registers
    - Saved register context
    - Old frame pointer (optional)

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


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

Examples

What is the C function corresponding to this assembly function?



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

`incr` function

```
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
```

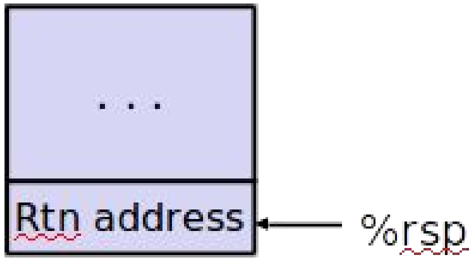
Register	Use(s)
`%rdi`	Argument `p`
`%rsi`	Argument `val`, `y`
`%rax`	`x`, Return value

Calling `incr` function

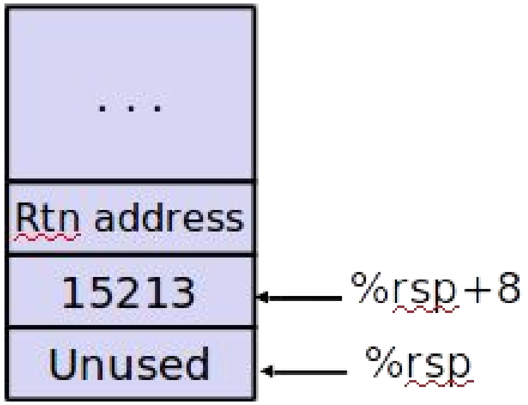
```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

```
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
```

Initial Stack Structure



Resulting Stack Structure

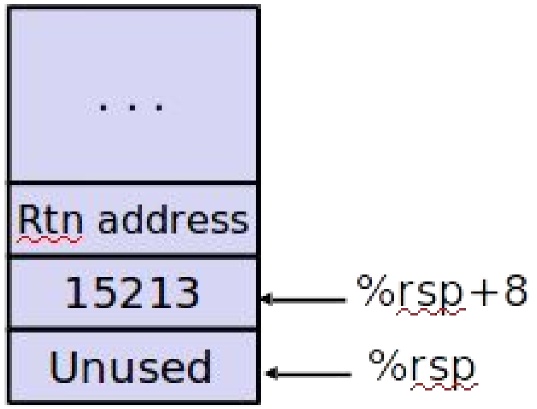


Calling `incr` function

```
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



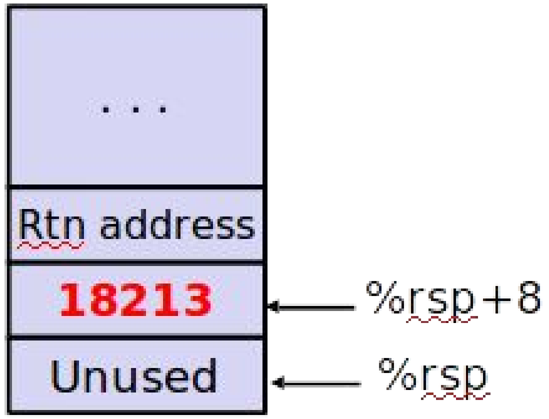
Register	Use(s)
%rdi	&v1
%rsi	3000

Calling `incr` function

```
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

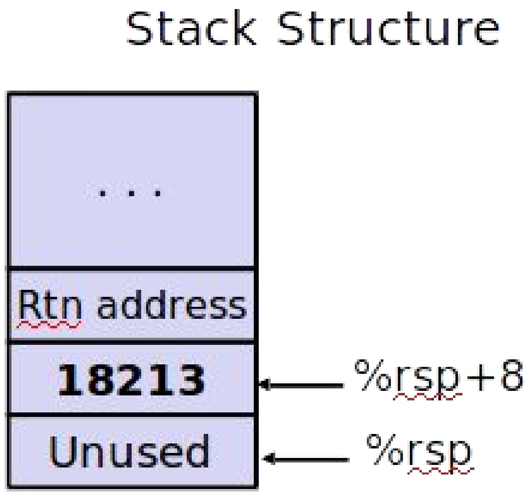


Register	Use(s)
%rdi	&v1
%rsi	3000

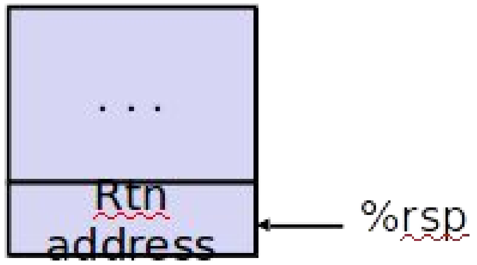
Calling `incr` function

```
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

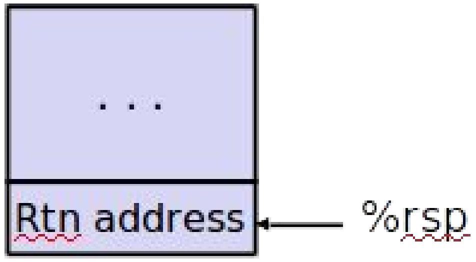


Calling `incr` function

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

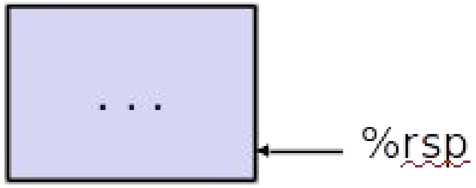
```
call_incr:
  subq    $16, %rsp
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

Updated Stack Structure



Register	Use(s)
%rax	Return value

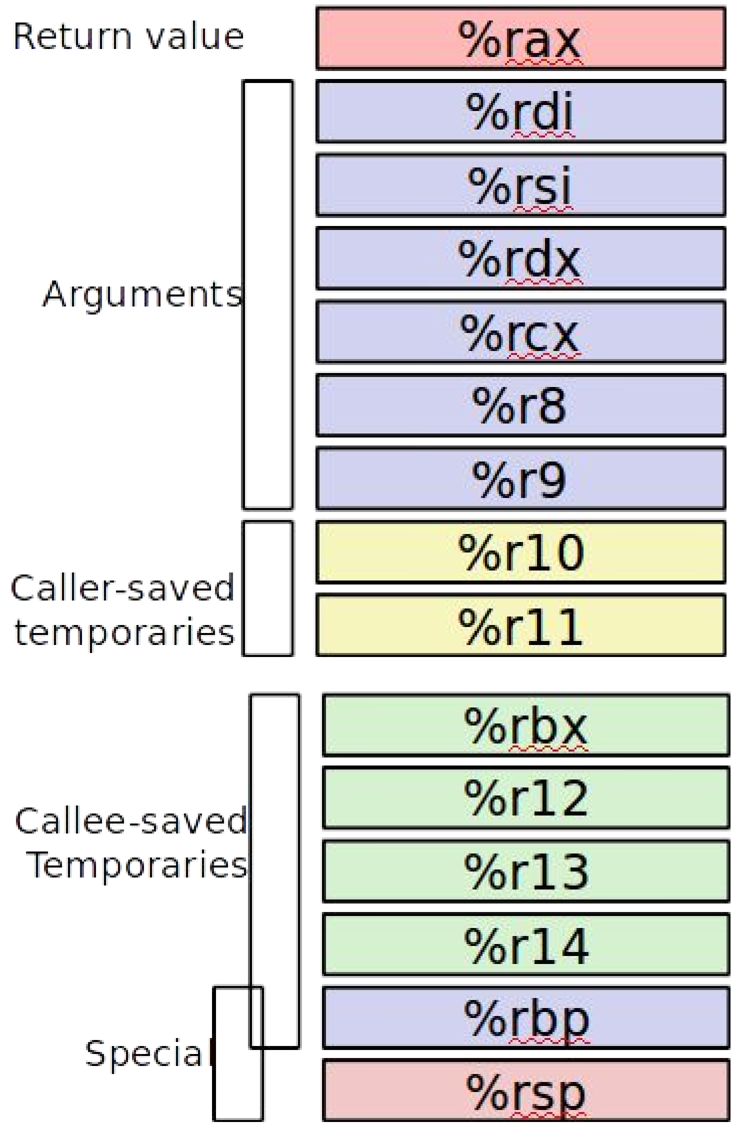
Final Stack Structure



Register Saving Conventions

- When procedure `yoo` calls `who`:
    - `yoo` is the **caller**
    - `who` is the **callee**
- Can 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)

Register Saving Convention

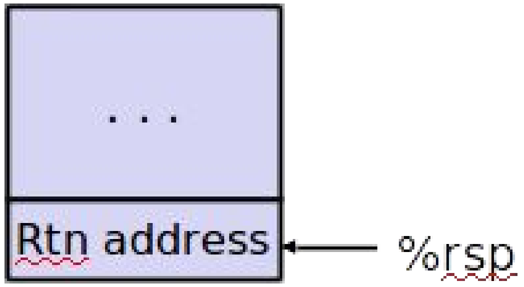


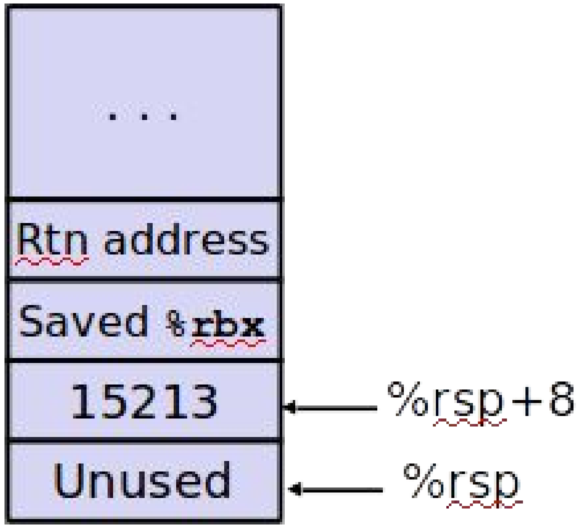
- %rax
  - Return value
  - Also caller-saved
  - Can be modified by procedure
- %rdi, ..., %r9
  - Arguments
  - Also caller-saved
  - Can be modified by procedure
- %r10, %r11
  - Caller-saved
  - Can be modified by procedure
- %rbx, %r12, %r13, %r14
  - Callee-saved
  - Callee must save & restore
- %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

```
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

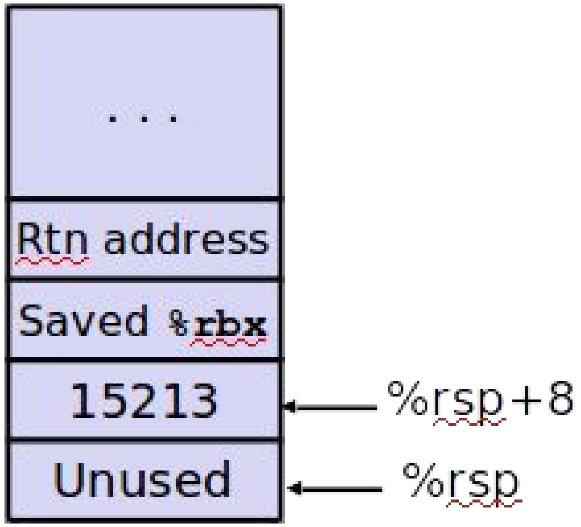




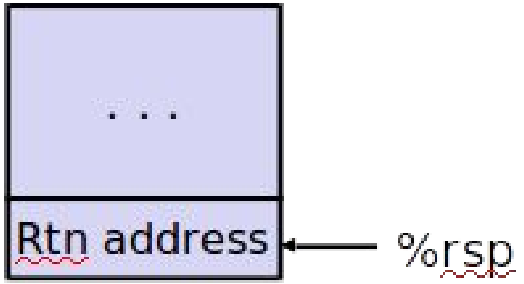
Resulting Stack Structure

```
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
```



. . .
Rtn address
Saved %rbx
15213
Unused
%rsp+8
%rsp



. . .
Rtn address
%rsp