CS 240
Foundations of Computer Systems

WELLESLEY

x86: Procedures and the Call Stack

The call stack discipline
x86 procedure call and return instructions
x86 calling conventions
x86 register-saving conventions

https://cs.wellesley.edu/~cs240/

Why procedures?

Why functions? Why methods?

```
int contains_char(char* haystack, char needle) {
  while (*haystack != '\0') {
    if (*haystack == needle) return 1;
    haystack++;
  }
  return 0;
}
```

Procedural Abstraction

Implementing procedures

1. How does a caller pass arguments to a procedure? ✓

2. How does a caller receive a return value from a procedure? ✓

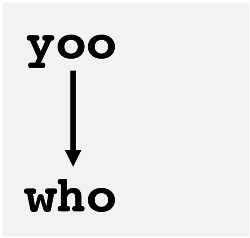
3. How does a procedure know where to return (what code to execute next when done)? ??

4. Where does a procedure store local variables? ✓?

1. How do procedures share limited registers and memory? ??

Procedure call/return: Jump?

Call Chain



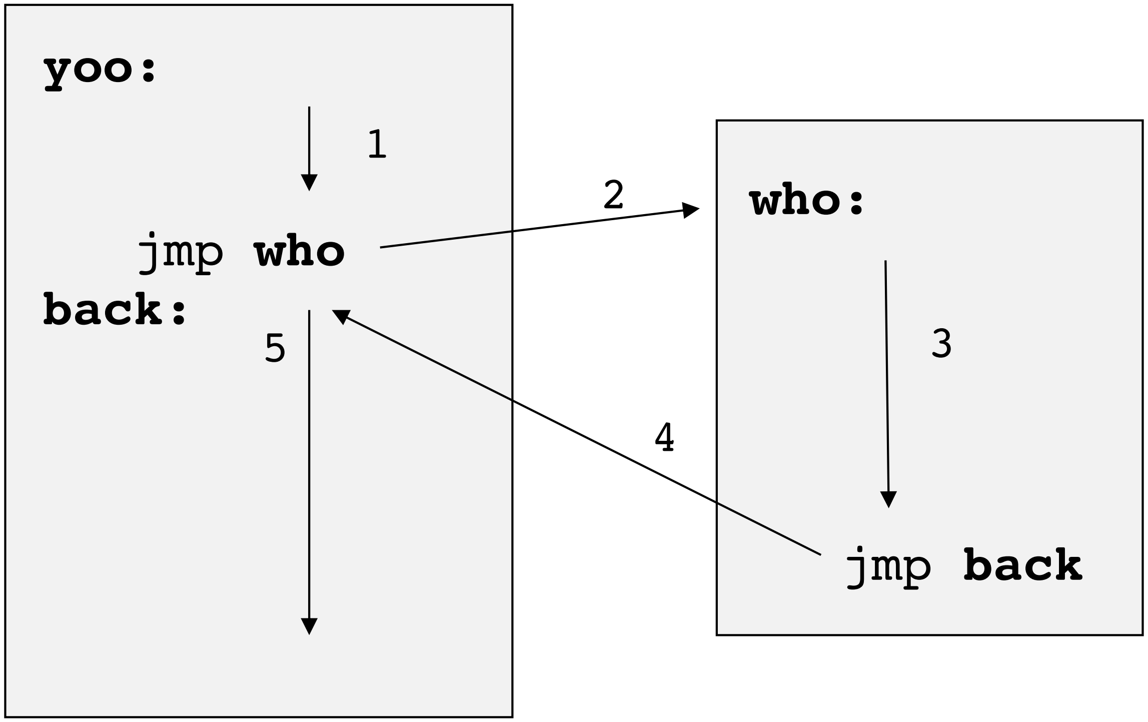
```
yoo(…) {
   • • •
   who();
   • • •
}
```

```
who(…) {
   • • •

   • • •

   • • •
}
```

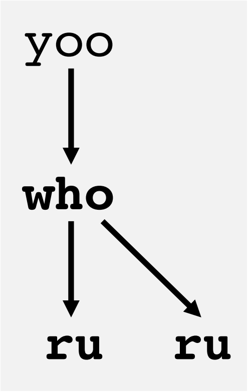
```
ru(…) {
    • • •
}
```



But what if we want to call a function from multiple places in the code?

Procedure call/return: Jump? Broken!

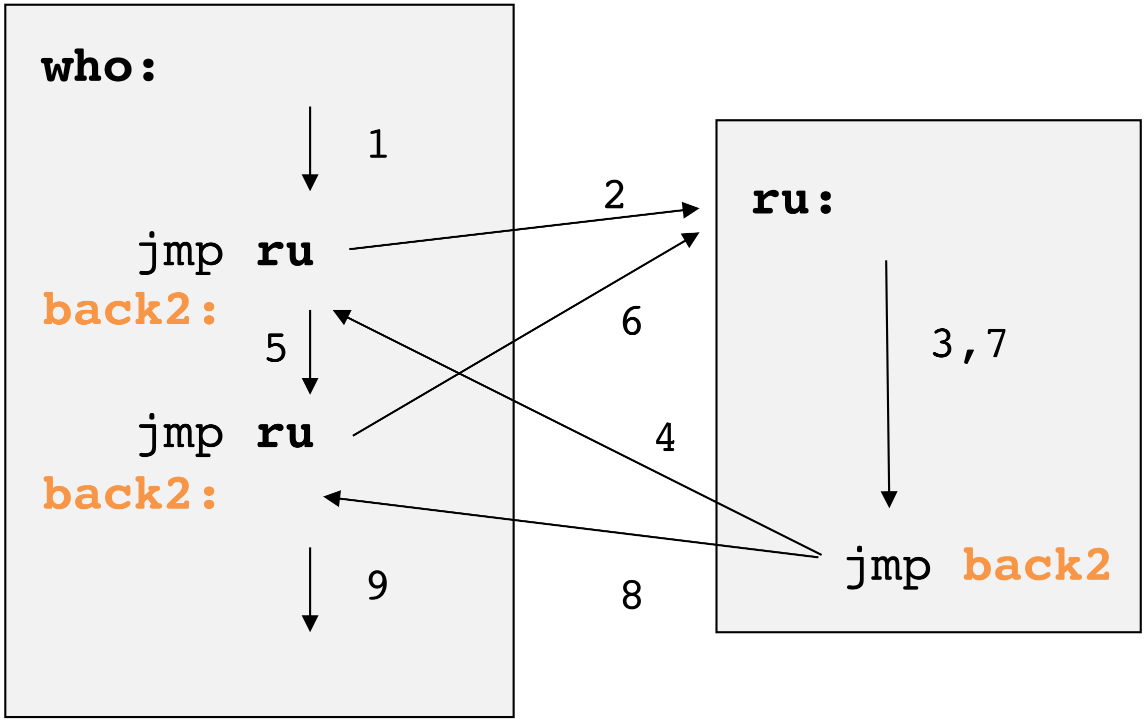
Call Chain



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

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

```
ru(…) {
    • • •
}
```



But what if we want to call a function from multiple places in the code?

Broken: needs to track context.

Implementing procedures

requires **separate storage** ***per call!***

(not just per procedure)

1. How does a caller pass arguments to a procedure? ✓

2. How does a caller receive a return value from a procedure? ✓

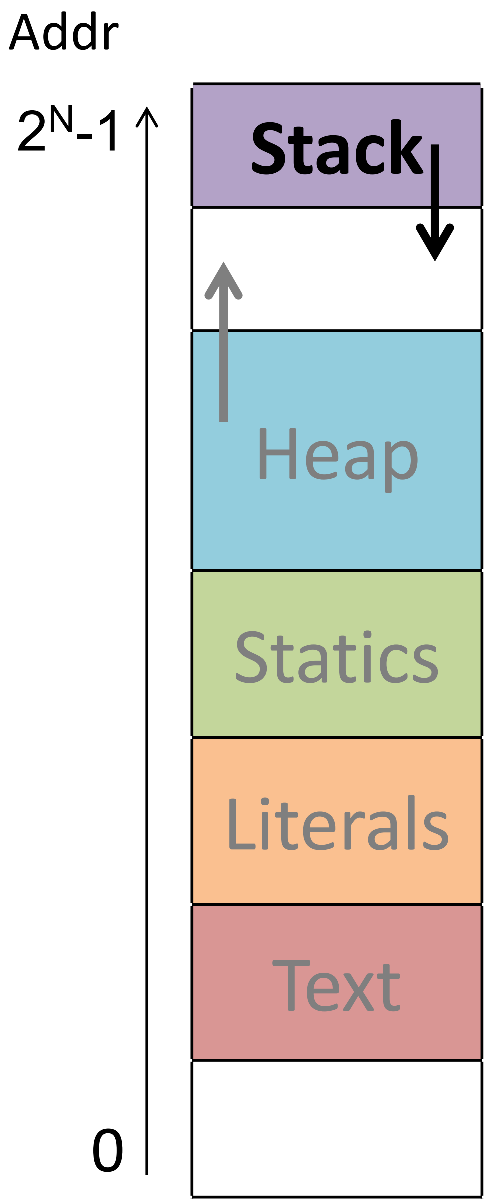
3. How does a procedure know where to return
   (what code to execute next when done)? ??

4. Where does a procedure store local variables? ✓?

1. How do procedures share limited registers and memory? ??

reminder

Memory Layout



Perm	Contents	Managed by	Initialized
RW	**Procedure context**	**Compiler**	**Run-time**
RW	Dynamic data structures	Programmer, malloc/free, new/GC	Run-time
RW	Global variables/ static data structures	Compiler/ Assembler/Linker	Startup
R	String literals	Compiler/ Assembler/Linker	Startup
X	Instructions	Compiler/ Assembler/Linker	Startup

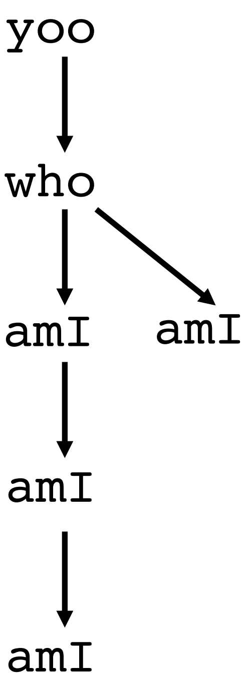
Call stack tracks context

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

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

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

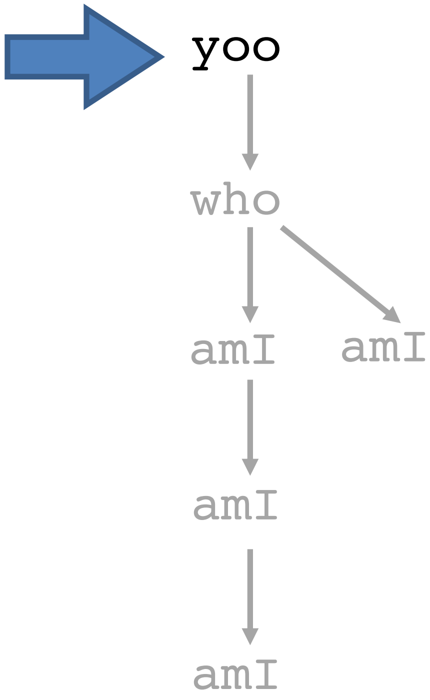
Example
Call Chain

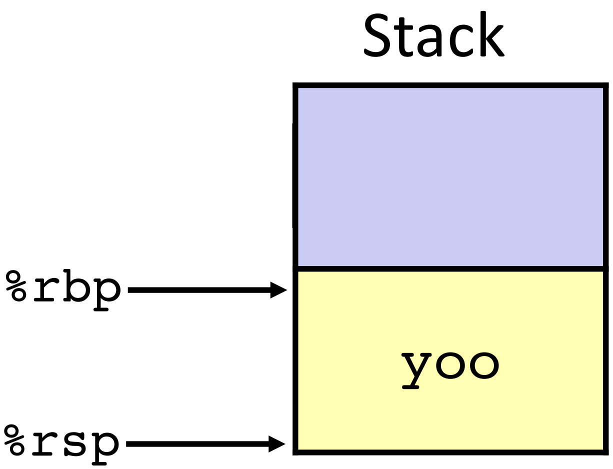


Procedure `amI` is recursive
(calls itself)

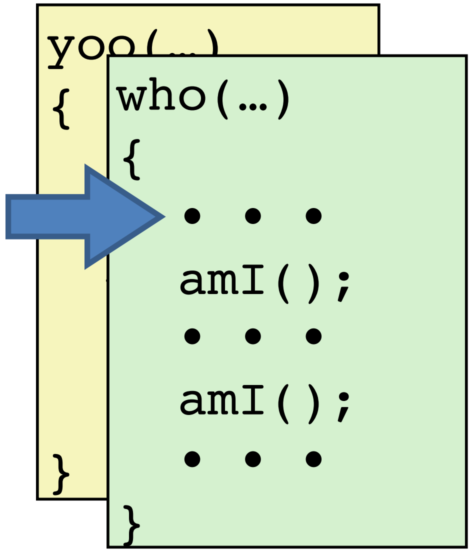
Call stack tracks context

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





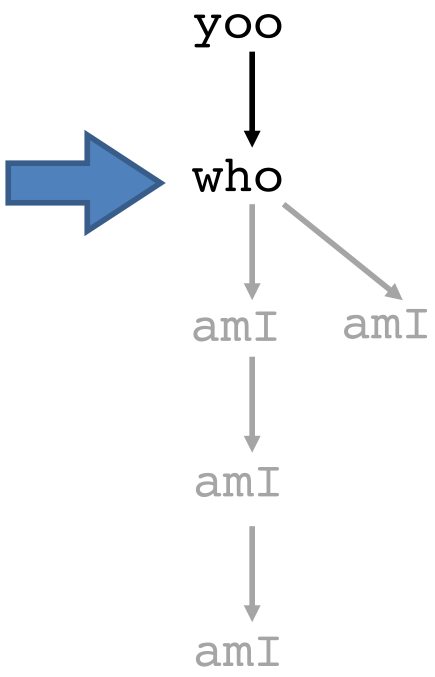
Call stack tracks context

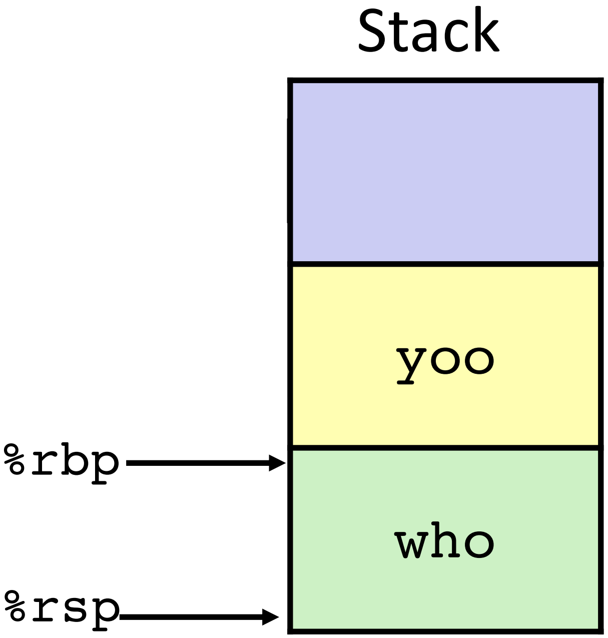


```
yoo(…)
{

}
```

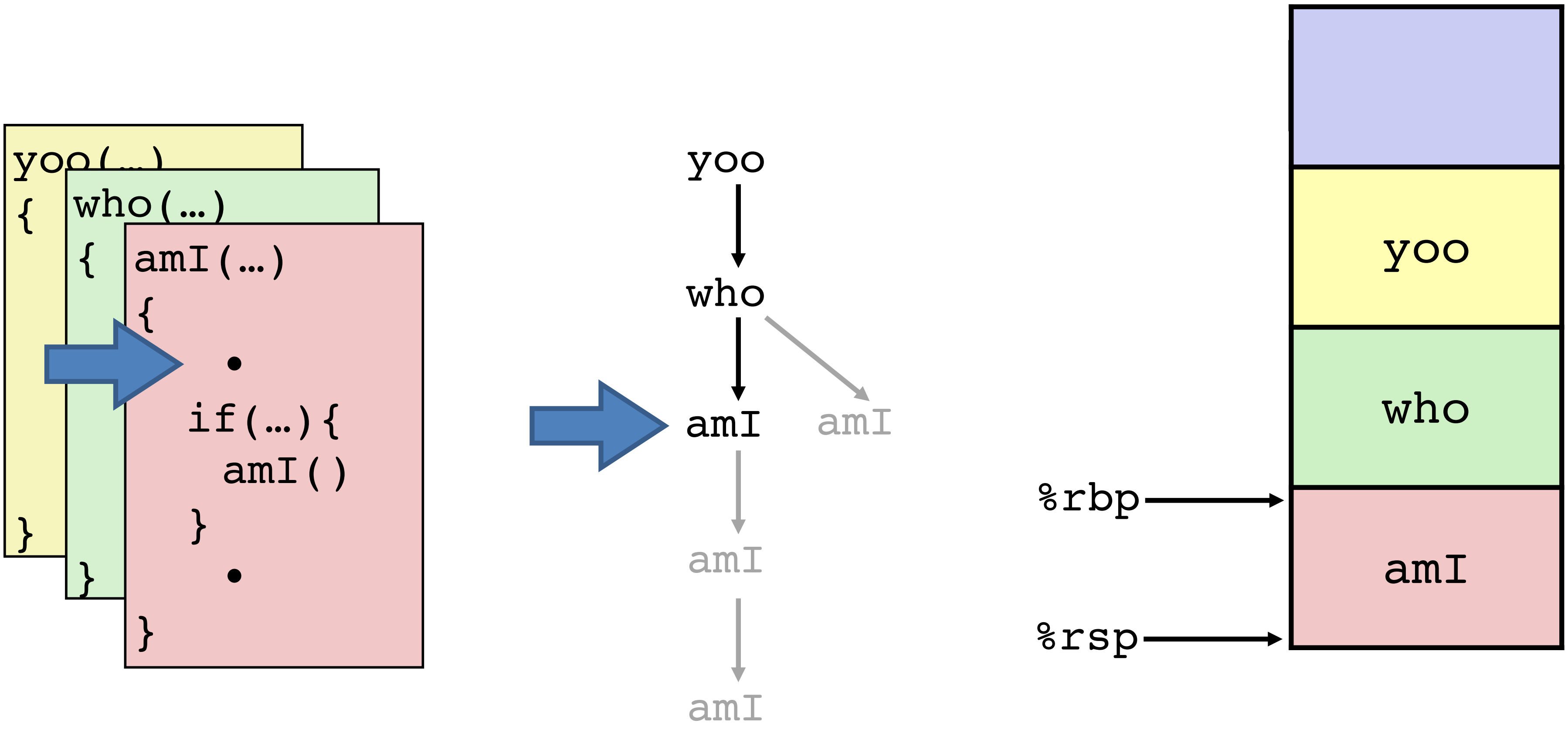
```
who(…)
{
  • • •
  amI();
  • • •
  amI();
  • • •
}
```





Call stack tracks context

Stack

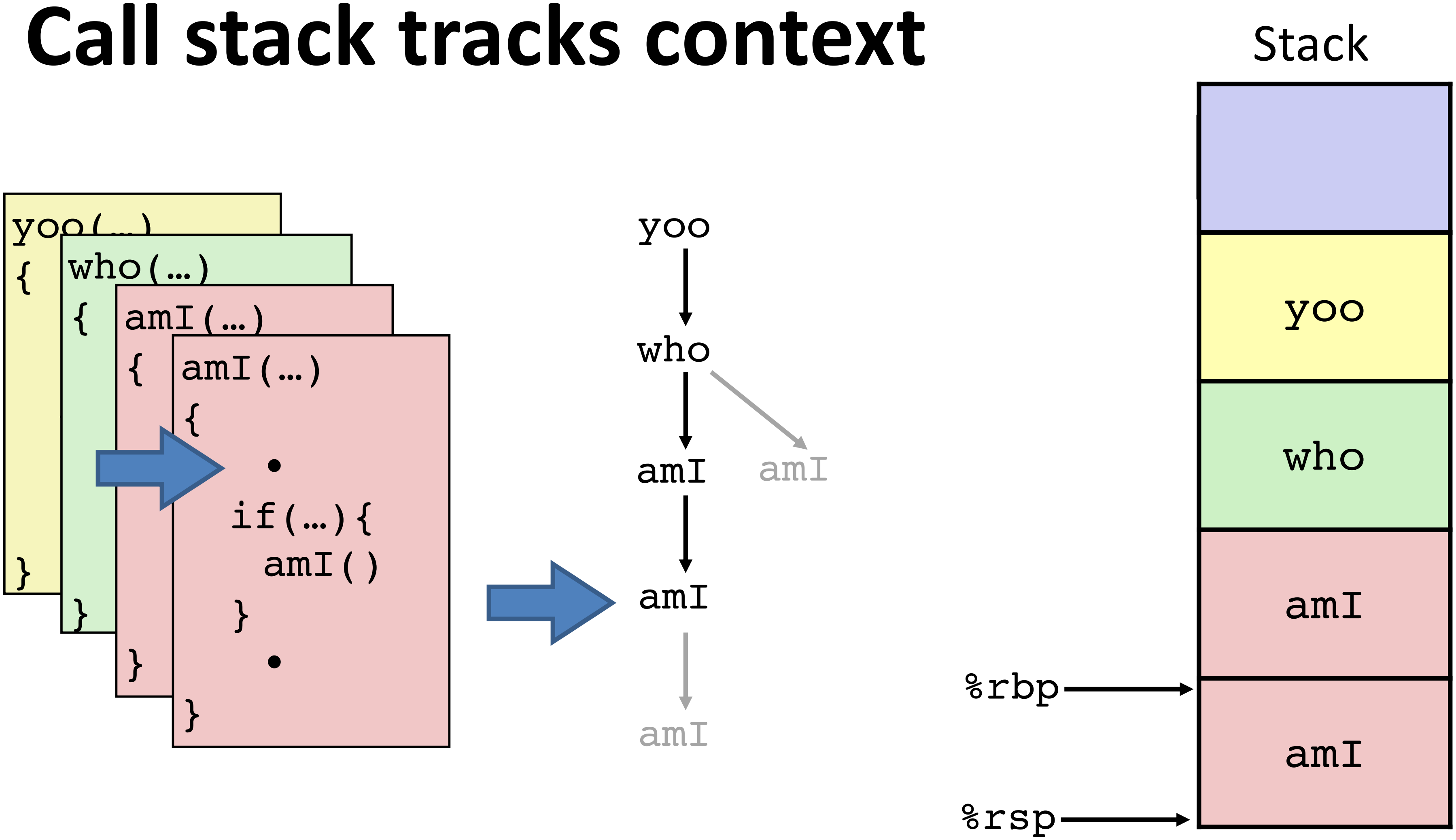


```
yoo(…)
{
}
```

```
who(…)
{
}
```

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

Call stack tracks context



```
yoo(…)
{
}
```

```
who(…)
{
}
```

```
amI(…)
{
}
```

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

yoo

who

amI

amI

amI

amI

Stack

yoo

who

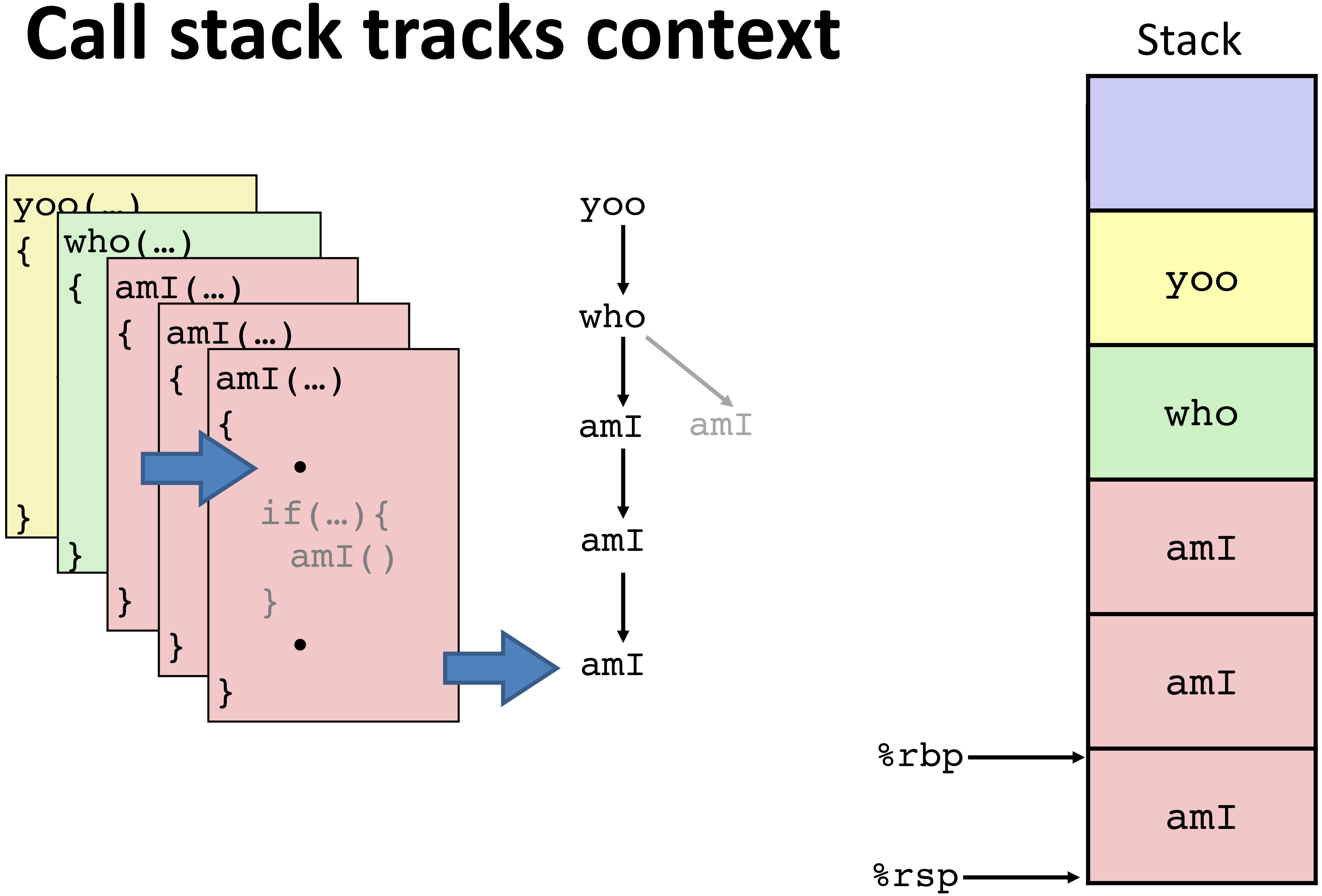
amI

amI

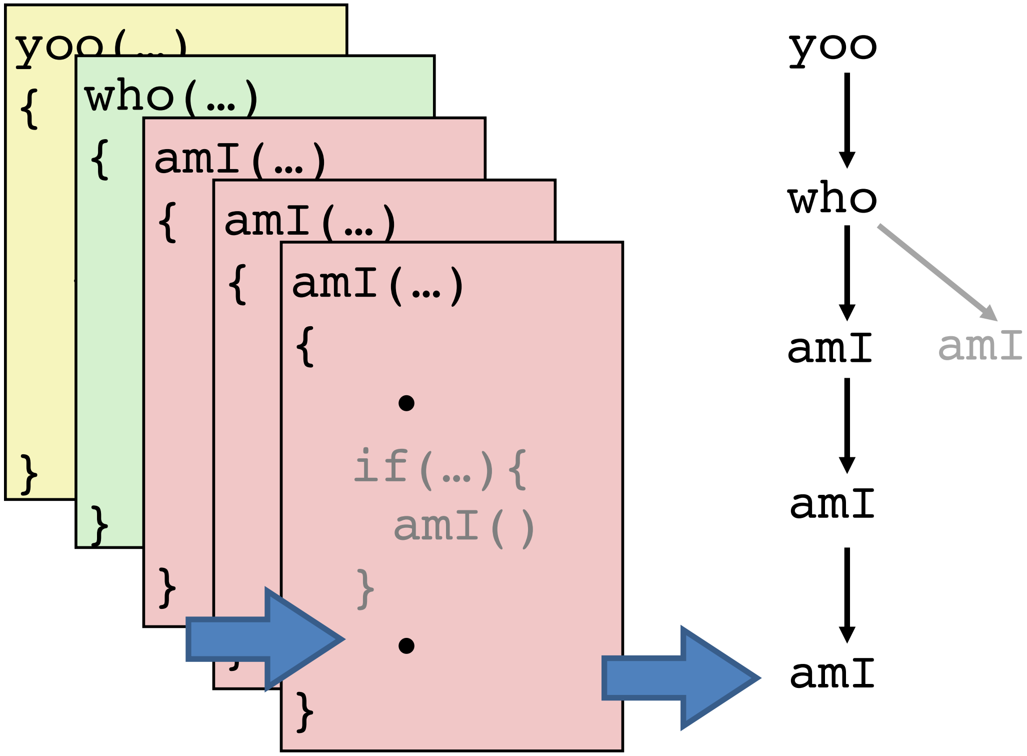
%rbp

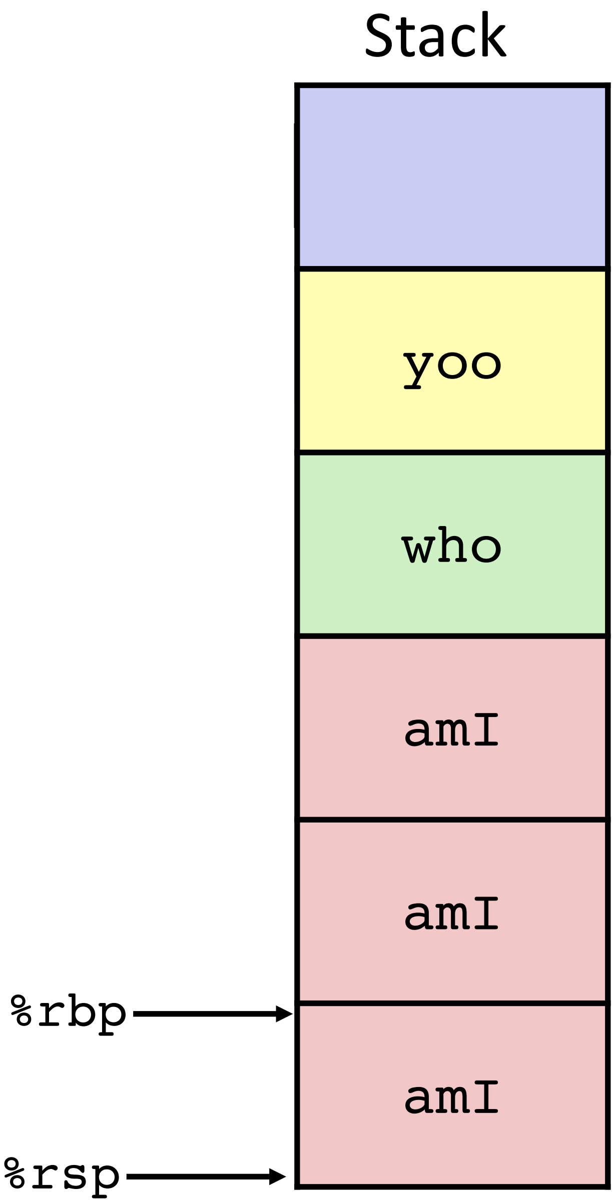
%rsp

Call stack tracks context

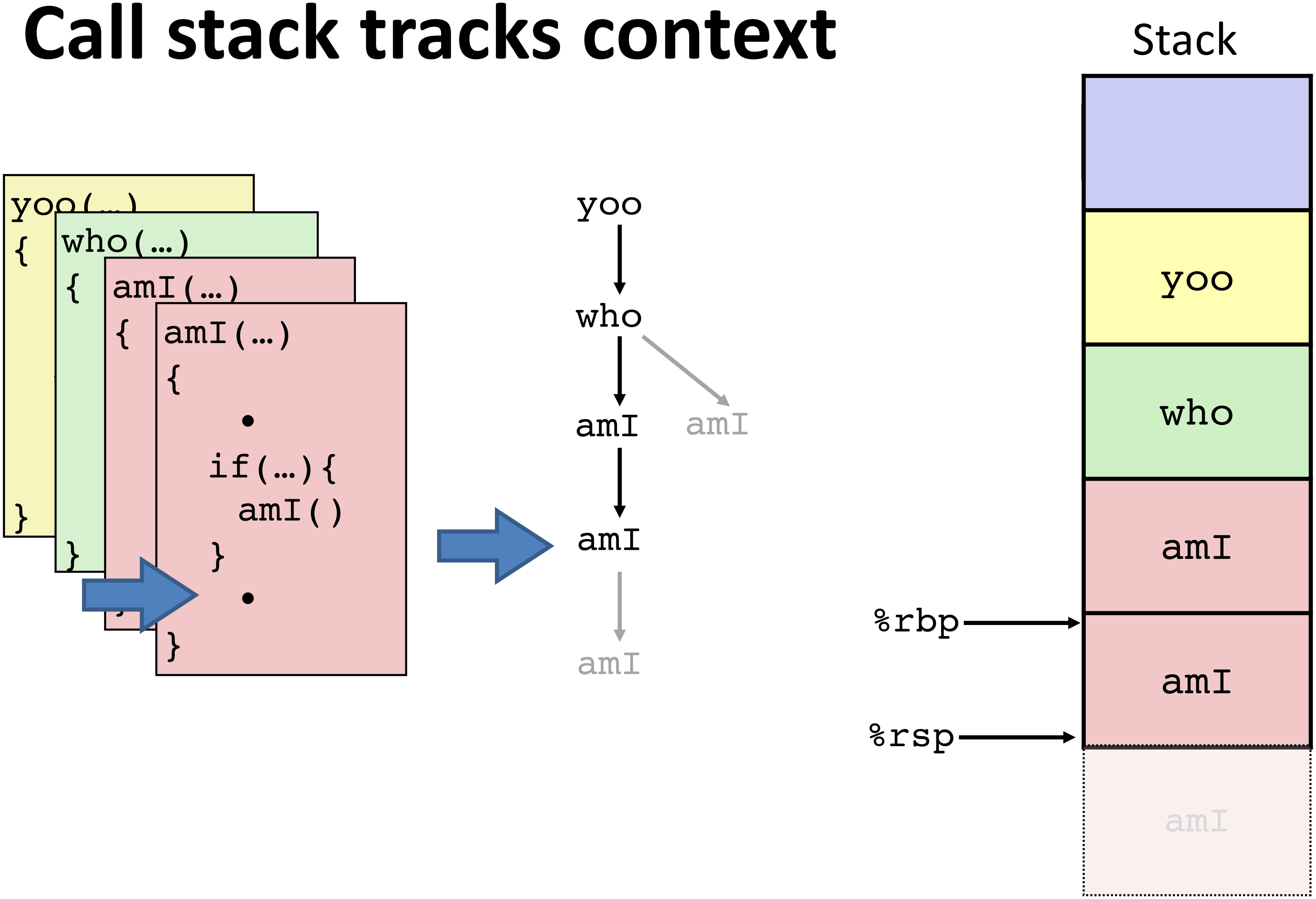


Call stack tracks context



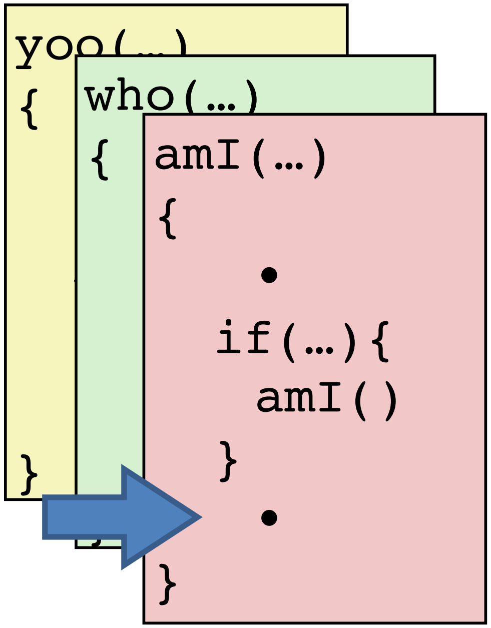


Call stack tracks context



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

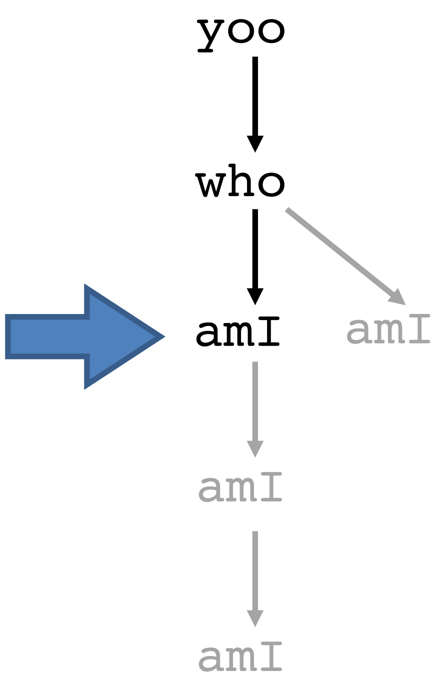
Call stack tracks context

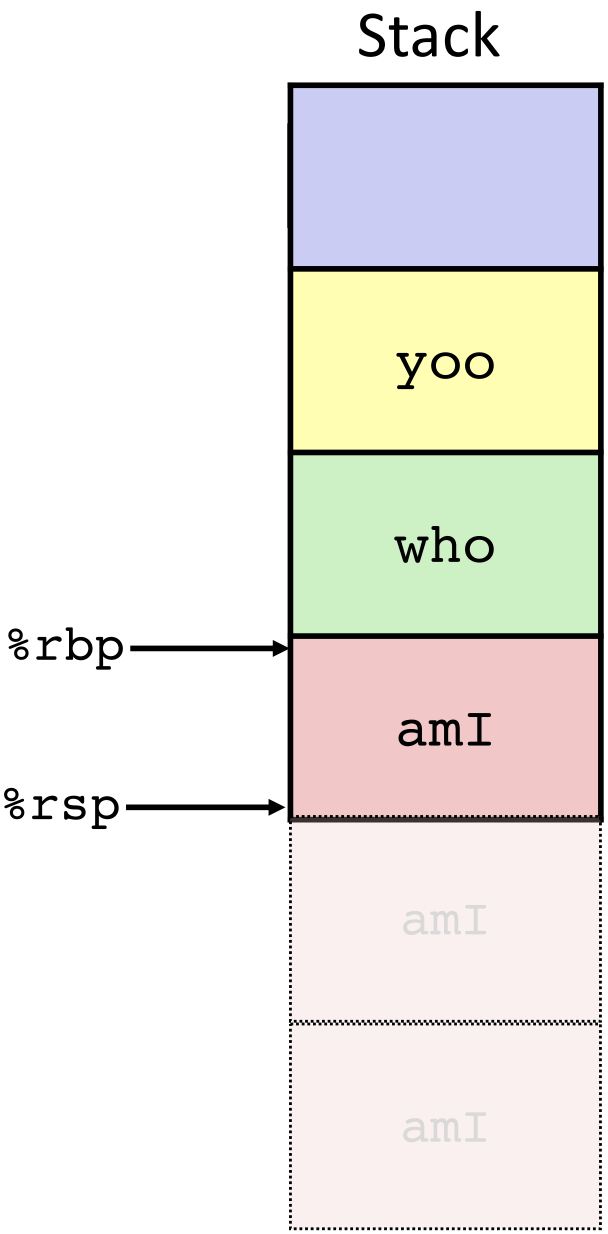


```
yoo(…)
{
}
```

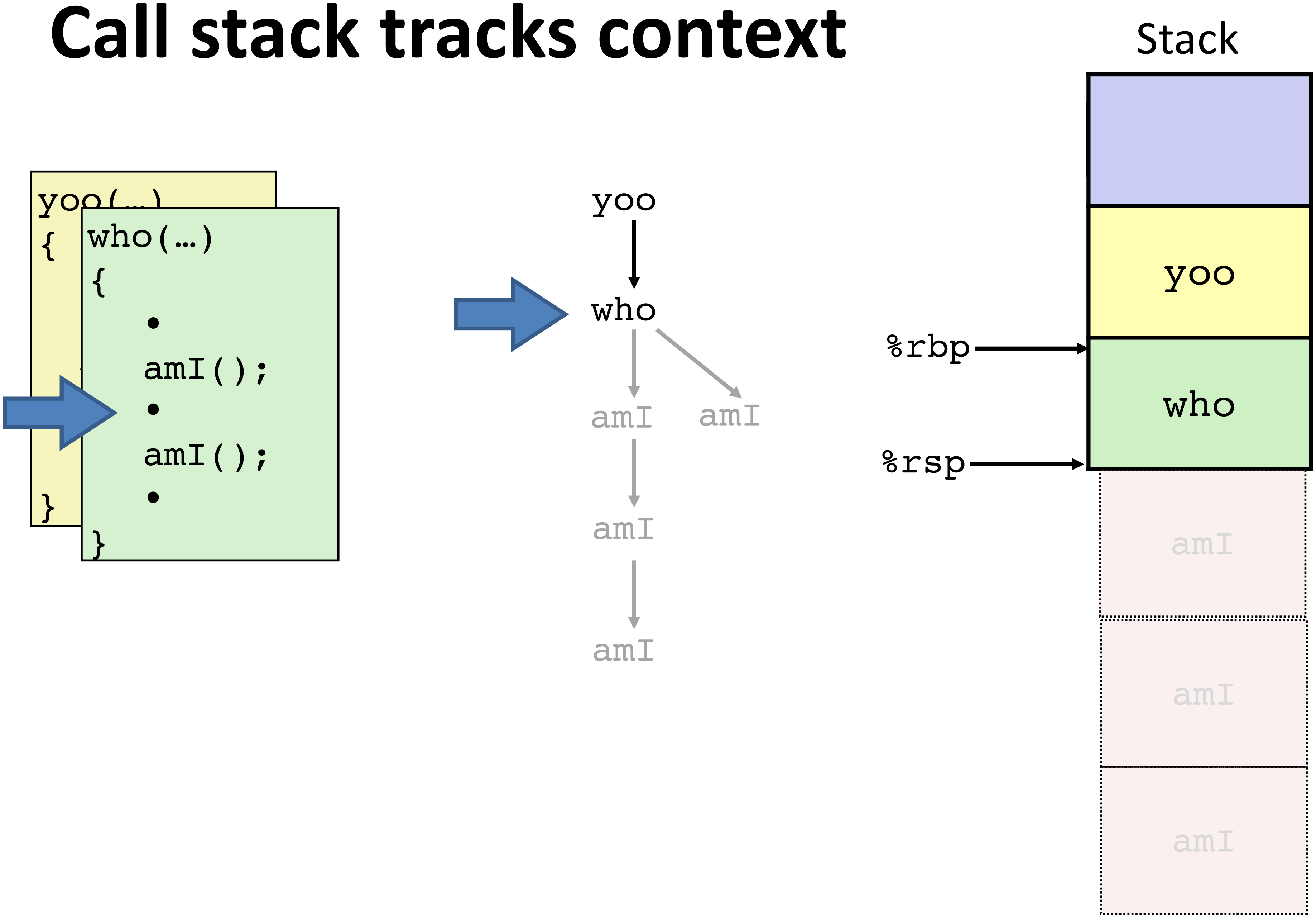
```
who(…)
{
}
```

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





Call stack tracks context

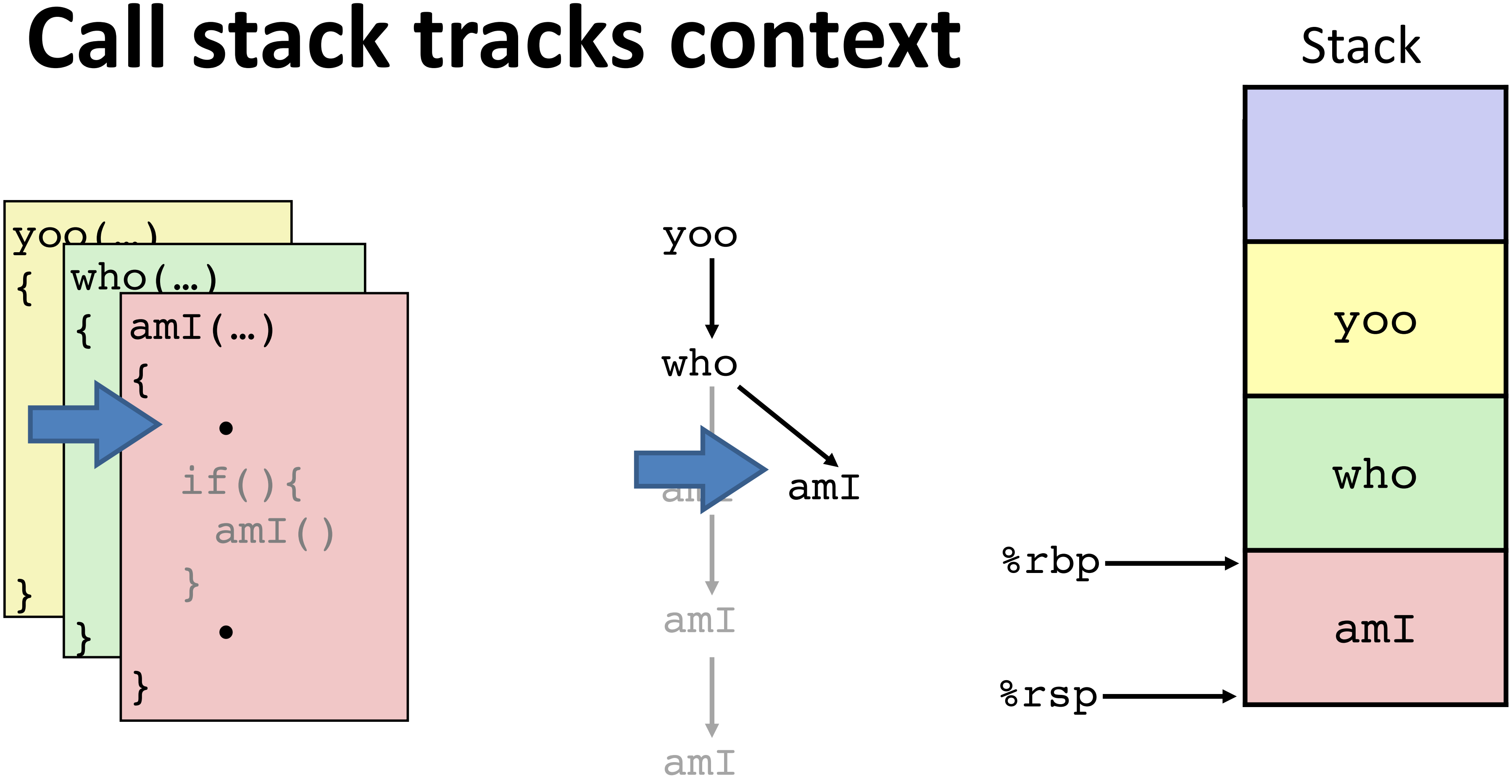


```
yoo(…)
{

}
```

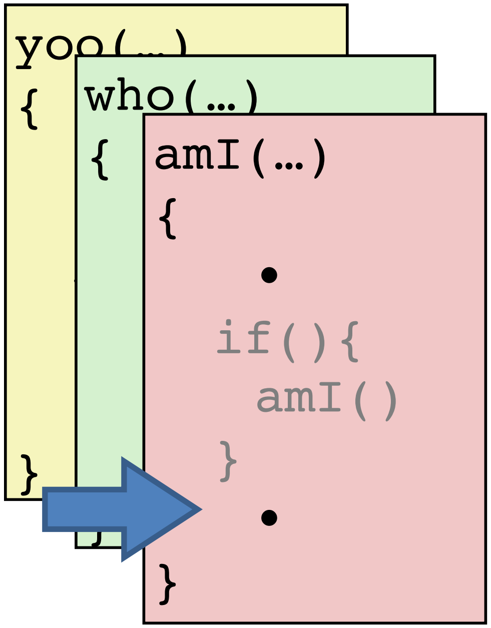
```
who(…)
{
   •
   amI();
   •
   amI();
   •
}
```

Call stack tracks context

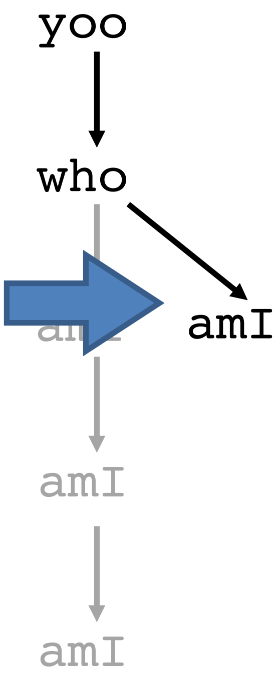


```
amI(…)
{
    •
  if(){
    amI()
  }
    •
}
```

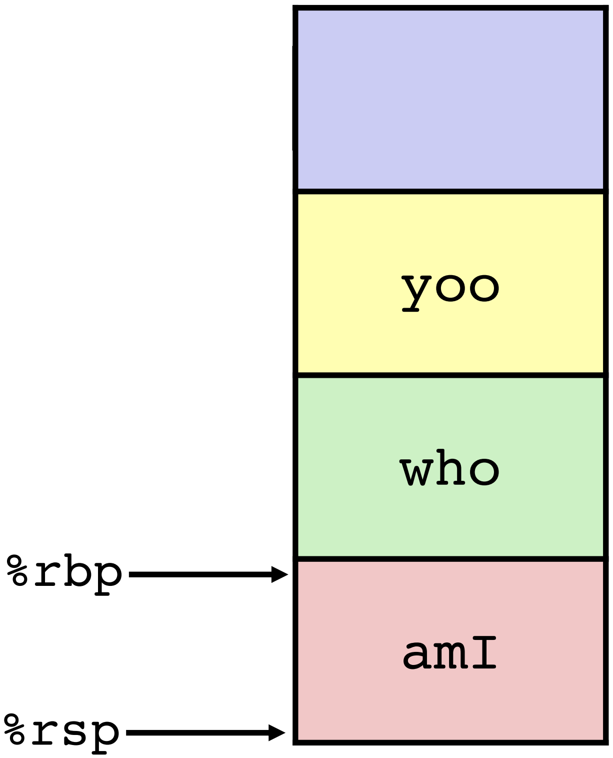
Call stack tracks context



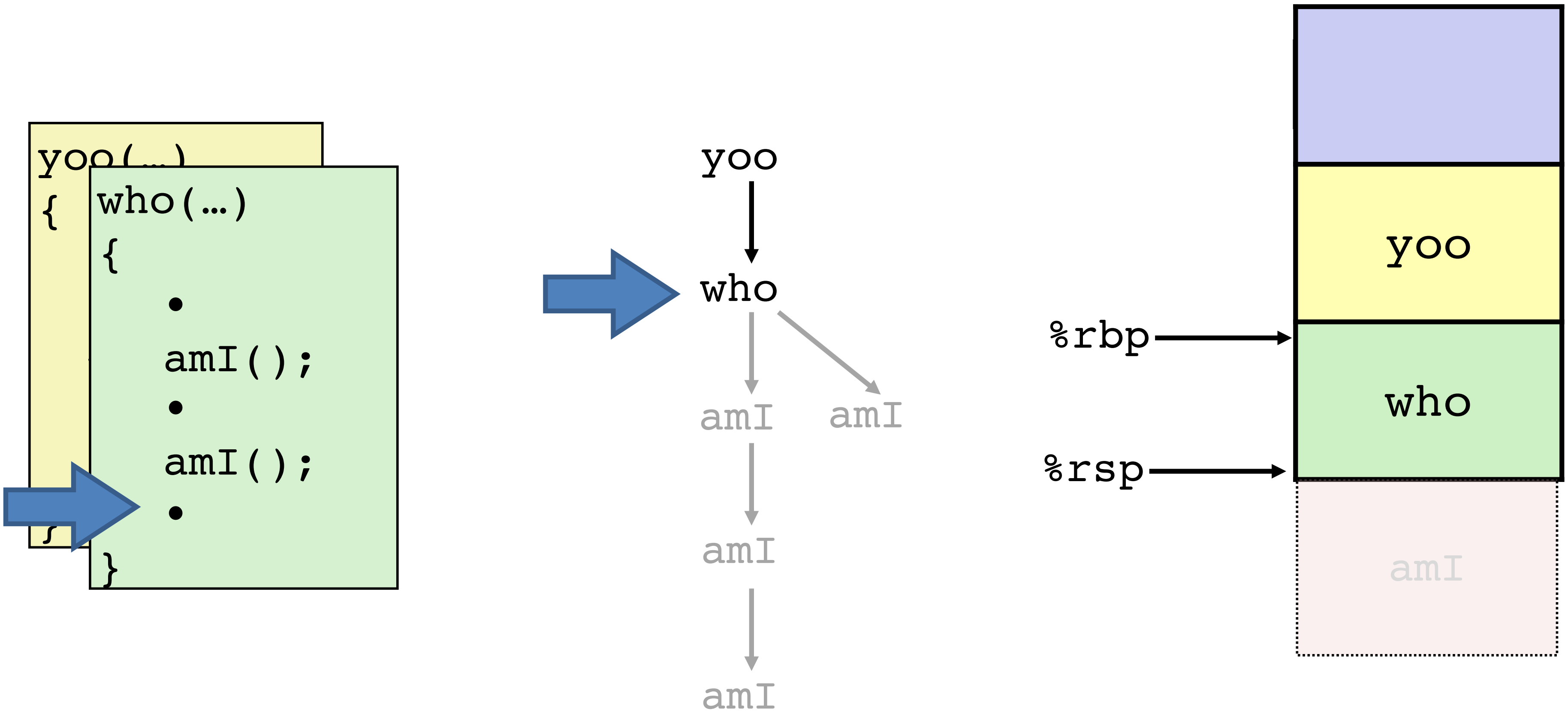
```
amI(…)
{
    •
   if(){
     amI()
   }
    •
}
```



Stack



Call stack tracks context



```
yoo(…)
{
```

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

yoo

who

amI amI

amI

amI

Stack

yoo

who

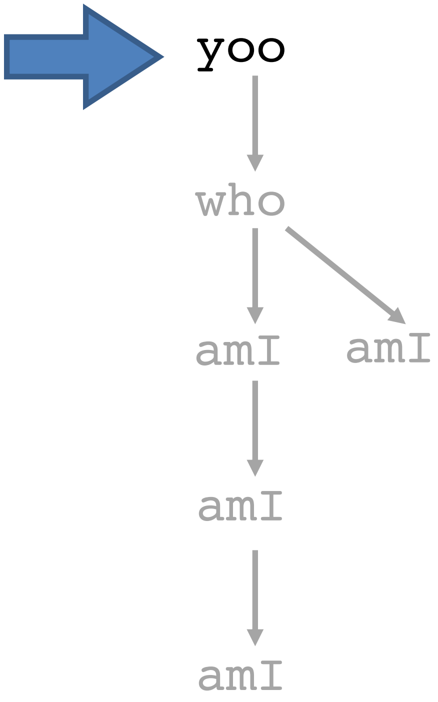
amI

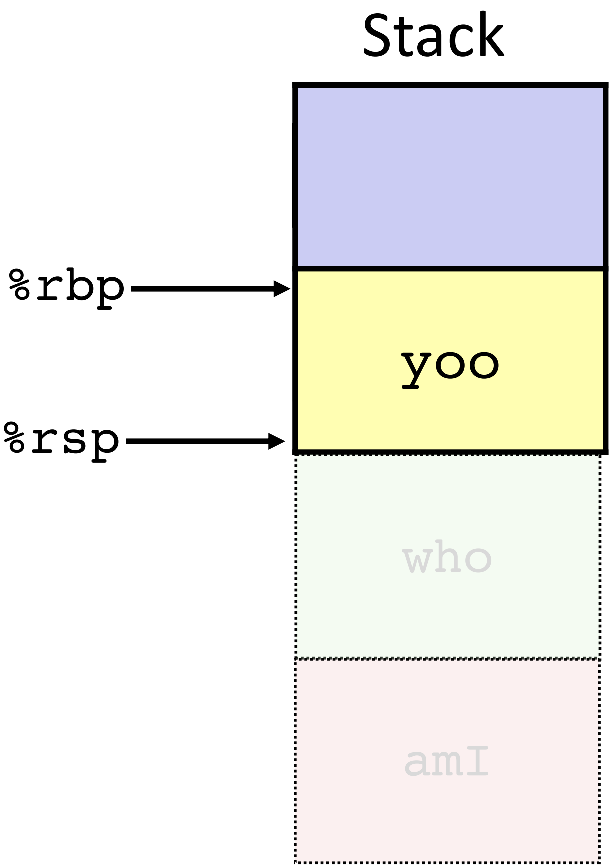
%rbp

%rsp

Call stack tracks context

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



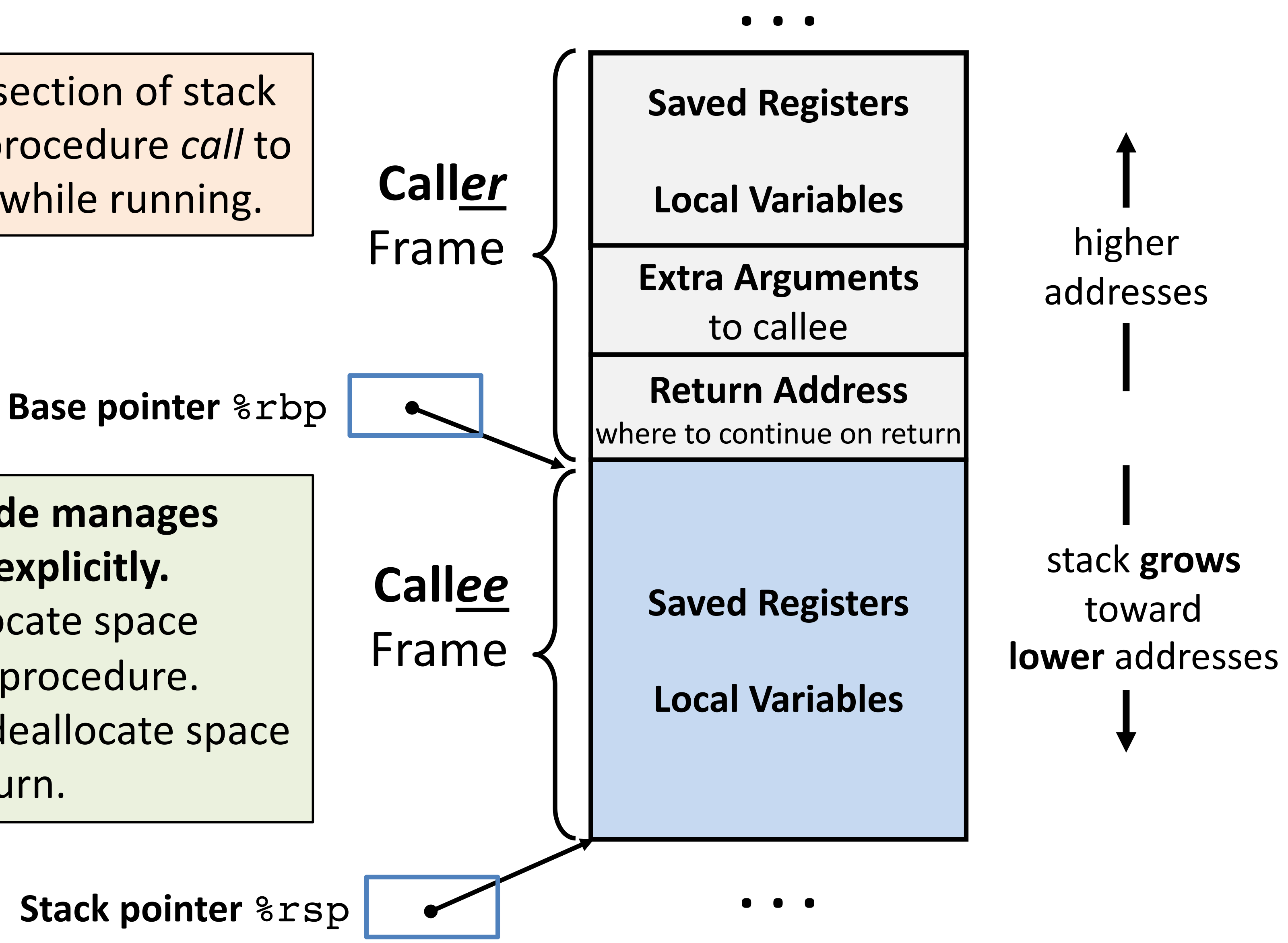


The call stack supports procedures

Stack frame: section of stack used by one procedure *call* to store context while running.

Procedure code manages stack frames explicitly.

- **Setup:** allocate space at start of procedure.
- **Cleanup:** deallocate space before return.



Procedure control flow instructions

Procedure call: `callq` *target*

1. Push return address on stack
2. Jump to *target*

Return address: Address of instruction after `call`.

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

Procedure return: `retq`

1. Pop return address from stack
2. Jump to return address

Call example

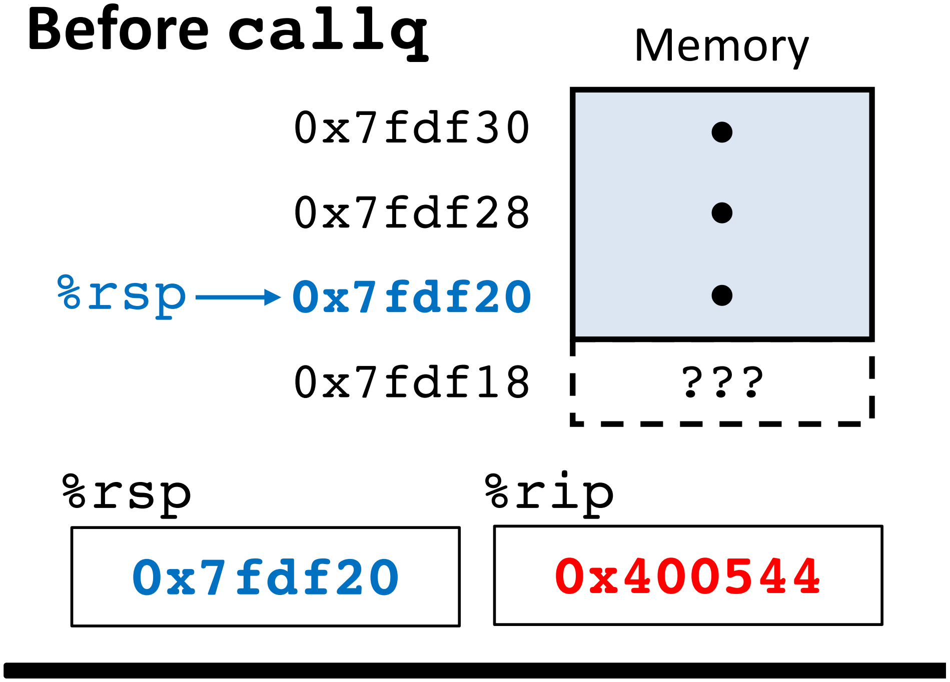
```
0000000000400540 <multstore>:
  •
  •
  400544: callq  400550 <mult2>
  400549: mov    %rax,(%rbx)
  •
```

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

`callq` *target*

1. Push return address on stack
2. Jump to *target*

Before `callq`



Call example

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

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

1. Push return address on stack
2. Jump to label

Before callq

		Memory
	0x7fdf30	•
	0x7fdf28	•
%rsp →	0x7fdf20	•
	0x7fdf18	???

%rsp	%rip
0x7fdf20	0x400544

After callq

		Memory
	0x7fdf30	•
	0x7fdf28	•
	0x7fdf20	•
%rsp →	**0x7fdf18**	**0x400549**

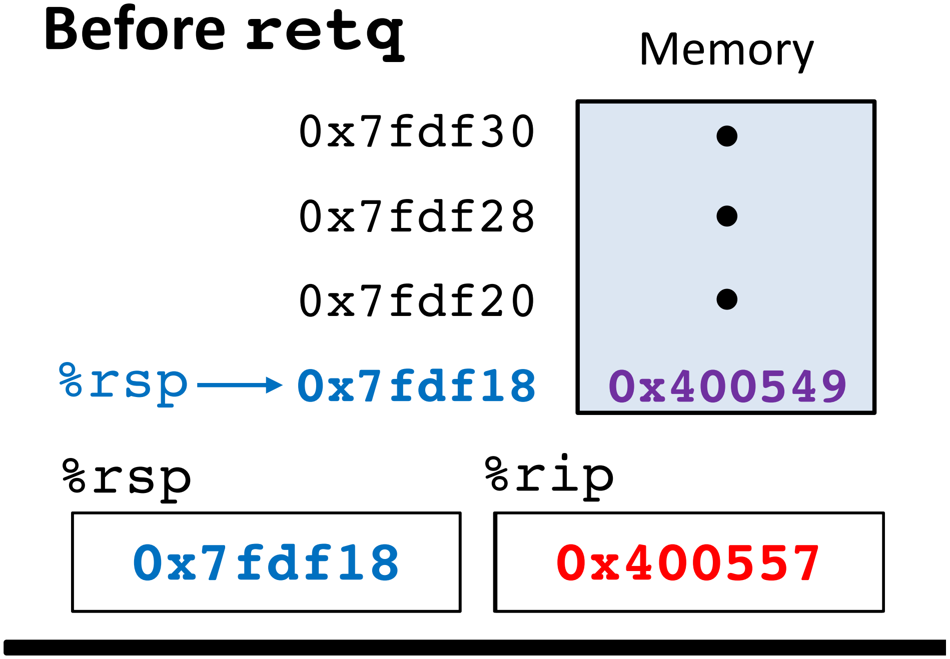
%rsp	%rip
0x7fdf18	**0x400550**

Return example

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

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

Before **`retq`**



`retq`

1. Pop return address from stack
2. Jump to return address

Return example

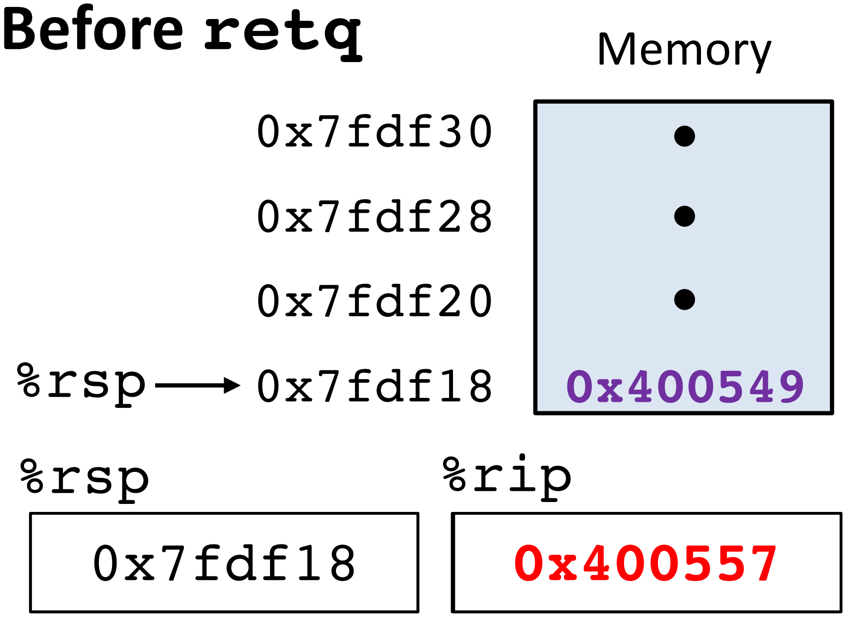
```
0000000000400540 <multstore>:
  •
  •
  400544: callq  400550 <mult2>
  400549: mov    %rax,(%rbx)
  •
```

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

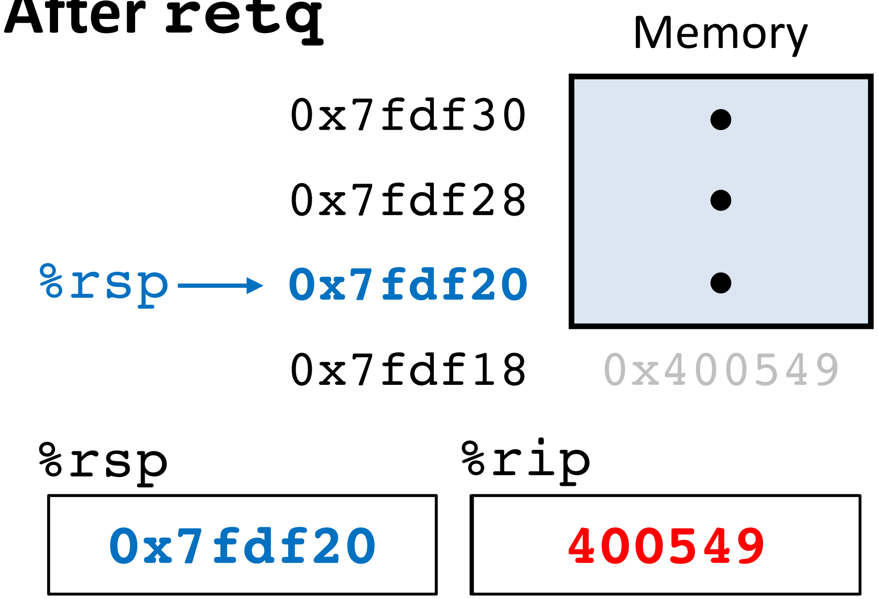
retq

1. Pop return address from stack
2. Jump to return address

Before retq



After retq



`callq` puzzle

optional

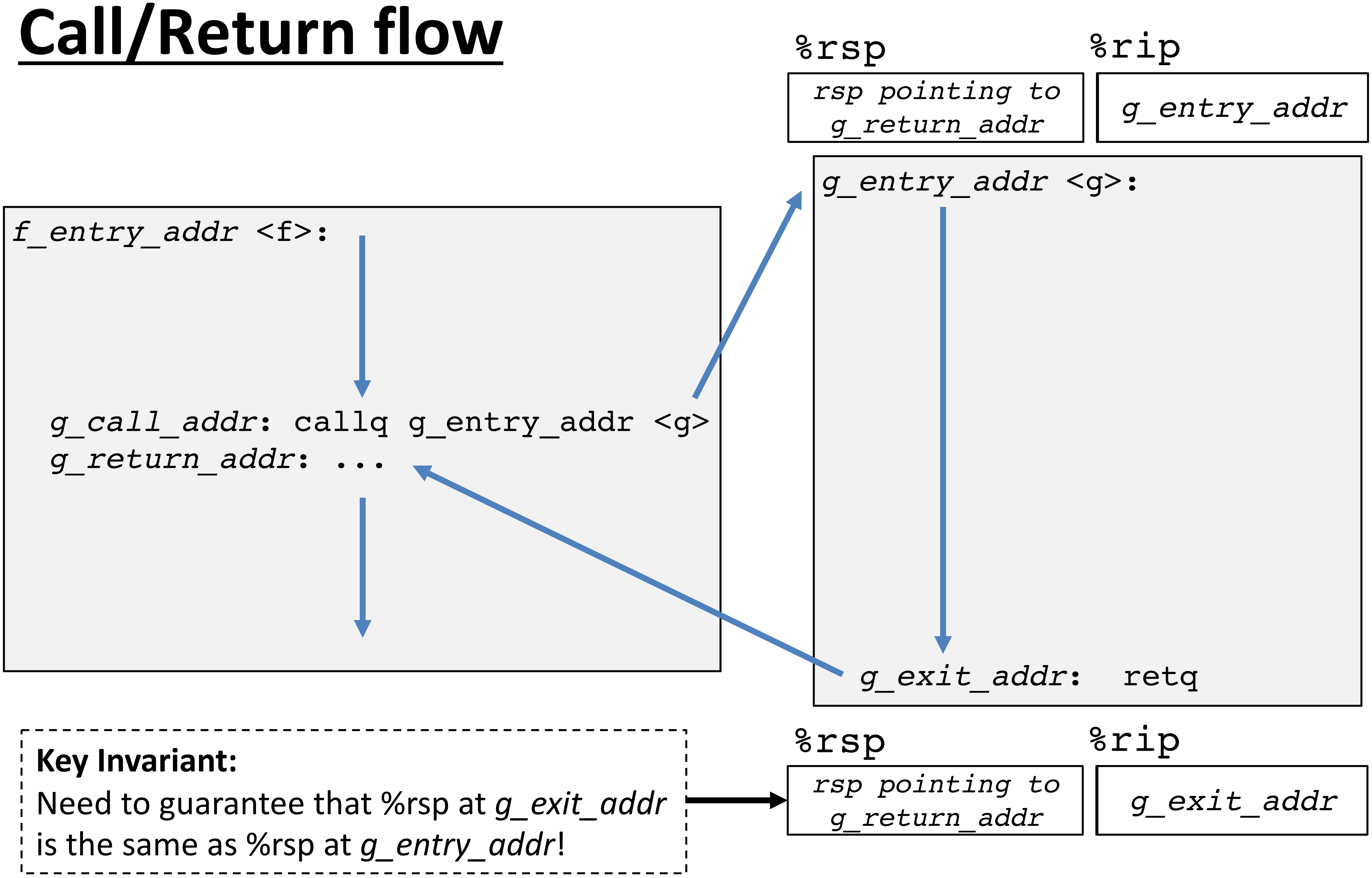
```
   callq next
 next:
   popq %rax
```

What gets stored into %rax?

Why is there no `ret` instruction corresponding to the `call`?

What does this code do? (Hint: unusual use of `call`.)

Call/Return flow



%rsp

rsp pointing to g_return_addr

%rip

g_entry_addr

```
f_entry_addr <f>:

  g_call_addr: callq g_entry_addr <g>
  g_return_addr: ...
```

```
g_entry_addr <g>:

  g_exit_addr:  retq
```

%rsp

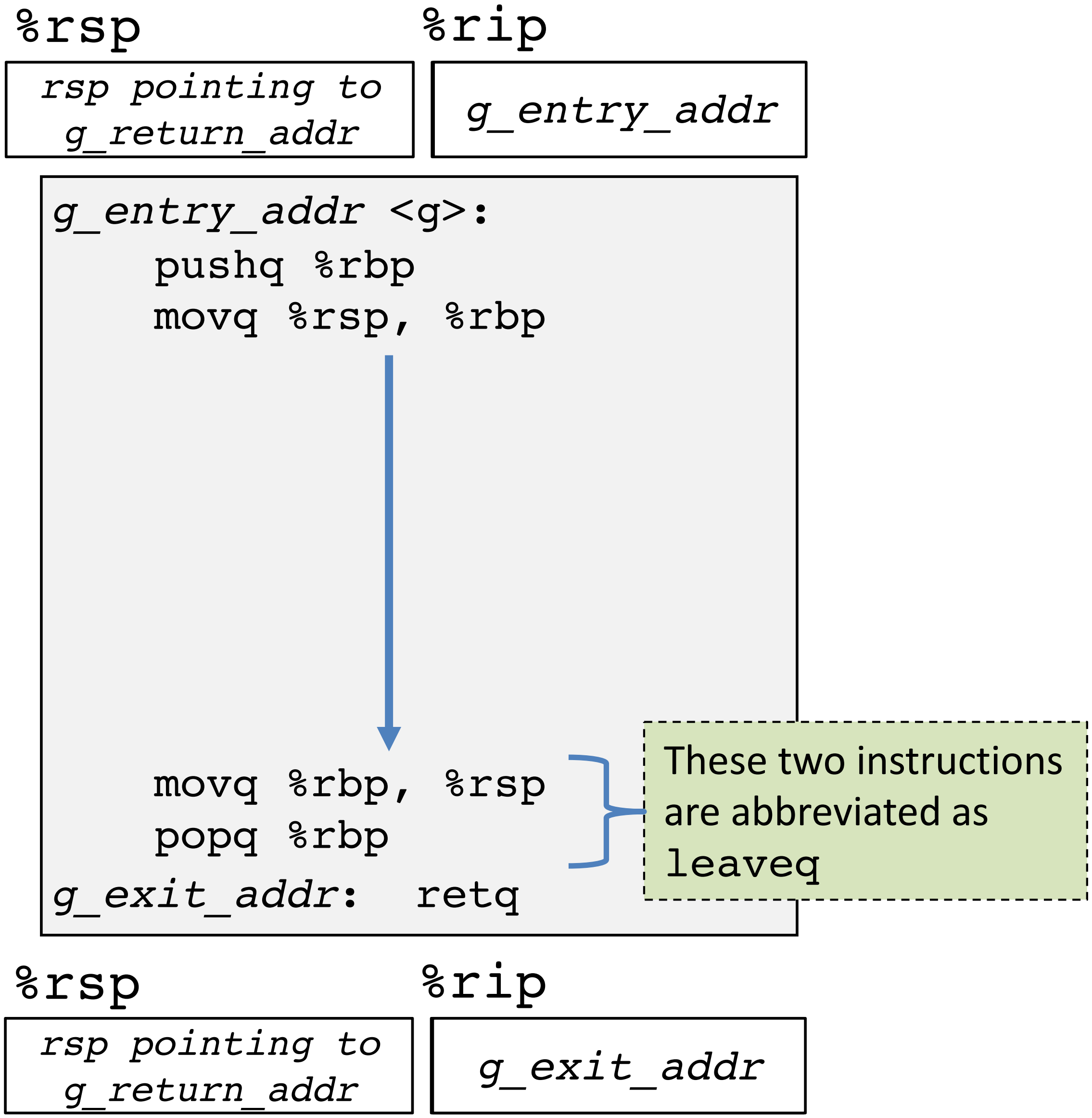
rsp pointing to g_return_addr

%rip

g_exit_addr

Key Invariant:
Need to guarantee that %rsp at *g_exit_addr* is the same as %rsp at *g_entry_addr*!

%rbp prolog/epilog is easy way to guarantee %rsp invariant



%rsp: *rsp pointing to g_return_addr*

%rip: *g_entry_addr*

```
g_entry_addr <g>:
    pushq %rbp
    movq %rsp, %rbp



    movq %rbp, %rsp
    popq %rbp
g_exit_addr:  retq
```

These two instructions are abbreviated as `leaveq`

%rsp: *rsp pointing to g_return_addr*

%rip: *g_exit_addr*

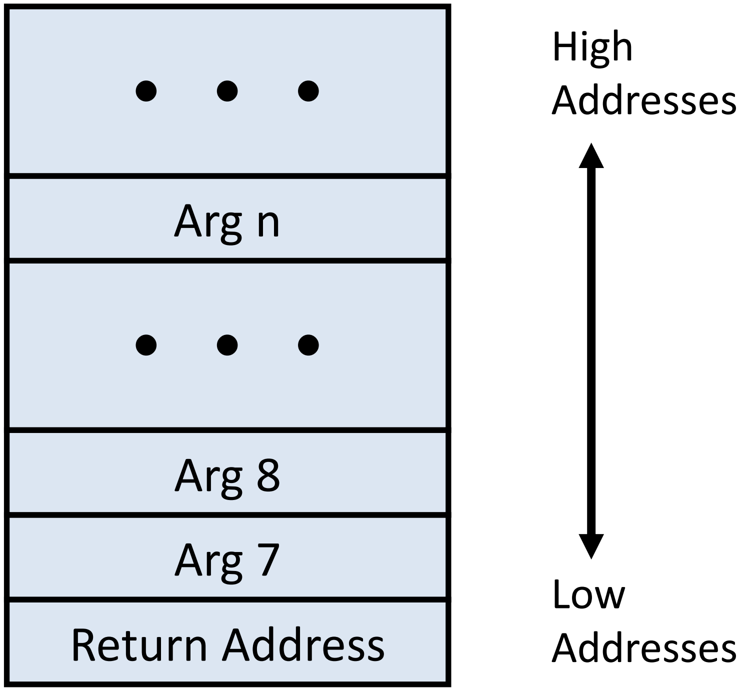
Procedure data flow conventions

First 6 arguments:
passed in **registers**

Arg 1	`%rdi`
Arg 2	`%rsi`
Arg 3	`%rdx`
Arg 4	`%rcx`
Arg 5	`%r8`
Arg 6	`%r9`

Diane's
Silk
Dress
Costs
$8 9

Remaining arguments:
passed on **stack** (in memory)



• • •

Arg n

• • •

Arg 8

Arg 7

Return Address

Return value:
passed in `%rax`

`%rax`

Allocate stack space for arguments only when needed.

Procedure data flow puzzle

ex

C function body:

```
______ huh(______ _, ______ _, ______ _, ______ _) {
    *p = d;
    return x - c;
}
```

Translated to x86 assembly:

```
huh:
  movsbl %dl,   %edx
  movl   %edx,  (%rsi)
  movswl %di,   %edi
  subl   %edi,  %ecx
  movl   %ecx,  %eax
  retq
```

Reverse engineer the x86 `huh` procedure and the body of the C `huh` function to fill blanks in the C `huh` function header with:

- the parameter types / order; and
- the return type.

`movsbl` = **move s**ign-extending a **b**yte to a **l**ong (4-byte)

`movswl` = **move s**ign-extending a **w**ord (2-byte) to a **l**ong (4-byte)

Procedure data flow puzzle

ex

C function body:

```
int    huh(short c, int*  p, char  d, int   x) {
    *p = d;
    return x - c;
}
```

Translated to x86 assembly:

```
huh:
  movsbl %dl,   %edx
  movl   %edx, (%rsi)
  movswl %di,   %edi
  subl   %edi, %ecx
  movl   %ecx, %eax
  retq
```

Reverse engineer the x86 `huh` procedure and the body of the C `huh` function to fill blanks in the C `huh` function header with:

- the parameter types / order; and
- the return type.

`movsbl` = **move s**ign-extending a **b**yte to a **l**ong (4-byte)

`movswl` = **move s**ign-extending a **w**ord (2-byte) to a **l**ong (4-byte)

Procedure call / stack frame example

```
long step_up() {
    long v1 = 240;
    long v2 = increment(&v1, 61);
    return v1+v2;
}
```

Passes address of local variable (in stack).

```
step_up:
400509:  subq   $8, %rsp
40050d:  movq   $240, (%rsp)
400515:  movq   %rsp, %rdi
400518:  movl   $61, %esi
40051d:  callq  4004cd <increment>
400522:  addq   (%rsp), %rax
400526:  addq   $8, %rsp
40052a:  retq
```

Uses memory through pointer.

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

```
increment:
4004cd:  movq   (%rdi), %rax
4004d0:  addq   %rax, %rsi
4004d3:  movq   %rsi, (%rdi)
4004d6:  retq
```

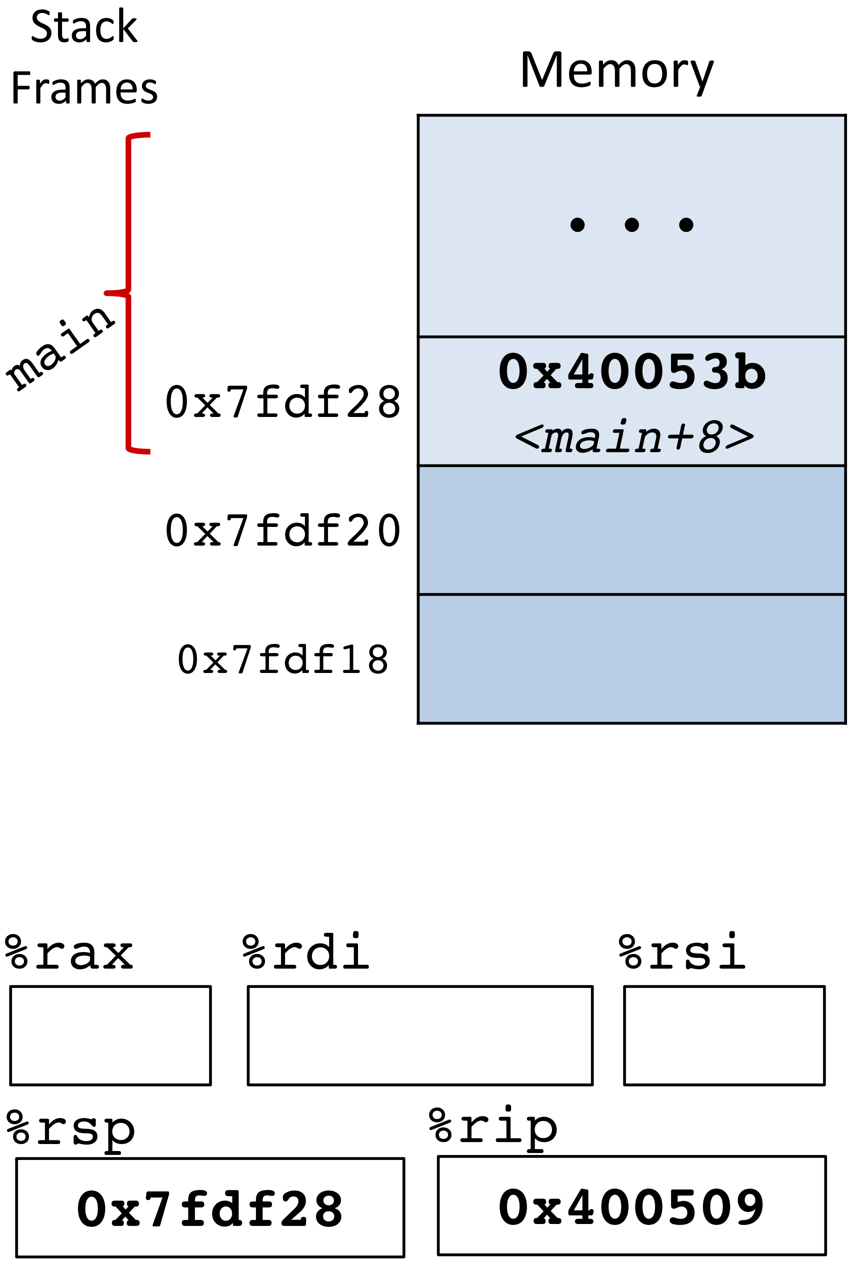
Procedure call example (step 0)

main called step_up

```
long step_up() {
    long v1 = 240;
    long v2 = increment(&v1, 61);
    return v1+v2;
}
```

```
step_up:
400509:  subq   $8, %rsp
40050d:  movq   $240, (%rsp)
400515:  movq   %rsp, %rdi
400518:  movl   $61, %esi
40051d:  callq  4004cd <increment>
400522:  addq   (%rsp), %rax
400526:  addq   $8, %rsp
40052a:  retq
```

```
increment:
4004cd:  movq   (%rdi), %rax
4004d0:  addq   %rax, %rsi
4004d3:  movq   %rsi, (%rdi)
4004d6:  retq
```



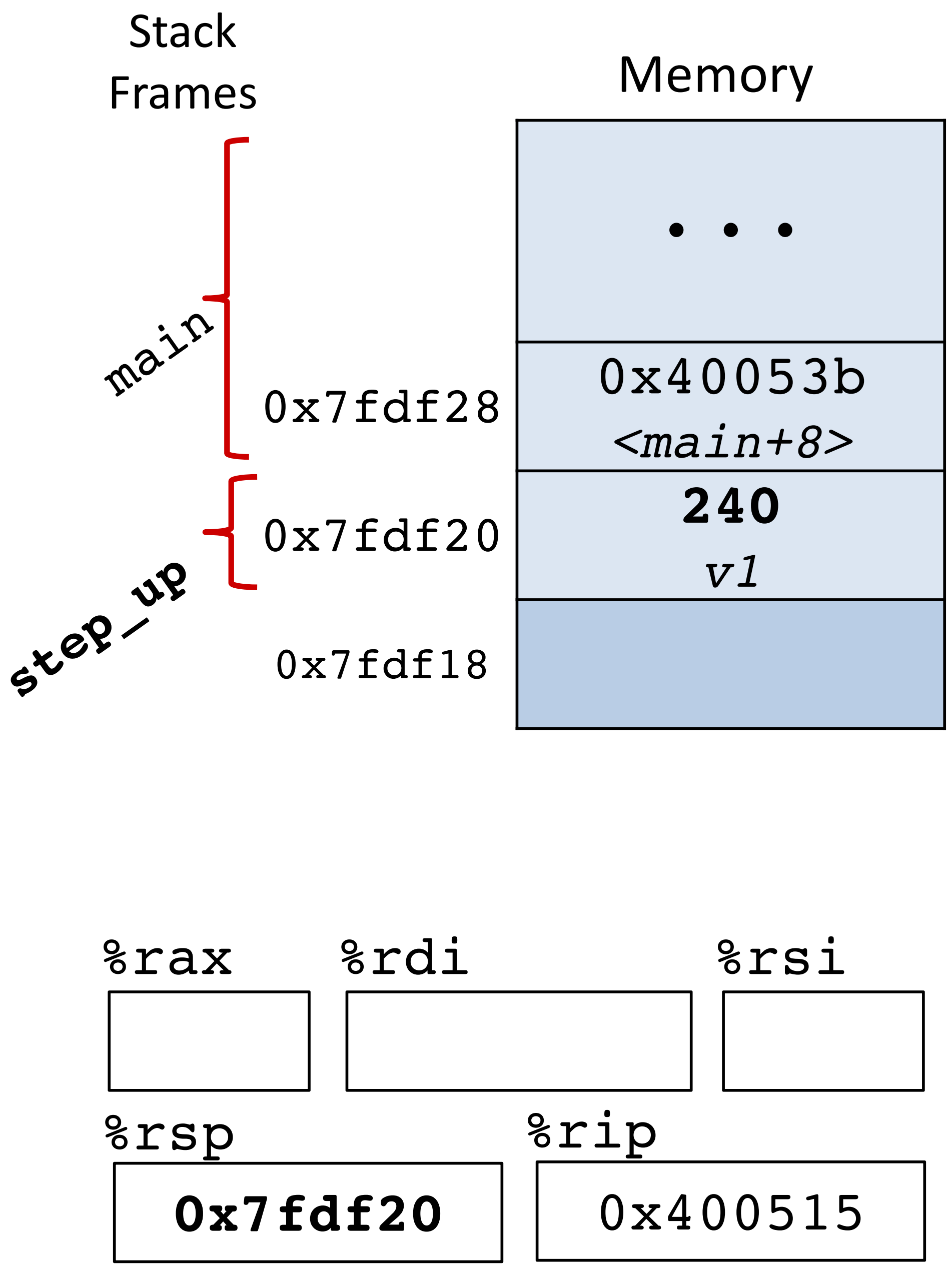
Procedure call example (step 1)

Allocate space for local vars

```
long step_up() {
    long v1 = 240;
    long v2 = increment(&v1, 61);
    return v1+v2;
}
```

```
step_up:
400509:  subq   $8, %rsp
40050d:  movq   $240, (%rsp)
400515:  movq   %rsp, %rdi
400518:  movl   $61, %esi
40051d:  callq  4004cd <increment>
400522:  addq   (%rsp), %rax
400526:  addq   $8, %rsp
40052a:  retq
```

```
increment:
4004cd:  movq   (%rdi), %rax
4004d0:  addq   %rax, %rsi
4004d3:  movq   %rsi, (%rdi)
4004d6:  retq
```



%rsp: 0x7fdf20

%rip: 0x400515

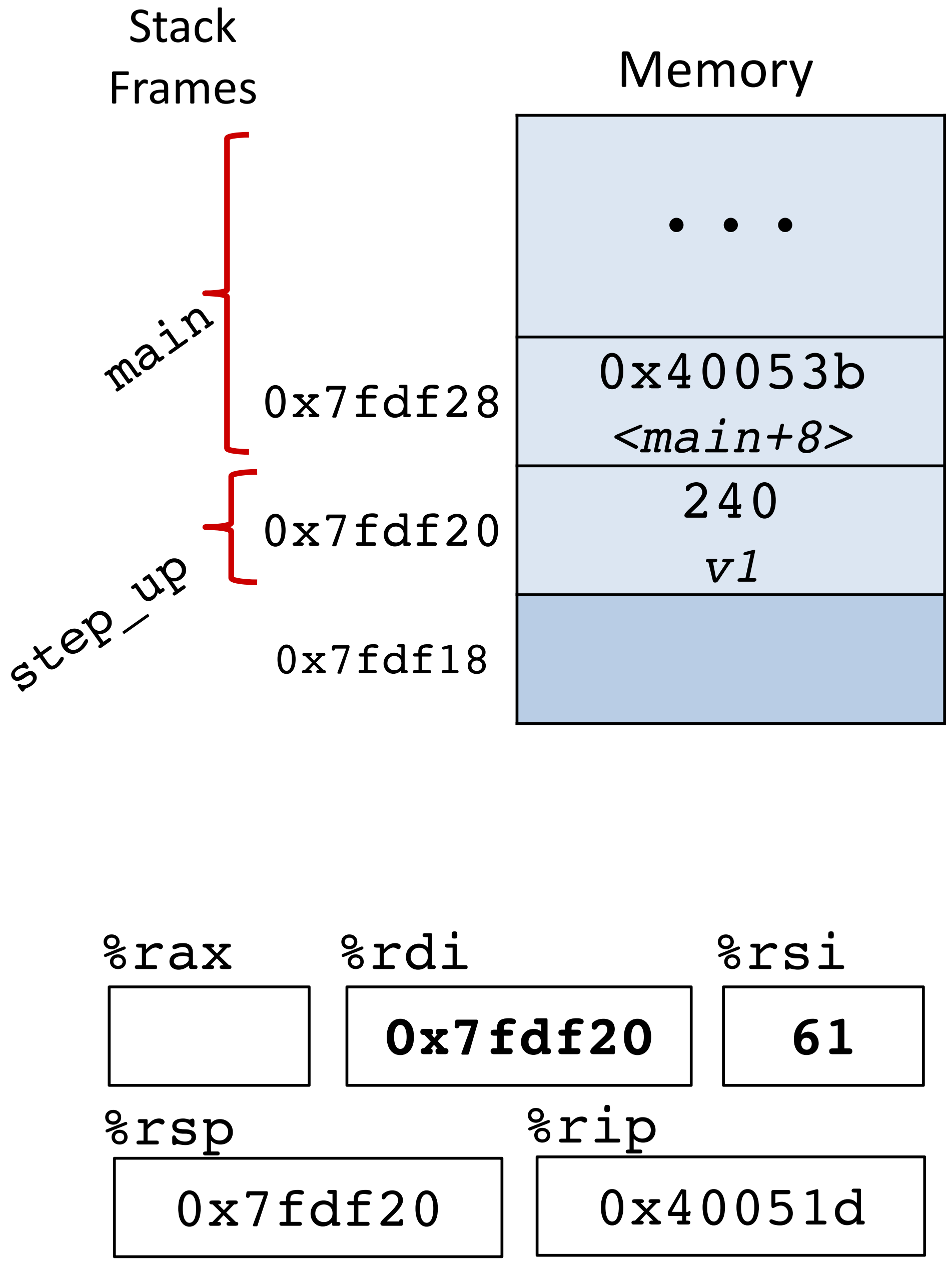
Procedure call example (step 2)

Set up args for call to `increment`

```
long step_up() {
    long v1 = 240;
    long v2 = increment(&v1, 61);
    return v1+v2;
}
```

```
step_up:
400509:  subq   $8, %rsp
40050d:  movq   $240, (%rsp)
400515:  movq   %rsp, %rdi
400518:  movl   $61, %esi
40051d:  callq  4004cd <increment>
400522:  addq   (%rsp), %rax
400526:  addq   $8, %rsp
40052a:  retq
```

```
increment:
4004cd:  movq   (%rdi), %rax
4004d0:  addq   %rax, %rsi
4004d3:  movq   %rsi, (%rdi)
4004d6:  retq
```



Stack Frames

Memory

main

step_up

. . .

0x7fdf28 0x40053b <main+8>

0x7fdf20 240 *v1*

0x7fdf18

%rax

%rdi 0x7fdf20

%rsi 61

%rsp 0x7fdf20

%rip 0x40051d

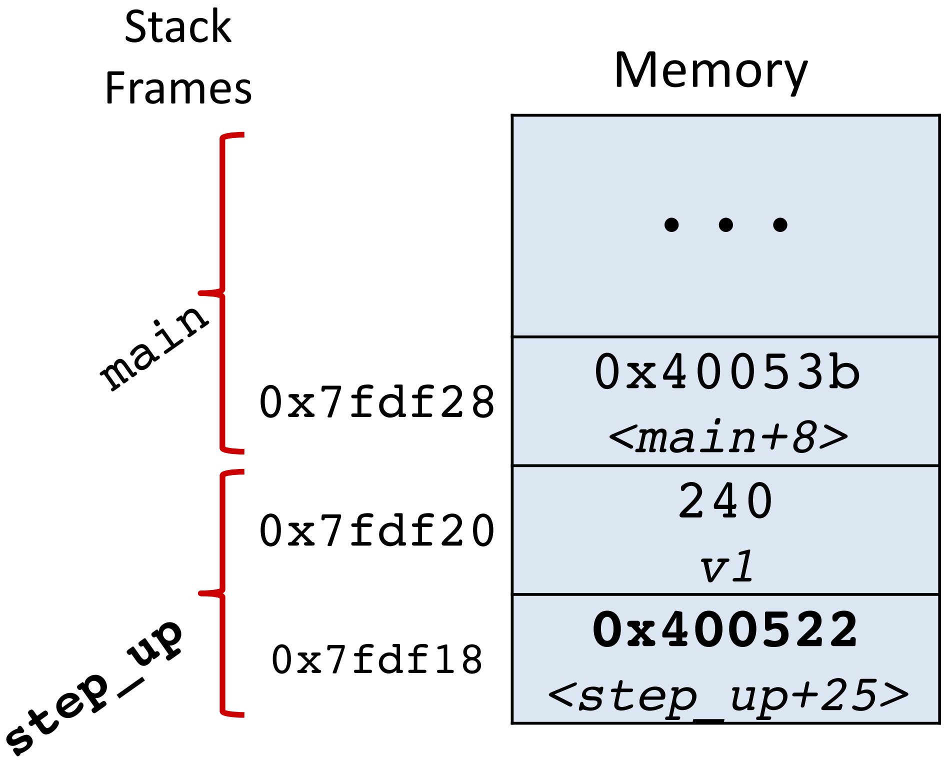
Procedure call example (step 3)

Call `increment`

```
long step_up() {
    long v1 = 240;
    long v2 = increment(&v1, 61);
    return v1+v2;
}
```

```
step_up:
400509:  subq   $8, %rsp
40050d:  movq   $240, (%rsp)
400515:  movq   %rsp, %rdi
400518:  movl   $61, %esi
40051d:  callq  4004cd <increment>
400522:  addq   (%rsp), %rax
400526:  addq   $8, %rsp
40052a:  retq
```

```
increment:
4004cd:  movq   (%rdi), %rax
4004d0:  addq   %rax, %rsi
4004d3:  movq   %rsi, (%rdi)
4004d6:  retq
```



Stack Frames

Memory

Stack Frames	Address	Memory
		. . .
main	0x7fdf28	0x40053b *<main+8>*
step_up	0x7fdf20	240 *v1*
	0x7fdf18	**0x400522** *<step_up+25>*

%rax	%rdi	%rsi
	0x7fdf20	61

%rsp	%rip
0x7fdf18	**0x4004cd**

Procedure call example (step 4)

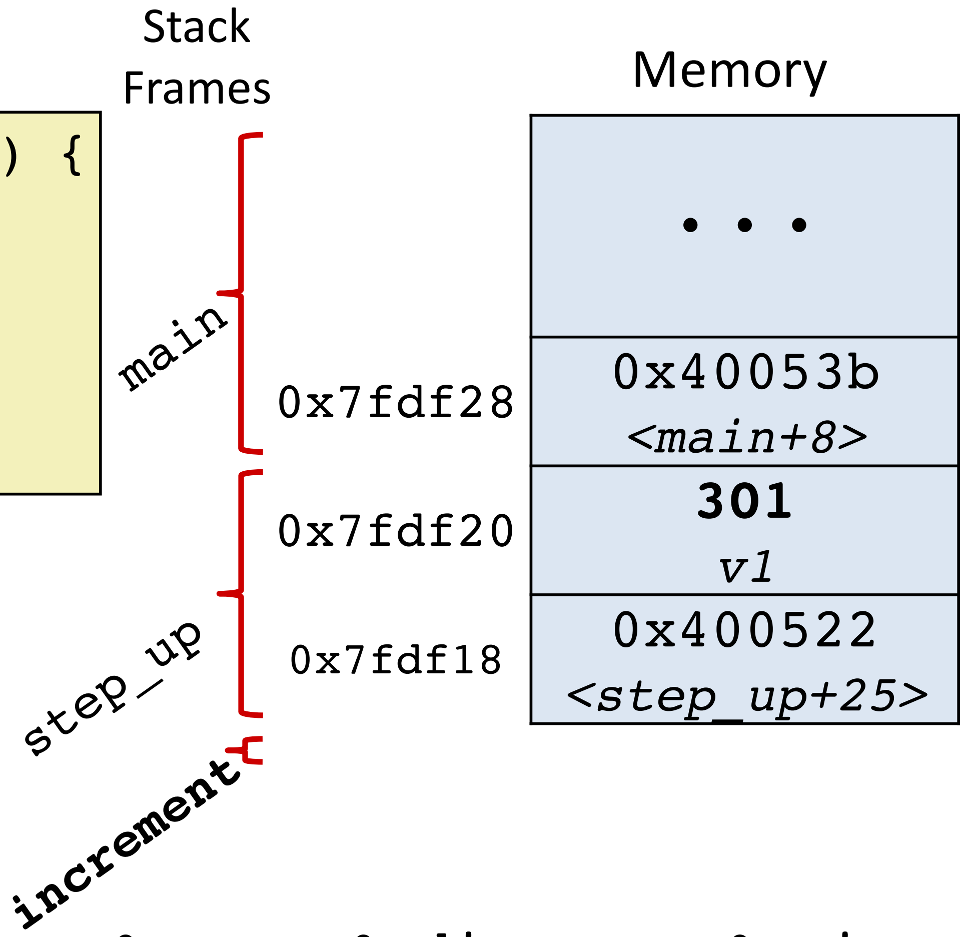
Run `increment`

```
long step_up() {
}
```

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

```
st
400509:  subq   $8, %rsp
40050d:  movq   $240, (%rsp)
400515:  movq   %rsp, %rdi
400518:  movl   $61, %esi
40051d:  callq  4004cd <increment>
400522:  addq   (%rsp), %rax
400526:  addq   $8, %rsp
40052a:  retq
```

```
increment:
4004cd:  movq   (%rdi), %rax
4004d0:  addq   %rax, %rsi
4004d3:  movq   %rsi, (%rdi)
4004d6:  retq
```



%rax	%rdi	%rsi
240	0x7fdf20	301

%rsp	%rip
0x7fdf18	0x4004d6

Procedure call example (step 5a)

Return from `increment` to `step_up`

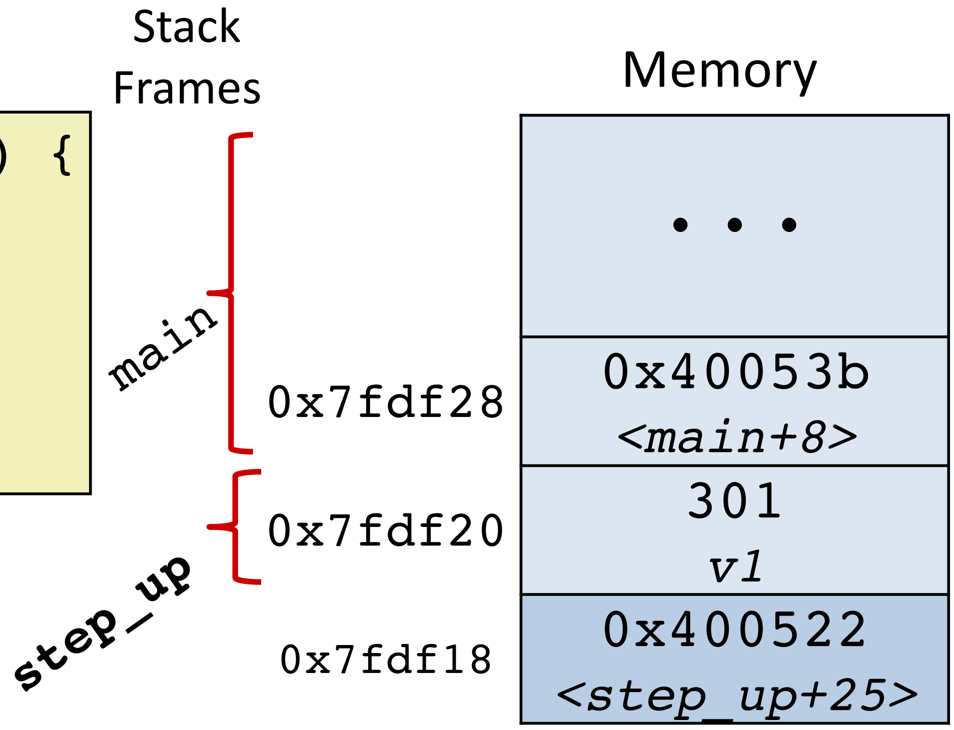
```
long step_up() {

}
```

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

```
st
400509:  subq   $8, %rsp
40050d:  movq   $240, (%rsp)
400515:  movq   %rsp, %rdi
400518:  movl   $61, %esi
40051d:  callq  4004cd <increment>
400522:  addq   (%rsp), %rax
400526:  addq   $8, %rsp
40052a:  retq
```

```
increment:
4004cd:  movq   (%rdi), %rax
4004d0:  addq   %rax, %rsi
4004d3:  movq   %rsi, (%rdi)
4004d6:  retq
```



%rax: 240

%rdi: 0x7fdf20

%rsi: 301

%rsp: **0x7fdf20**

%rip: **0x400522**

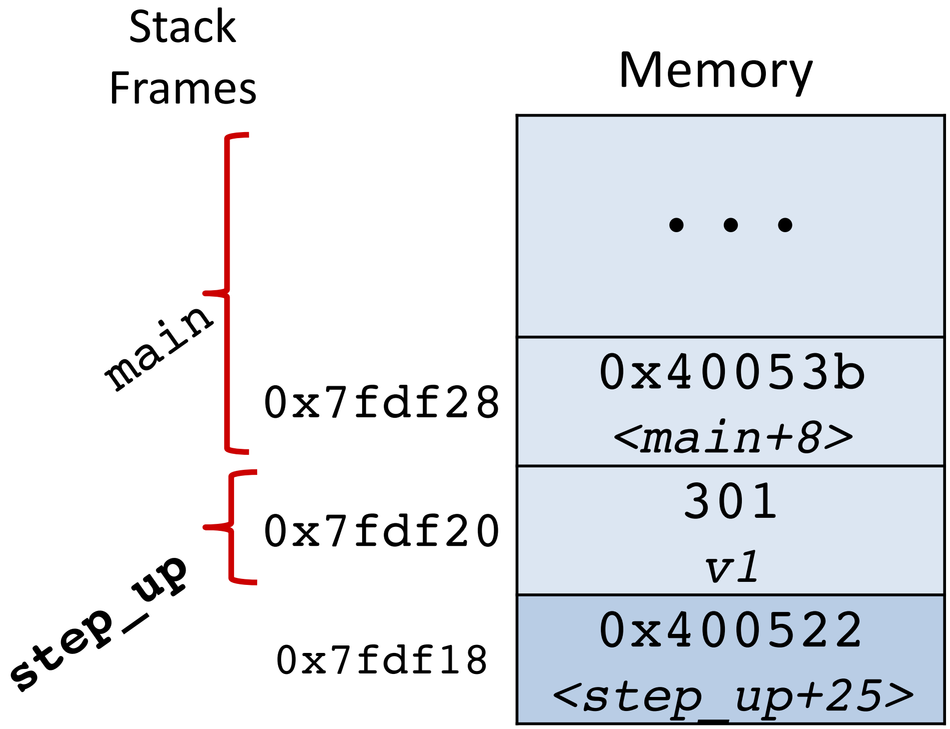
Procedure call example (step 5b)

Return from `increment` to `step_up`

```
long step_up() {
    long v1 = 240;
    long v2 = increment(&v1, 61);
    return v1+v2;
}
```

```
step_up:
400509:  subq   $8, %rsp
40050d:  movq   $240, (%rsp)
400515:  movq   %rsp, %rdi
400518:  movl   $61, %esi
40051d:  callq  4004cd <increment>
400522:  addq   (%rsp), %rax
400526:  addq   $8, %rsp
40052a:  retq
```

```
increment:
4004cd:  movq   (%rdi), %rax
4004d0:  addq   %rax, %rsi
4004d3:  movq   %rsi, (%rdi)
4004d6:  retq
```



Stack Frames

Memory

Stack Frames	Address	Memory
main		. . .
main	0x7fdf28	0x40053b *<main+8>*
step_up	0x7fdf20	301 *v1*
	0x7fdf18	0x400522 *<step_up+25>*

%rax	%rdi	%rsi
240	0x7fdf20	301

%rsp	%rip
0x7fdf20	**0x400522**

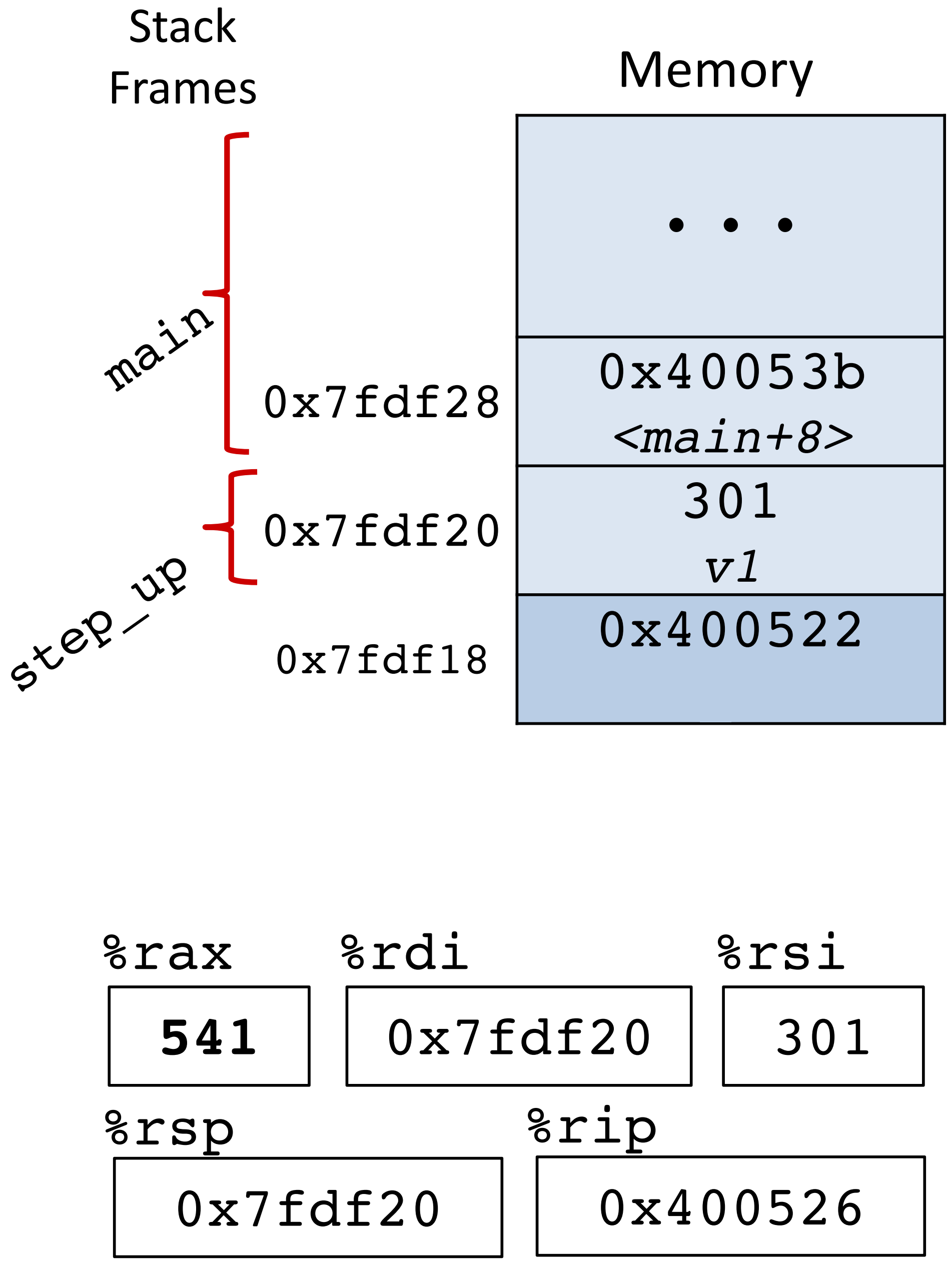
Procedure call example (step 6)

Prepare step_up result

```
long step_up() {
    long v1 = 240;
    long v2 = increment(&v1, 61);
    return v1+v2;
}
```

```
step_up:
400509:  subq   $8, %rsp
40050d:  movq   $240, (%rsp)
400515:  movq   %rsp, %rdi
400518:  movl   $61, %esi
40051d:  callq  4004cd <increment>
400522:  addq   (%rsp), %rax
400526:  addq   $8, %rsp
40052a:  retq
```

```
increment:
4004cd:  movq   (%rdi), %rax
4004d0:  addq   %rax, %rsi
4004d3:  movq   %rsi, (%rdi)
4004d6:  retq
```



Stack Frames

Memory

main

step_up

. . .

0x7fdf28 | 0x40053b <*main+8*>

0x7fdf20 | 301 *v1*

0x7fdf18 | 0x400522

%rax 541

%rdi 0x7fdf20

%rsi 301

%rsp 0x7fdf20

%rip 0x400526

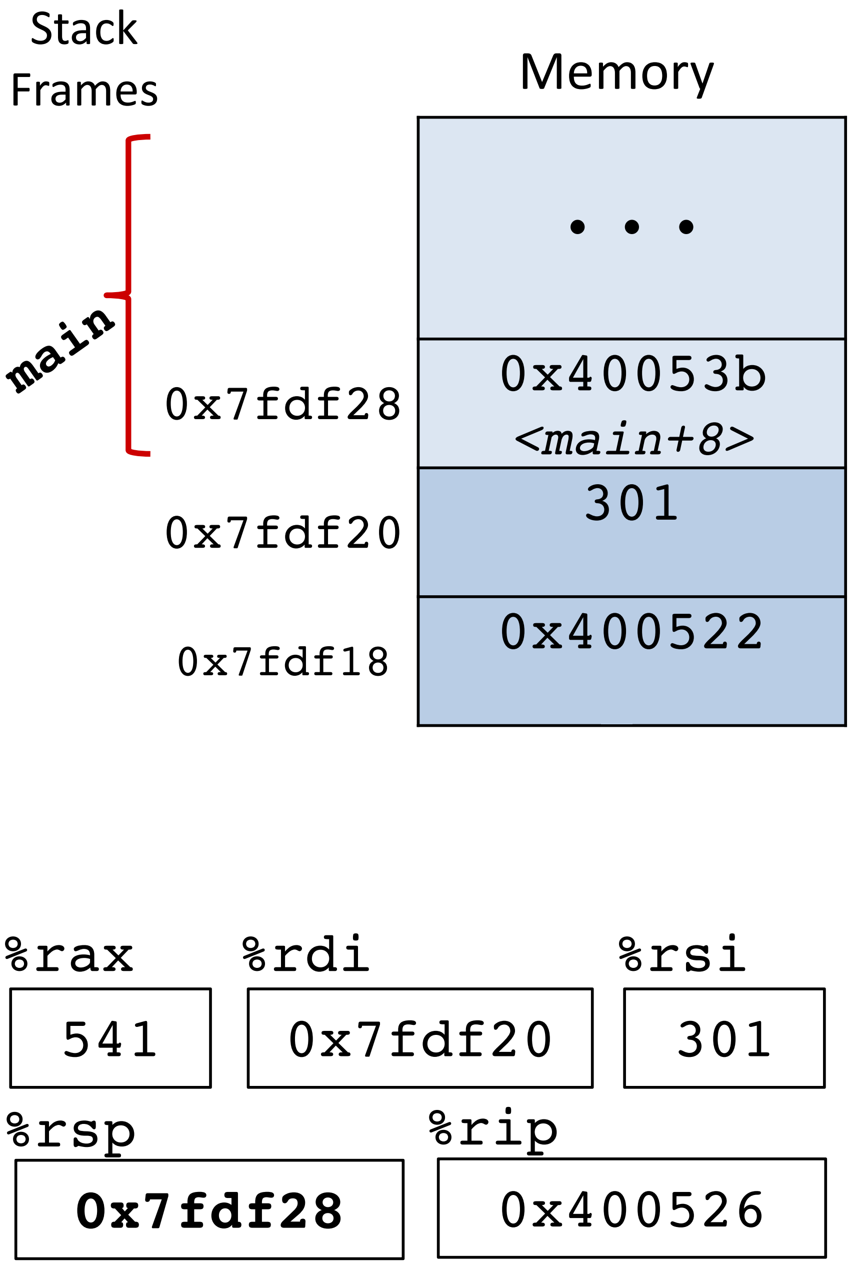
Procedure call example (step 7)

Deallocate space for local vars

```
long step_up() {
    long v1 = 240;
    long v2 = increment(&v1, 61);
    return v1+v2;
}
```

```
step_up:
400509:  subq  $8, %rsp
40050d:  movq  $240, (%rsp)
400515:  movq  %rsp, %rdi
400518:  movl  $61, %esi
40051d:  callq 4004cd <increment>
400522:  addq  (%rsp), %rax
400526:  **addq  $8, %rsp**
40052a:  retq
```

```
increment:
4004cd:  movq  (%rdi), %rax
4004d0:  addq  %rax, %rsi
4004d3:  movq  %rsi, (%rdi)
4004d6:  retq
```



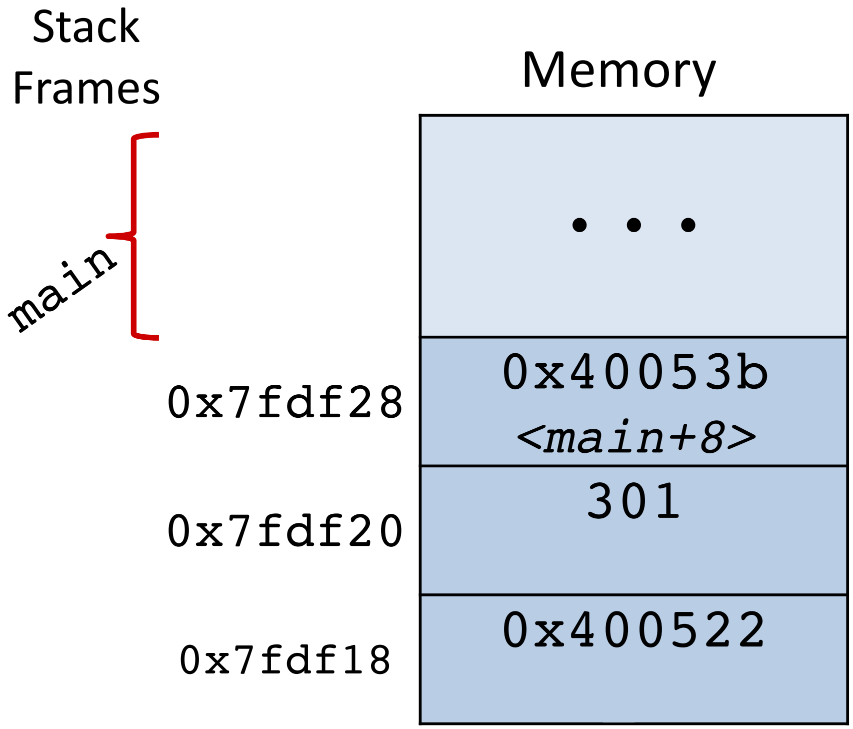
Procedure call example (step 8)

Return from step_up to main

```
long step_up() {
    long v1 = 240;
    long v2 = increment(&v1, 61);
    return v1+v2;
}
```

```
step_up:
400509:  subq  $8, %rsp
40050d:  movq  $240, (%rsp)
400515:  movq  %rsp, %rdi
400518:  movl  $61, %esi
40051d:  callq 4004cd <increment>
400522:  addq  (%rsp), %rax
400526:  addq  $8, %rsp
40052a:  retq
```

```
increment:
4004cd:  movq  (%rdi), %rax
4004d0:  addq  %rax, %rsi
4004d3:  movq  %rsi, (%rdi)
4004d6:  retq
```



%rax	%rdi	%rsi
541	0x7fdf20	301

%rsp	%rip
0x7fdf30	**0x40053b**

Implementing procedures

1. How does a caller pass arguments to a procedure? ✓

2. How does a caller receive a return value from a procedure? ✓

3. How does a procedure know where to return
   (what code to execute next when done)? ✓

4. Where does a procedure store local variables? ✓

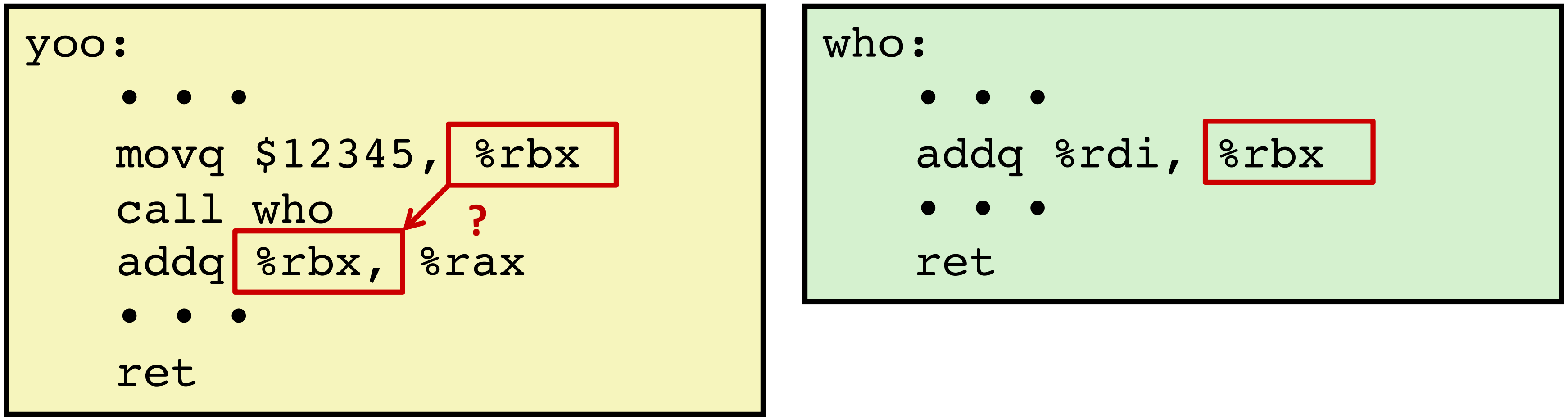
1. How do procedures share limited registers and memory? ??

Register saving conventions

`yoo` calls `who`:

Caller ***Callee***

Will register contents still be there after a procedure call?



```
yoo:
    • • •
    movq $12345, %rbx
    call who
    addq %rbx, %rax
    • • •
    ret
```

```
who:
    • • •
    addq %rdi, %rbx
    • • •
    ret
```

Conventions:

- ***Caller Save***
- ***Callee Save***

x86-64 register conventions

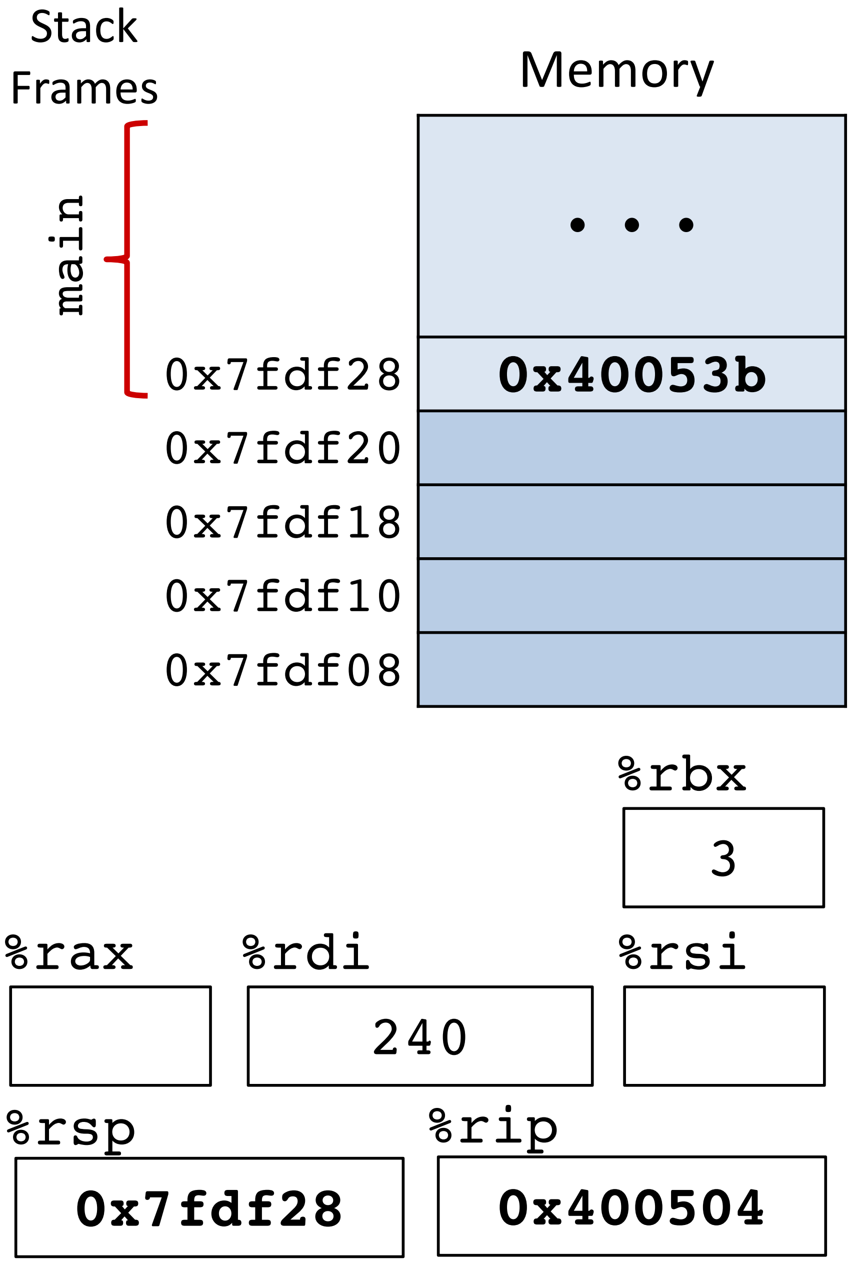
`%rax`	Return value – Caller saved	`%r8`	Argument #5 – Caller saved
`%rbx`	**Callee** saved	`%r9`	Argument #6 – Caller saved
`%rcx`	Argument #4 – Caller saved	`%r10`	Caller saved
`%rdx`	Argument #3 – Caller saved	`%r11`	Caller Saved
`%rsi`	Argument #2 – Caller saved	`%r12`	**Callee** saved
`%rdi`	Argument #1 – Caller saved	`%r13`	**Callee** saved
`%rsp`	Stack pointer	`%r14`	**Callee** saved
`%rbp`	**Callee** saved	`%r15`	**Callee** saved

Callee-save example (step 0)

main called step_by(240)

```
long step_by(long x) {
    long v1 = x;
    long v2 = increment(&v1, 61);
    return x + v2;
}
```

```
step_by:
400504:  pushq %rbx
400506:  movq  %rdi, %rbx
400509:  subq  $16, %rsp
40050d:  movq  %rdi, (%rsp)
400515:  movq  %rsp, %rdi
400518:  movl  $61, %esi
40051d:  callq 4004cd <increment>
400522:  addq  %rbx, %rax
400525:  addq  $16, %rsp
400529:  popq  %rbx
40052b:  retq
```



Stack Frames

Memory

main

. . .

0x7fdf28	0x40053b
0x7fdf20	
0x7fdf18	
0x7fdf10	
0x7fdf08	

%rbx: 3

%rax:

%rdi: 240

%rsi:

%rsp: 0x7fdf28

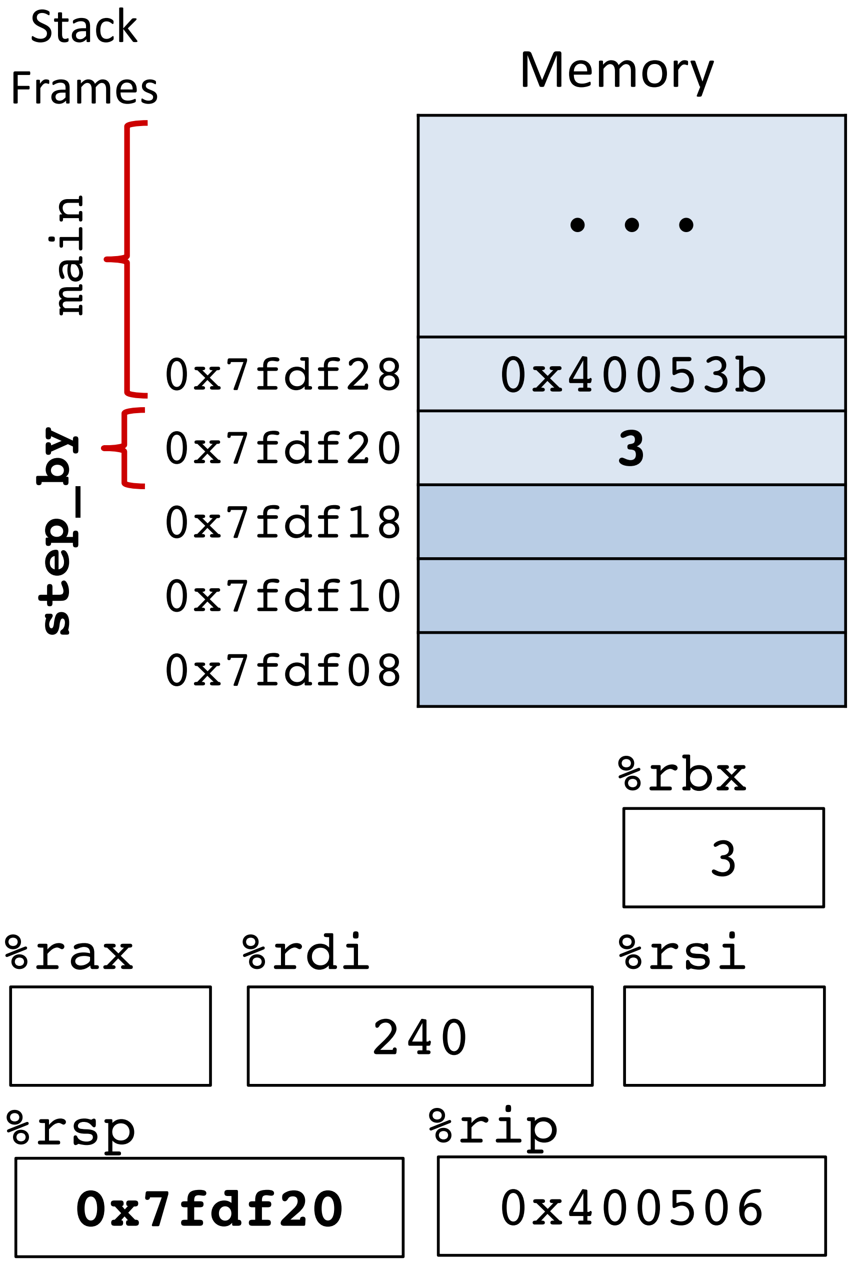
%rip: 0x400504

Callee-save example (step 1)

Save register %rbx

```
long step_by(long x) {
    long v1 = x;
    long v2 = increment(&v1, 61);
    return x + v2;
}
```

```
step_by:
400504:   pushq  %rbx
400506:   movq   %rdi, %rbx
400509:   subq   $16, %rsp
40050d:   movq   %rdi, (%rsp)
400515:   movq   %rsp, %rdi
400518:   movl   $61, %esi
40051d:   callq  4004cd <increment>
400522:   addq   %rbx, %rax
400525:   addq   $16, %rsp
400529:   popq   %rbx
40052b:   retq
```



Stack Frames

Memory

main

step_by

	. . .
0x7fdf28	0x40053b
0x7fdf20	3
0x7fdf18	
0x7fdf10	
0x7fdf08	

%rbx: 3

%rax:

%rdi: 240

%rsi:

%rsp: 0x7fdf20

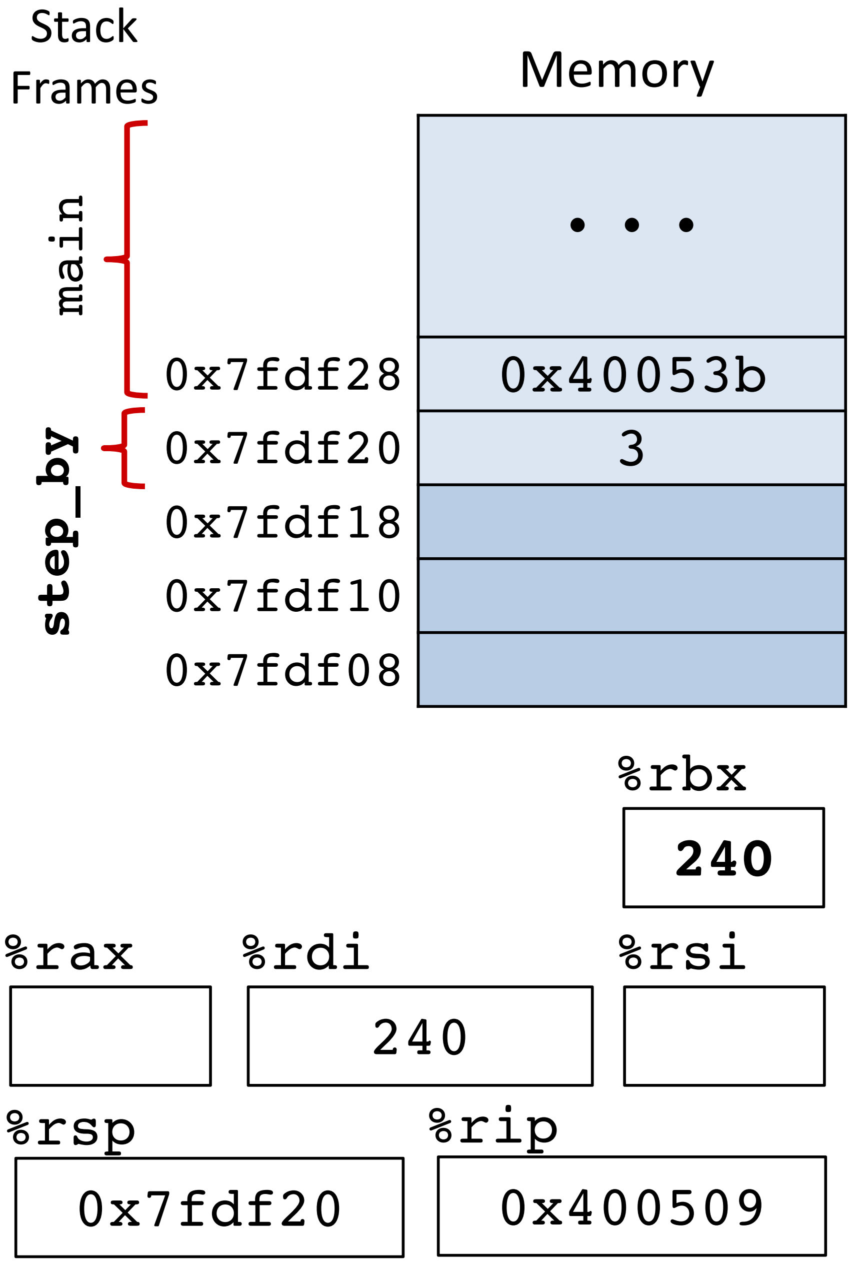
%rip: 0x400506

Callee-save example (step 2)

Copy argument x to %rbx for continued use after calling increment.

```
long step_by(long x) {
    long v1 = x;
    long v2 = increment(&v1, 61);
    return x + v2;
}
```

```
step_by:
400504:  pushq %rbx
400506:  movq  %rdi, %rbx
400509:  subq  $16, %rsp
40050d:  movq  %rdi, (%rsp)
400515:  movq  %rsp, %rdi
400518:  movl  $61, %esi
40051d:  callq 4004cd <increment>
400522:  addq  %rbx, %rax
400525:  addq  $16, %rsp
400529:  popq  %rbx
40052b:  retq
```



Stack Frames

Memory

Stack Frames	Address	Memory
main		. . .
	0x7fdf28	0x40053b
step_by	0x7fdf20	3
	0x7fdf18	
	0x7fdf10	
	0x7fdf08	

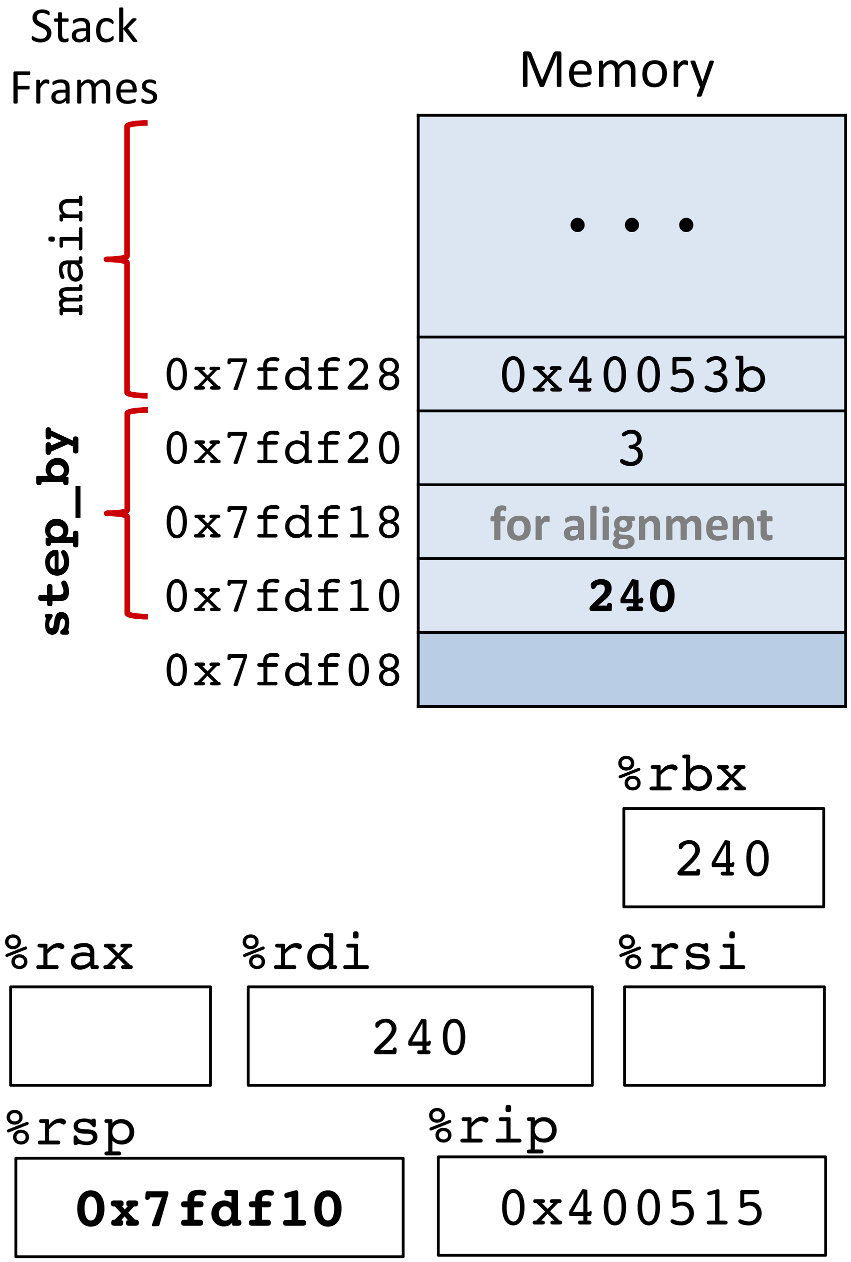
Register	Value
%rbx	240
%rax	
%rdi	240
%rsi	
%rsp	0x7fdf20
%rip	0x400509

Callee-save example (step 3)

Set up stack frame
Initialize v1

```
long step_by(long x) {
    long v1 = x;
    long v2 = increment(&v1, 61);
    return x + v2;
}
```

```
step_by:
400504:  pushq  %rbx
400506:  movq   %rdi, %rbx
400509:  subq   $16, %rsp
40050d:  movq   %rdi, (%rsp)
400515:  movq   %rsp, %rdi
400518:  movl   $61, %esi
40051d:  callq  4004cd <increment>
400522:  addq   %rbx, %rax
400525:  addq   $16, %rsp
400529:  popq   %rbx
40052b:  retq
```



Stack Frames

Memory

Stack Frames	Address	Memory
main		. . .
	0x7fdf28	0x40053b
step_by	0x7fdf20	3
	0x7fdf18	for alignment
	0x7fdf10	**240**
	0x7fdf08	

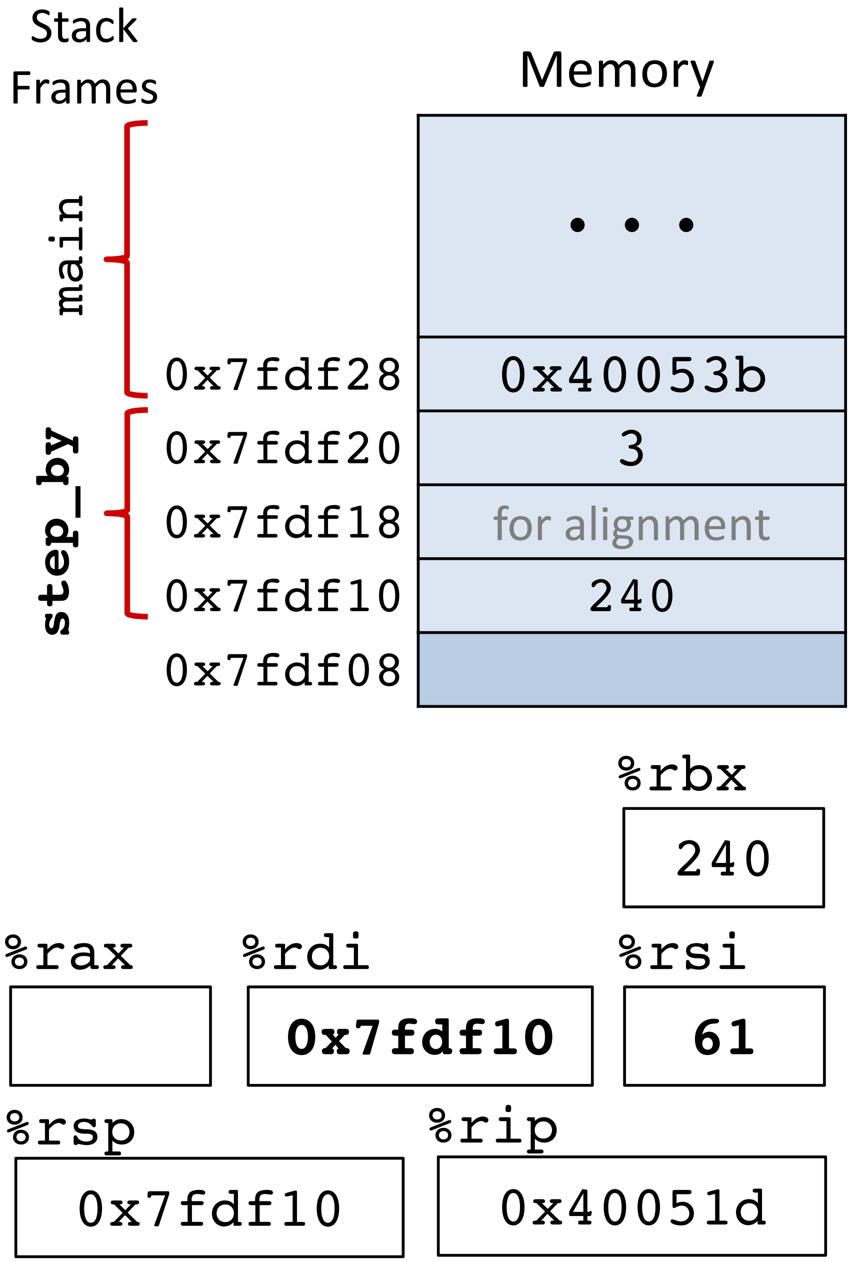
%rbx	%rax	%rdi	%rsi	%rsp	%rip
240		240		**0x7fdf10**	0x400515

Callee-save example (step 4)

Set up arguments

```
long step_by(long x) {
    long v1 = x;
    long v2 = increment(&v1, 61);
    return x + v2;
}
```

```
step_by:
400504:  pushq  %rbx
400506:  movq   %rdi, %rbx
400509:  subq   $16, %rsp
40050d:  movq   %rdi, (%rsp)
400515:  movq   %rsp, %rdi
400518:  movl   $61, %esi
40051d:  callq  4004cd <increment>
400522:  addq   %rbx, %rax
400525:  addq   $16, %rsp
400529:  popq   %rbx
40052b:  retq
```



Stack Frames

Memory

Stack Frames	Address	Memory
main		. . .
	0x7fdf28	0x40053b
step_by	0x7fdf20	3
	0x7fdf18	for alignment
	0x7fdf10	240
	0x7fdf08	

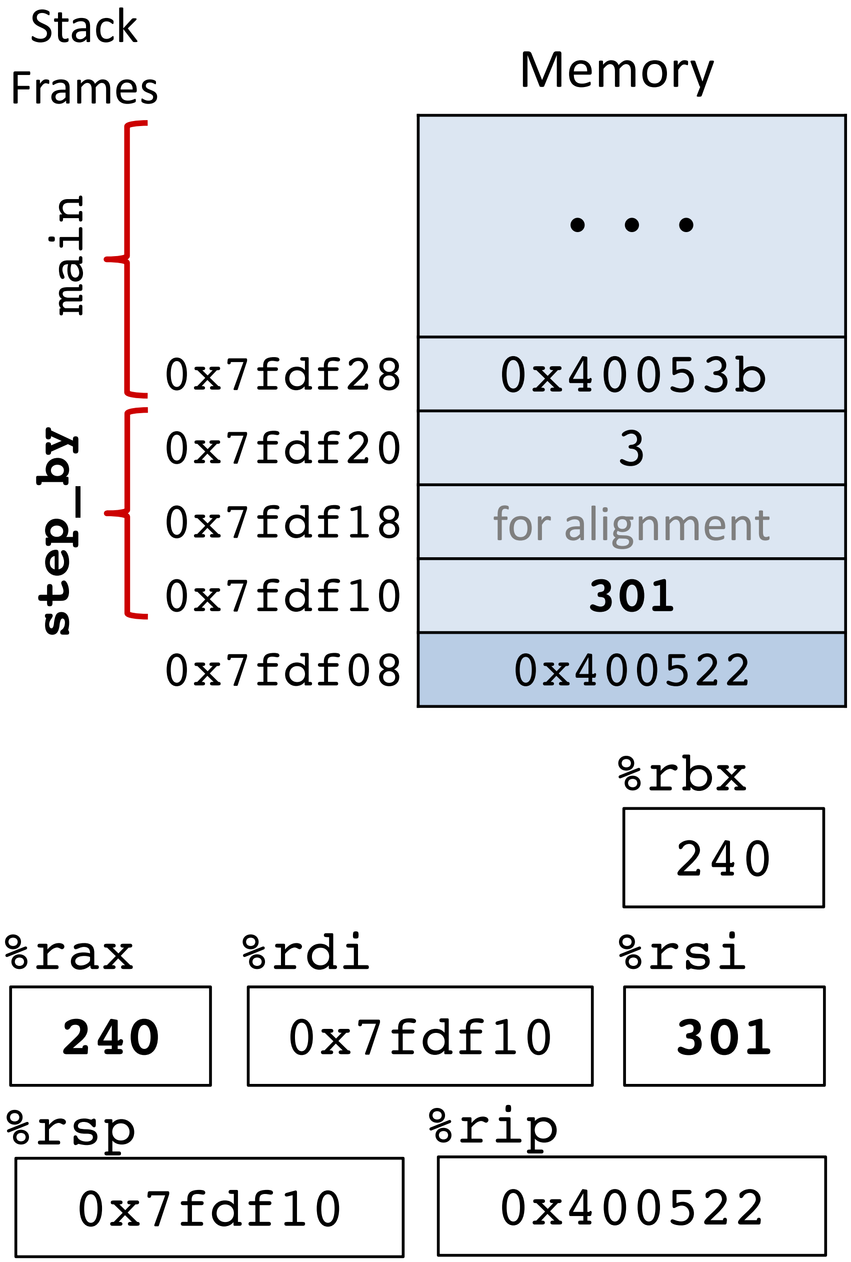
Register	Value
%rbx	240
%rax	
%rdi	**0x7fdf10**
%rsi	**61**
%rsp	0x7fdf10
%rip	0x40051d

Callee-save example (step 5)

Call, execute, and return from `increment`

```
long step_by(long x) {
    long v1 = x;
    long v2 = increment(&v1, 61);
    return x + v2;
}
```

```
step_by:
400504:  pushq  %rbx
400506:  movq   %rdi, %rbx
400509:  subq   $16, %rsp
40050d:  movq   %rdi, (%rsp)
400515:  movq   %rsp, %rdi
400518:  movl   $61, %esi
40051d:  callq  4004cd <increment>
400522:  addq   %rbx, %rax
400525:  addq   $16, %rsp
400529:  popq   %rbx
40052b:  retq
```



Stack Frames

Memory

Stack Frames	Address	Memory
main		. . .
	0x7fdf28	0x40053b
step_by	0x7fdf20	3
	0x7fdf18	for alignment
	0x7fdf10	**301**
	0x7fdf08	0x400522

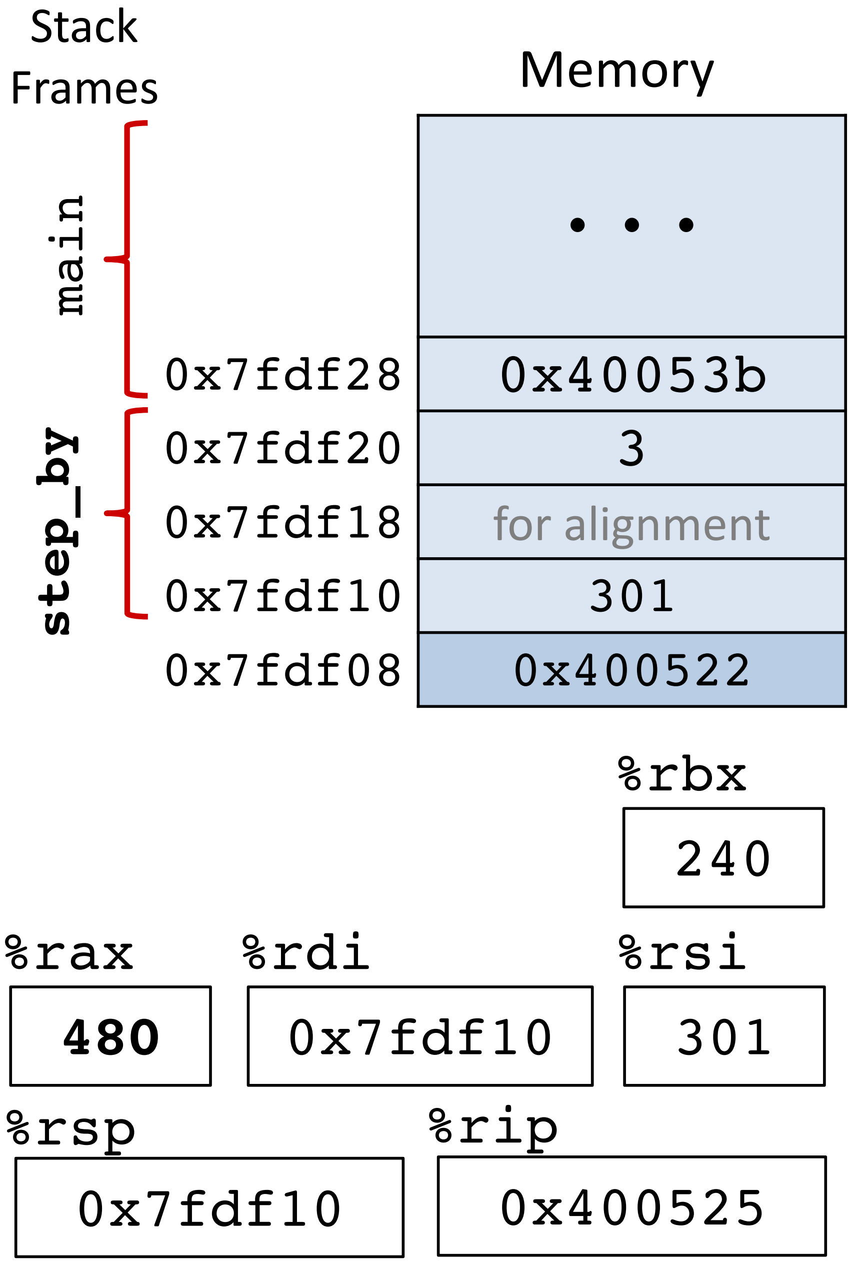
Register	Value
%rbx	240
%rax	**240**
%rdi	0x7fdf10
%rsi	**301**
%rsp	0x7fdf10
%rip	0x400522

Callee-save example (step 6)

Prepare return value

```
long step_by(long x) {
    long v1 = x;
    long v2 = increment(&v1, 61);
    return x + v2;
}
```

```
step_by:
400504:   pushq  %rbx
400506:   movq   %rdi, %rbx
400509:   subq   $16, %rsp
40050d:   movq   %rdi, (%rsp)
400515:   movq   %rsp, %rdi
400518:   movl   $61, %esi
40051d:   callq  4004cd <increment>
400522:   addq   %rbx, %rax
400525:   addq   $16, %rsp
400529:   popq   %rbx
40052b:   retq
```



Stack Frames

Memory

main

step_by

	. . .
0x7fdf28	0x40053b
0x7fdf20	3
0x7fdf18	for alignment
0x7fdf10	301
0x7fdf08	0x400522

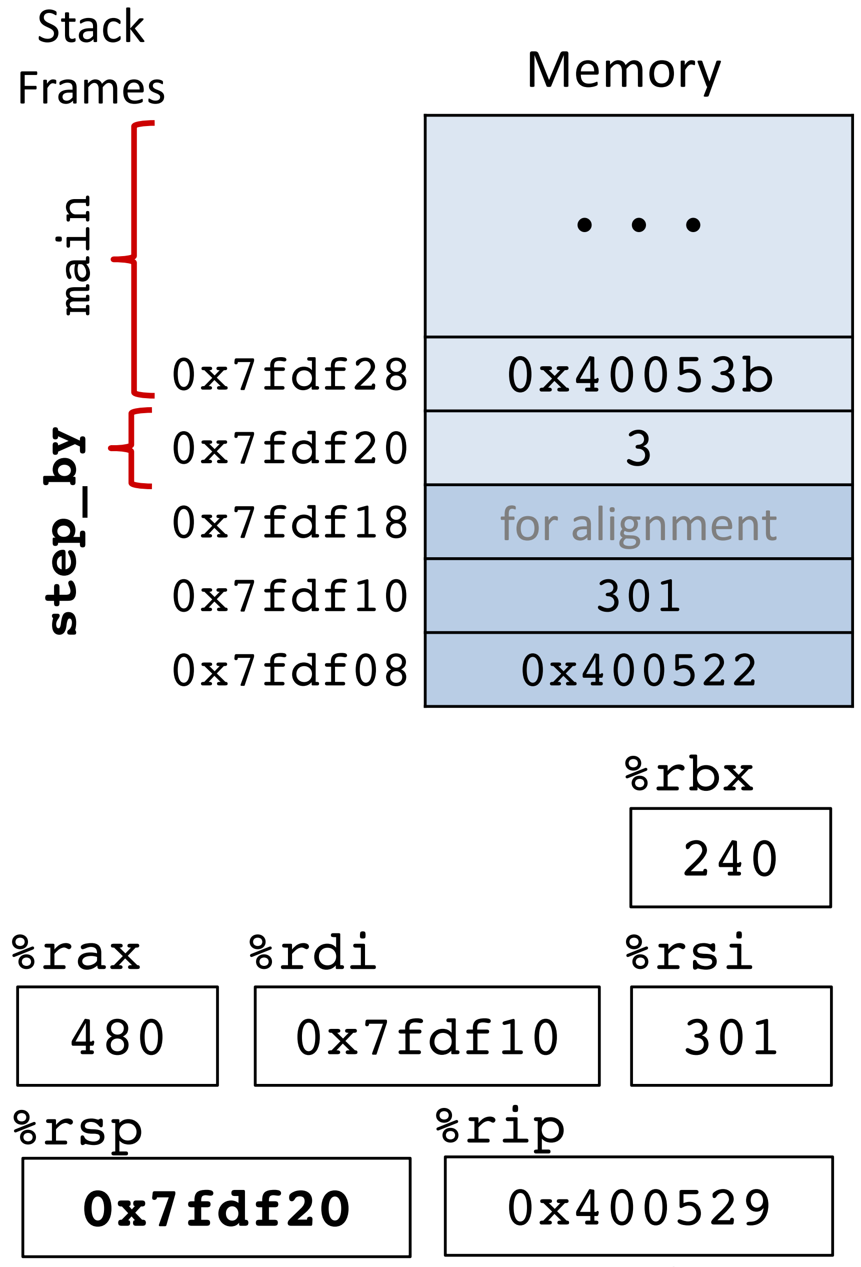
%rbx	240
%rax	480
%rdi	0x7fdf10
%rsi	301
%rsp	0x7fdf10
%rip	0x400525

Callee-save example (step 7)

Clean up stack frame

```
long step_by(long x) {
    long v1 = x;
    long v2 = increment(&v1, 61);
    return x + v2;
}
```

```
step_by:
400504:  pushq  %rbx
400506:  movq   %rdi, %rbx
400509:  subq   $16, %rsp
40050d:  movq   %rdi, (%rsp)
400515:  movq   %rsp, %rdi
400518:  movl   $61, %esi
40051d:  callq  4004cd <increment>
400522:  addq   %rbx, %rax
400525:  addq   $16, %rsp
400529:  popq   %rbx
40052b:  retq
```



Stack Frames

Memory

main

step_by

. . .

0x7fdf28 0x40053b

0x7fdf20 3

0x7fdf18 for alignment

0x7fdf10 301

0x7fdf08 0x400522

%rbx 240

%rax 480

%rdi 0x7fdf10

%rsi 301

%rsp **0x7fdf20**

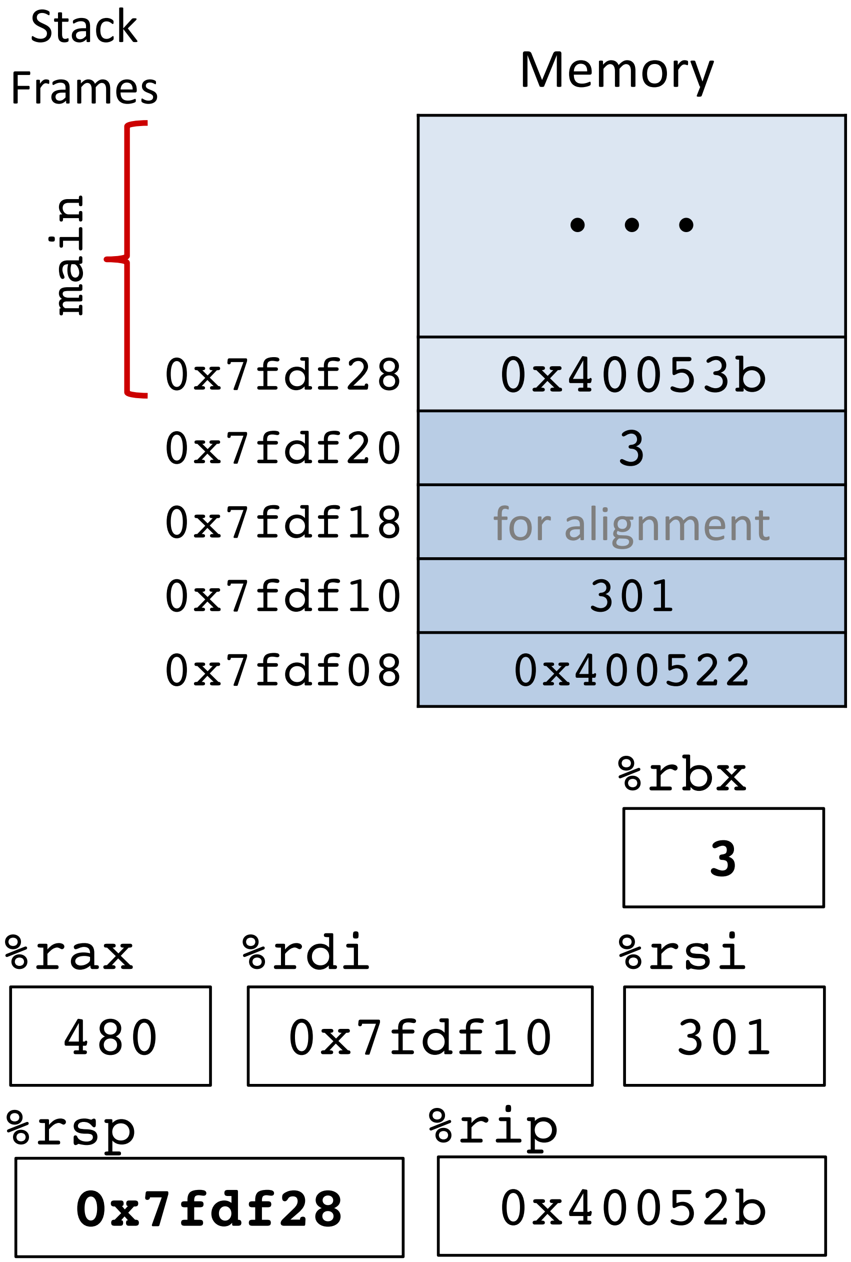
%rip 0x400529

Callee-save example (step 8)

Restore register %rbx
Ready to return

```
long step_by(long x) {
    long v1 = x;
    long v2 = increment(&v1, 61);
    return x + v2;
}
```

```
step_by:
400504:  pushq %rbx
400506:  movq  %rdi, %rbx
400509:  subq  $16, %rsp
40050d:  movq  %rdi, (%rsp)
400515:  movq  %rsp, %rdi
400518:  movl  $61, %esi
40051d:  callq 4004cd <increment>
400522:  addq  %rbx, %rax
400525:  addq  $16, %rsp
400529:  popq  %rbx
40052b:  retq
```



Memory

	. . .
0x7fdf28	0x40053b
0x7fdf20	3
0x7fdf18	for alignment
0x7fdf10	301
0x7fdf08	0x400522

%rbx: **3**

%rax: 480

%rdi: 0x7fdf10

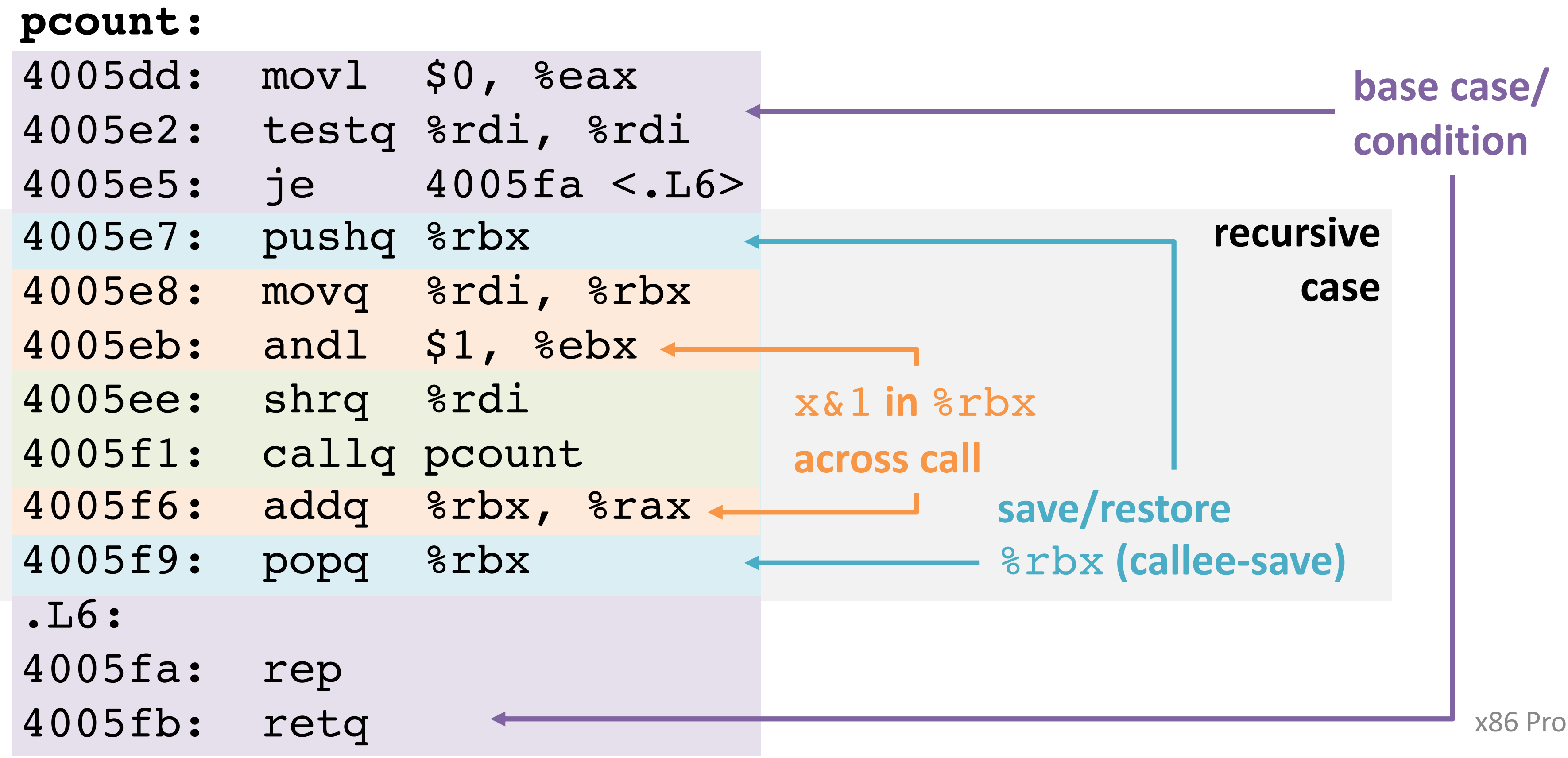
%rsi: 301

%rsp: **0x7fdf28**

%rip: 0x40052b

Recursion example: code

```c
long pcount(unsigned long x) {
  if (x == 0) {
    return 0;
  } else {
    return (x & 1) + pcount(x >> 1);
  }
}
```



```
pcount:
4005dd:  movl  $0, %eax
4005e2:  testq %rdi, %rdi
4005e5:  je    4005fa <.L6>
4005e7:  pushq %rbx
4005e8:  movq  %rdi, %rbx
4005eb:  andl  $1, %ebx
4005ee:  shrq  %rdi
4005f1:  callq pcount
4005f6:  addq  %rbx, %rax
4005f9:  popq  %rbx
.L6:
4005fa:  rep
4005fb:  retq
```

base case/ condition

recursive case

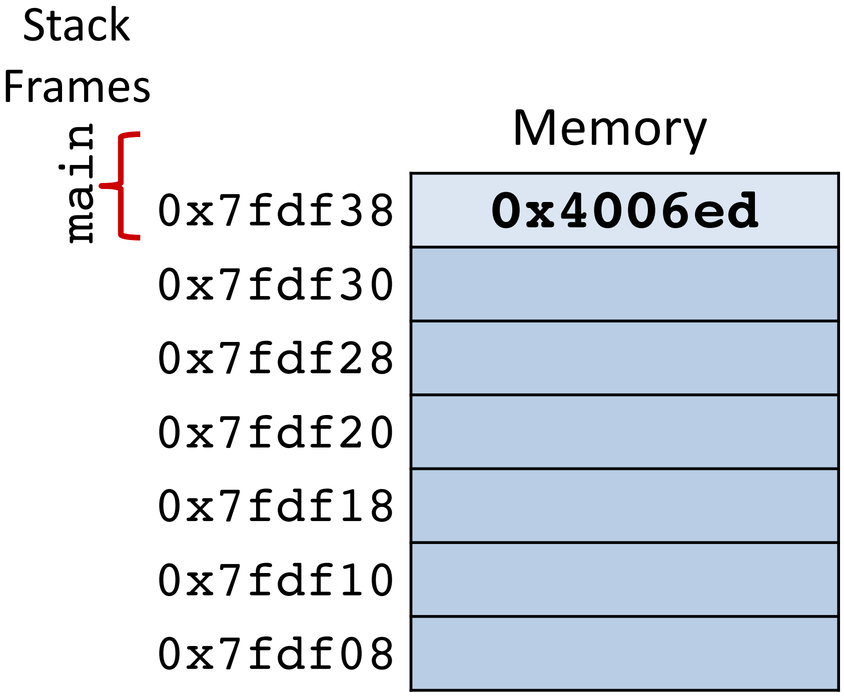
x&1 in %rbx across call

save/restore %rbx (callee-save)

Recursion Example: `pcount(2)`

```
long pcount(unsigned long x) {
  if (x == 0) {
    return 0;
  } else {
    return (x & 1) + pcount(x >> 1);
  }
}
```

```
pcount:
4005dd:  movl   $0, %eax
4005e2:  testq  %rdi, %rdi
4005e5:  je     4005fa <.L6>
4005e7:  pushq  %rbx
4005e8:  movq   %rdi, %rbx
4005eb:  andl   $1, %ebx
4005ee:  shrq   %rdi
4005f1:  callq  pcount
4005f6:  addq   %rbx, %rax
4005f9:  popq   %rbx
.L6:
4005fa:  rep
4005fb:  retq
```



	Memory
0x7fdf38	**0x4006ed**
0x7fdf30	
0x7fdf28	
0x7fdf20	
0x7fdf18	
0x7fdf10	
0x7fdf08	

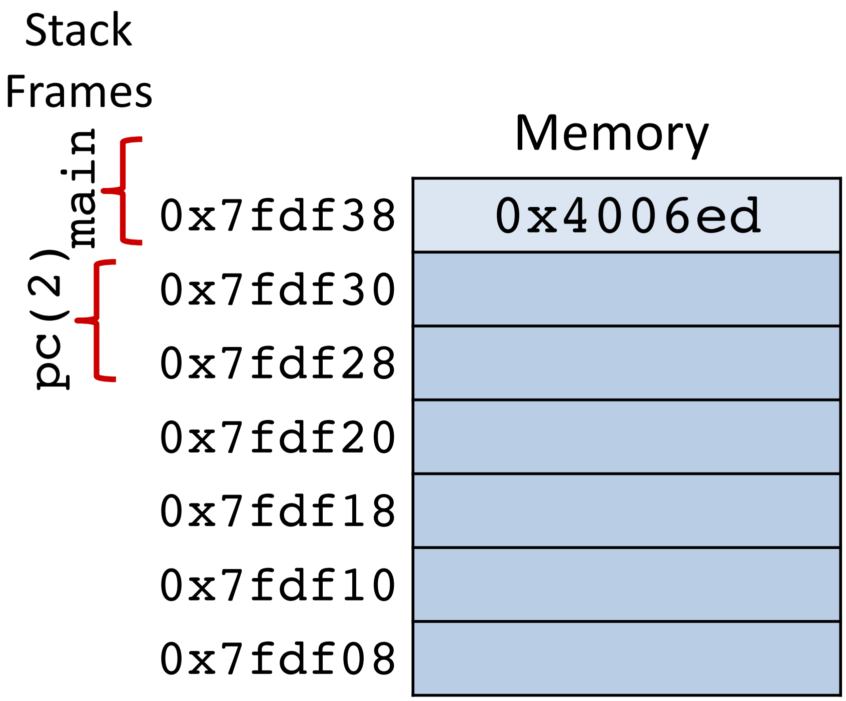
%rax	%rdi	%rbx
?	2	42

%rsp	%rip
0x7fdf38	**0x4005dd**

Recursion Example: `pcount(2)`

```
long pcount(unsigned long x) {
  if (x == 0) {
    return 0;
  } else {
    return (x & 1) + pcount(x >> 1);
  }
}
```

```
pcount:
4005dd:   movl   $0, %eax
4005e2:   testq  %rdi, %rdi
4005e5:   je     4005fa <.L6>
4005e7:   pushq  %rbx
4005e8:   movq   %rdi, %rbx
4005eb:   andl   $1, %ebx
4005ee:   shrq   %rdi
4005f1:   callq  pcount
4005f6:   addq   %rbx, %rax
4005f9:   popq   %rbx
.L6:
4005fa:   rep
4005fb:   retq
```



Stack Frames

main

pc(2)

Memory

0x7fdf38	0x4006ed
0x7fdf30	
0x7fdf28	
0x7fdf20	
0x7fdf18	
0x7fdf10	
0x7fdf08	

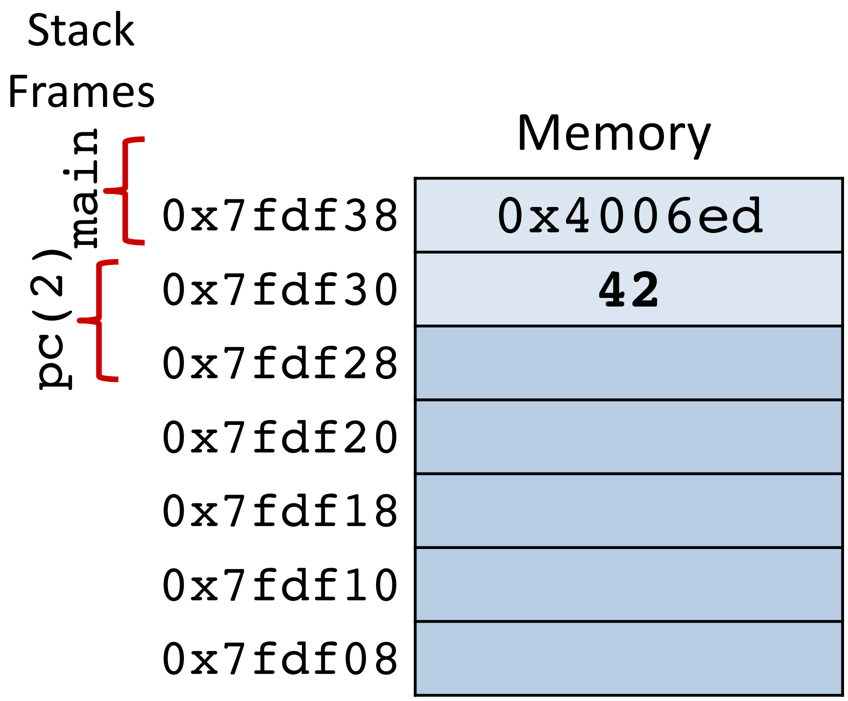
%rax	%rdi	%rbx
0	2	42

%rsp	%rip
0x7fdf38	0x4005e7

Recursion Example: `pcount(2)`

```
long pcount(unsigned long x) {
  if (x == 0) {
    return 0;
  } else {
    return (x & 1) + pcount(x >> 1);
  }
}
```

```
pcount:
4005dd:   movl   $0, %eax
4005e2:   testq  %rdi, %rdi
4005e5:   je     4005fa <.L6>
4005e7:   pushq  %rbx
4005e8:   movq   %rdi, %rbx
4005eb:   andl   $1, %ebx
4005ee:   shrq   %rdi
4005f1:   callq  pcount
4005f6:   addq   %rbx, %rax
4005f9:   popq   %rbx
.L6:
4005fa:   rep
4005fb:   retq
```



Address	Memory
0x7fdf38	0x4006ed
0x7fdf30	**42**
0x7fdf28	
0x7fdf20	
0x7fdf18	
0x7fdf10	
0x7fdf08	

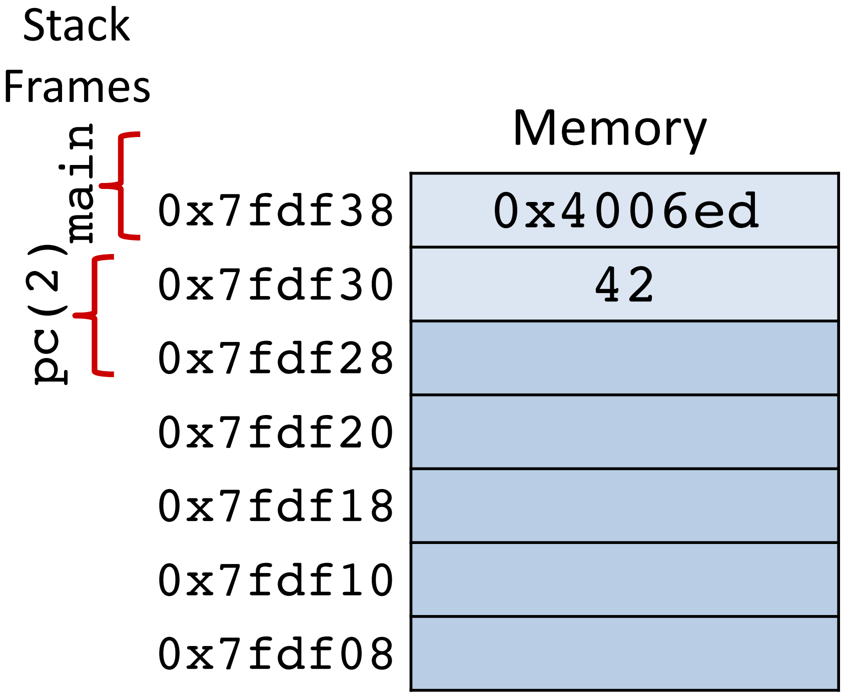
%rax	%rdi	%rbx
0	2	**2**

%rsp	%rip
0x7fdf30	0x4005eb

Recursion Example: `pcount(2)`

```
long pcount(unsigned long x) {
  if (x == 0) {
    return 0;
  } else {
    return (x & 1) + pcount(x >> 1);
  }
}
```

```
pcount:
4005dd:  movl   $0, %eax
4005e2:  testq  %rdi, %rdi
4005e5:  je     4005fa <.L6>
4005e7:  pushq  %rbx
4005e8:  movq   %rdi, %rbx
4005eb:  andl   $1, %ebx
4005ee:  shrq   %rdi
4005f1:  callq  pcount
4005f6:  addq   %rbx, %rax
4005f9:  popq   %rbx
.L6:
4005fa:  rep
4005fb:  retq
```



Stack Frames		Memory
main	0x7fdf38	0x4006ed
pc(2)	0x7fdf30	42
pc(2)	0x7fdf28	
	0x7fdf20	
	0x7fdf18	
	0x7fdf10	
	0x7fdf08	

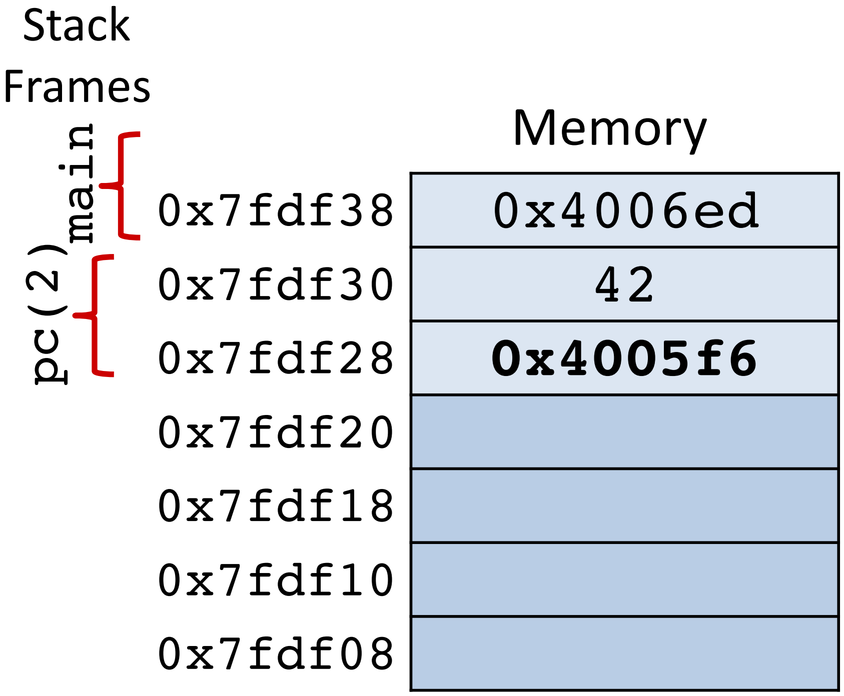
%rax	%rdi	%rbx
0	1	0

%rsp	%rip
0x7fdf30	0x4005f1

Recursion Example: `pcount(2)` → **`pcount(1)`**

```
long pcount(unsigned long x) {
  if (x == 0) {
    return 0;
  } else {
    return (x & 1) + pcount(x >> 1);
  }
}
```

```
pcount:
4005dd:  movl   $0, %eax
4005e2:  testq  %rdi, %rdi
4005e5:  je     4005fa <.L6>
4005e7:  pushq  %rbx
4005e8:  movq   %rdi, %rbx
4005eb:  andl   $1, %ebx
4005ee:  shrq   %rdi
4005f1:  callq  pcount
4005f6:  addq   %rbx, %rax
4005f9:  popq   %rbx
.L6:
4005fa:  rep
4005fb:  retq
```



	Memory
0x7fdf38	0x4006ed
0x7fdf30	42
0x7fdf28	**0x4005f6**
0x7fdf20	
0x7fdf18	
0x7fdf10	
0x7fdf08	

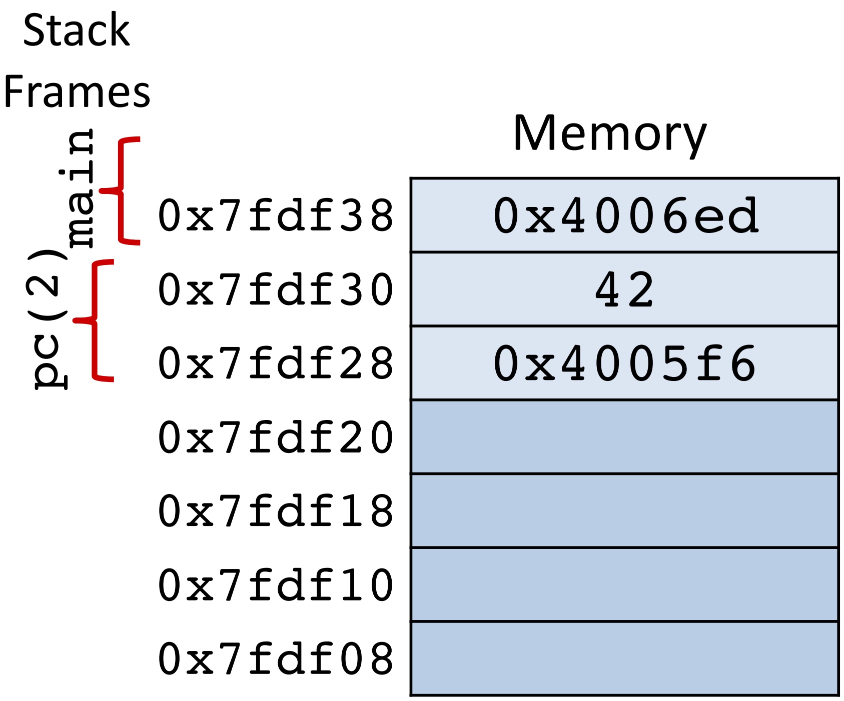
%rax	%rdi	%rbx
0	1	0

%rsp	%rip
0x7fdf28	**0x4005dd**

Recursion Example: pcount(2) → **pcount(1)**

```
long pcount(unsigned long x) {
  if (x == 0) {
    return 0;
  } else {
    return (x & 1) + pcount(x >> 1);
  }
}
```

```
pcount:
4005dd:   movl   $0, %eax
4005e2:   testq  %rdi, %rdi
4005e5:   je     4005fa <.L6>
4005e7:   pushq  %rbx
4005e8:   movq   %rdi, %rbx
4005eb:   andl   $1, %ebx
4005ee:   shrq   %rdi
4005f1:   callq  pcount
4005f6:   addq   %rbx, %rax
4005f9:   popq   %rbx
.L6:
4005fa:   rep
4005fb:   retq
```



Stack Frames		Memory
main	0x7fdf38	0x4006ed
pc(2)	0x7fdf30	42
	0x7fdf28	0x4005f6
	0x7fdf20	
	0x7fdf18	
	0x7fdf10	
	0x7fdf08	

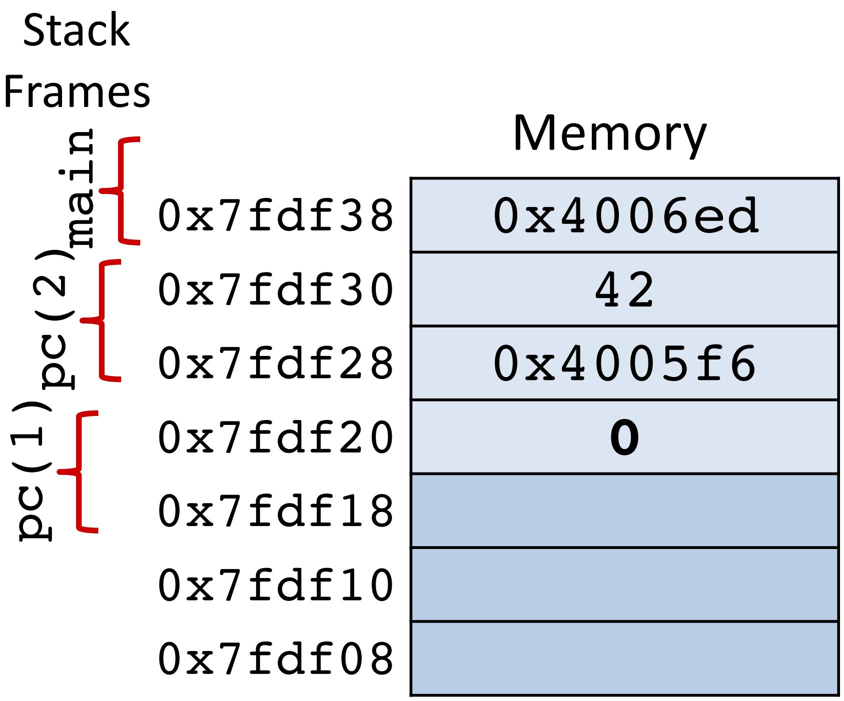
%rax	%rdi	%rbx
0	1	0

%rsp	%rip
0x7fdf28	**0x4005e7**

Recursion Example: pcount(2) → **pcount(1)**

```
long pcount(unsigned long x) {
  if (x == 0) {
    return 0;
  } else {
    return (x & 1) + pcount(x >> 1);
  }
}
```

```
pcount:
4005dd:  movl   $0, %eax
4005e2:  testq  %rdi, %rdi
4005e5:  je     4005fa <.L6>
4005e7:  pushq  %rbx
4005e8:  movq   %rdi, %rbx
4005eb:  andl   $1, %ebx
4005ee:  shrq   %rdi
4005f1:  callq  pcount
4005f6:  addq   %rbx, %rax
4005f9:  popq   %rbx
.L6:
4005fa:  rep
4005fb:  retq
```



Address	Memory
0x7fdf38	0x4006ed
0x7fdf30	42
0x7fdf28	0x4005f6
0x7fdf20	**0**
0x7fdf18	
0x7fdf10	
0x7fdf08	

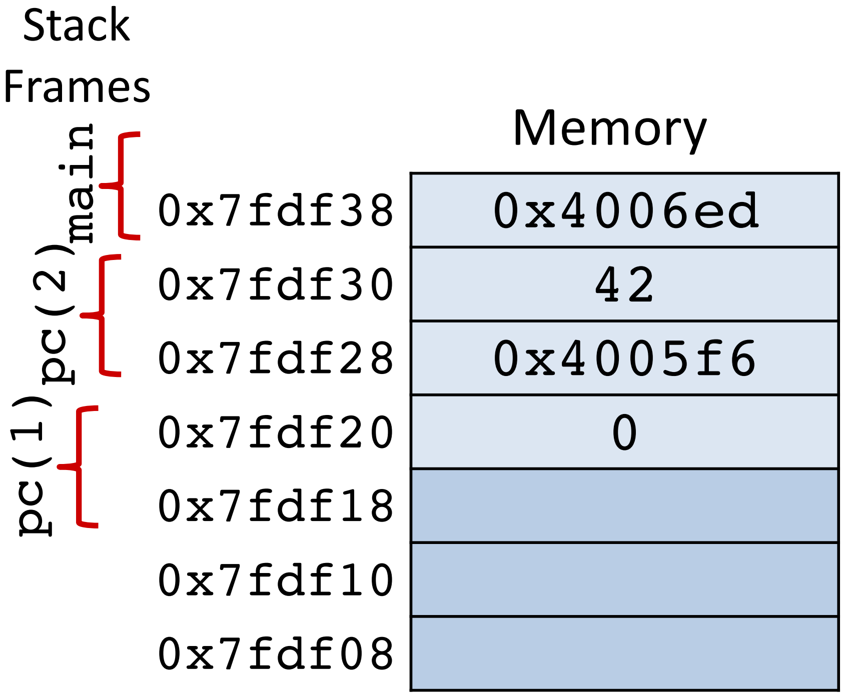
%rax	%rdi	%rbx
0	1	**1**

%rsp	%rip
0x7fdf20	0x4005eb

Recursion Example: pcount(2) → **pcount(1)**

```
long pcount(unsigned long x) {
  if (x == 0) {
    return 0;
  } else {
    return (x & 1) + pcount(x >> 1);
  }
}
```

```
pcount:
4005dd:   movl   $0, %eax
4005e2:   testq  %rdi, %rdi
4005e5:   je     4005fa <.L6>
4005e7:   pushq  %rbx
4005e8:   movq   %rdi, %rbx
4005eb:   andl   $1, %ebx
4005ee:   shrq   %rdi
4005f1:   callq  pcount
4005f6:   addq   %rbx, %rax
4005f9:   popq   %rbx
.L6:
4005fa:   rep
4005fb:   retq
```



Stack Frames		Memory
main	0x7fdf38	0x4006ed
pc(2)	0x7fdf30	42
	0x7fdf28	0x4005f6
pc(1)	0x7fdf20	0
	0x7fdf18	
	0x7fdf10	
	0x7fdf08	

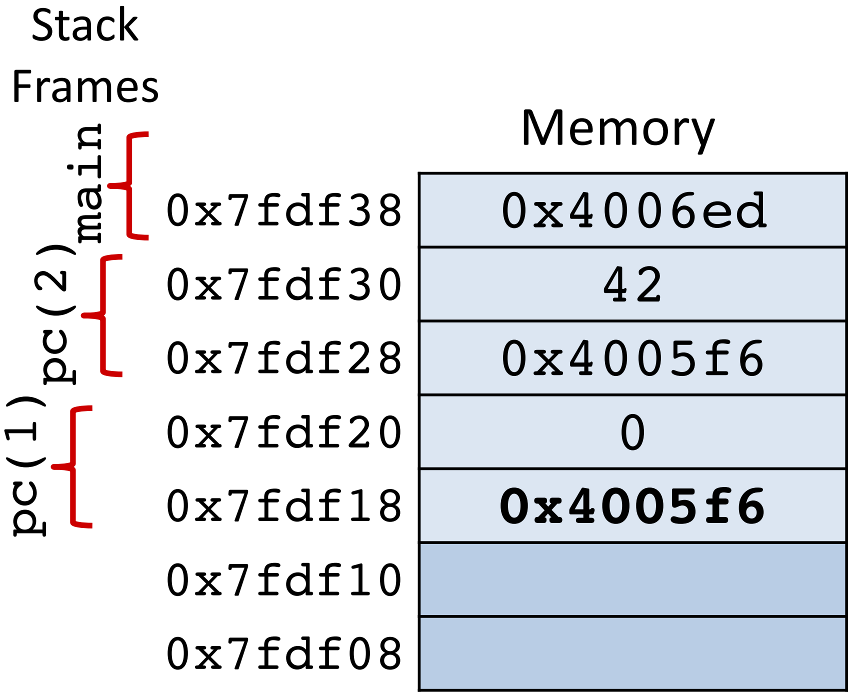
%rax	%rdi	%rbx
0	**0**	**1**

%rsp	%rip
0x7fdf20	0x4005f1

Recursion Example: pcount(2) → pcount(1) → **pcount(0)**

```
long pcount(unsigned long x) {
  if (x == 0) {
    return 0;
  } else {
    return (x & 1) + pcount(x >> 1);
  }
}
```

```
pcount:
4005dd:  movl   $0, %eax
4005e2:  testq  %rdi, %rdi
4005e5:  je     4005fa <.L6>
4005e7:  pushq  %rbx
4005e8:  movq   %rdi, %rbx
4005eb:  andl   $1, %ebx
4005ee:  shrq   %rdi
4005f1:  callq  pcount
4005f6:  addq   %rbx, %rax
4005f9:  popq   %rbx
.L6:
4005fa:  rep
4005fb:  retq
```



Stack Frames		Memory
main	0x7fdf38	0x4006ed
pc(2)	0x7fdf30	42
	0x7fdf28	0x4005f6
pc(1)	0x7fdf20	0
	0x7fdf18	**0x4005f6**
	0x7fdf10	
	0x7fdf08	

%rax	%rdi	%rbx
0	0	1

%rsp	%rip
0x7fdf18	**0x4005dd**

Recursion Example: pcount(2) → pcount(1) → **pcount(0)**

```
long pcount(unsigned long x) {
  if (x == 0) {
    return 0;
  } else {
    return (x & 1) + pcount(x >> 1);
  }
}
```

```
pcount:
4005dd:  movl   $0, %eax
4005e2:  testq  %rdi, %rdi
4005e5:  je     4005fa <.L6>
4005e7:  pushq  %rbx
4005e8:  movq   %rdi, %rbx
4005eb:  andl   $1, %ebx
4005ee:  shrq   %rdi
4005f1:  callq  pcount
4005f6:  addq   %rbx, %rax
4005f9:  popq   %rbx
.L6:
4005fa:  rep
4005fb:  retq
```

Stack Frames		Memory
main	0x7fdf38	0x4006ed
pc(2)	0x7fdf30	42
	0x7fdf28	0x4005f6
pc(1)	0x7fdf20	0
	0x7fdf18	0x4005f6
	0x7fdf10	
	0x7fdf08	

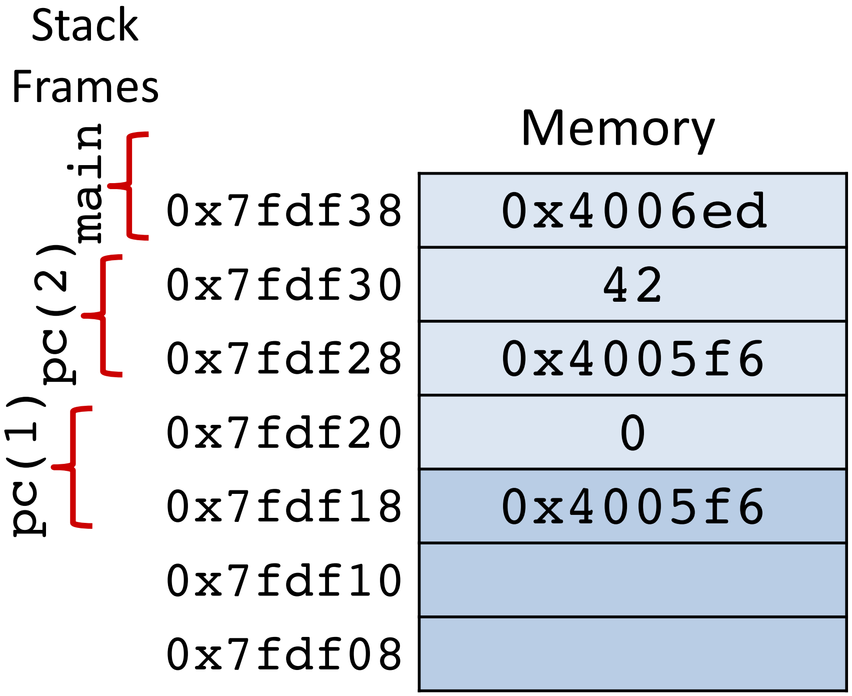
%rax	%rdi	%rbx
0	0	1

%rsp	%rip
0x7fdf18	**0x4005fa**

Recursion Example: `pcount(2)` → `pcount(1)` → **`pcount(0)`**

```
long pcount(unsigned long x) {
  if (x == 0) {
    return 0;
  } else {
    return (x & 1) + pcount(x >> 1);
  }
}
```

```
pcount:
4005dd:  movl  $0, %eax
4005e2:  testq %rdi, %rdi
4005e5:  je    4005fa <.L6>
4005e7:  pushq %rbx
4005e8:  movq  %rdi, %rbx
4005eb:  andl  $1, %ebx
4005ee:  shrq  %rdi
4005f1:  callq pcount
4005f6:  addq  %rbx, %rax
4005f9:  popq  %rbx
.L6:
4005fa:  rep
4005fb:  retq
```



Memory

0x7fdf38	0x4006ed
0x7fdf30	42
0x7fdf28	0x4005f6
0x7fdf20	0
0x7fdf18	0x4005f6
0x7fdf10	
0x7fdf08	

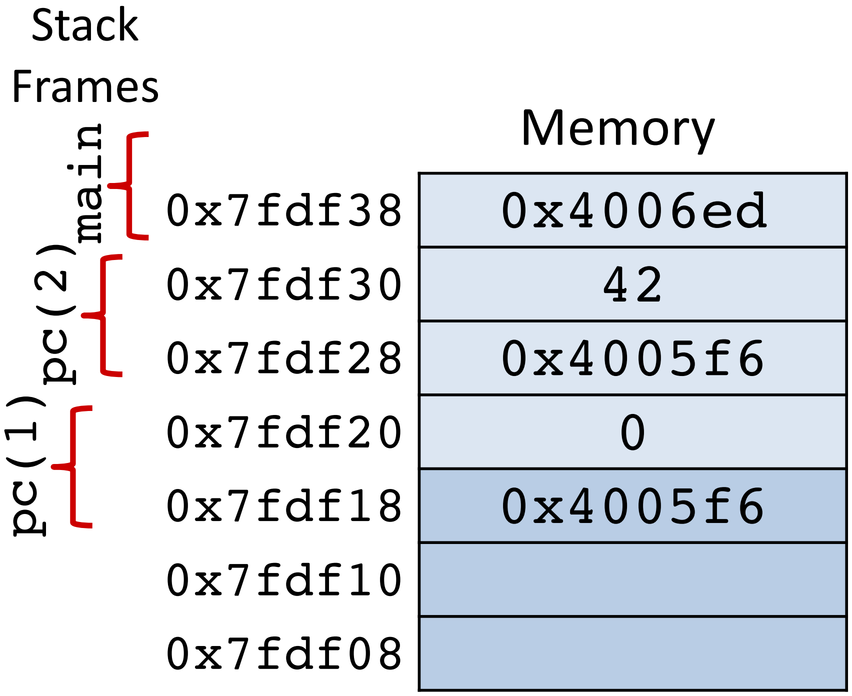
%rax	%rdi	%rbx
0	0	1

%rsp	%rip
0x7fdf20	**0x4005f6**

Recursion Example: `pcount(2)` → `pcount(1)` → **`pcount(0)`**

```
long pcount(unsigned long x) {
  if (x == 0) {
    return 0;
  } else {
    return (x & 1) + pcount(x >> 1);
  }
}
```

```
pcount:
4005dd:   movl   $0, %eax
4005e2:   testq  %rdi, %rdi
4005e5:   je     4005fa <.L6>
4005e7:   pushq  %rbx
4005e8:   movq   %rdi, %rbx
4005eb:   andl   $1, %ebx
4005ee:   shrq   %rdi
4005f1:   callq  pcount
4005f6:   addq   %rbx, %rax
4005f9:   popq   %rbx
.L6:
4005fa:   rep
4005fb:   retq
```



Memory

0x7fdf38	0x4006ed
0x7fdf30	42
0x7fdf28	0x4005f6
0x7fdf20	0
0x7fdf18	0x4005f6
0x7fdf10	
0x7fdf08	

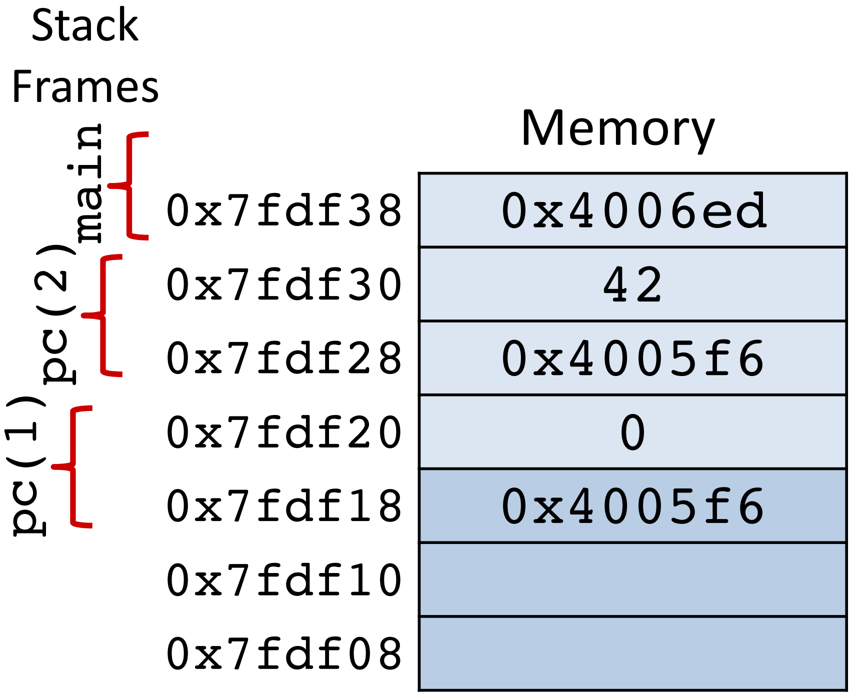
%rax	%rdi	%rbx
0	0	1

%rsp	%rip
0x7fdf20	**0x4005f6**

Recursion Example: pcount(2) → **pcount(1)** → pcount(0)

```
long pcount(unsigned long x) {
  if (x == 0) {
    return 0;
  } else {
    return (x & 1) + pcount(x >> 1);
  }
}
```

```
pcount:
4005dd:   movl   $0, %eax
4005e2:   testq  %rdi, %rdi
4005e5:   je     4005fa <.L6>
4005e7:   pushq  %rbx
4005e8:   movq   %rdi, %rbx
4005eb:   andl   $1, %ebx
4005ee:   shrq   %rdi
4005f1:   callq  pcount
4005f6:   addq   %rbx, %rax
4005f9:   popq   %rbx
.L6:
4005fa:   rep
4005fb:   retq
```



Stack Frames

Memory

Stack Frames		Memory
main	0x7fdf38	0x4006ed
pc(2)	0x7fdf30	42
	0x7fdf28	0x4005f6
pc(1)	0x7fdf20	0
	0x7fdf18	0x4005f6
	0x7fdf10	
	0x7fdf08	

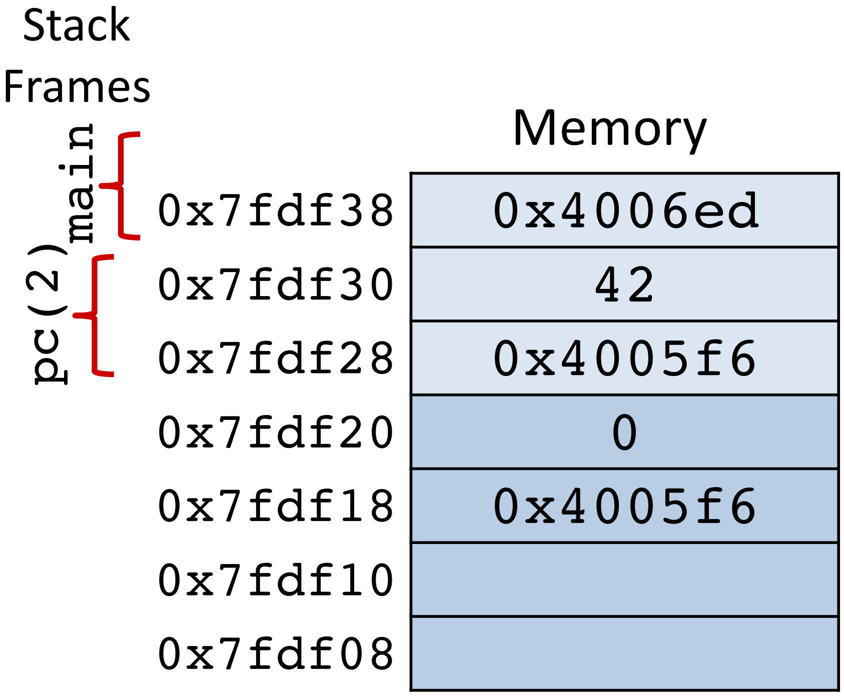
%rax	%rdi	%rbx
1	0	1

%rsp	%rip
0x7fdf20	0x4005f9

Recursion Example: pcount(2) → **pcount(1)** → pcount(0)

```
long pcount(unsigned long x) {
  if (x == 0) {
    return 0;
  } else {
    return (x & 1) + pcount(x >> 1);
  }
}
```

```
pcount:
4005dd:  movl   $0, %eax
4005e2:  testq  %rdi, %rdi
4005e5:  je     4005fa <.L6>
4005e7:  pushq  %rbx
4005e8:  movq   %rdi, %rbx
4005eb:  andl   $1, %ebx
4005ee:  shrq   %rdi
4005f1:  callq  pcount
4005f6:  addq   %rbx, %rax
4005f9:  popq   %rbx
.L6:
4005fa:  rep
4005fb:  retq
```



	Memory
0x7fdf38	0x4006ed
0x7fdf30	42
0x7fdf28	0x4005f6
0x7fdf20	0
0x7fdf18	0x4005f6
0x7fdf10	
0x7fdf08	

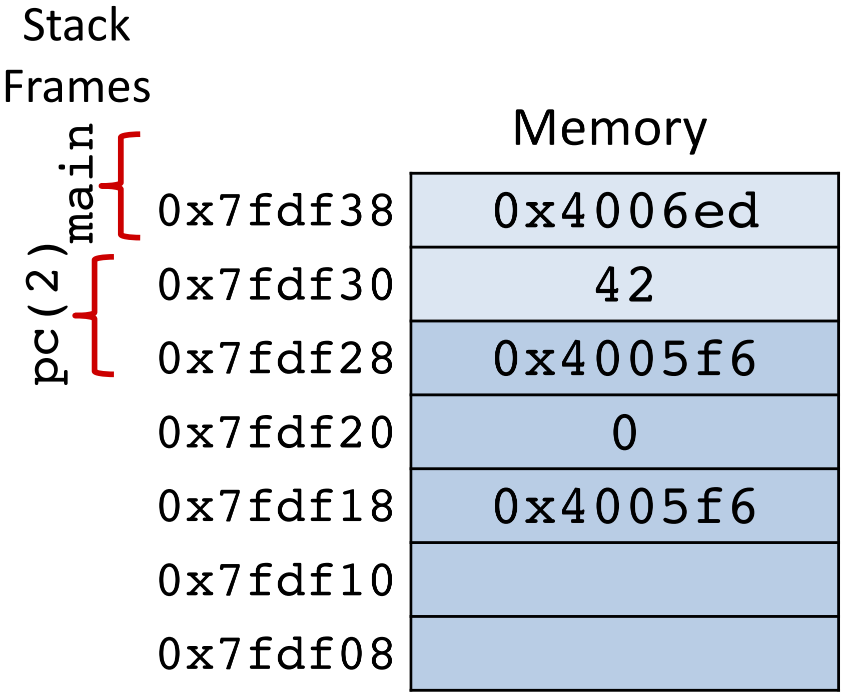
%rax	%rdi	%rbx
1	0	**0**

%rsp	%rip
0x7fdf28	0x4005fa

Recursion Example: pcount(2) → **pcount(1)** → pcount(0)

```
long pcount(unsigned long x) {
  if (x == 0) {
    return 0;
  } else {
    return (x & 1) + pcount(x >> 1);
  }
}
```

```
pcount:
4005dd:   movl   $0, %eax
4005e2:   testq  %rdi, %rdi
4005e5:   je     4005fa <.L6>
4005e7:   pushq  %rbx
4005e8:   movq   %rdi, %rbx
4005eb:   andl   $1, %ebx
4005ee:   shrq   %rdi
4005f1:   callq  pcount
4005f6:   addq   %rbx, %rax
4005f9:   popq   %rbx
.L6:
4005fa:   rep
4005fb:   retq
```



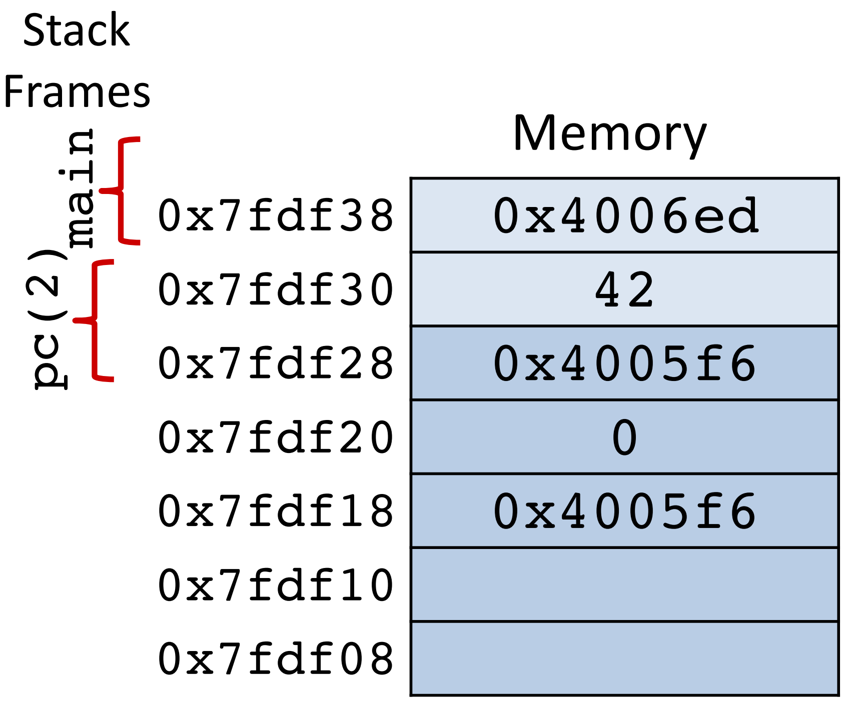
Address	Memory
0x7fdf38	0x4006ed
0x7fdf30	42
0x7fdf28	0x4005f6
0x7fdf20	0
0x7fdf18	0x4005f6
0x7fdf10	
0x7fdf08	

%rax: 1
%rdi: 0
%rbx: 0
%rsp: **0x7fdf30**
%rip: **0x4005f6**

Recursion Example: pcount(2) → **pcount(1)** → pcount(0)

```
long pcount(unsigned long x) {
  if (x == 0) {
    return 0;
  } else {
    return (x & 1) + pcount(x >> 1);
  }
}
```

```
pcount:
4005dd:  movl   $0, %eax
4005e2:  testq  %rdi, %rdi
4005e5:  je     4005fa <.L6>
4005e7:  pushq  %rbx
4005e8:  movq   %rdi, %rbx
4005eb:  andl   $1, %ebx
4005ee:  shrq   %rdi
4005f1:  callq  pcount
4005f6:  addq   %rbx, %rax
4005f9:  popq   %rbx
.L6:
4005fa:  rep
4005fb:  retq
```



	Memory
0x7fdf38	0x4006ed
0x7fdf30	42
0x7fdf28	0x4005f6
0x7fdf20	0
0x7fdf18	0x4005f6
0x7fdf10	
0x7fdf08	

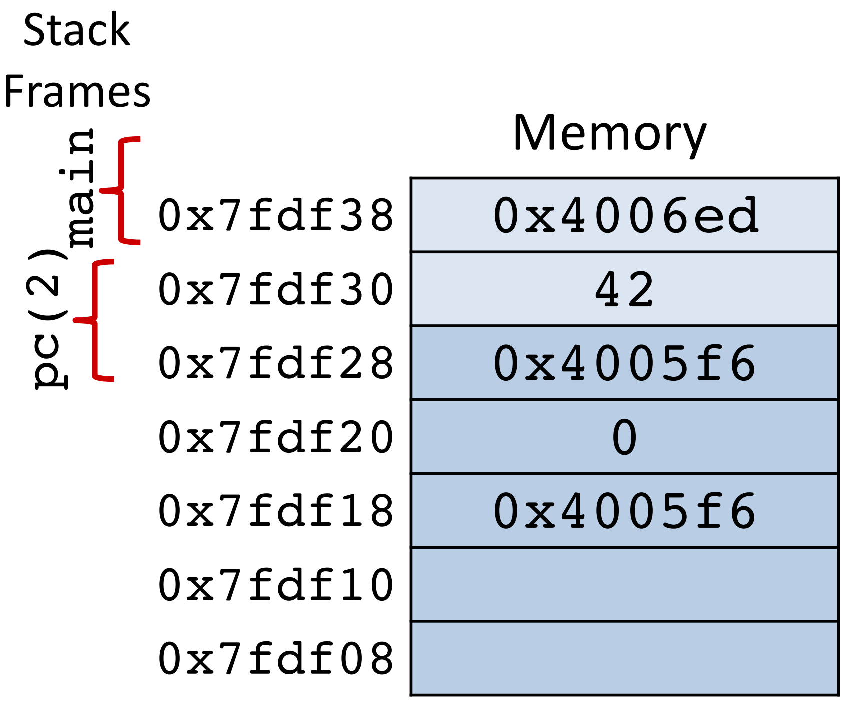
%rax	%rdi	%rbx
1	0	0

%rsp	%rip
0x7fdf30	**0x4005f6**

Recursion Example: pcount(2) → pcount(1) → pcount(0)

```
long pcount(unsigned long x) {
  if (x == 0) {
    return 0;
  } else {
    return (x & 1) + pcount(x >> 1);
  }
}
```

```
pcount:
4005dd:  movl   $0, %eax
4005e2:  testq  %rdi, %rdi
4005e5:  je     4005fa <.L6>
4005e7:  pushq  %rbx
4005e8:  movq   %rdi, %rbx
4005eb:  andl   $1, %ebx
4005ee:  shrq   %rdi
4005f1:  callq  pcount
4005f6:  addq   %rbx, %rax
4005f9:  popq   %rbx
.L6:
4005fa:  rep
4005fb:  retq
```



	Memory
0x7fdf38	0x4006ed
0x7fdf30	42
0x7fdf28	0x4005f6
0x7fdf20	0
0x7fdf18	0x4005f6
0x7fdf10	
0x7fdf08	

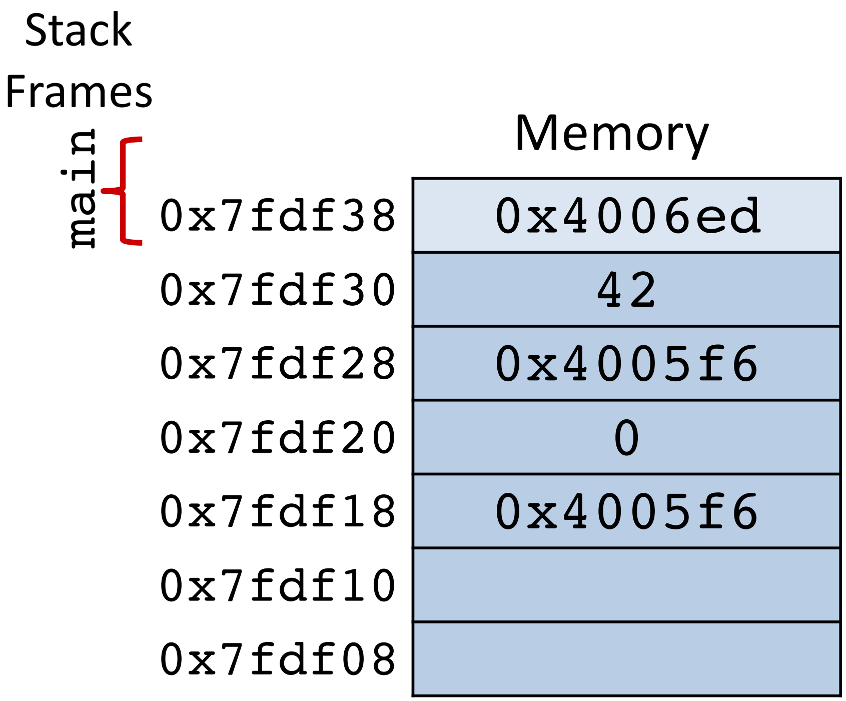
%rax	%rdi	%rbx
1	0	0

%rsp	%rip
0x7fdf30	0x4005f9

Recursion Example: `pcount(2)` → `pcount(1)` → `pcount(0)`

```
long pcount(unsigned long x) {
  if (x == 0) {
    return 0;
  } else {
    return (x & 1) + pcount(x >> 1);
  }
}
```

```
pcount:
4005dd:  movl   $0, %eax
4005e2:  testq  %rdi, %rdi
4005e5:  je     4005fa <.L6>
4005e7:  pushq  %rbx
4005e8:  movq   %rdi, %rbx
4005eb:  andl   $1, %ebx
4005ee:  shrq   %rdi
4005f1:  callq  pcount
4005f6:  addq   %rbx, %rax
4005f9:  popq   %rbx
.L6:
4005fa:  rep
4005fb:  retq
```



	Memory
0x7fdf38	0x4006ed
0x7fdf30	42
0x7fdf28	0x4005f6
0x7fdf20	0
0x7fdf18	0x4005f6
0x7fdf10	
0x7fdf08	

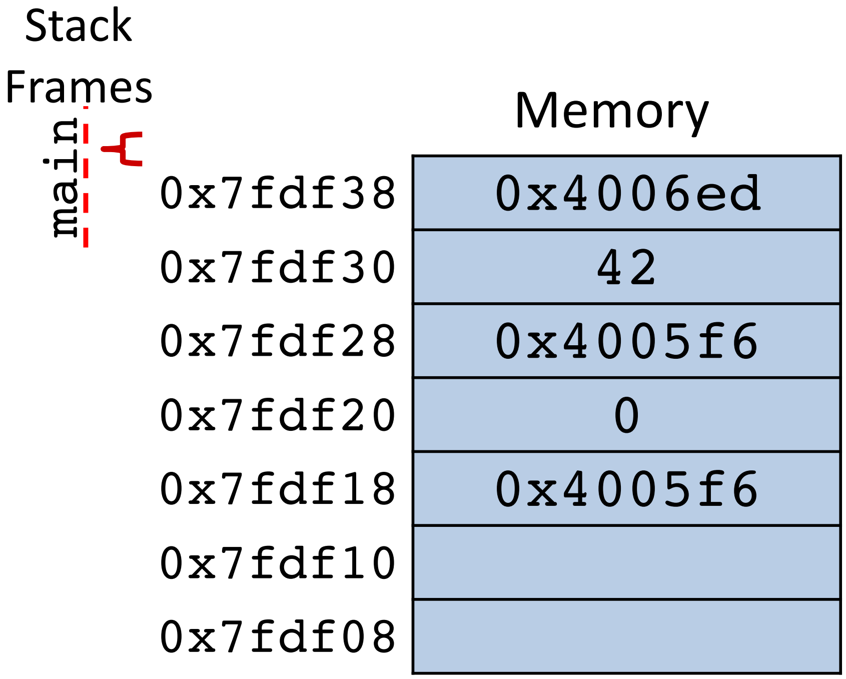
%rax	%rdi	%rbx
1	0	**42**

%rsp	%rip
0x7fdf38	0x4005f9

Recursion Example: `pcount(2)` → `pcount(1)` → `pcount(0)`

```
long pcount(unsigned long x) {
  if (x == 0) {
    return 0;
  } else {
    return (x & 1) + pcount(x >> 1);
  }
}
```

```
pcount:
4005dd:  movl   $0, %eax
4005e2:  testq  %rdi, %rdi
4005e5:  je     4005fa <.L6>
4005e7:  pushq  %rbx
4005e8:  movq   %rdi, %rbx
4005eb:  andl   $1, %ebx
4005ee:  shrq   %rdi
4005f1:  callq  pcount
4005f6:  addq   %rbx, %rax
4005f9:  popq   %rbx
.L6:
4005fa:  rep
4005fb:  retq
```



	Memory
0x7fdf38	0x4006ed
0x7fdf30	42
0x7fdf28	0x4005f6
0x7fdf20	0
0x7fdf18	0x4005f6
0x7fdf10	
0x7fdf08	

%rax	%rdi	%rbx
1	0	42

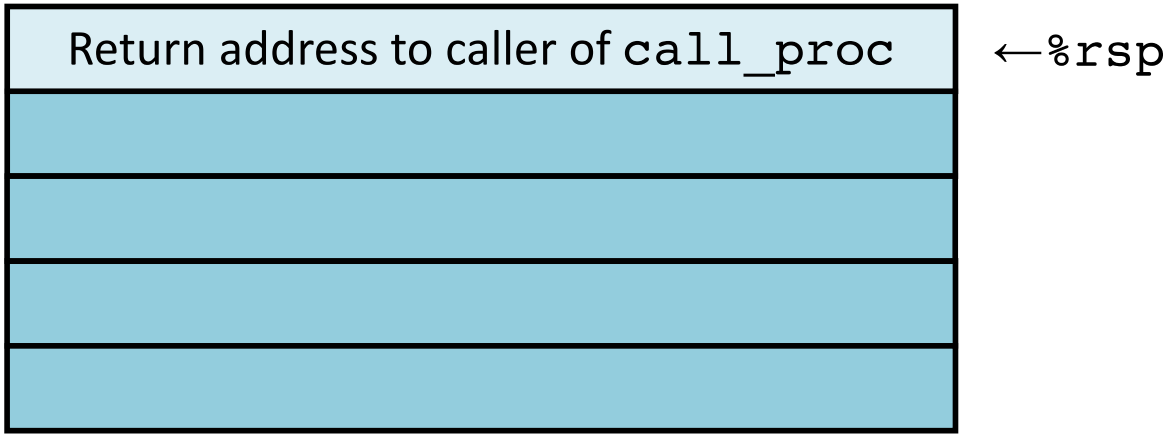
%rsp	%rip
0x7fdf40	0x4006ed

Stack storage example (1)

optional

```
long int call_proc()
{
  long  x1 = 1;
  int   x2 = 2;
  short x3 = 3;
  char  x4 = 4;
  proc(x1, &x1, x2, &x2,
       x3, &x3, x4, &x4);
  return (x1+x2)*(x3-x4);
}
```

```
call_proc:
  subq  $32,%rsp
  movq  $1,16(%rsp) # x1
  movl  $2,24(%rsp) # x2
  movw  $3,28(%rsp) # x3
  movb  $4,31(%rsp) # x4
  • • •
```



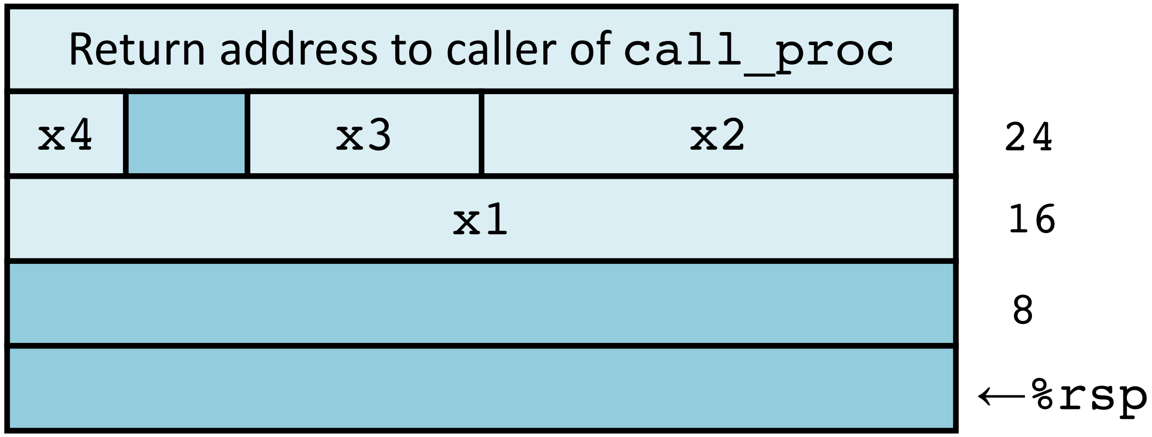
Stack storage example

optional

(2) Allocate local vars

```
long int call_proc()
{
  long  x1 = 1;
  int   x2 = 2;
  short x3 = 3;
  char  x4 = 4;
  proc(x1, &x1, x2, &x2,
       x3, &x3, x4, &x4);
  return (x1+x2)*(x3-x4);
}
```

```
call_proc:
  subq  $32,%rsp
  movq  $1,16(%rsp) # x1
  movl  $2,24(%rsp) # x2
  movw  $3,28(%rsp) # x3
  movb  $4,31(%rsp) # x4
  • • •
```



Return address to caller of call_proc

x4 | | x3 | x2 — 24

x1 — 16

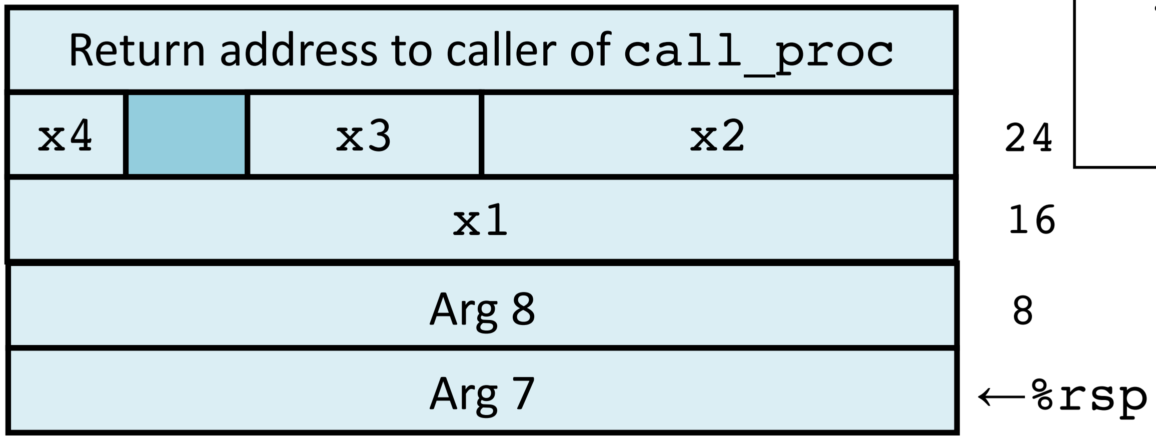
8

←%rsp

Stack storage example
(3) setup args to proc

optional

```
long int call_proc()
{
  long  x1 = 1;
  int   x2 = 2;
  short x3 = 3;
  char  x4 = 4;
  proc(x1, &x1, x2, &x2,
       x3, &x3, x4, &x4);
  return (x1+x2)*(x3-x4);
}
```



```
call_proc:
  • • •
  leaq   24(%rsp),%rcx # &x2
  leaq   16(%rsp),%rsi # &x1
  leaq   31(%rsp),%rax # &x4
  movq   %rax,8(%rsp)  # ...
  movl   $4,(%rsp)     # 4
  leaq   28(%rsp),%r9  # &x3
  movl   $3,%r8d       # 3
  movl   $2,%edx       # 2
  movq   $1,%rdi       # 1
  call   proc
  • • •
```

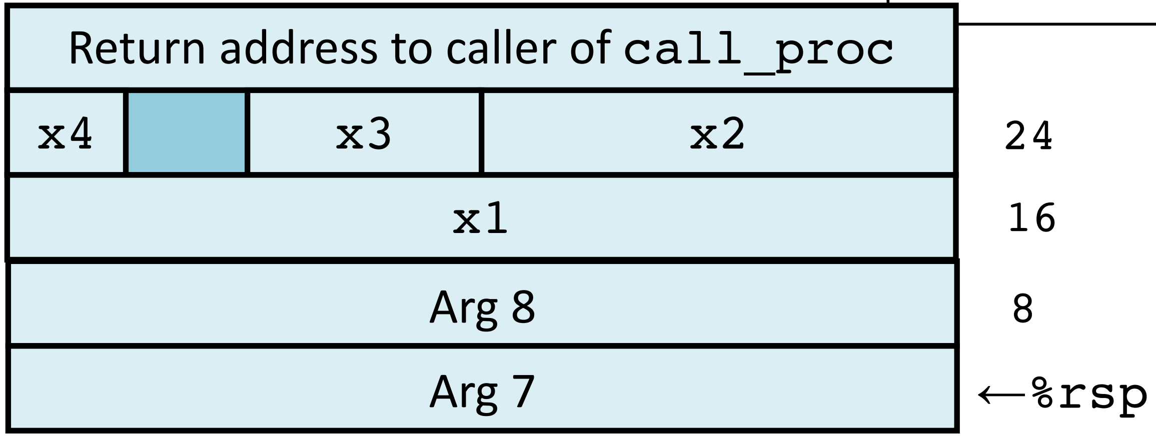
Arguments passed in (in order):
`%rdi, %rsi, %rdx, %rcx, %r8, %r9`

Stack storage example
(4) after call to `proc`

optional

```
long int call_proc()
{
  long  x1 = 1;
  int   x2 = 2;
  short x3 = 3;
  char  x4 = 4;
  proc(x1, &x1, x2, &x2,
       x3, &x3, x4, &x4);
  return (x1+x2)*(x3-x4);
}
```

```
call_proc:
  • • •
  movswl 28(%rsp),%eax # x3
  movsbl 31(%rsp),%edx # x4
  subl   %edx,%eax     # x3-x4
  cltq   # sign-extend %eax->%rax
  movslq 24(%rsp),%rdx # x2
  addq   16(%rsp),%rdx # x1+x2
  imulq  %rdx,%rax     # *
  addq   $32,%rsp
  ret
```



Return address to caller of `call_proc`

x4		x3	x2	24
x1				16
Arg 8				8
Arg 7				←%rsp

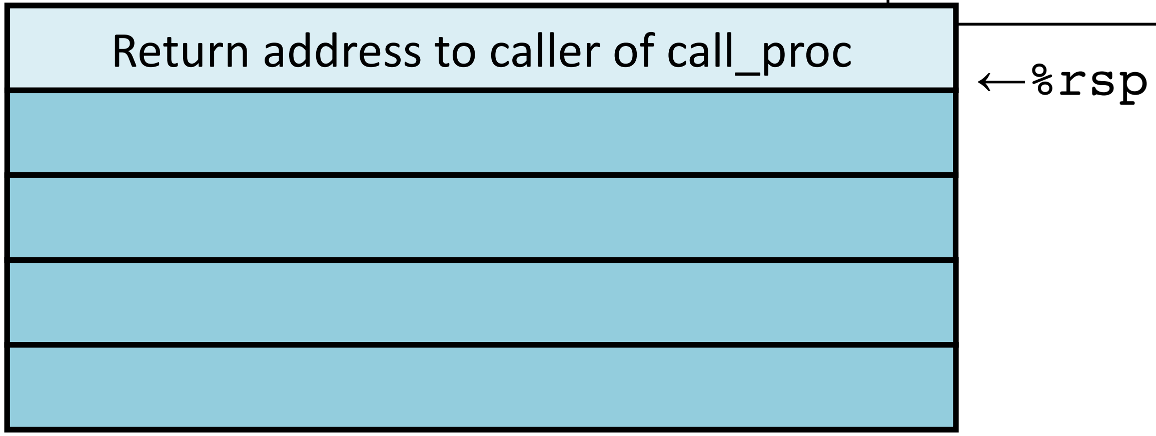
Stack storage example

(5) deallocate local vars

optional

```
long int call_proc()
{
  long  x1 = 1;
  int   x2 = 2;
  short x3 = 3;
  char  x4 = 4;
  proc(x1, &x1, x2, &x2,
       x3, &x3, x4, &x4);
  return (x1+x2)*(x3-x4);
}
```

```
call_proc:
  • • •
  movswl 28(%rsp),%eax
  movsbl 31(%rsp),%edx
  subl   %edx,%eax
  cltq
  movslq 24(%rsp),%rdx
  addq   16(%rsp),%rdx
  imulq  %rdx,%rax
  addq   $32,%rsp
  ret
```



Procedure Summary

`call, ret, push, pop`

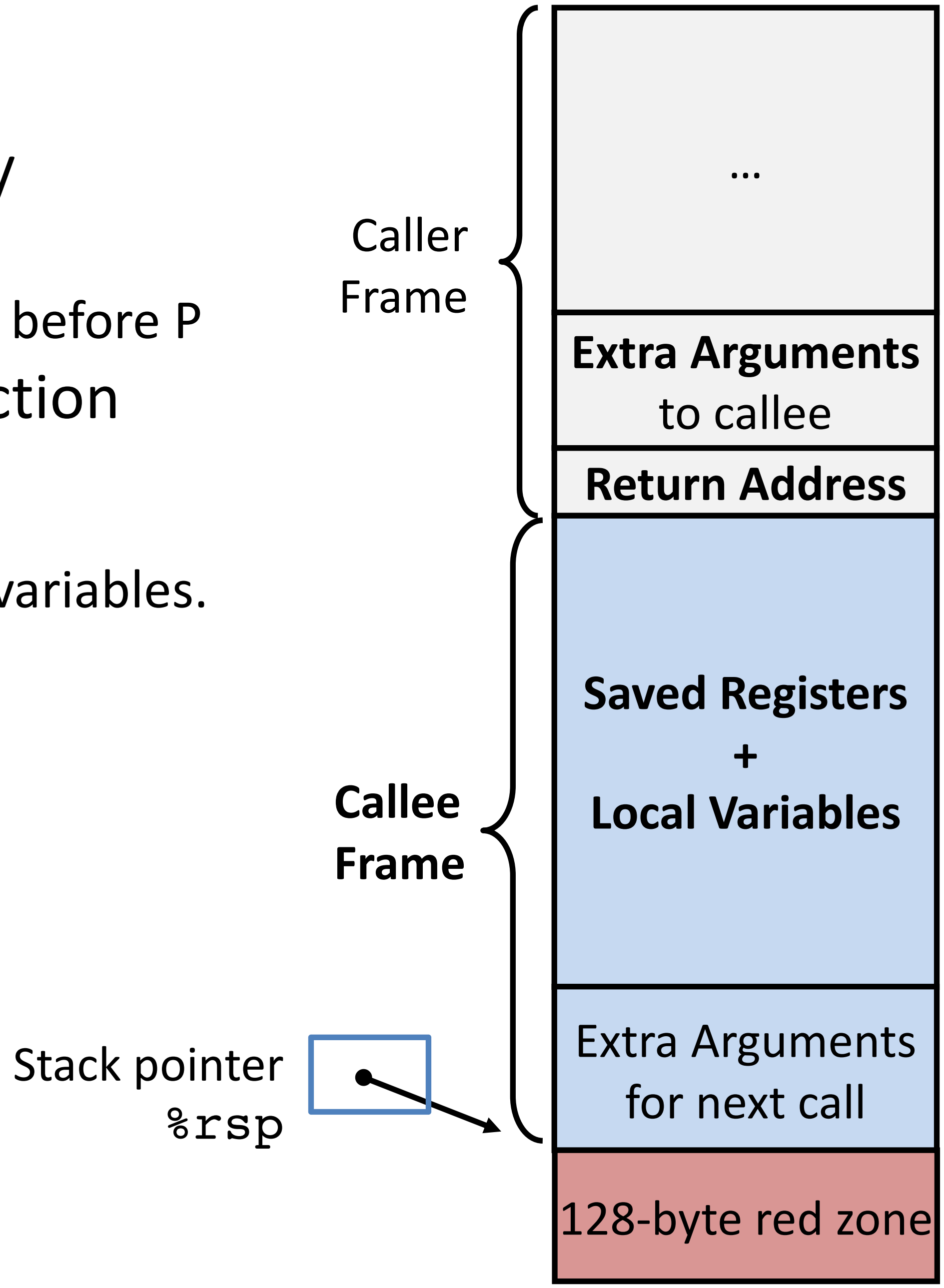
Stack discipline fits procedure call / return.*

If P calls Q: Q (and calls by Q) returns before P

Conventions support arbitrary function calls.

Register-save conventions.
Stack frame saves extra args or local variables.
Result returned in `%rax`

`%rax`	Return value – Caller saved	`%r8`	Argument #5 – Caller saved
`%rbx`	Callee saved	`%r9`	Argument #6 – Caller saved
`%rcx`	Argument #4 – Caller saved	`%r10`	Caller saved
`%rdx`	Argument #3 – Caller saved	`%r11`	Caller Saved
`%rsi`	Argument #2 – Caller saved	`%r12`	Callee saved
`%rdi`	Argument #1 – Caller saved	`%r13`	Callee saved
`%rsp`	Stack pointer	`%r14`	Callee saved
`%rbp`	Callee saved	`%r15`	Callee saved


Caller Frame
…
Extra Arguments to callee
Return Address
Callee Frame
Saved Registers + Local Variables
Extra Arguments for next call
128-byte red zone
Stack pointer %rsp