



CS 240

Foundations of Computer Systems



x86: Procedures and the Call Stack

The call stack discipline

x86 procedure call and return instructions

x86 calling conventions

x86 register-saving conventions

x86: Procedures and the Call Stack

Outline

1. Motivation
 - a. *(video 1)* What we have seen so far
 - b. *(video 1)* Why we can't implement procedure calls with jumps alone
2. *(video 1)* High-level call stack example
3. Procedure control flow instructions: call and ret
4. Procedure call example (in depth!) on whiteboard
5. Caller vs/callee example
6. *(Covered in lab, video)* Recursion example

Does understanding the call stack really matter?



Alexa VanHattum 9:31 AM

looking at popcnt now!

**Recent research chat on Slack with
PhD student collaborator**

```
avh veri/veri - (popcnt-expand)$ ./script/veri.sh -- --rule popcnt_8
Blocking waiting for file lock on build directory
Compiling cranelift-isle-veri v0.1.0
Writing generated file:
/var/folders/9r/4bqb01xs60b8kpv59bk68cpc0000gn/T/tmp.AsPxX054/clif_lower.is
le
#990      popcnt_8
```

```
thread 'main' has overflowed its stack
fatal runtime error: stack overflow
```

Yes, the call stack comes up in
large-scale software
engineering/research!



mbm 9:36 AM

There's an environment variable you can set to increase stack size. The default is not that big.



Alexa VanHattum 10:49 AM

process **And** with a loop instead of recursion

Why procedures?

Why functions? Why methods?

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

Answer: procedural abstraction

Implementing procedures

Have we already seen
how this is done?

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

Procedure call/return: Jump?

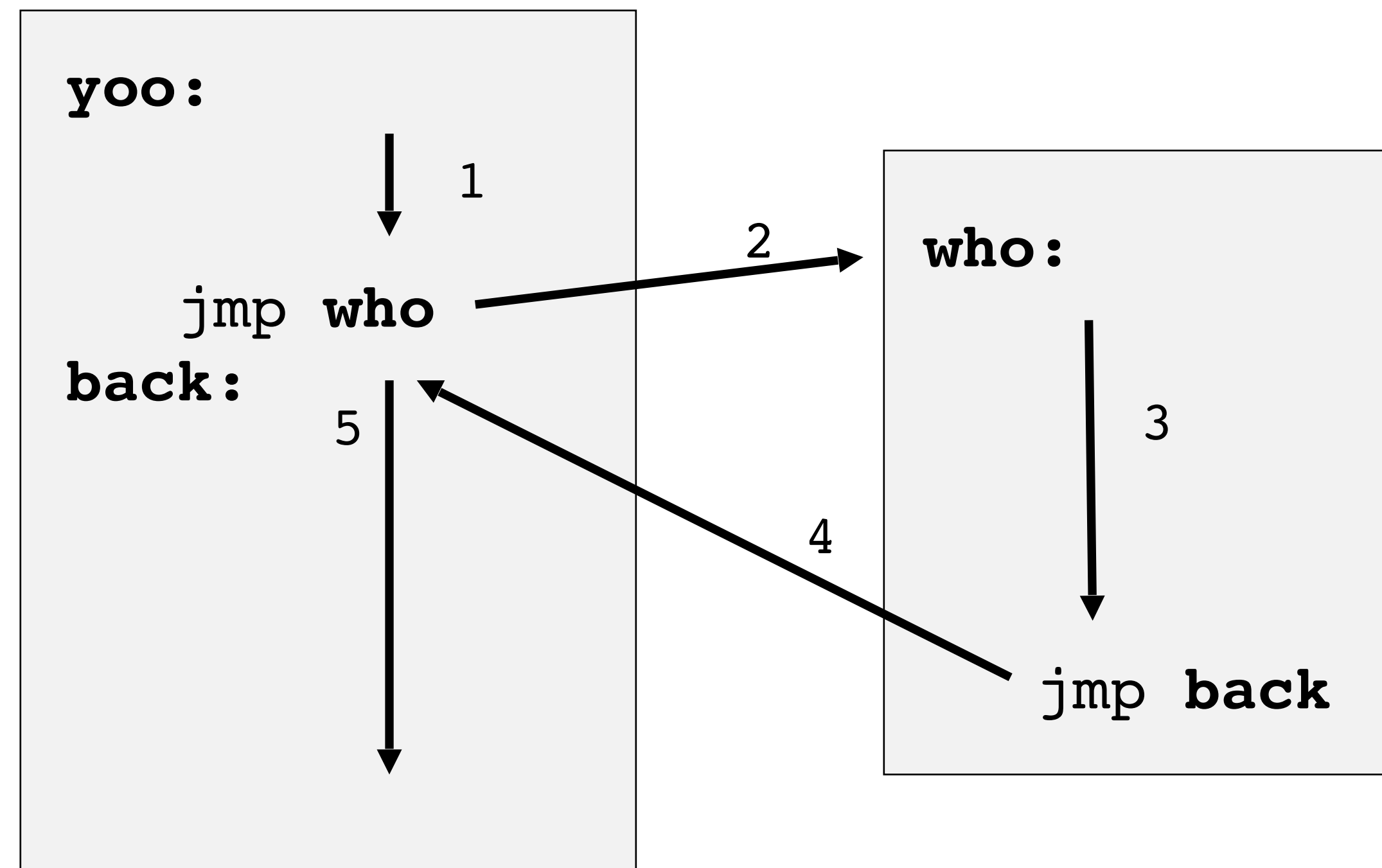
```
yoo (...) {  
  . . .  
  who ( ) ;  
  . . .  
}
```

```
who (...) {  
  . . .  
  . . .  
  . . .  
}
```

```
ru (...) {  
  . . .  
}
```

Call Chain

yoo
↓
who



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

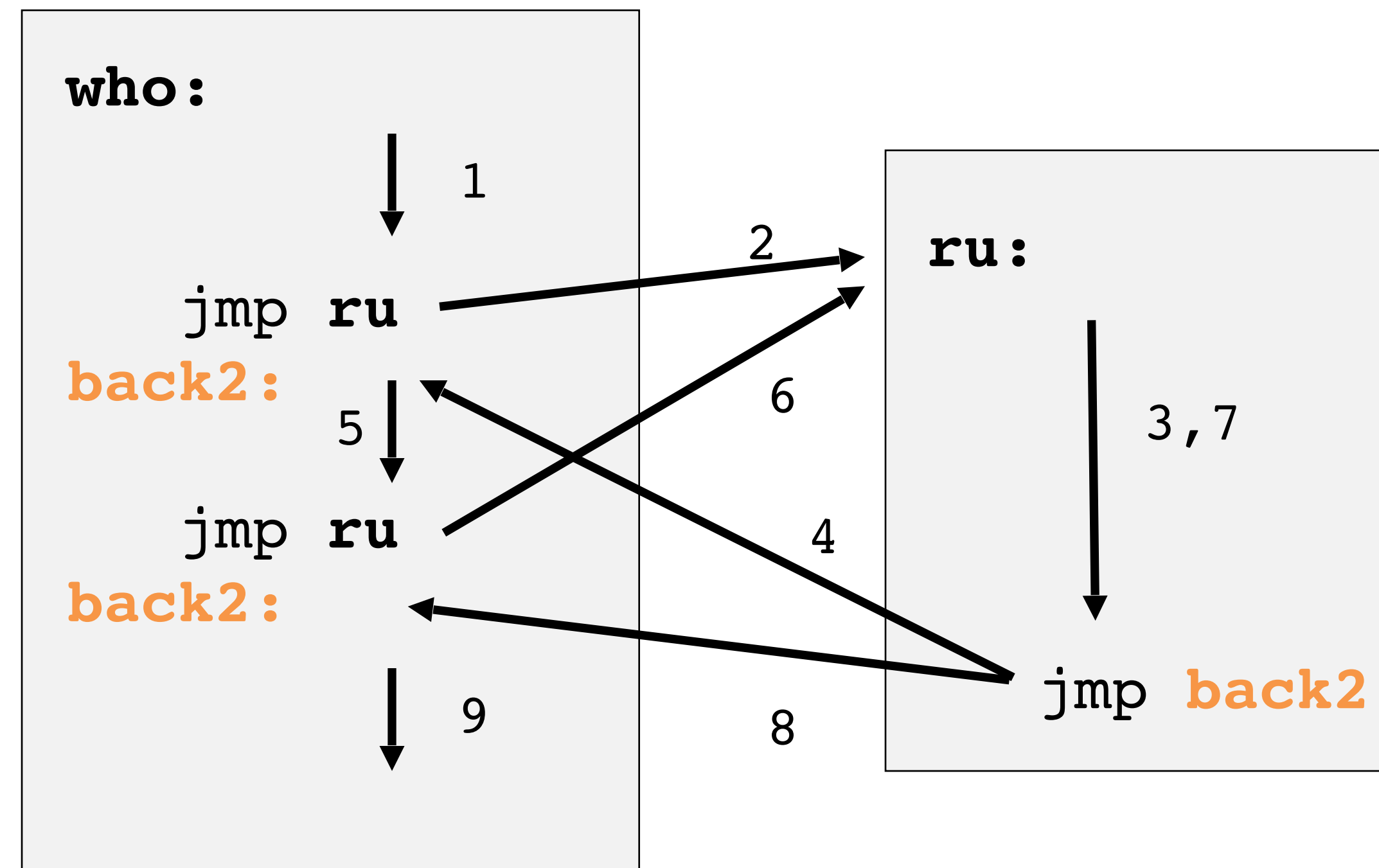
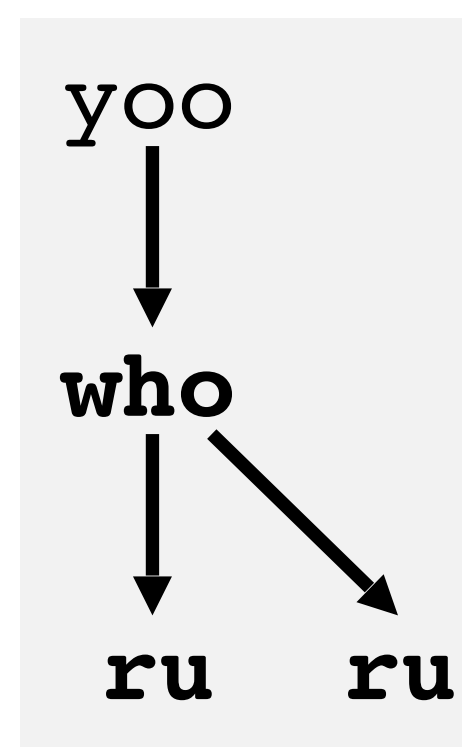
Procedure call/return: Jump? **Broken!**

```
yoo(...) {  
  . . .  
  who( );  
  . . .  
}
```

```
who(...) {  
  . . .  
  ru( );  
  . . .  
  ru( );  
  . . .  
}
```

```
ru(...) {  
  . . .  
}
```

Call Chain



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)

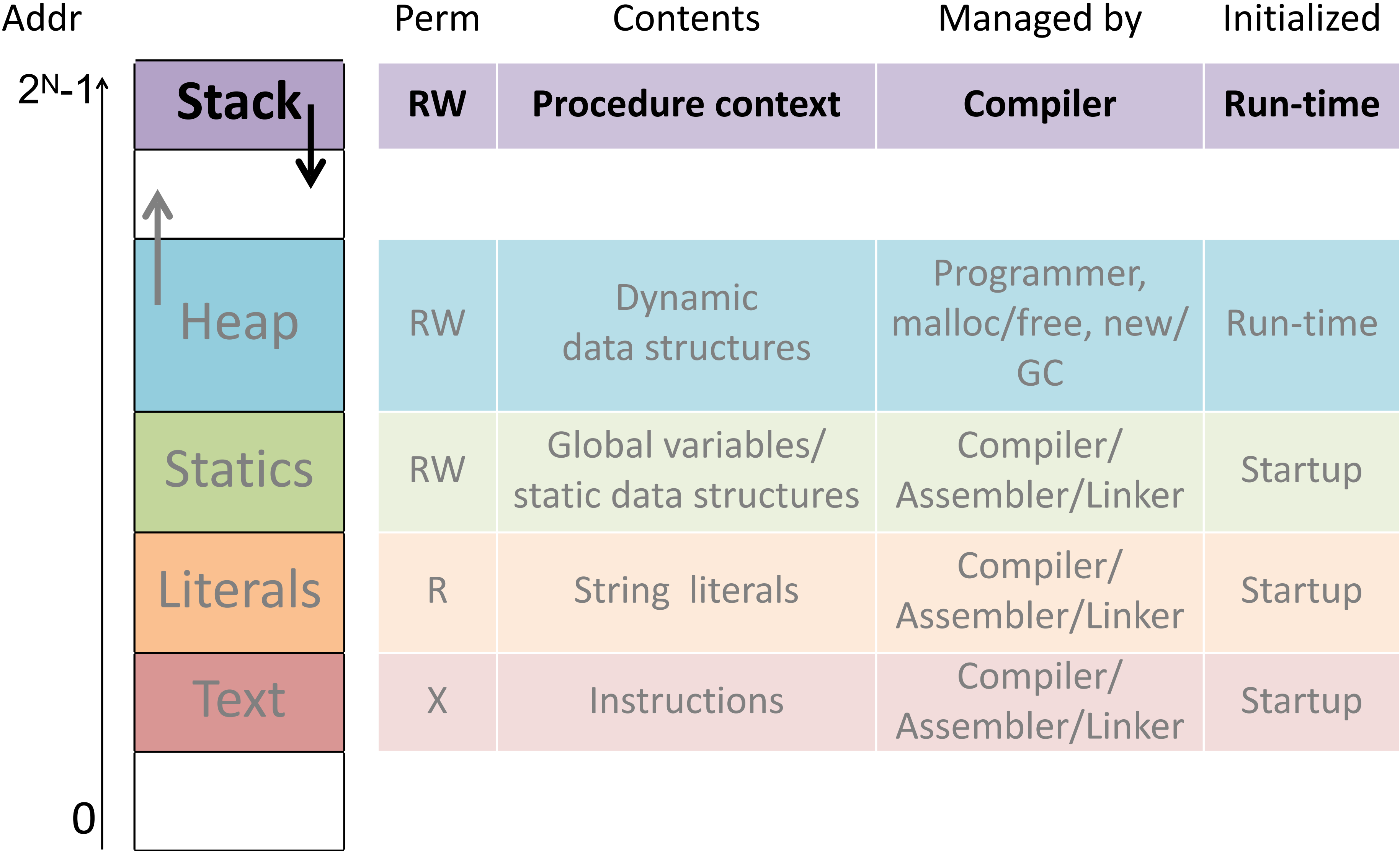
Have we already seen
how this is done?

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?

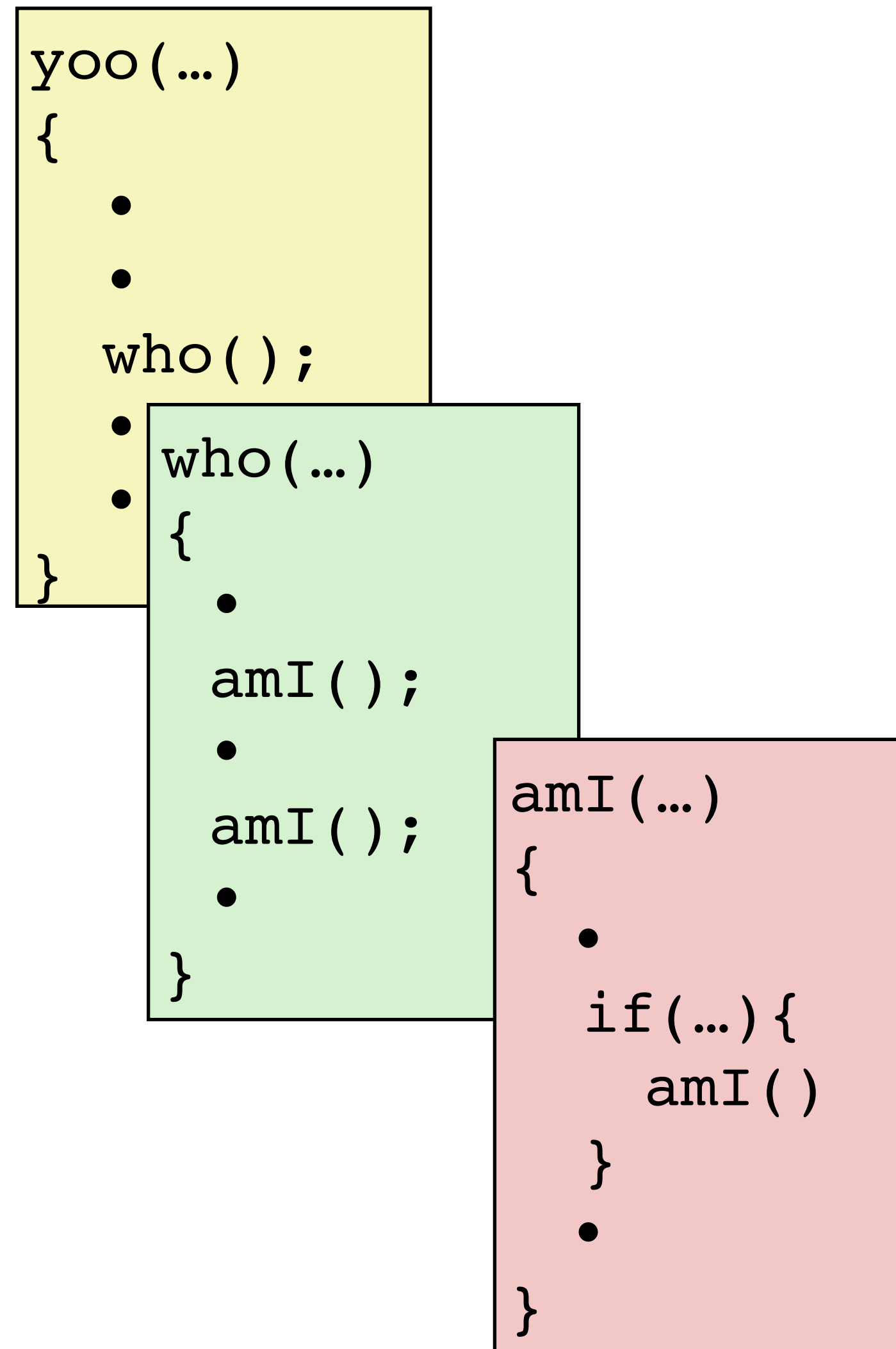


Memory Layout

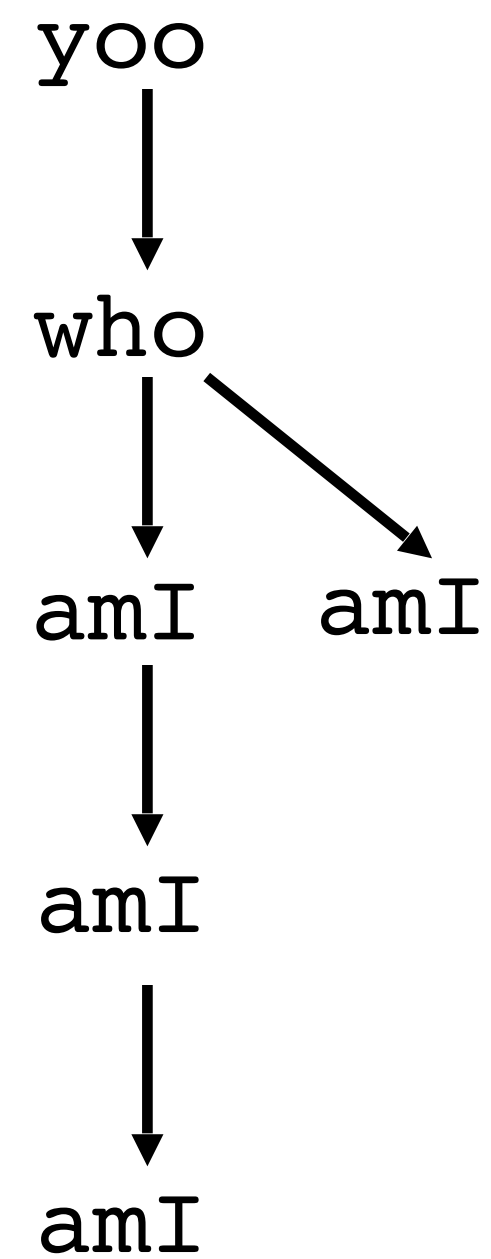
reminder



Call stack tracks context

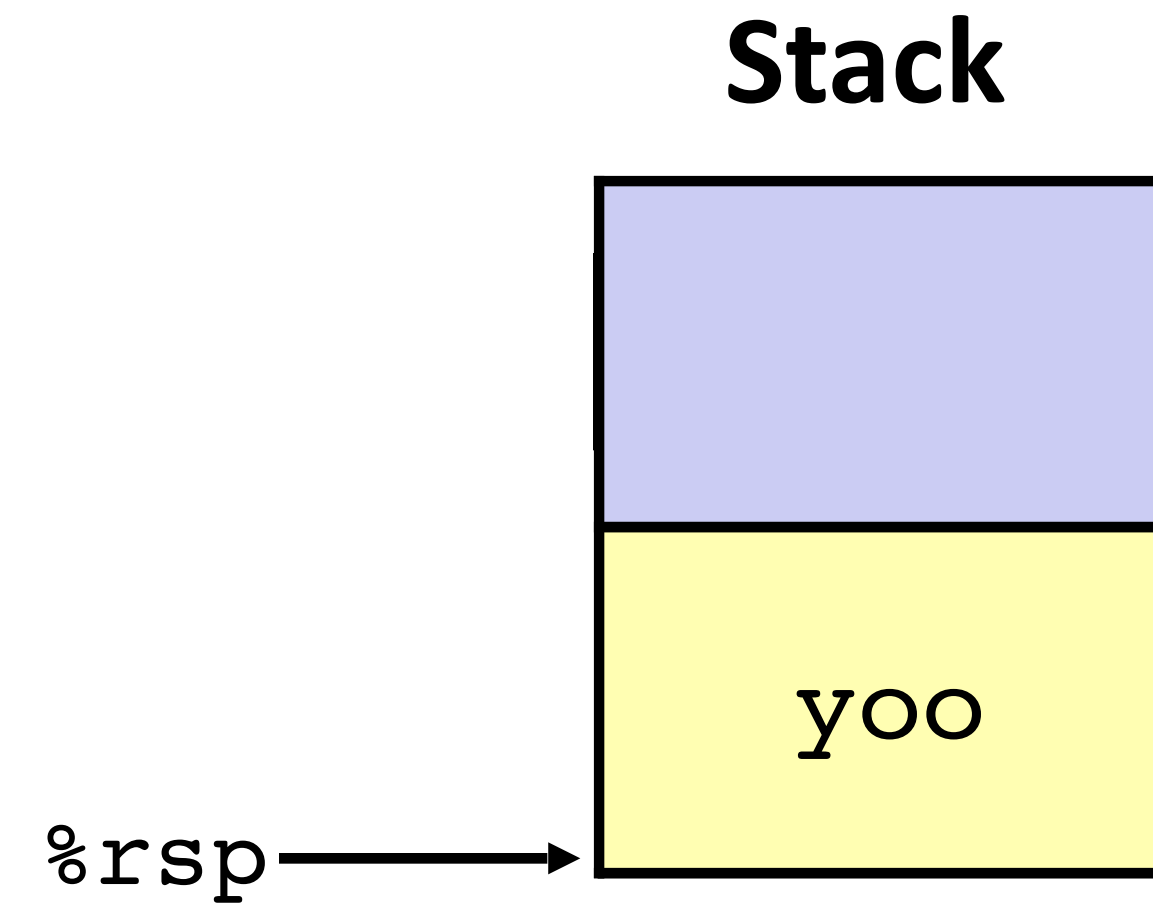
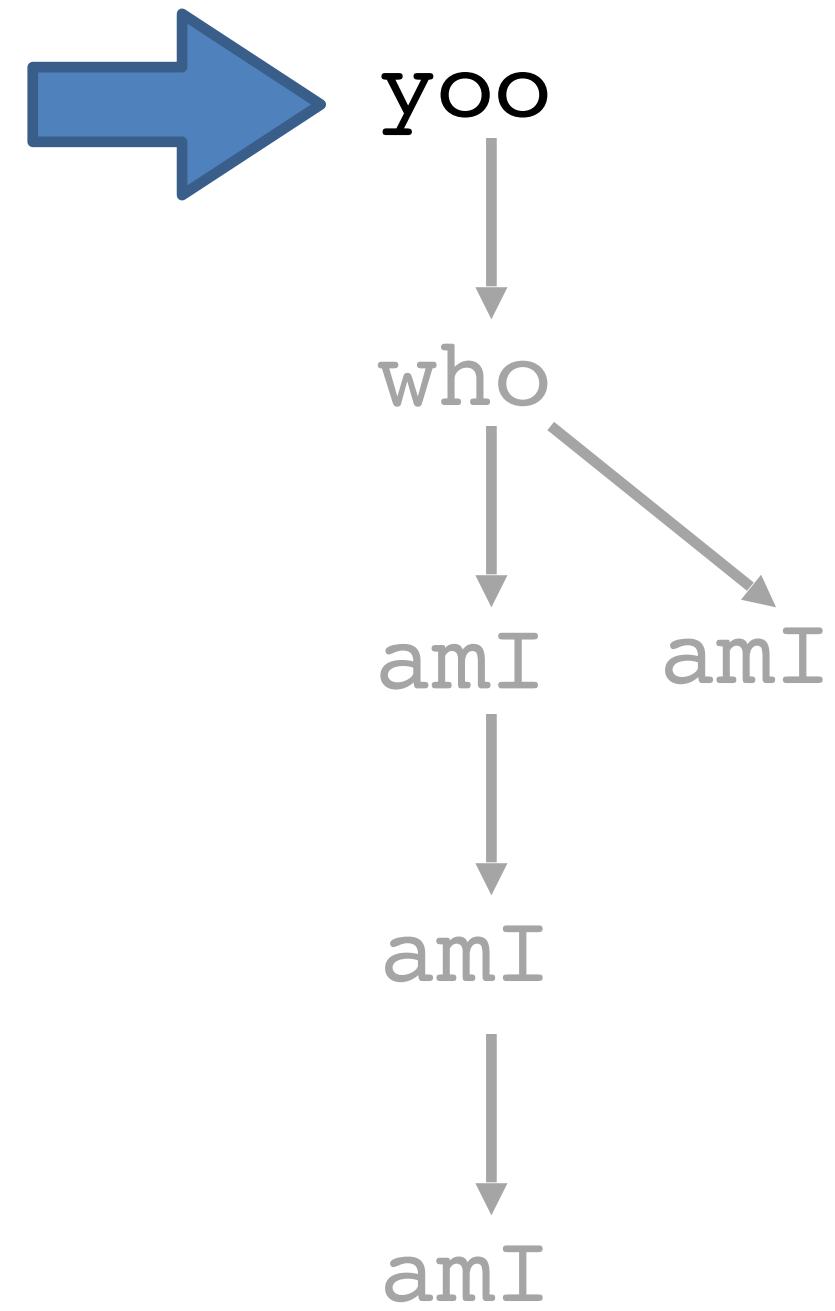
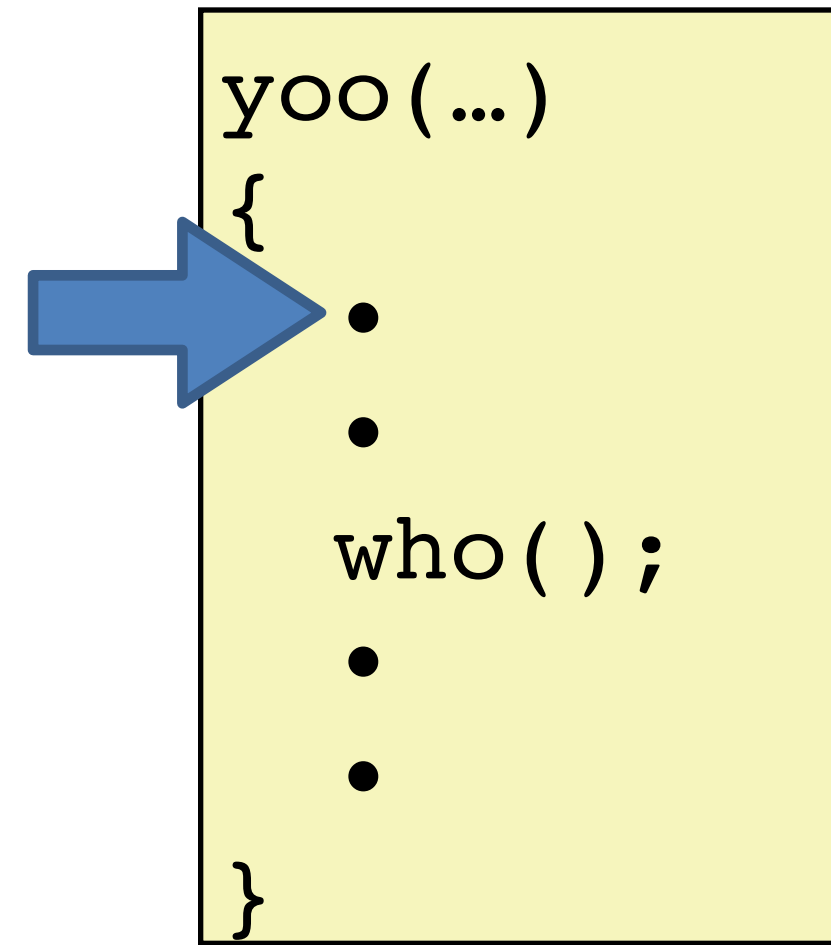


Example
Call Chain

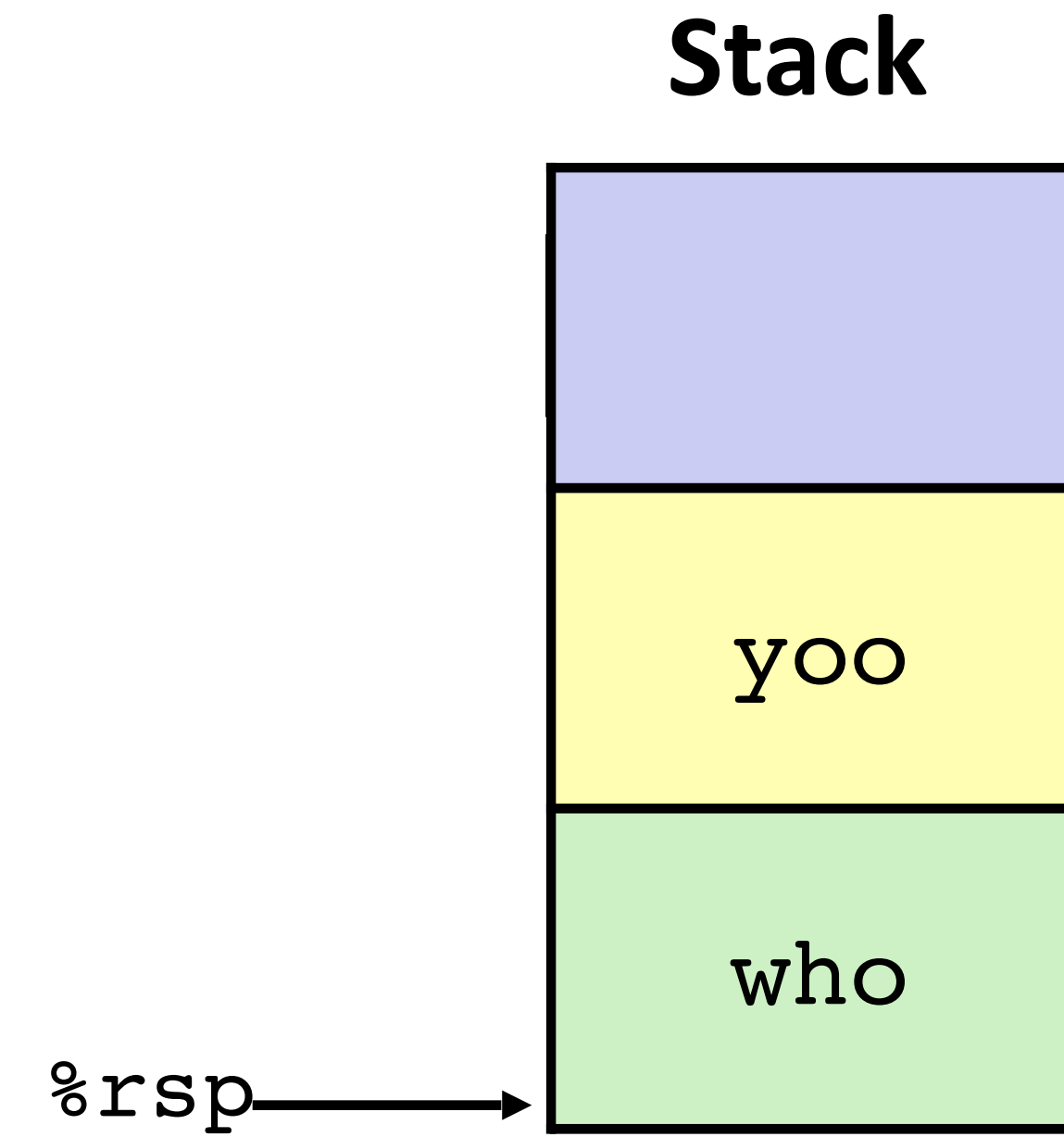
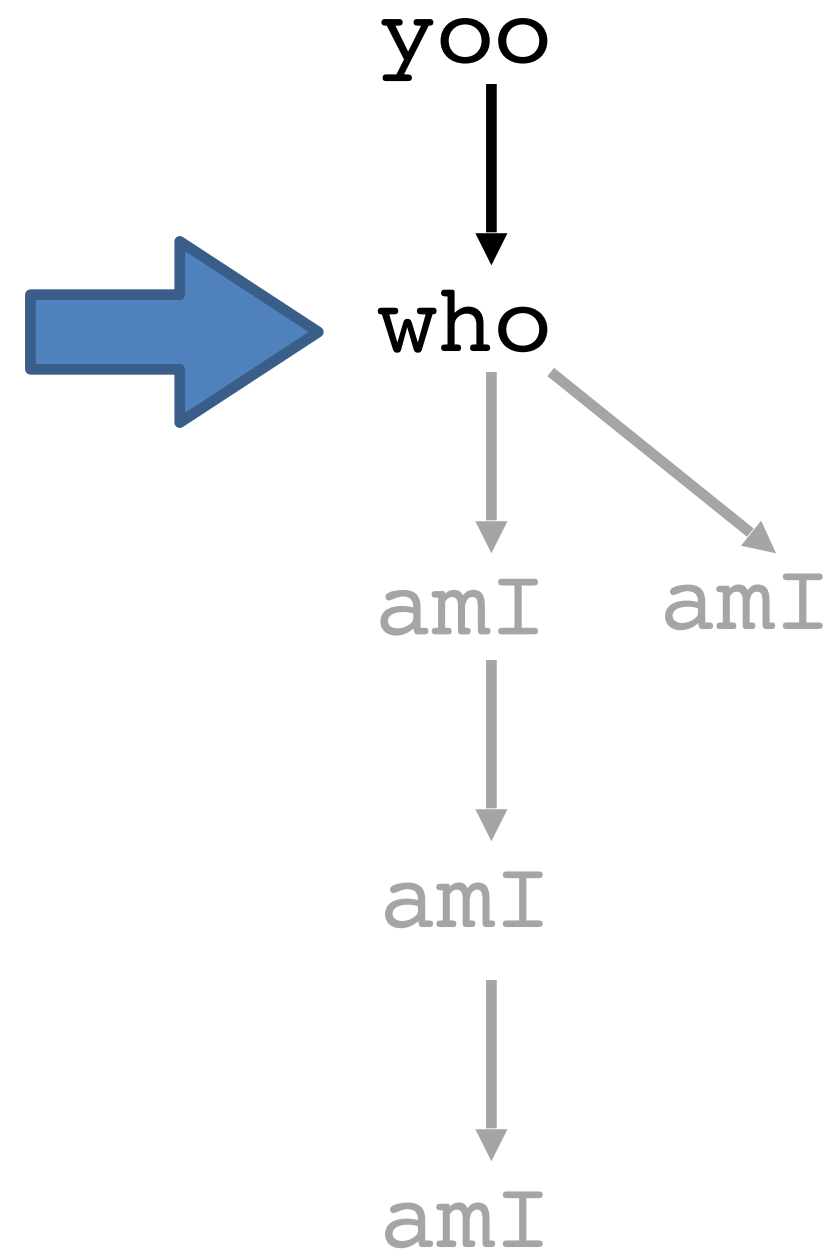
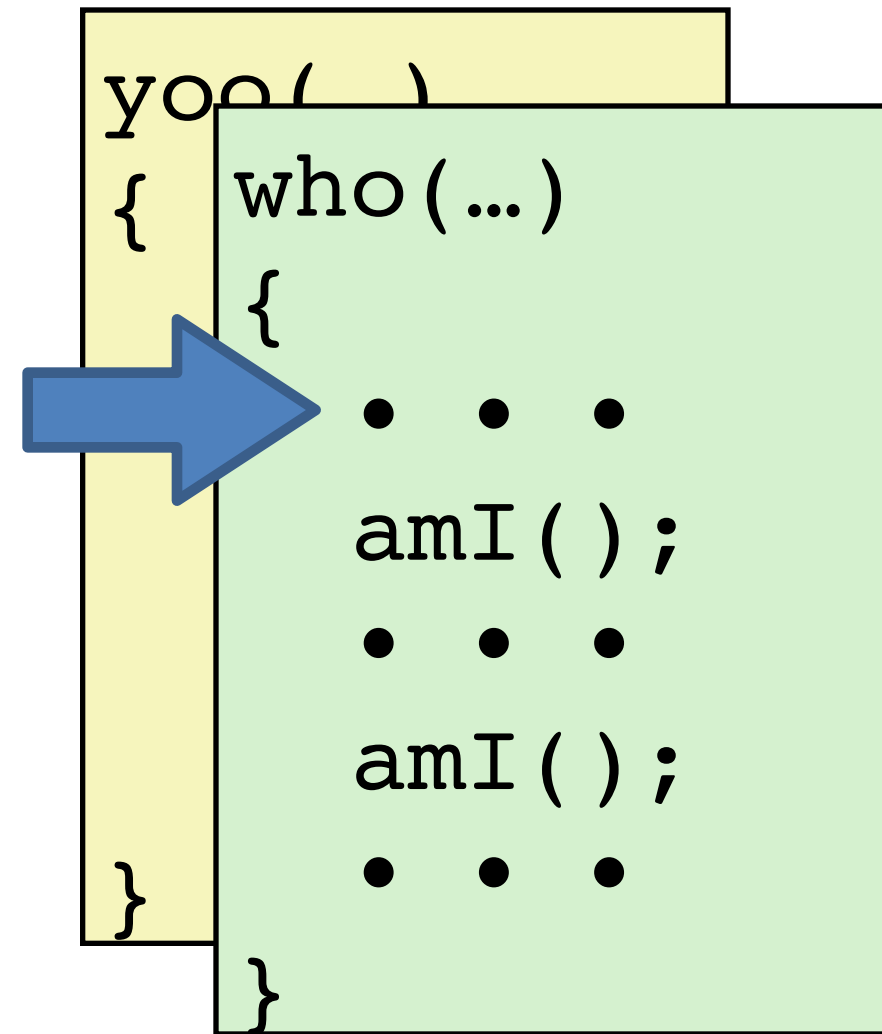


Procedure amI is recursive
(calls itself)

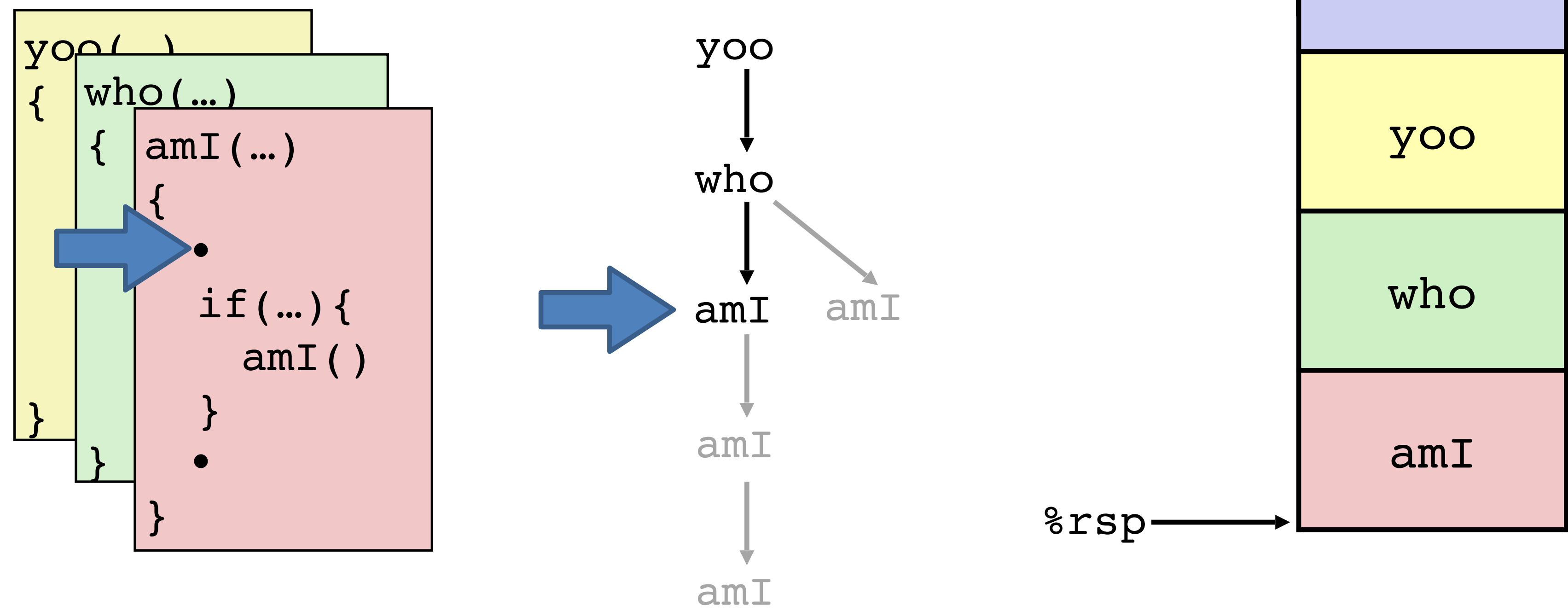
Call stack tracks context



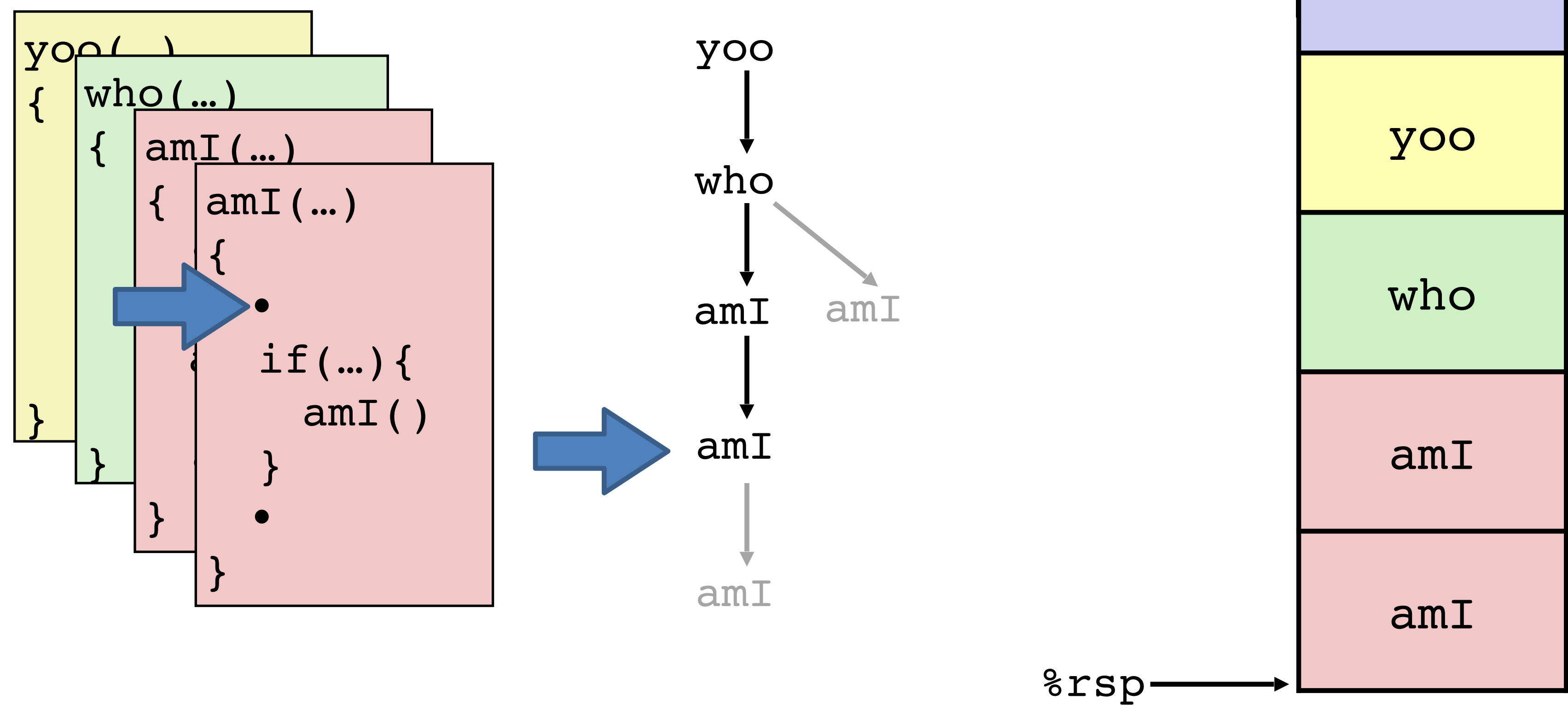
Call stack tracks context



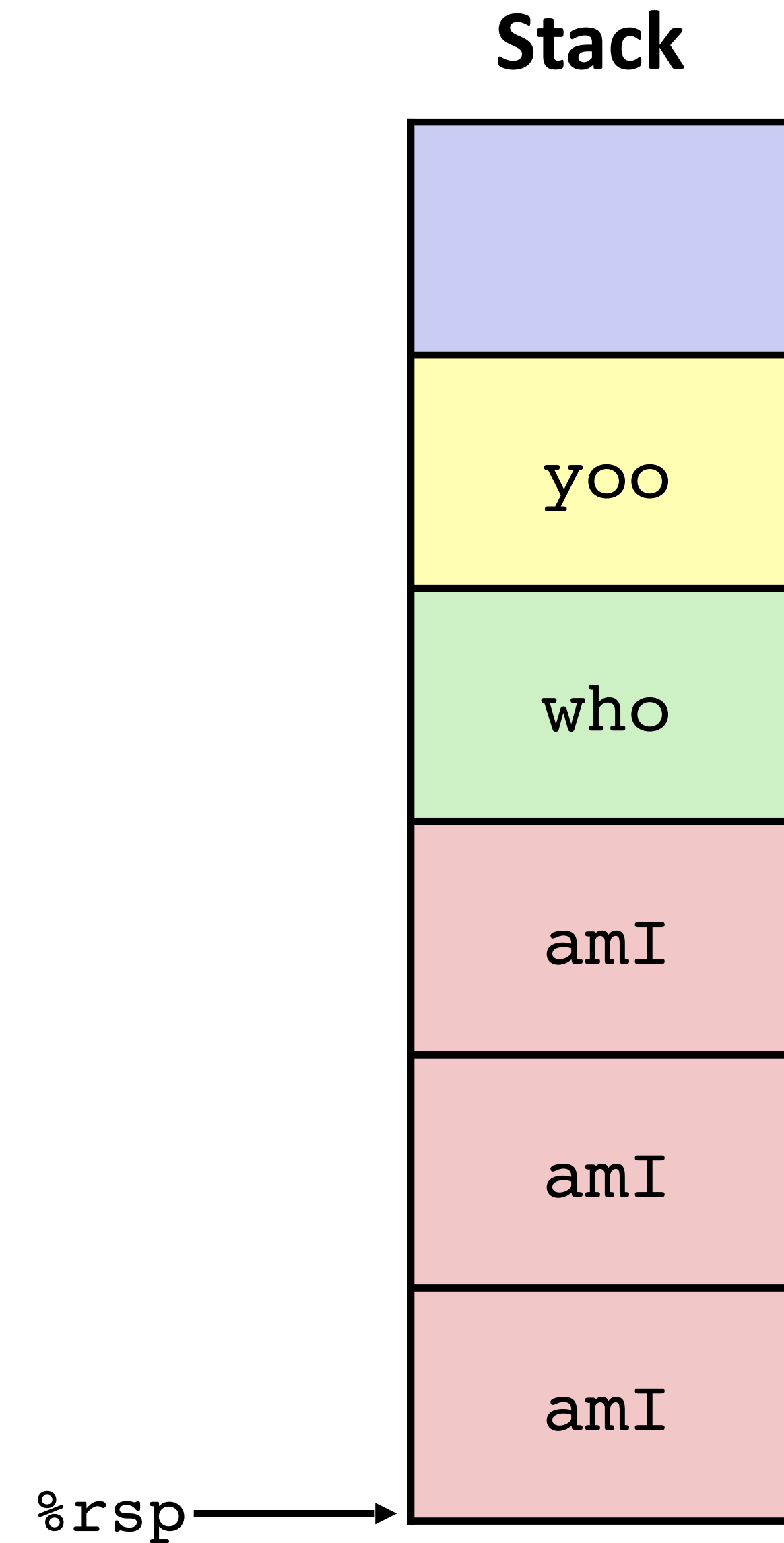
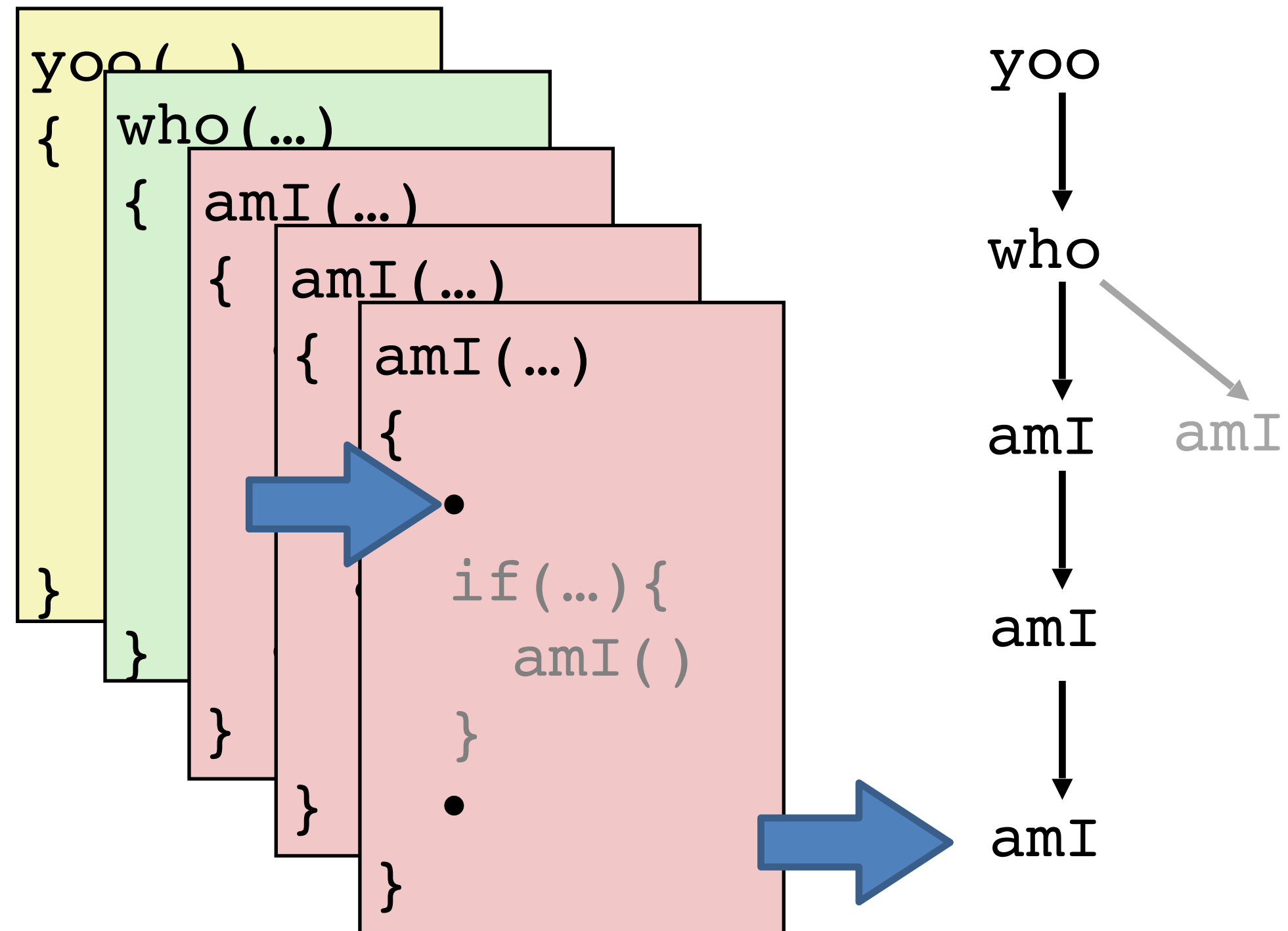
Call stack tracks context



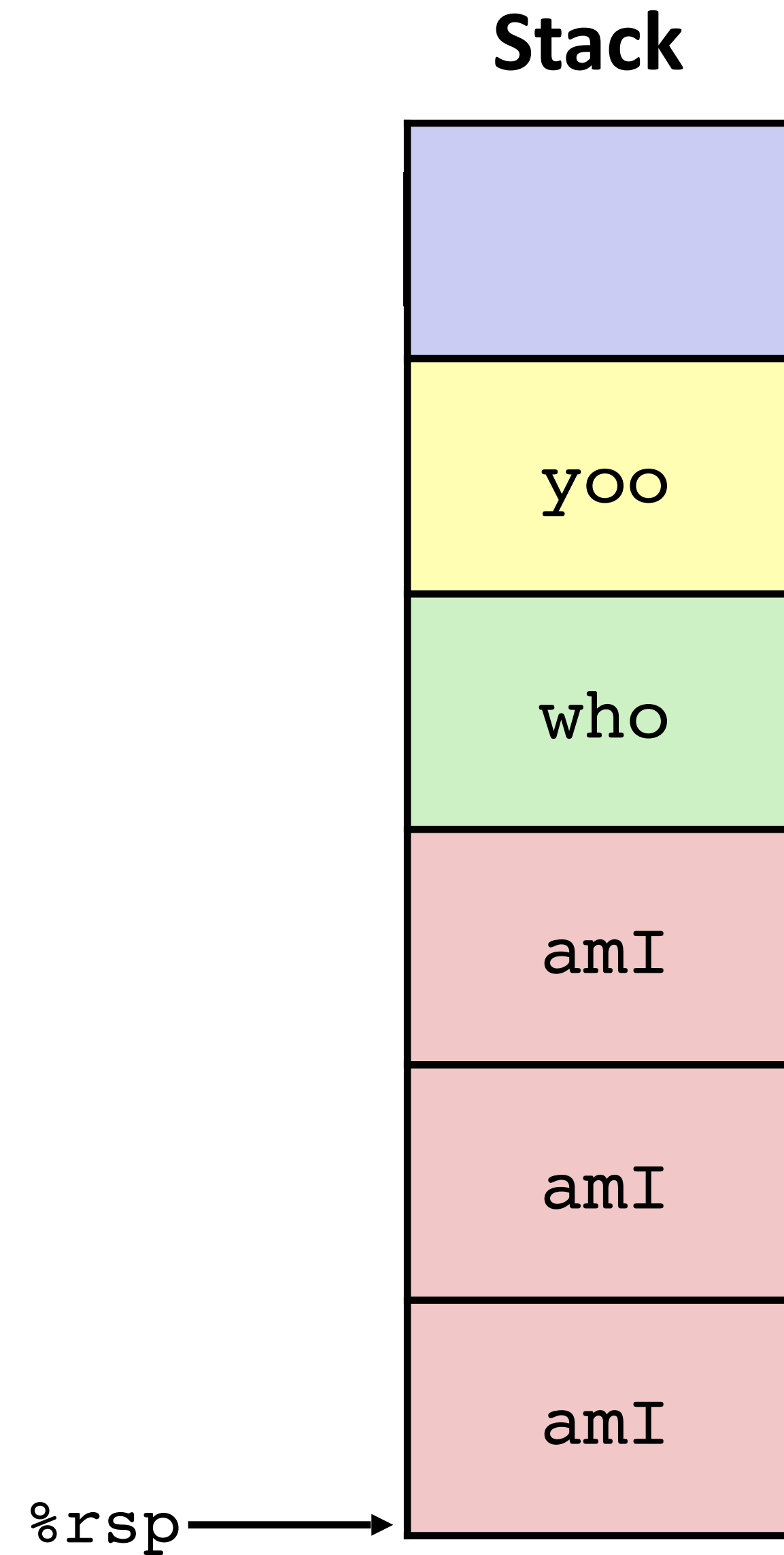
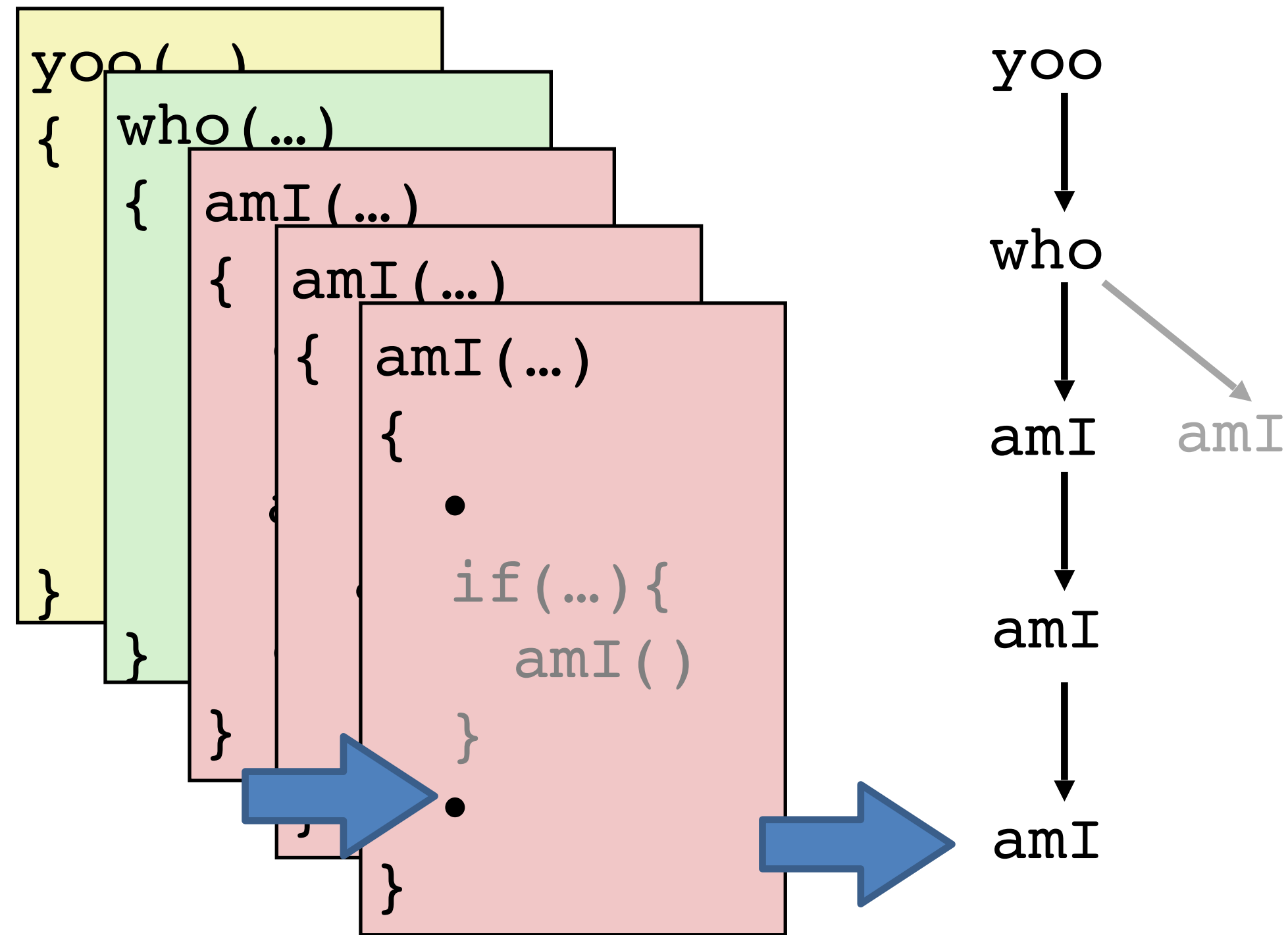
Call stack tracks context



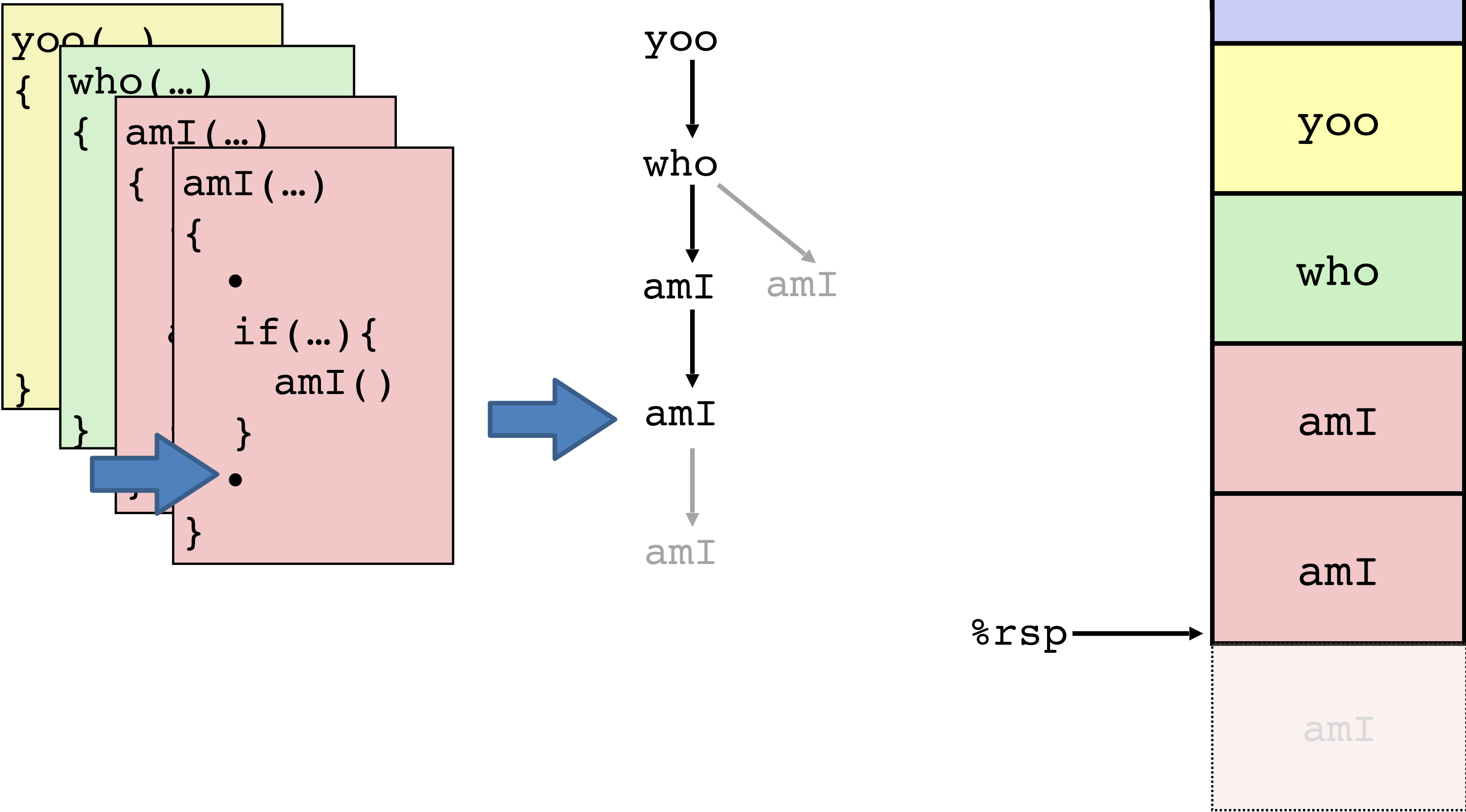
Call stack tracks context



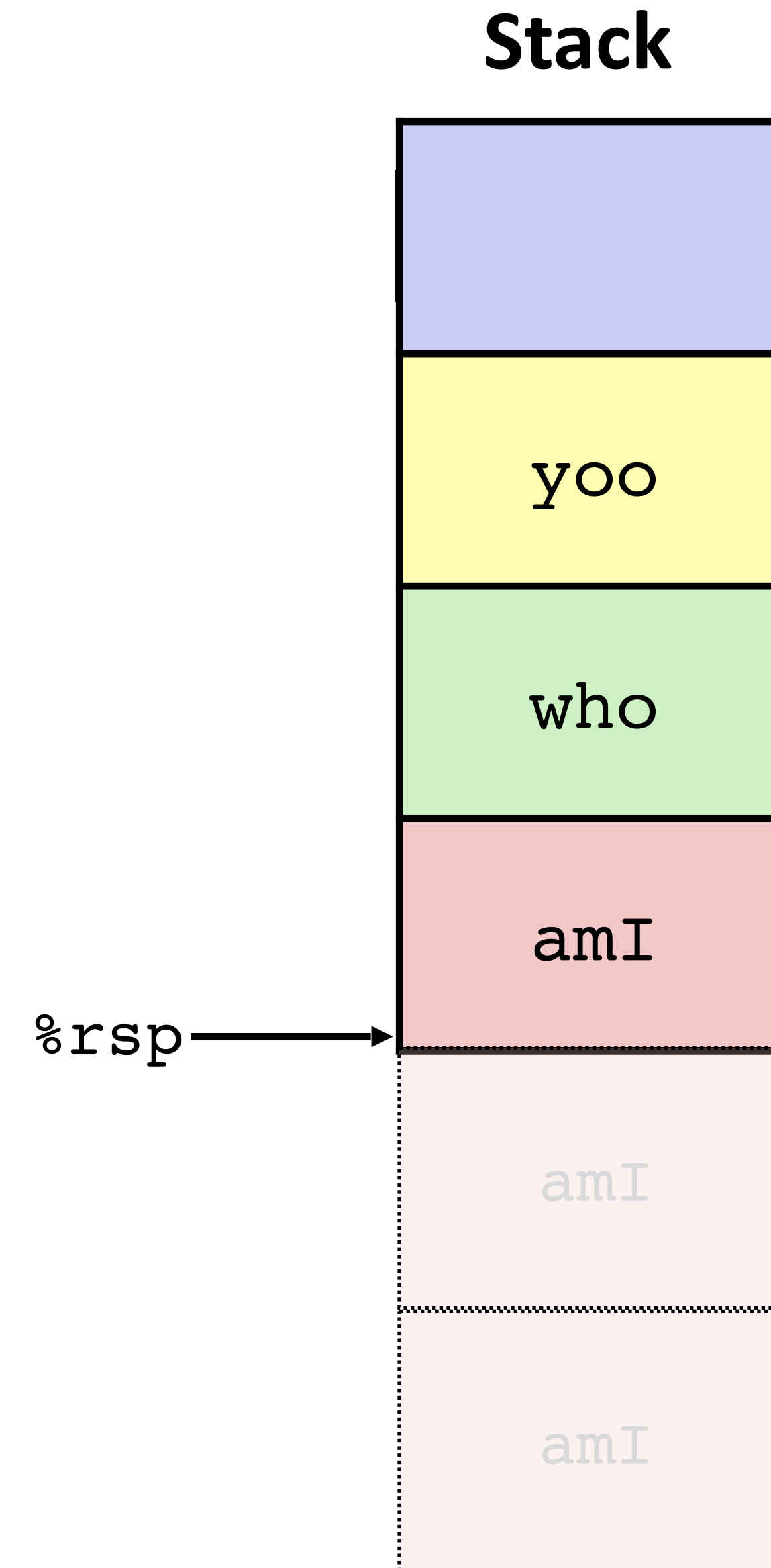
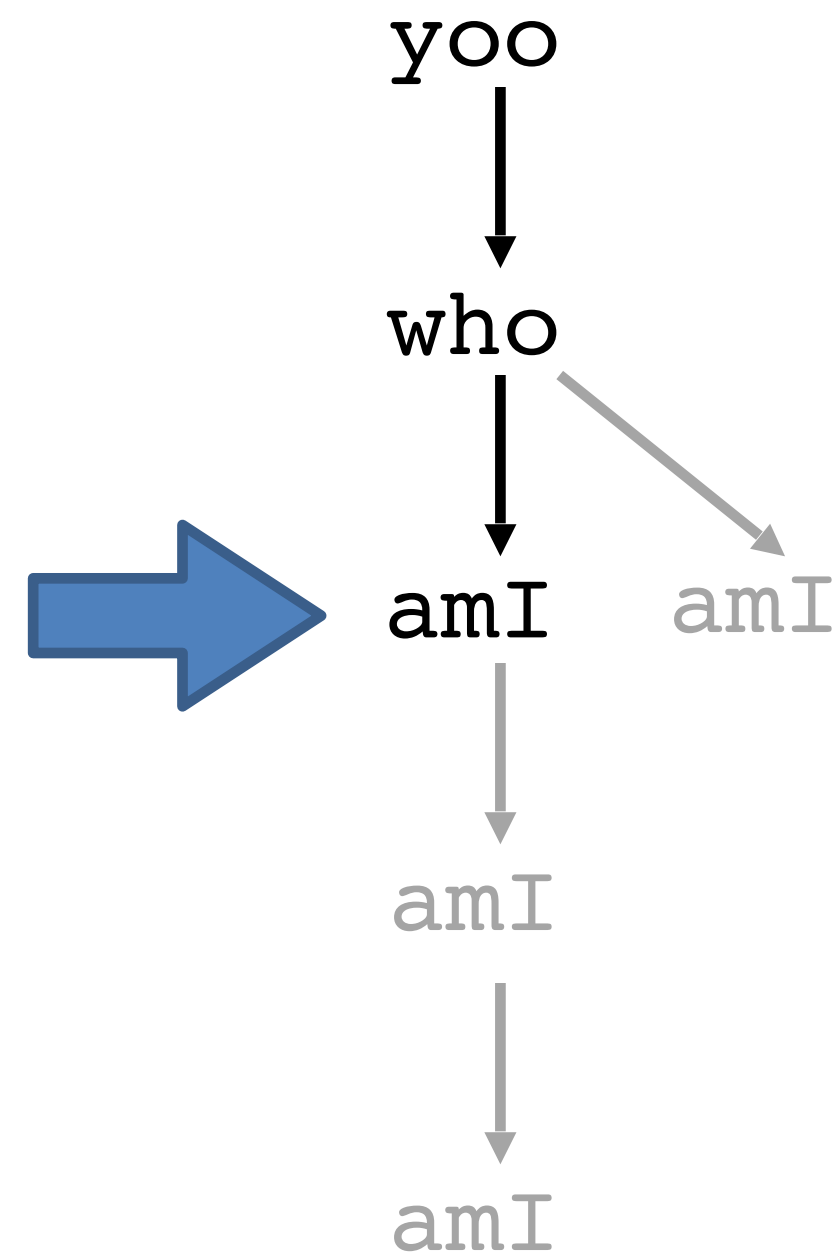
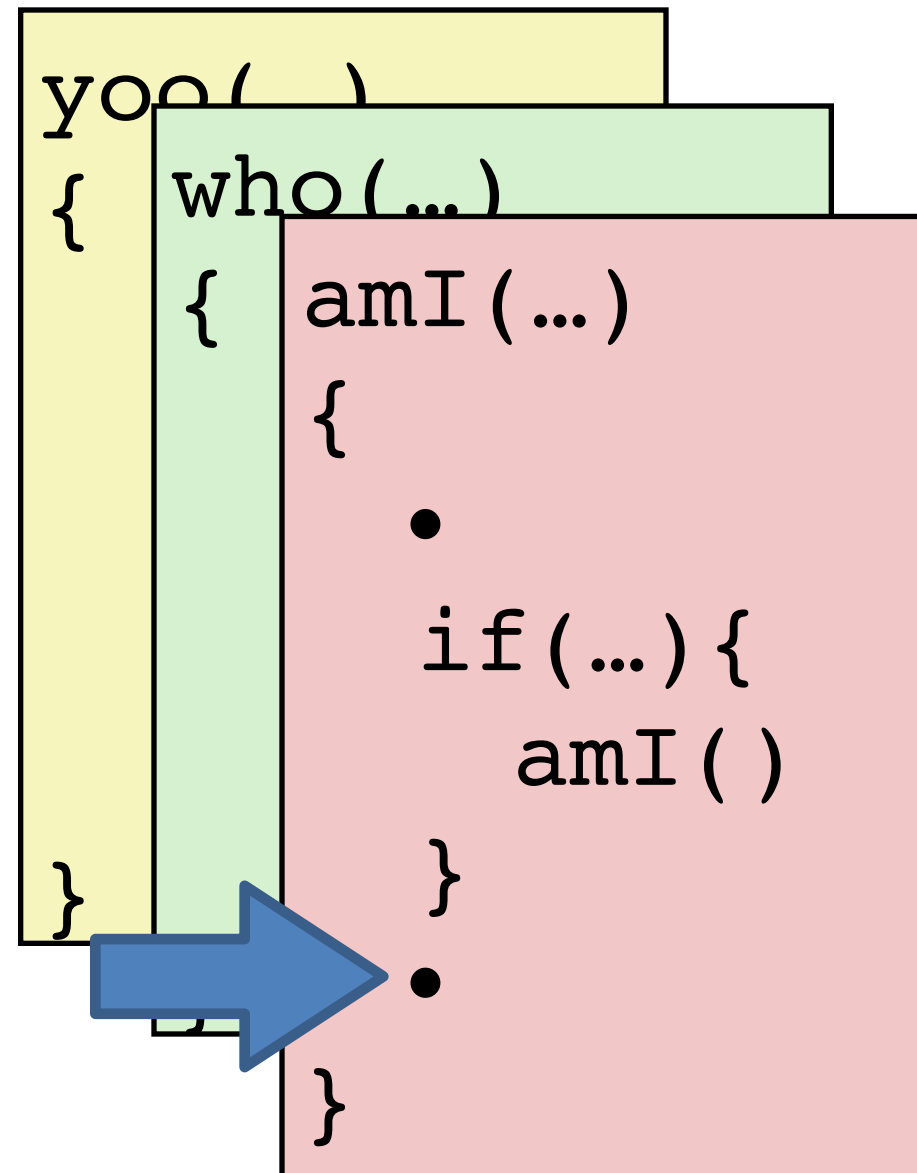
Call stack tracks context



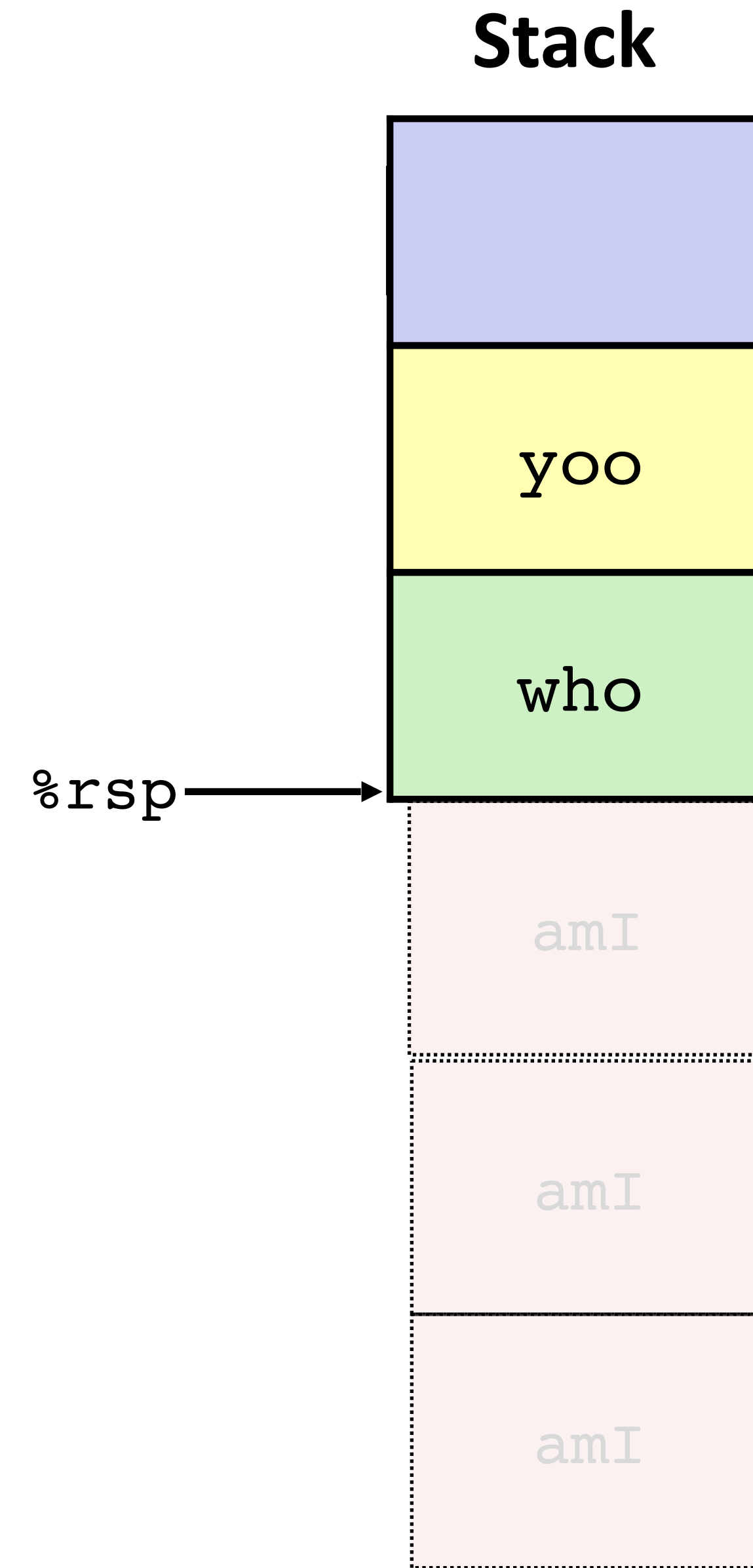
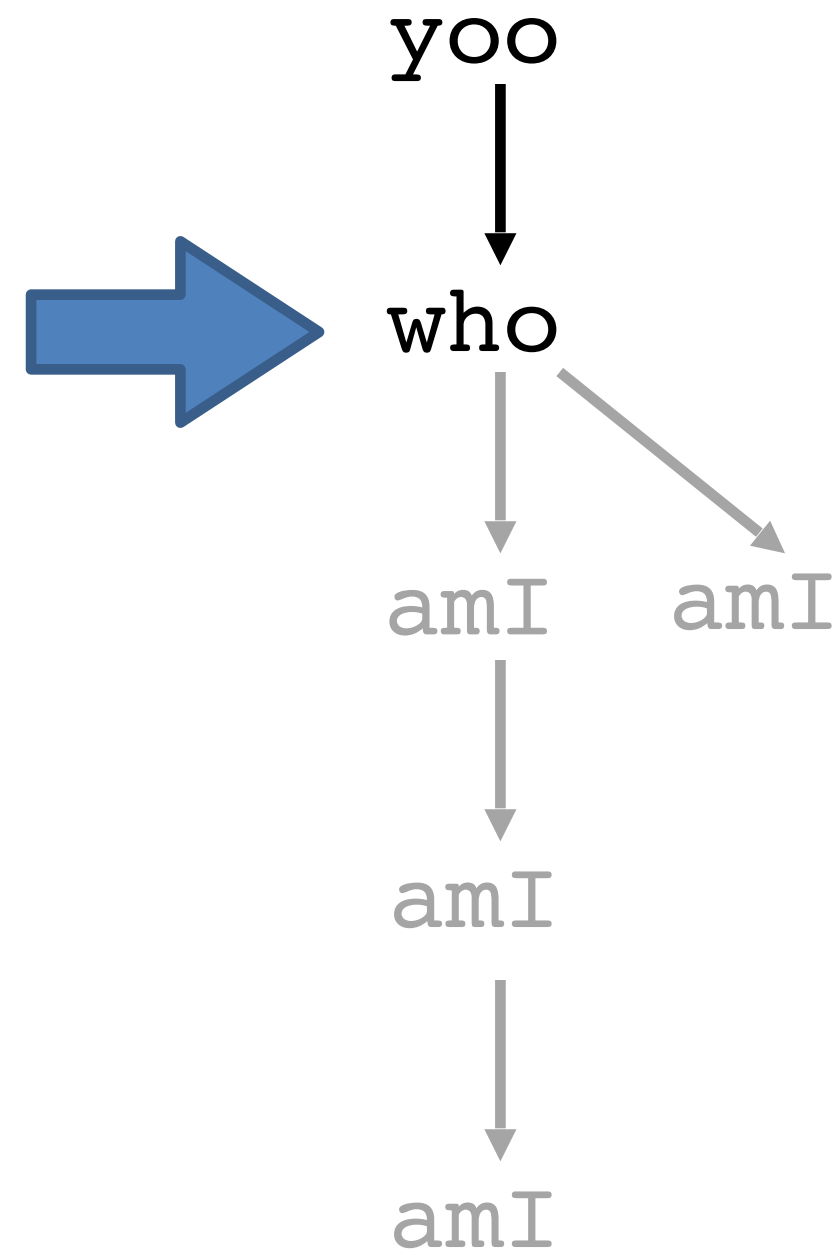
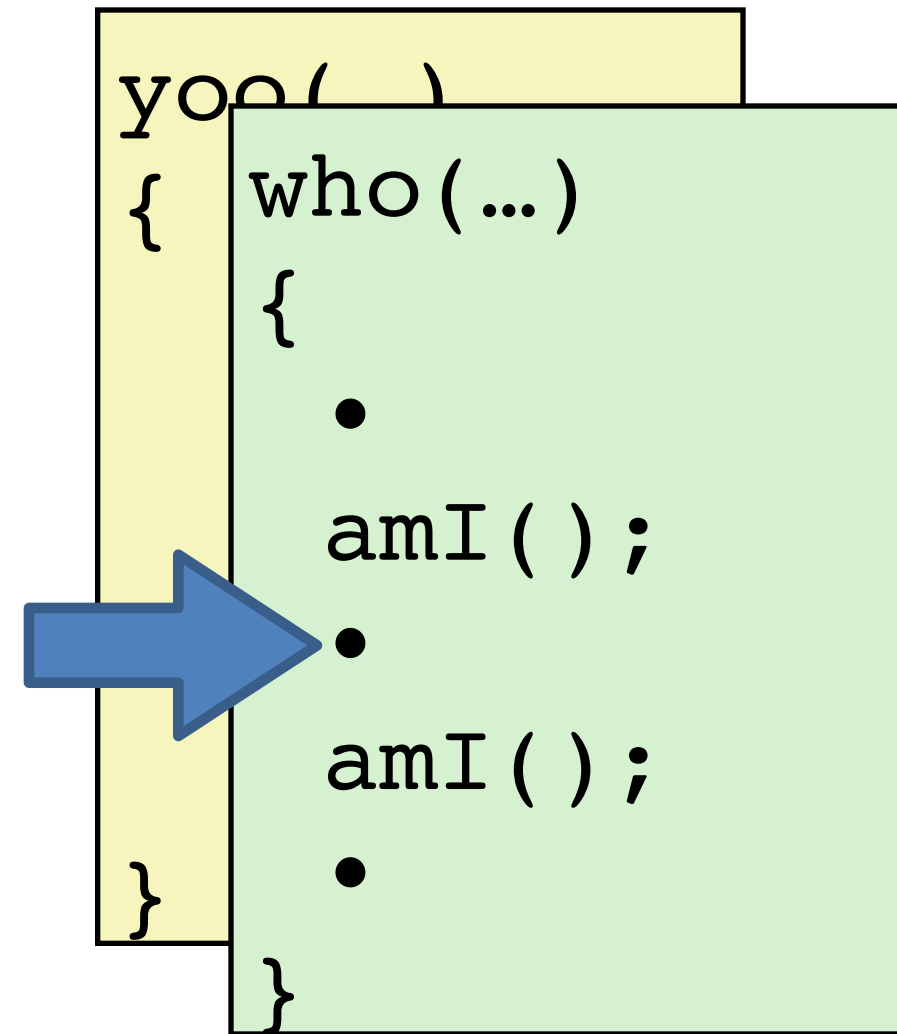
Call stack tracks context



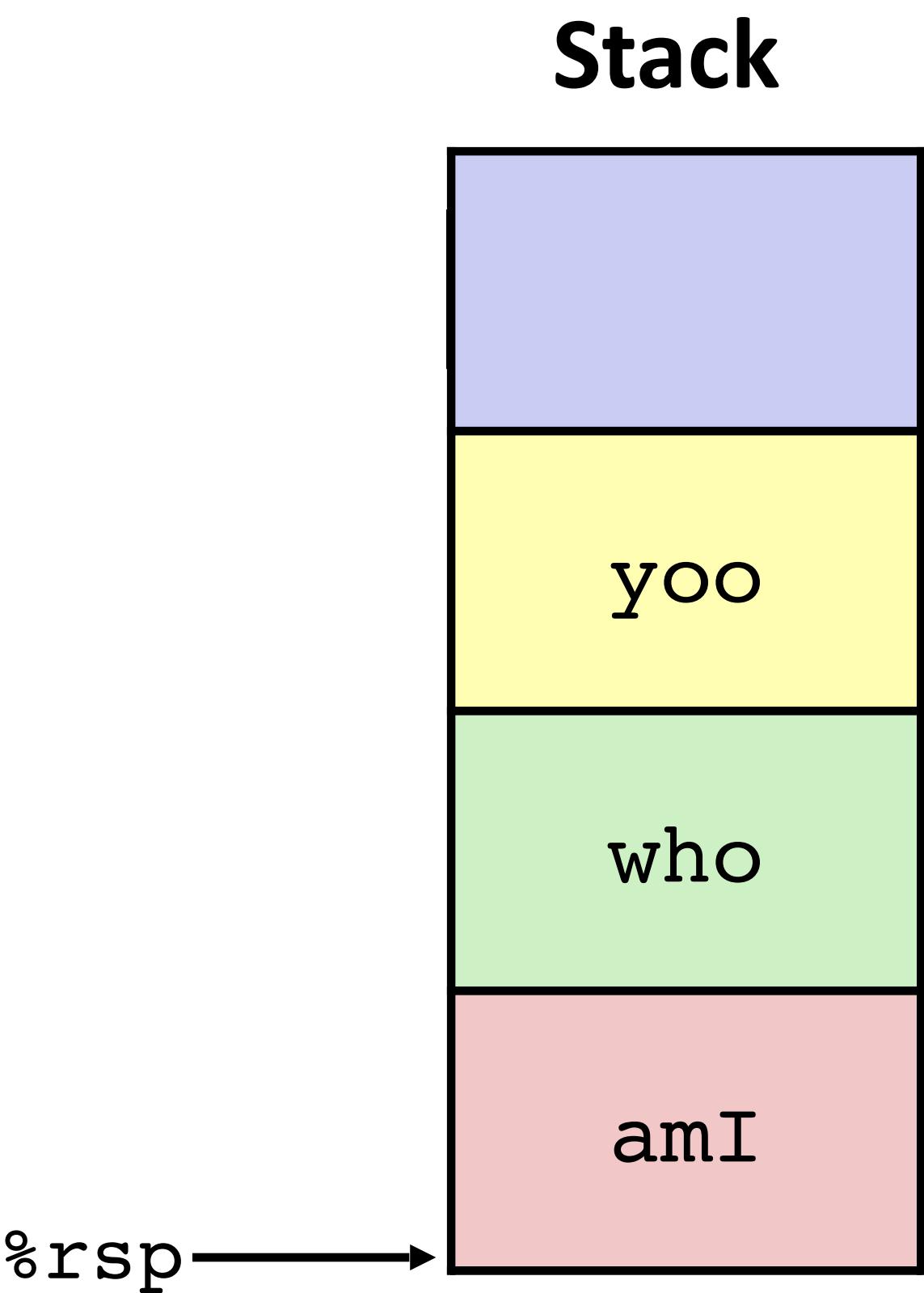
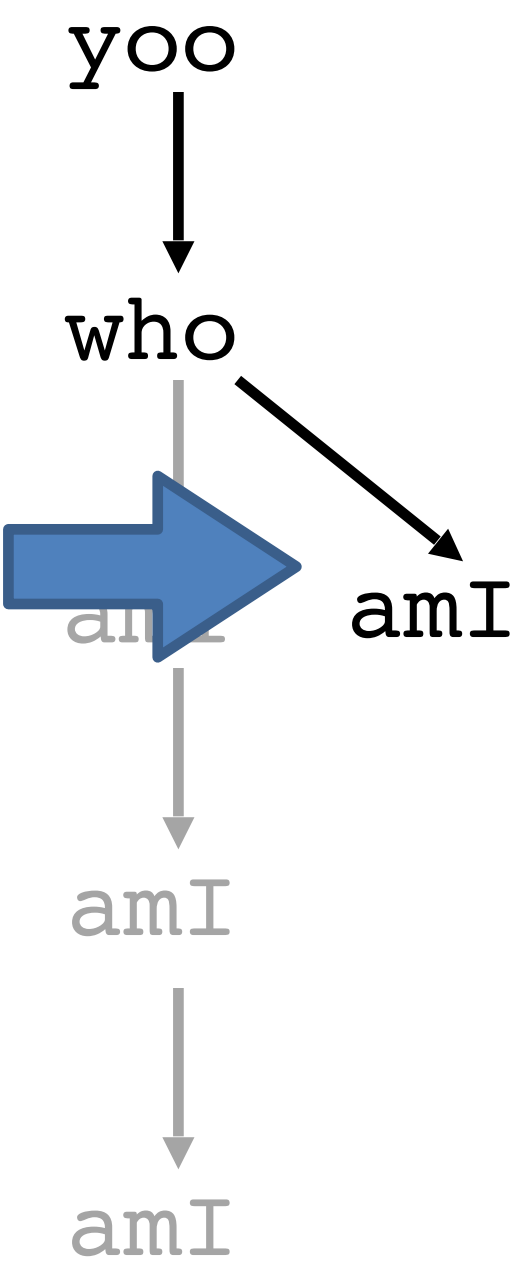
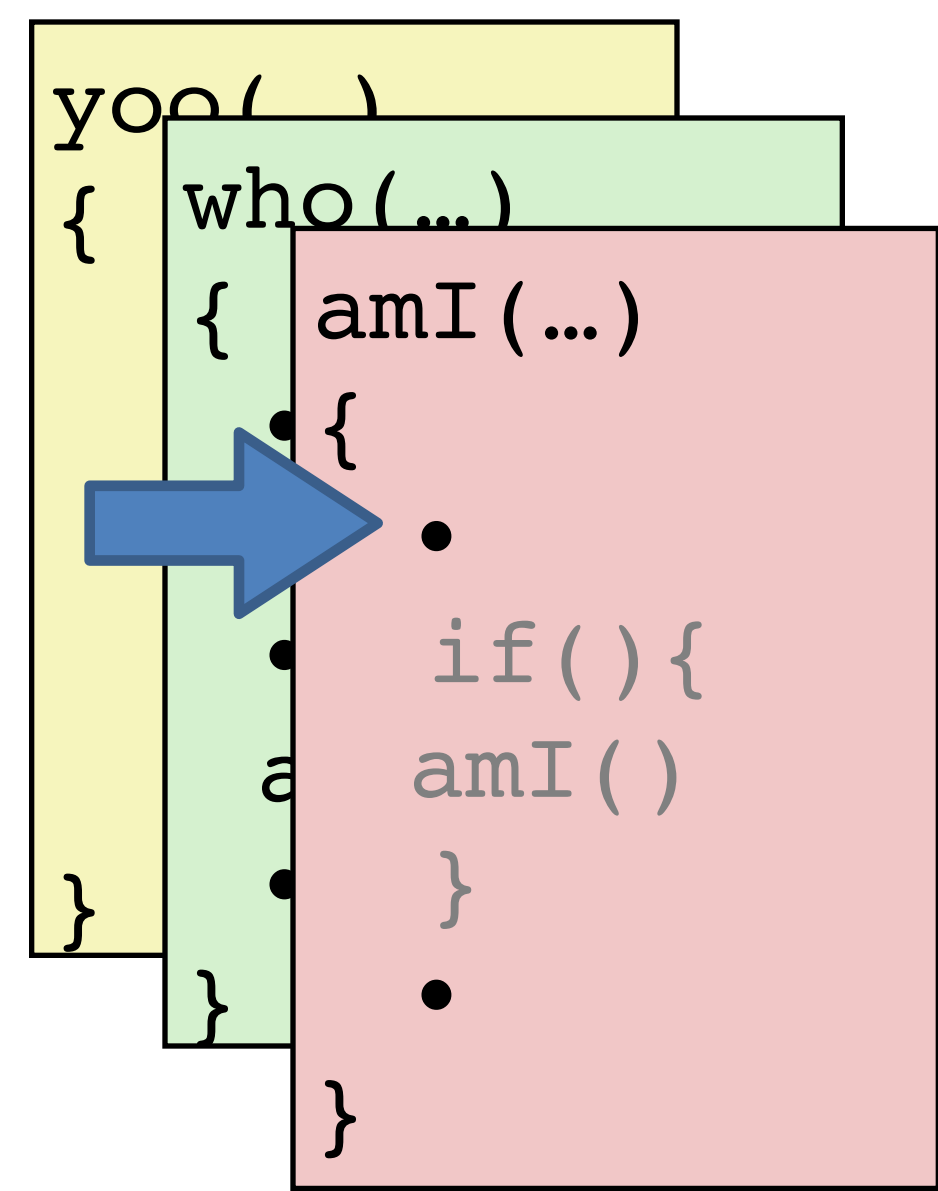
Call stack tracks context



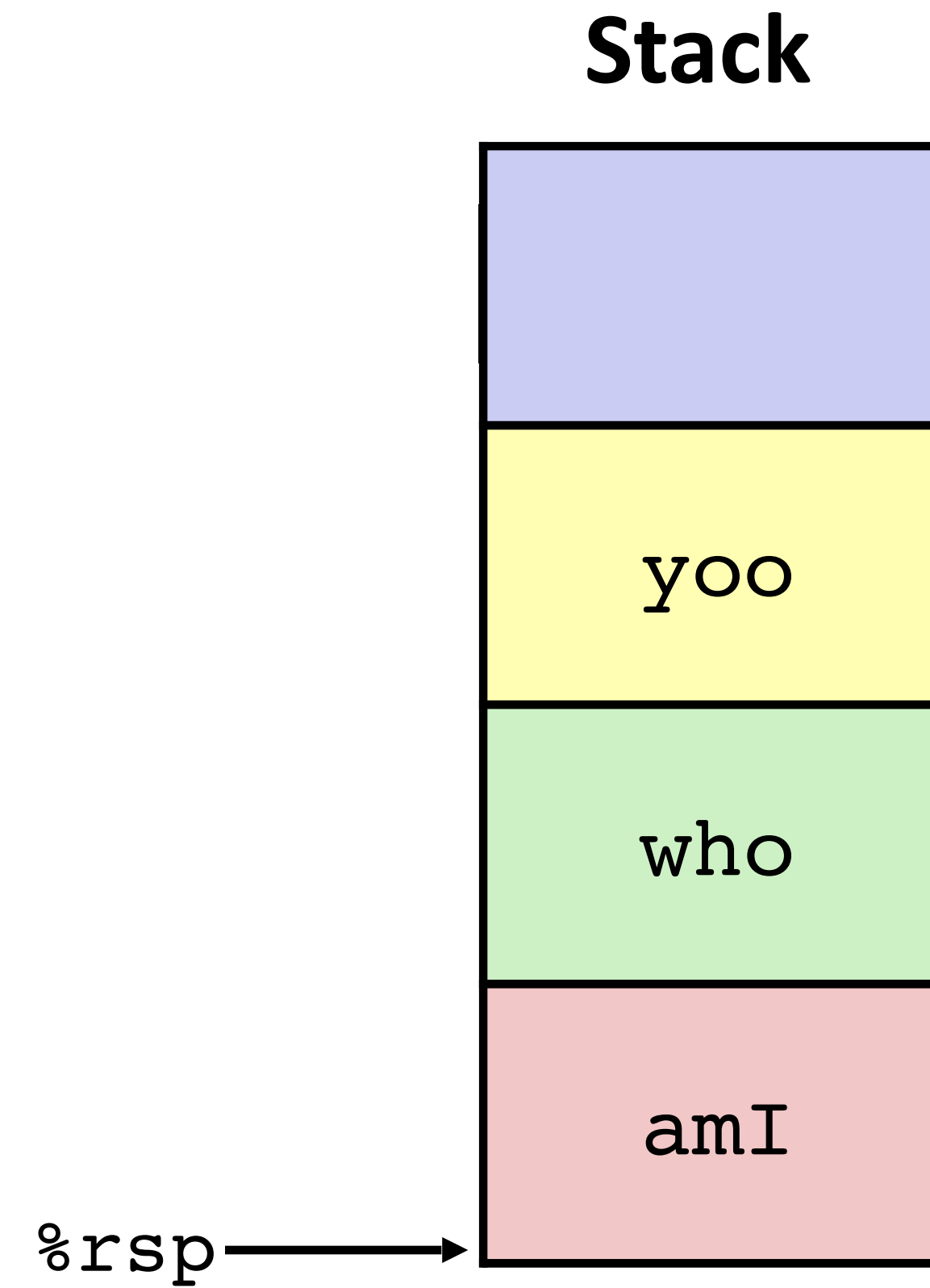
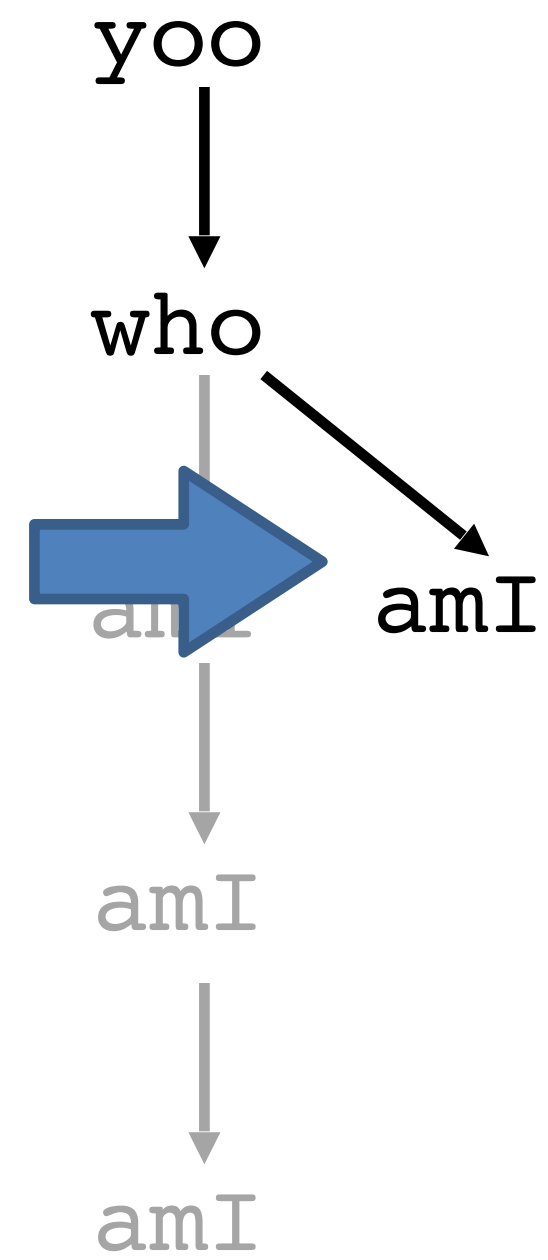
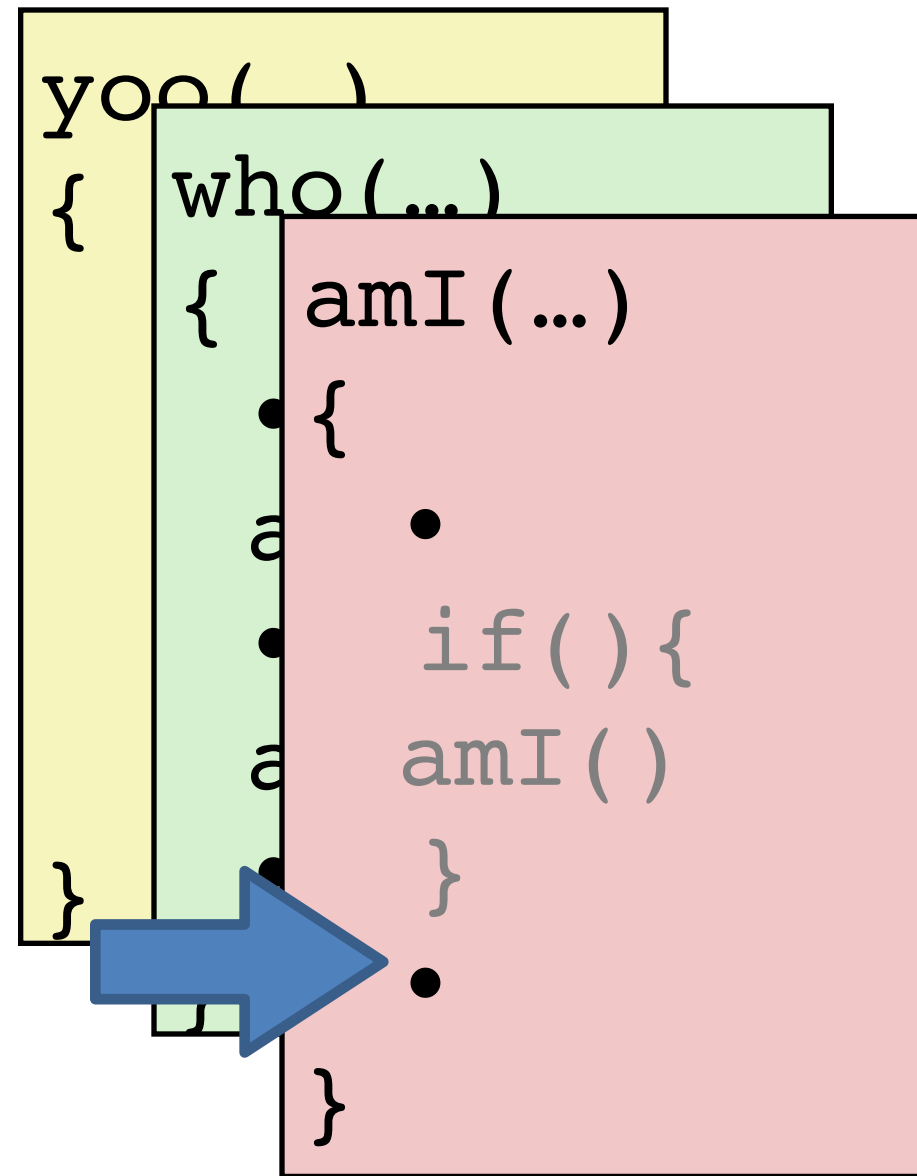
Call stack tracks context



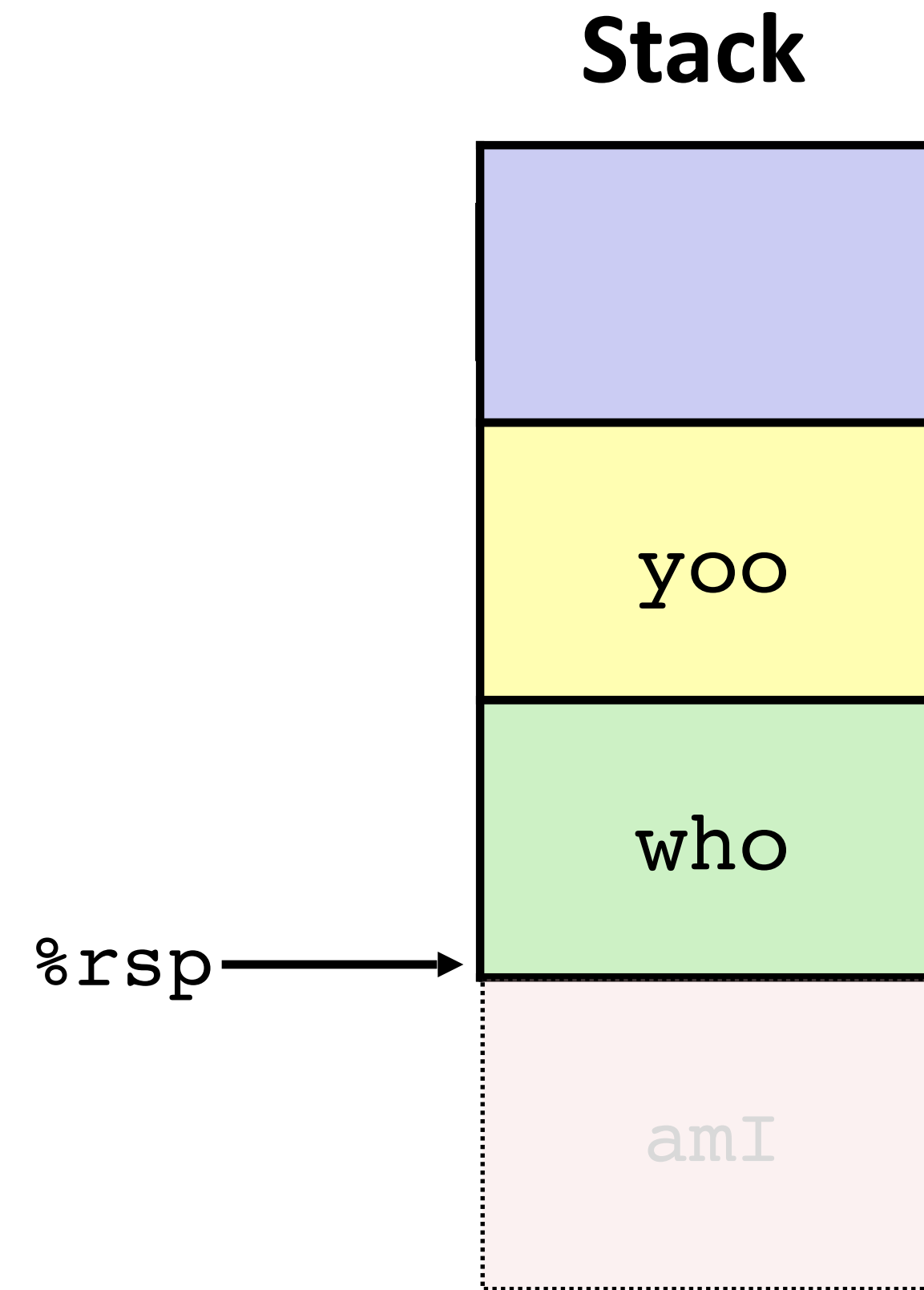
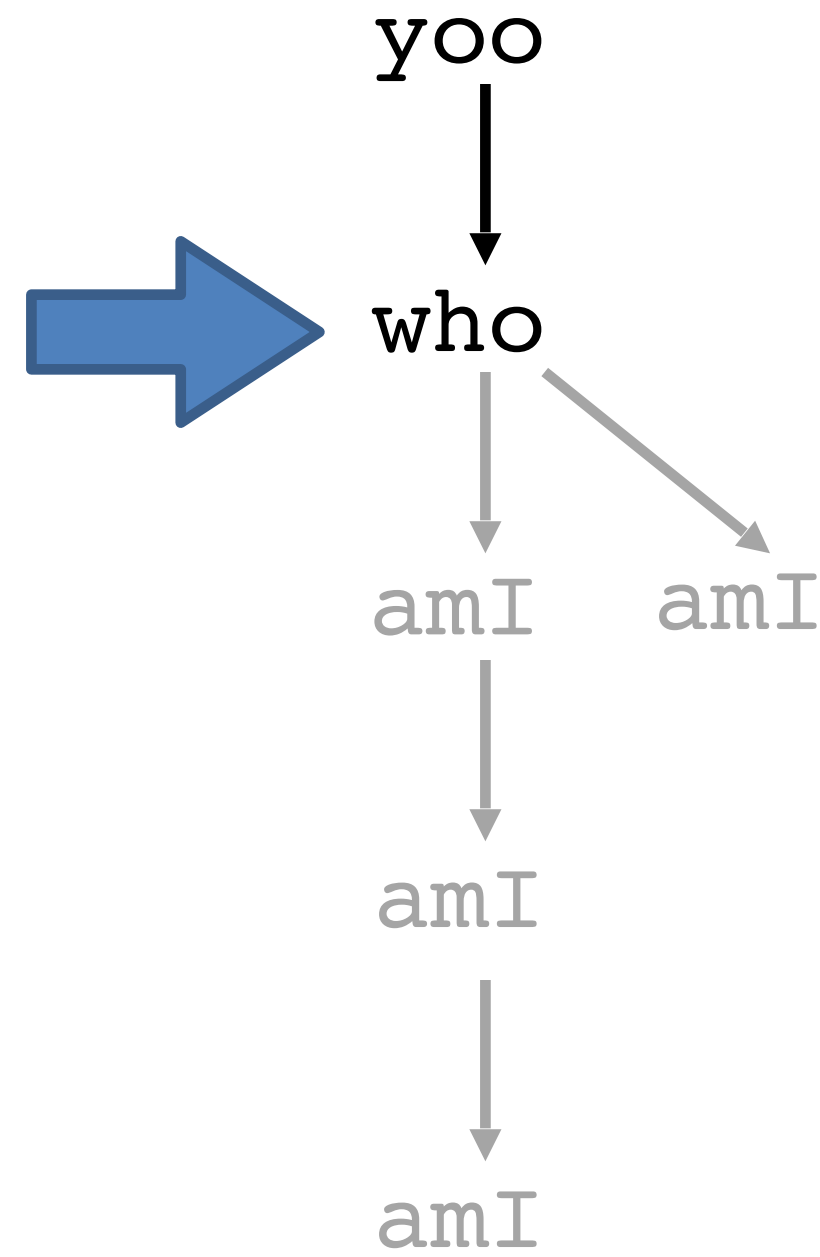
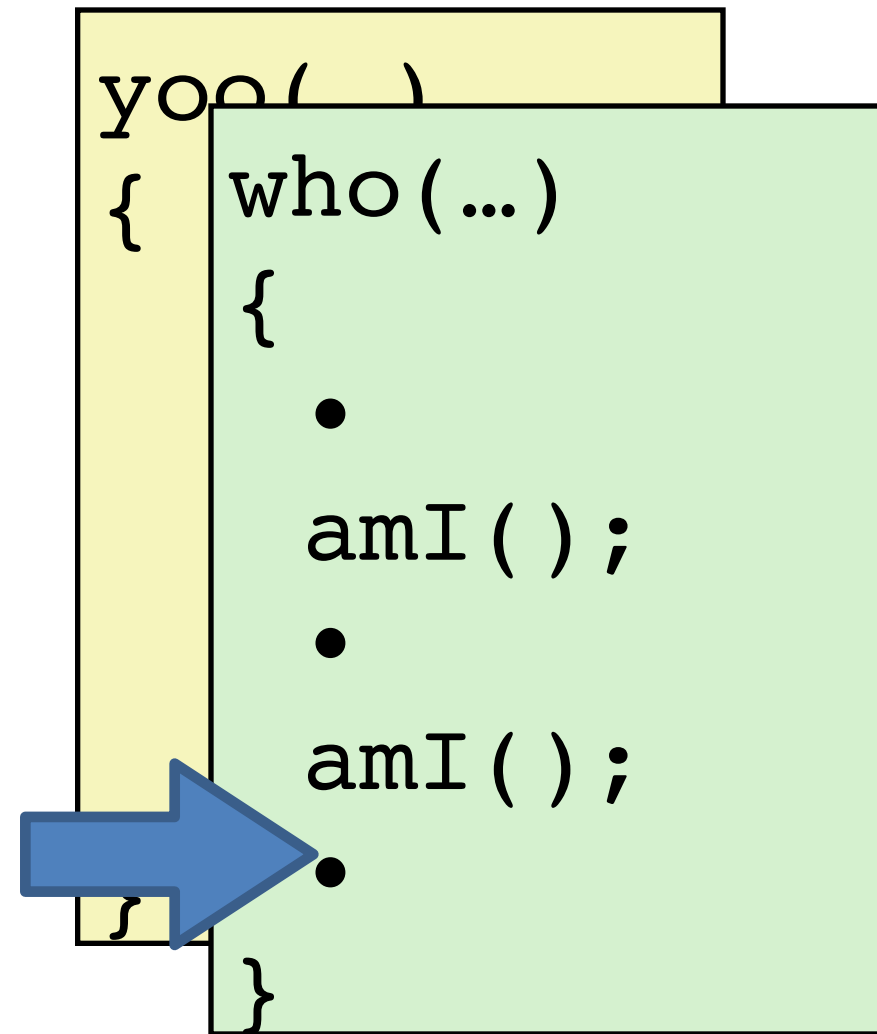
Call stack tracks context



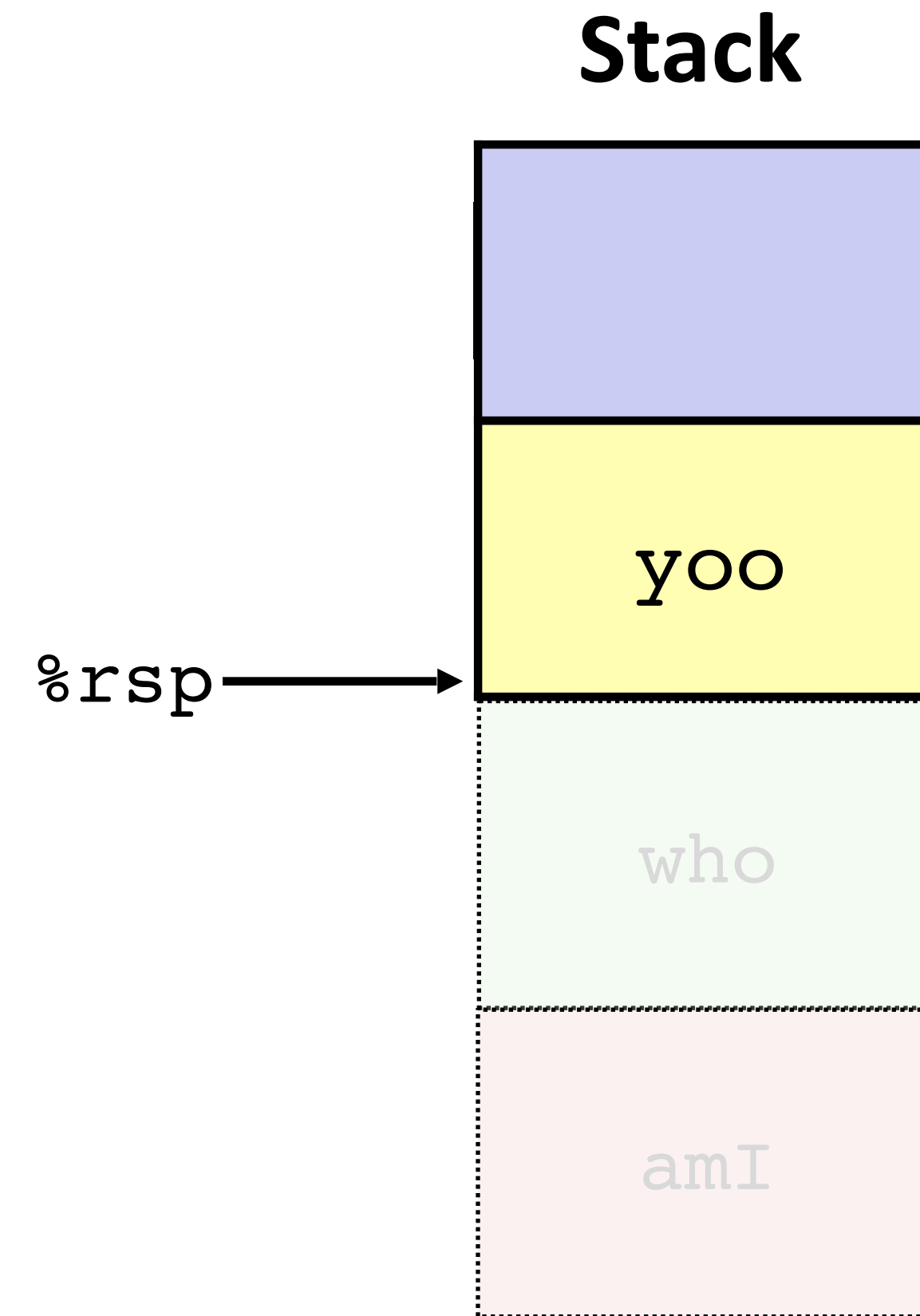
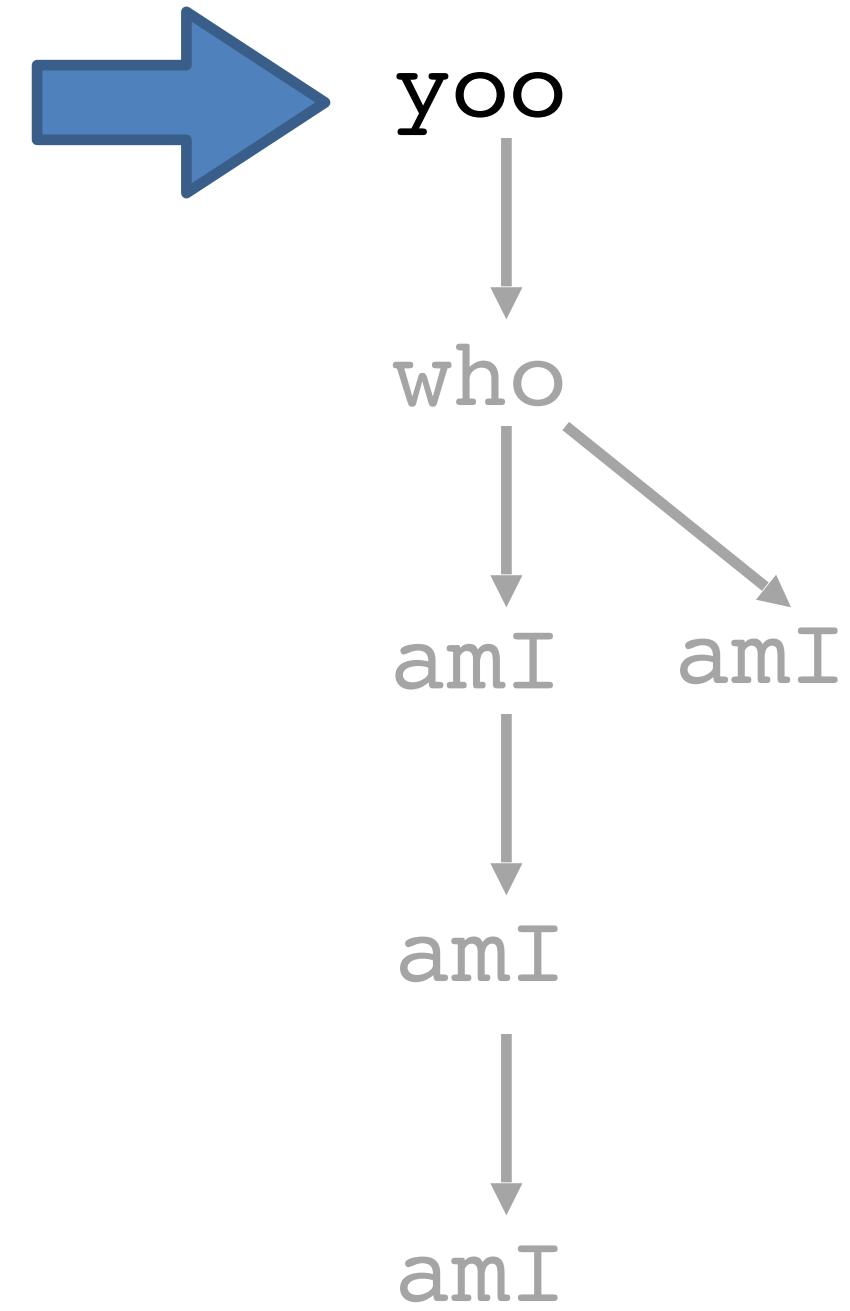
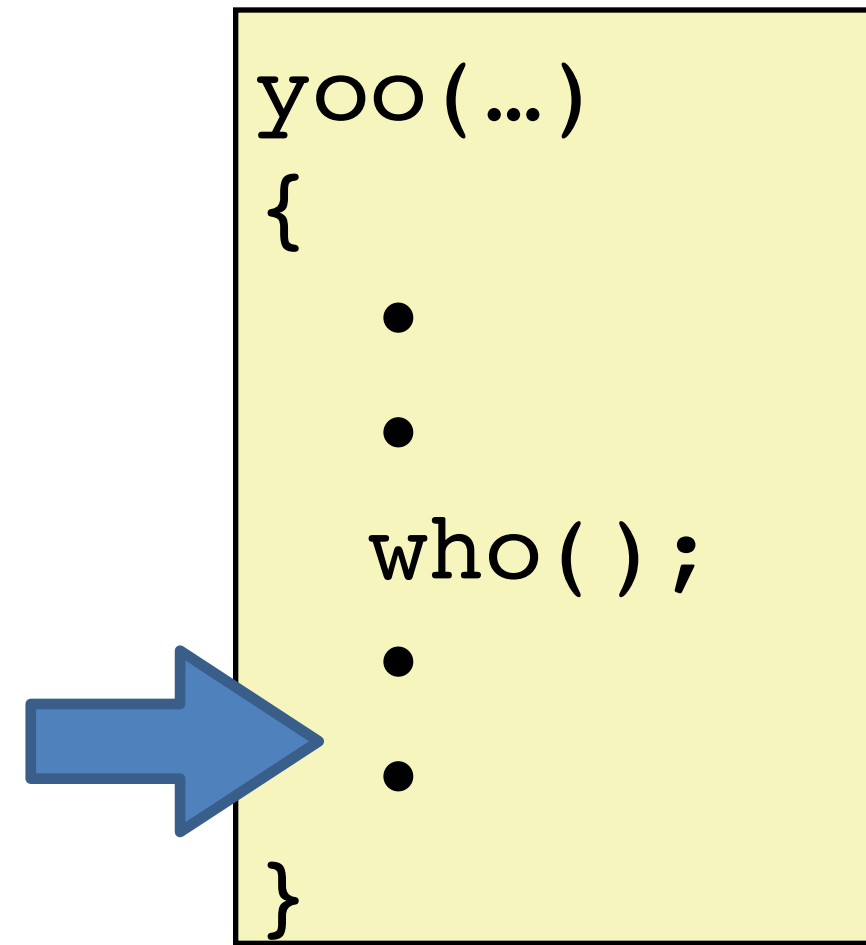
Call stack tracks context



Call stack tracks context



Call stack tracks context



In x86, where is the return address located for a "call" instruction?

✓ 0

In the instructions of the caller function

In the instructions of the callee (being called) function

On the call stack, once per function

On the call stack, once per call

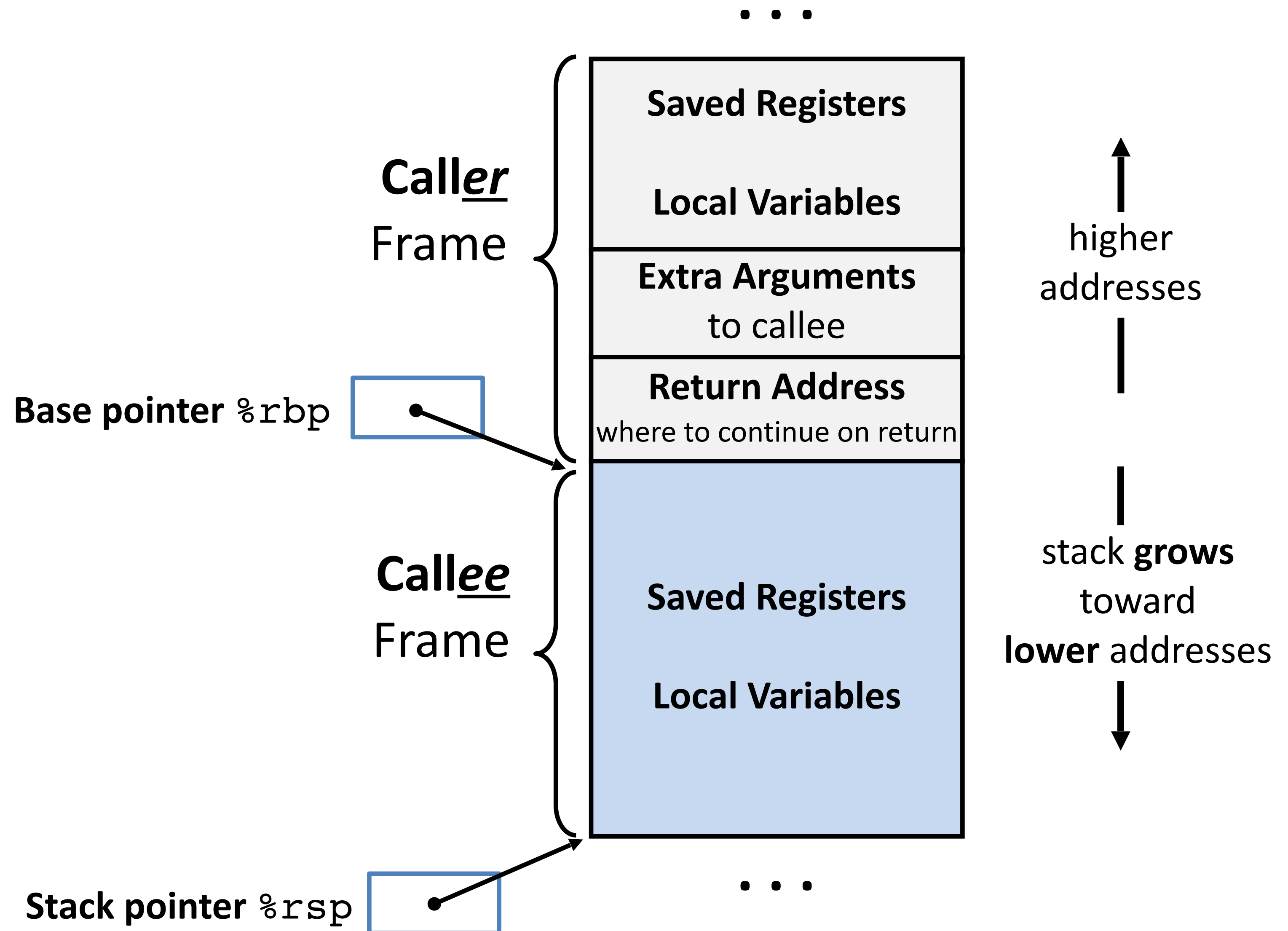
None of the above

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.

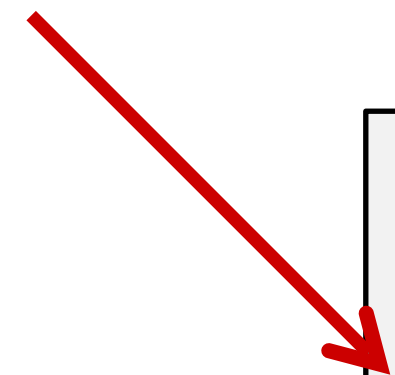


Procedure control flow instructions

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

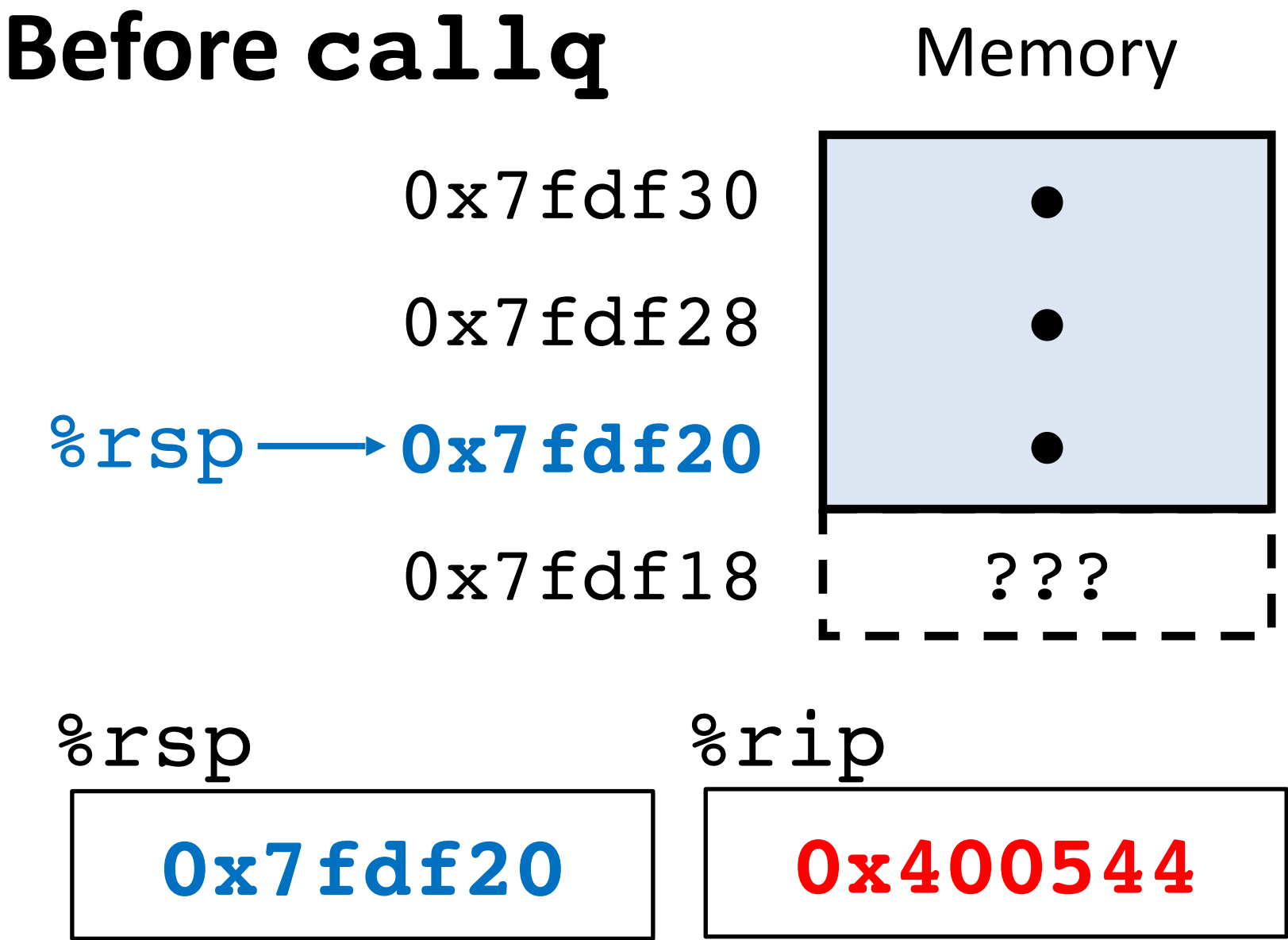
Call example

```
00000000000400540 <multstore>:
•
•
400544: callq 400550 <mult2>
400549: mov %rax, (%rbx)
•
•
```

```
00000000000400550 <mult2>:
400550: mov %rdi,%rax
•
•
400557: retq
```

callq target

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



Call example

```
00000000000400540 <multstore>:
•
•
400544: callq 400550 <mult2>
400549: mov   %rax, (%rbx)
•
•
```

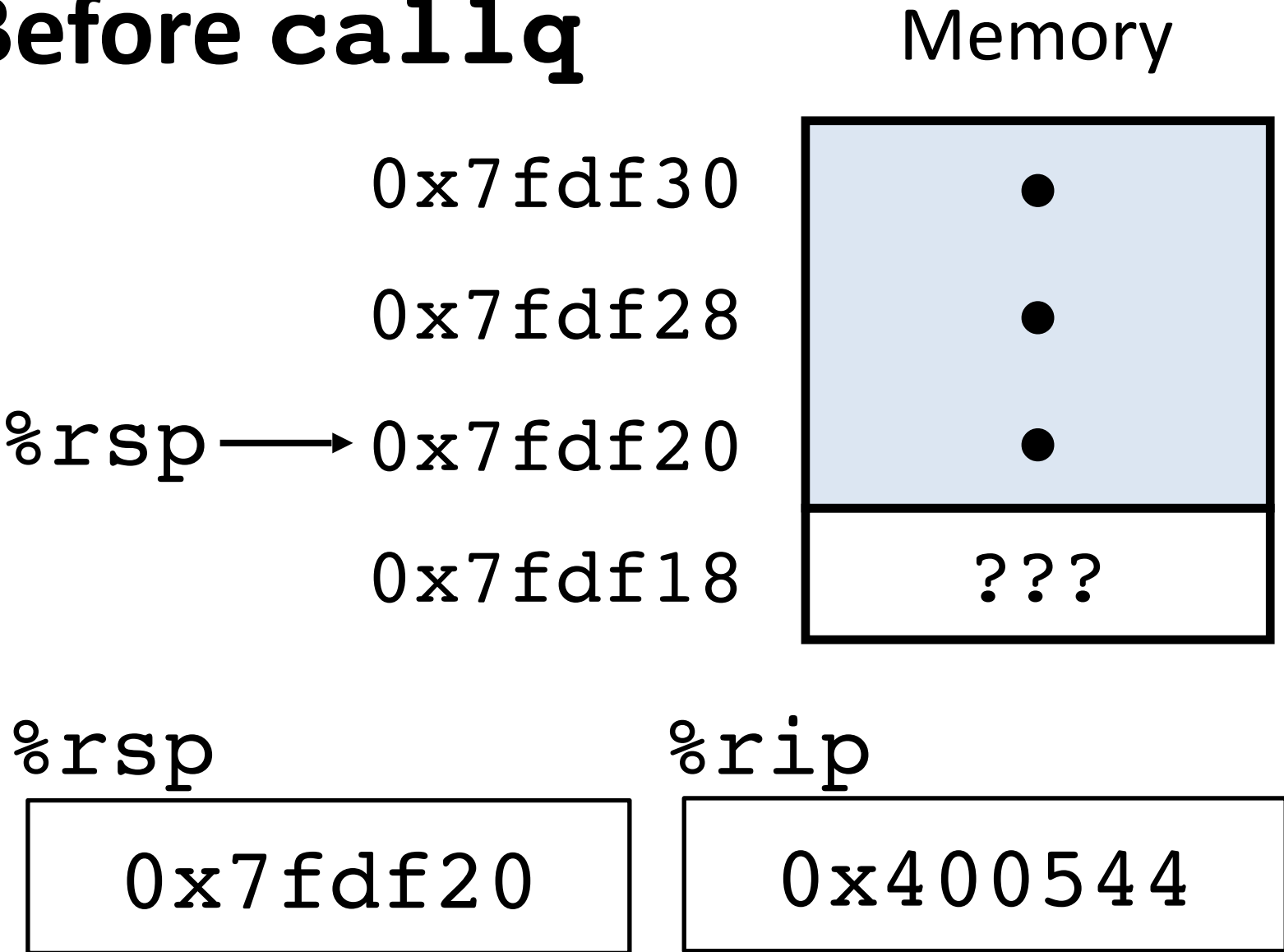
```
00000000000400550 <mult2>:
400550: mov   %rdi, %rax
•
•
400557: retq
```

callq target

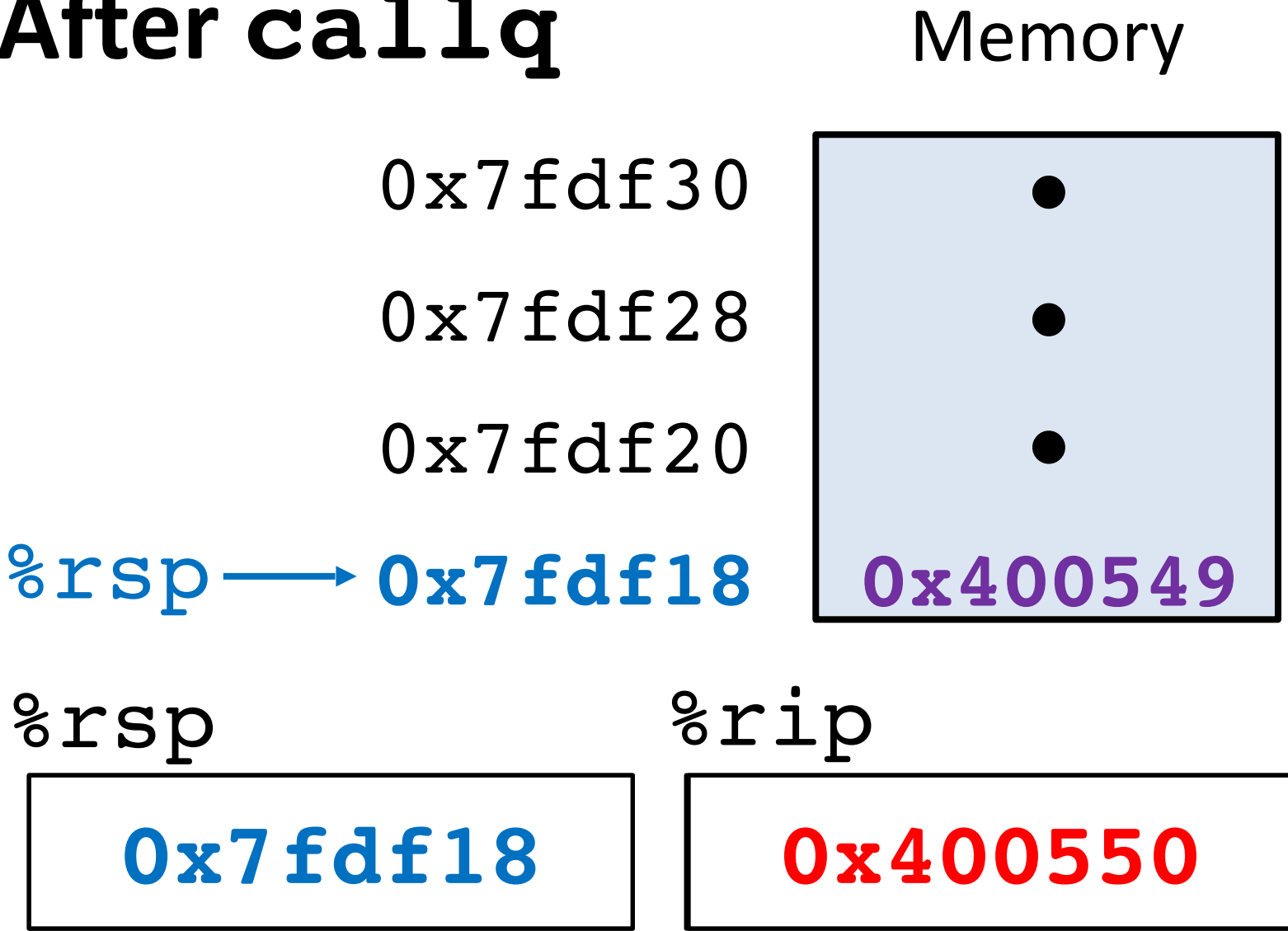
1. Push return address on stack
2. Jump to *target*



Before callq



After callq



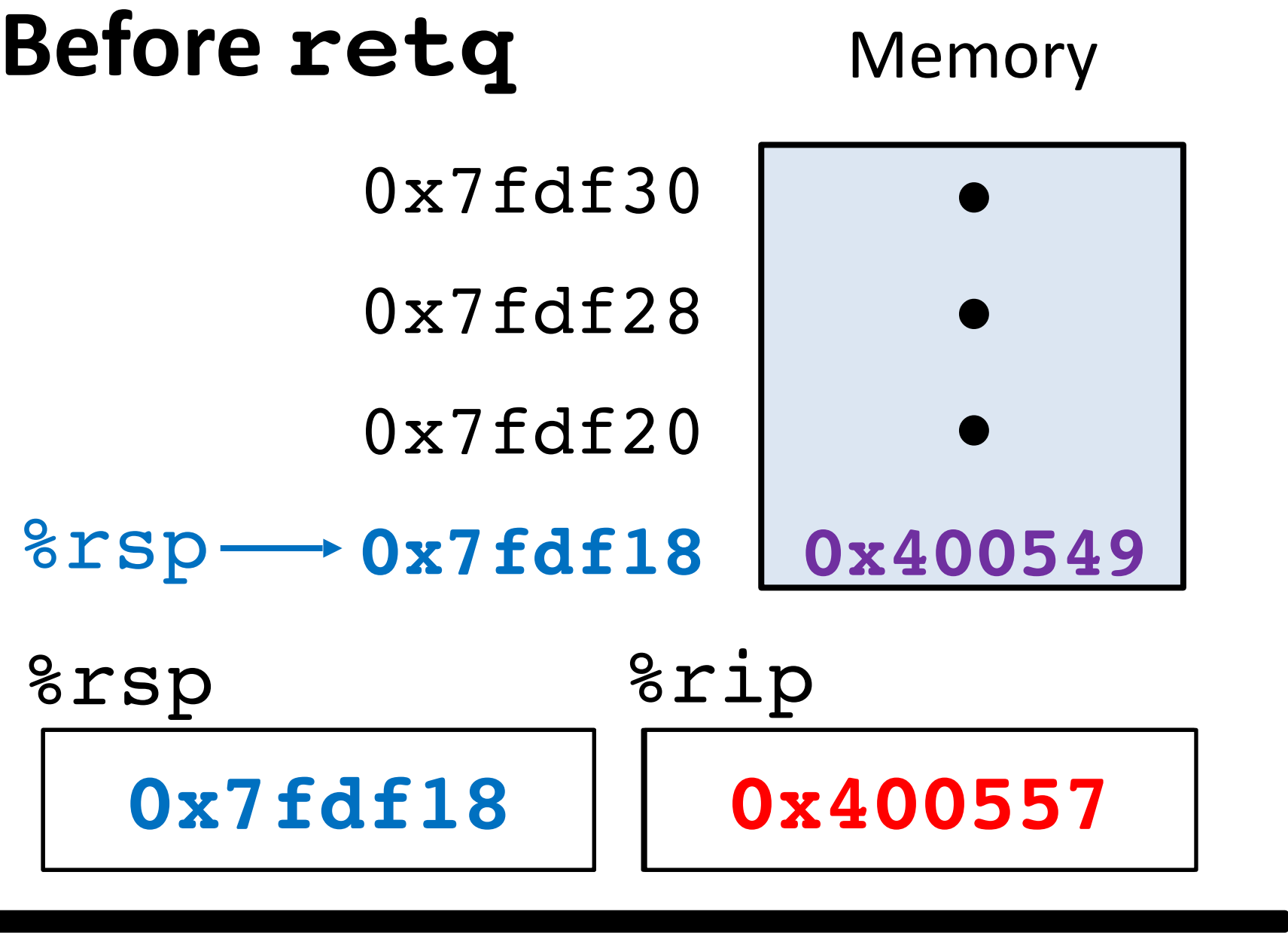
Return example

```
00000000000400540 <multstore>:
•
•
400544: callq 400550 <mult2>
400549: mov   %rax, (%rbx)
•
•
```

```
00000000000400550 <mult2>:
400550: mov   %rdi,%rax
•
•
400557: retq
```

retq

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



Return example

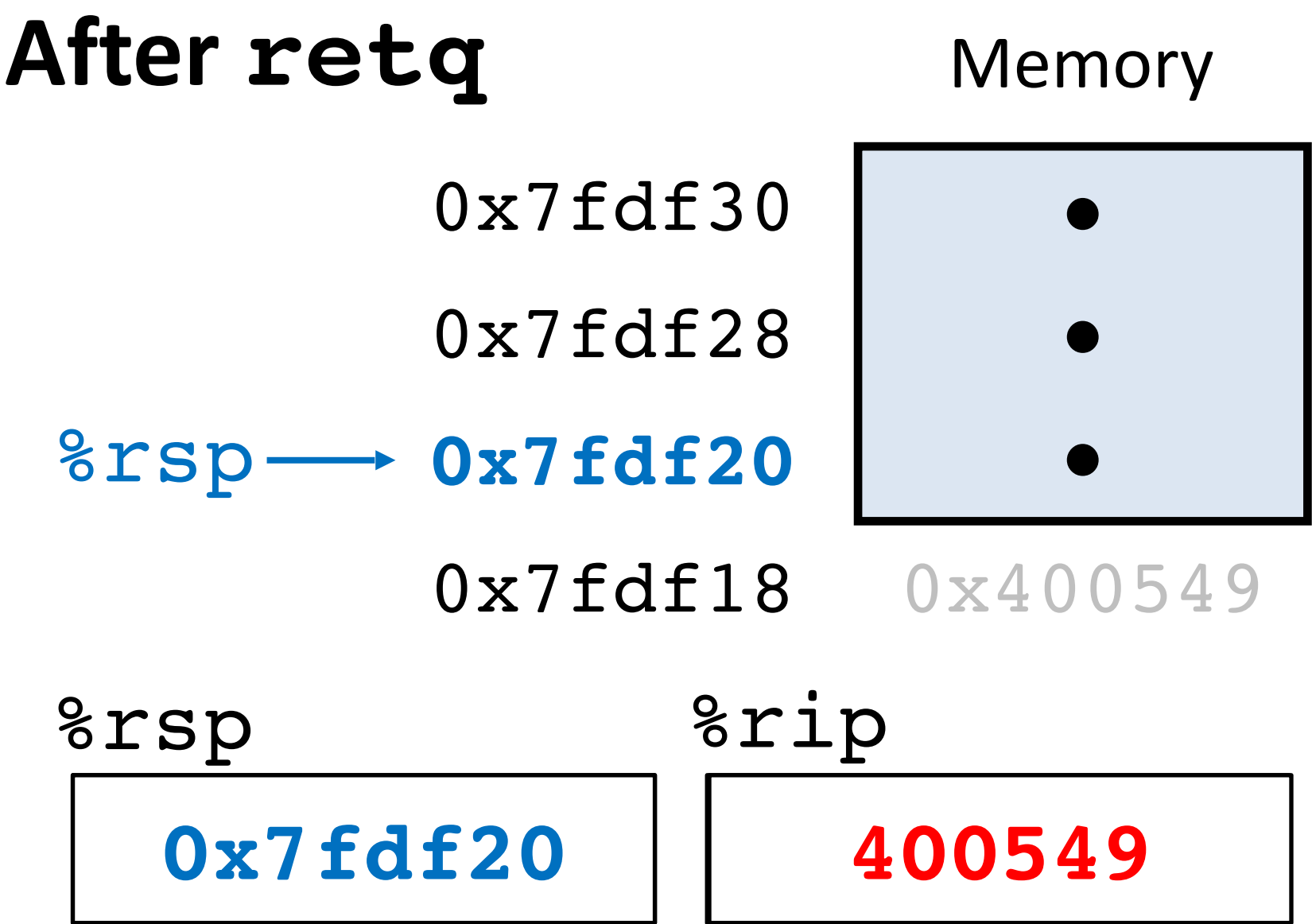
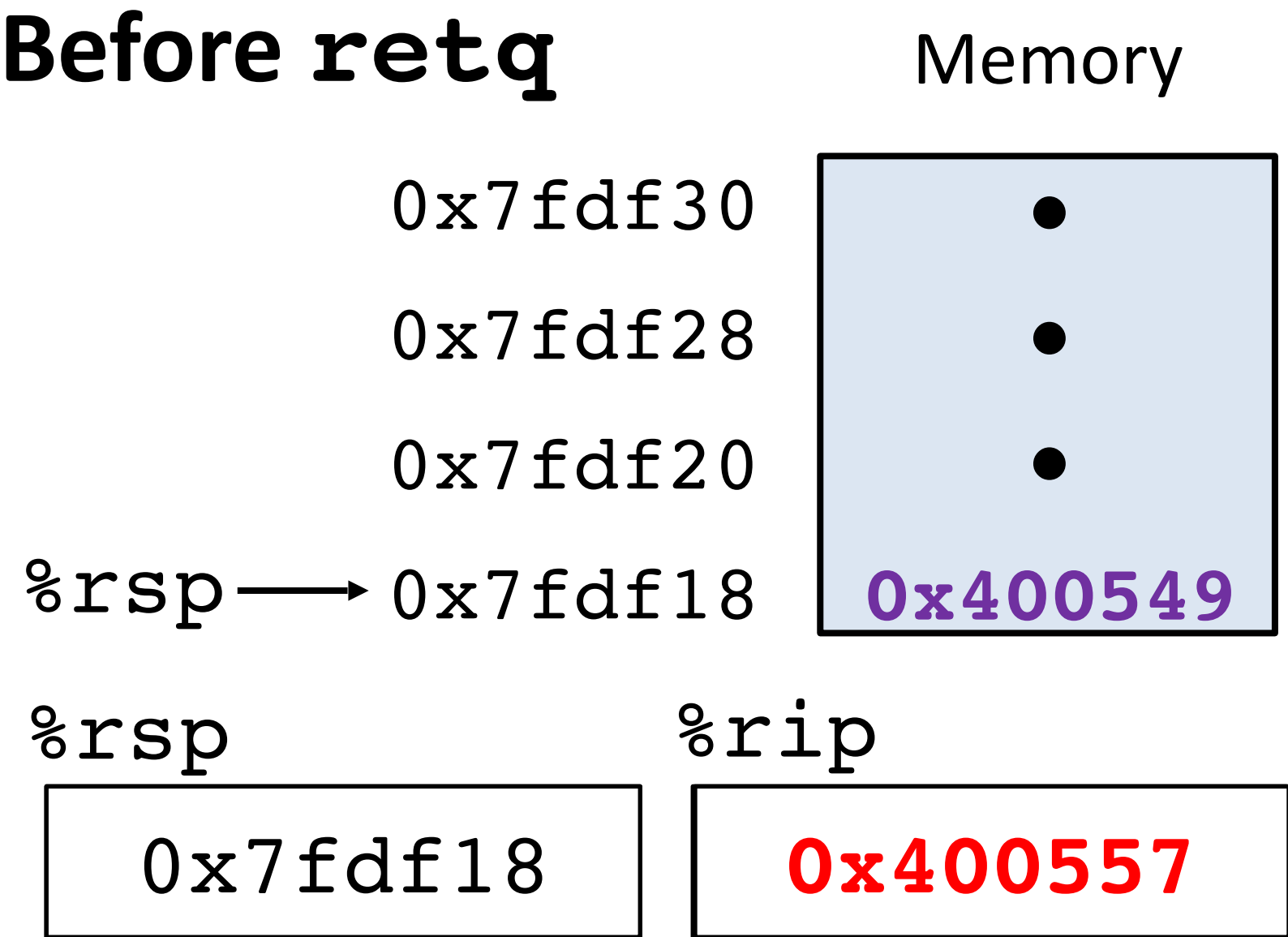
```
00000000000400540 <multstore>:
•
•
400544: callq 400550 <mult2>
400549: mov    %rax, (%rbx)
•
•
```

```
00000000000400550 <mult2>:
400550: mov    %rdi,%rax
•
•
400557: retq
```

retq

1. Pop return address from stack

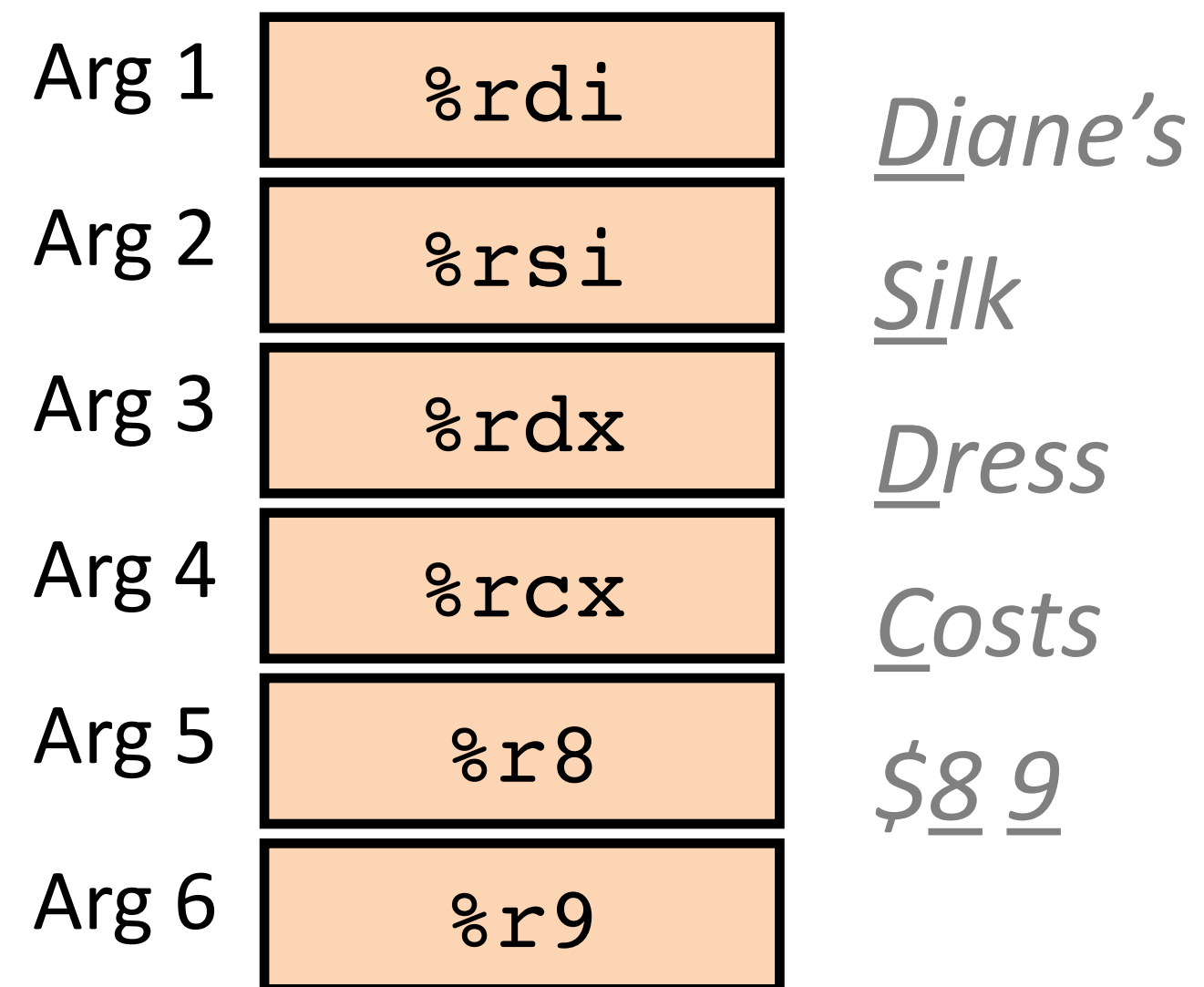
2. Jump to return address



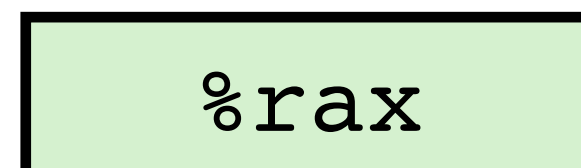
Procedure data flow conventions

Recall:

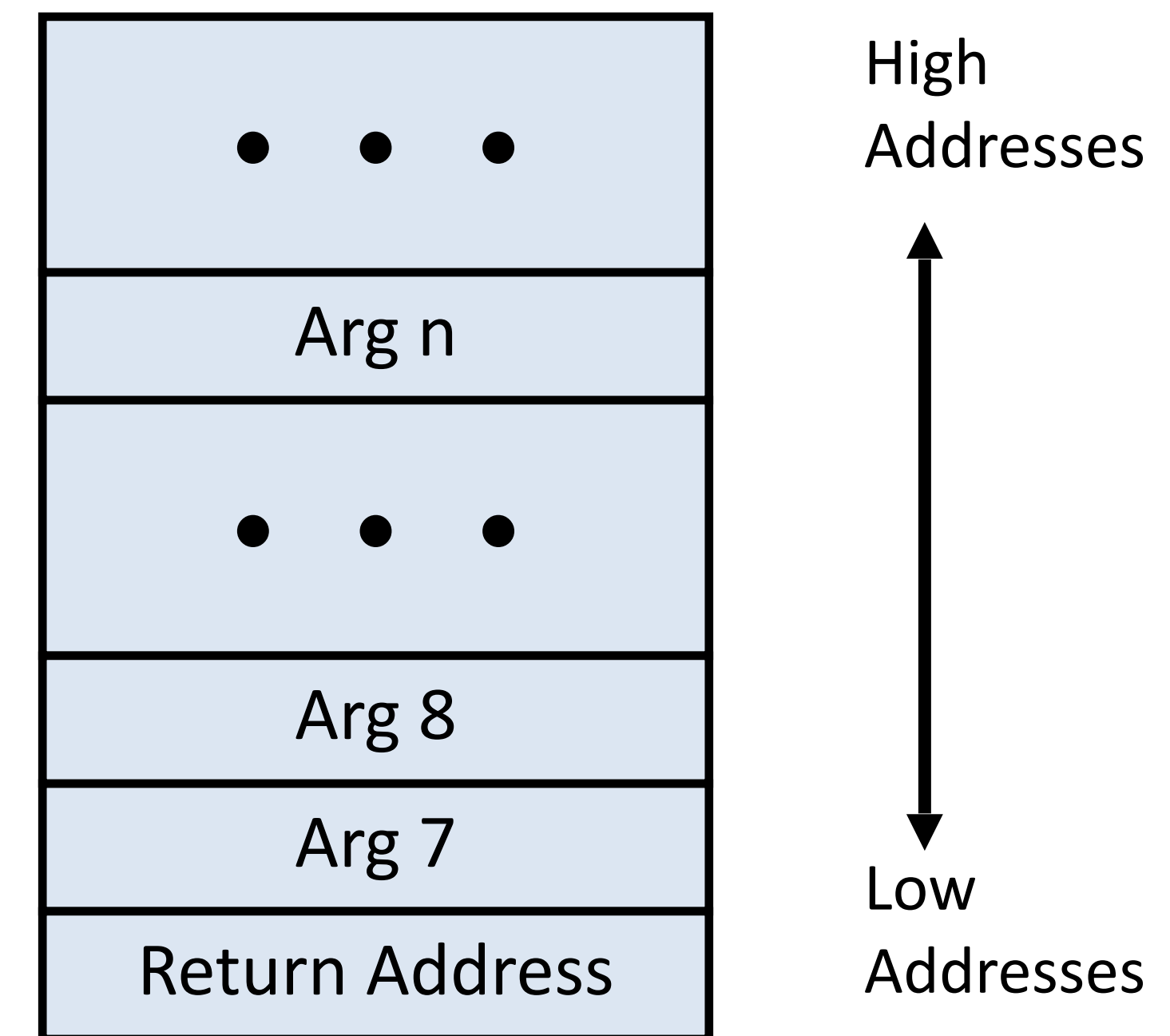
First 6 arguments: passed in **registers**



Return value: passed in **%rax**



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



Procedure call / stack frame example

ex

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
```

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

Passes address of local variable (in stack).

increment:

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

Uses memory through pointer.

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

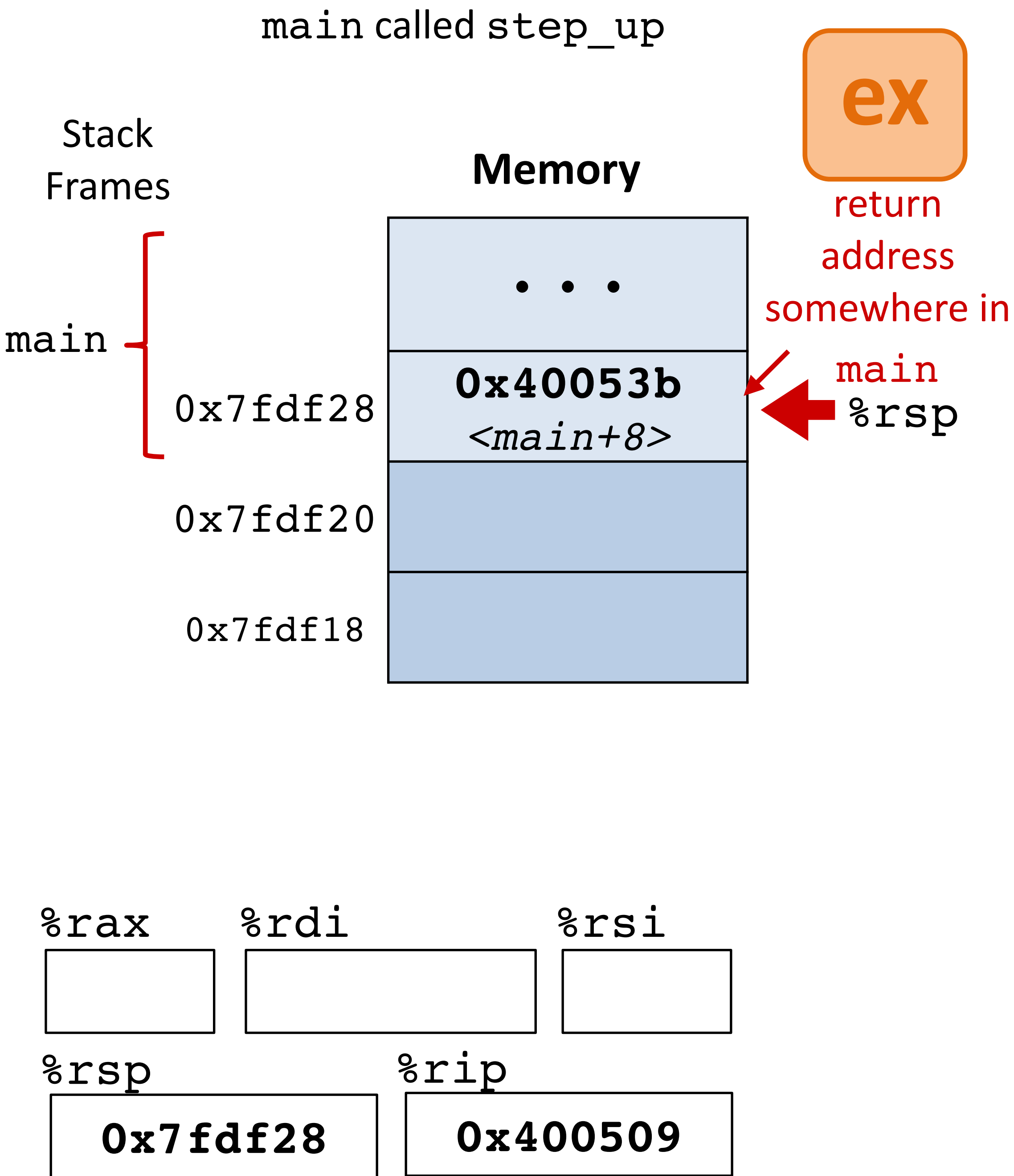
Procedure call example (step 0)

```
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

ex

```
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

main

step_up

0x7fdf28

0x7fdf20

0x7fdf18

Memory

• • •

```
0x40053b
<main+8>
```

240
v1

local variable
v1 stored in
stack address

`%rsp`

```
%rax
```

```
%rdi
```

%rsi

`%rsp`

```
%rip
```

0x7fdf20

0x400515

Procedure call example (step 2)

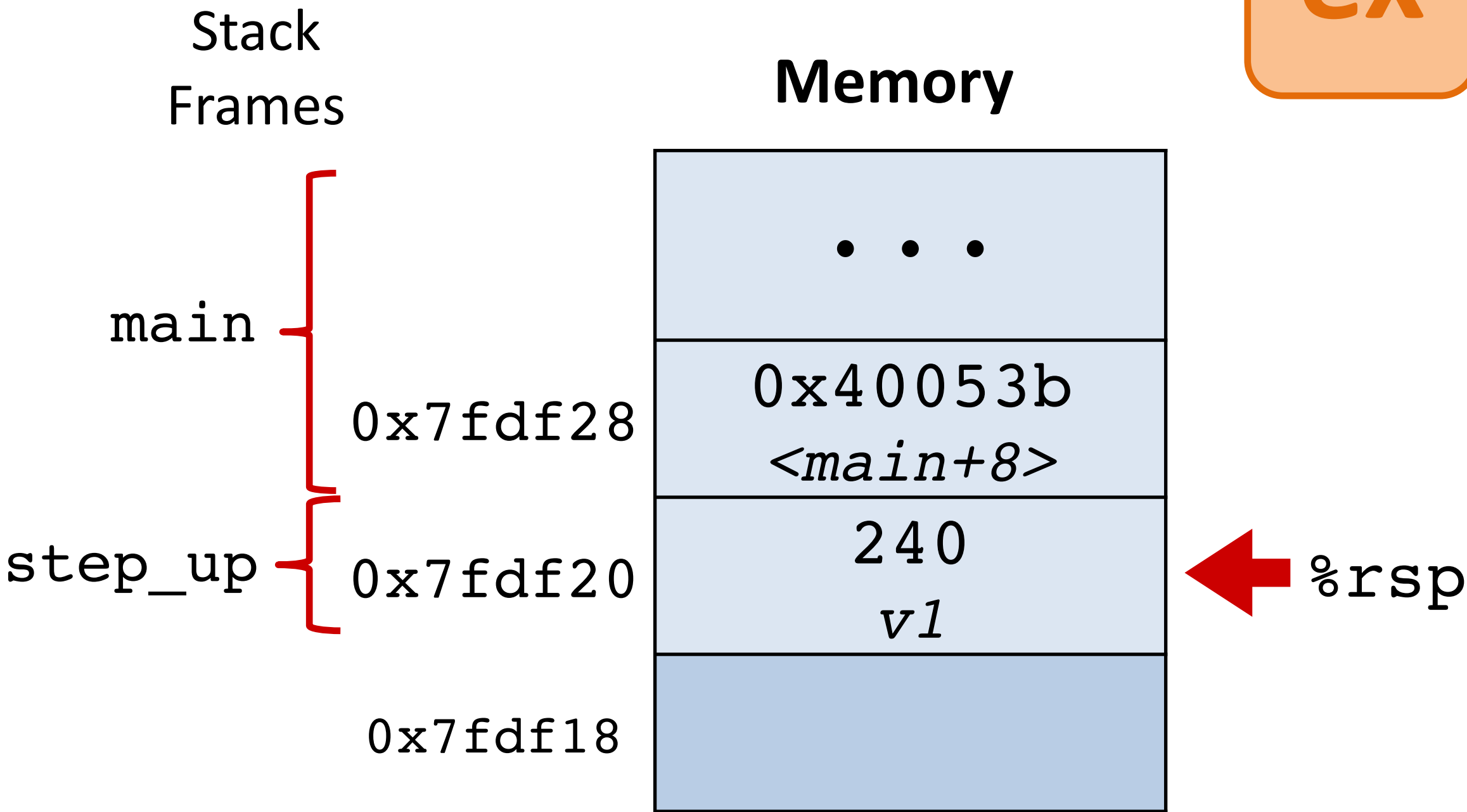
Set up args for call
to increment



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

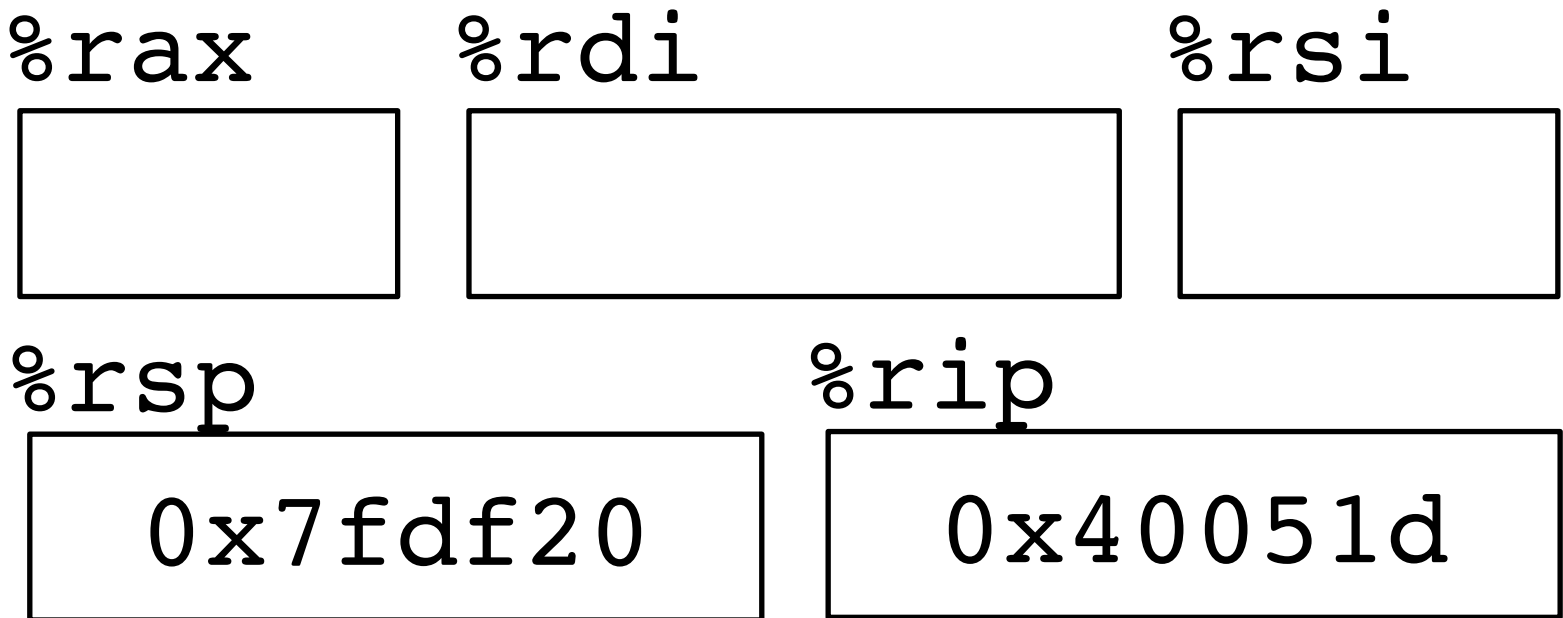
```
step_up:
400509:  subq    $8, %rsp
40050d:  movq    $240, (%rsp)
400515:  movq    %rsp, %rdi
400518:  movl    $61, %esi
40051d:  callq   4004cd <increment>
400522:  addq    (%rsp), %rax
400526:  addq    $8, %rsp
40052a:  retq
```

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



Place args in registers before call

★ Move the two args to
the right registers



Procedure call example (step 2)

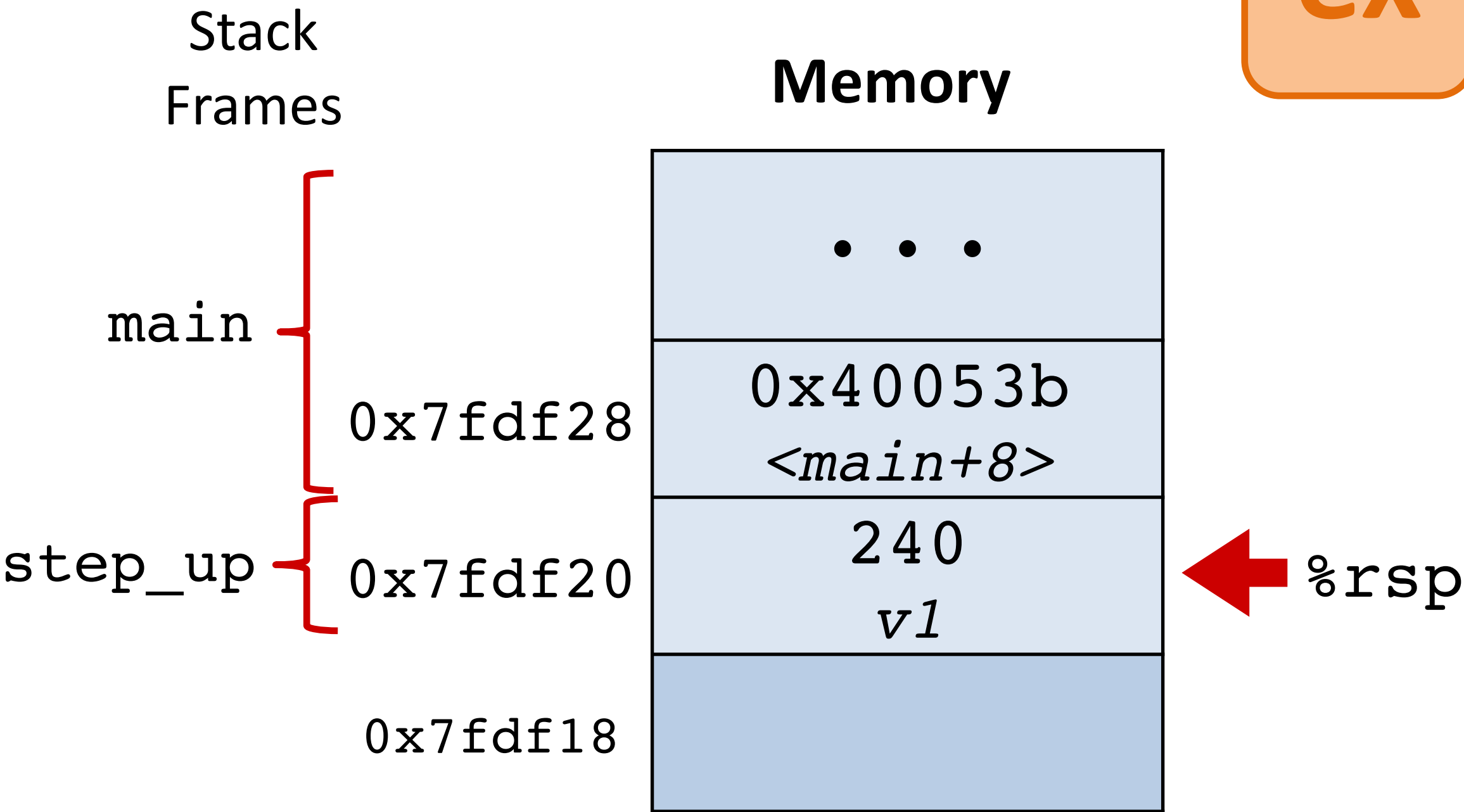
Set up args for call
to increment



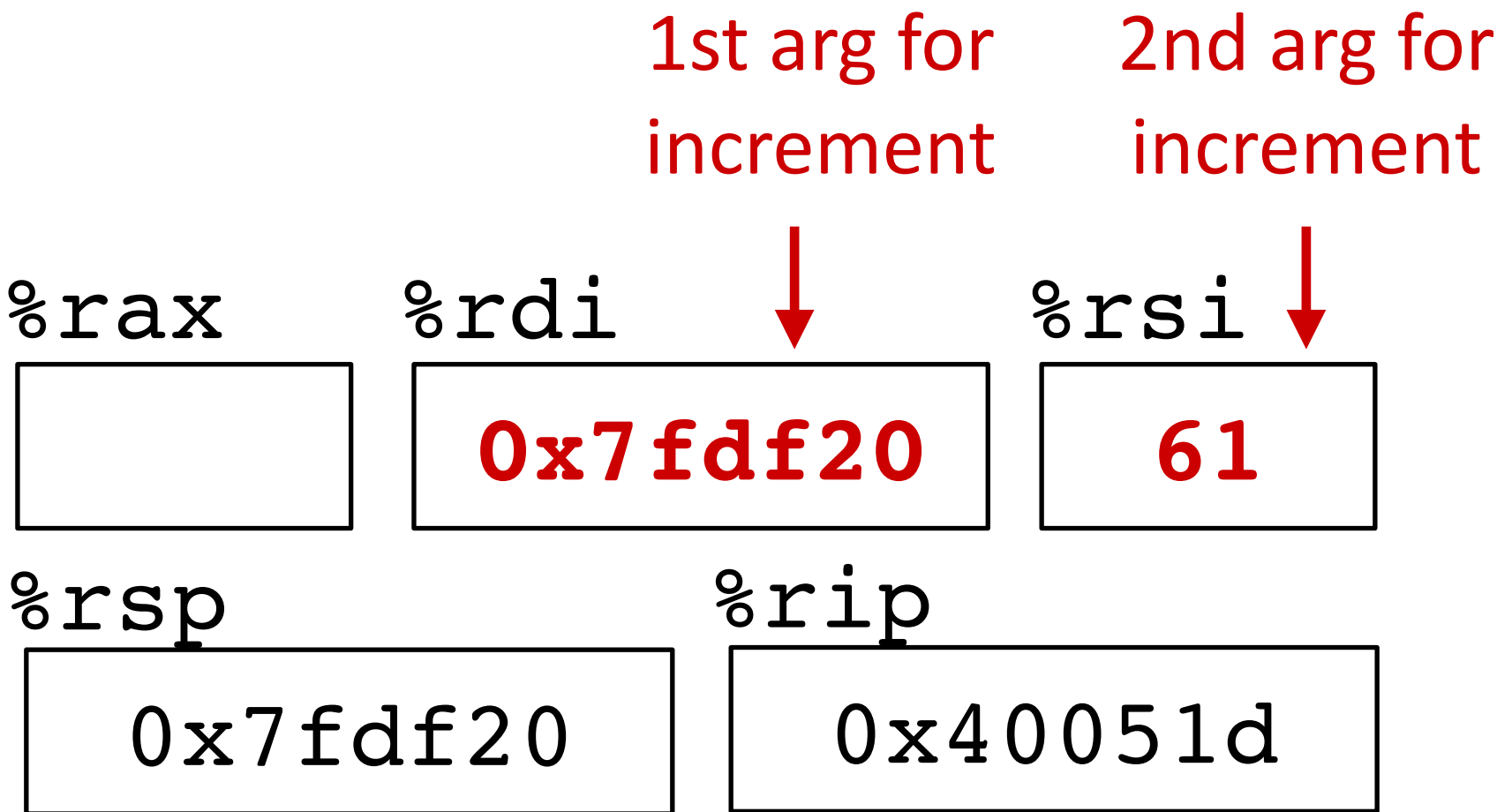
```
long step_up() {
    long v1 = 240;
    long v2 = increment(&v1, 61);
    return v1+v2;
}
```

```
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
```



Place args in registers before call



Procedure call example (step 3)

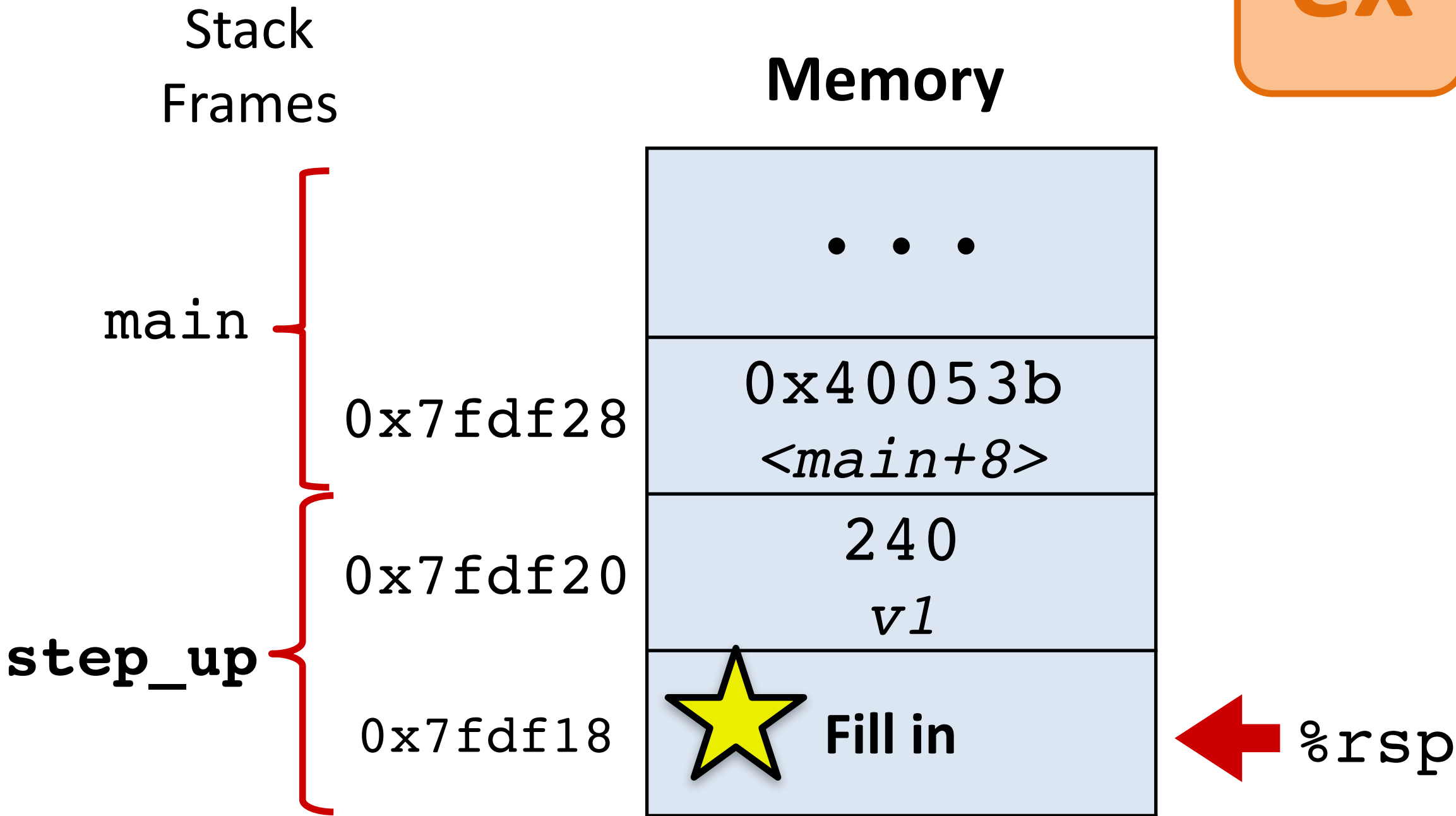
Call increment



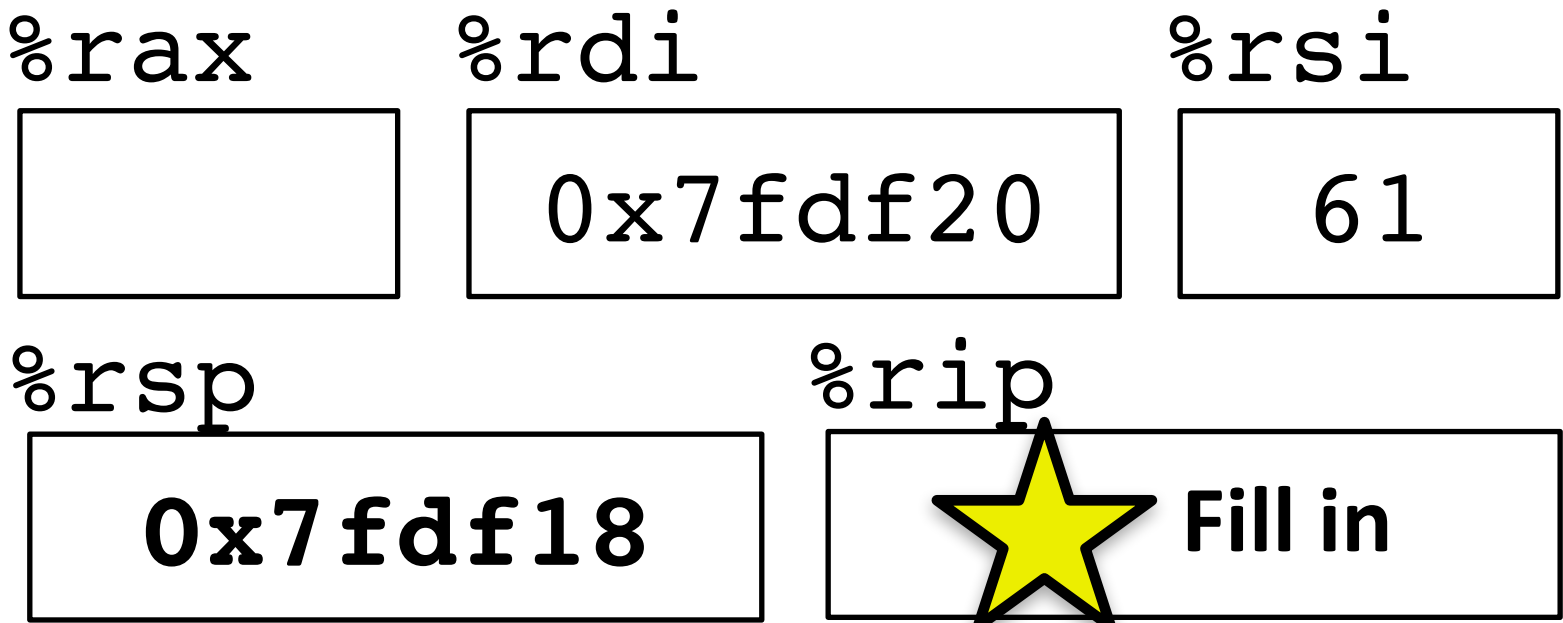
```
long step_up() {
    long v1 = 240;
    long v2 = increment(&v1, 61);
    return v1+v2;
}
```

```
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
```



call has two steps
(1) Push return address on stack
(2) Jump to target



Procedure call example (step 3)

Call increment

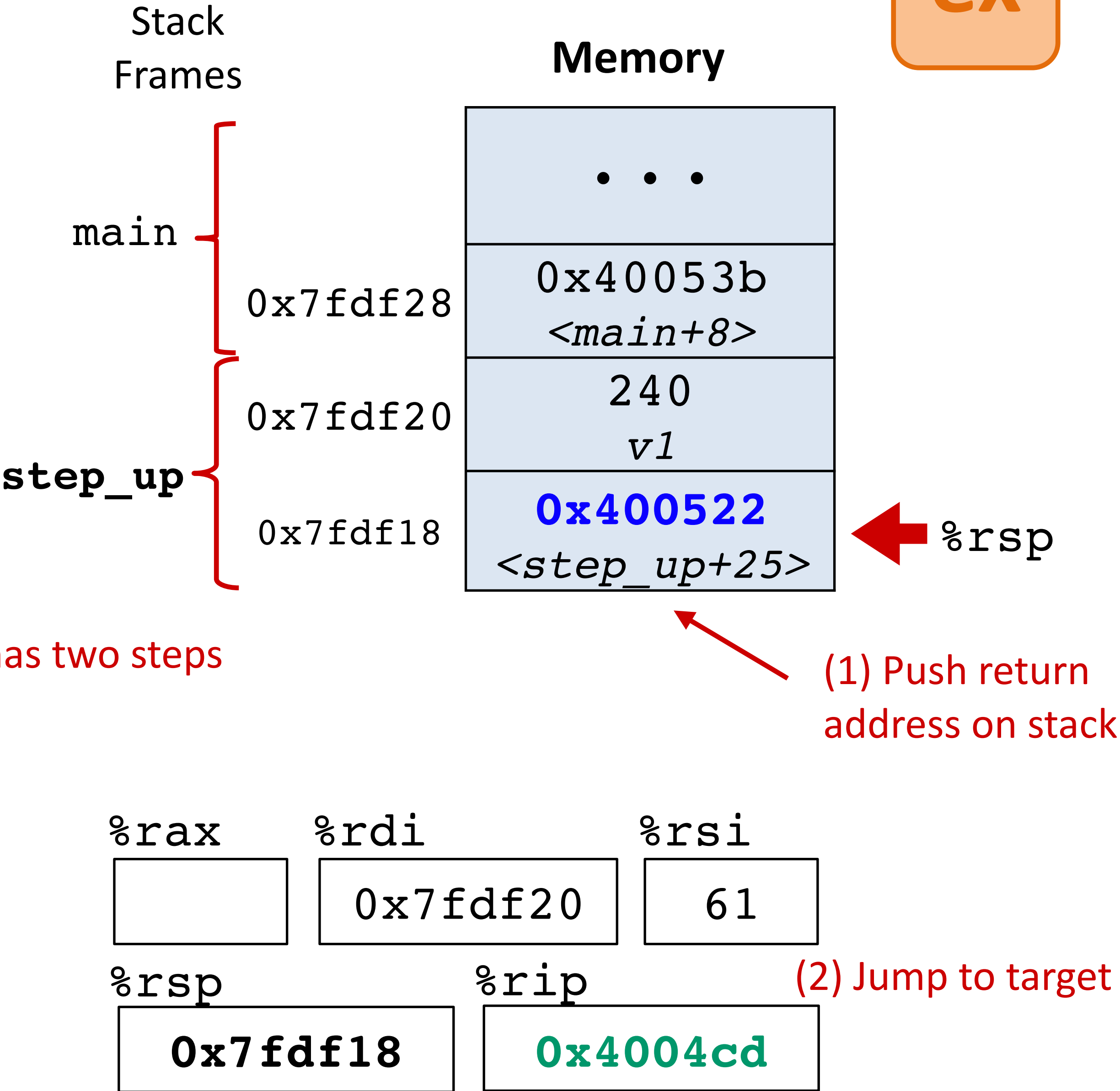
ex

```
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 4)

Run increment

ex

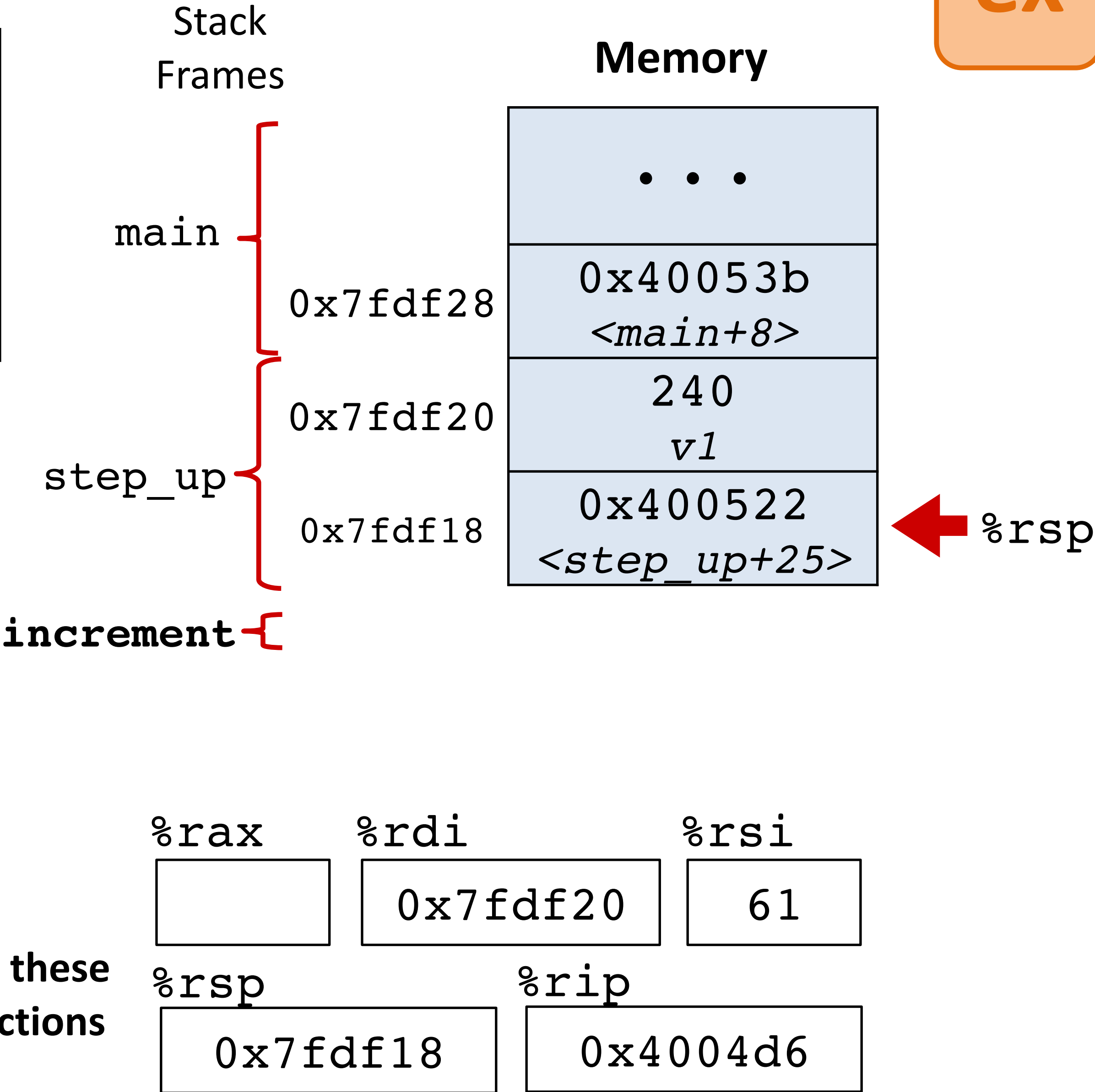
```
10 long increment(long* p, long val) {
    long x = *p;
    long y = x + val;
    *p = y;
    return x;
}
```

```
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
```



Execute these 3 instructions



Procedure call example (step 4)

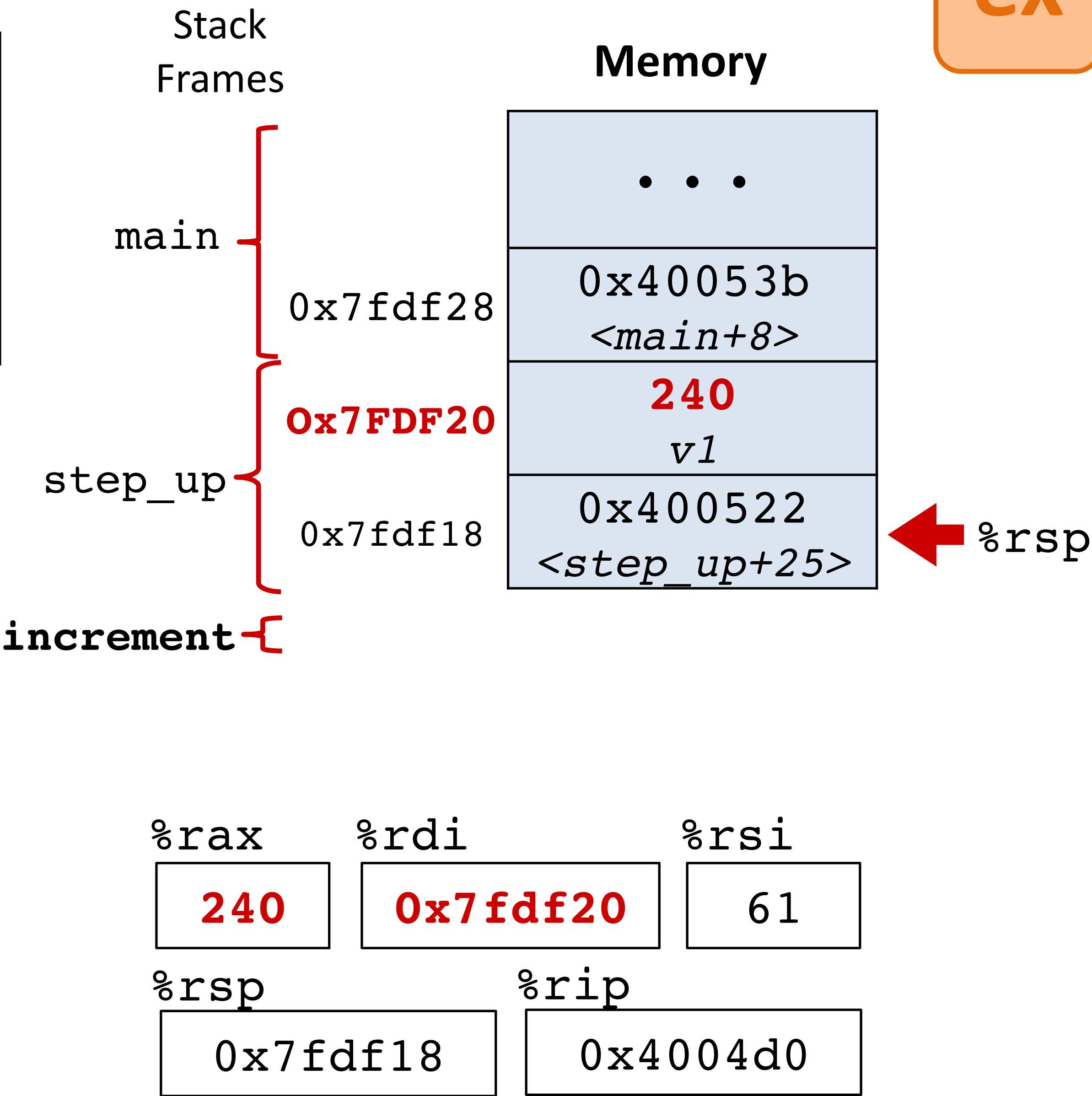
Run increment



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

```
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 4)

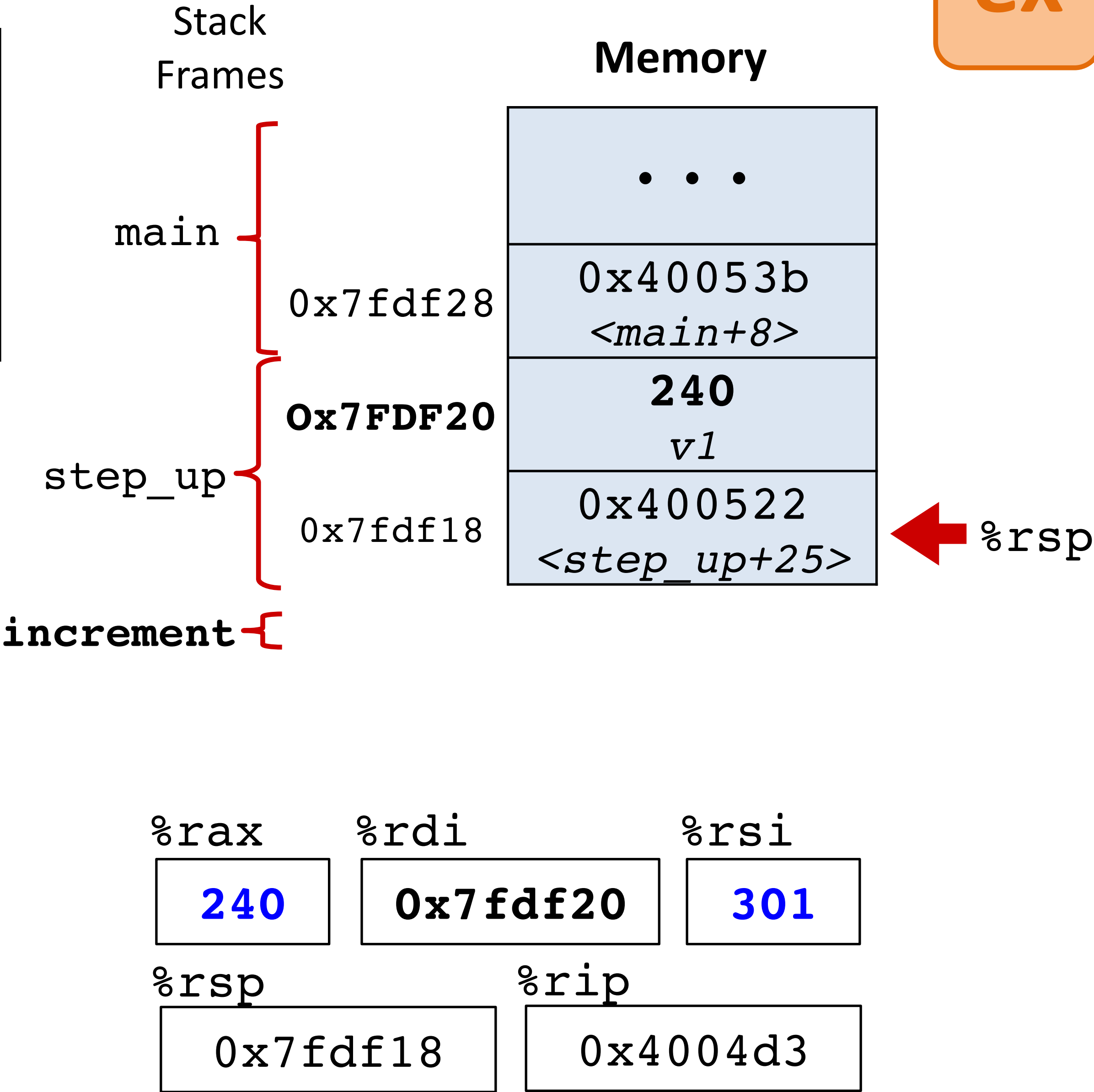
Run increment



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

```
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 4)

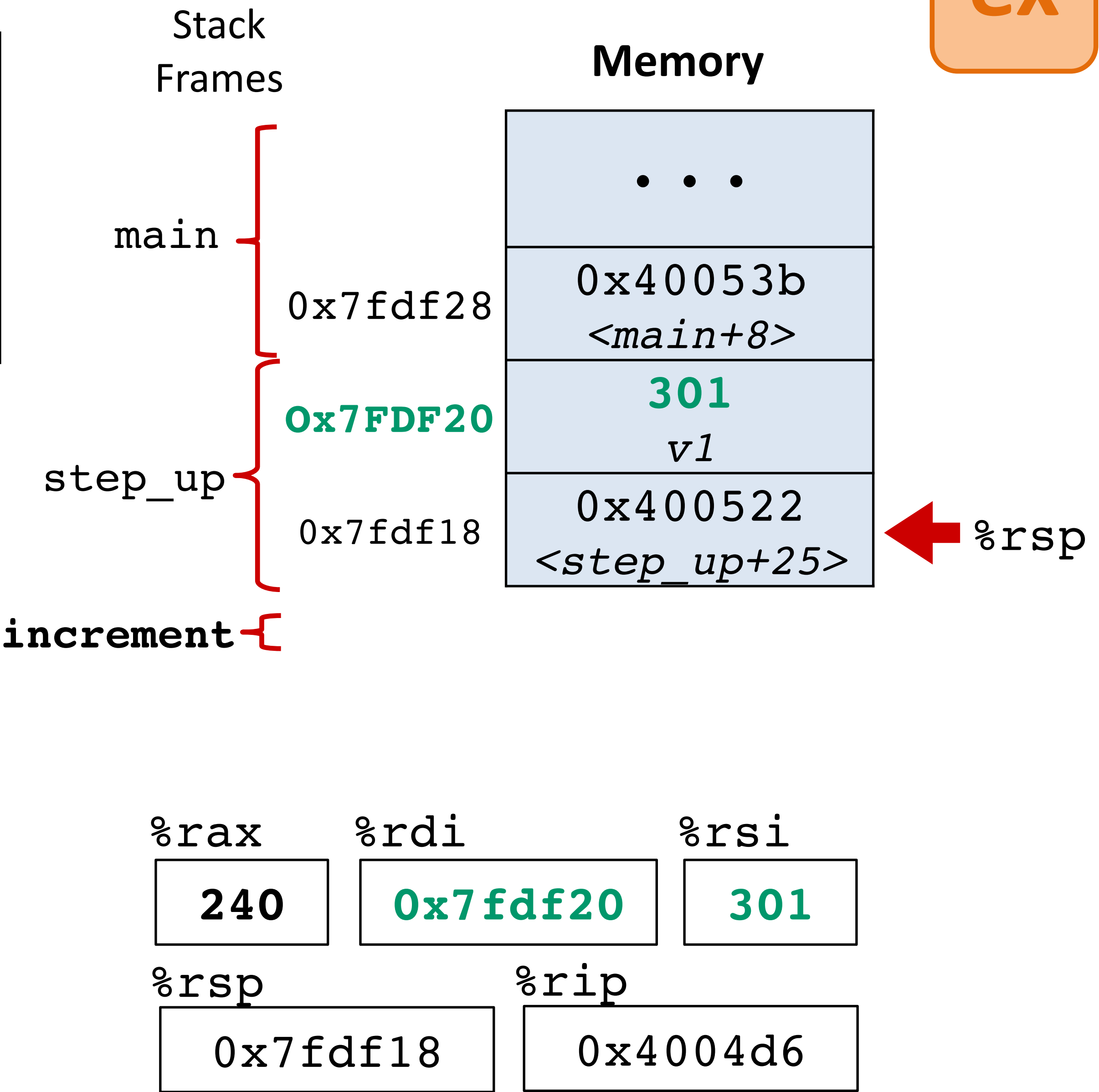
Run increment

ex

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

```
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 5a)

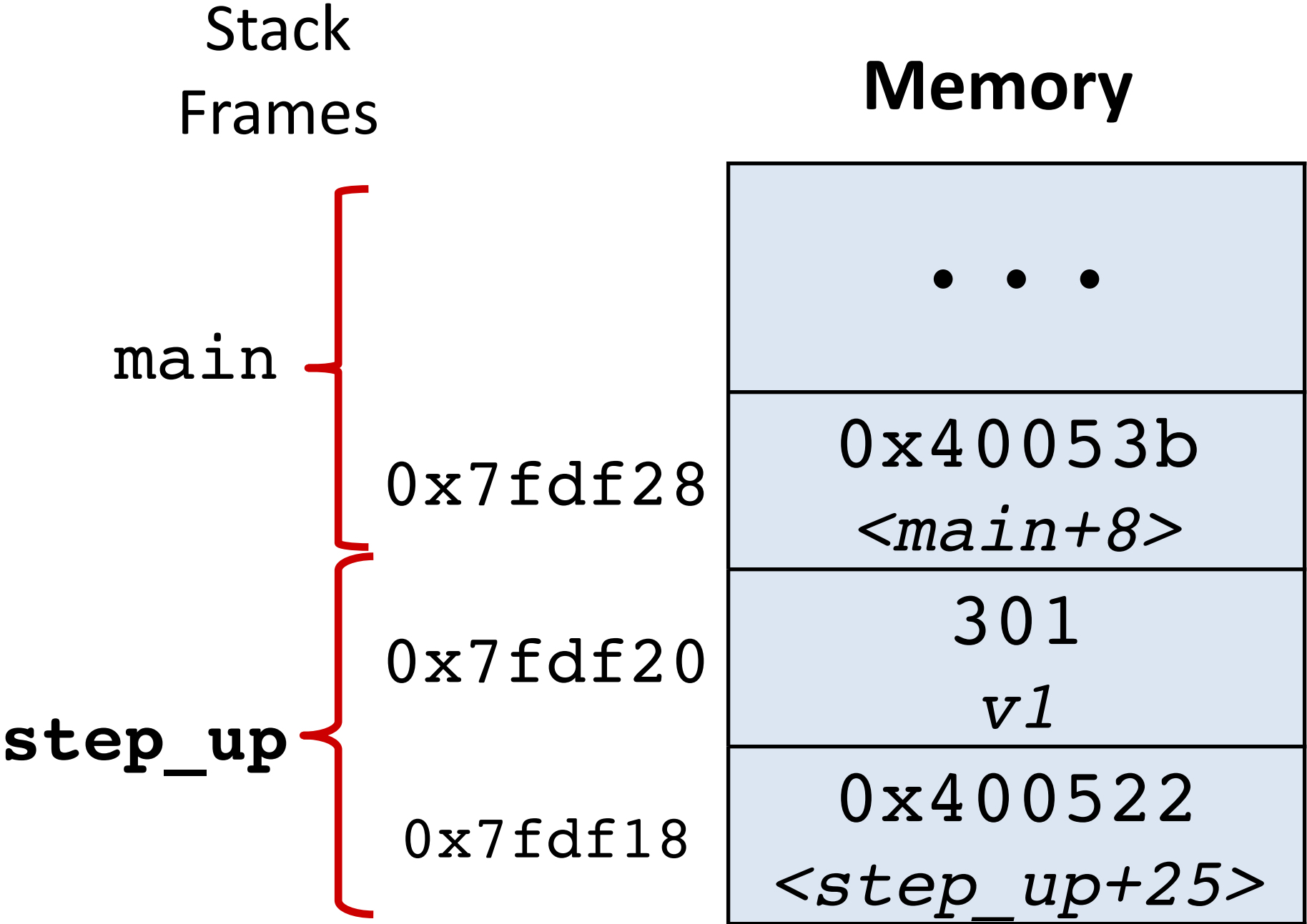
Return from increment
to step_up

ex

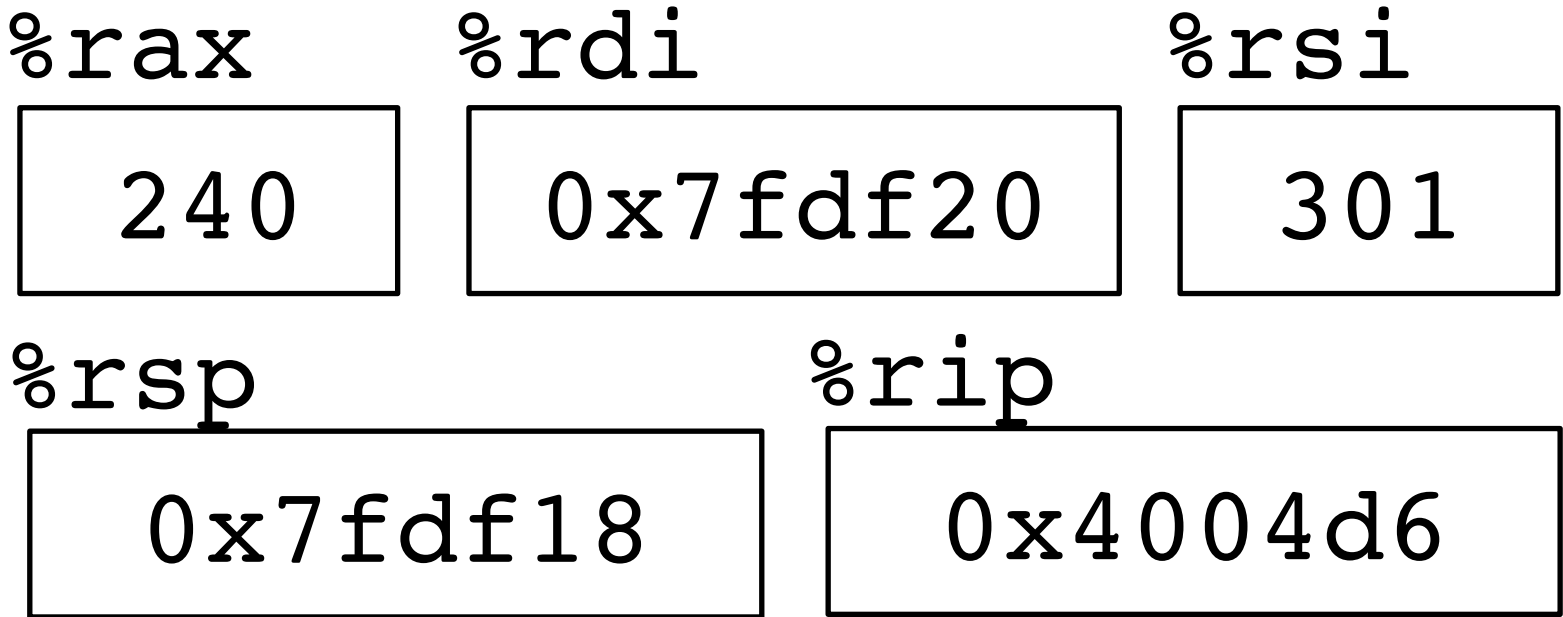
```
10 long increment(long* p, long val) {  
    long x = *p;  
    long y = x + val;  
    *p = y;  
    return x;  
}
```

```
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
```



ret has two steps
(1) pop return address from stack
(2) jump to return address



Execute ret



Procedure call example (step 5b)

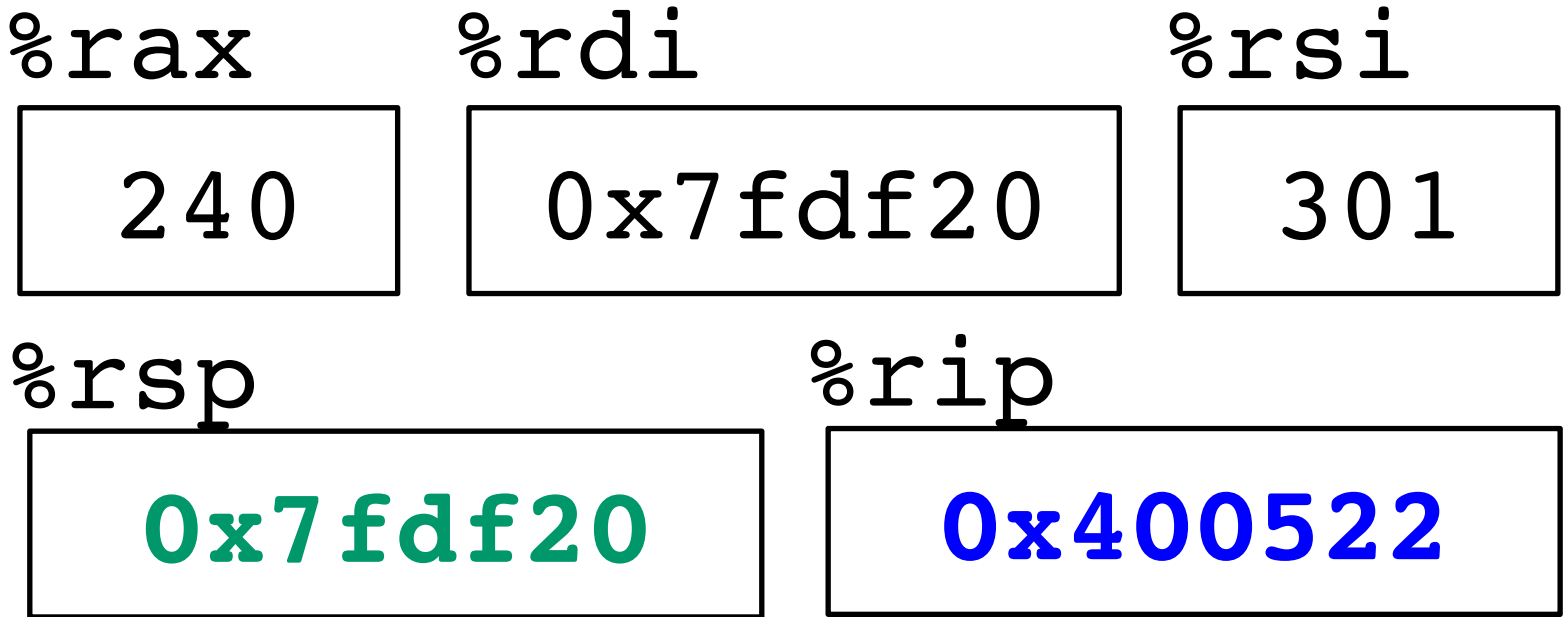
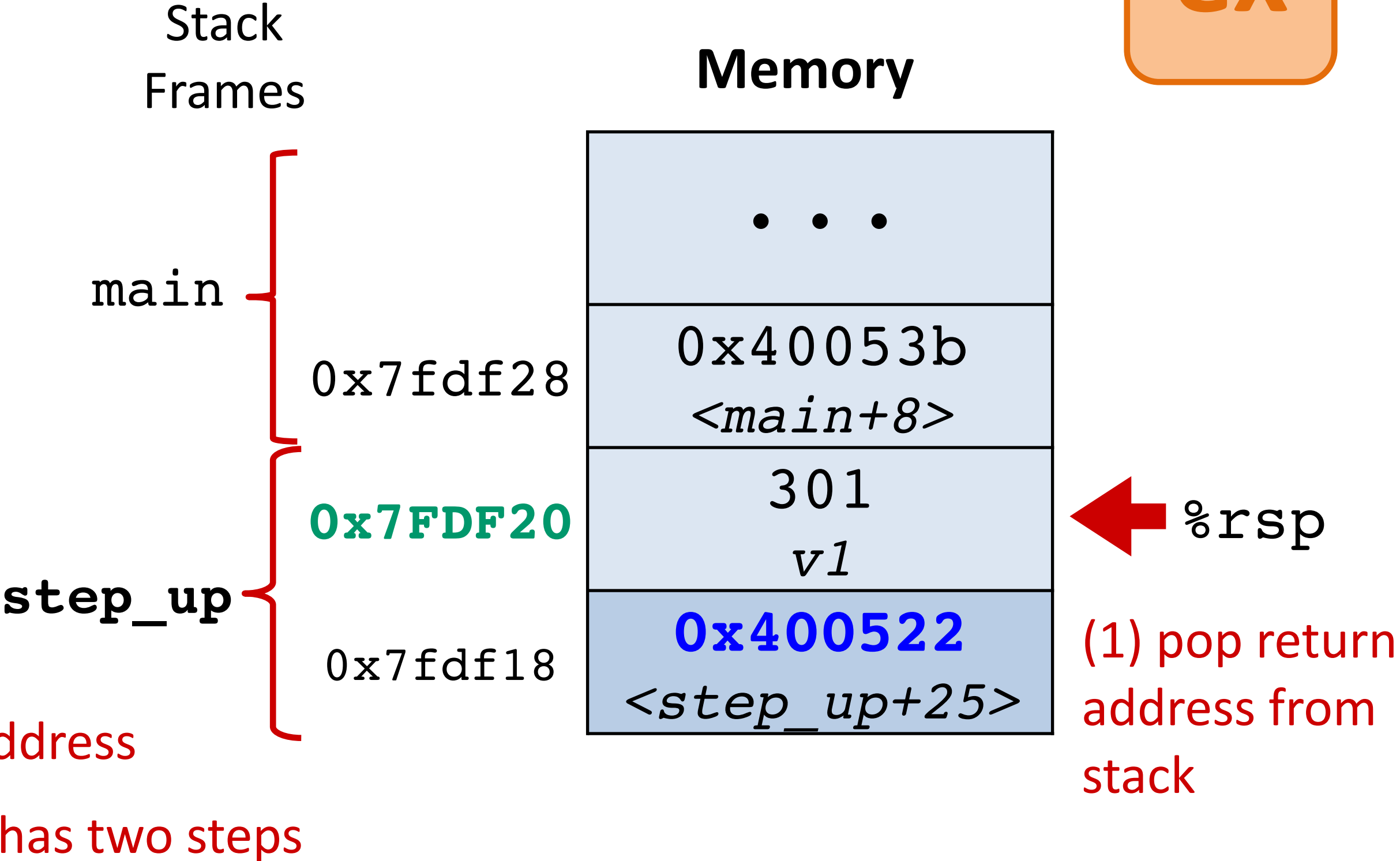
Return from increment
to step_up

ex

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

```
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 6)

Prepare step_up
result



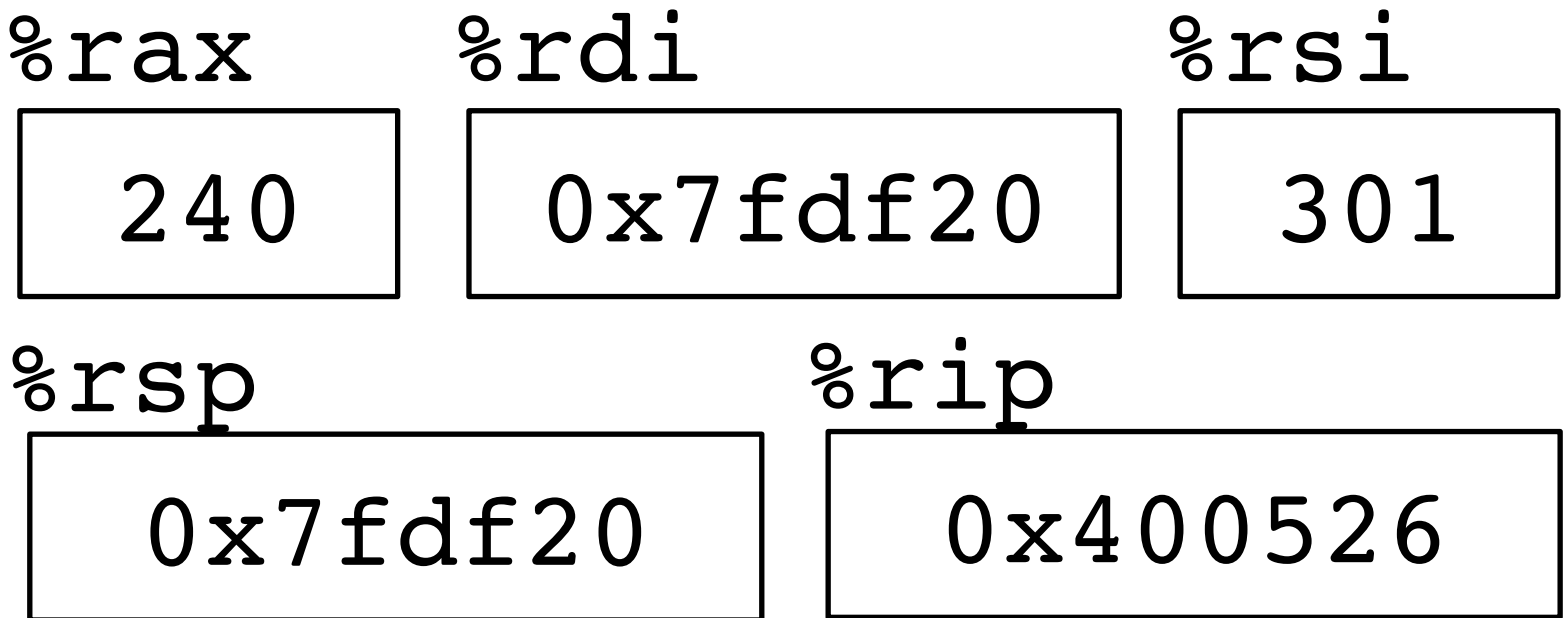
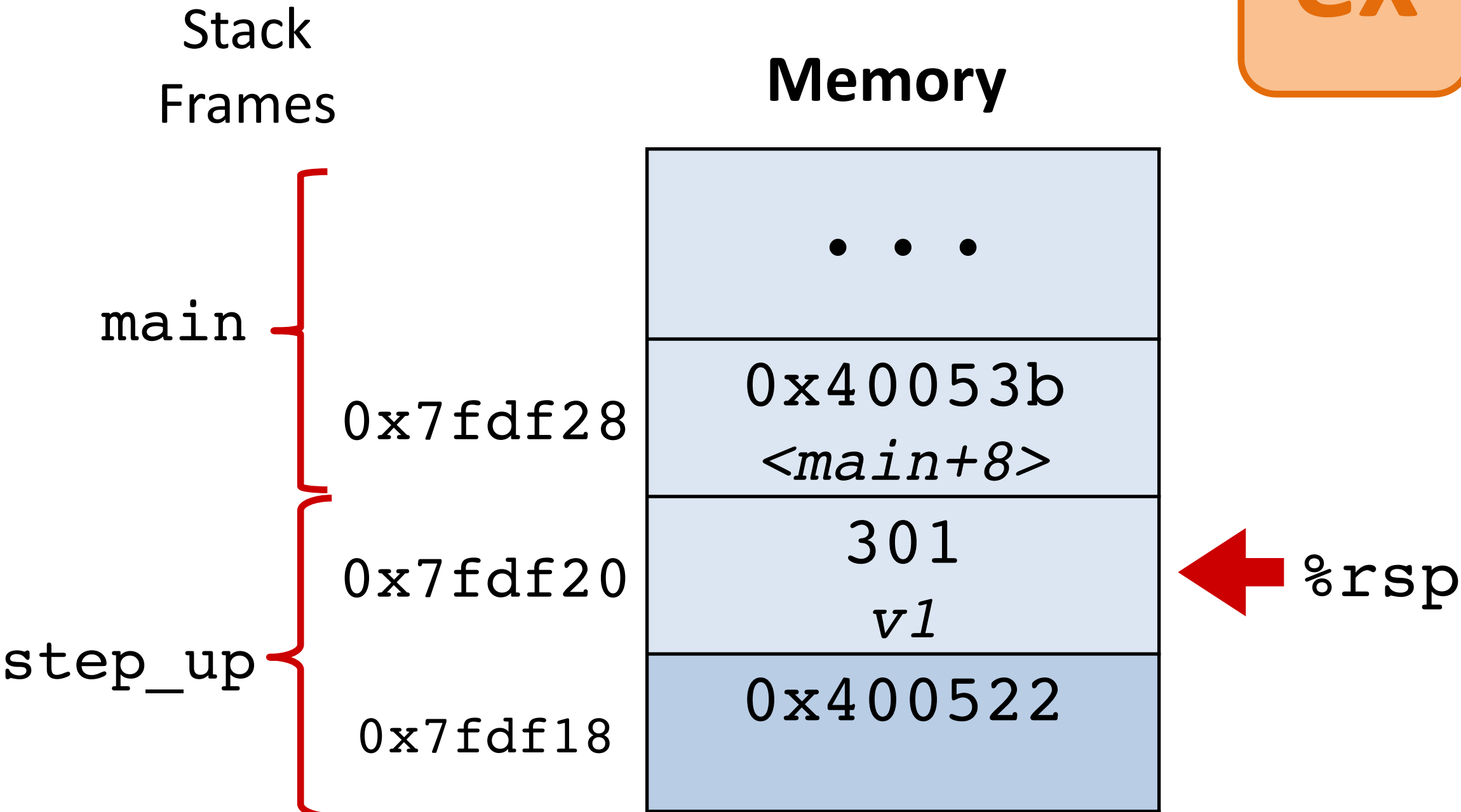
```
long step_up() {
    long v1 = 240;
    long v2 = increment(&v1, 61);
    return v1+v2;
}
```

```
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
```



Execute this instruction

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



Procedure call example (step 6)

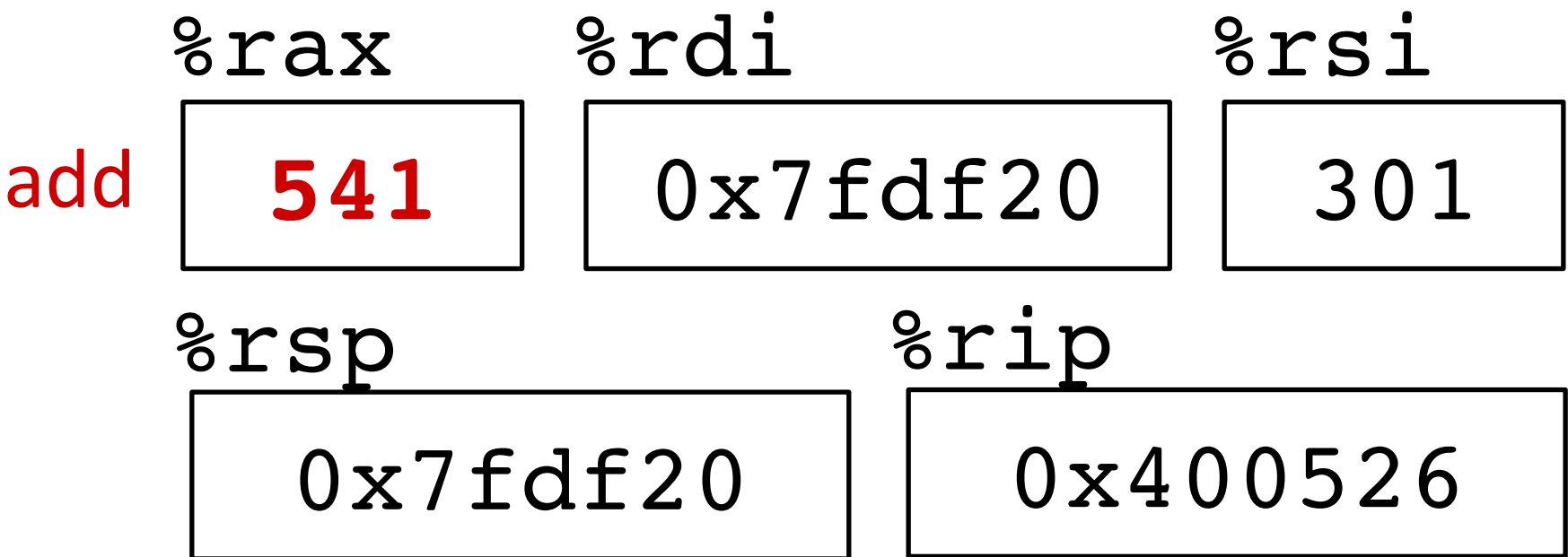
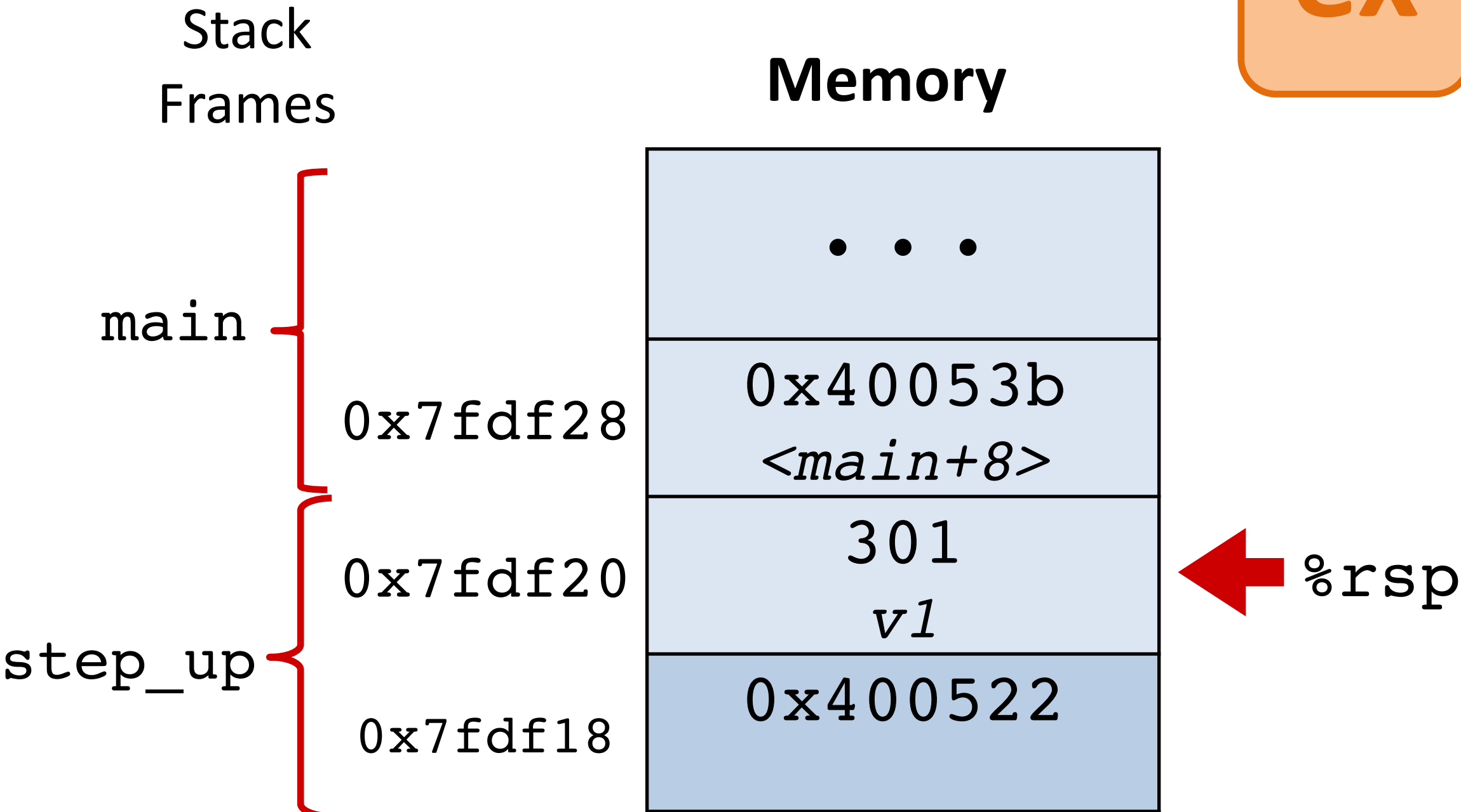
Prepare step_up
result



```
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
```



Procedure call example (step 7)

Deallocate space
for local vars



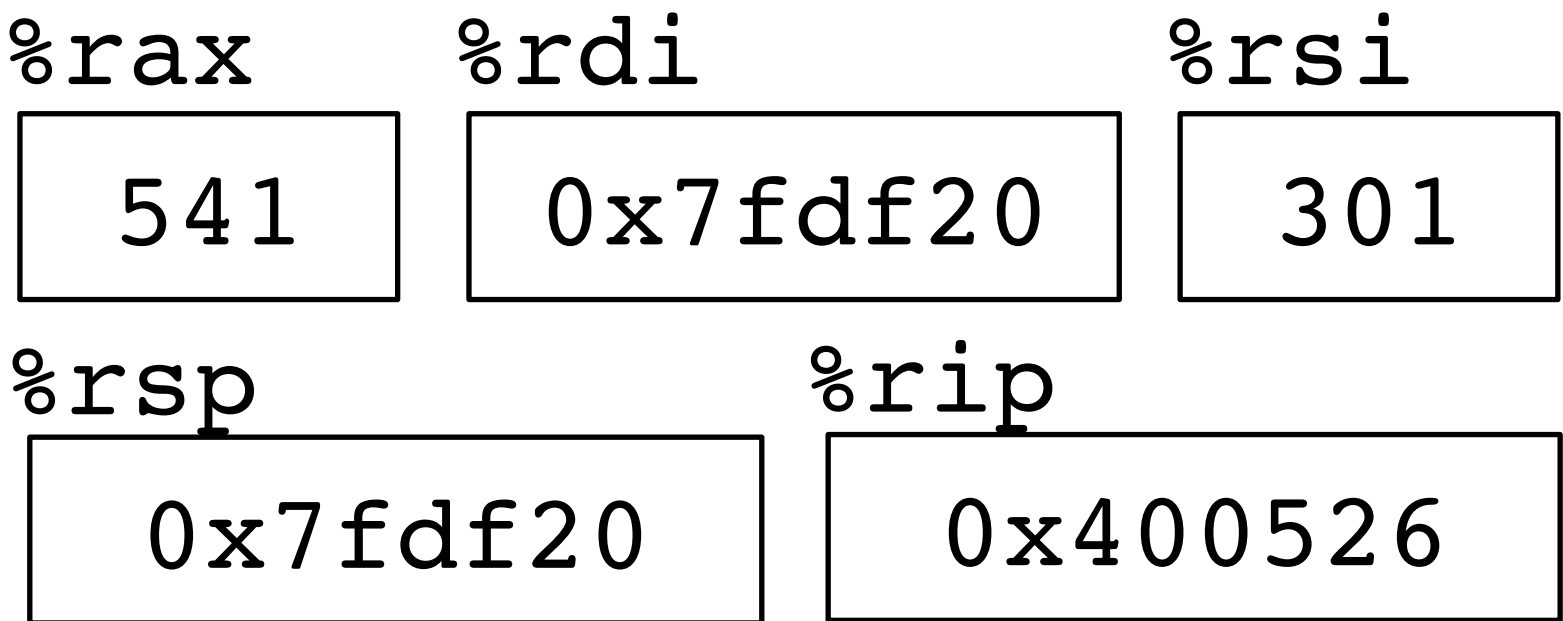
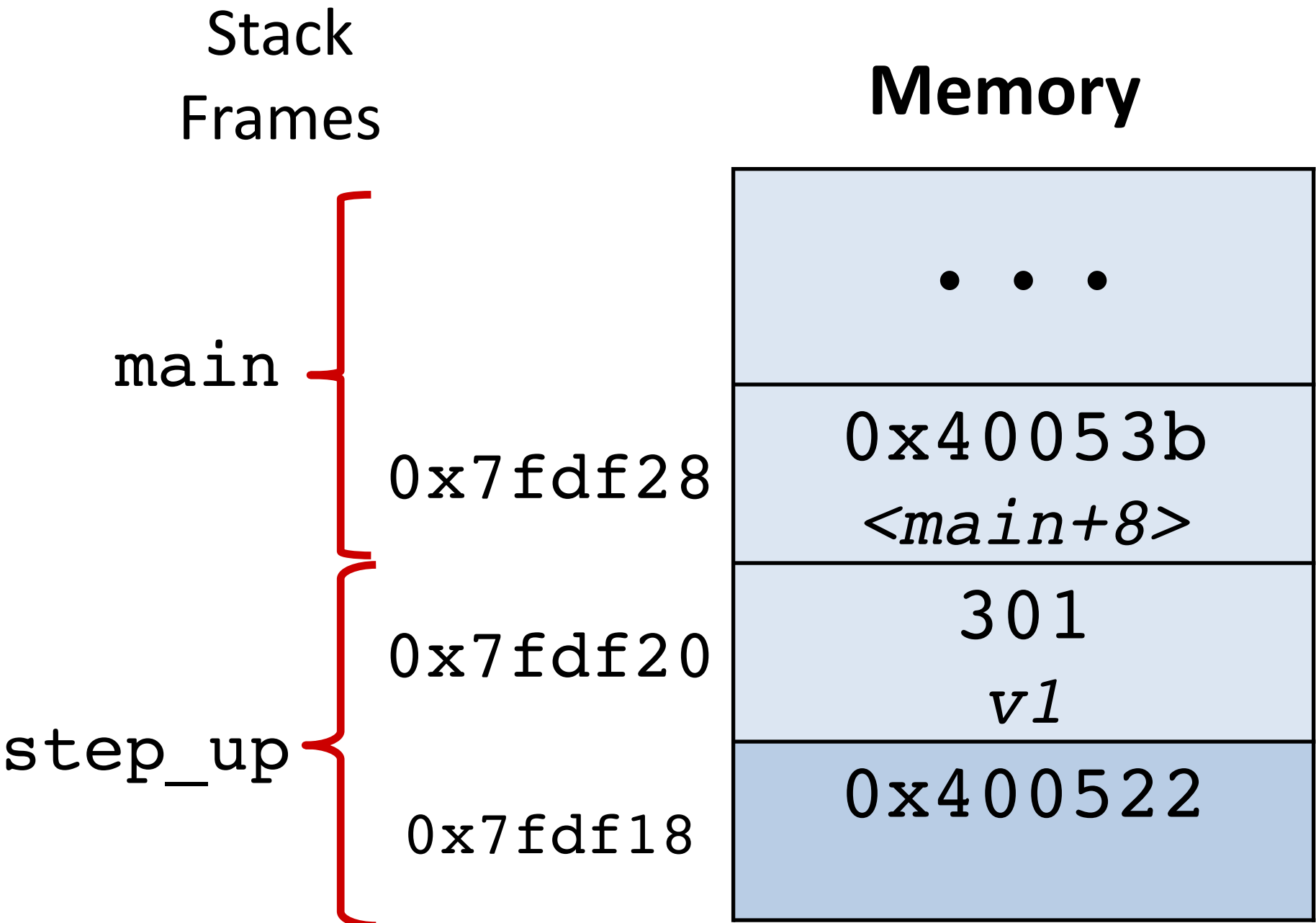
```
long step_up() {  
    long v1 = 240;  
    long v2 = increment(&v1, 61);  
    return v1+v2;  
}
```

```
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
```



Execute this instruction

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



Procedure call example (step 7)

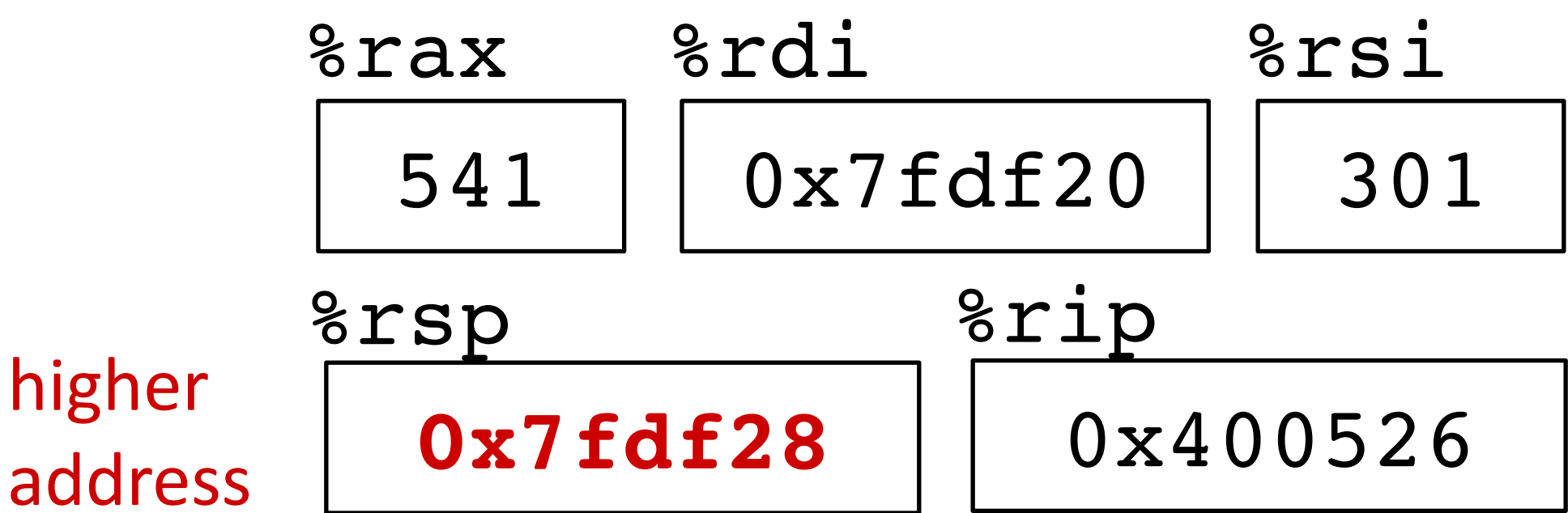
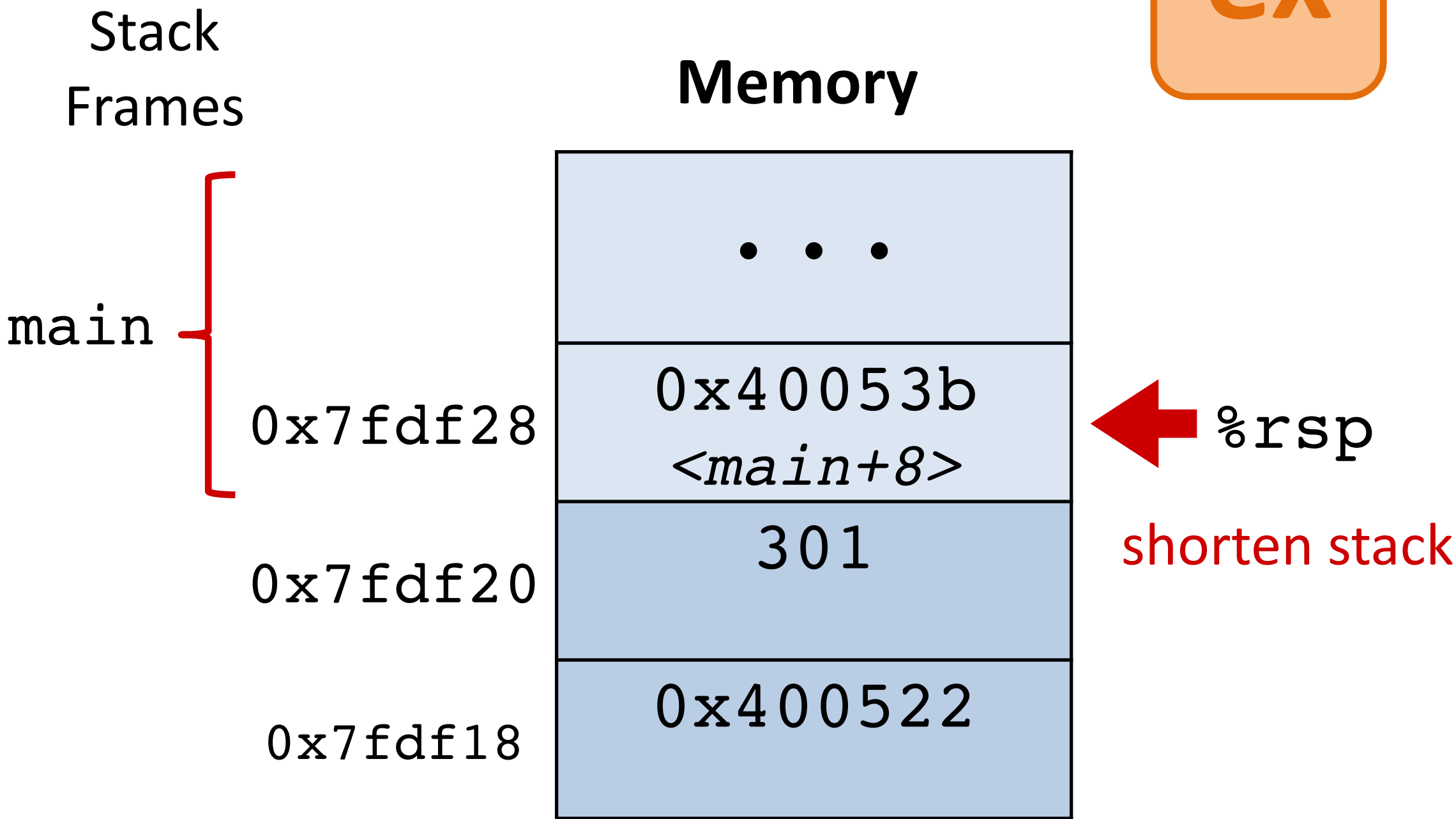
Deallocate space
for local vars



```
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
```



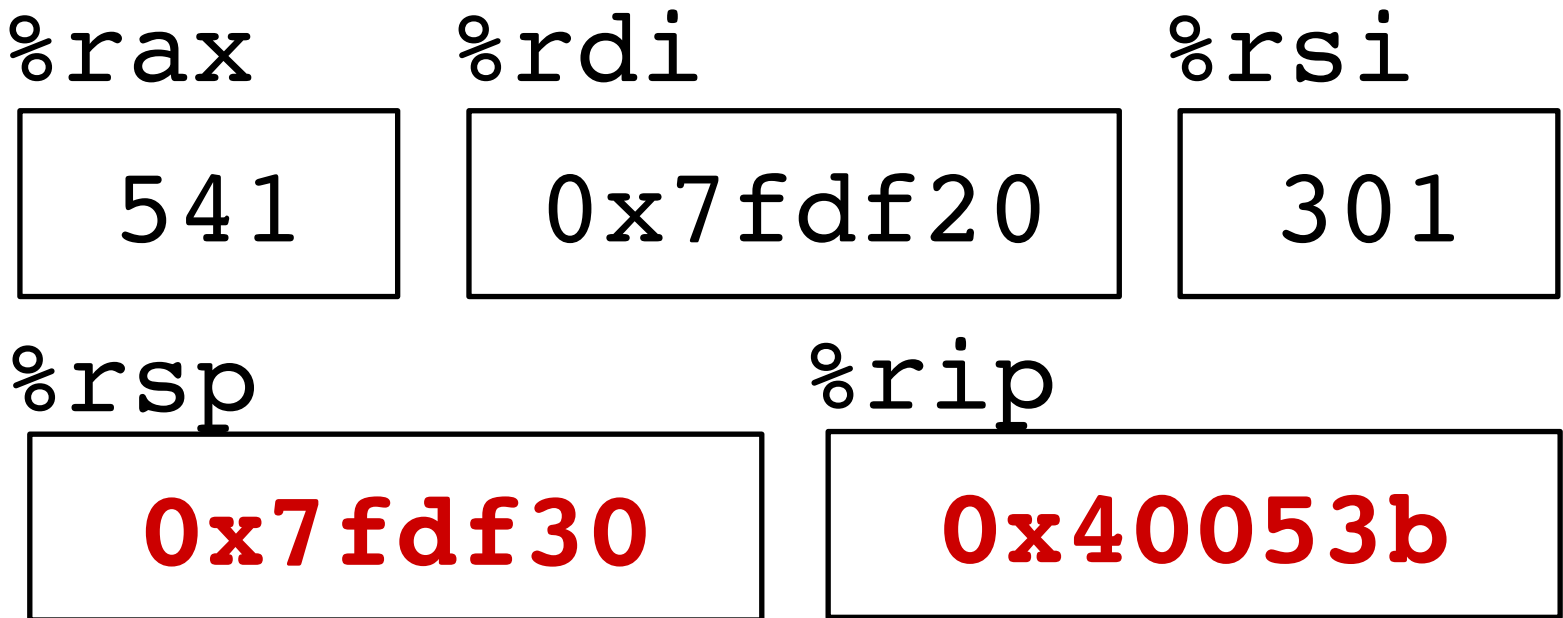
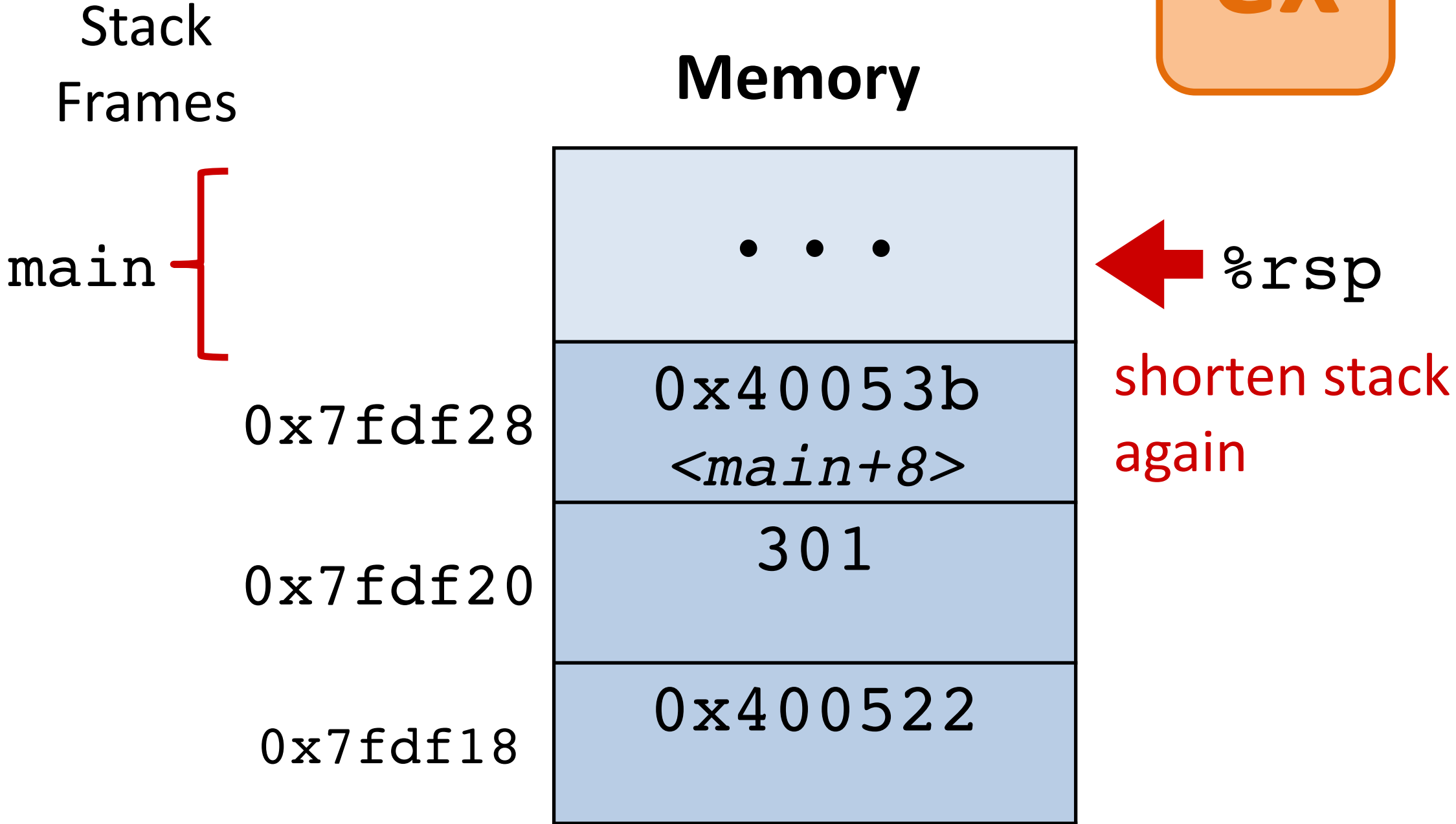
Procedure call example (step 8)

```
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
```

Return from step_up
to main



Implementing procedures

Have we now seen
how this is done?

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



Register saving conventions

yoo calls who:
Caller *Callee*

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

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)

Similar function, but now takes an arg for the local variable

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

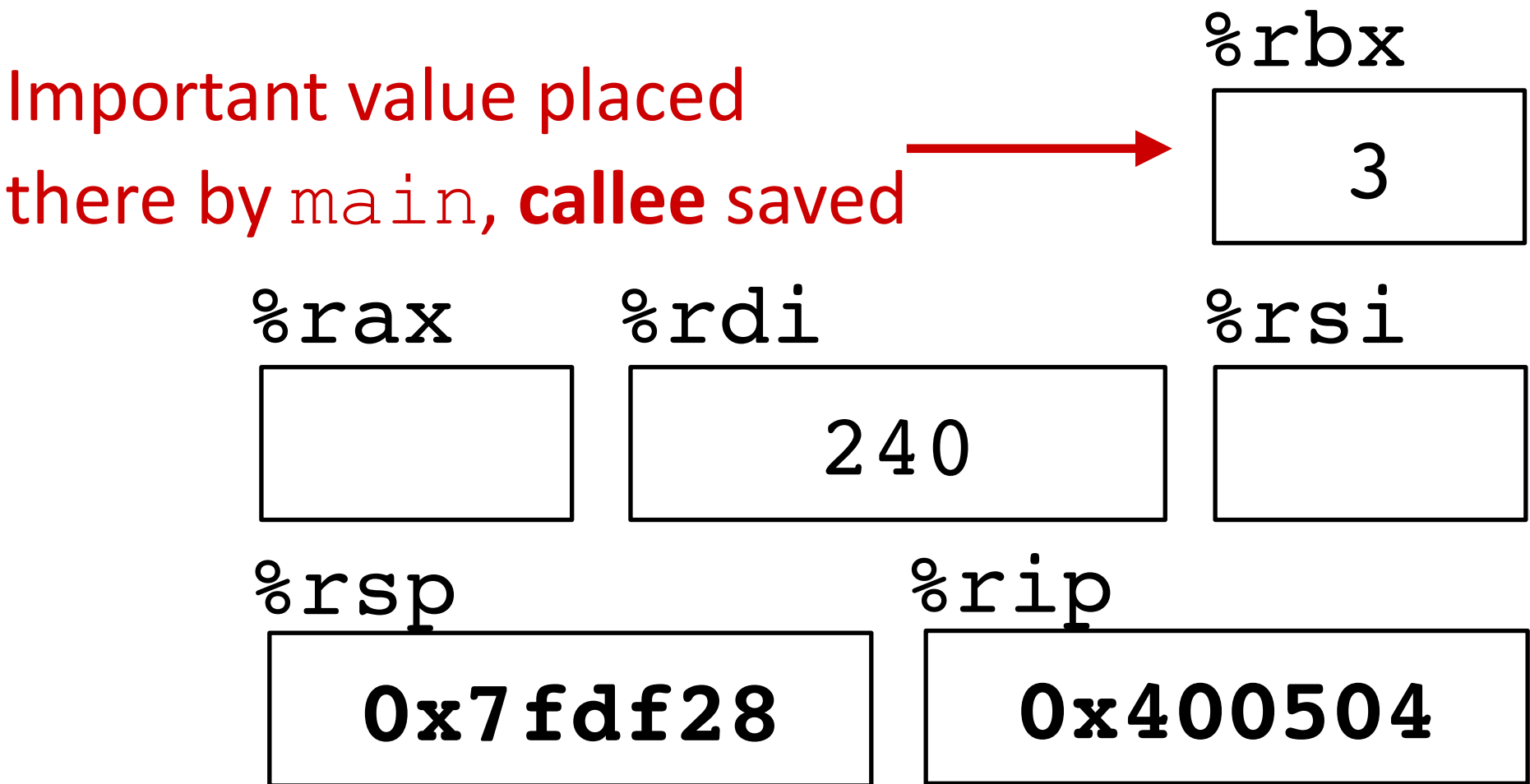
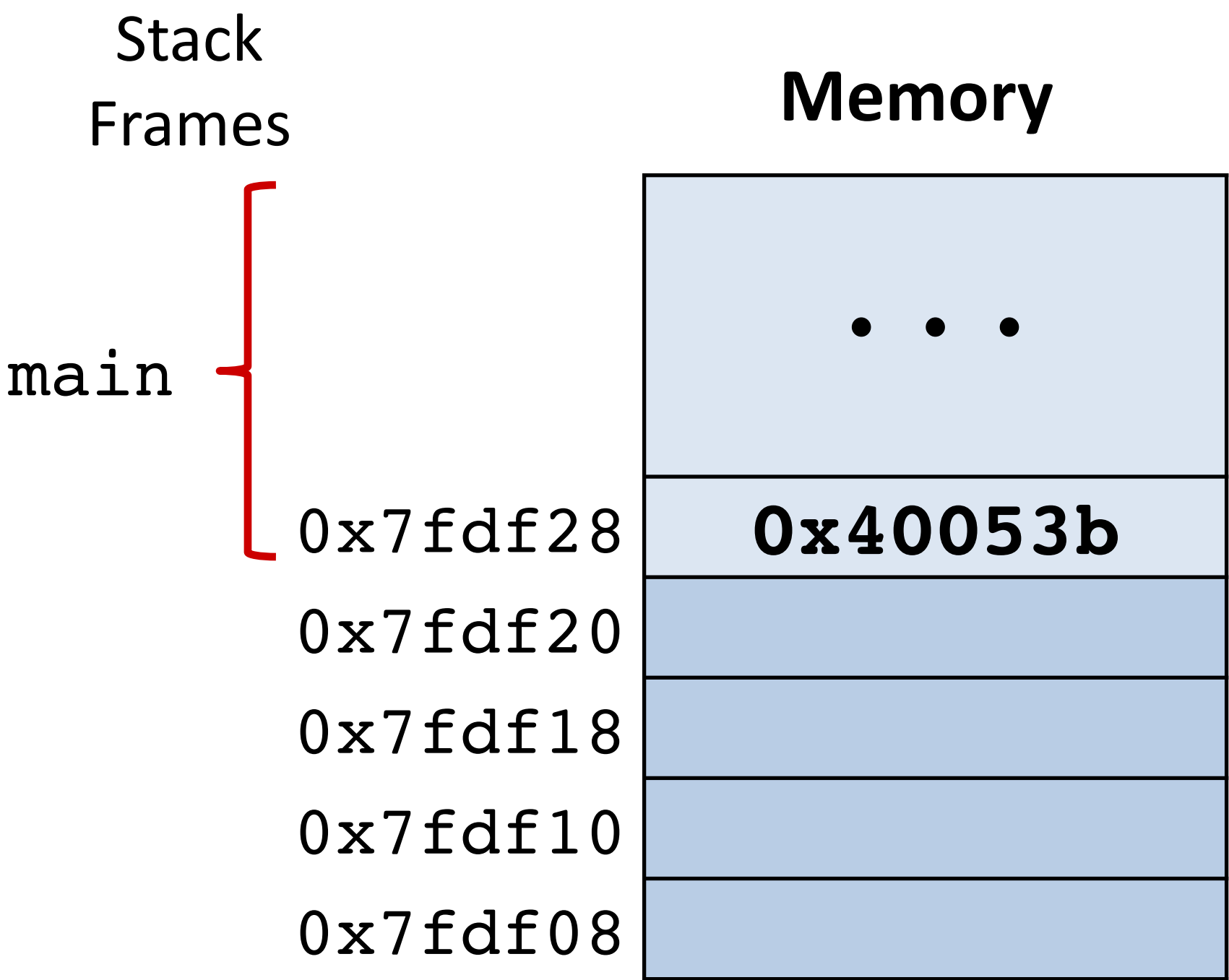
step_by:

```
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
```

caller saved: %rax, %rdi, %rsi

callee saved: %rbx

main called step_by(240)



Callee-save example (step 1)

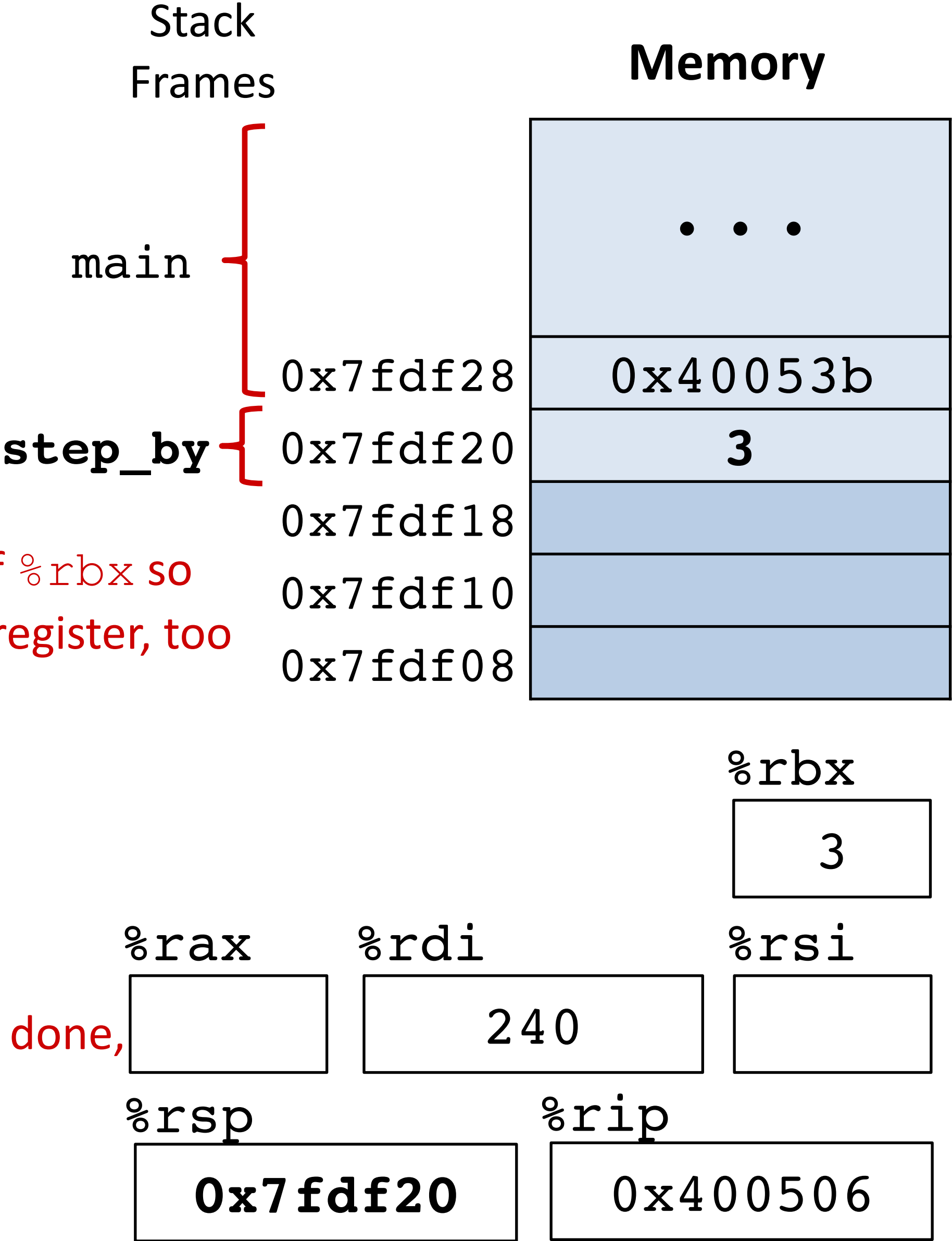
Save register %rbx

```
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
```

Save the value of %rbx so we can use that register, too

Once this function is done, restore saved value



caller saved: %rax, %rdi, %rsi

callee saved: %rbx

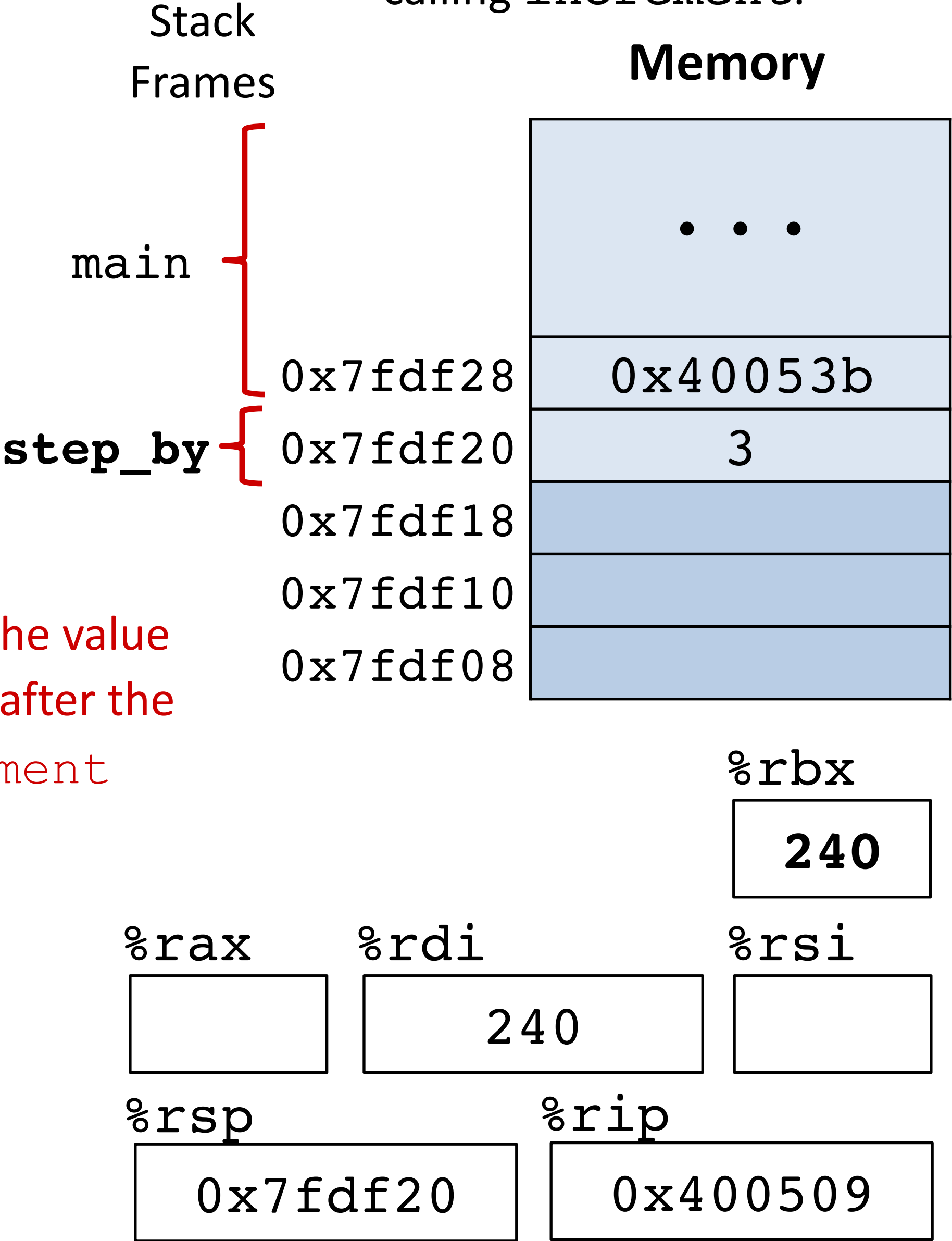
Callee-save example (step 2)

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
```

Need to save the value x to use later, after the call to increment



caller saved: %rax, %rdi, %rsi

callee saved: %rbx

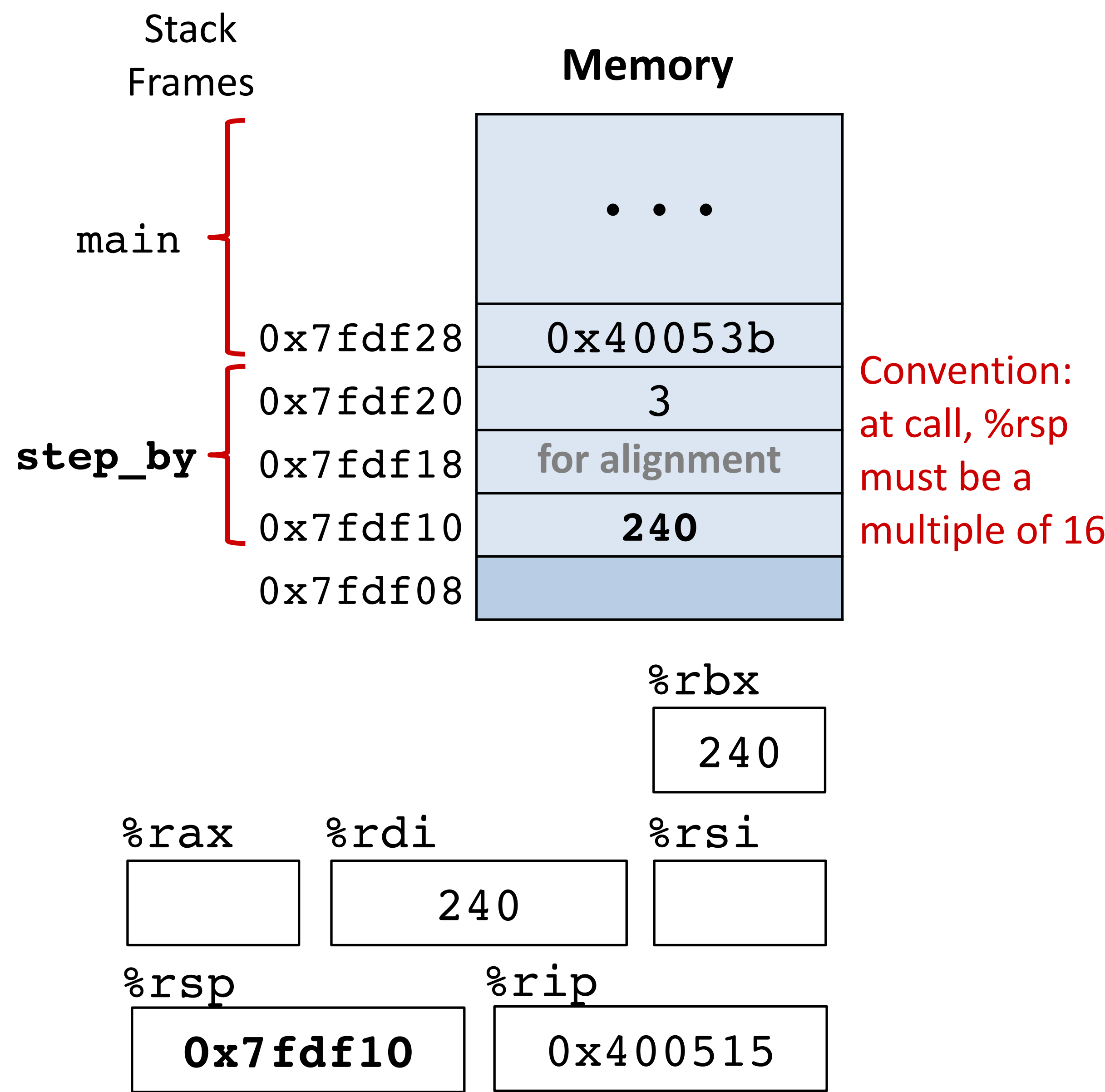
Callee-save example (step 3)

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
```

caller saved: %rax, %rdi, %rsi
callee saved: %rbx



Callee-save example (step 4)

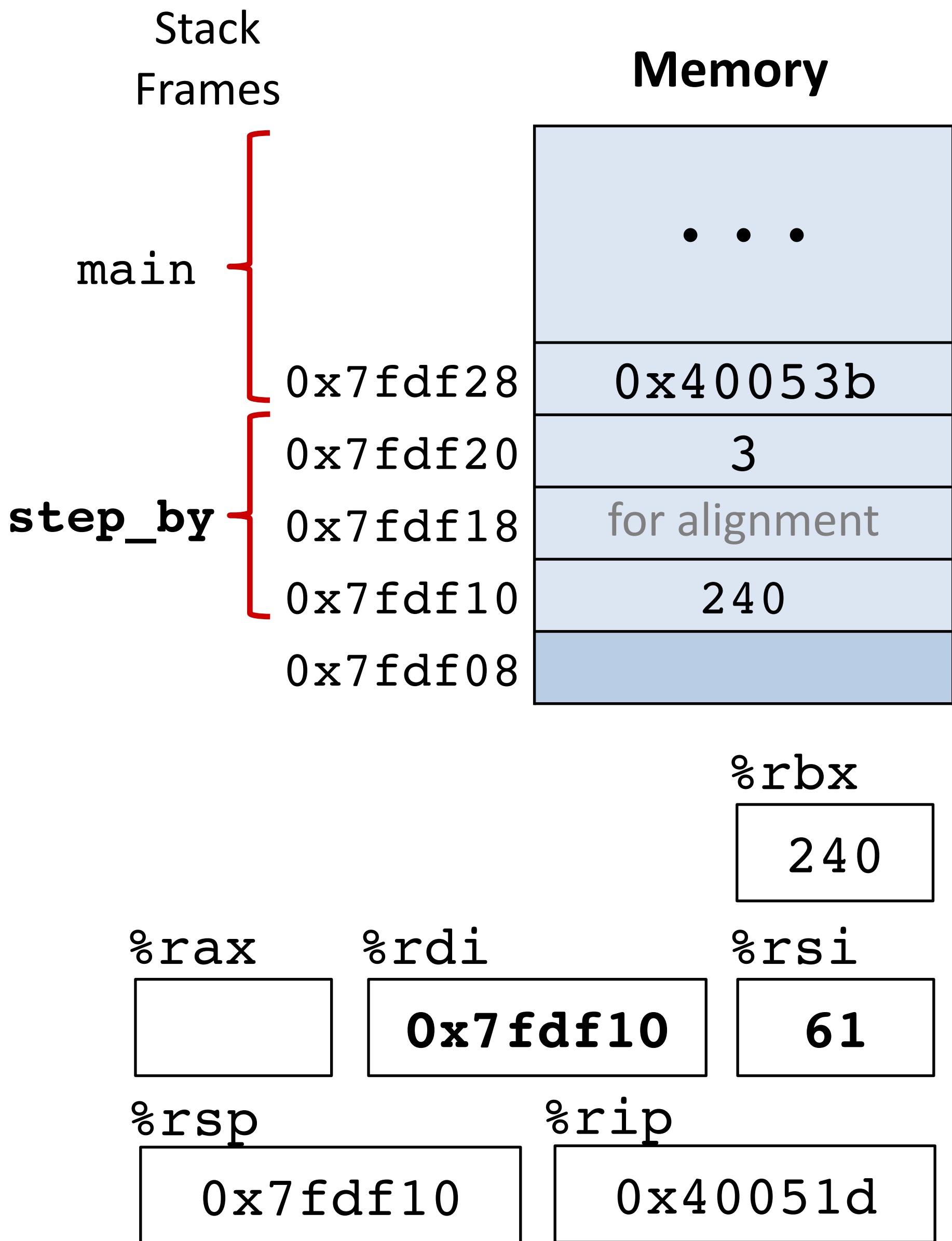
Set up arguments

```
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
```

caller saved: %rax, %rdi, %rsi

callee saved: %rbx



Callee-save example (step 5)

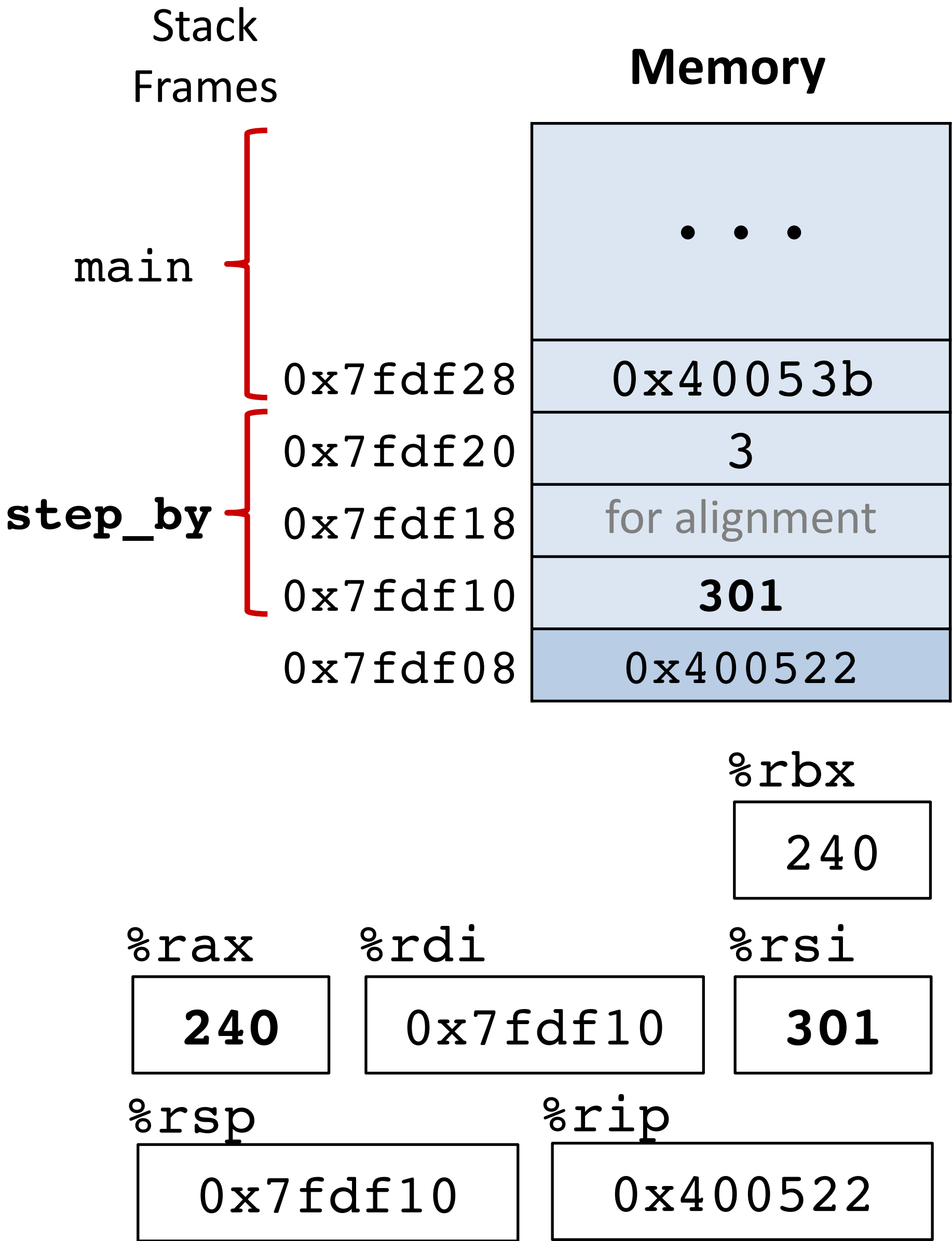
Call, execute, and return
from increment

```
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
```

caller saved: %rax, %rdi, %rsi

callee saved: %rbx

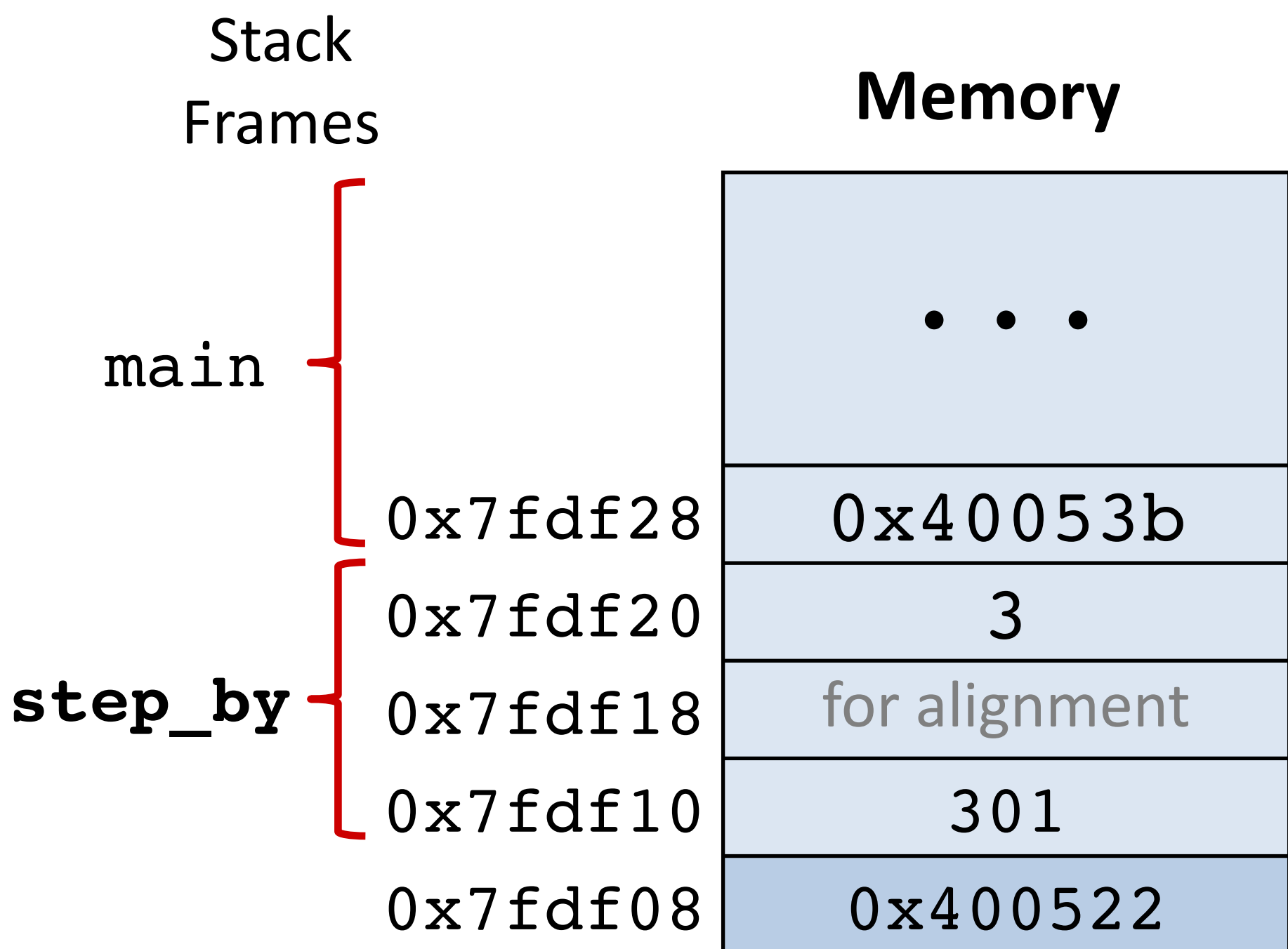


Callee-save example (step 6)

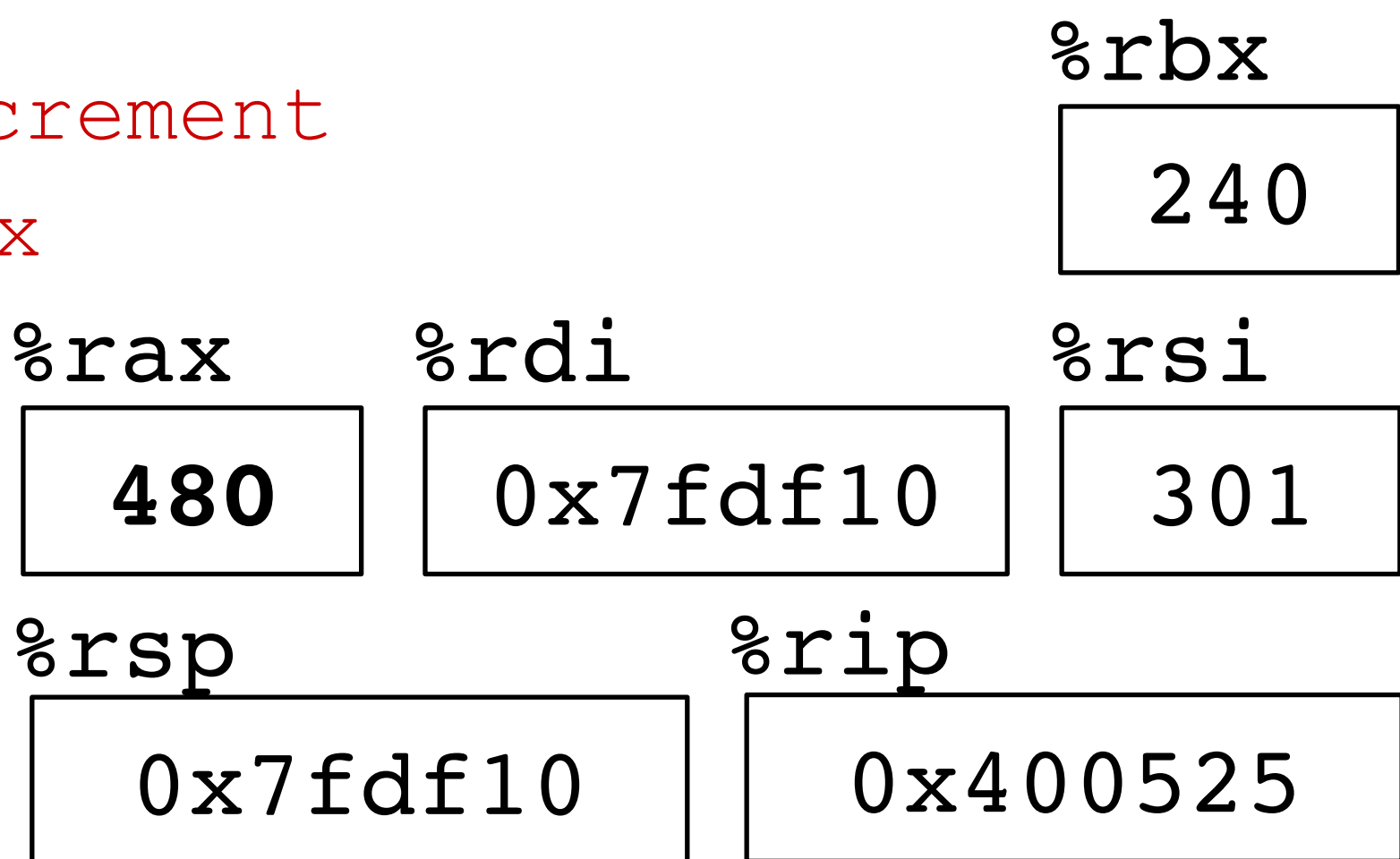
Prepare return value

```
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
```



We know *increment*
restored *%rbx*



caller saved: %rax, %rdi, %rsi

callee saved: %rbx

Callee-save example (step 7)

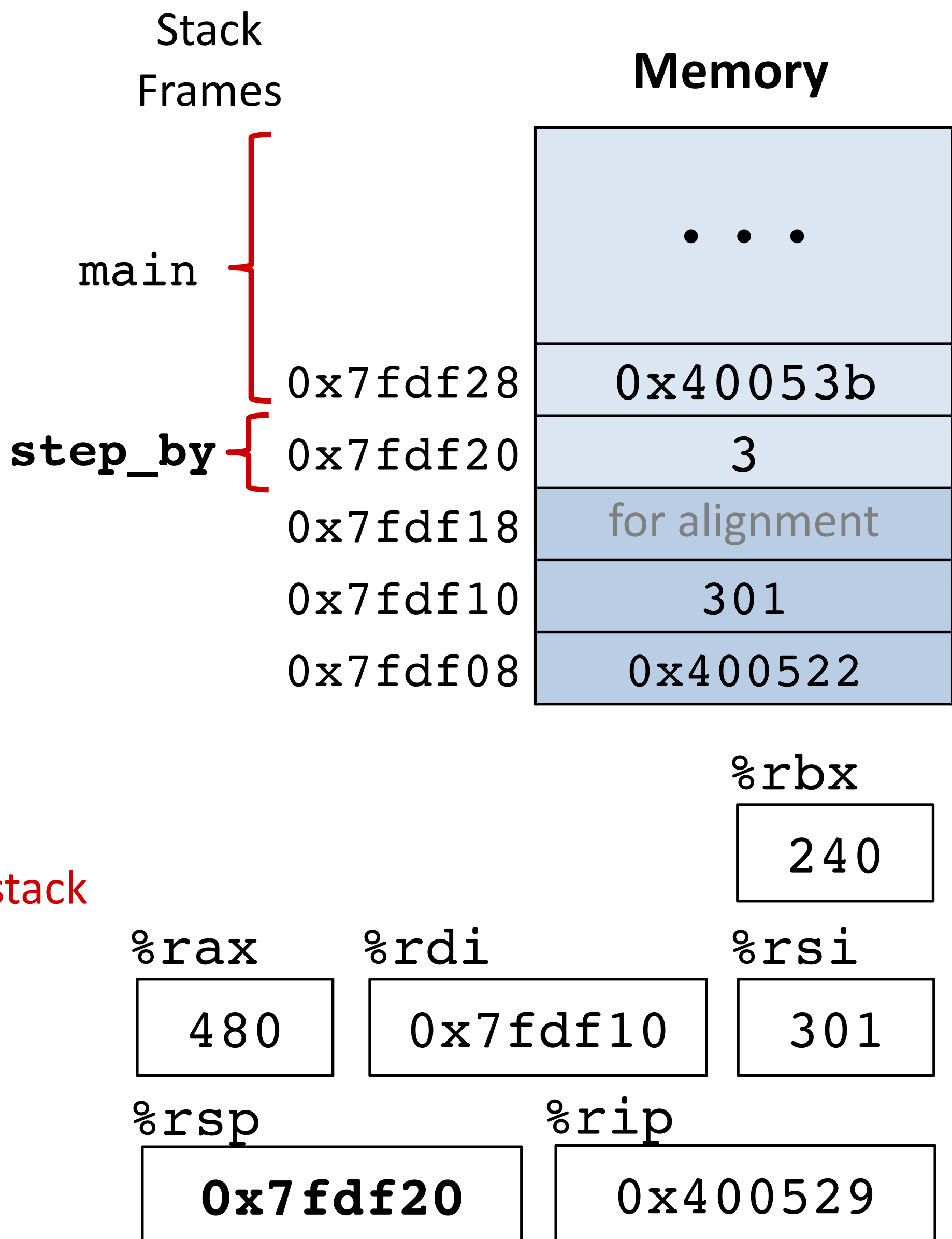
Clean up stack frame

```
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
```

caller saved: %rax, %rdi, %rsi

callee saved: %rbx

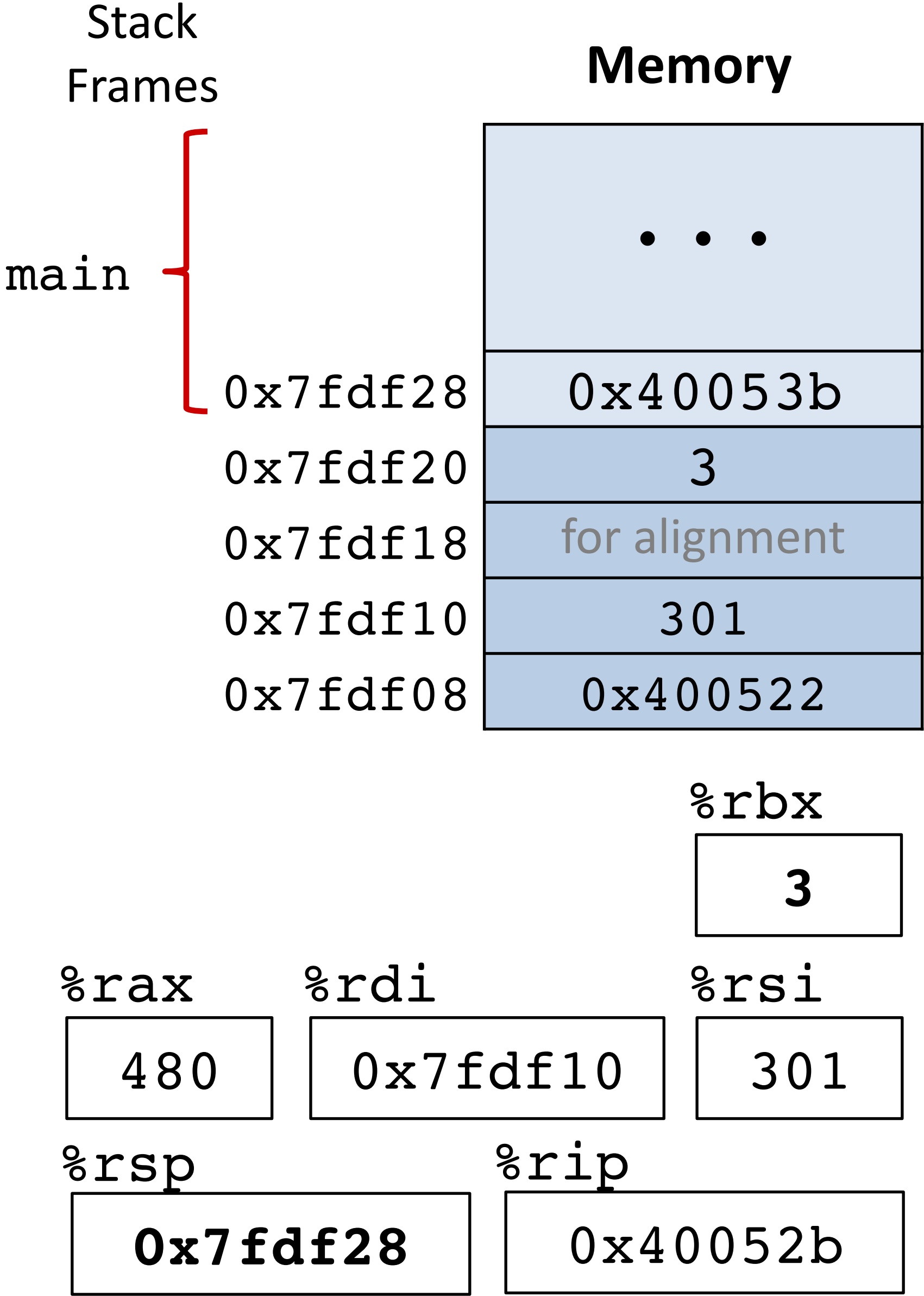


Callee-save example (step 8)

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 ← Restore %rbx for main  
40052b:  retq
```



caller saved: %rax, %rdi, %rsi

callee saved: %rbx

Recursion example: code

```
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

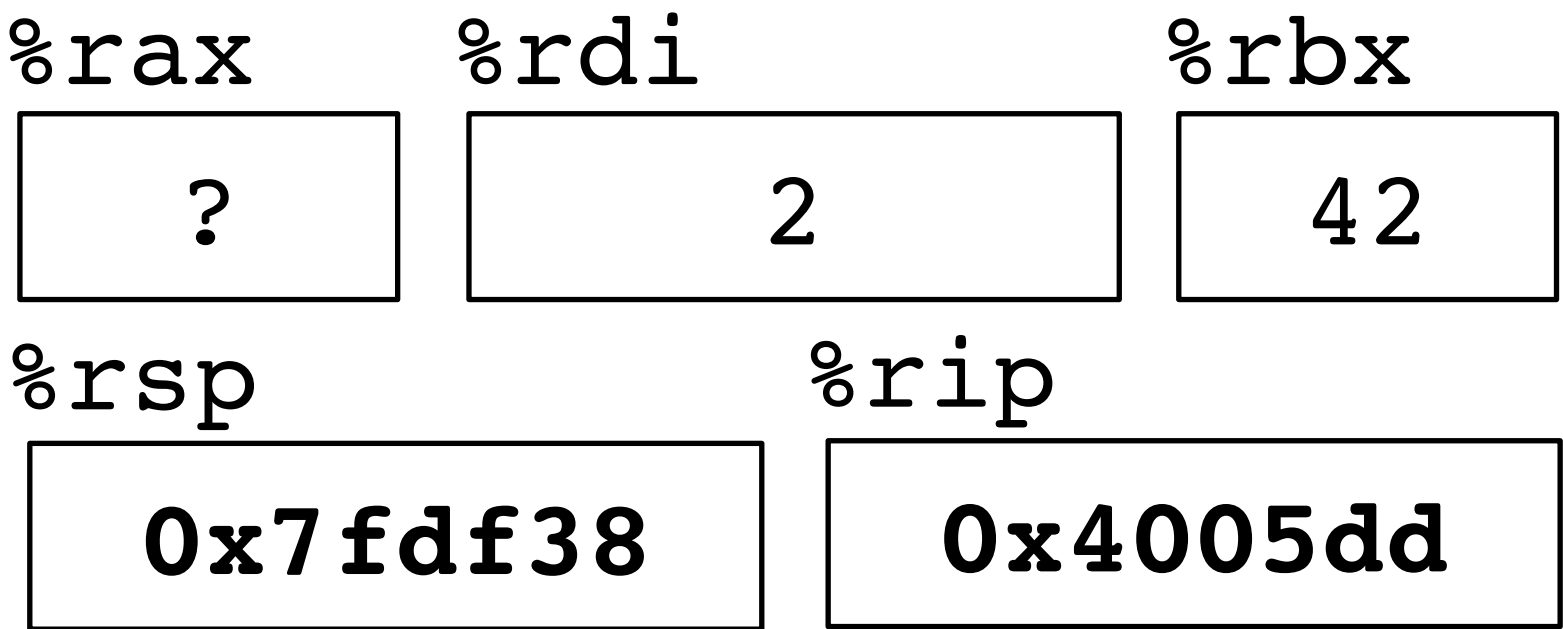
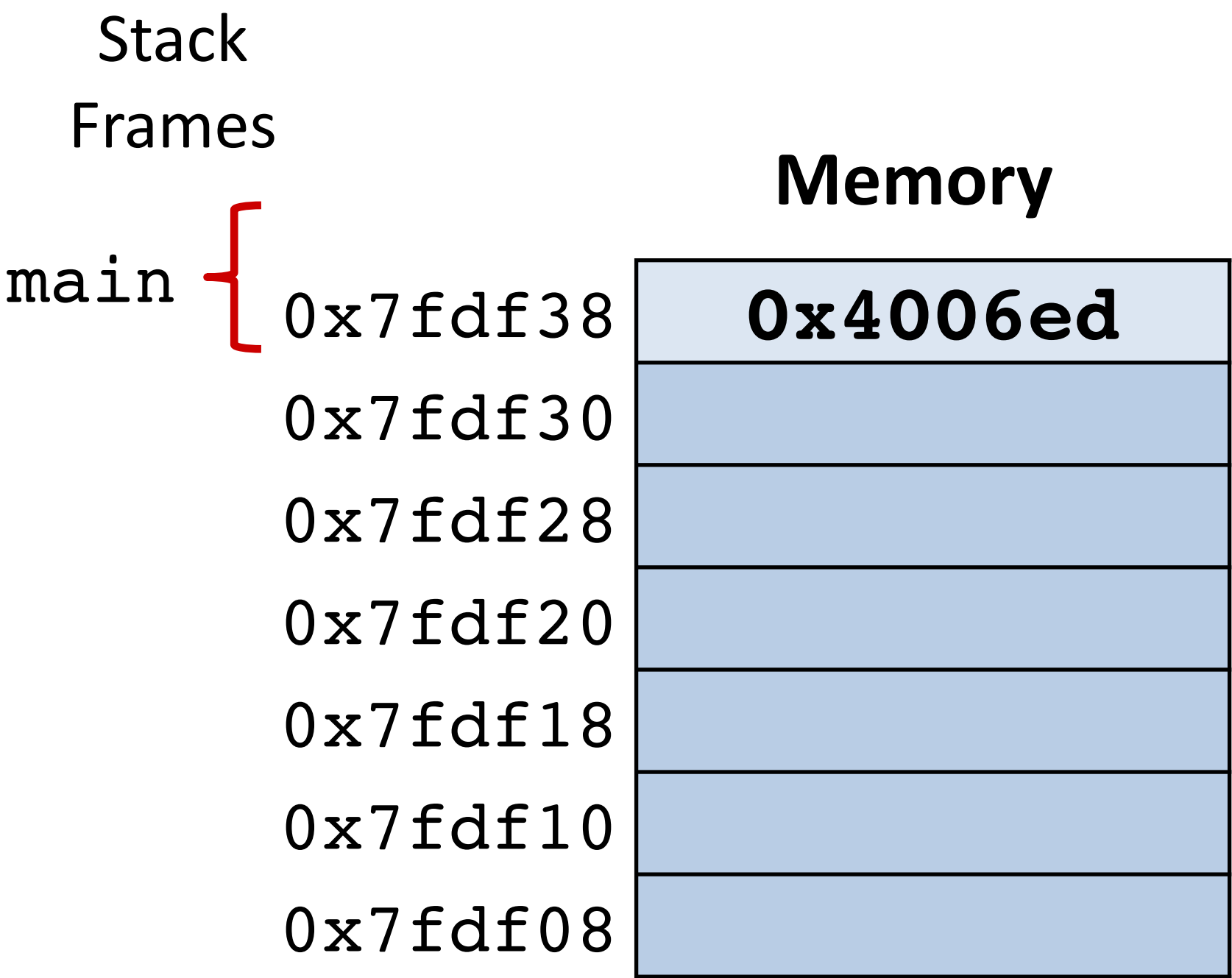
%rbx (callee-save)

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
```

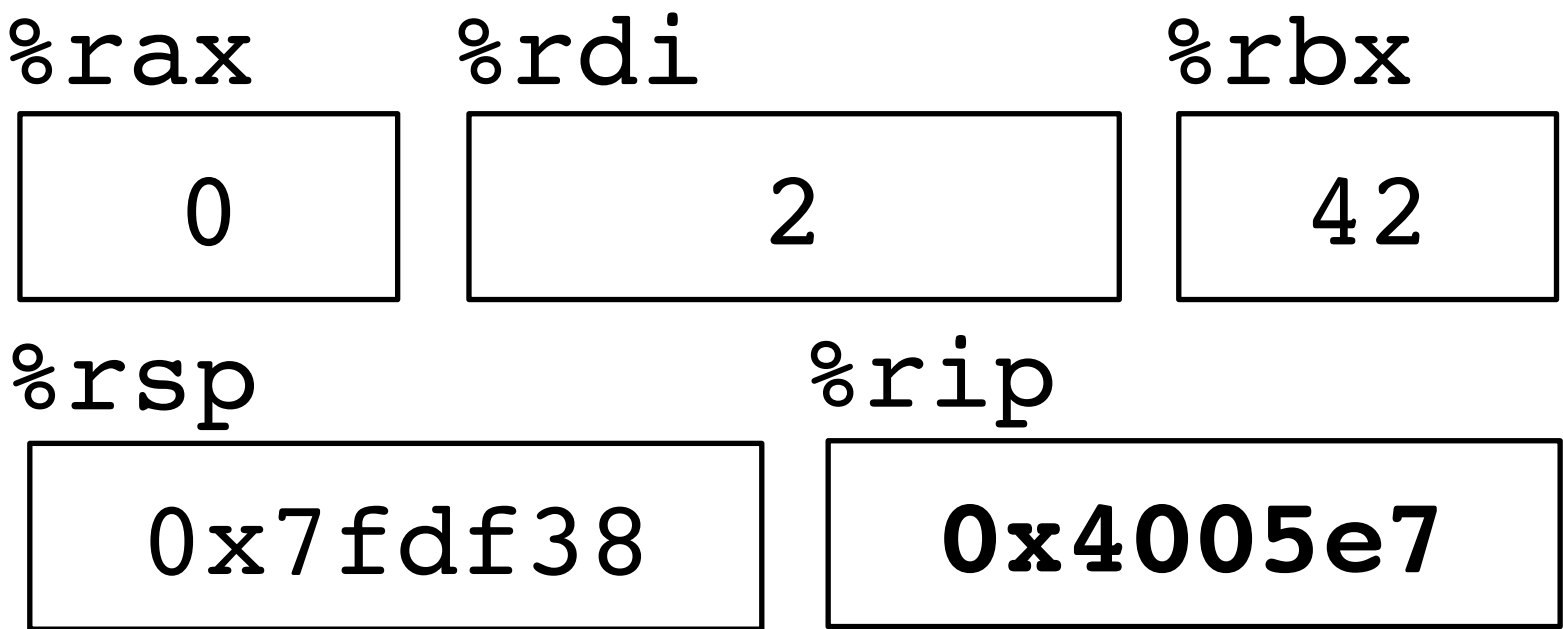
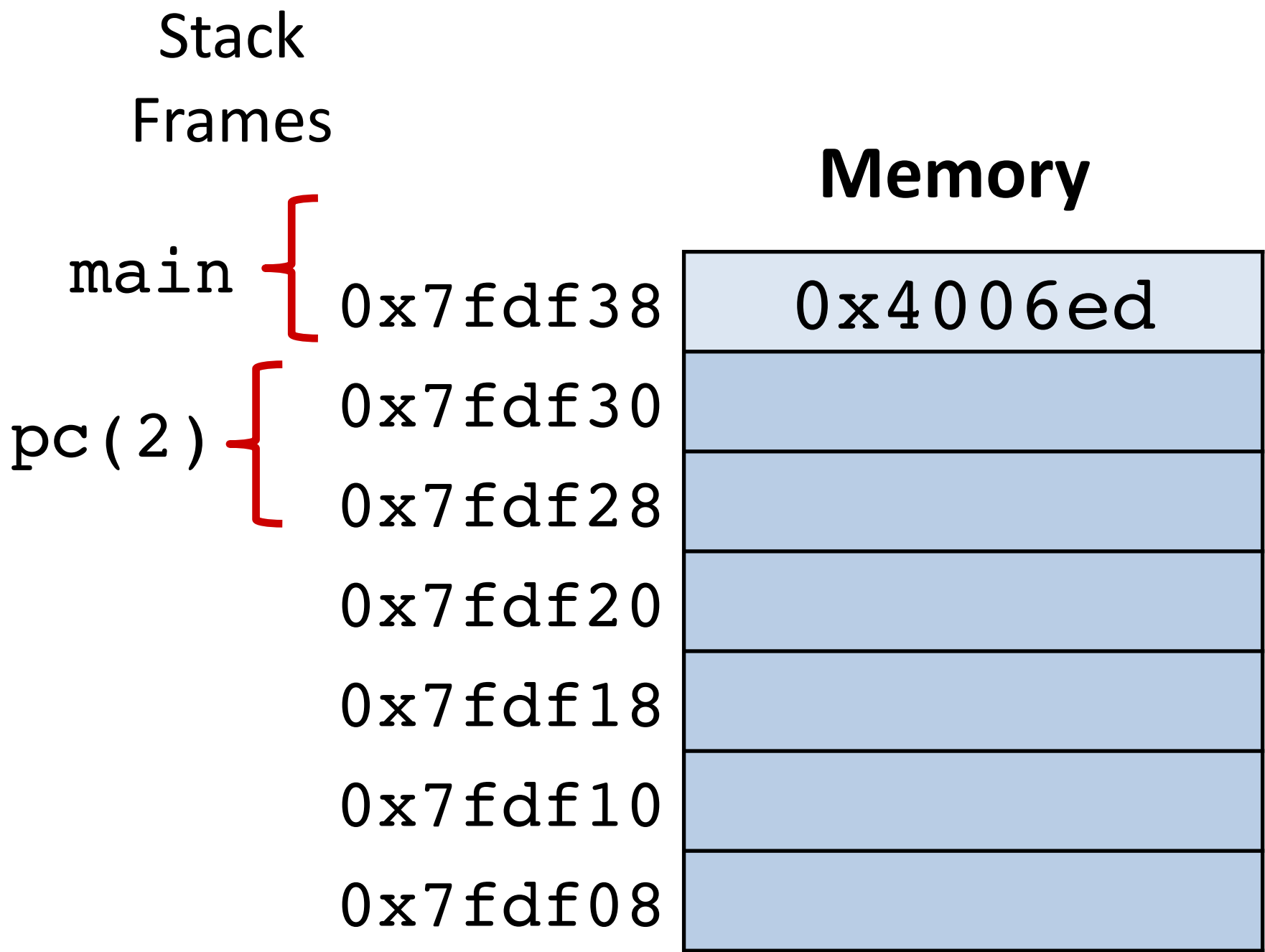


Recursion Example: pcount (2)

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

pcount:

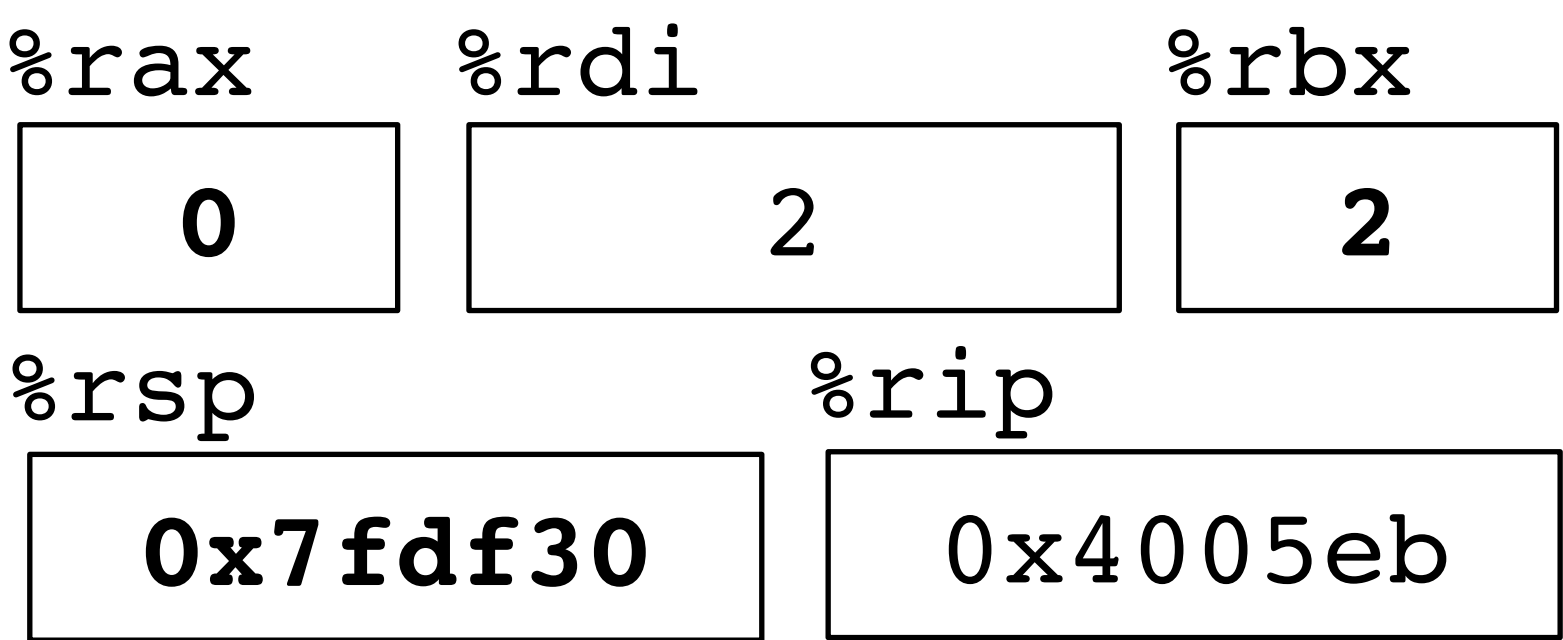
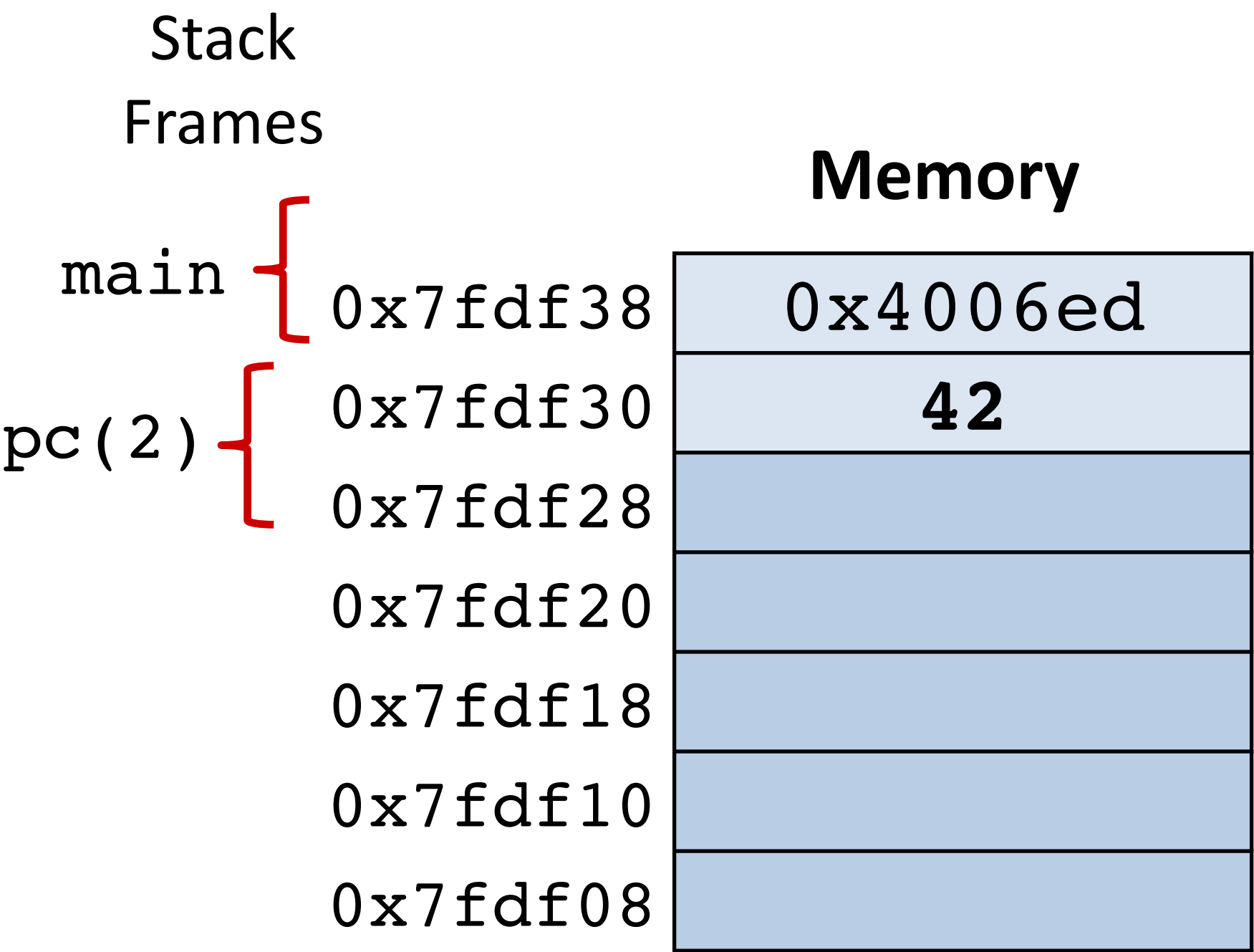
```
4005dd: movl $0, %eax
4005e2: testq %rdi, %rdi
4005e5: je 4005fa <.L6>
4005e7: pushq %rbx
4005e8: movq %rdi, %rbx
4005eb: andl $1, %ebx
4005ee: shrq %rdi
4005f1: callq pcount
4005f6: addq %rbx, %rax
4005f9: popq %rbx
.L6:
4005fa: rep
4005fb: retq
```



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
```

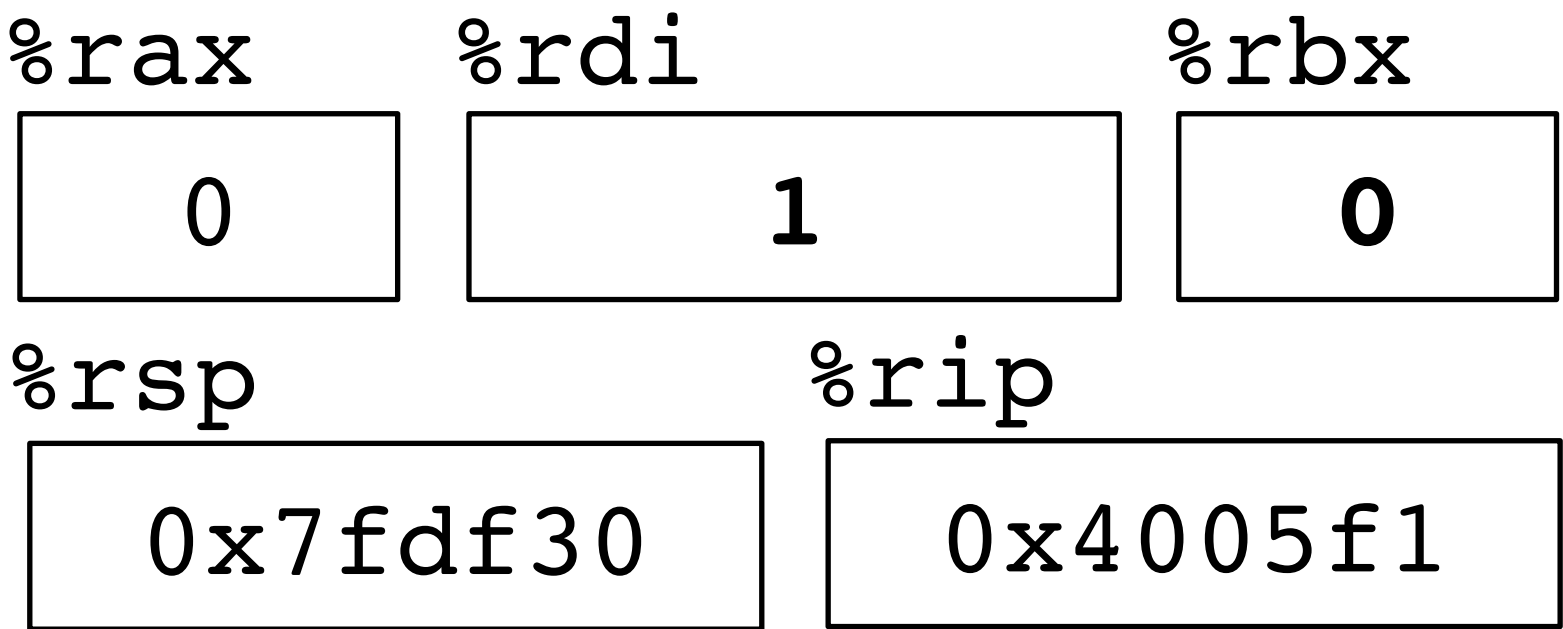
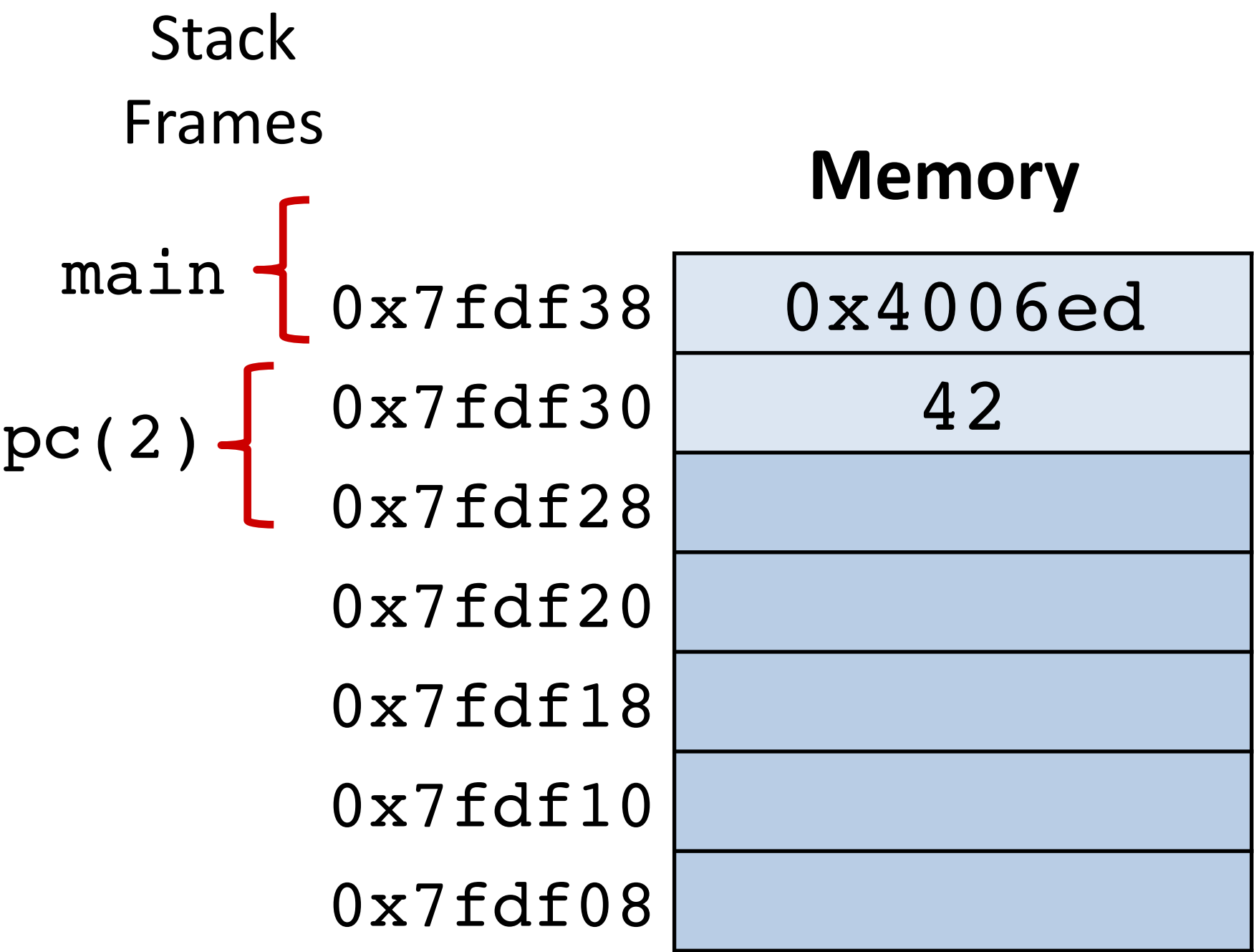


Recursion Example: pcount (2)

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

pcount:

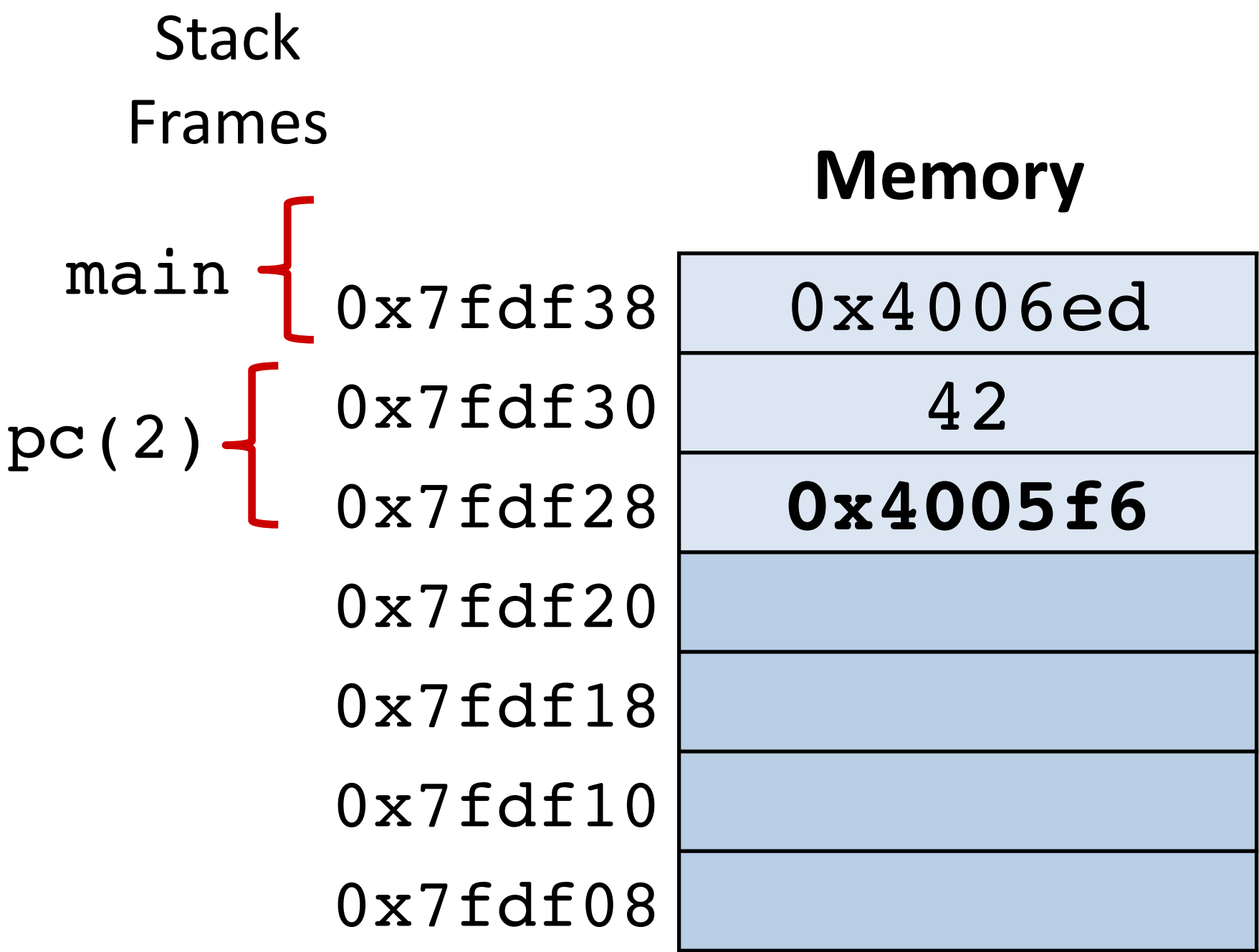
```
4005dd: movl $0, %eax
4005e2: testq %rdi, %rdi
4005e5: je 4005fa <.L6>
4005e7: pushq %rbx
4005e8: movq %rdi, %rbx
4005eb: andl $1, %ebx
4005ee: shrq %rdi
4005f1: callq pcount
4005f6: addq %rbx, %rax
4005f9: popq %rbx
.L6:
4005fa: rep
4005fb: retq
```



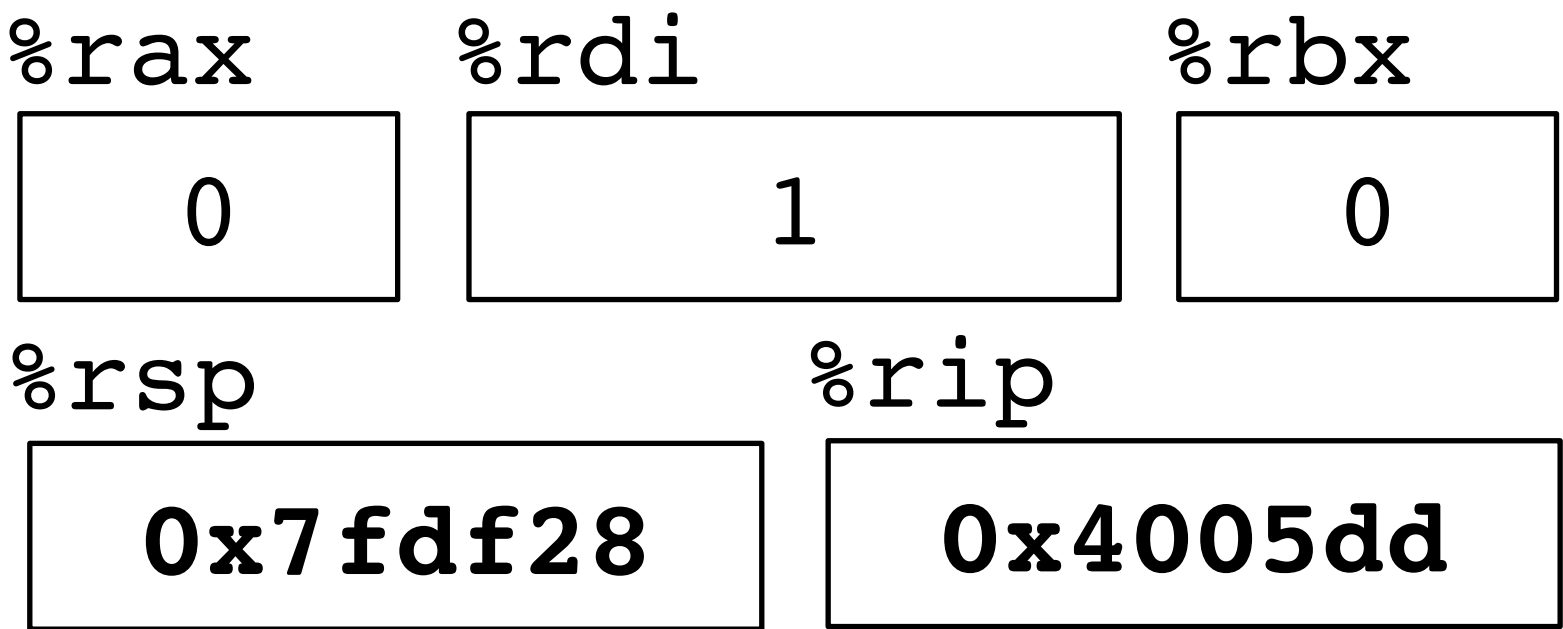
Recursion Example: pcount(2) → pcount(1)

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

<u>pcount:</u>		
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

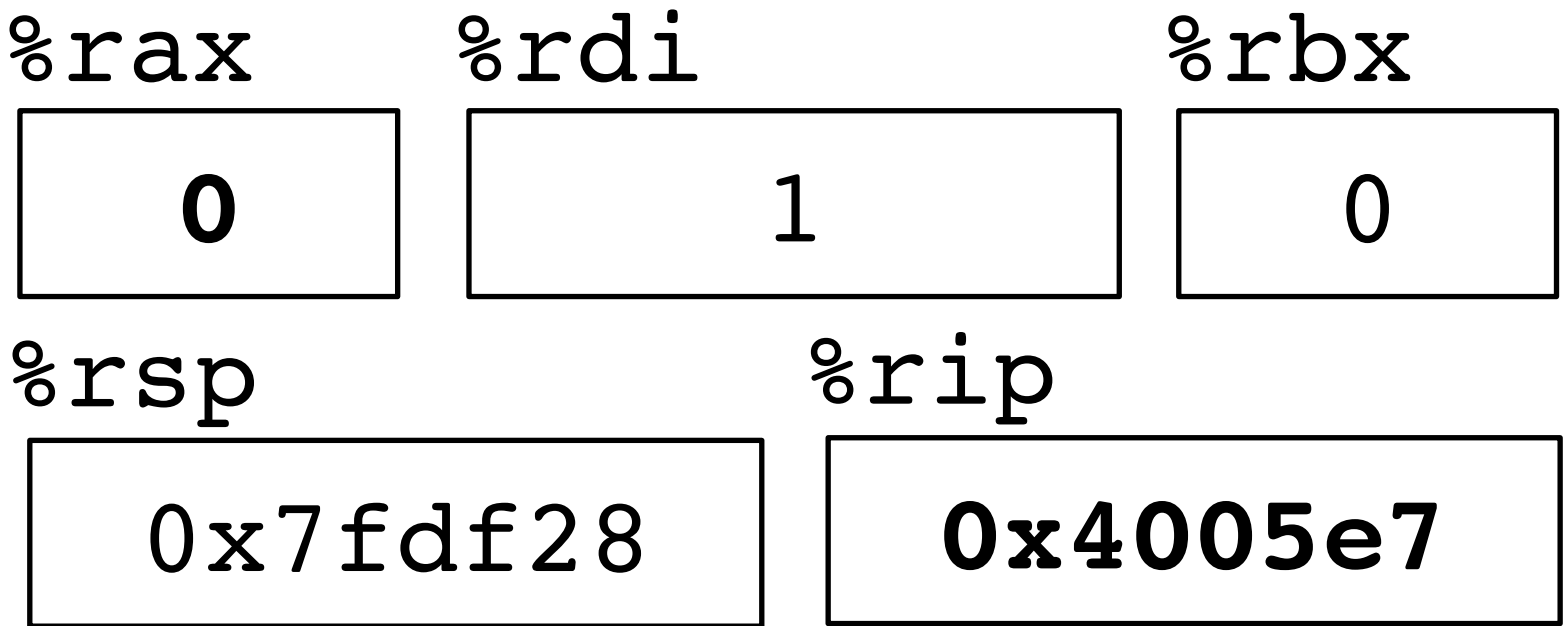
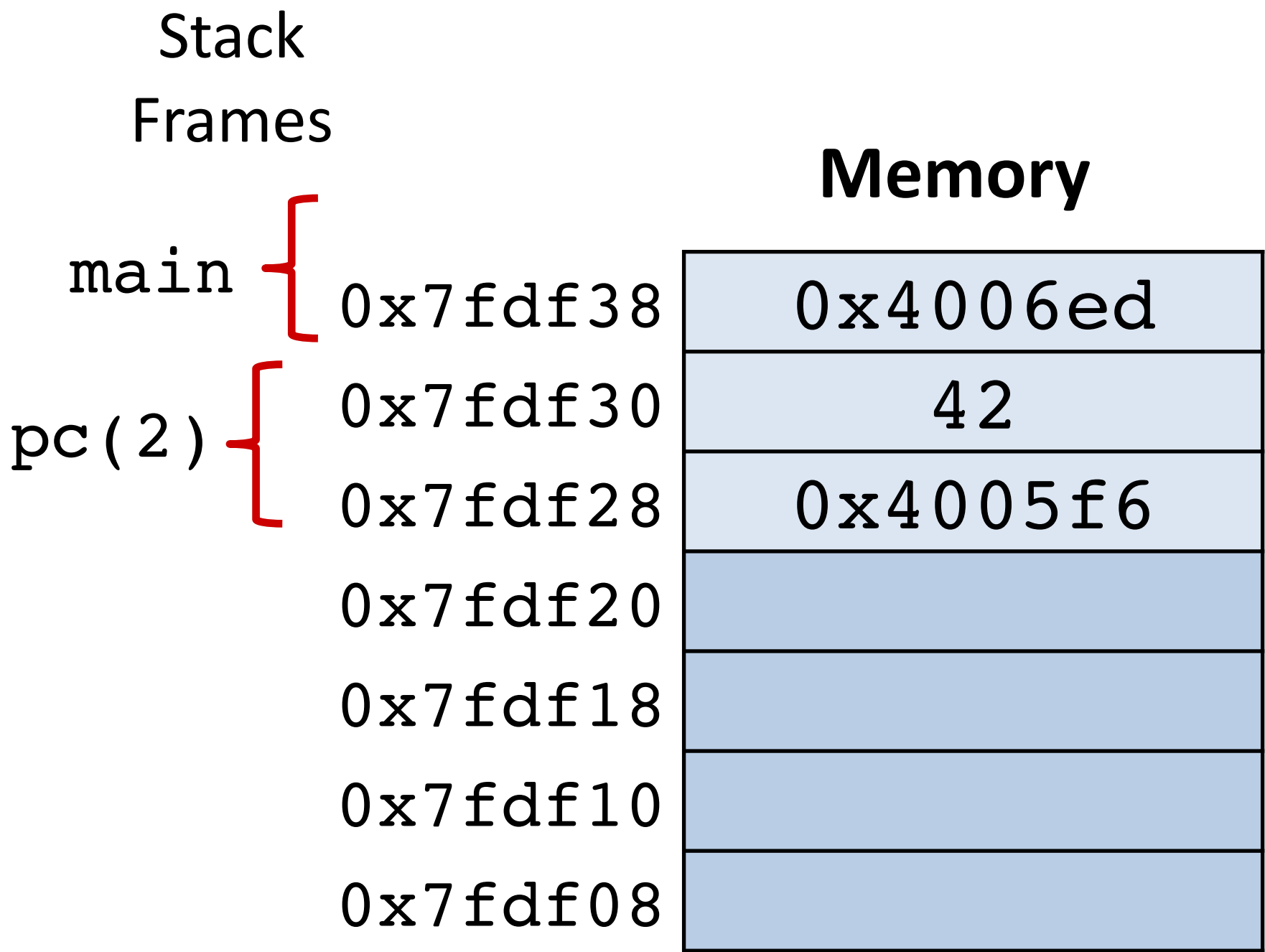


Recursion Example: pcount(2) → pcount(1)

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

pcount:

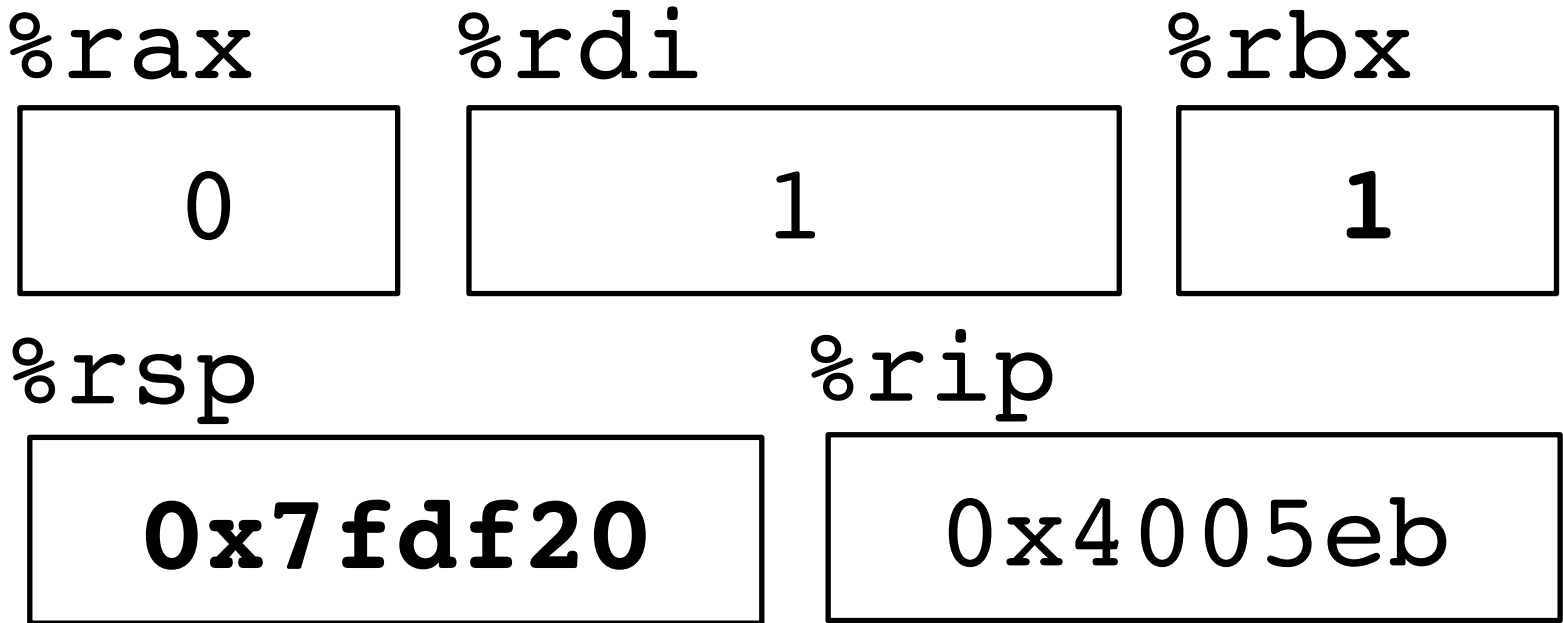
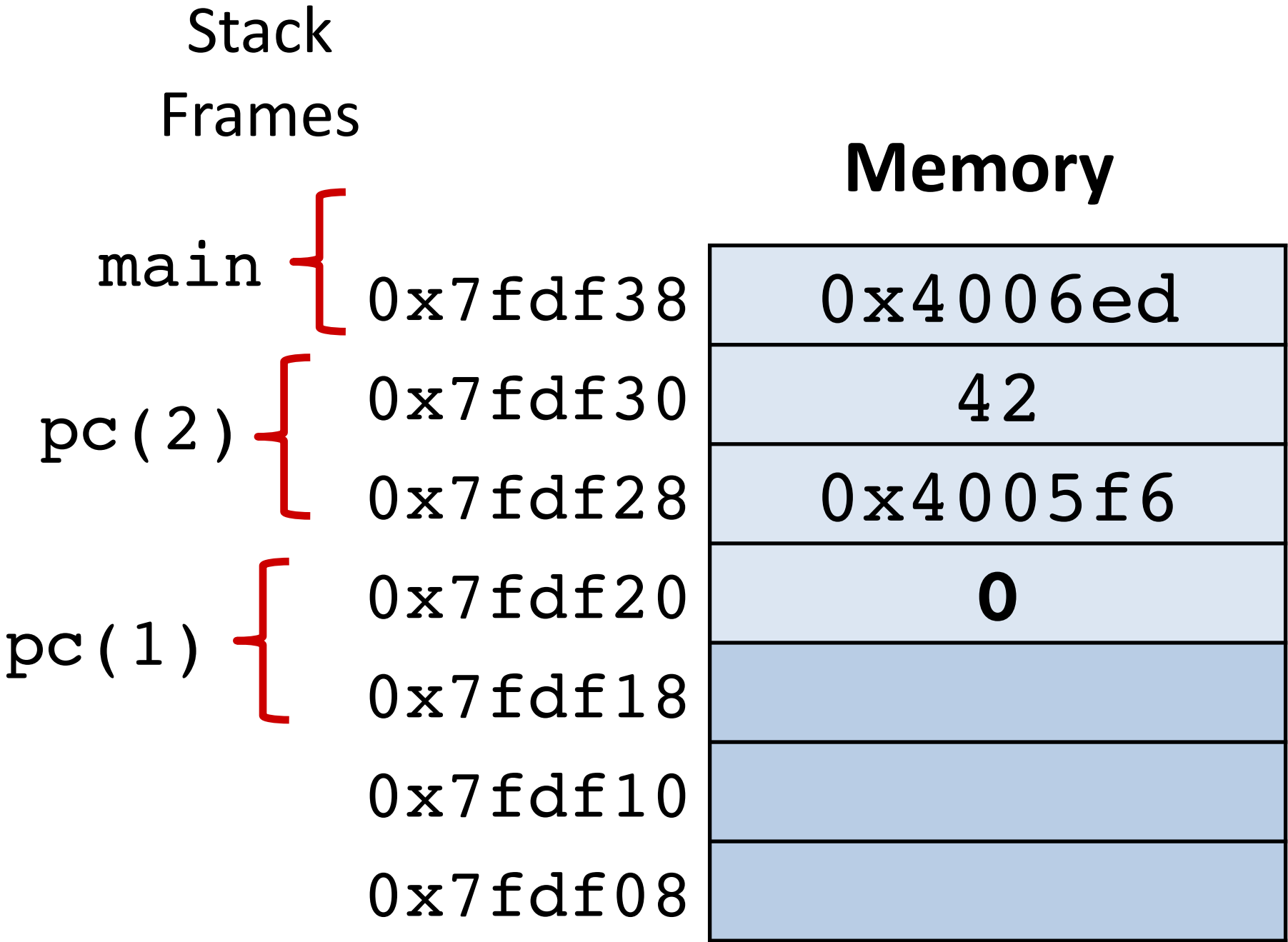
```
4005dd: movl $0, %eax  
4005e2: testq %rdi, %rdi  
4005e5: je 4005fa <.L6>  
4005e7: pushq %rbx  
4005e8: movq %rdi, %rbx  
4005eb: andl $1, %ebx  
4005ee: shrq %rdi  
4005f1: callq pcount  
4005f6: addq %rbx, %rax  
4005f9: popq %rbx  
.L6:  
4005fa: rep  
4005fb: retq
```



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

```
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
```



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

```
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

main	{	0x7fdf38
pc(2)	{	0x7fdf30
		0x7fdf28
pc(1)	{	0x7fdf20
		0x7fdf18
		0x7fdf10
		0x7fdf08

Memory

0x4006ed
42
0x4005f6
0

%rax	%rdi	%rbx
0	0	1
%rsp	%rip	
0x7fdf20	0x4005f1	

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

```
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

main	{	0x7fdf38
pc(2)	{	0x7fdf30
		0x7fdf28
pc(1)	{	0x7fdf20
		0x7fdf18
		0x7fdf10
		0x7fdf08

Memory

0x4006ed
42
0x4005f6
0
0x4005f6

%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);
    }
}
```

```
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 {	0x7fdf38
pc(2) {	0x7fdf30
	0x7fdf28
pc(1) {	0x7fdf20
	0x7fdf18
	0x7fdf10
	0x7fdf08

Memory

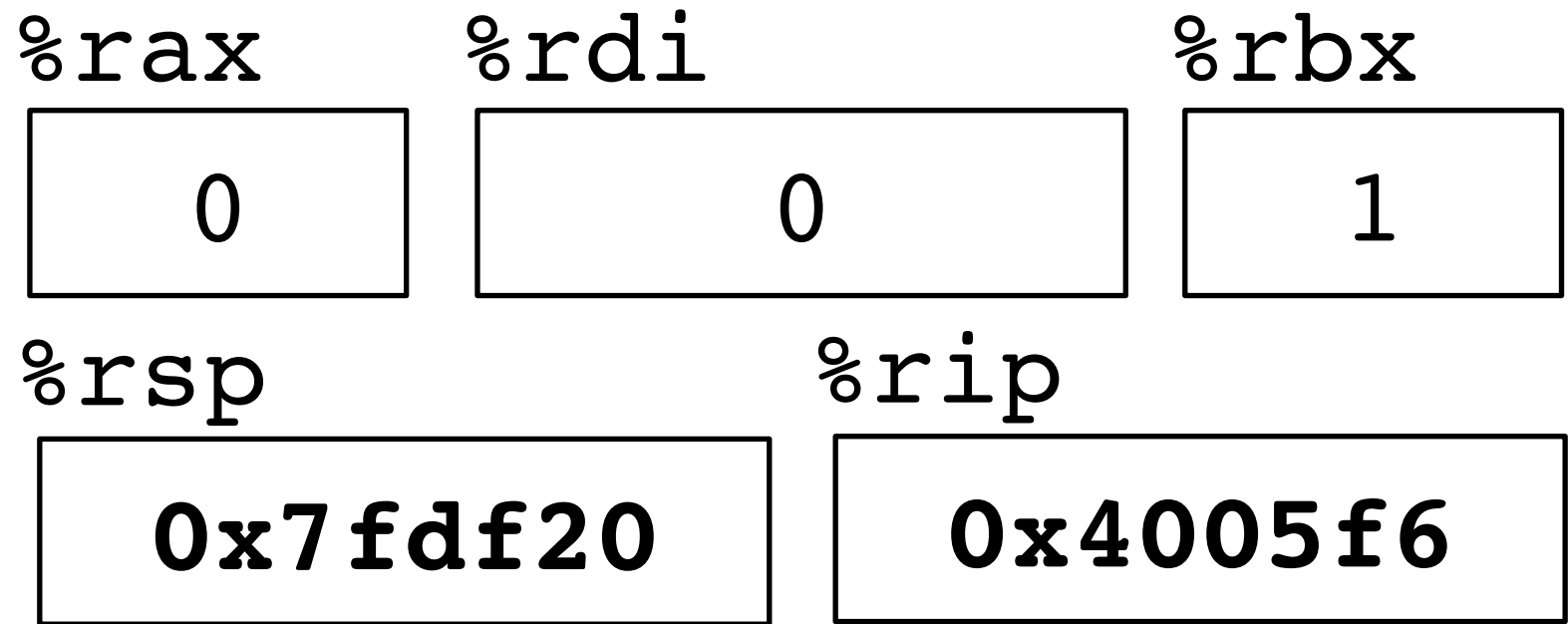
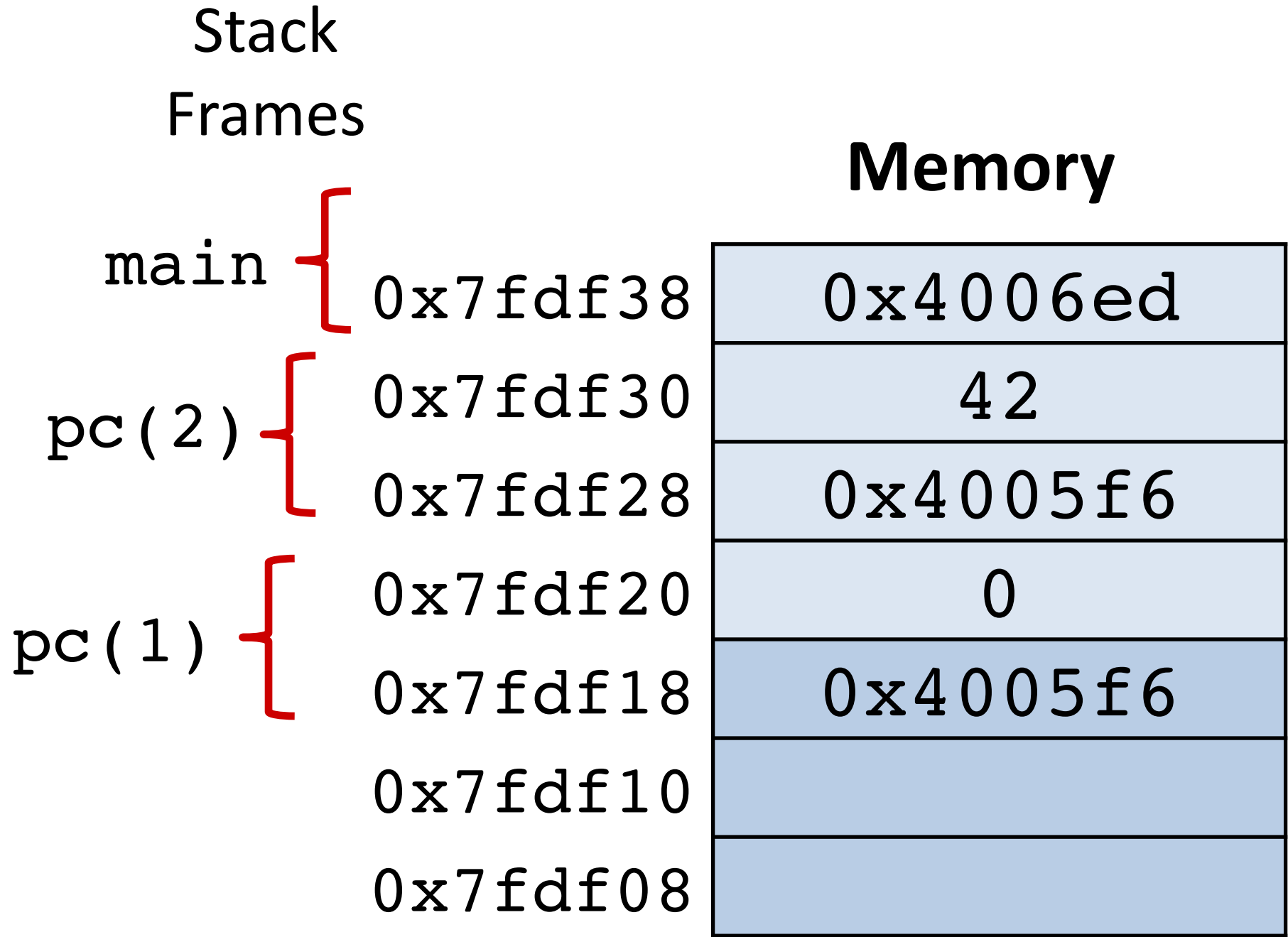
0x4006ed
42
0x4005f6
0
0x4005f6

%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);
    }
}
```

```
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
```



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

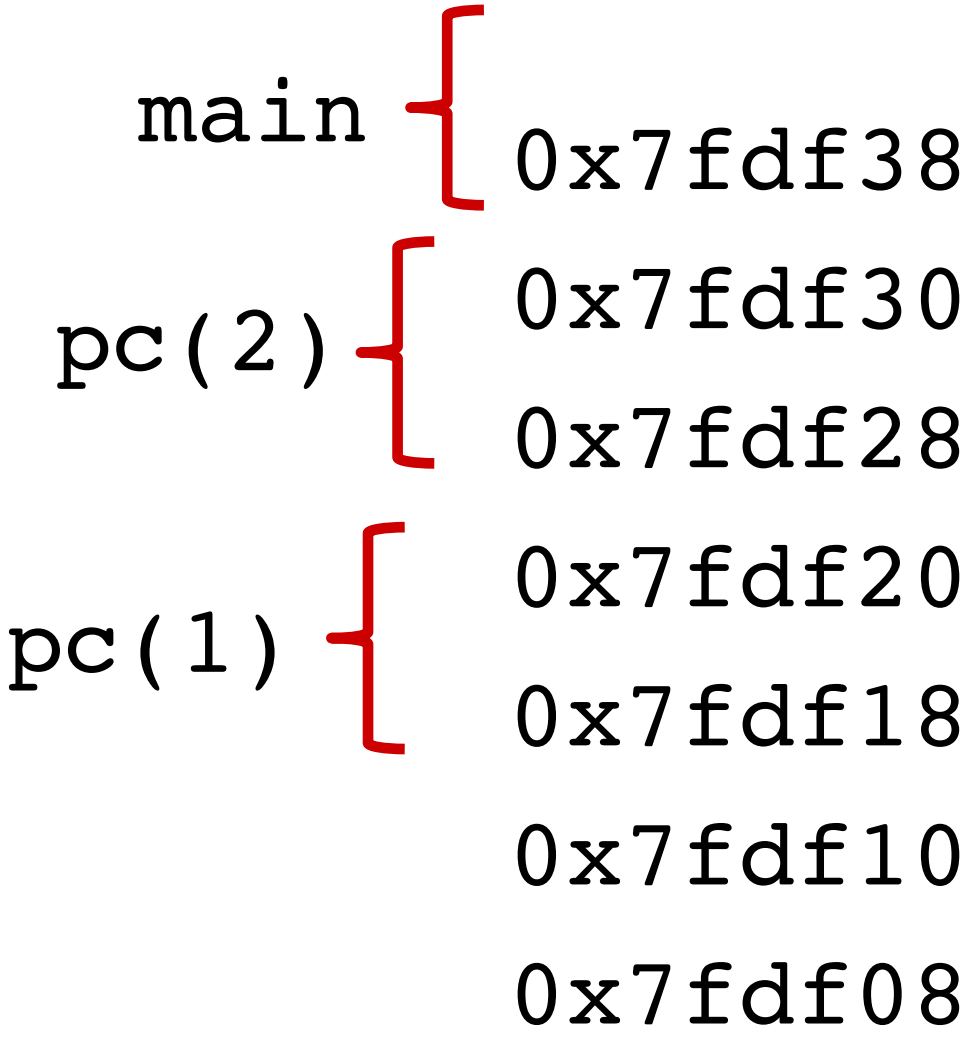
```
1 long pcount(unsigned long x) {  
    if (x == 0) {  
        return 0;  
    } else {  
        return (x & 1) + pcount(x >> 1);  
    }  
}
```

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
```

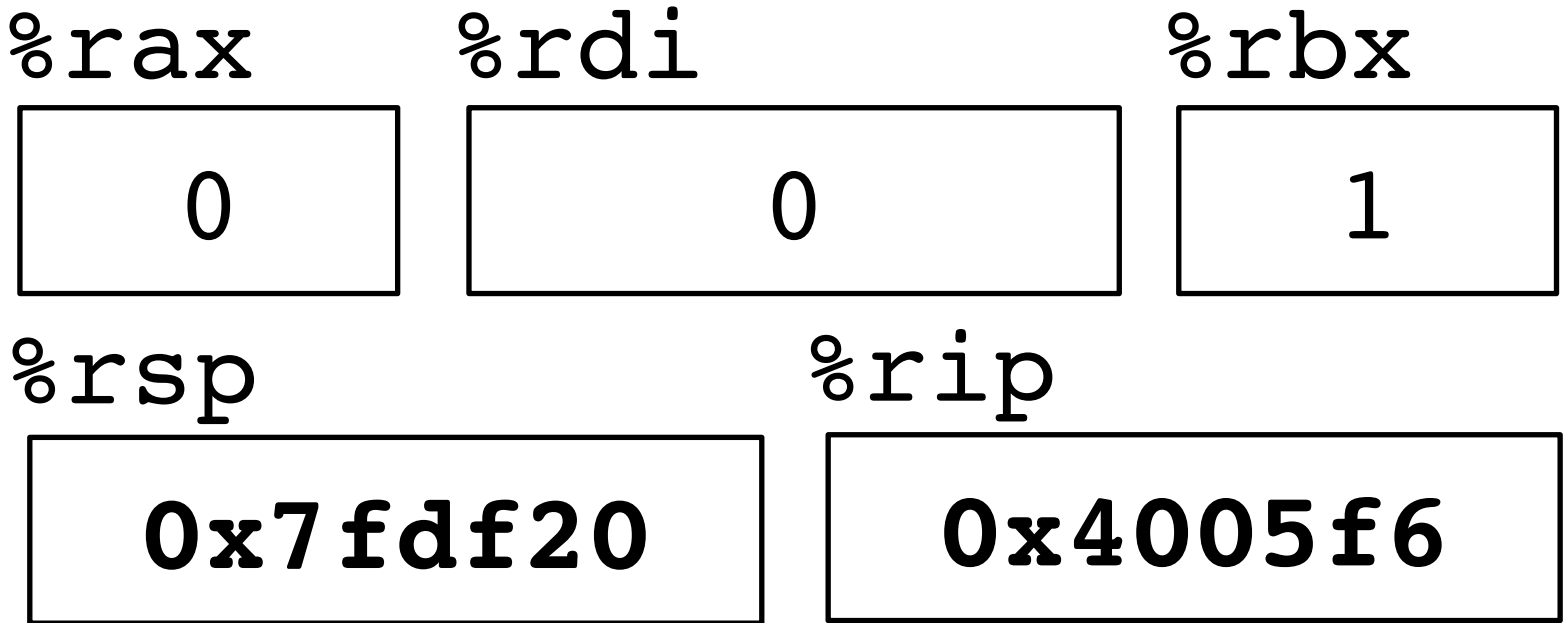
Stack

Frames



Memory

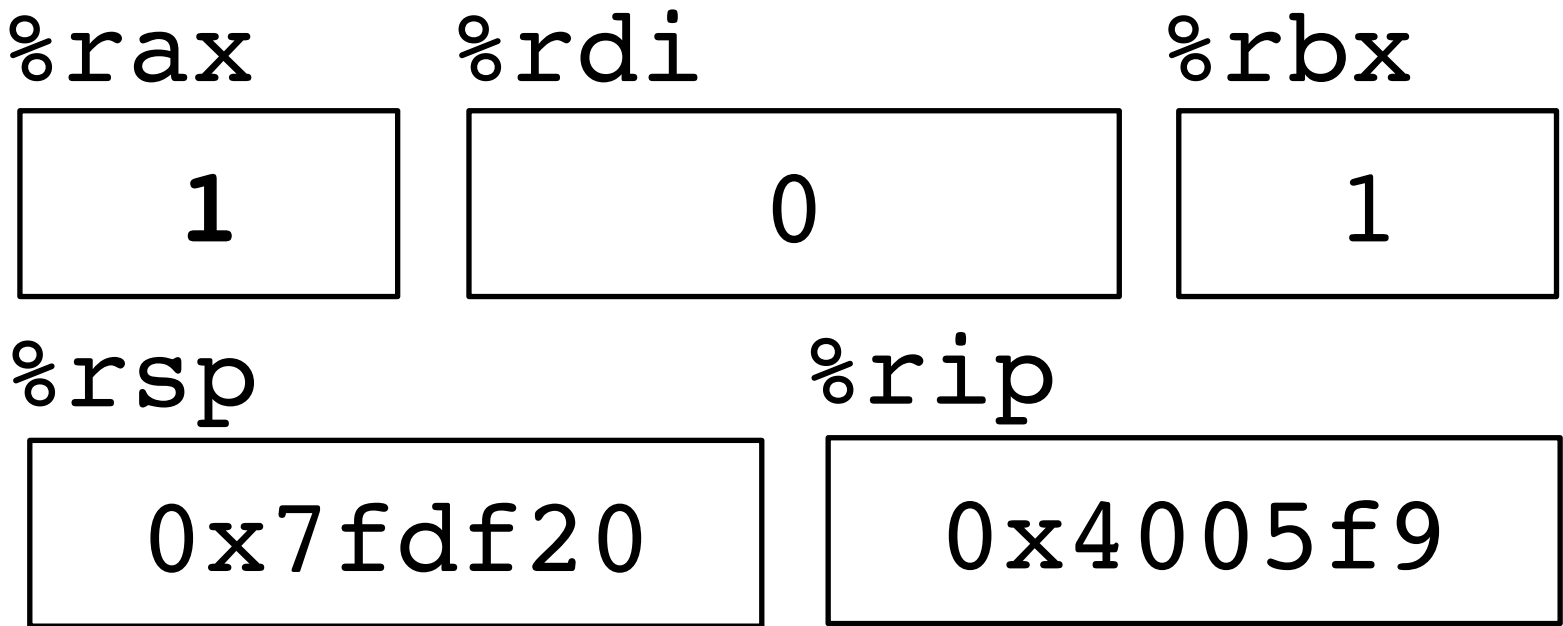
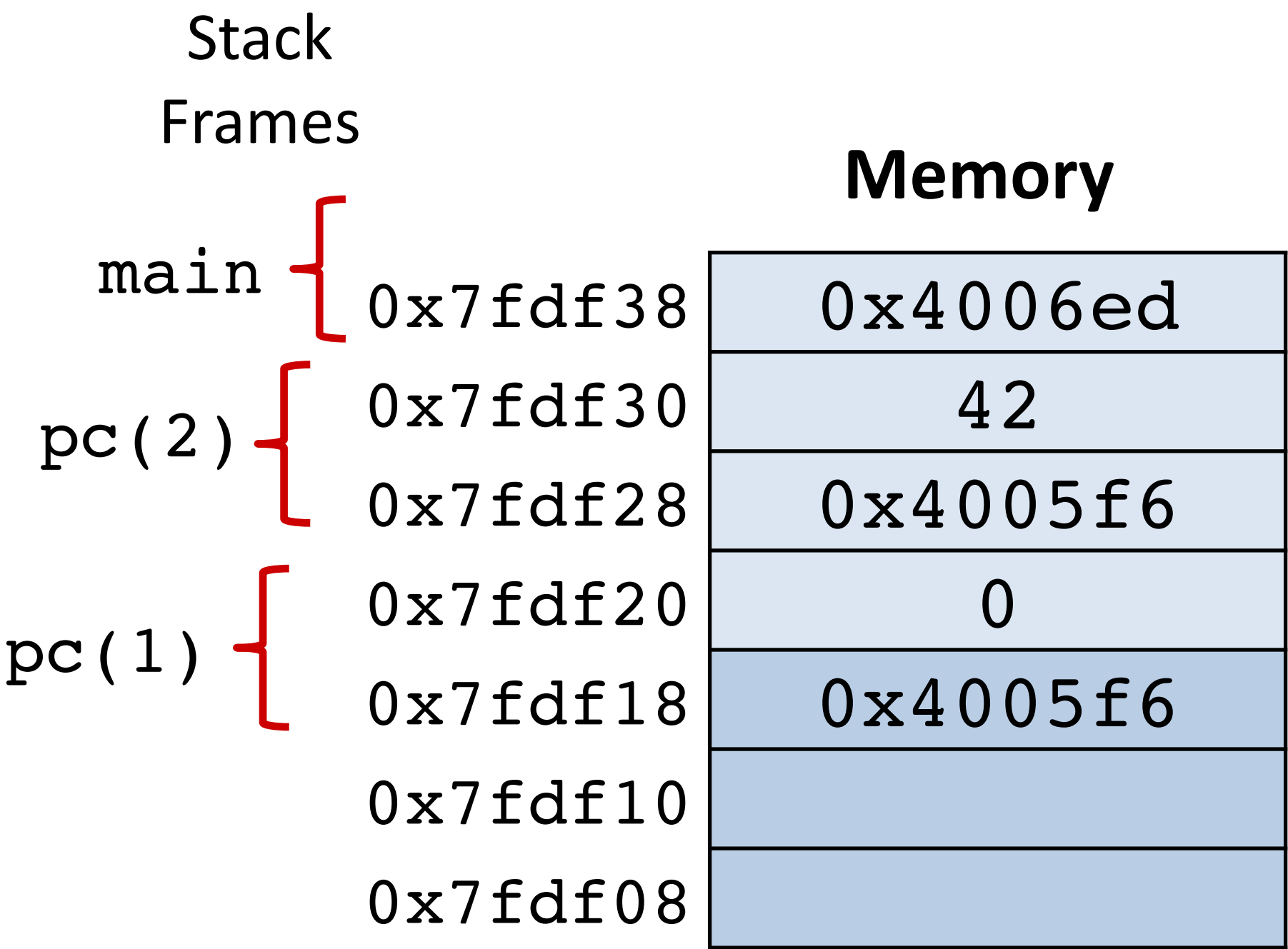
0x4006ed
42
0x4005f6
0
0x4005f6



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

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

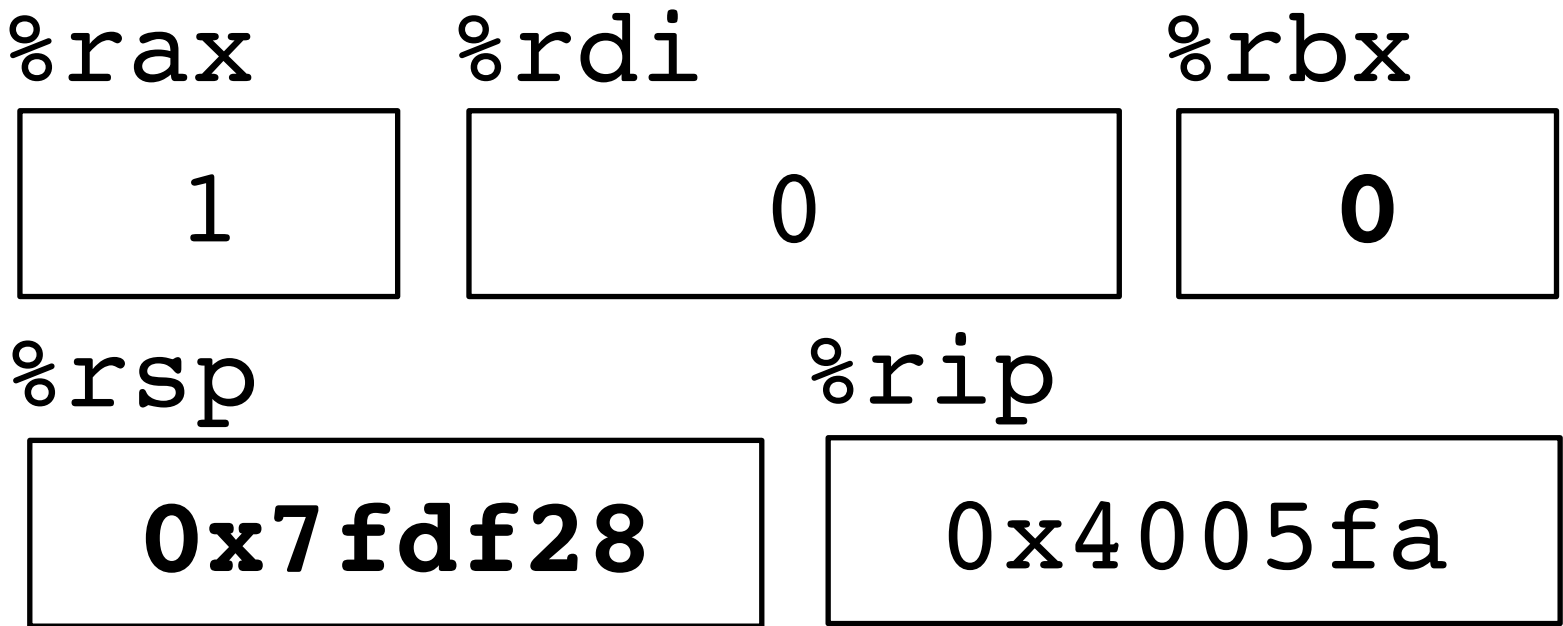
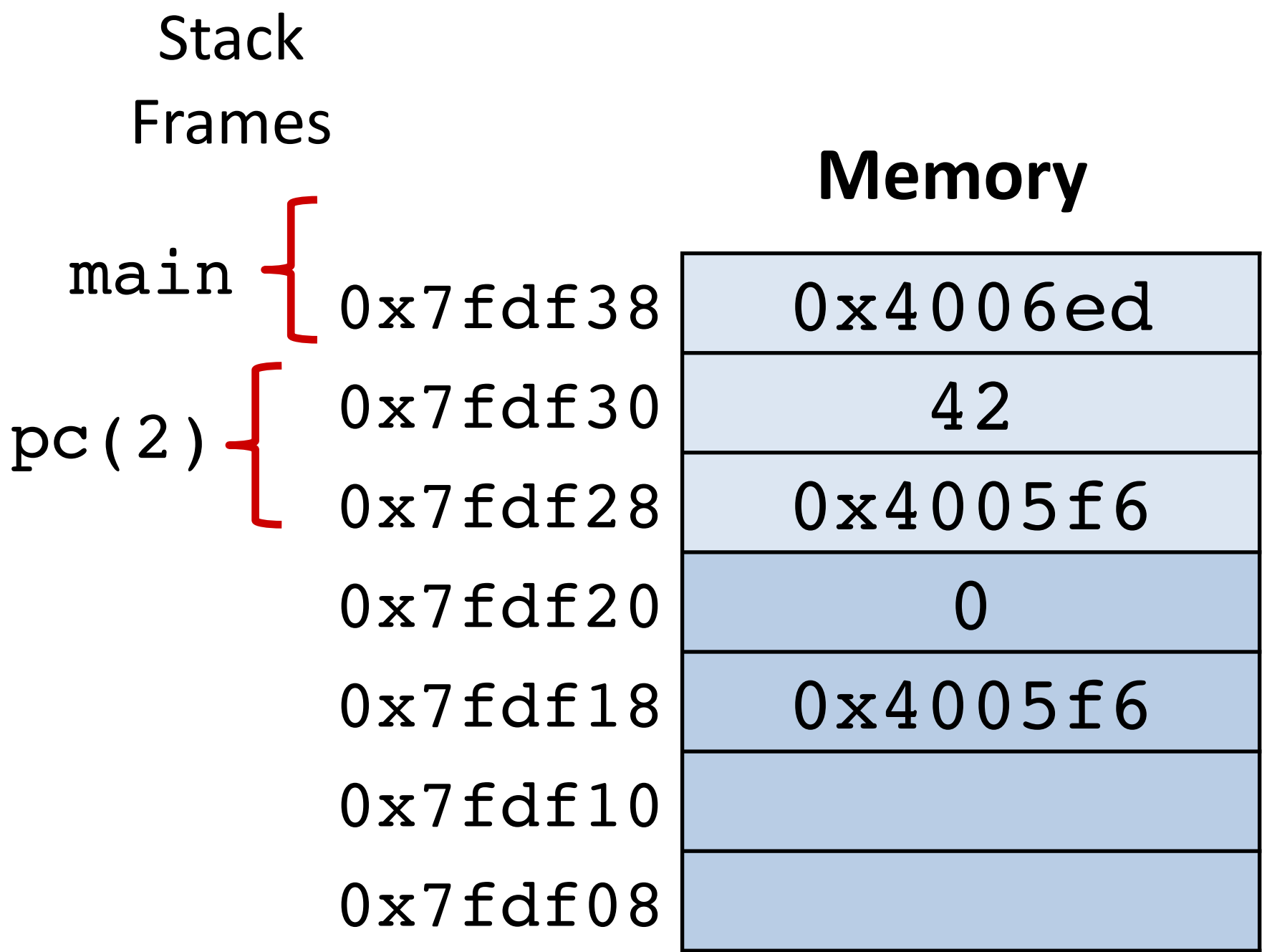
```
pcount:  
4005dd: movl    $0, %eax  
4005e2: testq   %rdi, %rdi  
4005e5: je      4005fa <.L6>  
4005e7: pushq   %rbx  
4005e8: movq    %rdi, %rbx  
4005eb: andl    $1, %ebx  
4005ee: shrq    %rdi  
4005f1: callq   pcount  
4005f6: addq    %rbx, %rax  
4005f9: popq    %rbx  
.L6:  
4005fa: rep  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



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

```
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
```

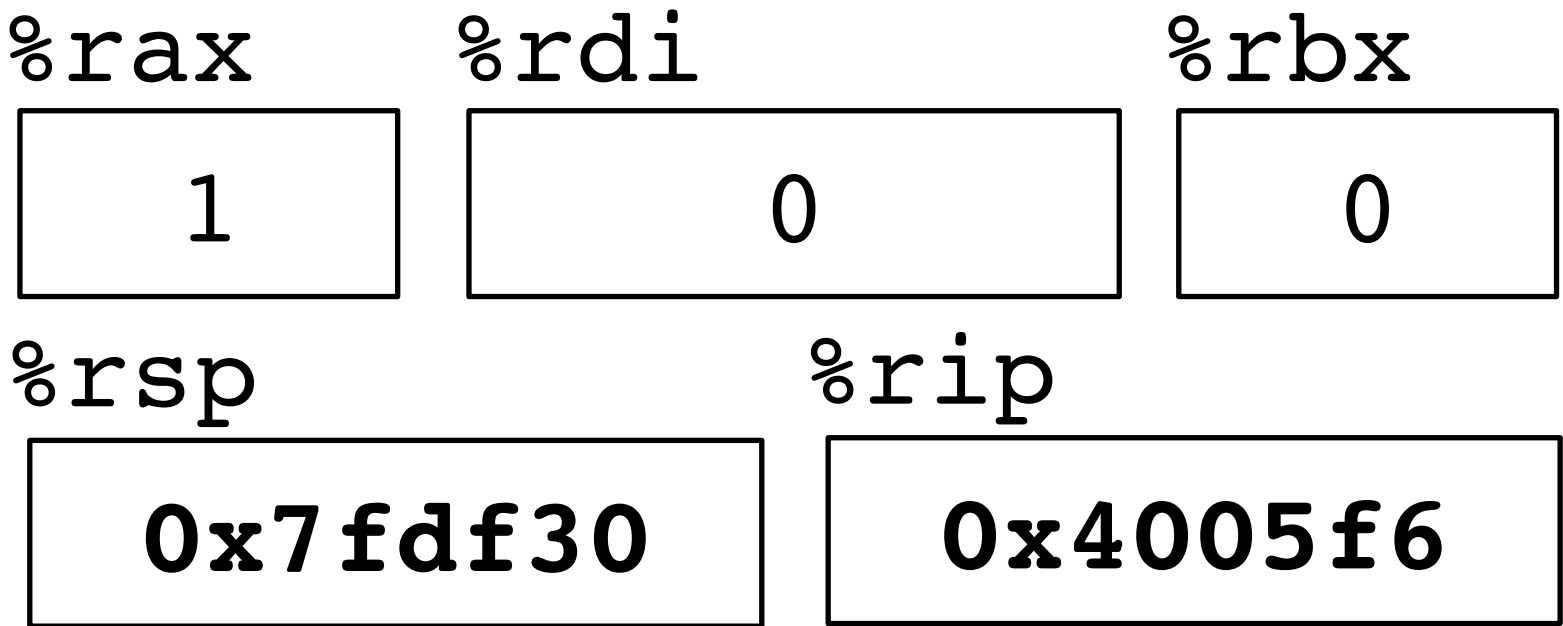
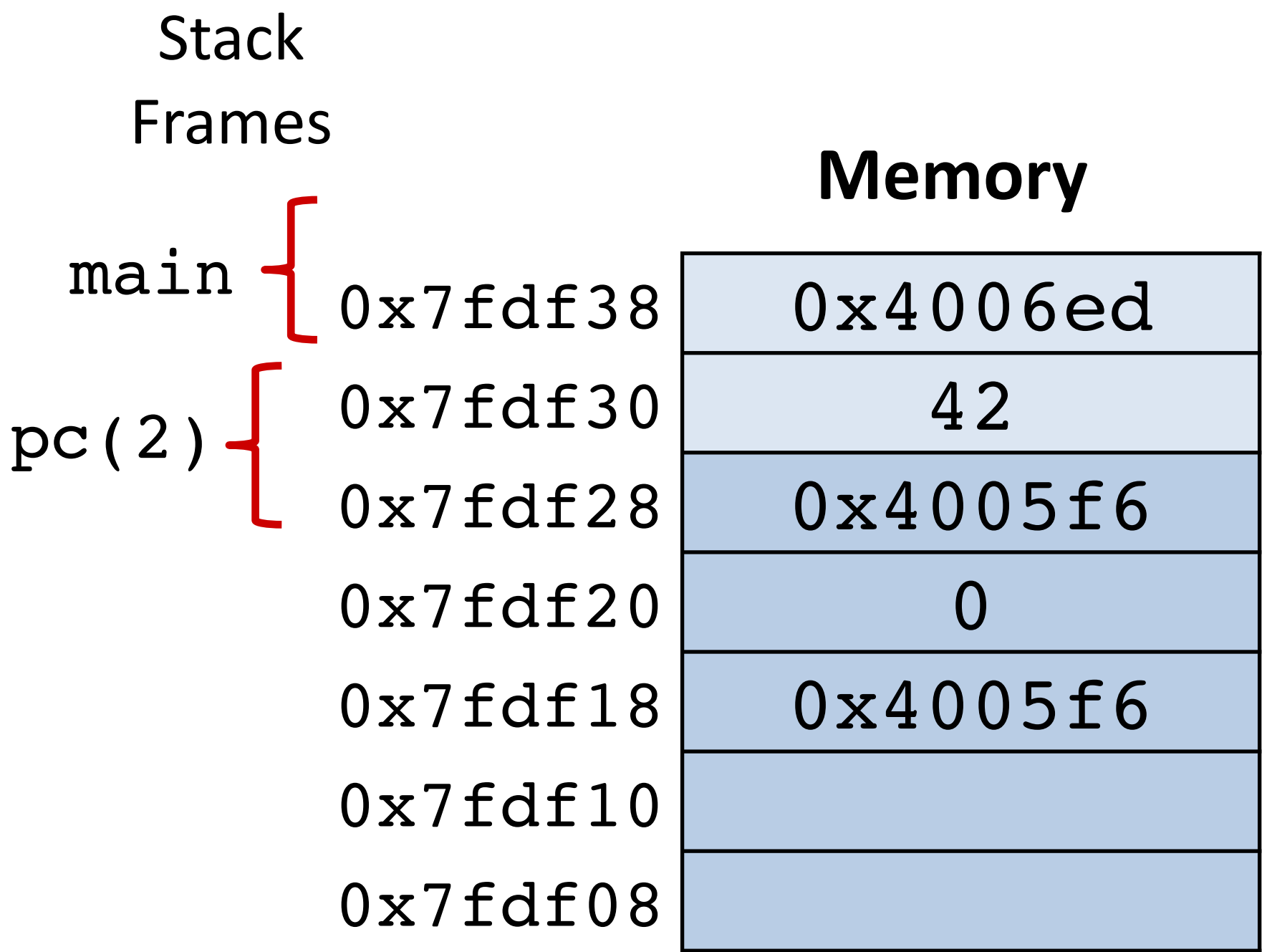


Recursion Example: pcount(2) → pcount(1) → pcount(0)

```
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
```

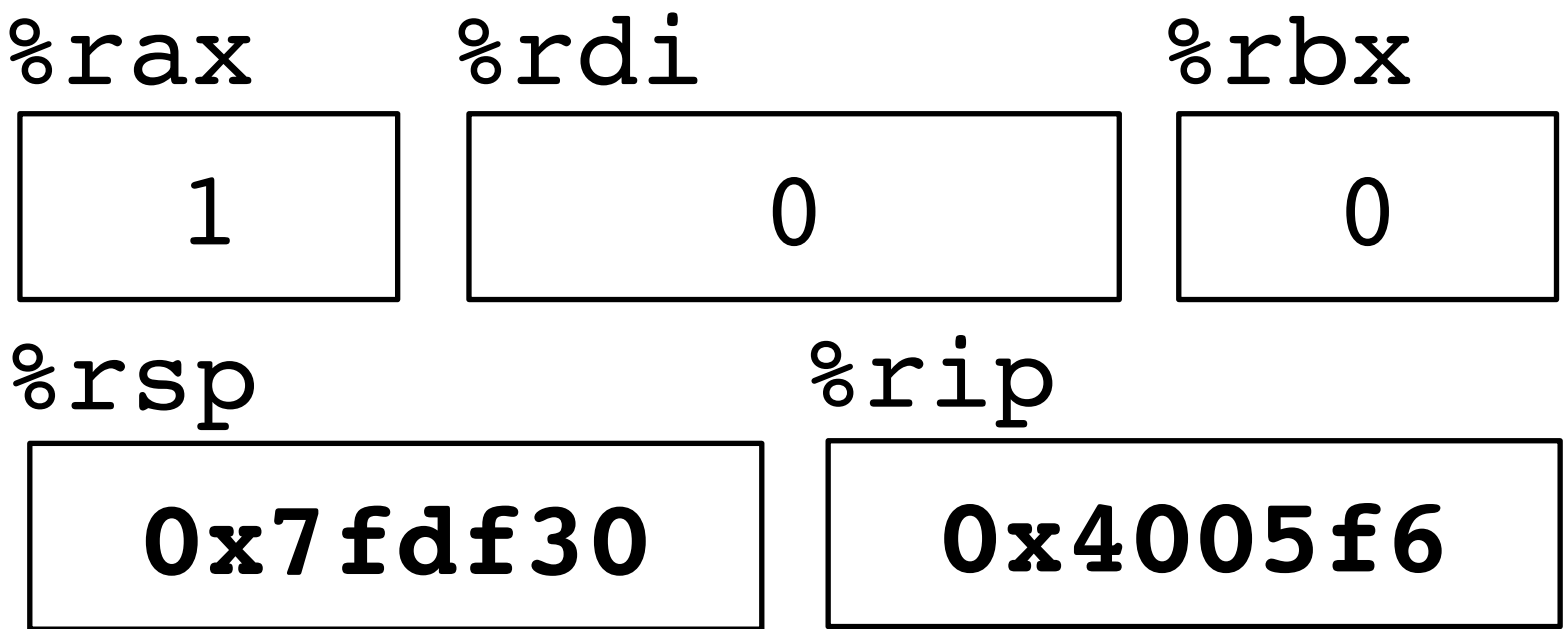
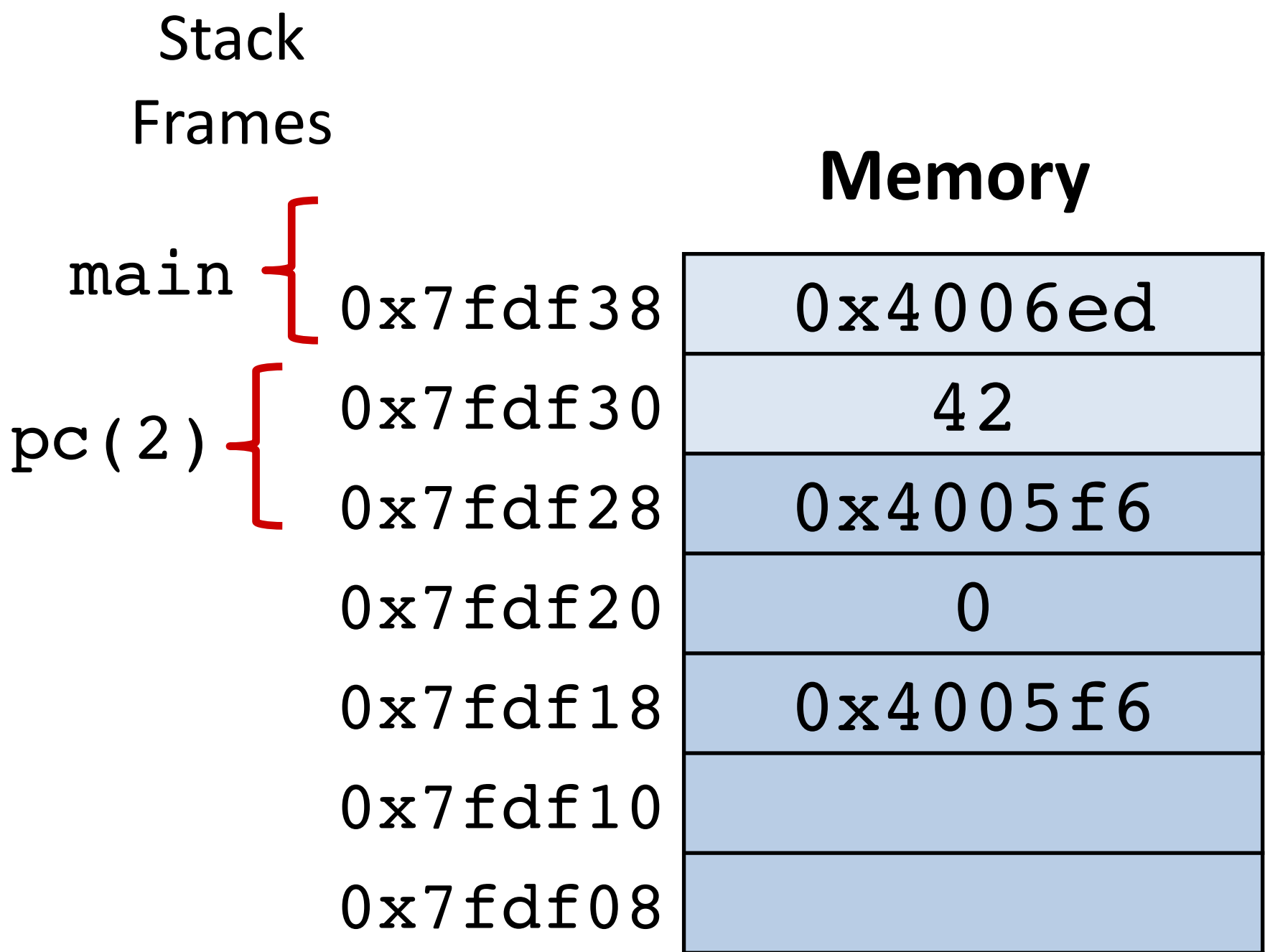


Recursion Example: pcount(2) → pcount(1) → pcount(0)

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

pcount:

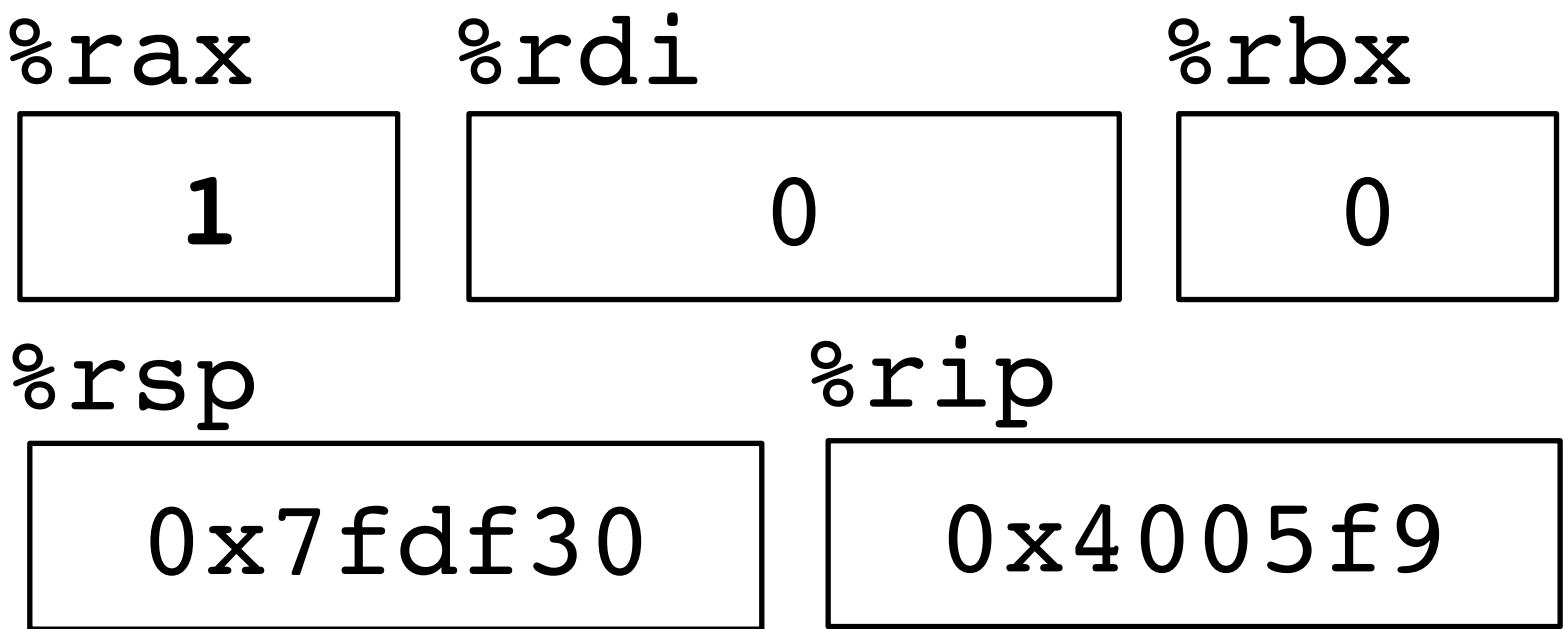
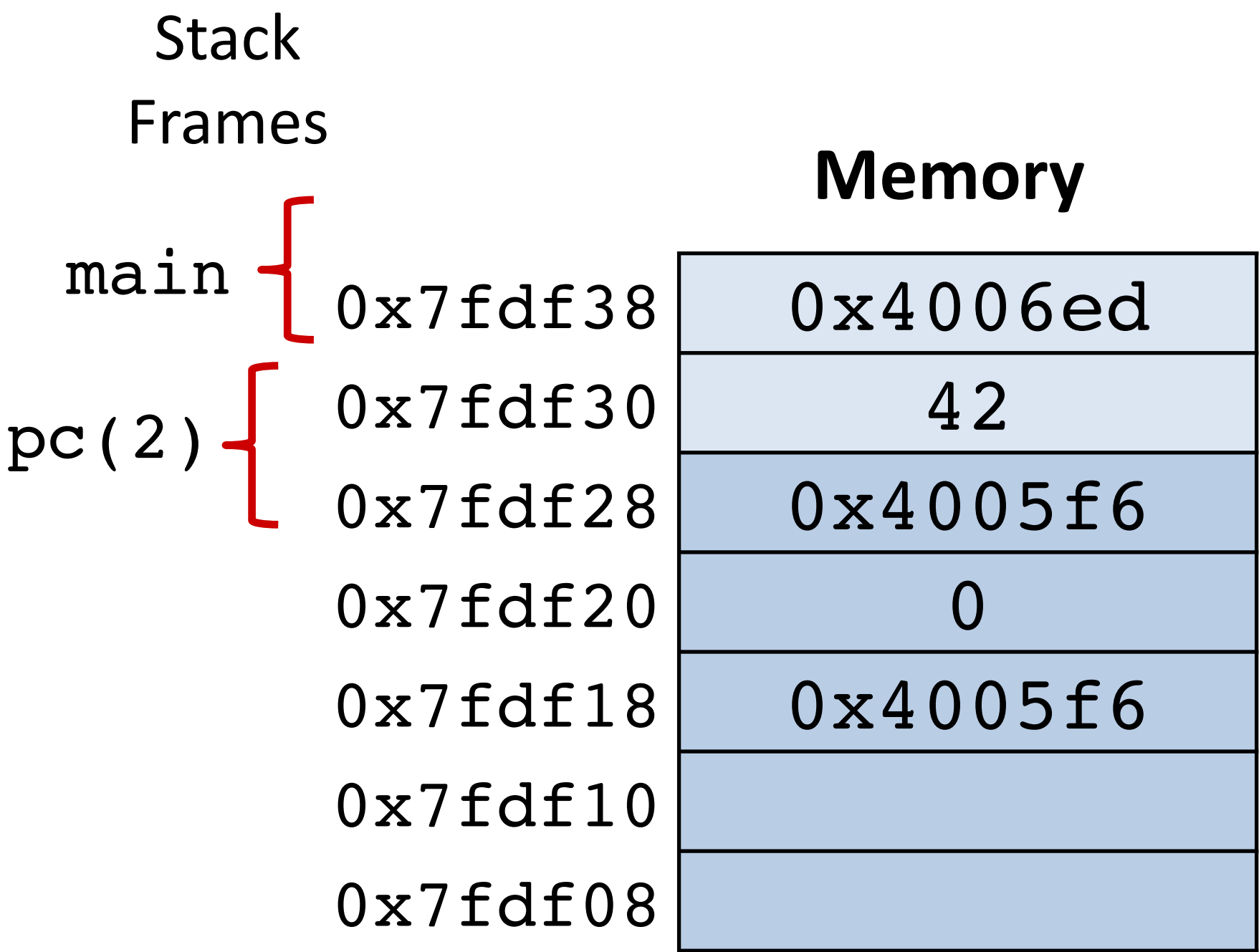
```
4005dd: movl $0, %eax
4005e2: testq %rdi, %rdi
4005e5: je 4005fa <.L6>
4005e7: pushq %rbx
4005e8: movq %rdi, %rbx
4005eb: andl $1, %ebx
4005ee: shrq %rdi
4005f1: callq pcount
4005f6: addq %rbx, %rax
4005f9: popq %rbx
.L6:
4005fa: rep
4005fb: retq
```



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);
    }
}
```

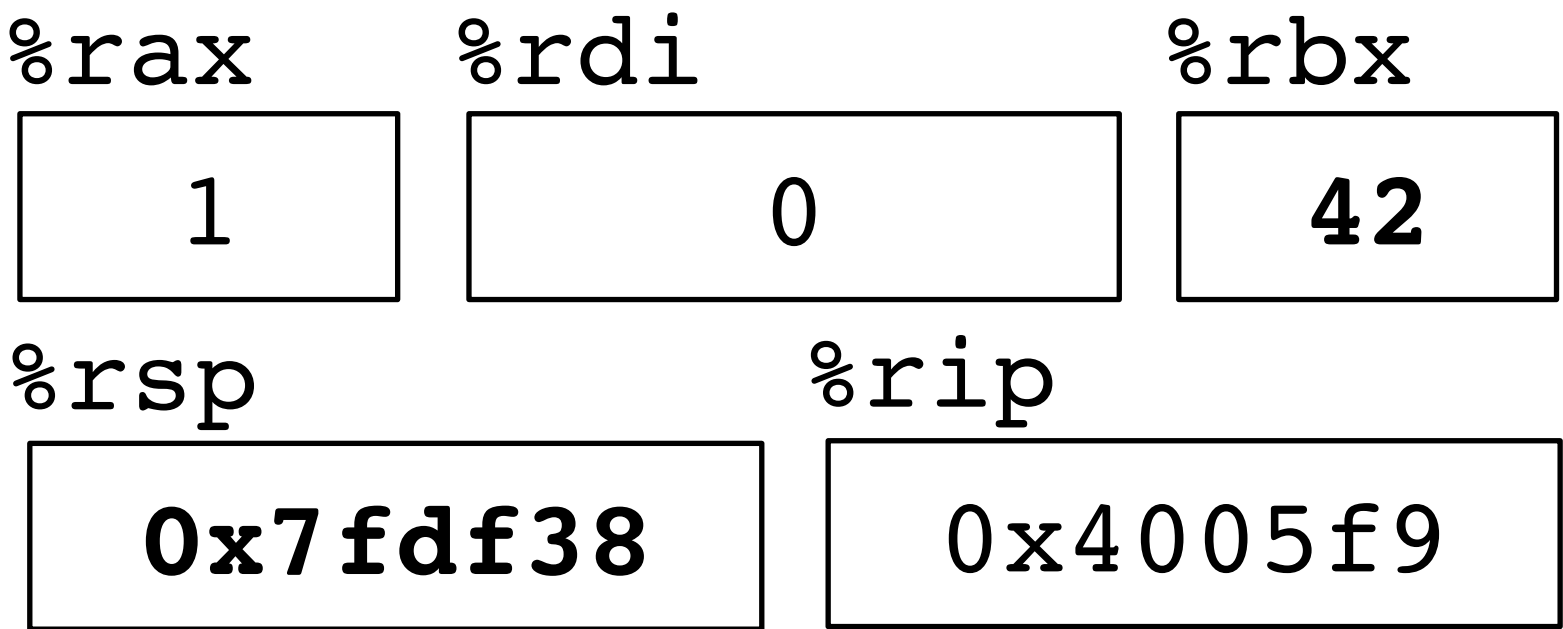
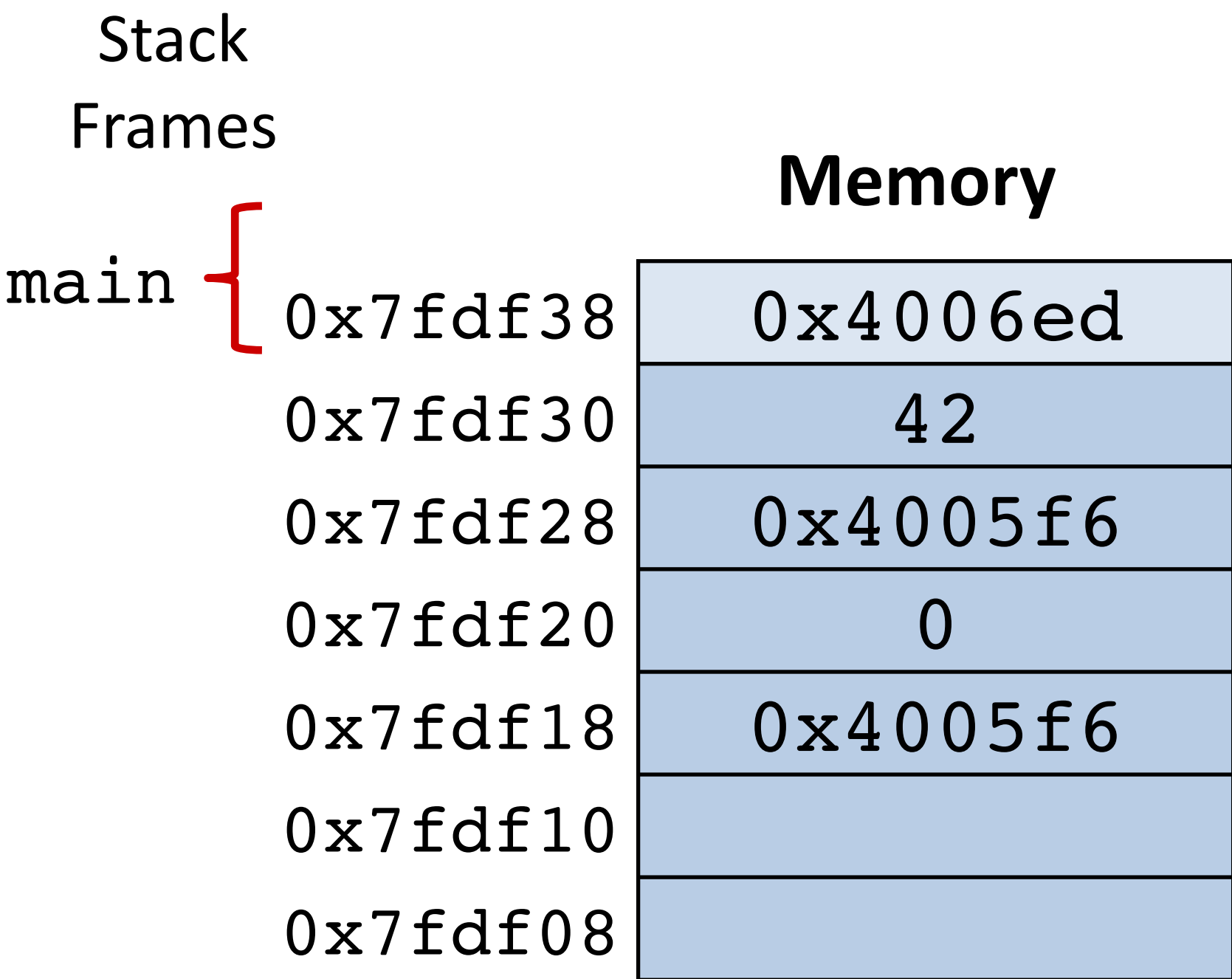
```
pcount:
4005dd: movl    $0, %eax
4005e2: testq   %rdi, %rdi
4005e5: je      4005fa <.L6>
4005e7: pushq   %rbx
4005e8: movq    %rdi, %rbx
4005eb: andl    $1, %ebx
4005ee: shrq    %rdi
4005f1: callq   pcount
4005f6: addq    %rbx, %rax
4005f9: popq    %rbx
.L6:
4005fa: rep
4005fb: retq
```



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);
    }
}
```

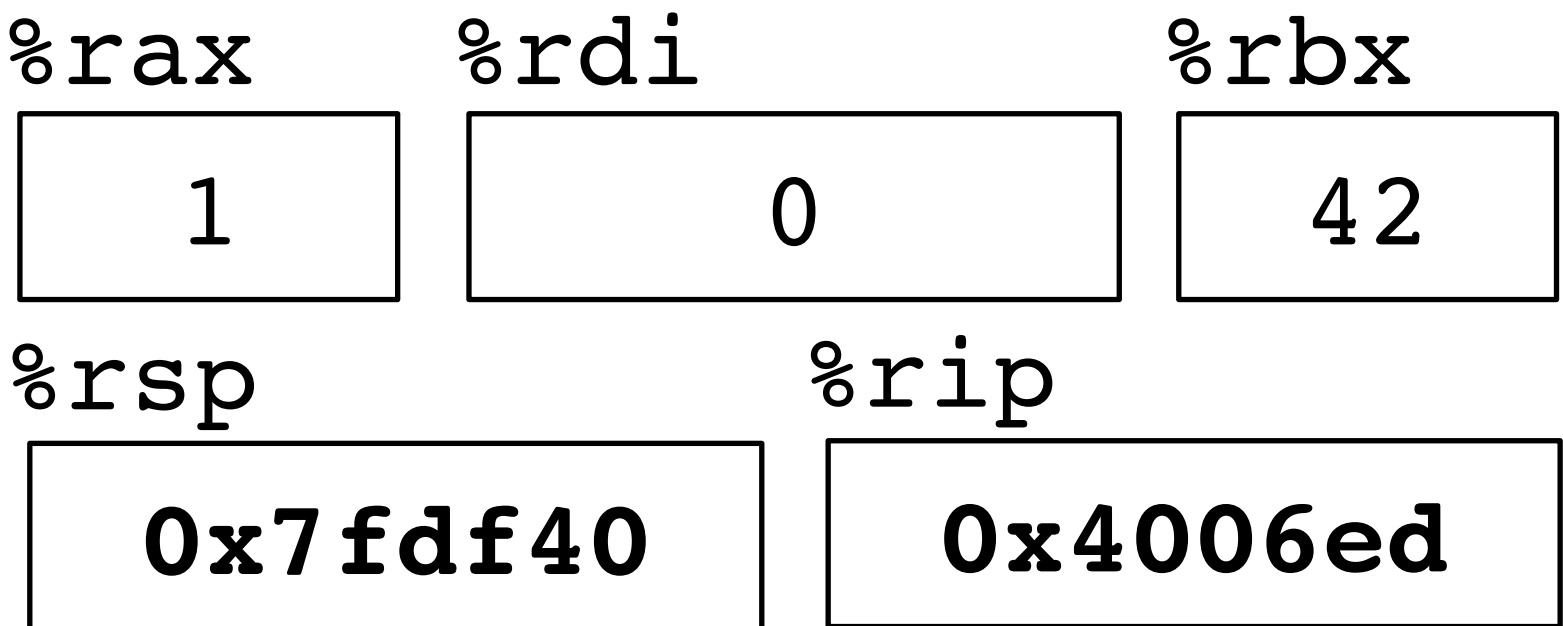
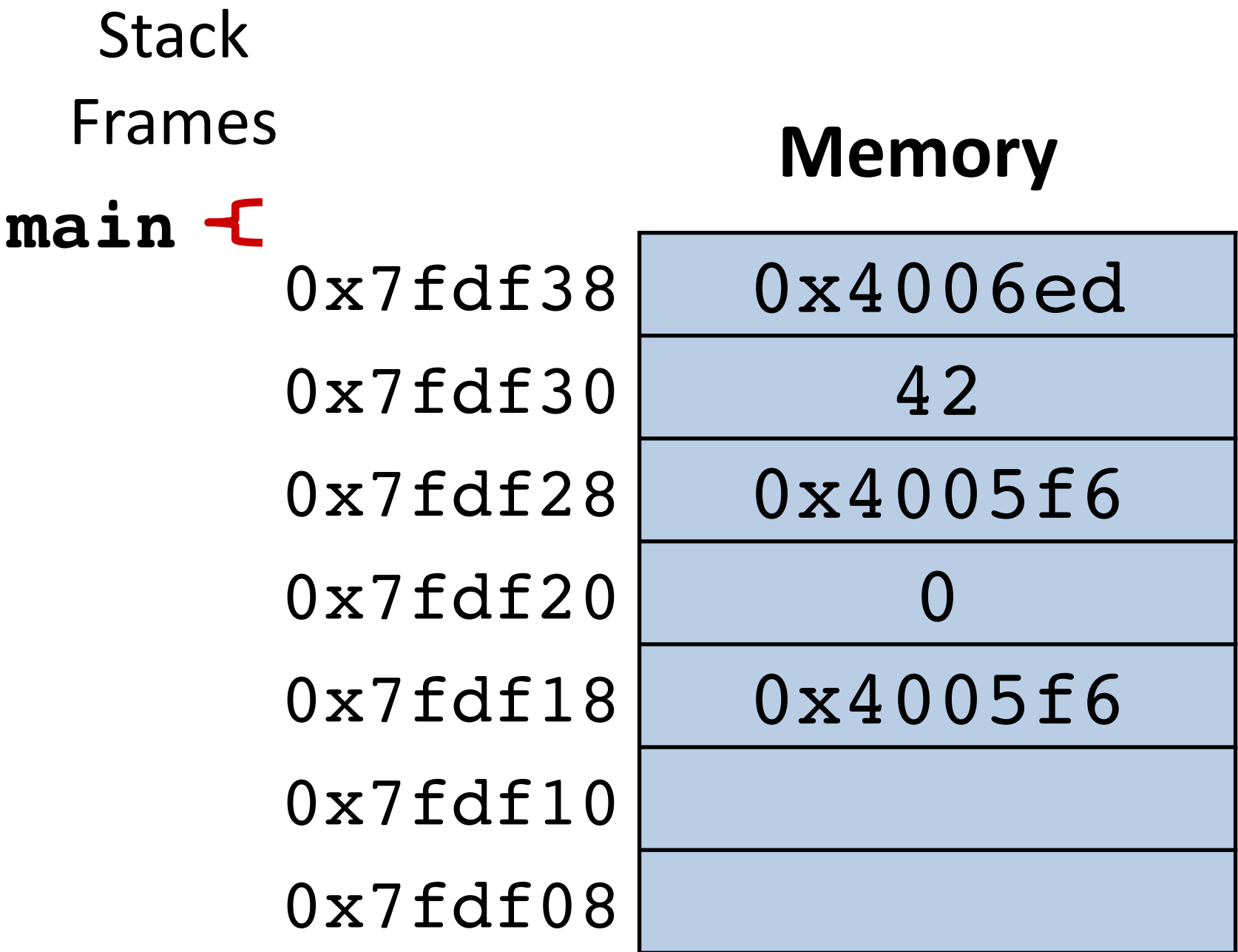
```
pcount:
4005dd: movl    $0, %eax
4005e2: testq   %rdi, %rdi
4005e5: je      4005fa <.L6>
4005e7: pushq   %rbx
4005e8: movq    %rdi, %rbx
4005eb: andl    $1, %ebx
4005ee: shrq    %rdi
4005f1: callq   pcount
4005f6: addq    %rbx, %rax
4005f9: popq    %rbx
.L6:
4005fa: rep
4005fb: retq
```



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
```



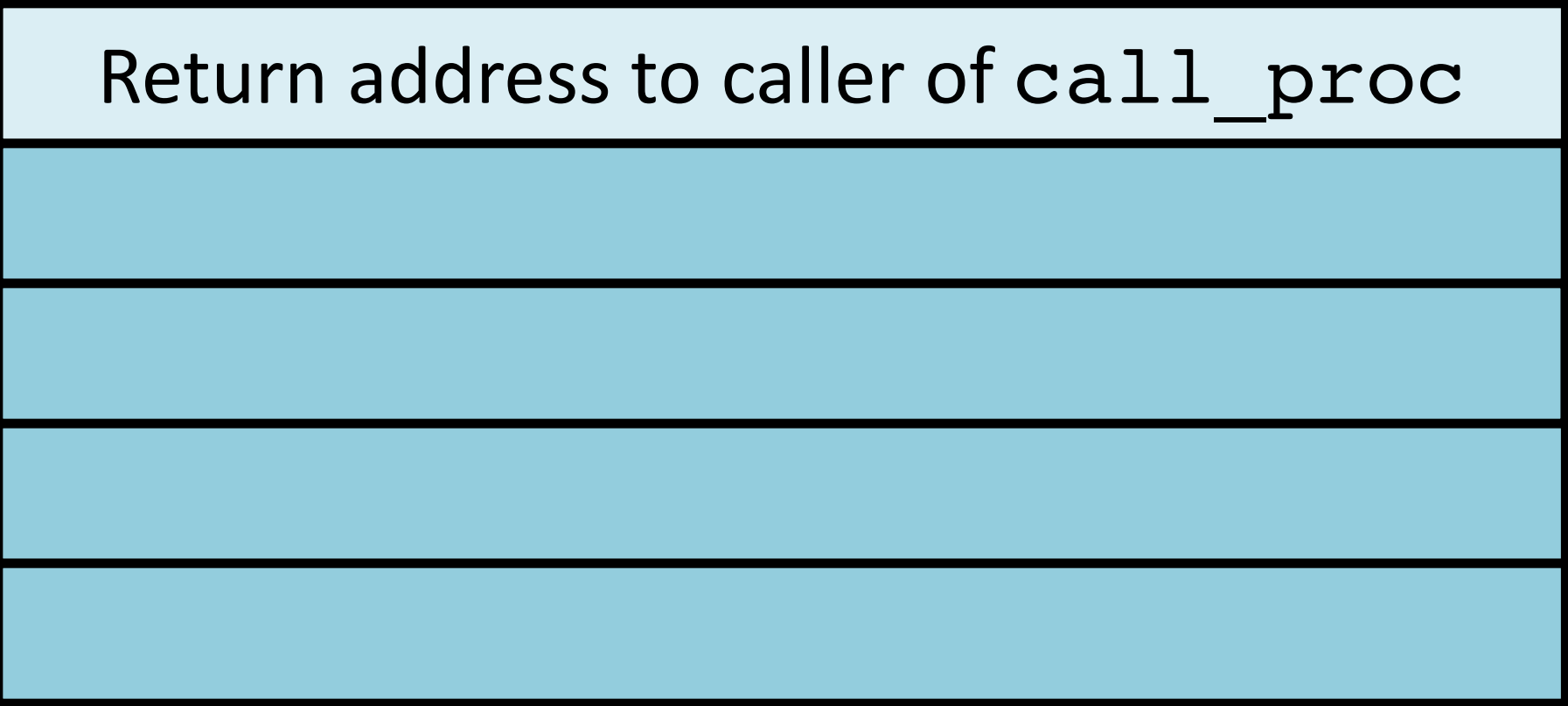
Stack storage example

(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  
    . . .
```



←%rsp

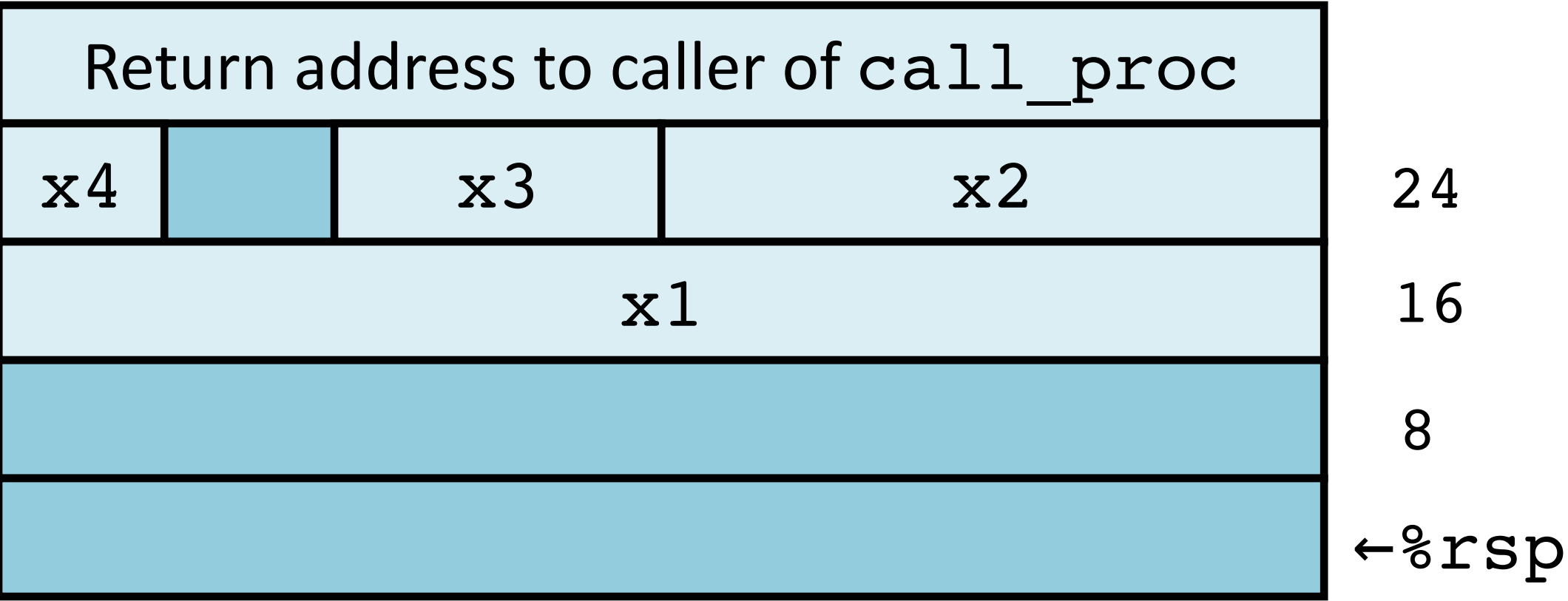
Stack storage example

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

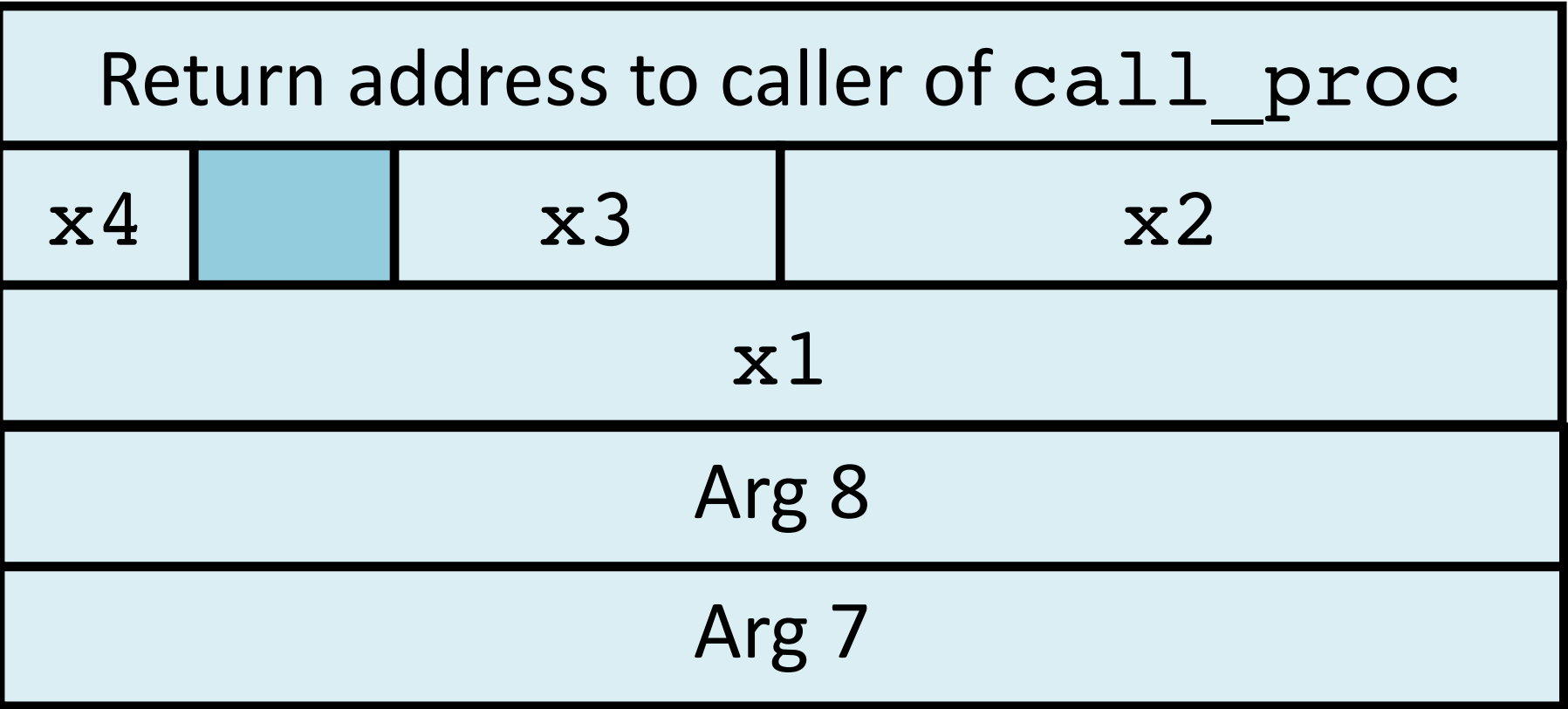


Stack storage example

(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  
    . . .
```

24
16
8
←%rsp

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

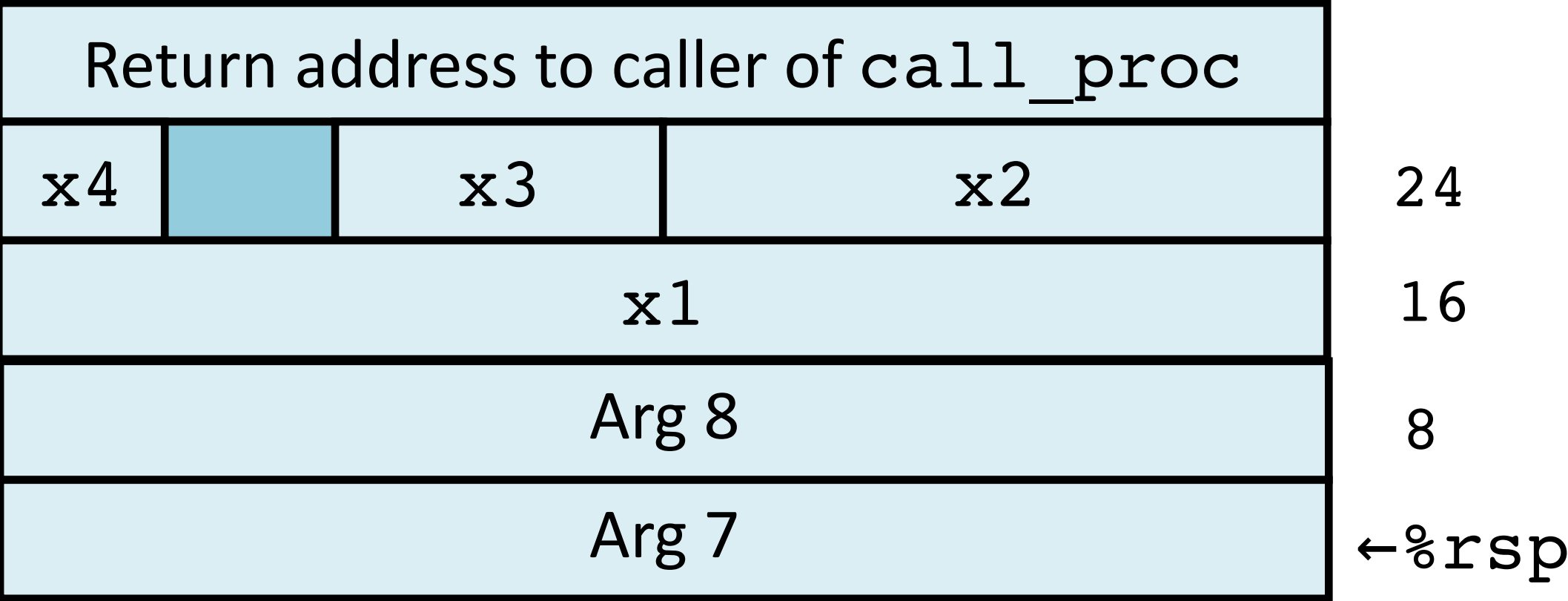
Stack storage example

(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
```



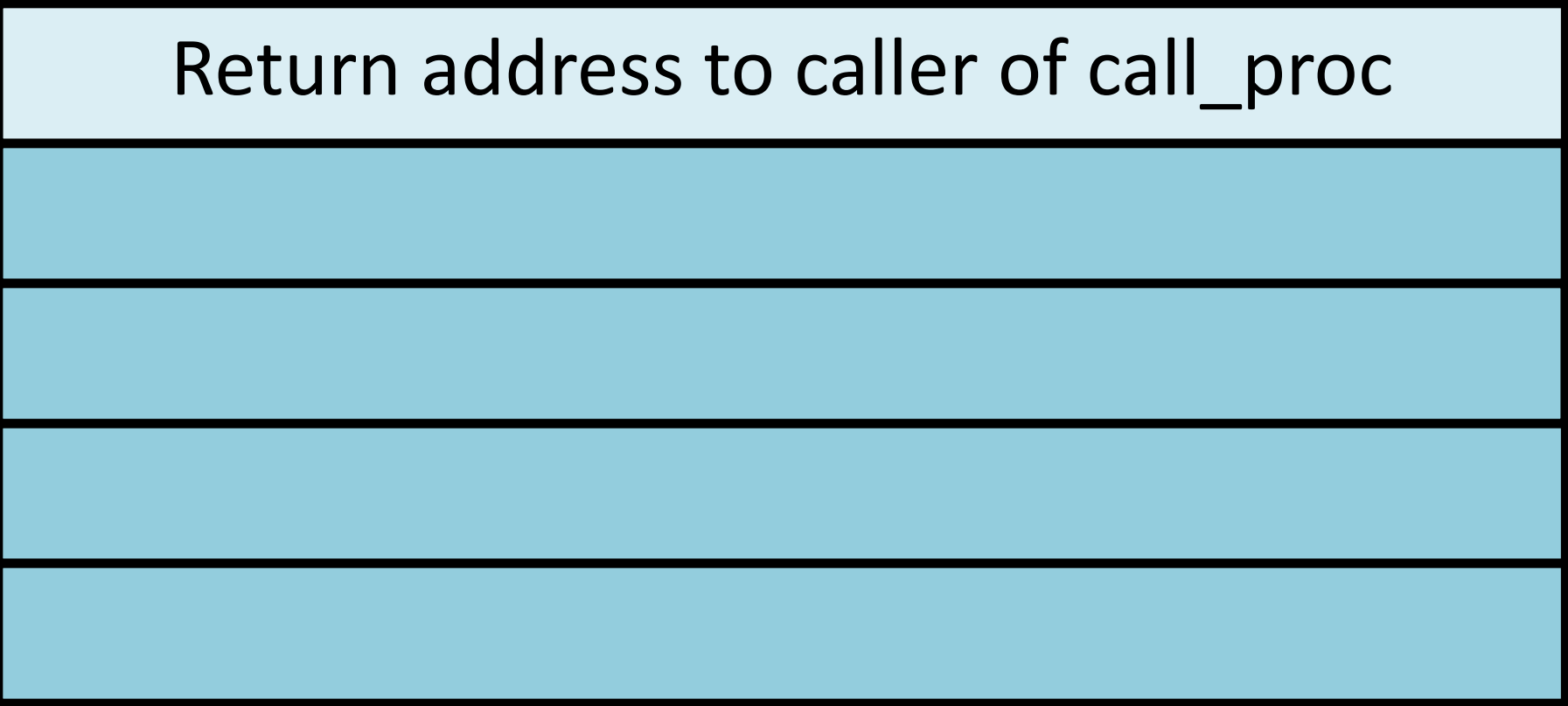
Stack storage example

(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
```



←%rsp

Procedure Summary

call, ret, push, pop

Stack discipline fits procedure call / return.*

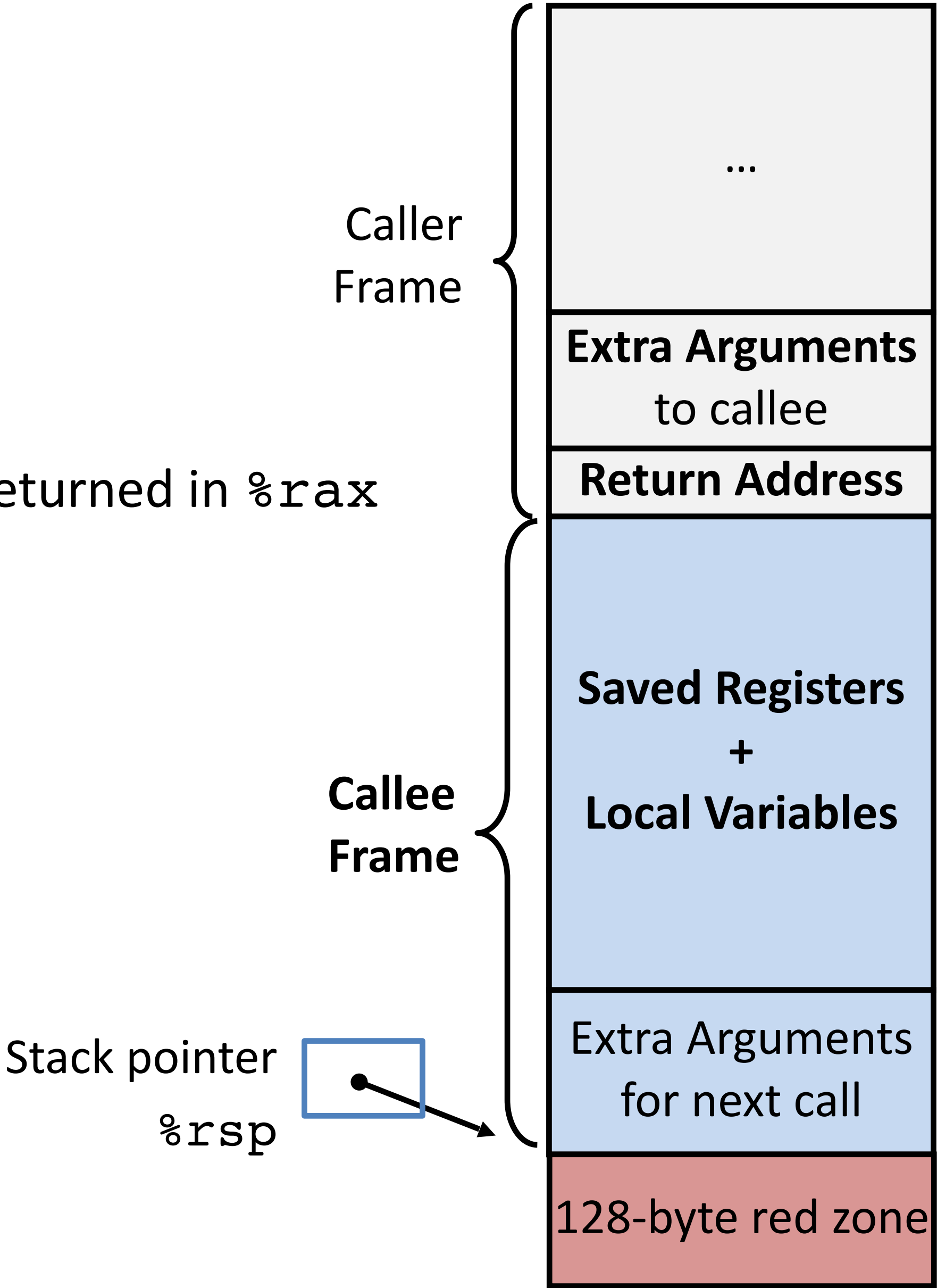
If P calls Q: Q (and calls by Q) returns before P

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



functions allowed to use this before changing %rsp