+

Conditional Branches

Jumping

- jX Instructions
  - Jump to different part of code depending on condition codes

jX	Condition	Description
`jmp`	`1`	Unconditional
`je`	`ZF`	Equal / Zero
`jne`	`~ZF`	Not Equal / Not Zero
`js`	`SF`	Negative
`jns`	`~SF`	Nonnegative
`jg`	`~(SF^OF)&~ZF`	Greater (Signed)
`jge`	`~(SF^OF)`	Greater or Equal (Signed)
`jl`	`(SF^OF)`	Less (Signed)
`jle`	`(SF^OF)|ZF`	Less or Equal (Signed)
`ja`	`~CF&~ZF`	Above (unsigned)
`jb`	`CF`	Below (unsigned)

Conditional Branching by Jumping

```
long absdiff(long x, long y)
{
    long result;
    if (x > y)
        result = x - y;
    else
        result = y - x;
    return result;
}
```

```
absdiff:
    cmpq    %rsi, %rdi # y, x
    jle     .L4
    movq    %rdi, %rax
    subq    %rsi, %rax
    ret
.L4:                   # x <= y
    movq    %rsi, %rax
    subq    %rdi, %rax
    ret
```

- Note: must use ***–fno-if-conversion*** argument to gcc, otherwise assembly will not use jumps in this program, we'll see why in a moment.

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

Rewriting C with `goto` Statements

- C allows **`goto`** statement
- Jump to position designated by label…

```
long absdiff(long x, long y)
{
    long result;
    if (x > y)
        result = x-y;
    else
        result = y-x;
    return result;
}
```

```
long absdiff_j(long x, long y)
{
    long result;
    int ntest = x <= y;
    if (ntest) goto Else;
    result = x-y;
    return result;
 Else:
    result = y-x;
    return result;
}
```

- **`goto`** form resembles assembly instructions using jumps

Rewriting C with `goto` Statements *con't*

- C code

```
val = test ? then_expr : else_expr;
```

- Example

```
val = x > y ? x - y : y - x;
```

- Goto version

```
  if (!test) goto Else;
  val = then_expr;
  goto Done:
Else:
  val = else_Expr;
Done:
  return val;
```

- Create separate code regions for then & else expressions
- Execute appropriate one
- This is how we can think about 'jumping' in assembly

Alternate Approach: Conditional Moves

- **Conditional Move Instructions**
  - Instruction supports:
    - if (test) dest <- src
  - GCC tries to use them, but, only when known to be safe
- **Why?**
  - Branches are very disruptive to instruction flow through pipelines
  - Conditional moves do not require control transfer

C Code

```
val = test ? then_exp : else_exp;
```

Goto Version

```
result = then_expr;
eval = else_expr;
neg_test = !test;
if (neg_test) result = eval;
return result;
```

Conditional Move Example

```
long absdiff(long x, long y){
    long result;
    if (x > y)
        result = x-y;
    else
        result = y-x;
    return result;
}
```

Register	Use(s)
%rdi	Argument **x**
%rsi	Argument **y**
%rax	Return value
%rdx	Temp variable

```
absdiff:
   movq     %rdi, %rax   # x
   subq     %rsi, %rax   # if-val = x-y
   movq     %rsi, %rdx
   subq     %rdi, %rdx   # else-val = y-x
   cmpq     %rsi, %rdi   # x:y
   cmovle   %rdx, %rax   # if y <= x, result = eval
   ret
```

Bad Cases for Conditional Move

- **Expensive computations**

```
val = Test(x) ? Hard1(x) : Hard2(x);
```

  - Both values get computed
  - Only makes sense when computations are very simple

- **Risky computations**

```
val = p == 0 ? 0 : 5 / p;
```

  - Both values get computed
  - May have undesirable effects

- **Computations with side effects**

```
val = x > 0 ? x *= 7 : x += 3;
```

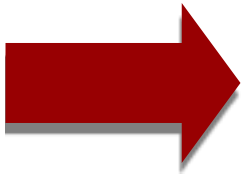
  - Both values get computed
  - Must be side-effect free

Loops

General "Do-While" Translation

C Code

```
do
   Body
   while (Test);
```



Goto Version

```
loop:
   Body
   if (Test)
      goto loop
```

Body

```
{
   statement1;
   statement2;
      …
   statementn;
}
```

"Do-While" Loop Example

C Code

```
long pcount_do(unsigned long x)
{
  long result = 0;
  do {
    result += x & 0x1;
    x >>= 1;
  } while (x);
  return result;
}
```

Goto Version

```
long pcount_goto(unsigned long x)
{
  long result = 0;
 loop:
  result += x & 0x1;
  x >>= 1;
  if(x) goto loop;
  return result;
}
```

- Count number of 1's in argument x ('popcount')
- Use conditional jump to either continue looping or to exit loop

"Do-While" Loop Compilation

```
long pcount_goto(unsigned long x){
   long result = 0;
 loop:
   result += x & 0x1;
   x >>= 1;
   if(x) goto loop;
   return result;
}
```

Register	Use(s)
%rdi	Argument x
%rax	result

```
   movl     $0, %rax          #  result = 0
.L2:                          # loop:
   movq     %rdi, %rdx
   andq     $1, %rdx          #  t = x & 0x1
   addq     %rdx, %rax        #  result += t
   shrq     %rdi              #  x >>= 1
   jne      .L2               #  if (x) goto loop
   rep; ret
```

- Note: some processors' branch predictors behave badly when a branch's target or fall-through is a **ret** instruction, and adding the **rep;** prefix avoids this.

General "While" Translation

- "Jump-to-middle" translation
- Used with **gcc -Og** (our setting)

While Version

```
while (Test)
  Body
```

Goto Version

```
  goto test;
loop:
  Body
test:
  if (Test)
    goto loop;
done:
```

While Loop Example

C Code

```
long pcount_while(unsigned long x)
{
  long result = 0;
  while (x) {
    result += x & 0x1;
    x >>= 1;
  }
  return result;
}
```

Jump to Middle

```
long pcount_goto_jtm(unsigned long x)
{
  long result = 0;
  goto test;
 loop:
  result += x & 0x1;
  x >>= 1;
 test:
  if(x) goto loop;
  return result;
}
```

- Compare to do-while version of function
- Initial goto starts loop at test

For Loop: Derived From While

For Version

```
for (Init; Test; Update )
   Body
```

While Version

```
Init;

while (Test) {
   Body
   Update;
}
```

For-While Conversion

```
#define WSIZE 8*sizeof(int)
long pcount_for(unsigned long x)
{
  int i;
  long result = 0;
  for (i = 0; i < WSIZE; i++)
  {
    unsigned bit = (x >> i) & 0x1;
    result += bit;
  }
  return result;
}
```

```
long pcount_for_while(unsigned long x)
{
  size_t i;
  long result = 0;
  i = 0;
  while (i < WSIZE)
  {
    unsigned bit = (x >> i) & 0x1;
    result += bit;
    i++;
  }
  return result;
}
```

Init

```
i = 0
```

Test

```
i < WSIZE
```

Update

```
i++
```

Body

```
{
  unsigned bit =
     (x >> i) & 0x1;
  result += bit;
}
```

"For" Loop Do-While Conversion

```
long pcount_for(unsigned long x)
{
  size_t i;
  long result = 0;
  for (i = 0; i < WSIZE; i++)
  {
    unsigned bit =
      (x >> i) & 0x1;
    result += bit;
  }
  return result;
}
```

```
long pcount_for_goto_dw(unsigned long x) {
  size_t i;
  long result = 0;
  i = 0;
  goto TEST
 LOOP:
  {
    unsigned bit =
      (x >> i) & 0x1;
    result += bit;
  }
  i++;
 TEST:
  if (i < WSIZE)
    goto LOOP;
 DONE:
  return result;
}
```

Init

Test *(jump to middle)*

Body

Update

Test

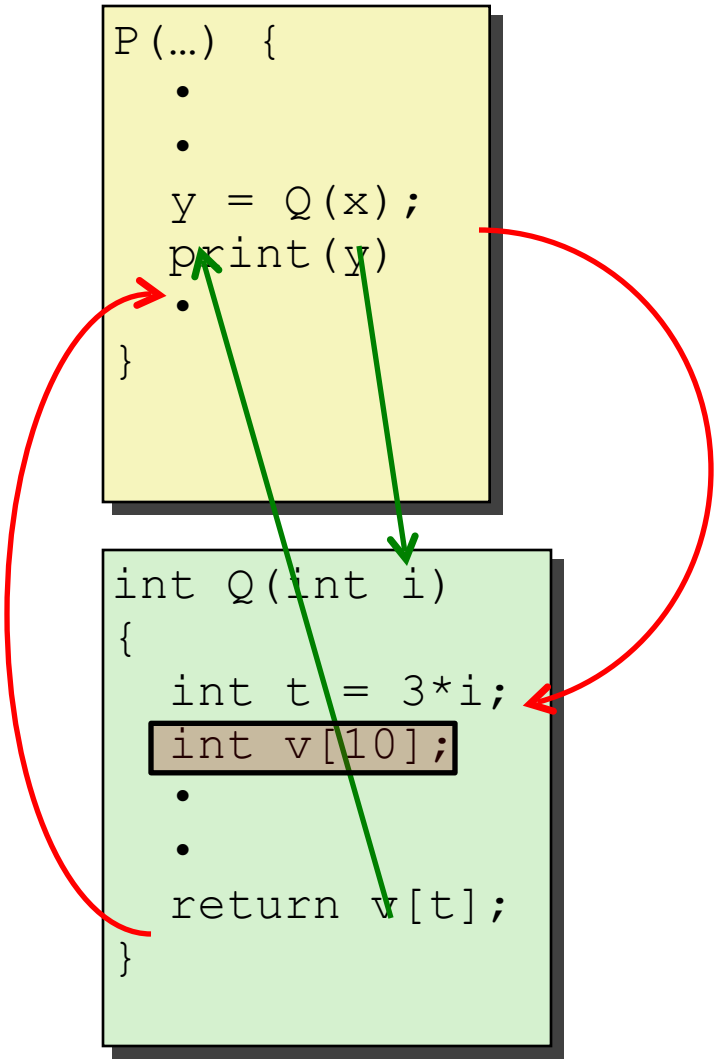
Initial test may be optimized away if compiler knows its safe

+

Procedures

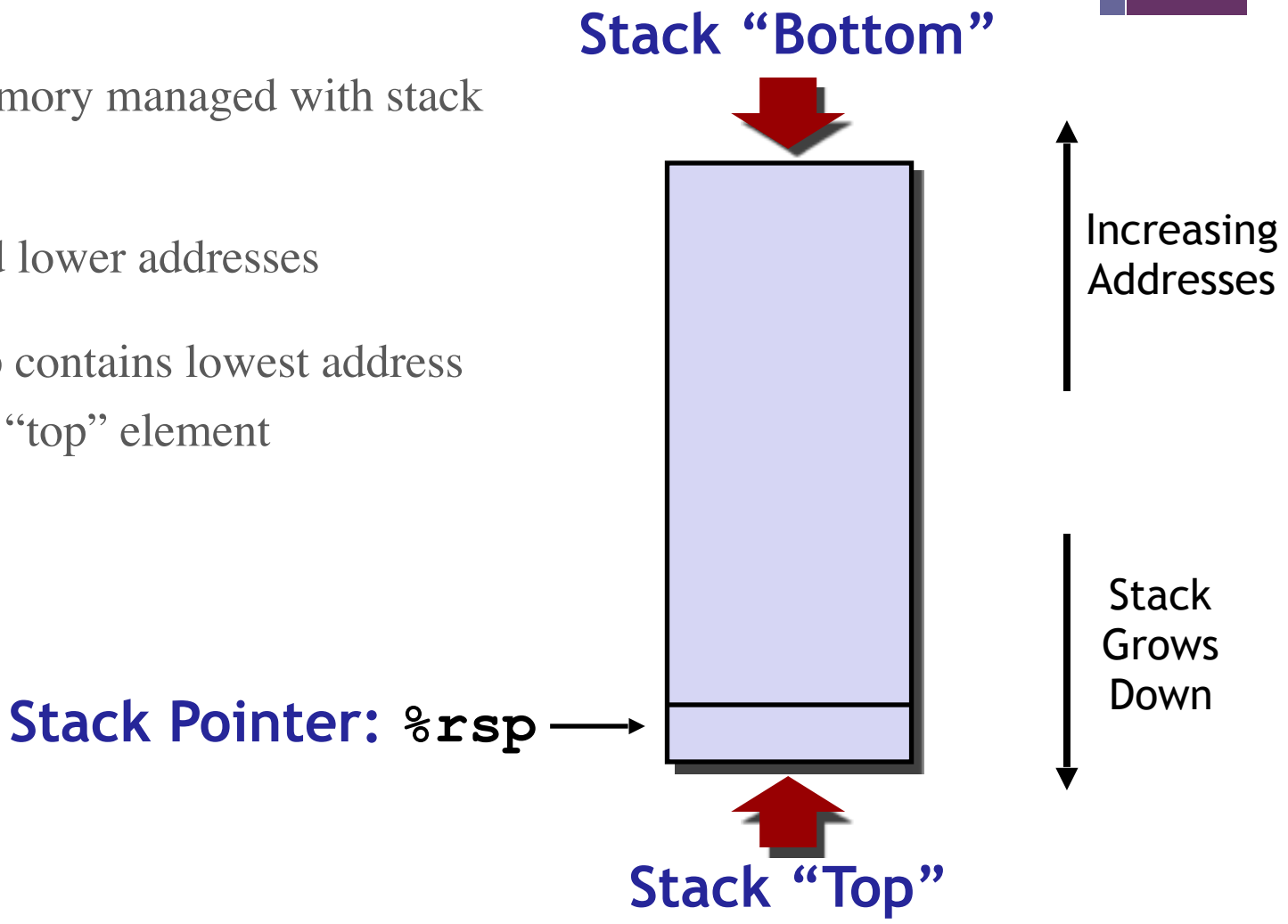
Mechanisms in Procedures

- **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**



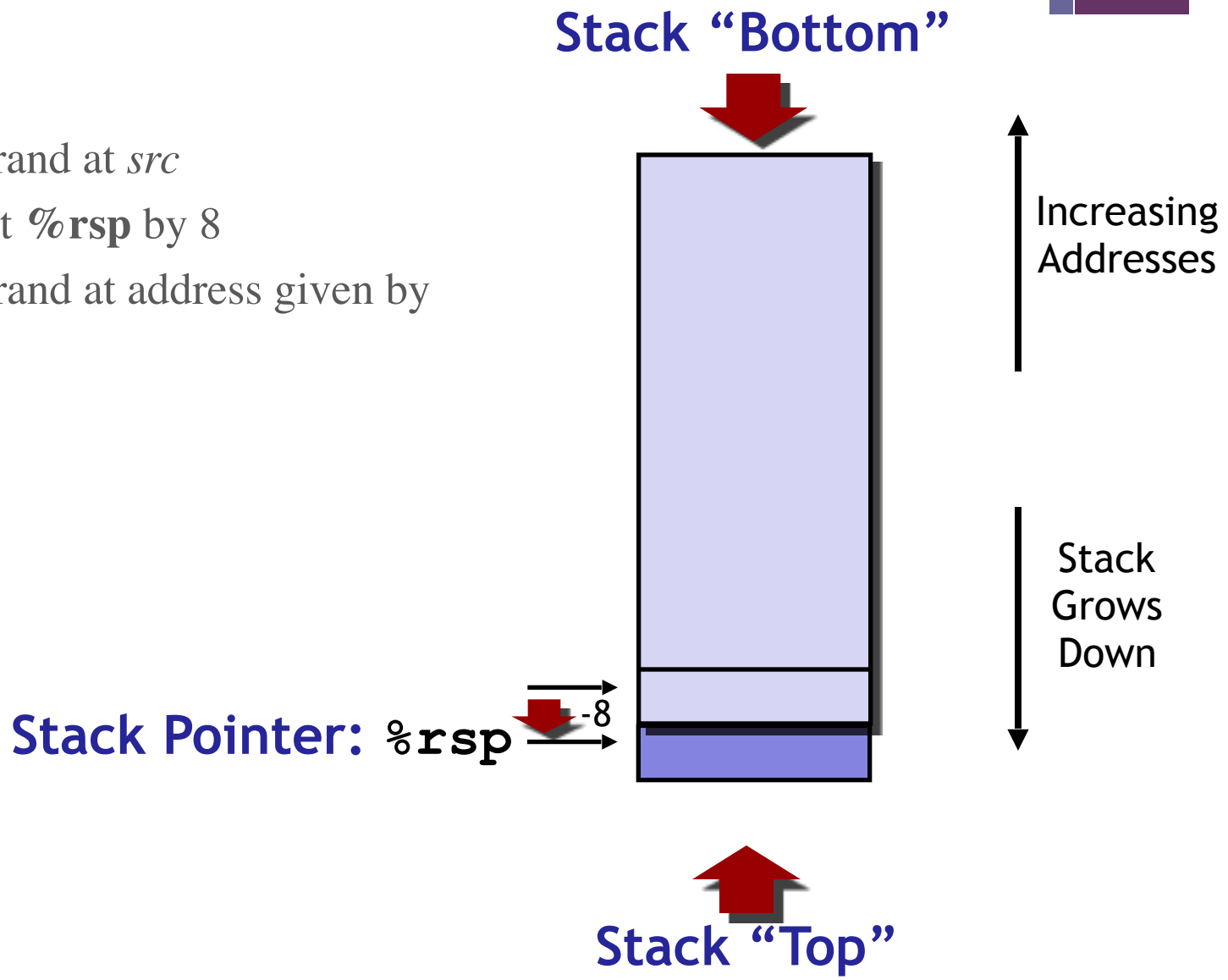
x86-64 Stack

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



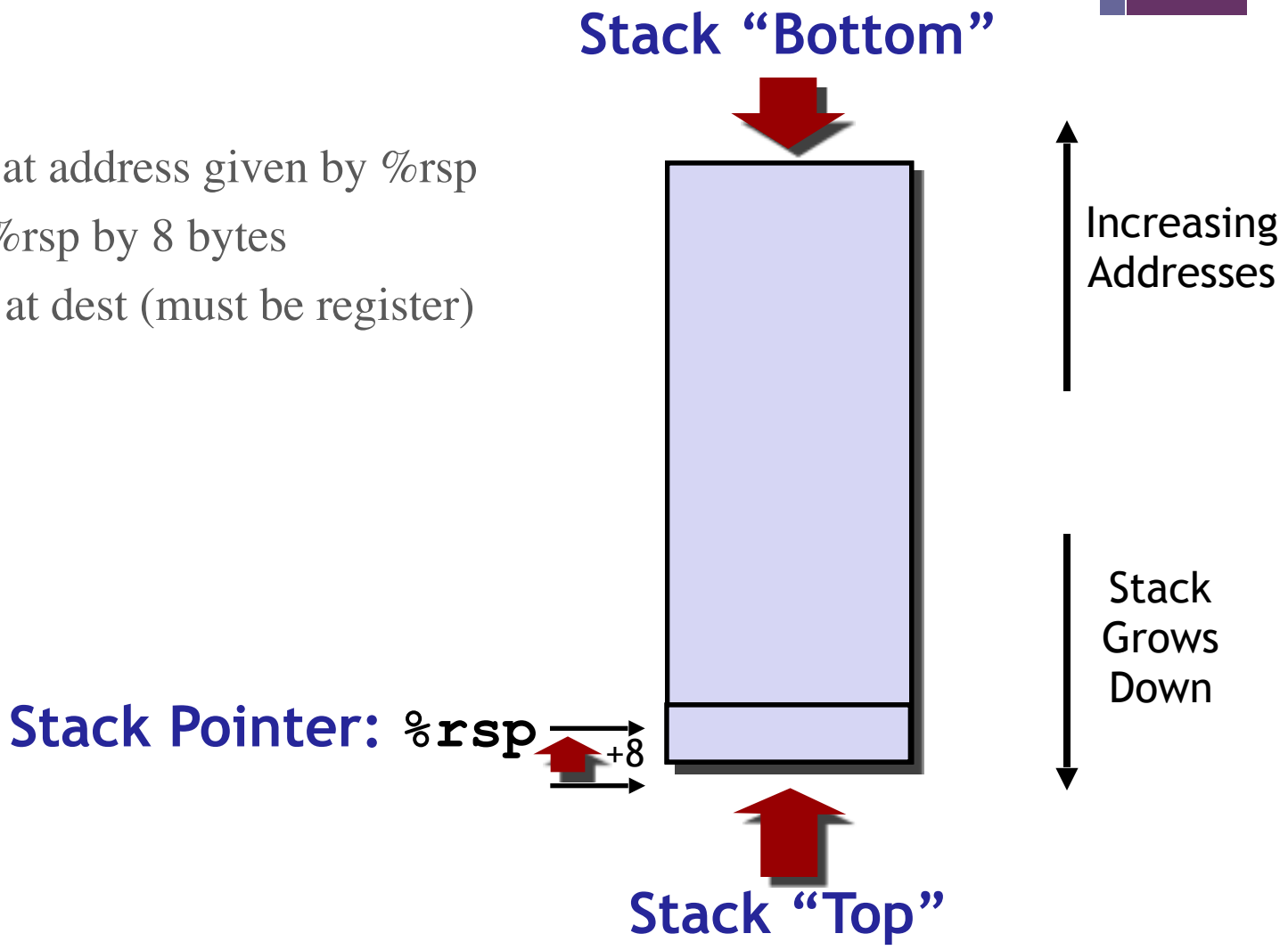
x86-64 Stack: Push

- **`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 bytes
  - Store value at dest (must be register)



+

Passing Control

Code Examples

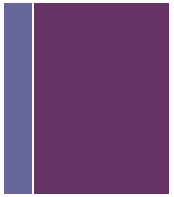
```
void multstore (long x, long y, long *dest){
    long t = mult2(x, y);
    *dest = t;
}
```

```
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

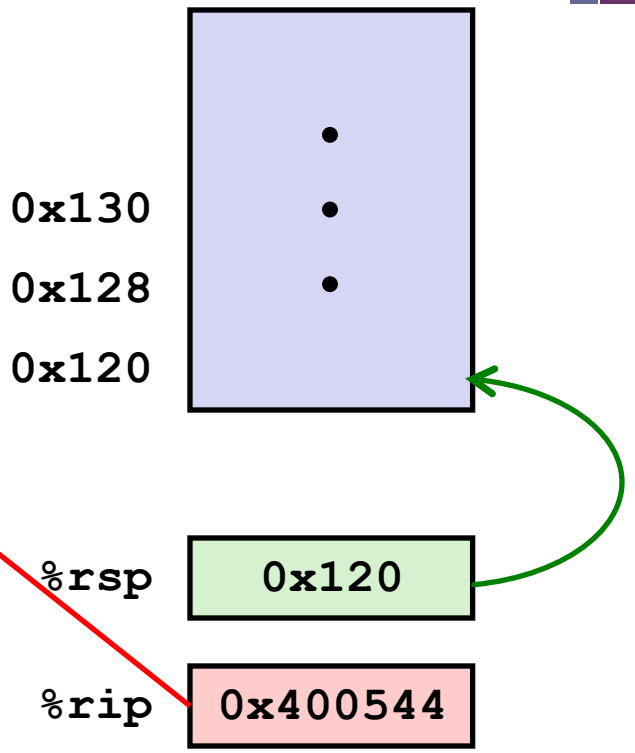


- Use stack to support procedure call and return
- **Procedure `call` with label**
  - Pushes ***return address*** on stack
    - Address of the next instruction right after call
  - Jumps to ***label***
- **Procedure return: `ret`**
  - Pops ***return address*** from stack
  - Jumps to ***return address***

Control Flow Example #1

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

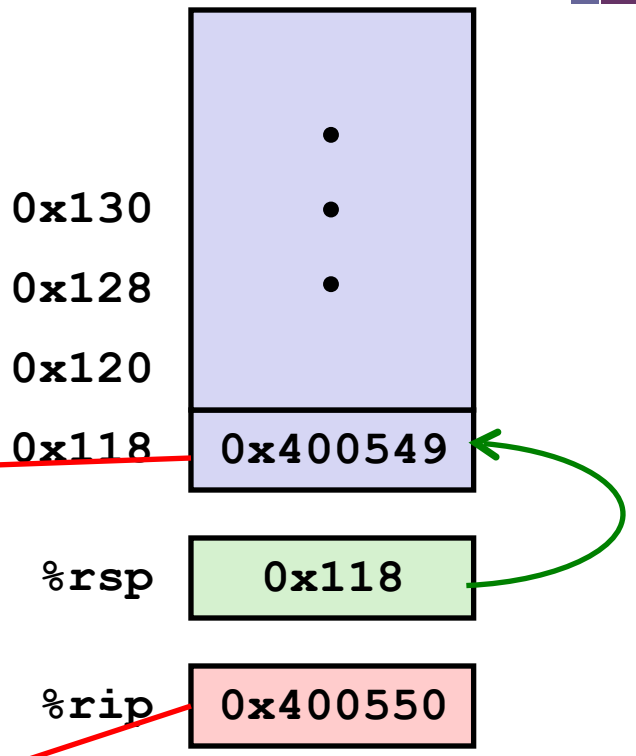
```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  ret
```



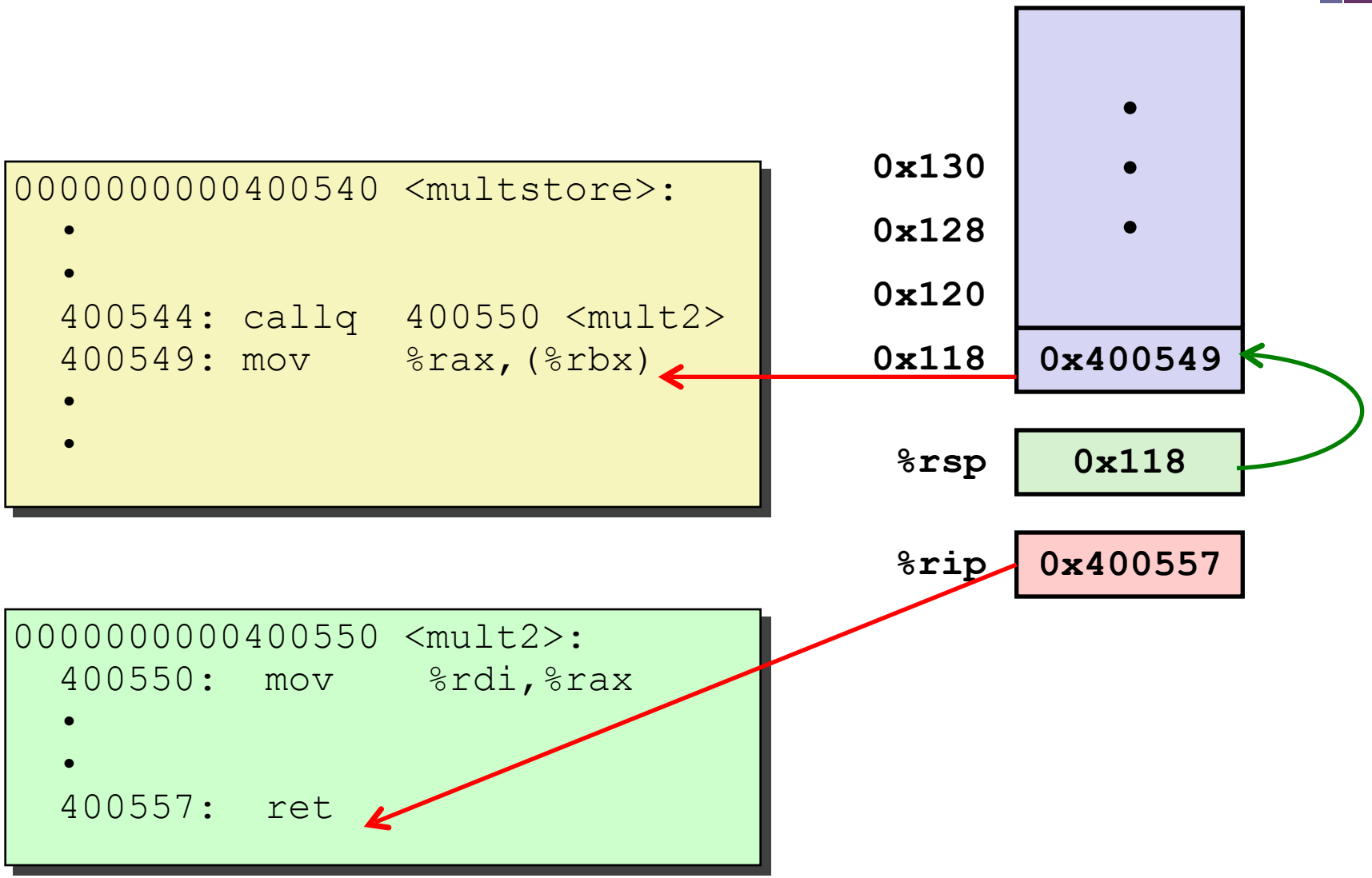
Control Flow Example #2

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

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



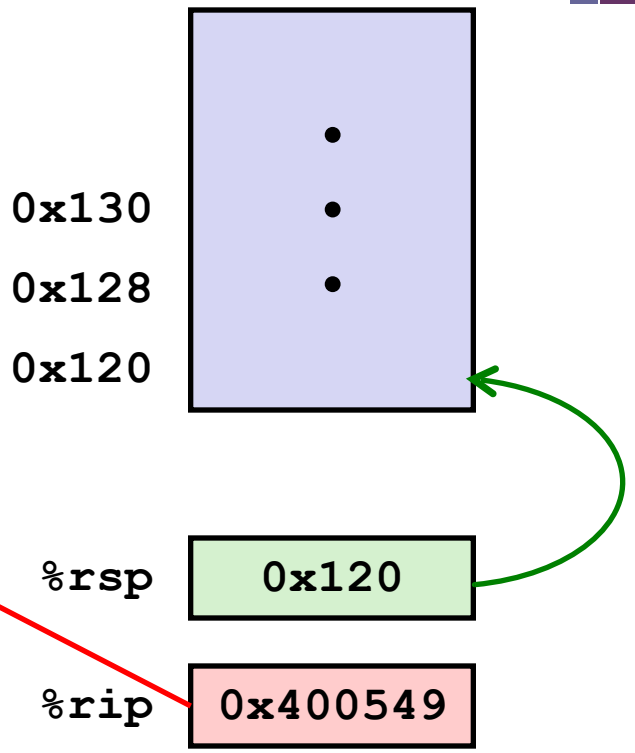
Control Flow Example #3



Control Flow Example #4

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

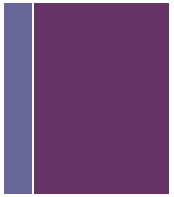
```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  ret
```



+

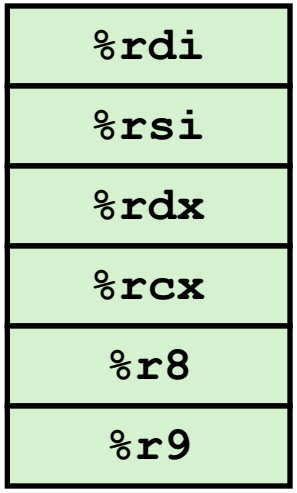
Passing Data

Procedure Data Flow

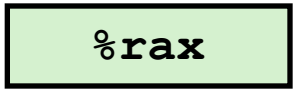


Registers

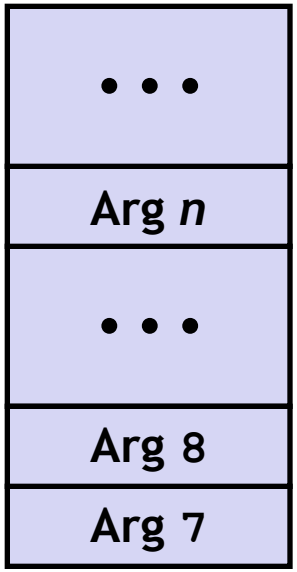
First 6 arguments



Return value



Stack



- 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
  • • •
```

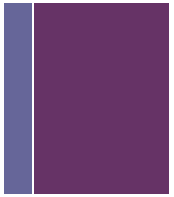
```
long mult2(long a, long b){
  long s = a * b;
  return s;
}
```

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

+

Managing Local Data

Stack Frames



- **Functions have "instantiations"**
  - Every function call is a distinct execution with distinct data.
  - Need some place to store state of each instantiation
    - Arguments
    - Local variables
    - Return pointer (next instruction in caller)
- **Stack allocated in *frames***
  - State for single procedure instantiation
  - Moreover, an allocation of memory holding all the data for some function call.
- **Recursion**
  - Supported by this idea of *instantiation* and *stack discipline*.

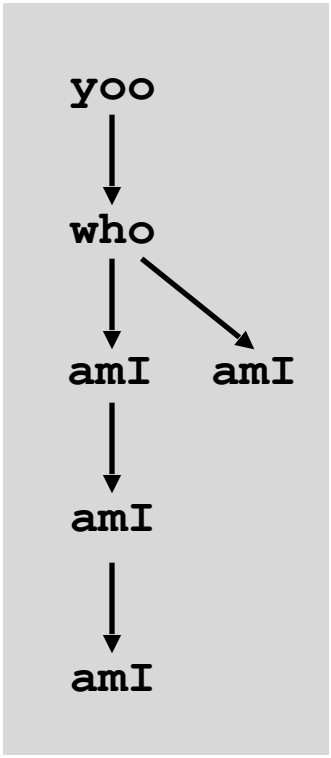
Call Chain Example

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

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

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

Example Call Chain

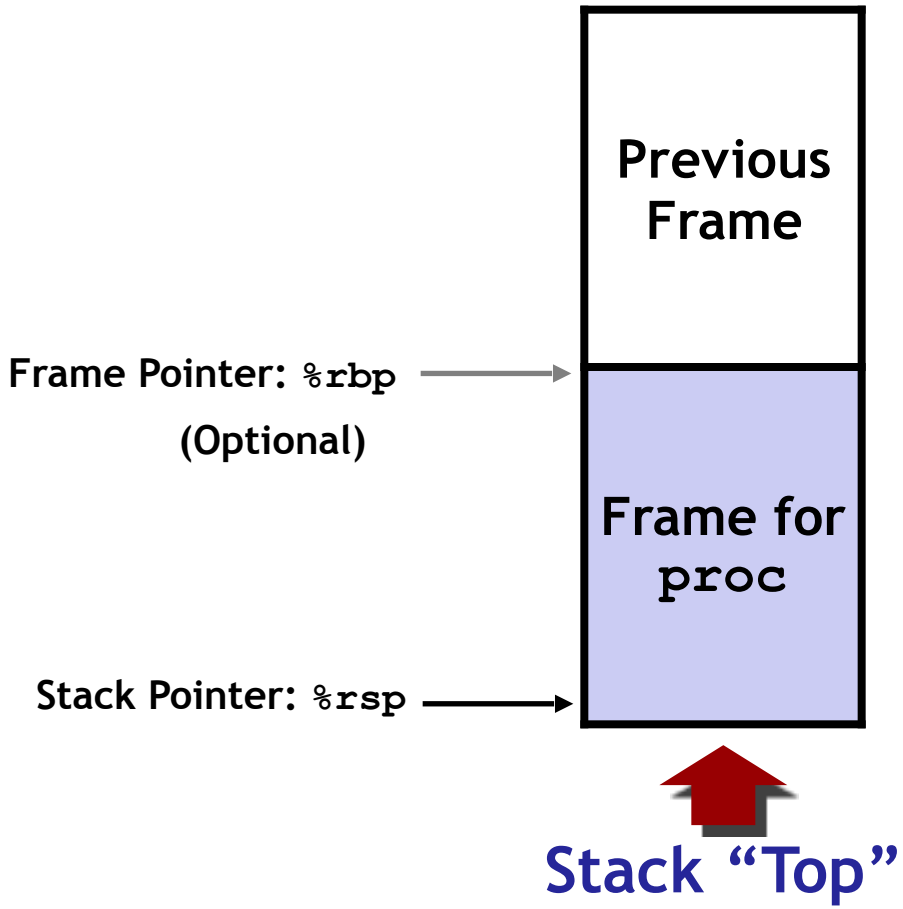


Procedure `amI()` is recursive

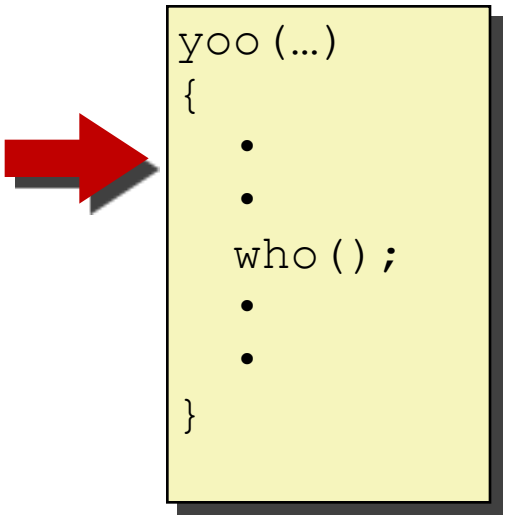
Stack Frames

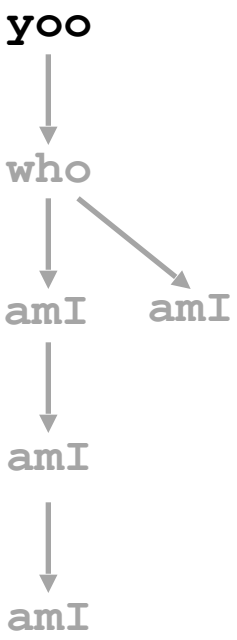
- **Contents**
  - Return information
  - Local storage (if needed)

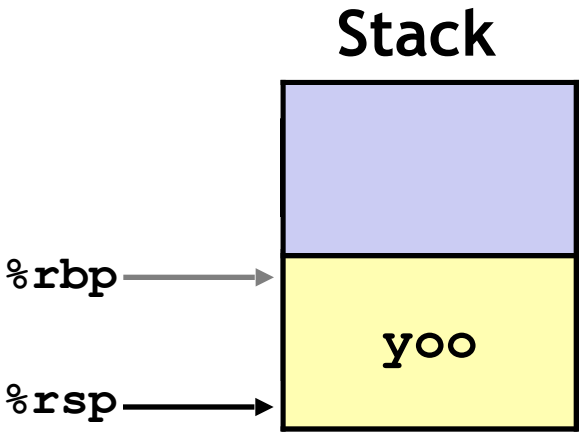
- **Management**
  - Space allocated on procedure call
    - push by **`call`** instruction
  - Space deallocated on return
    - pop by **`ret`** instruction



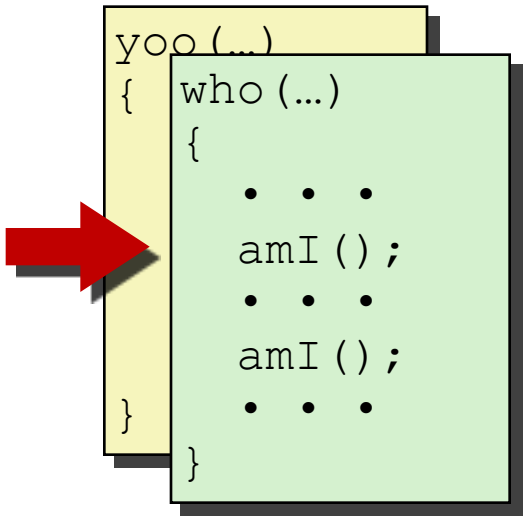
Example

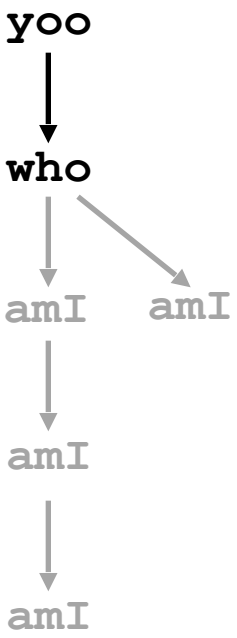


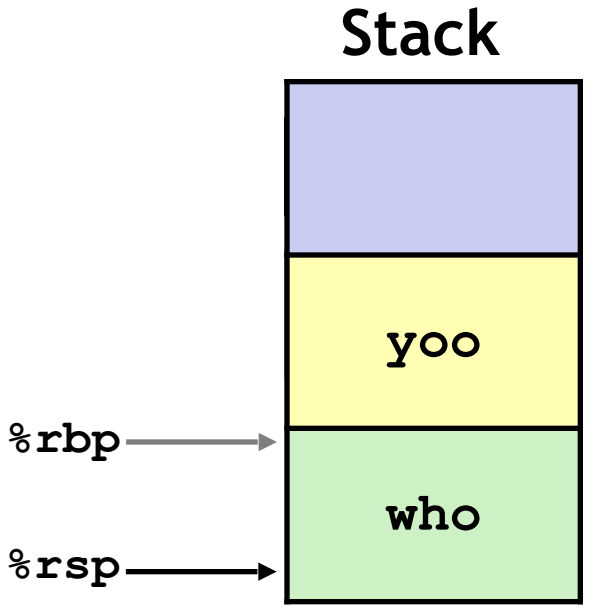




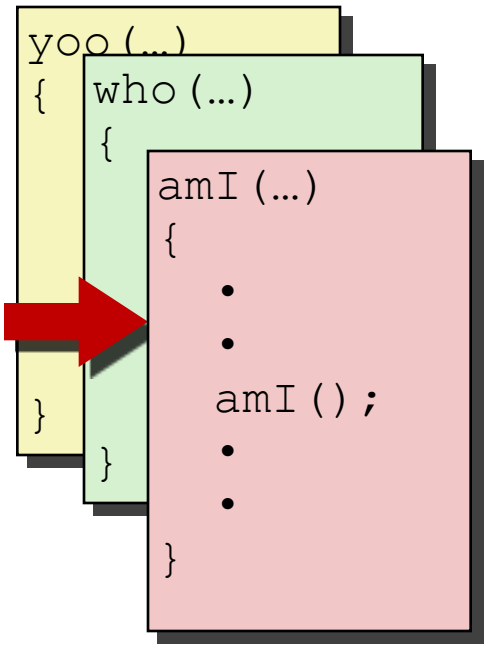
Example

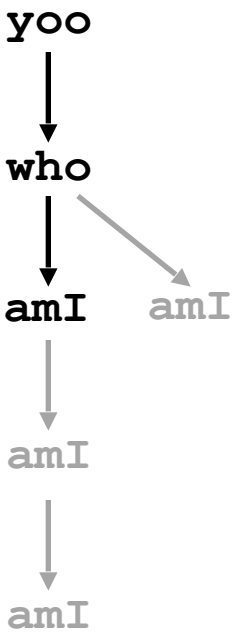


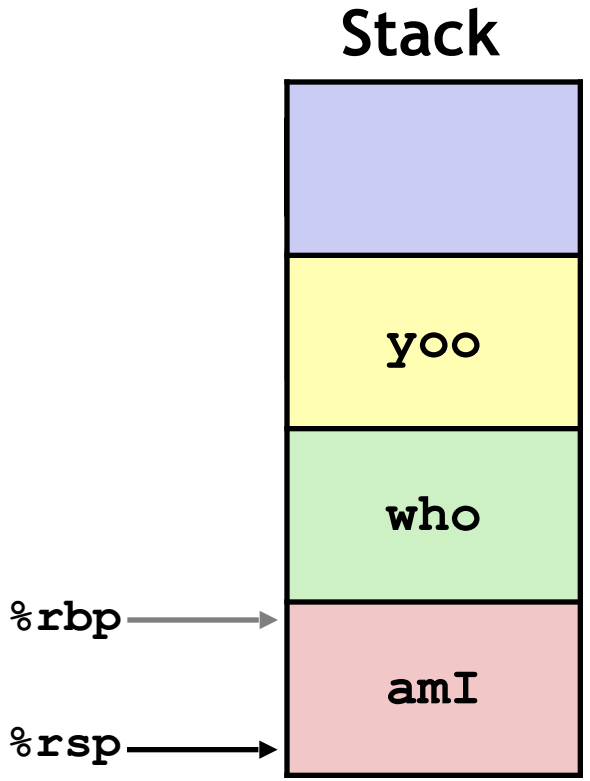




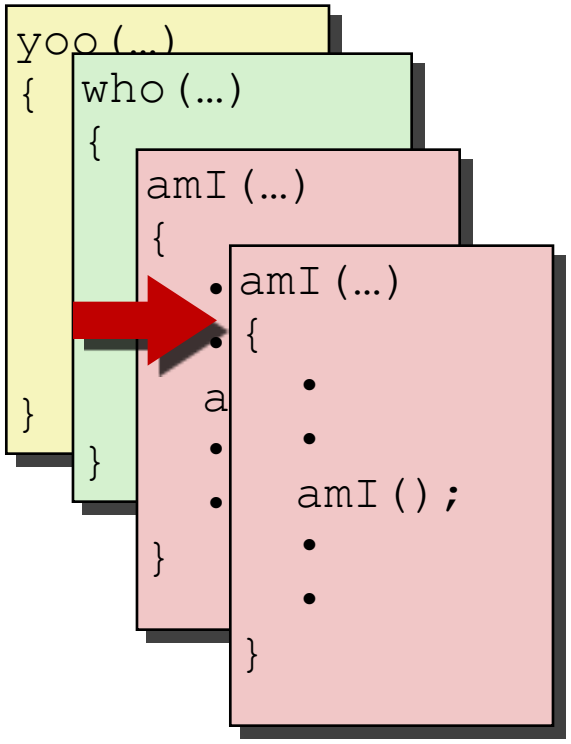
Example

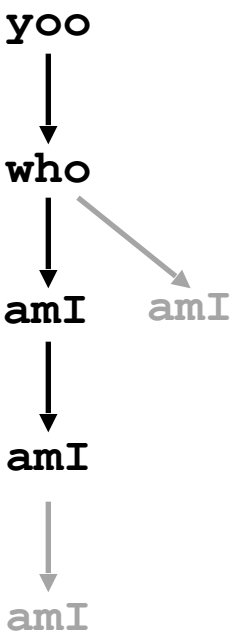

yoo(…)
{
who(…)
{
amI(…)
{
•
•
amI();
•
•
}
}
}

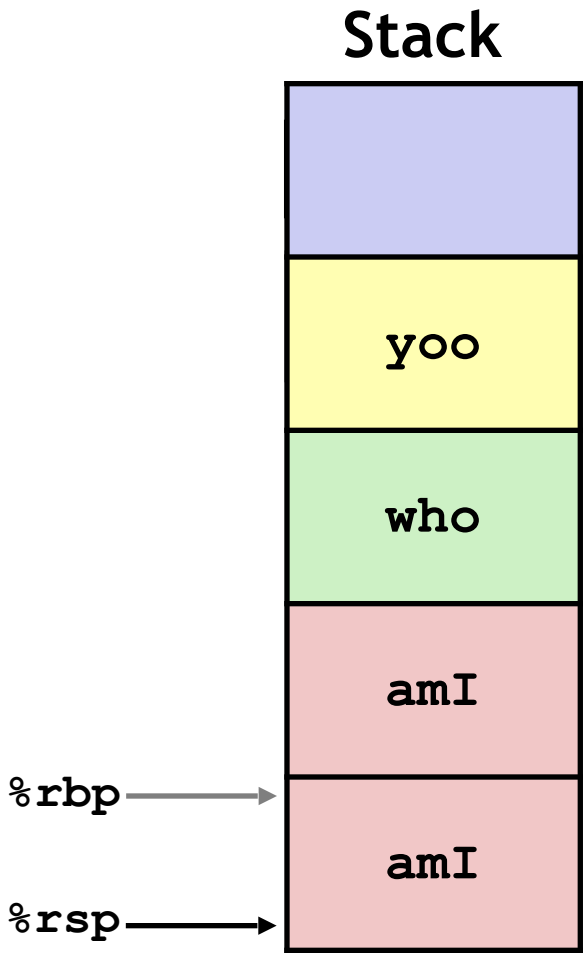

yoo
who
amI
amI
amI
amI


Stack
yoo
who
amI
%rbp
%rsp

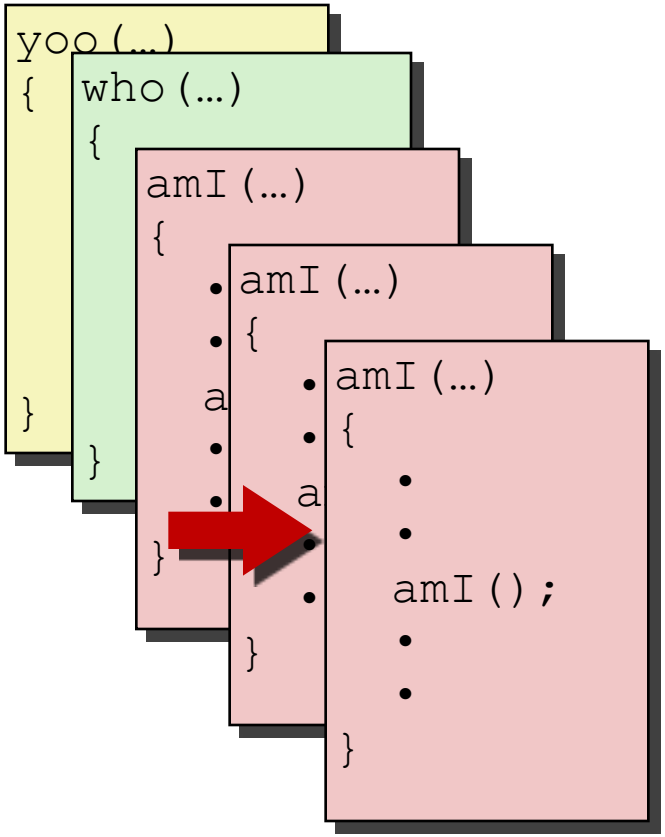
Example

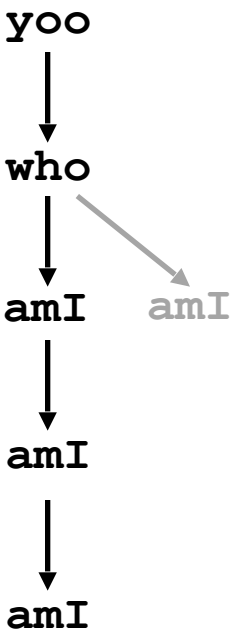

yoo(…)
{
who(…)
{
amI(…)
{
amI(…)
{
a
amI();
}
}
}

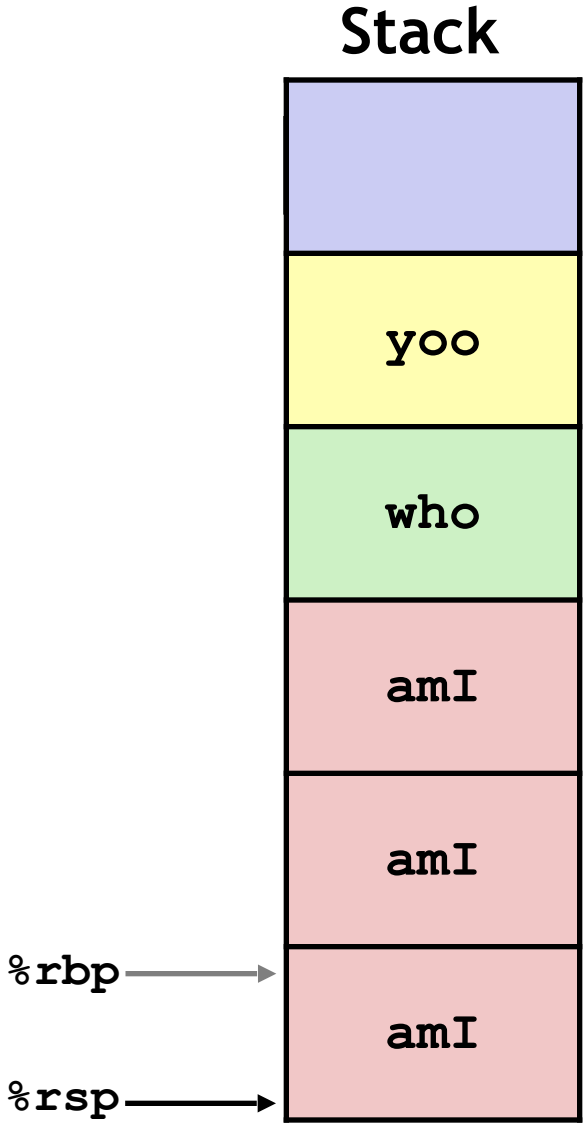

yoo
who
amI
amI
amI
amI


Stack
yoo
who
amI
amI
%rbp
%rsp

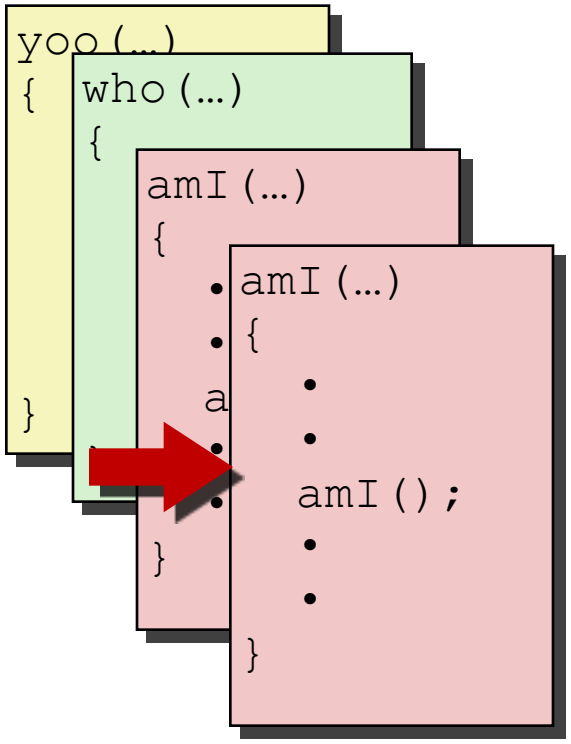
Example

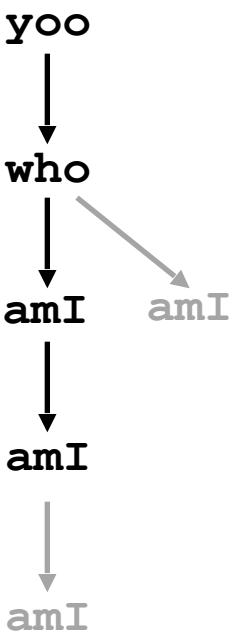

yoo (…)
{
who (…)
{
amI (…)
{
amI (…)
{
amI (…)
{
amI();
}

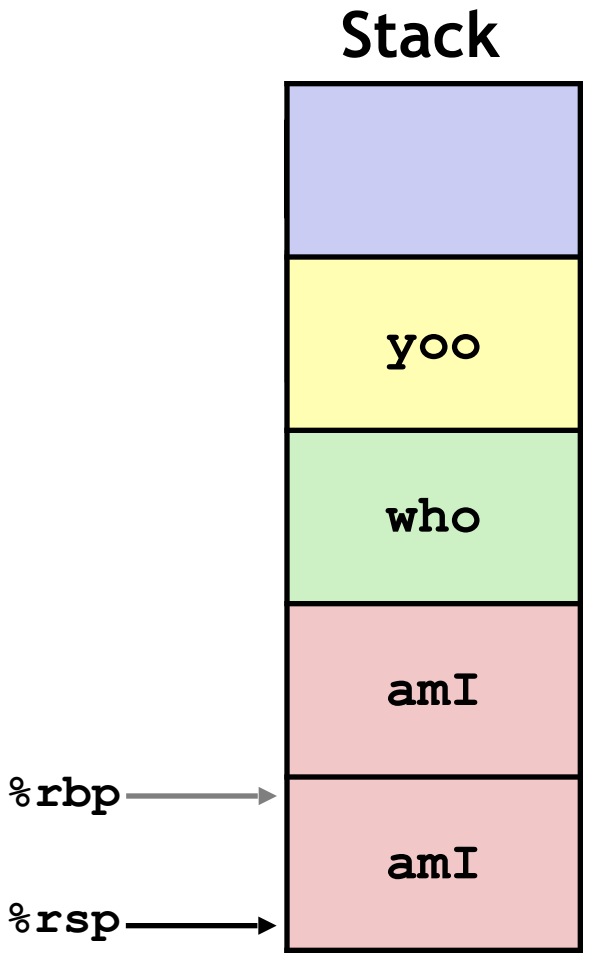

yoo
who
amI
amI
amI
amI


Stack
yoo
who
amI
amI
amI
%rbp
%rsp

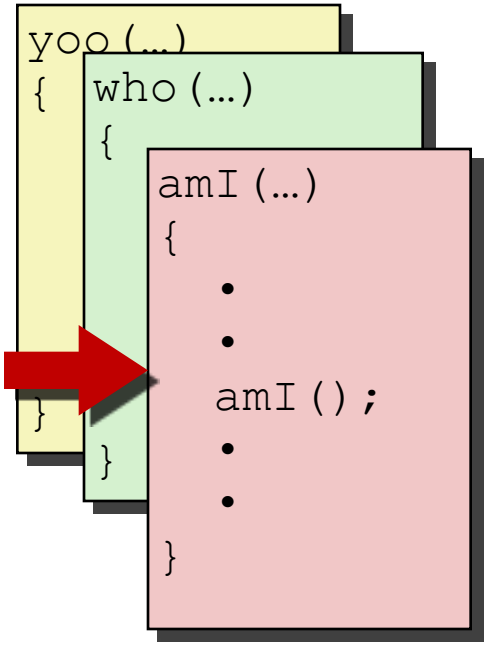
Example

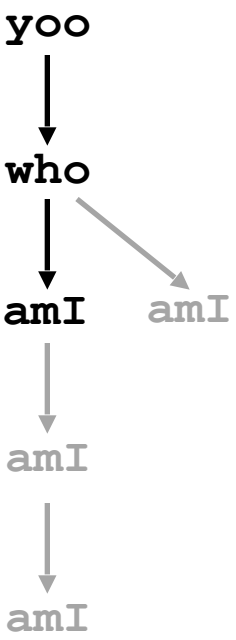

yoo(…)
{
who(…)
{
amI(…)
{
amI(…)
{
amI();
}
}

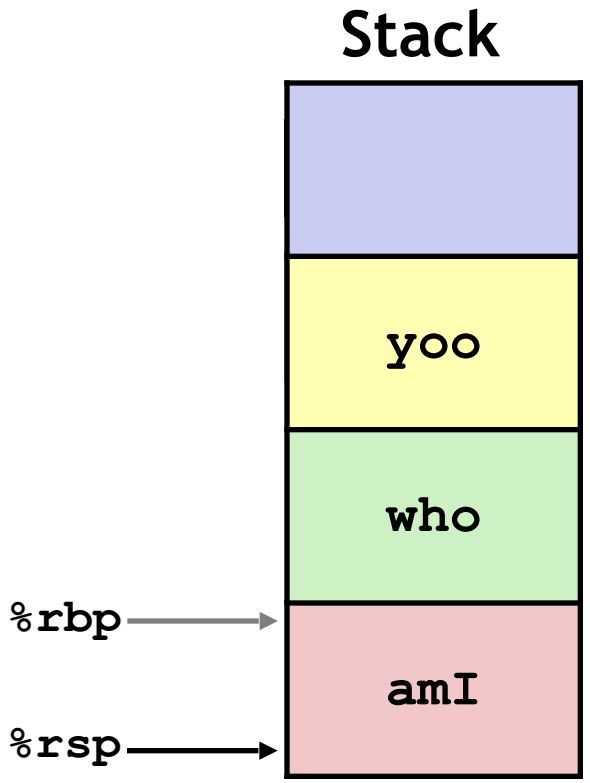

yoo
who
amI
amI
amI
amI


Stack
yoo
who
amI
amI
%rbp
%rsp

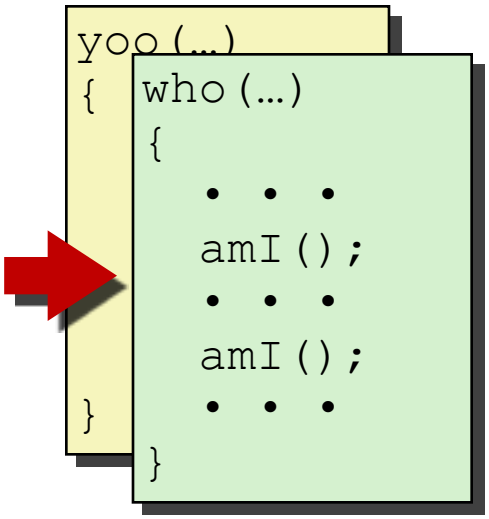
Example

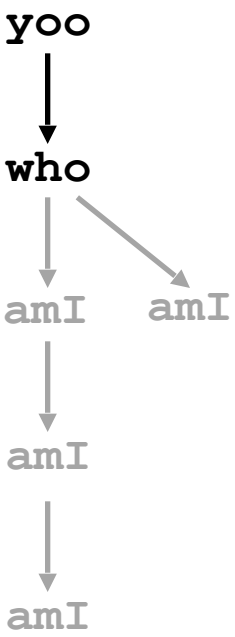

yoo(…)
{
who(…)
{
amI(…)
{
•
•
amI();
•
•
}
}
}

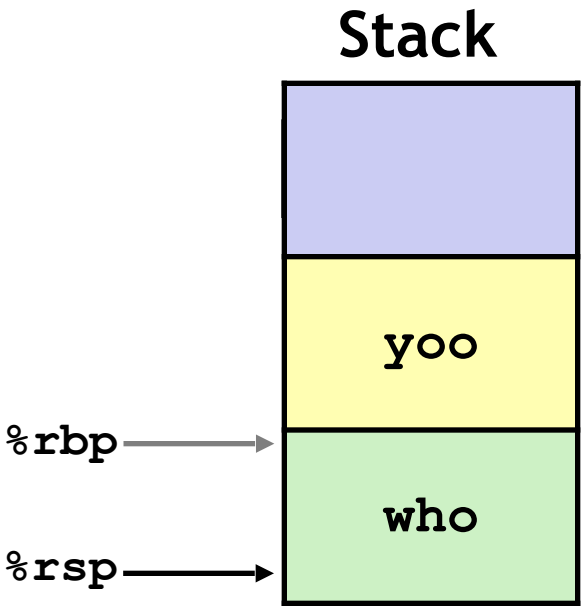

yoo
who
amI
amI
amI
amI


Stack
yoo
who
amI
%rbp
%rsp

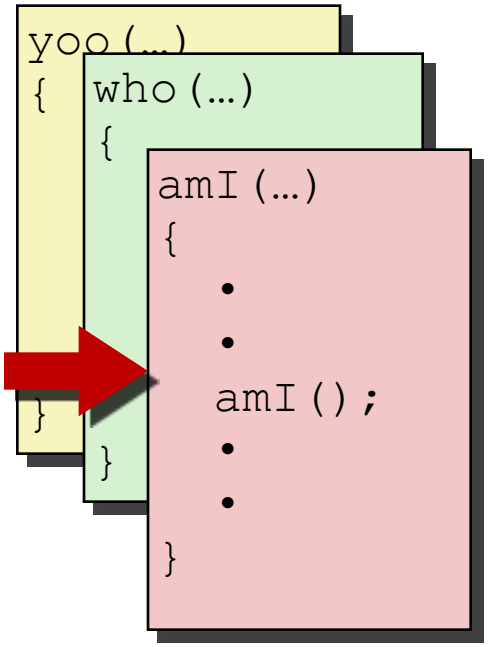
Example

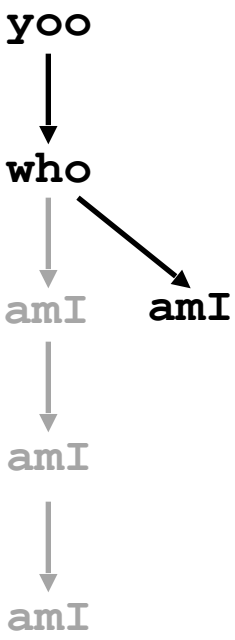


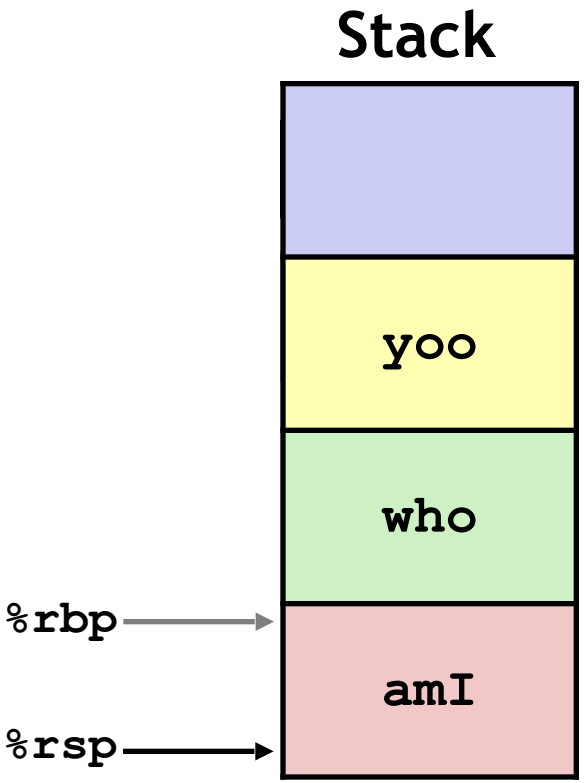




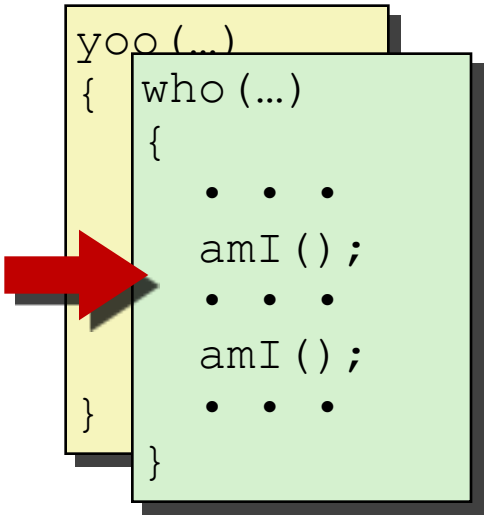
Example

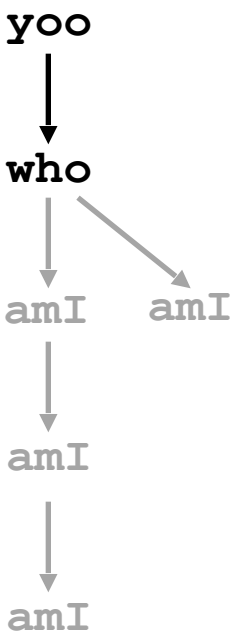

yoo(…)
{
who(…)
{
amI(…)
{
•
•
amI();
•
•
}
}
}

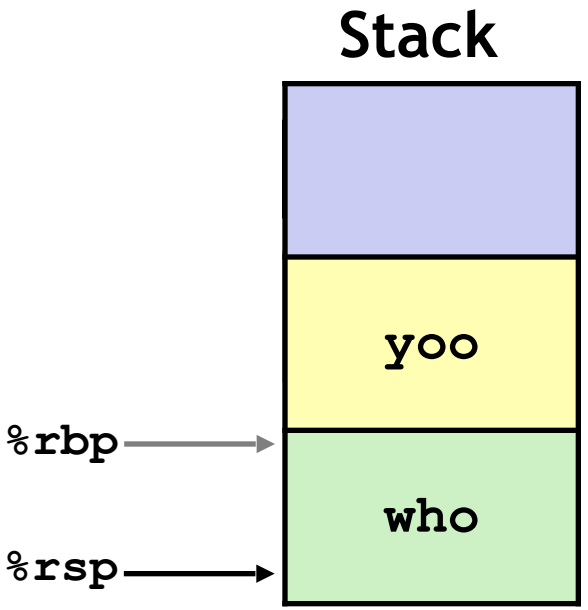

yoo
who
amI
amI
amI
amI


Stack
yoo
who
amI
%rbp
%rsp

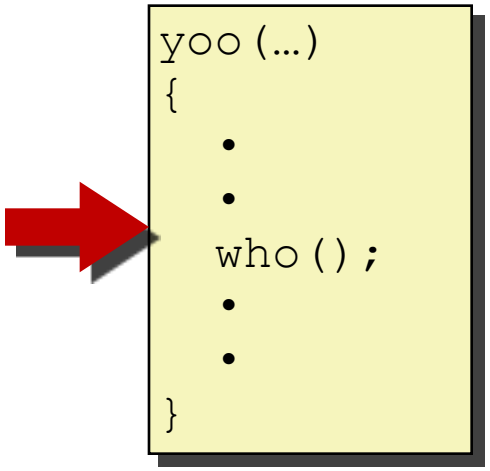
Example

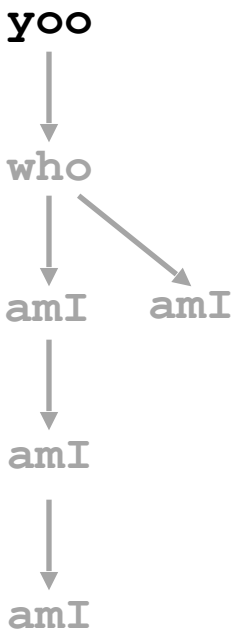


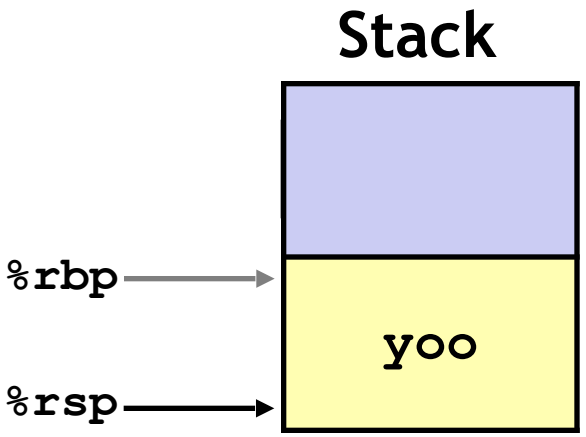




Example







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