The Stack & Procedures

CSE 351 Autumn 2017

Instructor:

Justin Hsia

Teaching Assistants:

Lucas Wotton

Michael Zhang

Parker DeWilde

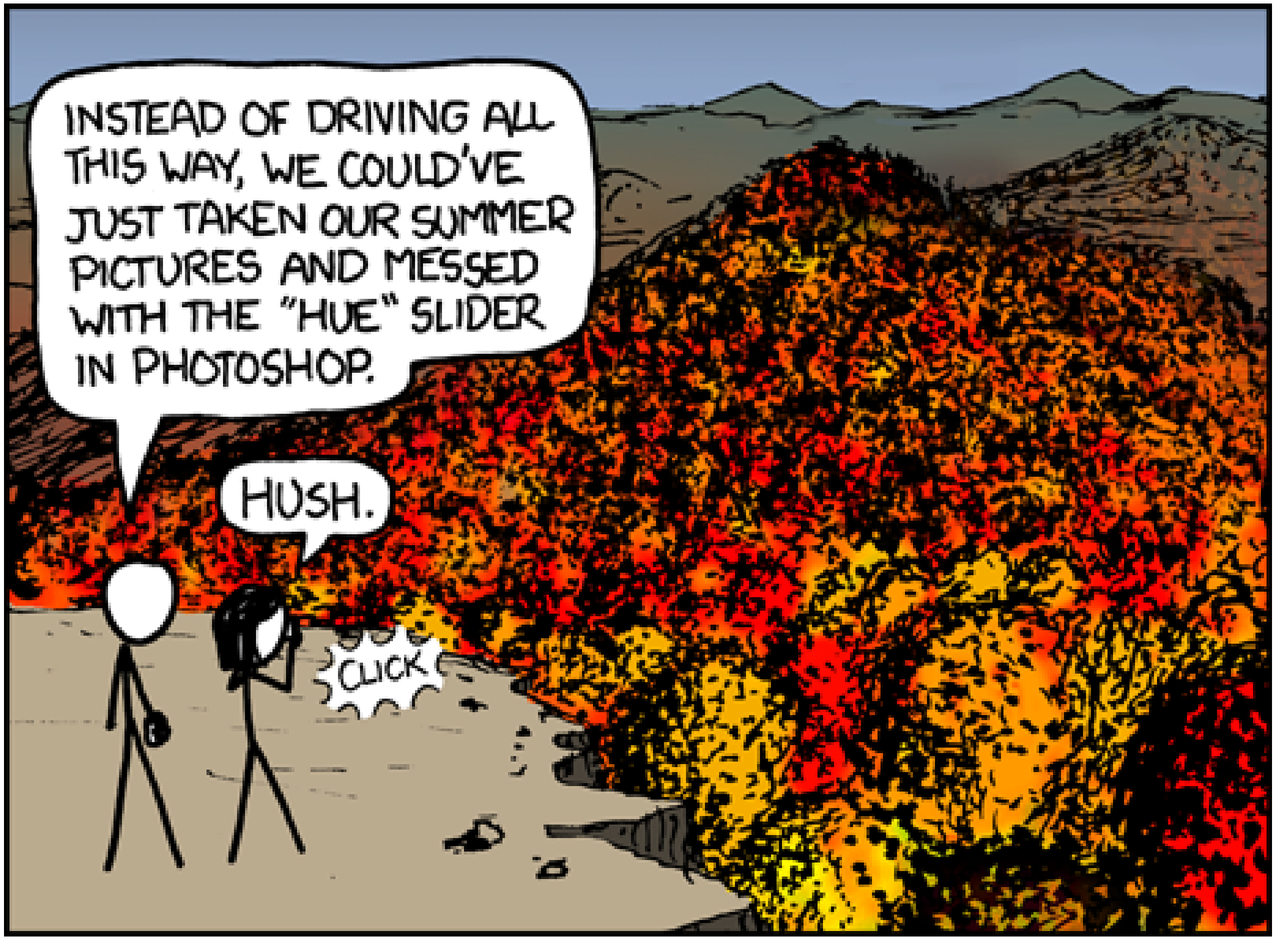
Ryan Wong

Sam Gehman

Sam Wolfson

Savanna Yee

Vinny Palaniappan



http://xkcd.com/648/

Administrivia

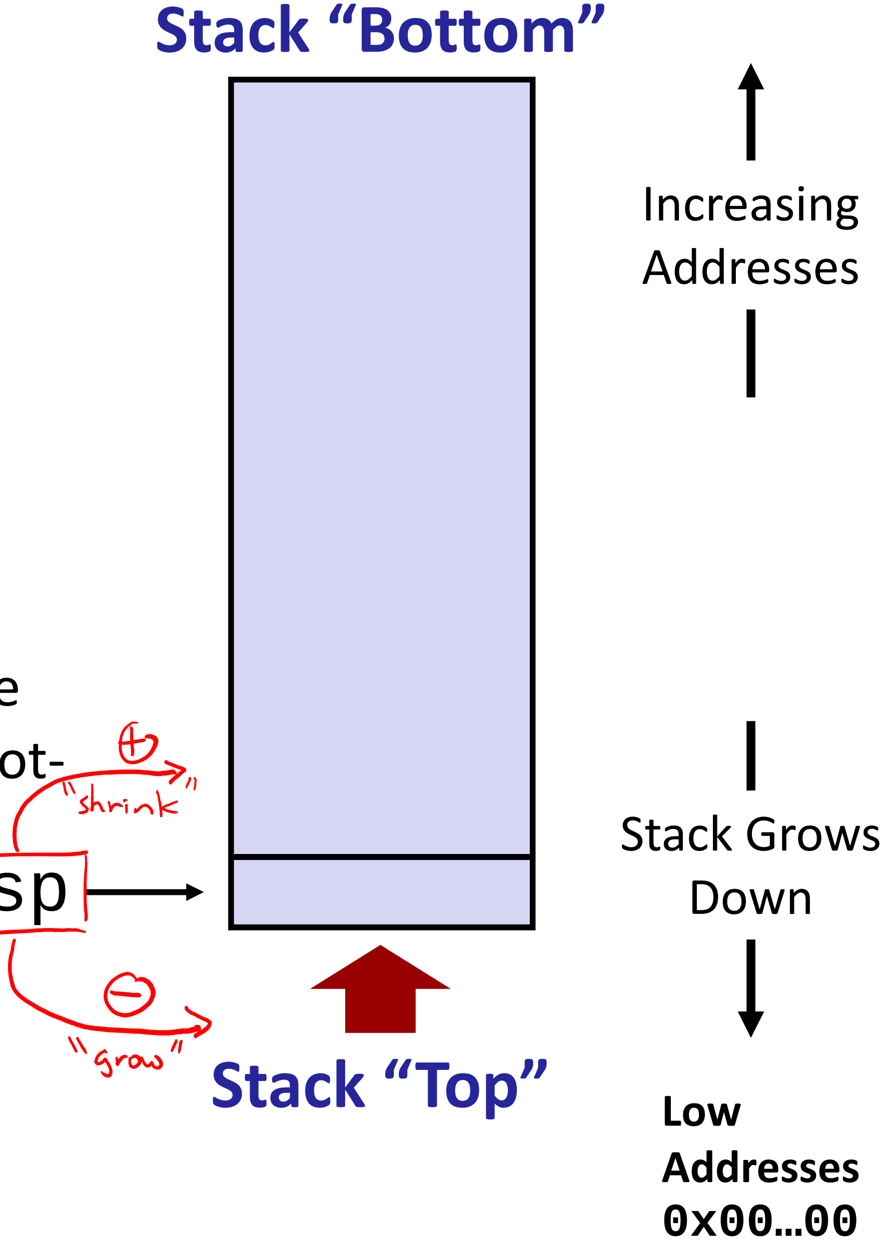
- Homework 2 due tonight
- Lab 2 due next Friday (10/27)
  - Ideally want to finish well before the midterm
- Homework 3 released next week
  - On midterm material, but due after the midterm

- **Midterm** (10/30, 5-6:30pm, KNE 120)
  - Reference sheet + 1 *handwritten* cheat sheet
  - Find a study group! Look at past exams!
  - Average is typically around 70%
  - **Review session** (10/27) in EEB 105 from 5:30-7:30pm

x86-64 Stack

Last In, First Out (LIFO)

- Region of memory managed with stack "discipline"
  - Grows toward lower addresses
  - Customarily shown "upside-down"
- Register `%rsp` contains *lowest* stack address
  - `%rsp` = address of *top* element, the most-recently-pushed item that is not-yet-popped



x86-64 Stack: Push

- `pushq` *src*
  ↑ size specifier
  - Fetch operand at *src*
    - *Src* can be reg, memory, immediate
  - ***Decrement*** `%rsp` by 8
  - Store value at address given by `%rsp`
- Example:
  - **`pushq %rcx`**
  - Adjust `%rsp` and store contents of `%rcx` on the stack

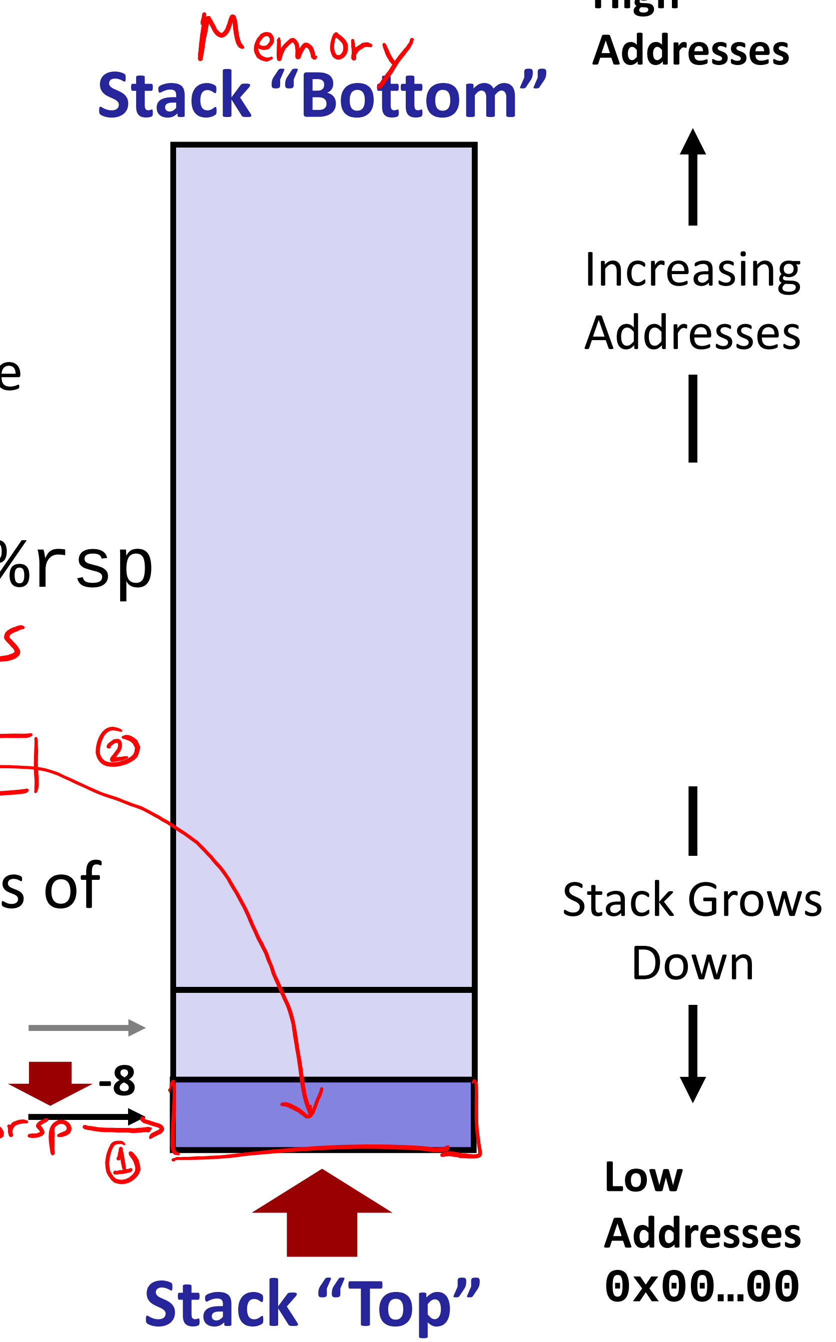
Registers

%rcx ②

Stack Pointer: `%rsp`

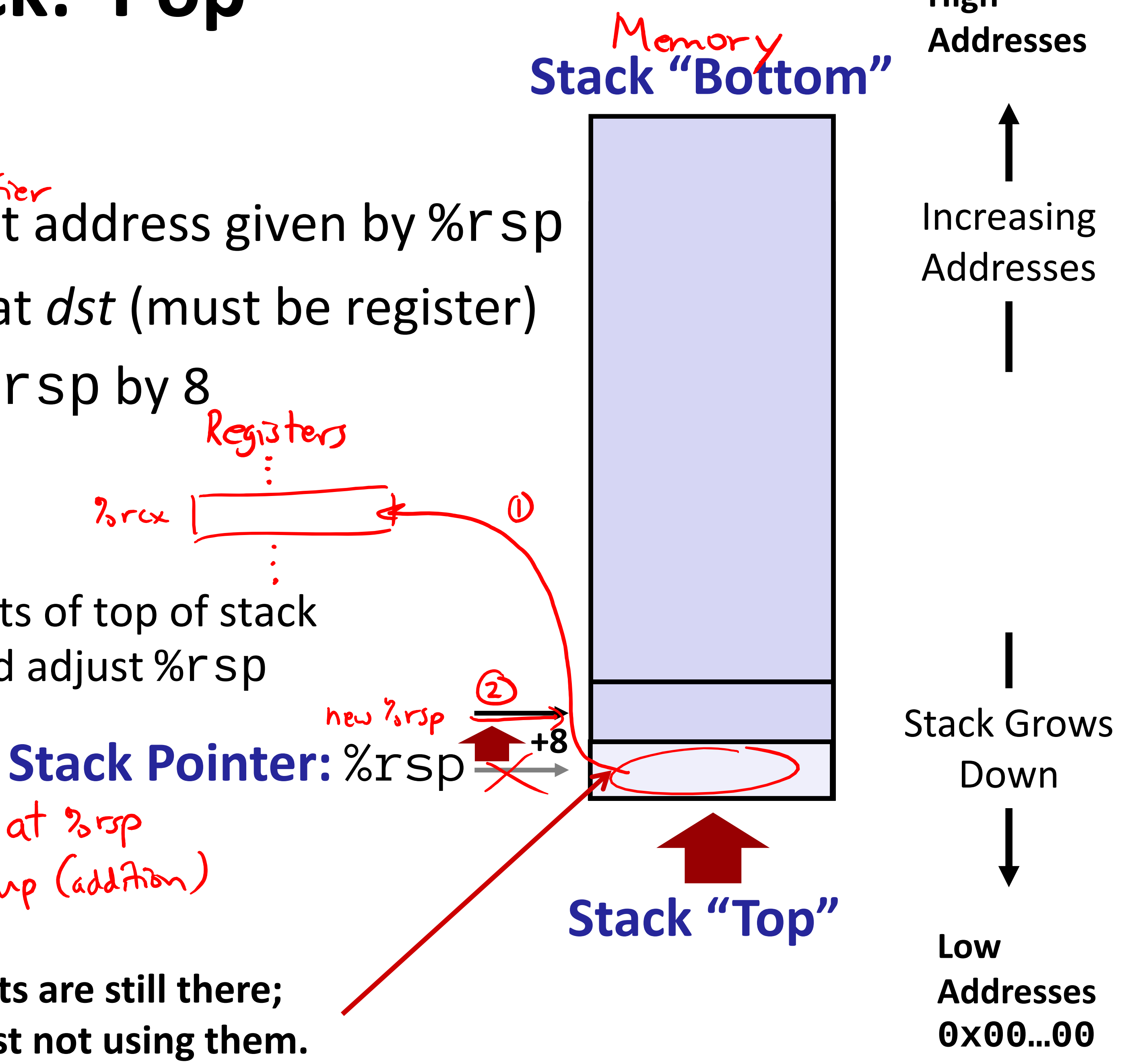
① move %rsp down (subtract)

② store src at %rsp



x86-64 Stack: Pop

- popq *dst*
  ↑ size specifier
  - Load value at address given by `%rsp`
  - Store value at *dst* (must be register)
  - ***Increment*** `%rsp` by 8
- Example:
  - **`popq %rcx`**
  - Stores contents of top of stack into `%rcx` and adjust `%rsp`



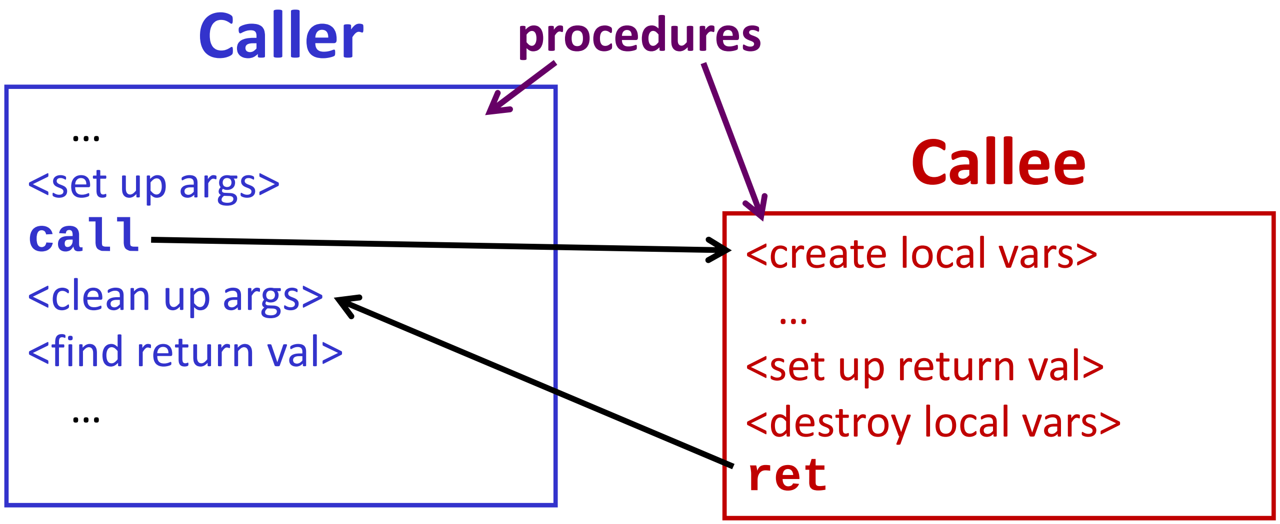
① read out data at %rsp
② move %rsp up (addition)

Those bits are still there; we're just not using them.

Procedures

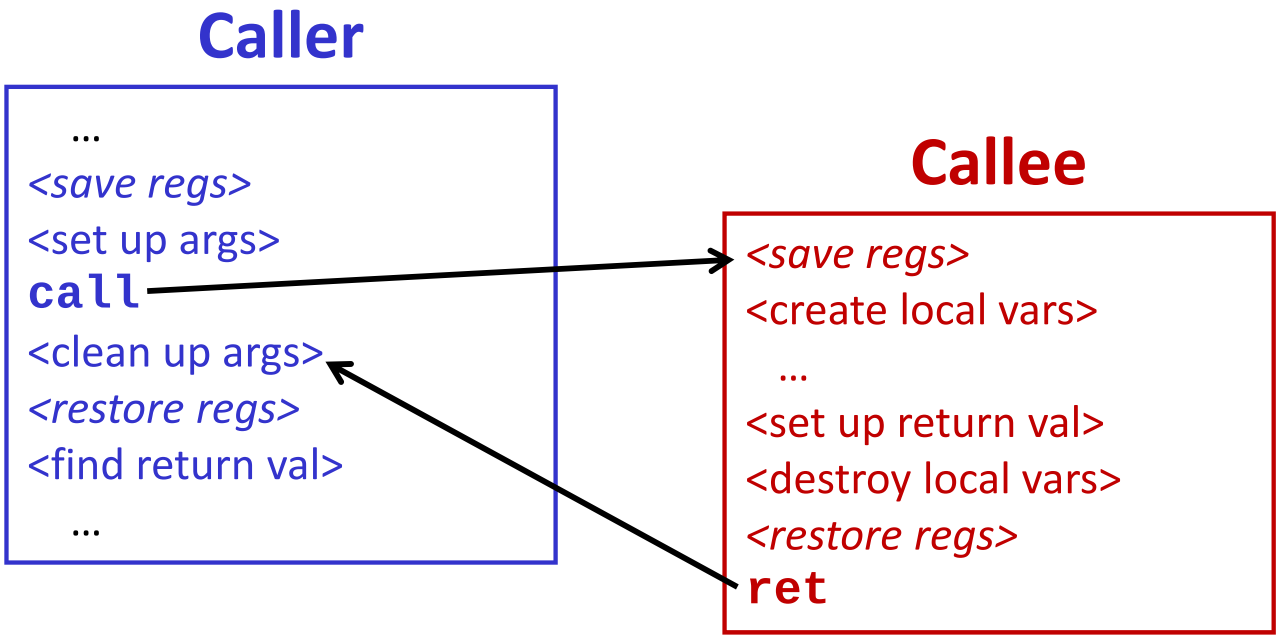
- Stack Structure
- **Calling Conventions**
  - **Passing control**
  - Passing data
  - Managing local data
- Register Saving Conventions
- Illustration of Recursion

Procedure Call Overview



- **Callee** must know where to find args
- **Callee** must know where to find *return address*
- **Caller** must know where to find *return value*
- **Caller** and **Callee** run on same CPU, so use the same registers
  - How do we deal with register reuse?
- Unneeded steps can be skipped (*e.g.* no arguments)

Procedure Call Overview



Caller

…
<save regs>
<set up args>
`call`
<clean up args>
<restore regs>
<find return val>
…

Callee

<save regs>
<create local vars>
…
<set up return val>
<destroy local vars>
<restore regs>
`ret`

- The *convention* of where to leave/find things is called the calling convention (or procedure call linkage)
  - Details vary between systems
  - We will see the convention for x86-64/Linux in detail
  - What could happen if our program didn't follow these conventions?

Code Example (Preview)

Caller

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

Compiler Explorer:
https://godbolt.org/g/cKKDZn

executable disassembly

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

these are instruction addresses

Callee

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

```
0000000000400550 <mult2>:
  400550:   movq    %rdi,%rax   # a
  400553:   imulq   %rsi,%rax   # a * b
  400557:   ret                 # Return
```

Procedure Control Flow

- Use stack to support procedure call and return
- Procedure call: `call` *`label`* (special push)
  1) Push return address on stack (*why? which address?*)
  2) Jump to ***`label`***

① move %rsp down
② store ret addr at %rsp
③ label → %rip

Procedure Control Flow

- Use stack to support procedure call and return
- Procedure call: `call label` (special push)
  1) Push return address on stack (*why? which address?*)
     ① move %rsp down
     ② store ret addr at %rsp
     ③ label → %rip
  2) Jump to **`label`**
- Return address:
  - Address of instruction immediately after **`call`** instruction
  - Example from disassembly:

```
400544: call   400550 <mult2>
400549: movq   %rax,(%rbx)
```

Return address = **`0x400549`**

next instruction happens to be a move, but could be anything

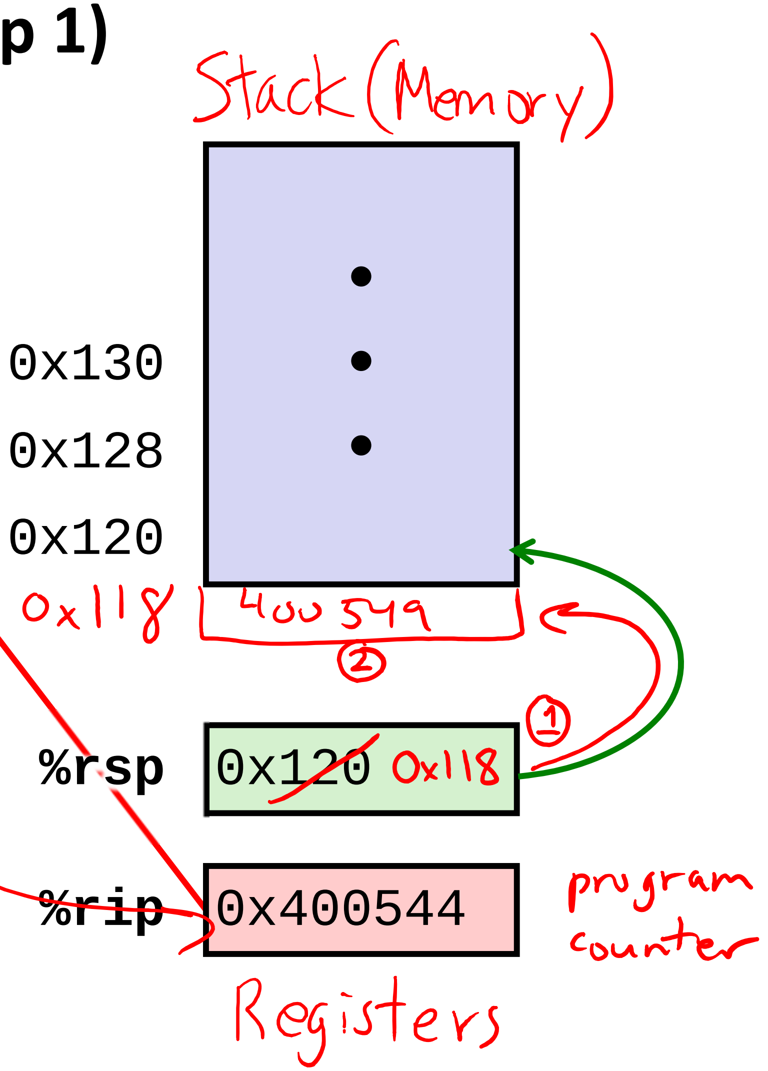
- Procedure return: `ret` (special pop)
  1) Pop return address from stack
     ① read ret addr at %rsp (into %rip)
     ② move %rsp up
  2) Jump to address

Procedure Call Example (step 1)

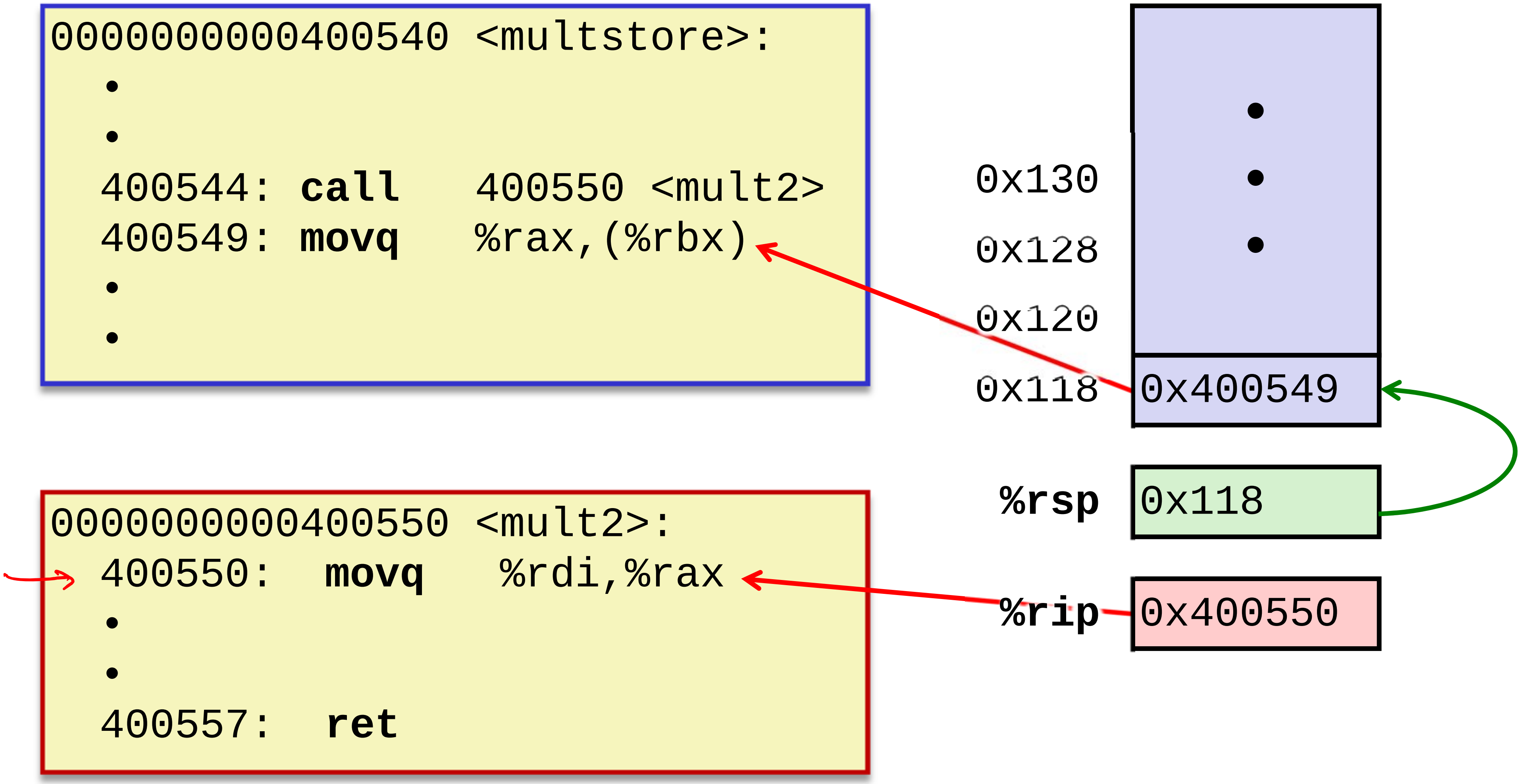
```
0000000000400540 <multstore>:
  •
  •
  400544: call   400550 <mult2>
  400549: movq   %rax,(%rbx)
  •
  •
```

③

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



Procedure Call Example (step 2)



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

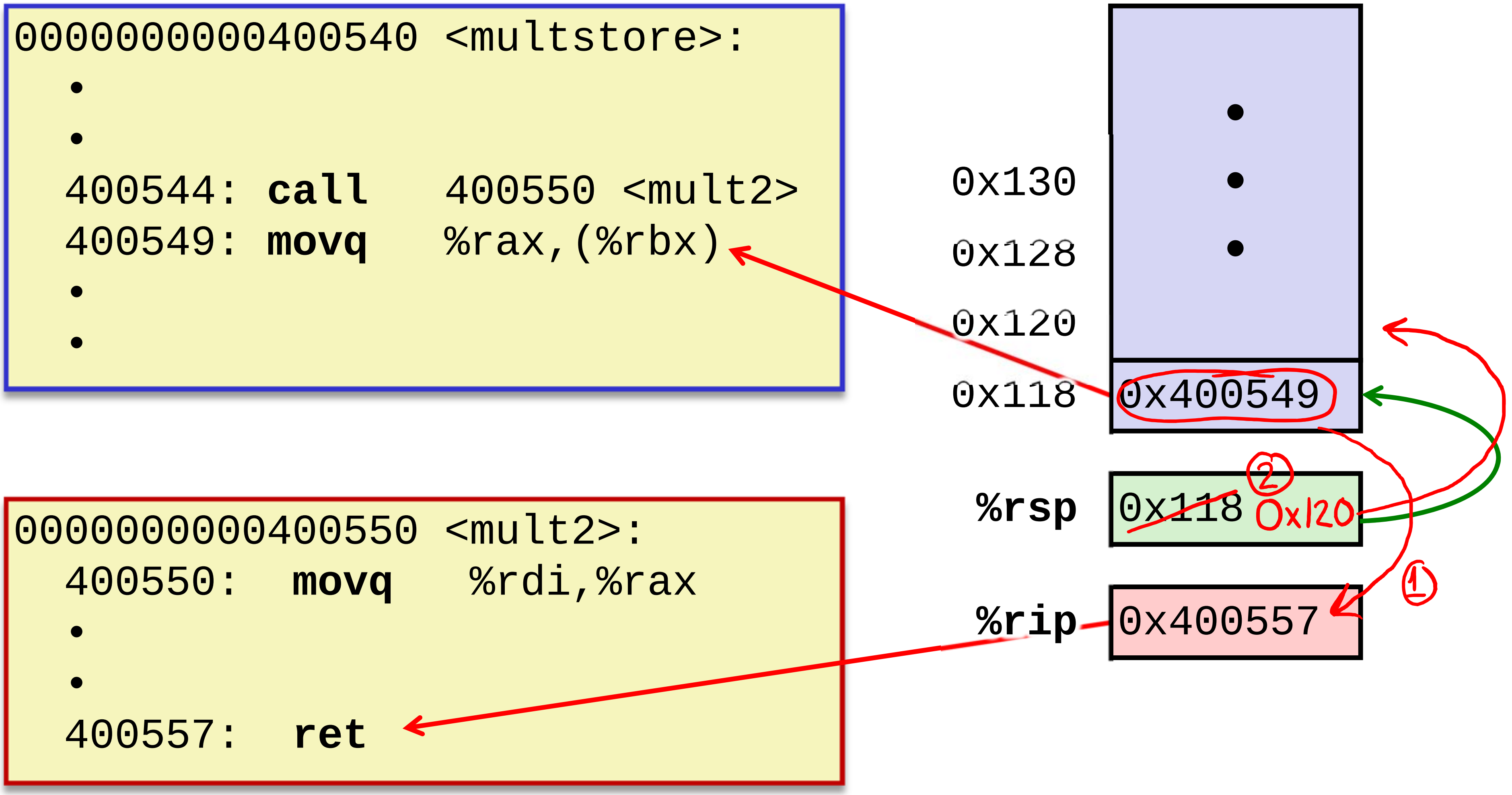
```
0000000000400550 <mult2>:
  400550:  movq   %rdi,%rax
  •
  •
  400557:  ret
```

	•
0x130	•
0x128	•
0x120	
0x118	0x400549

%rsp 0x118

%rip 0x400550

Procedure Return Example (step 1)



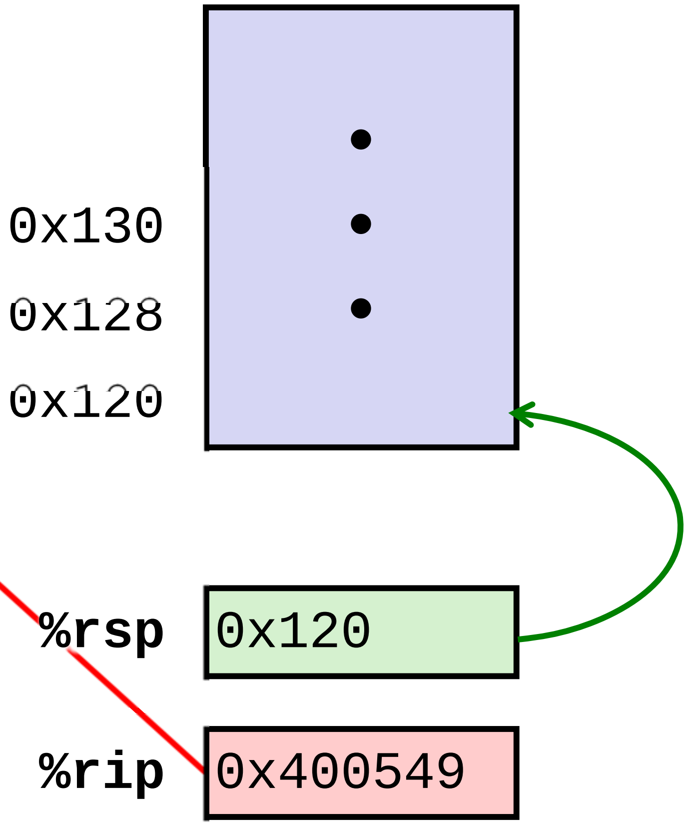
```
0000000000400540 <multstore>:
  •
  •
  400544: call   400550 <mult2>
  400549: movq   %rax,(%rbx)
  •
  •
```

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

Procedure **Return** Example (step 2)

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

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



Procedures

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

Procedure Data Flow

Registers (NOT in Memory)

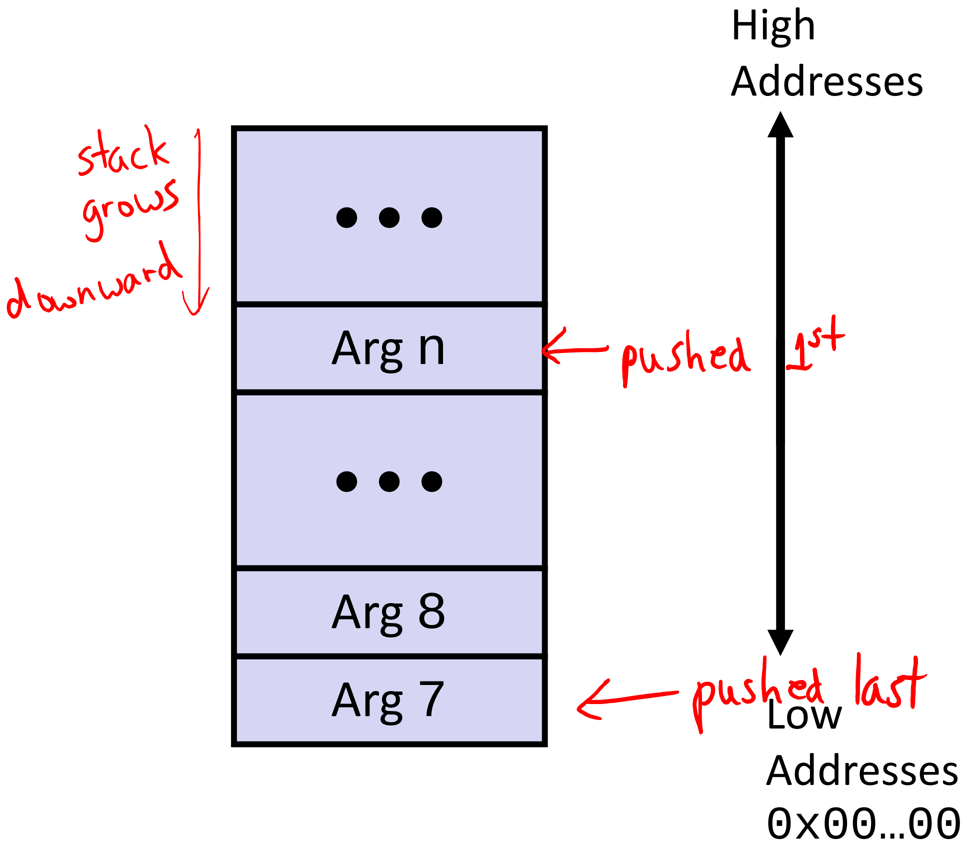
- First 6 arguments

①	`%rdi`	*Diane's*
②	`%rsi`	*Silk*
③	`%rdx`	*Dress*
④	`%rcx`	*Costs*
⑤	`%r8`	*$8*
⑥	`%r9`	*9*

- Return value

`%rax`

Stack (Memory)



- Only allocate stack space when needed

x86-64 Return Values

- By convention, values returned by procedures are placed in `%rax`
  - Choice of `%rax` is arbitrary

1) Caller must make sure to save the contents of `%rax` before calling a callee that returns a value
   - Part of register-saving convention
2) Callee places return value into `%rax`
   - Any type that can fit in 8 bytes – integer, float, pointer, etc.
   - For return values greater than 8 bytes, best to return a *pointer* to them
3) Upon return, caller finds the return value in `%rax`

Data Flow Examples

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

rdi rsi rdx

rdi rsi

lined up nicely so we didn't have to manipulate arguments

```
0000000000400540 <multstore>:
  # x in %rdi, y in %rsi, dest in %rdx
    • • •
  400541: movq   %rdx,%rbx       # “Save” dest
  400544: call   400550 <mult2>  # mult2(x,y)
  # t in %rax
  400549: movq   %rax,(%rbx)     # Save at dest
    • • •
```

(will explain later)

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

```
0000000000400550 <mult2>:
  # a in %rdi, b in %rsi
  400550:  movq   %rdi,%rax  # a
  400553:  imulq  %rsi,%rax  # a * b
  # s in %rax
  400557:  ret               # Return
```

Procedures

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

Stack-Based Languages

- Languages that support recursion
  - *e.g.* C, Java, most modern languages
  - Code must be *re-entrant*
    - Multiple simultaneous instantiations of single procedure
  - Need some place to store *state* of each instantiation
    - Arguments, local variables, return pointer
- Stack allocated in *frames*
  - State for a single procedure instantiation
- Stack discipline
  - State for a given procedure needed for a limited time
    - Starting from when it is called to when it returns
  - Callee always returns before caller does

Call Chain Example

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

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

1st call recurses twice

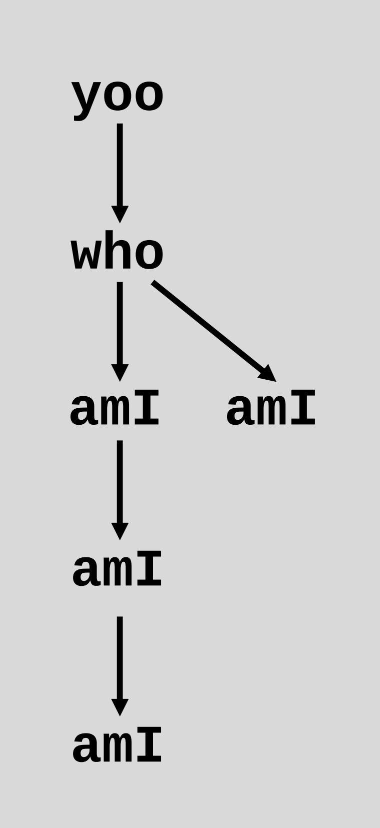
2nd call doesn't recurse

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

based on condition

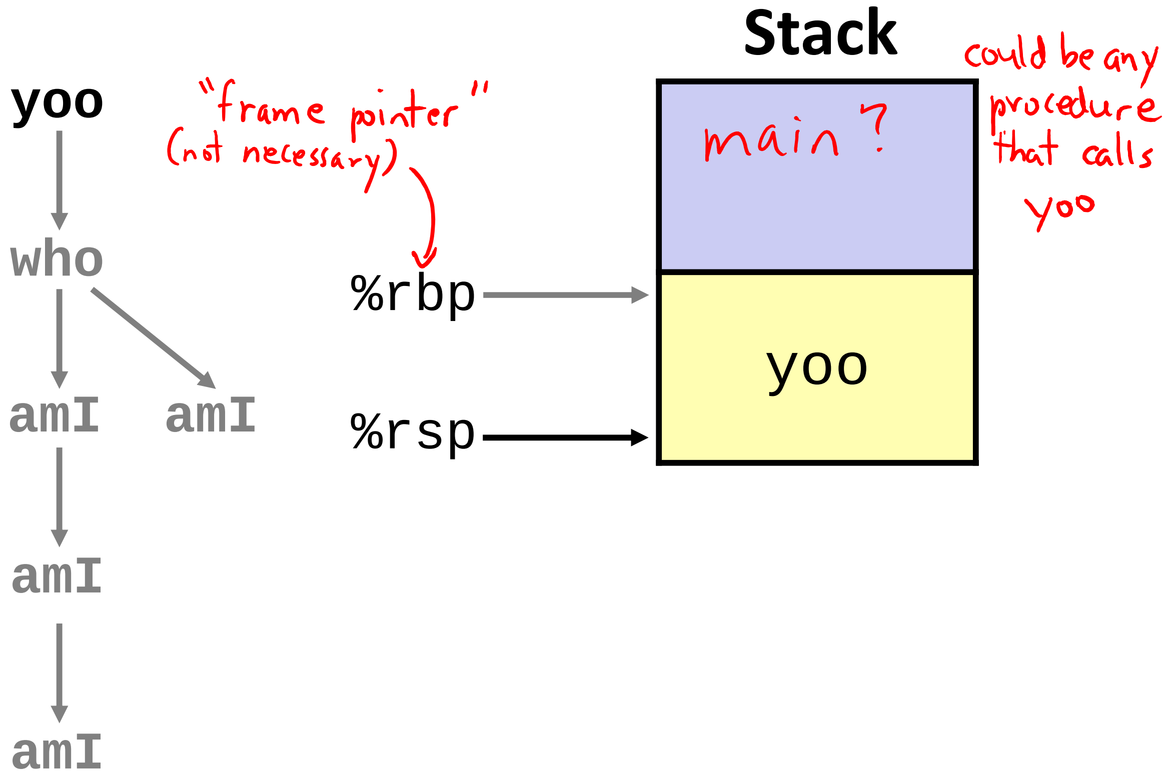
Procedure `amI` is recursive (calls itself)

Example Call Chain

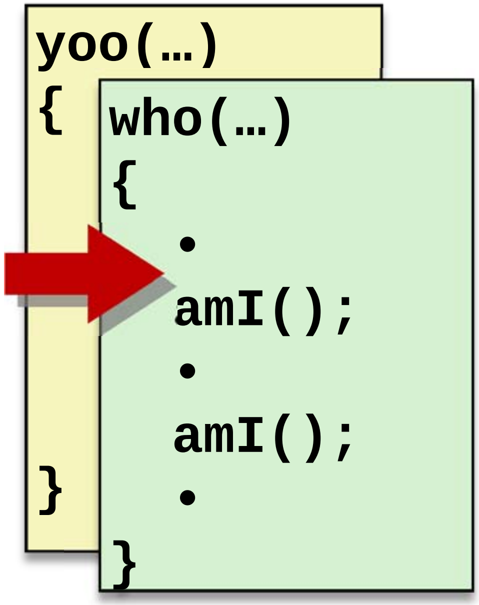


1) Call to yoo

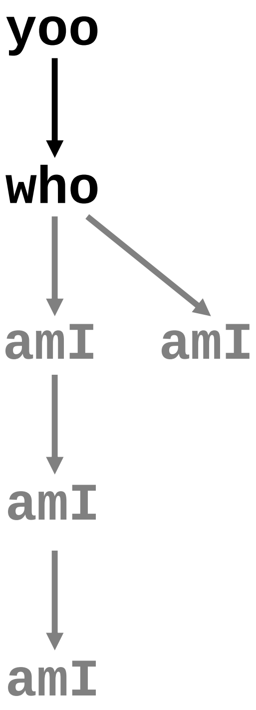
```
yoo(…)
{
  •
  •
  who();
  •
  •
}
```

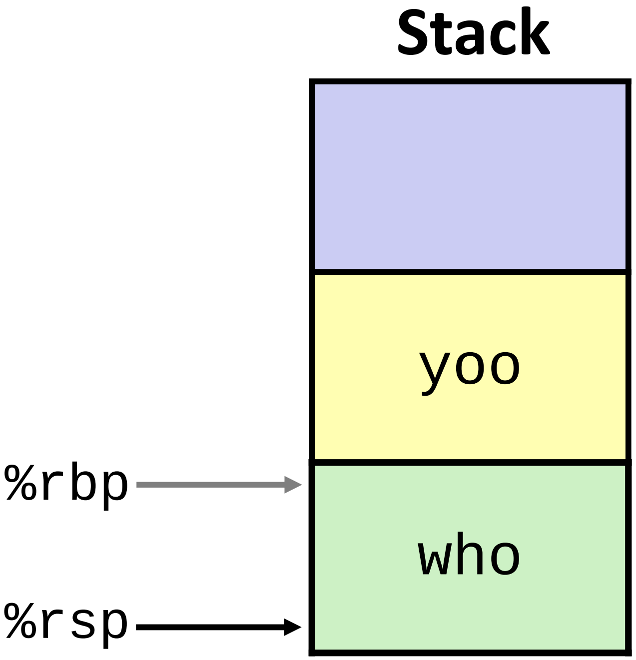


2) Call to who



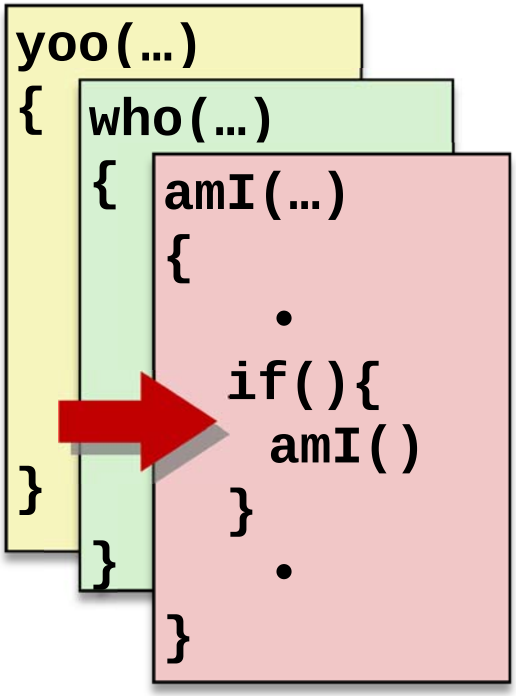
```
yoo(…)
{
    who(…)
    {
      •
      amI();
      •
      amI();
      •
    }
}
```

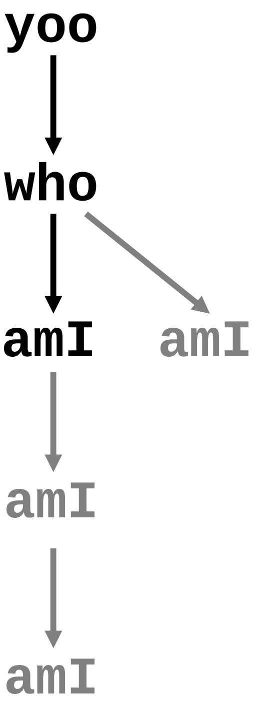


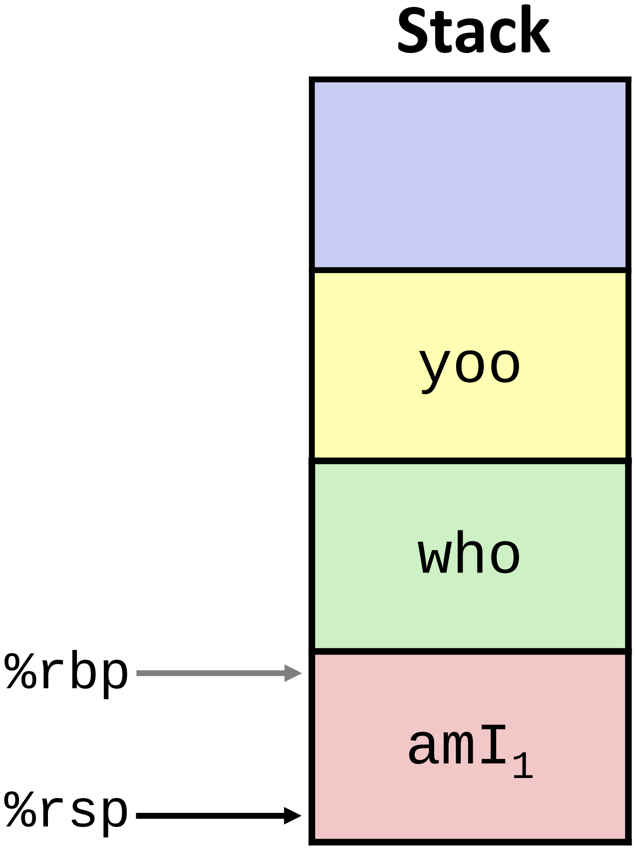


"create" frame
by manipulating %rsp

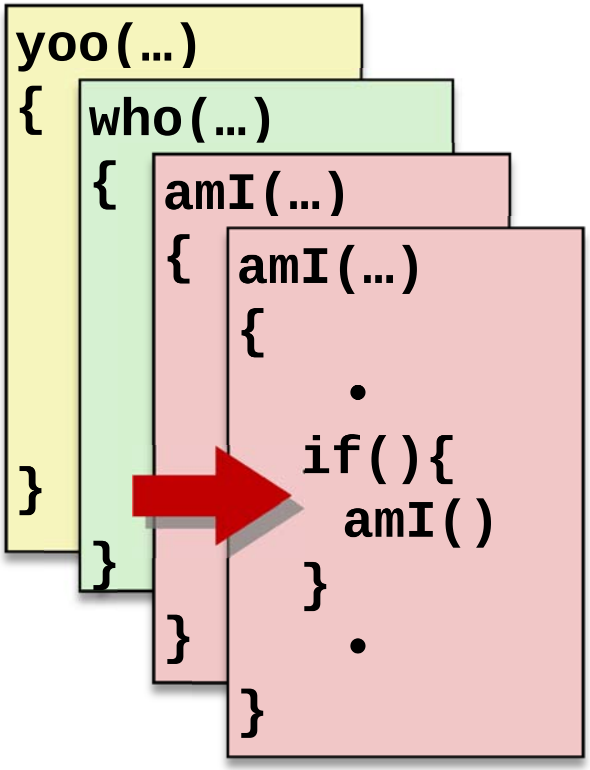
3) Call to `amI` (1)



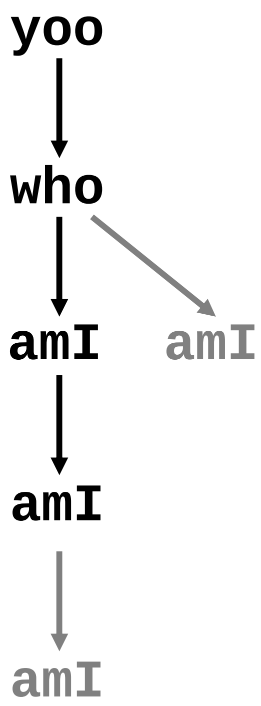


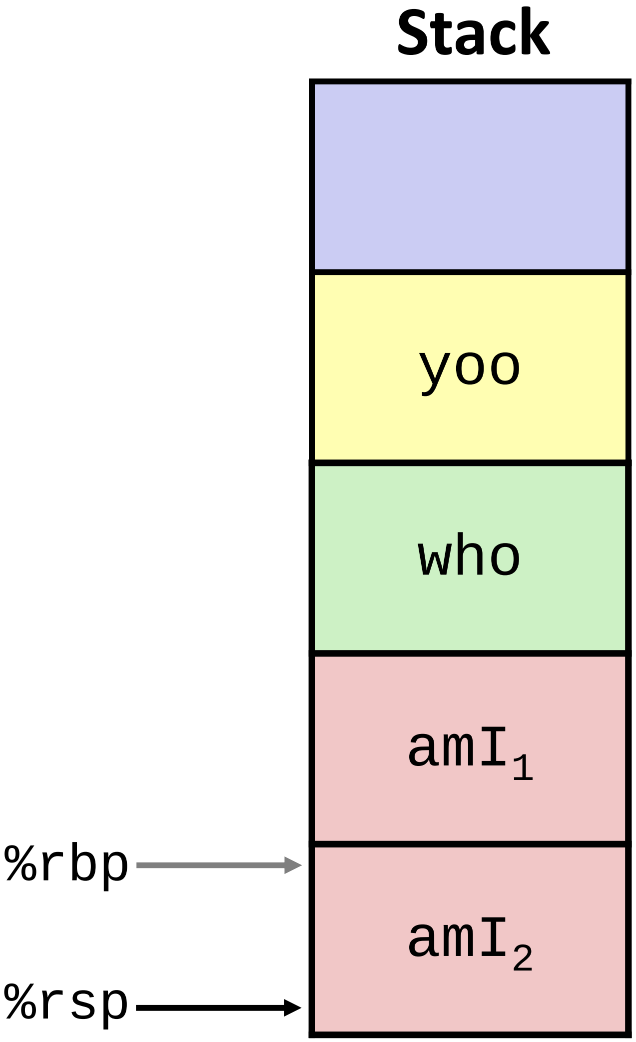


4) Recursive call to amI (2)

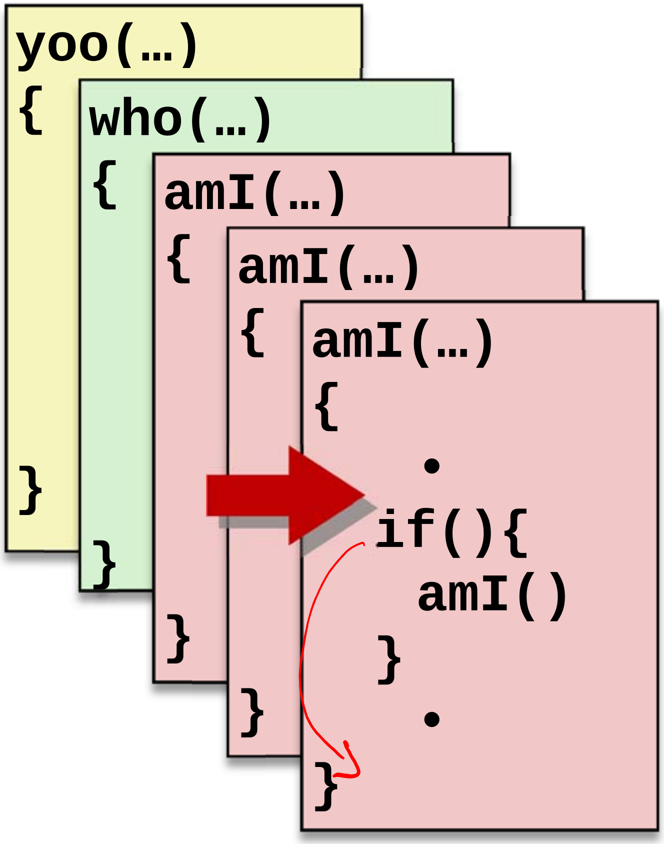


```
yoo(…)
{
  who(…)
  {
    amI(…)
    {
      amI(…)
      {
          •
        if(){
          amI()
        }
          •
      }
    }
  }
}
```

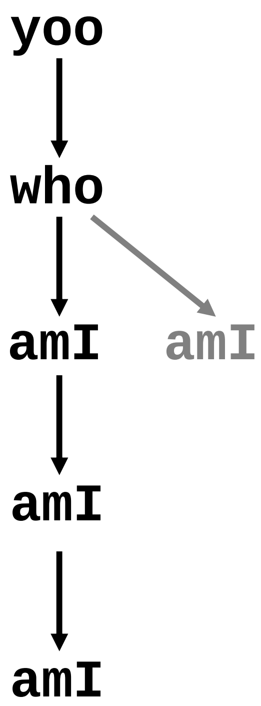




5) (another) Recursive call to `amI` (3)



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



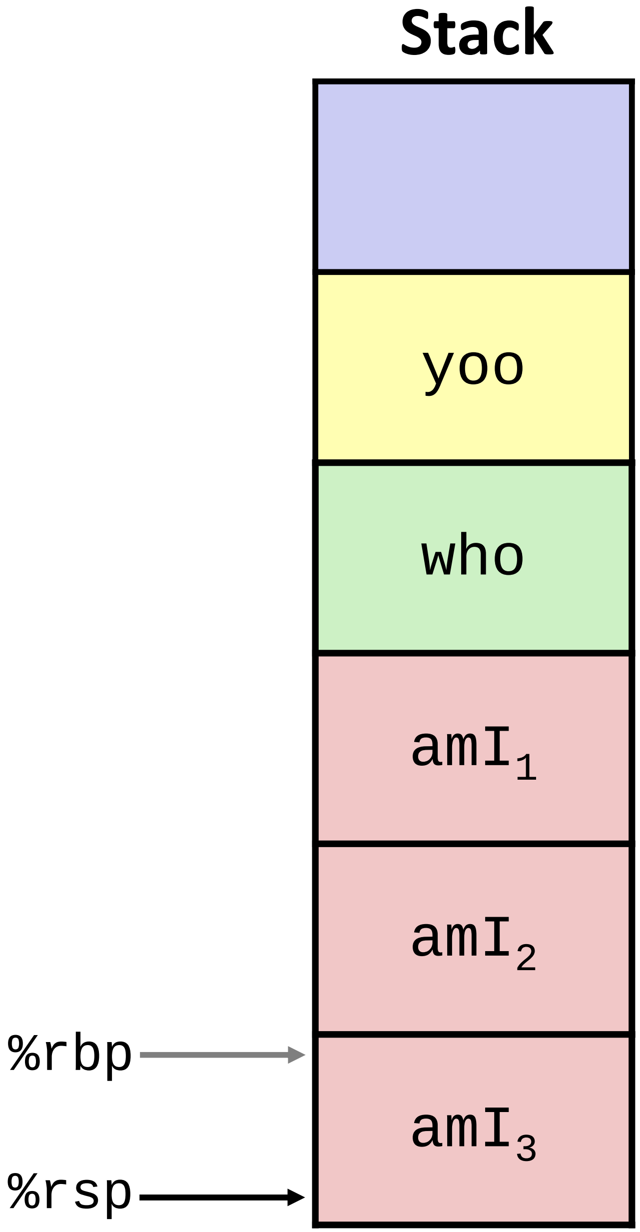
yoo

who

amI amI

amI

amI



Stack

yoo

who

$\texttt{amI}_1$

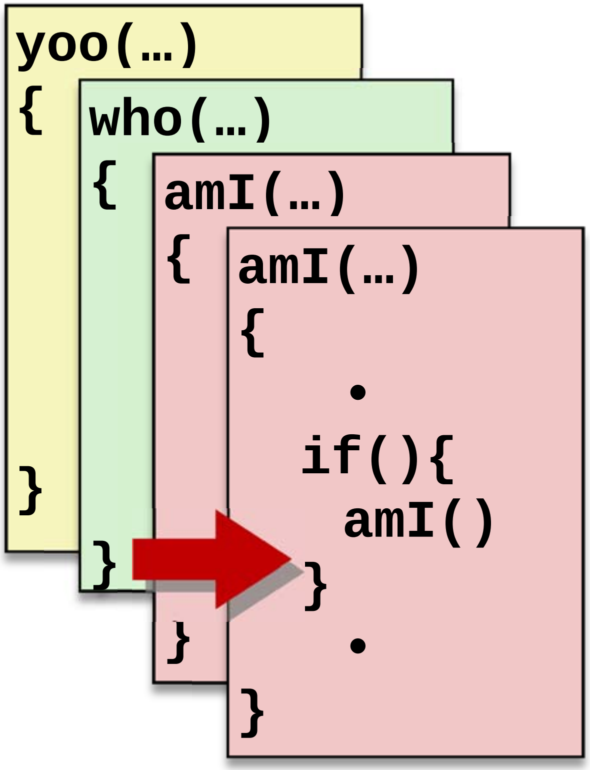
$\texttt{amI}_2$

$\texttt{amI}_3$

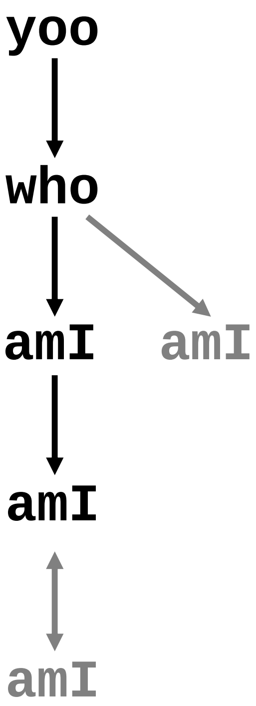
%rbp

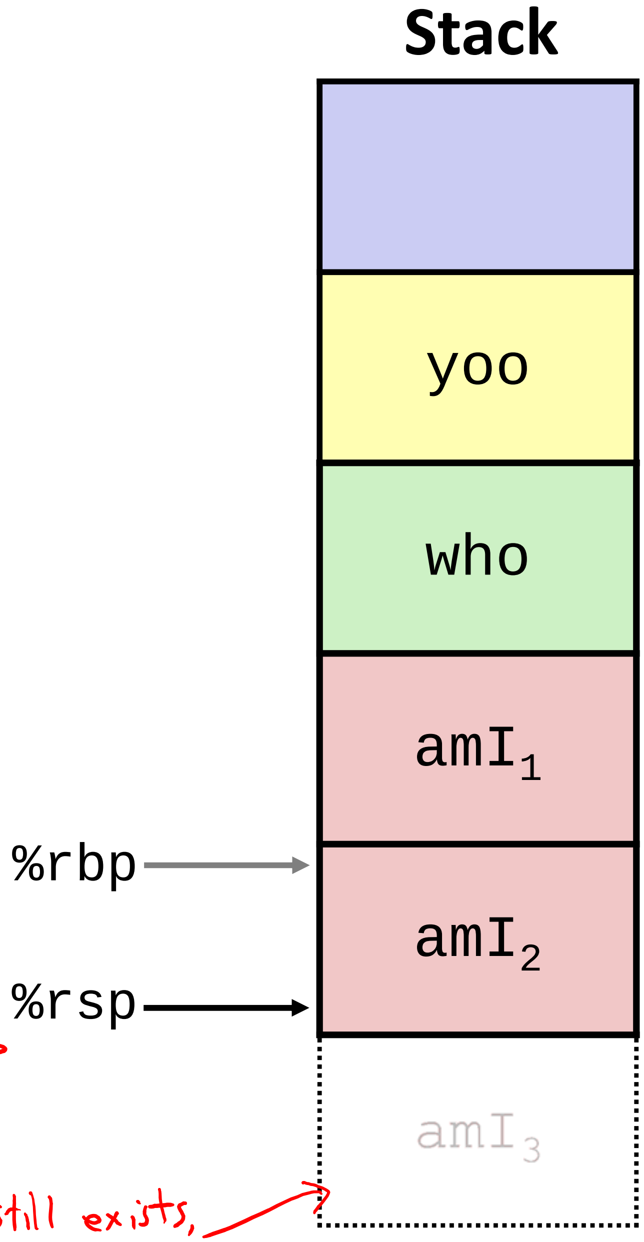
%rsp

6) Return from (another) recursive call to amI



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

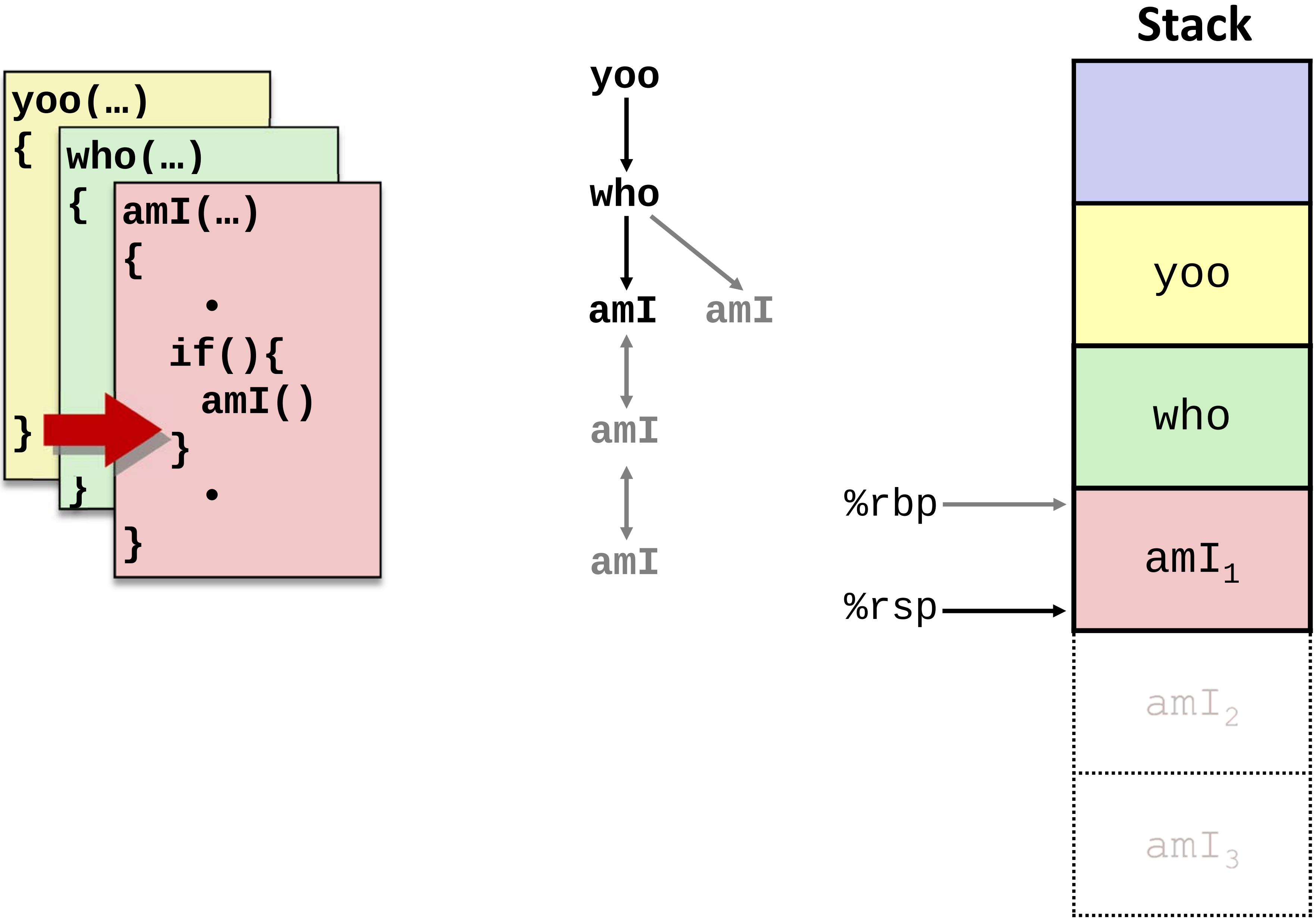




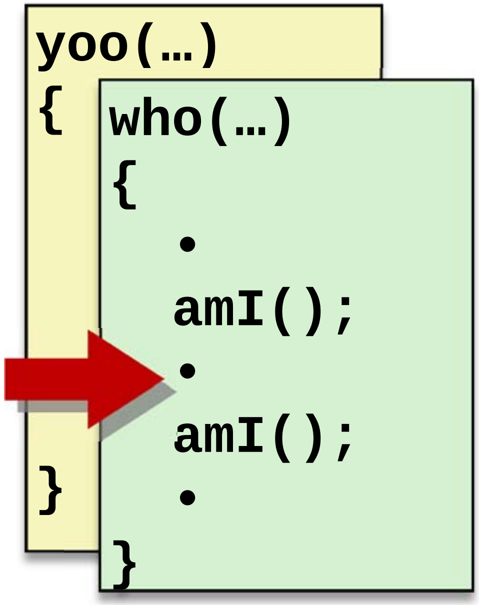
"deallocate" stack frame by moving %rsp back up

data still exists, but you shouldn't use it

7) Return from recursive call to `amI`



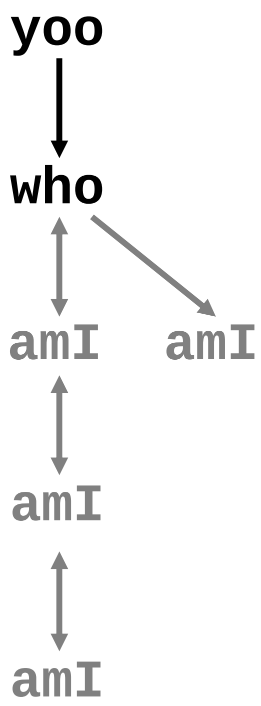
8) Return from call to `amI`



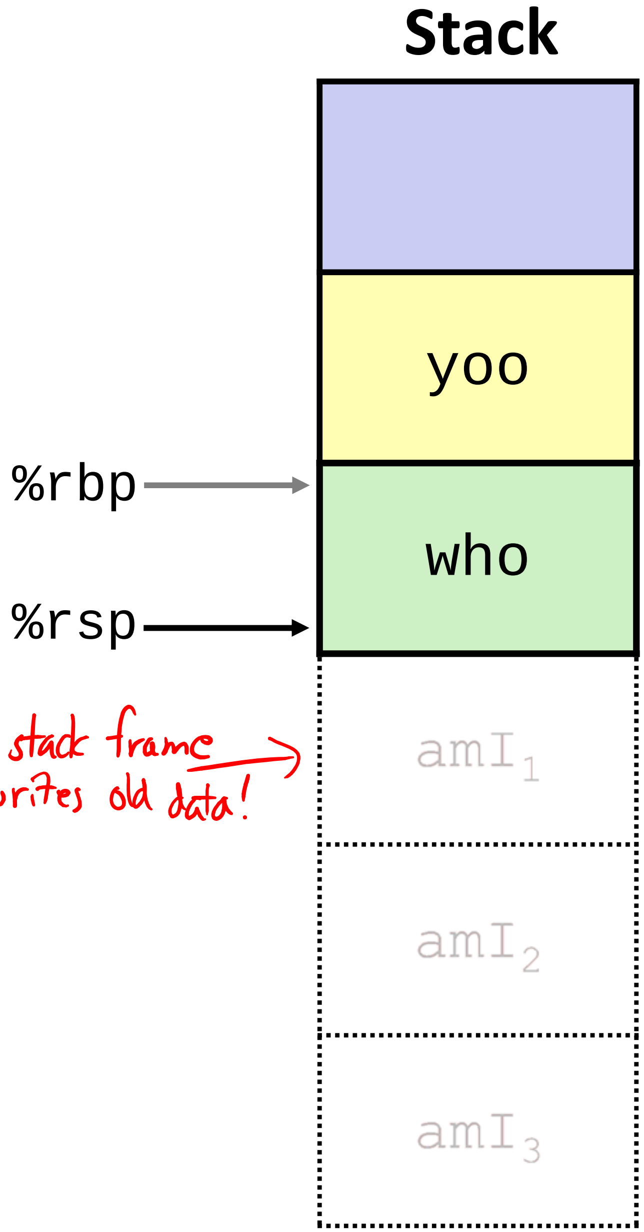
```
yoo(…)
{

}
```

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

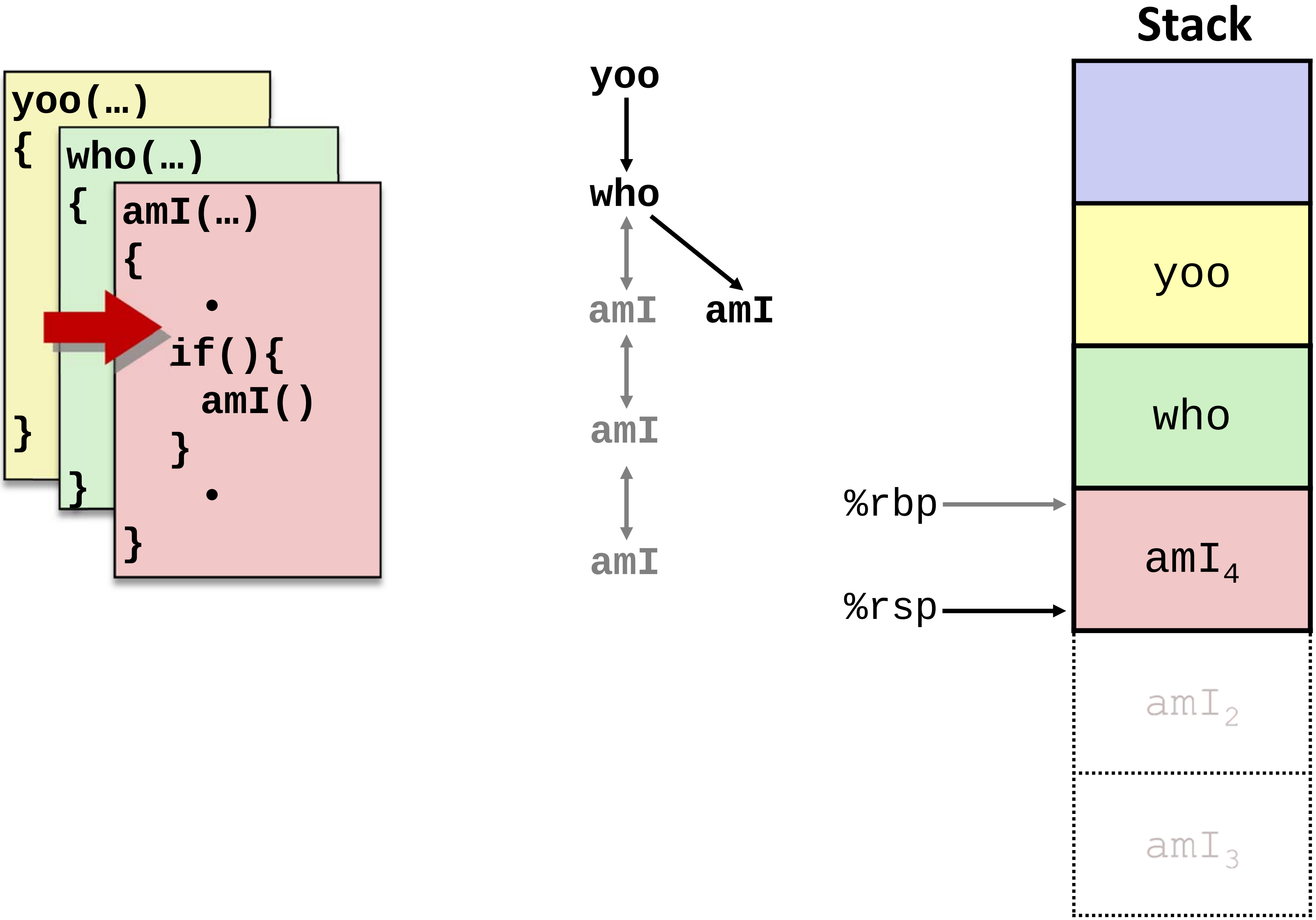


Stack



new stack frame overwrites old data!

9) (second) Call to amI (4)



```
yoo(…)
{
}
```

```
who(…)
{
}
```

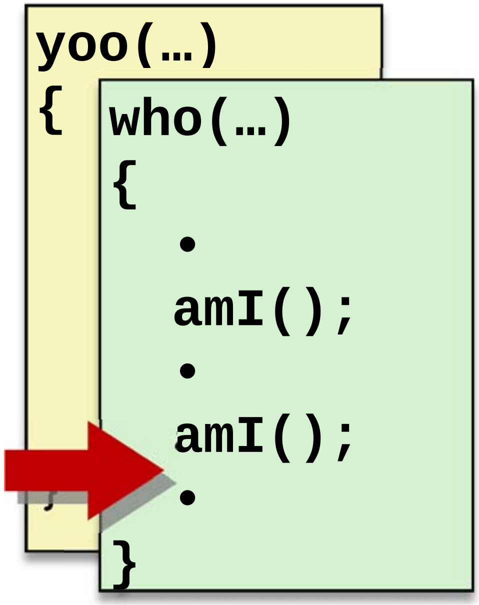
```
amI(…)
{
    •
  if(){
    amI()
  }
    •
}
```

yoo
who
amI amI
amI
amI

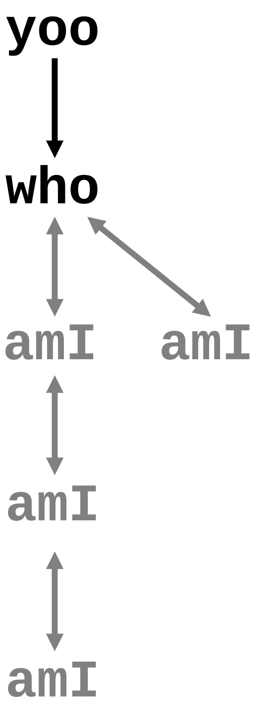
Stack

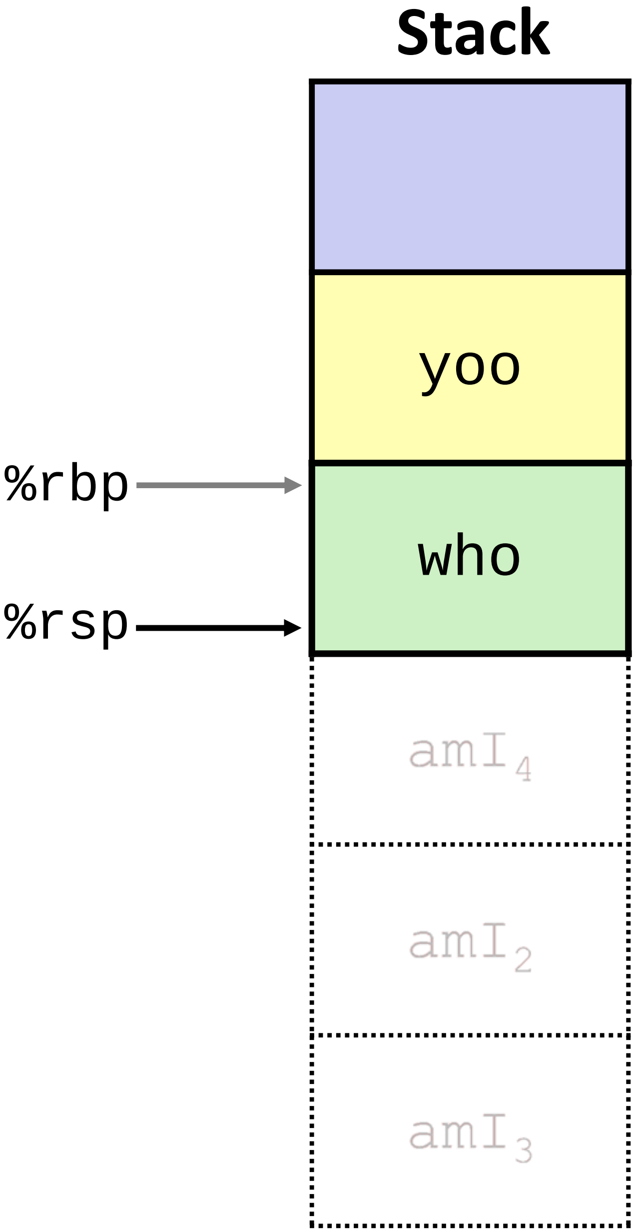
yoo
who
%rbp
amI$_4$
%rsp
amI$_2$
amI$_3$

10) Return from (second) call to amI



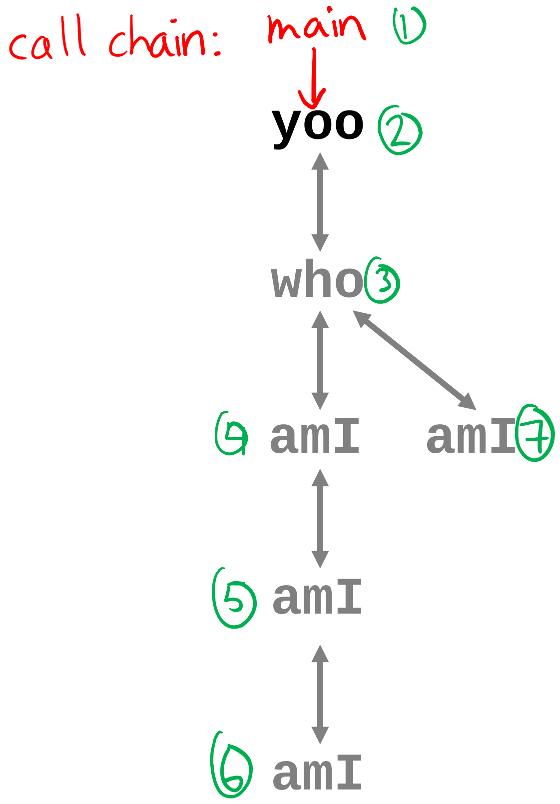
```
yoo(…)
{
  who(…)
  {
    •
    amI();
    •
    amI();
    •
  }
```

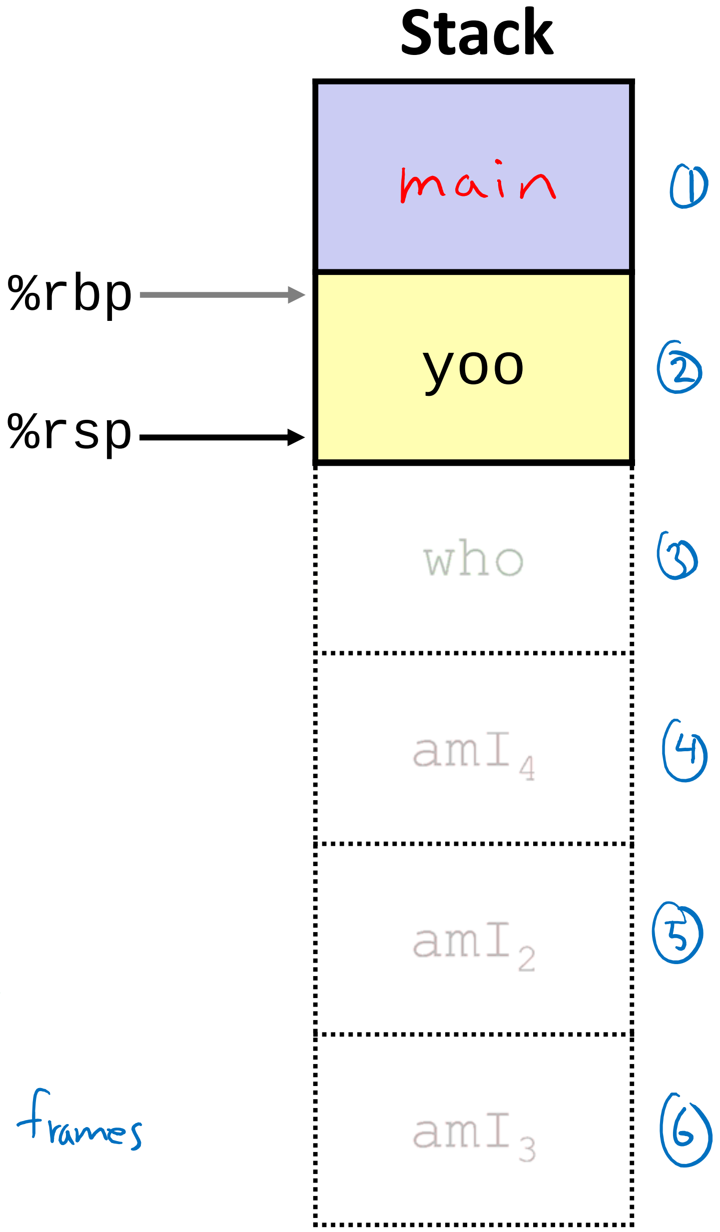




11) Return from call to who

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



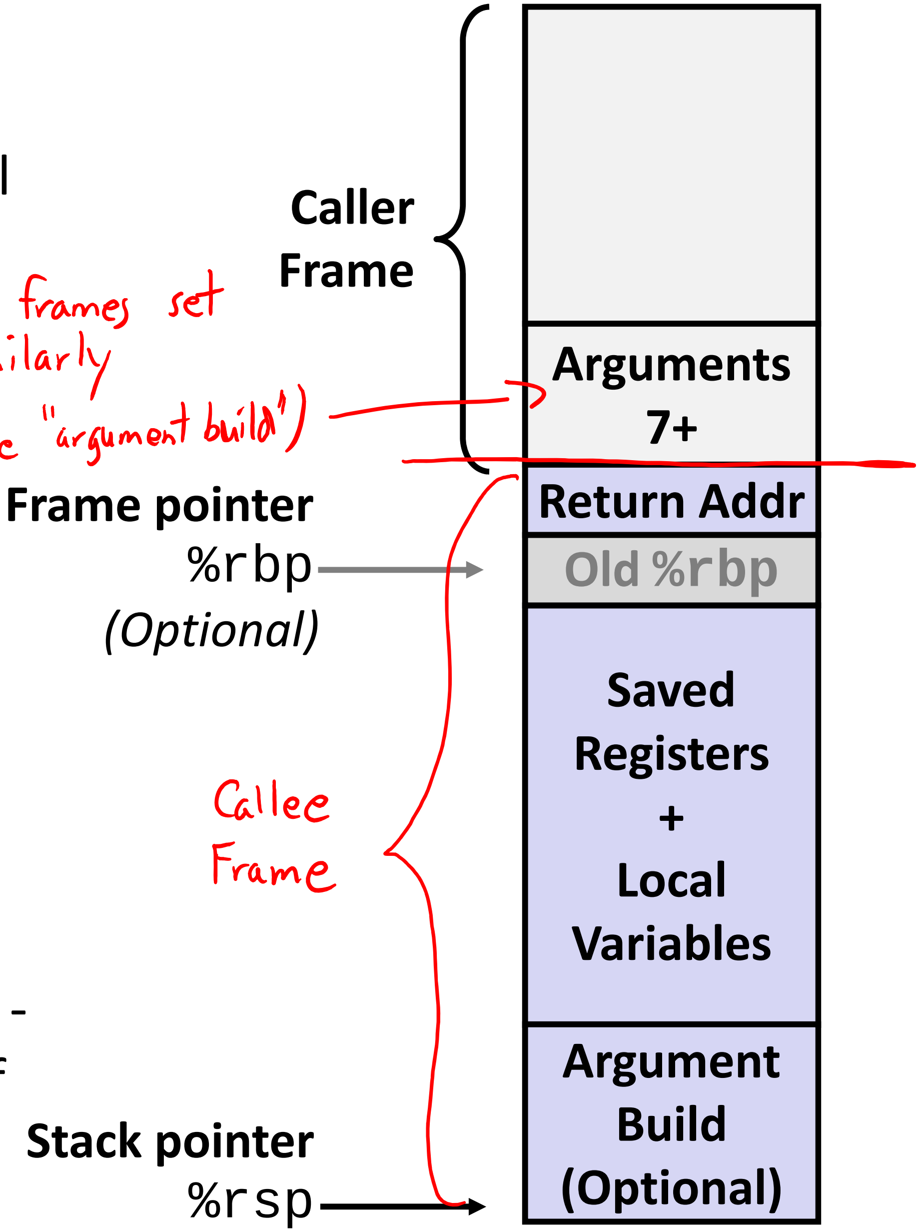


total stack frames created: 7

maximum stack depth: 6 frames

x86-64/Linux Stack Frame

- Caller's Stack Frame
  - Extra arguments (if > 6 args) for this call
- Current/Callee Stack Frame
  - Return address
    - Pushed by `call` instruction
  - Old frame pointer (optional)
  - Saved register context (when reusing registers)
  - Local variables (If can't be kept in registers)
  - "Argument build" area (If callee needs to call another function - parameters for function about to call, if needed)



all frames set up similarly (this is the "argument build")

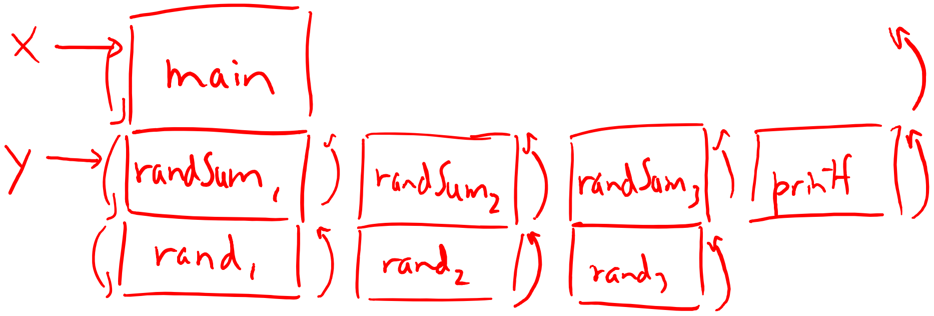
Peer Instruction Question

Vote only on 3rd question at
http://PollEv.com/justinh

- Answer the following questions about when `main()` is run (assume `x` and `y` stored on the Stack):

```
int main() {
    int i, x = 0;
    for(i=0;i<3;i++)
        x = randSum(x);
    printf("x = %d\n",x);
    return 0;
}
```

```
int randSum(int n) {
    int y = rand()%20;
    return n+y;
}
```



- *Higher/larger address*: [✓ x] or y?
- How many total stack frames are *created*? 8
- What is the maximum *depth* (# of frames) of the Stack?

A. 1 B. 2 [✓ C. 3] D. 4

Example: `increment`

adding val to value store at p

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

written this way to correspond to assembly

incr.c and incr.s posted on website so you can step through this example in gdb

```
increment:
  movq    (%rdi), %rax    # x=*p
  addq    %rax, %rsi      # y=x+val
  movq    %rsi, (%rdi)    # *p=y
  ret
```

val, y

Register	Use(s)
%rdi	1st arg (p)
%rsi	2nd arg (val), y
%rax	x, return value

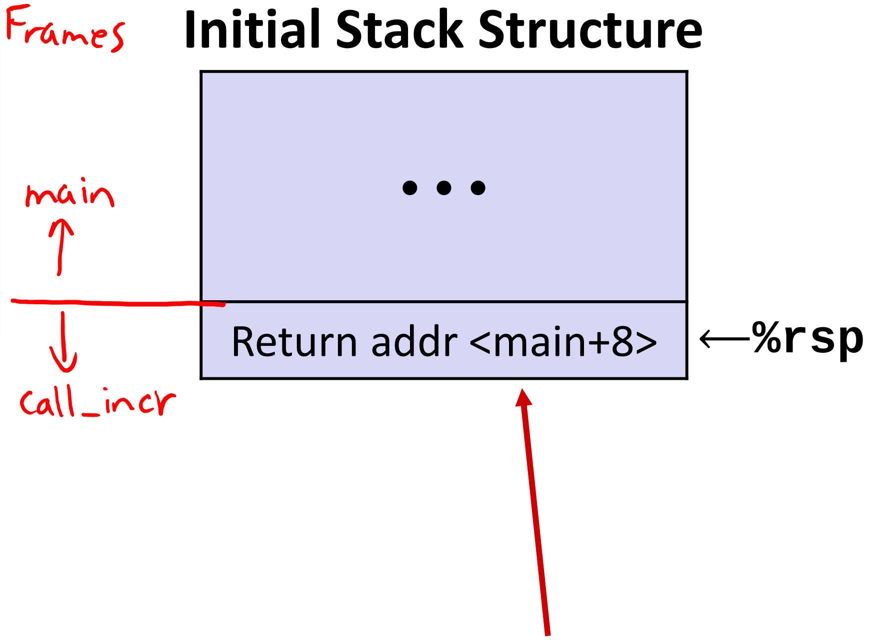
Procedure Call Example (initial state)

```
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1+v2;
}
```

p val

```
call_incr:
  subq    $16, %rsp
  movq    $351, 8(%rsp)
  movl    $100, %esi
  leaq    8(%rsp), %rdi
  call    increment
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

Initial Stack Structure



- Return address on stack is the address of instruction immediately *following* the call to “`call_incr`”
  - Shown here as `main`, but could be anything)
  - Pushed onto stack by `call call_incr`

Procedure Call Example (step 1)

```
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1+v2;
}
```

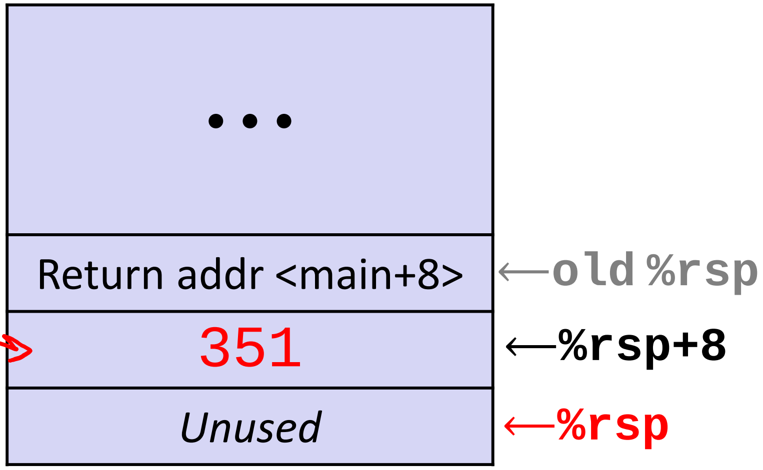
allocated on stack

```
call_incr:
  subq    $16, %rsp
  movq    $351, 8(%rsp)
  movl    $100, %esi
  leaq    8(%rsp), %rdi
  call    increment
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

Allocate space for local vars

"manual push"

Stack Structure



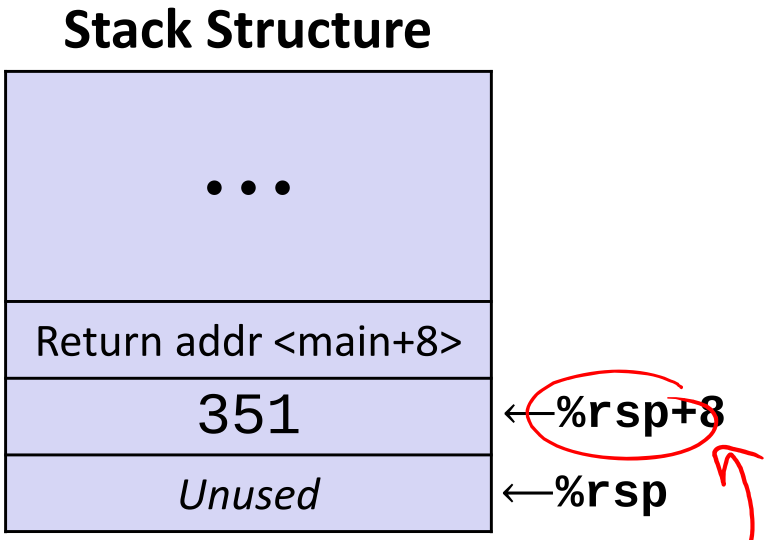
- Setup space for local variables
  - Only v1 needs space on the stack
- Compiler allocated extra space
  - Often does this for a variety of reasons, including alignment

Procedure Call Example (step 2)

```
long call_incr() {
                                %rdi   %rsi
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1+v2;
}
```

```
call_incr:
  subq    $16, %rsp
  movq    $351, 8(%rsp)
  movl    $100, %esi    # set val
  leaq    8(%rsp), %rdi # set p
  call    increment
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

Set up parameters for call to `increment`



Stack Structure

...

Return addr <main+8>

351 ← %rsp+8

Unused ← %rsp

Aside: `movl` is used because 100 is a small positive value that fits in 32 bits. High order bits of `rsi` get set to zero automatically. It takes *one less byte* to encode a `movl` than a `movq`.

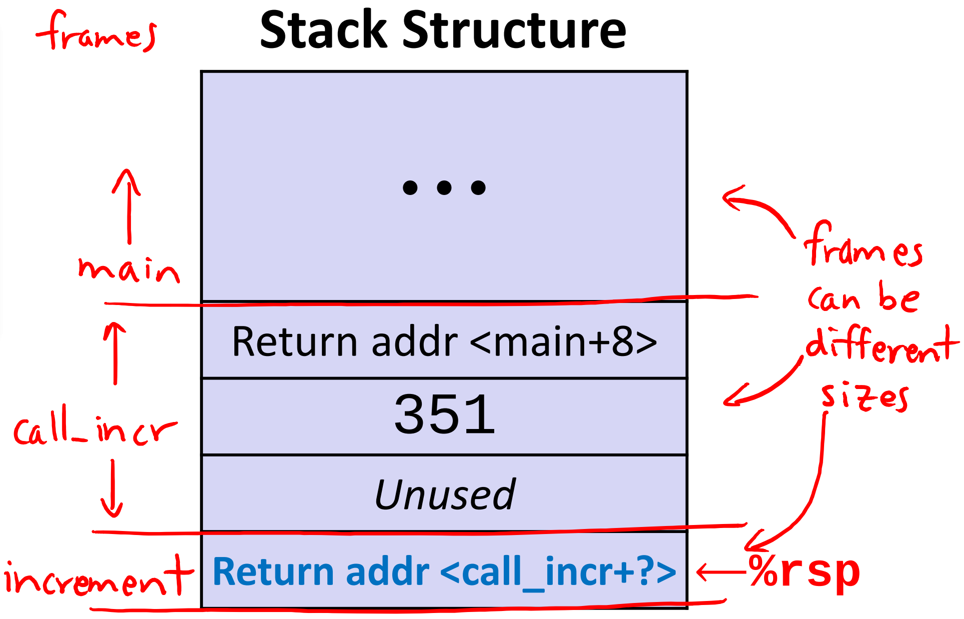
Register	Use(s)
`%rdi`	`&v1`
`%rsi`	`100`

Procedure Call Example (step 3)

```
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1+v2;
}
```

```
call_incr:
  subq    $16, %rsp
  movq    $351, 8(%rsp)
  movl    $100, %esi
  leaq    8(%rsp), %rdi
  call    increment
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

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



Stack Structure

frames

• • •

main

Return addr <main+8>

351

call_incr

Unused

increment

Return addr <call_incr+?>

%rsp

frames can be different sizes

- State while inside `increment`
  - **Return address** on top of stack is address of the `addq` instruction immediately following call to `increment`

Register	Use(s)
`%rdi`	`&v1`
`%rsi`	`100`
`%rax`	

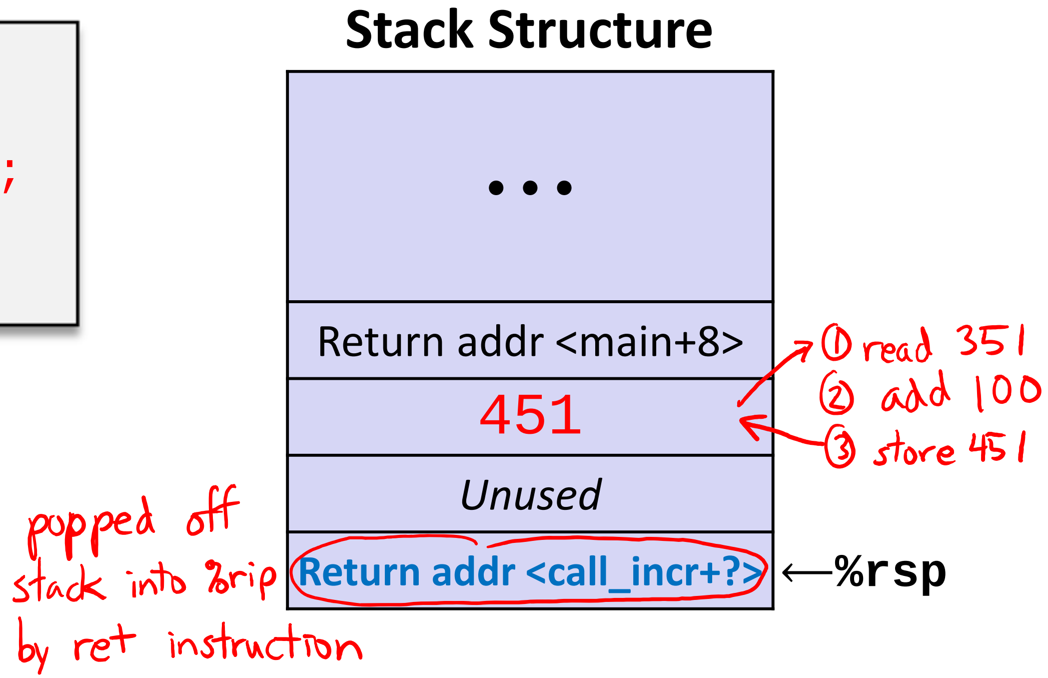
Procedure Call Example (step 4)

```
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1+v2;
}
```

```
call_incr:
  subq    $16, %rsp
  movq    $351, 8(%rsp)
  movl    $100, %esi
  leaq    8(%rsp), %rdi
  call    increment
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

```
increment:
① movq    (%rdi), %rax # x = *p
② addq    %rax, %rsi   # y = x+100
③ movq    %rsi, (%rdi) # *p = y
  ret
```

Stack Structure



• • •

Return addr <main+8>

451

Unused

Return addr <call_incr+?>

←%rsp

① read 351
② add 100
③ store 451

popped off stack into %rip by ret instruction

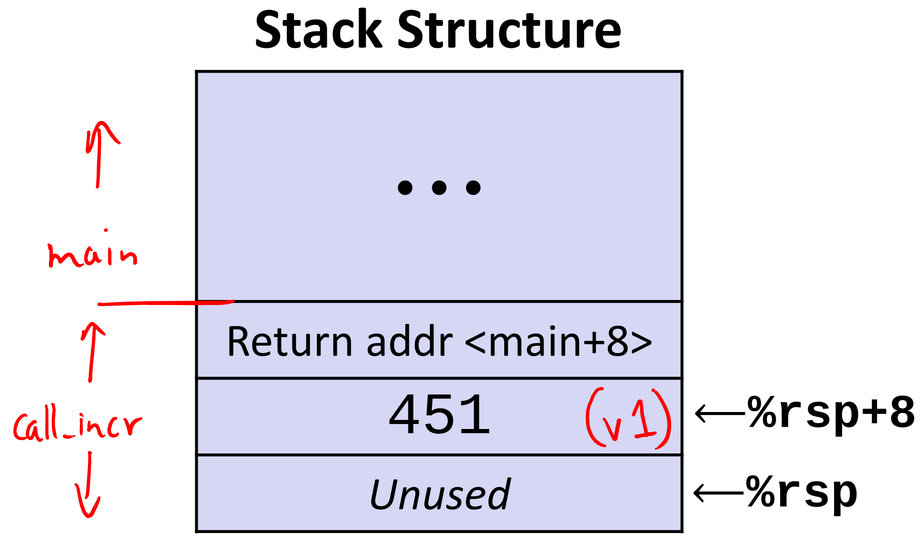
- State while inside `increment`
  - *After* code in body has been executed

Register	Use(s)
%rdi	&v1
%rsi	451
%rax	351

Procedure Call Example (step 5)

```
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1+v2;
}
```

```
call_incr:
  subq    $16, %rsp
  movq    $351, 8(%rsp)
  movl    $100, %esi
  leaq    8(%rsp), %rdi
  call    increment
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```



Stack Structure

• • •

Return addr <main+8>

451 (v1) ←%rsp+8

Unused ←%rsp

- After returning from call to `increment`
  - Registers and memory have been modified and return address has been popped off stack

Register	Use(s)
%rdi	&v1
%rsi	451
%rax	351 (v2)

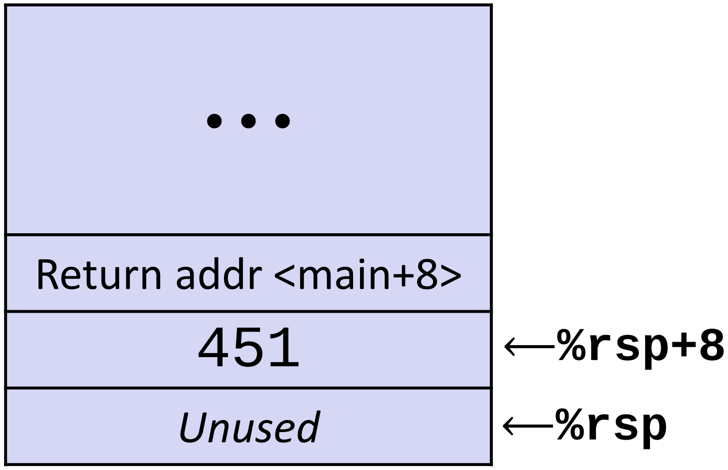
Procedure Call Example (step 6)

```
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1+v2;
}
```

```
call_incr:
  subq    $16, %rsp
  movq    $351, 8(%rsp)
  movl    $100, %esi
  leaq    8(%rsp), %rdi
  call    increment
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

← Update **%rax** to contain v1+v2

Stack Structure



Register	Use(s)
%rdi	&v1
%rsi	451
%rax	451+351

Procedure Call Example (step 7)

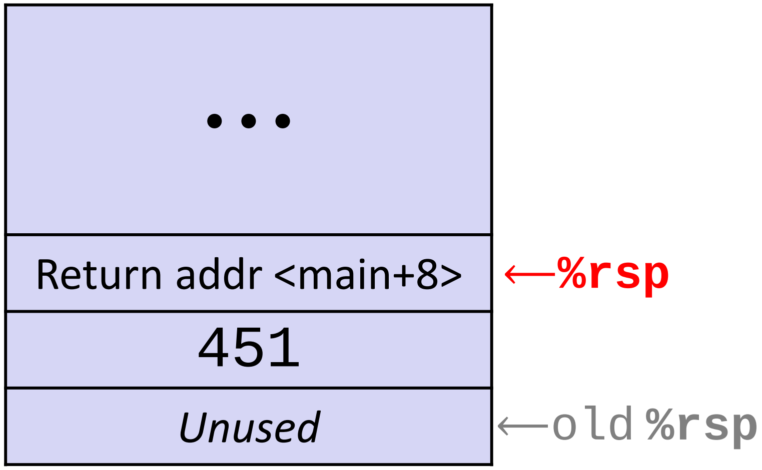
```
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1+v2;
}
```

```
call_incr:
  subq    $16, %rsp
  movq    $351, 8(%rsp)
  movl    $100, %esi
  leaq    8(%rsp), %rdi
  call    increment
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

De-allocate space for local **vars**

(make sure %rsp points to return addr before ret)

Stack Structure



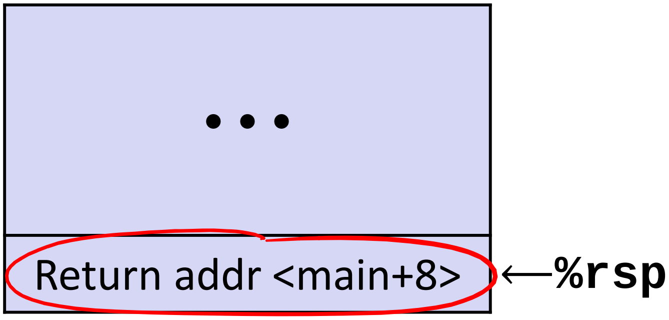
Register	Use(s)
%rdi	&v1
%rsi	451
%rax	802

Procedure Call Example (step 8)

```
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1+v2;
}
```

```
call_incr:
  subq    $16, %rsp
  movq    $351, 8(%rsp)
  movl    $100, %esi
  leaq    8(%rsp), %rdi
  call    increment
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

Stack Structure



popped off
stack into %rip
by ret

- State *just before* returning from call to `call_incr`

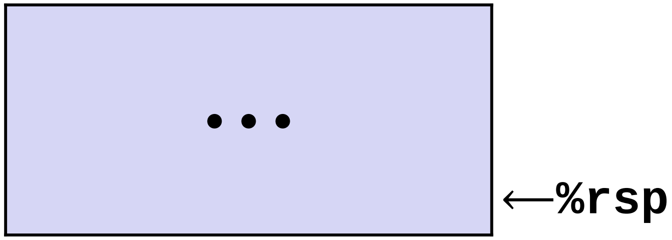
Register	Use(s)
%rdi	&v1
%rsi	451
%rax	802

Procedure Call Example (step 9)

```
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1+v2;
}
```

```
call_incr:
  subq    $16, %rsp
  movq    $351, 8(%rsp)
  movl    $100, %esi
  leaq    8(%rsp), %rdi
  call    increment
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

Final Stack Structure



- State immediately *after* returning from call to `call_incr`
  - Return addr has been popped off stack
  - Control has returned to the instruction immediately following the call to `call_incr` (not shown here)

Register	Use(s)
`%rdi`	`&v1`
`%rsi`	`451`
`%rax`	`802`