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/ 1

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

Why procedures?

Why functions? Why methods?

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

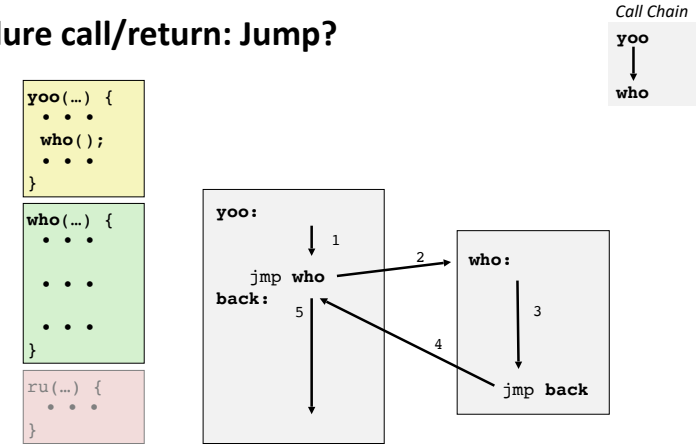
Answer: procedural abstraction

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? ??

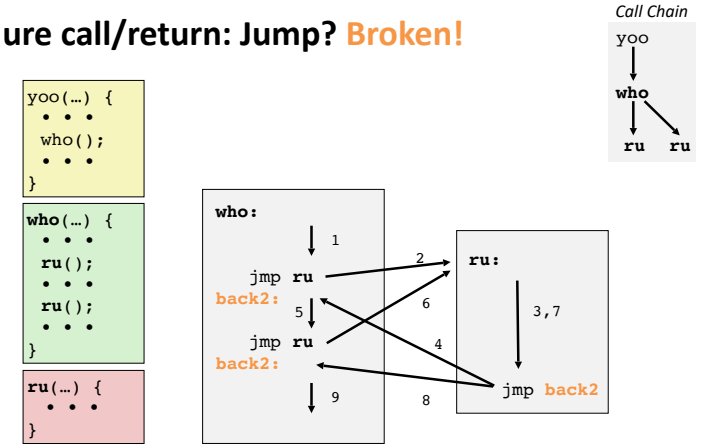
Procedure call/return: Jump?



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

5

Procedure call/return: Jump? Broken!



But what if we want to call a function from multiple places in the code?
Broken: needs to track context.

6

Implementing procedures

requires **separate storage *per call!***
(not just per procedure)

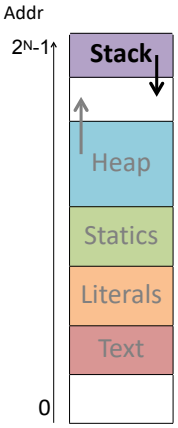
Have we already seen how this is done?

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? ??

7

Memory Layout

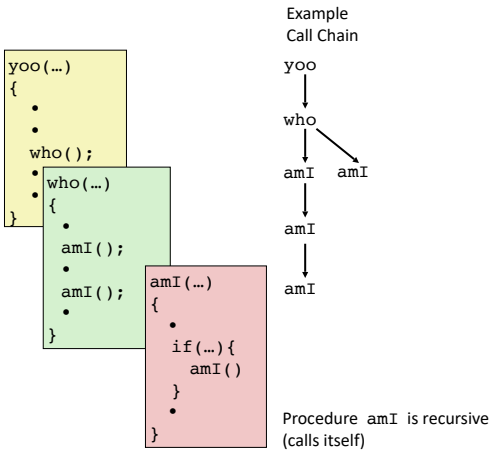
reminder



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

8

Call stack tracks context

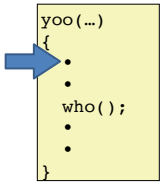


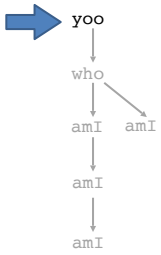
Example
Call Chain

Procedure `amI` is recursive
(calls itself)

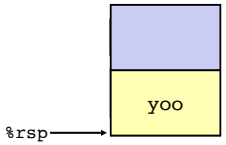
9

Call stack tracks context



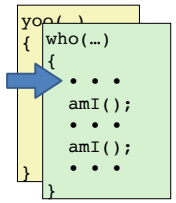


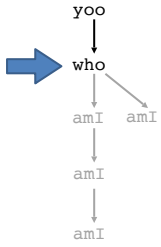
Stack



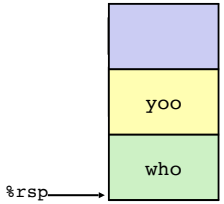
10

Call stack tracks context



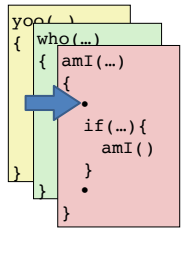


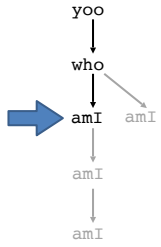
Stack



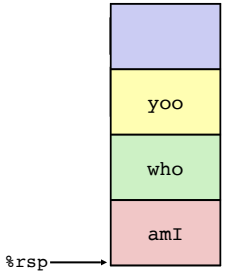
11

Call stack tracks context



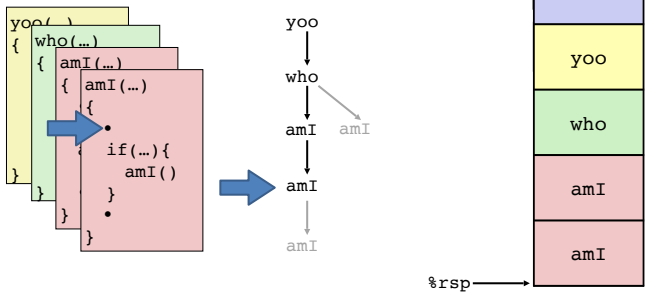


Stack



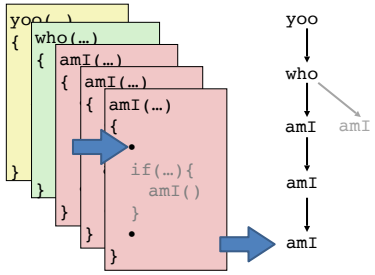
12

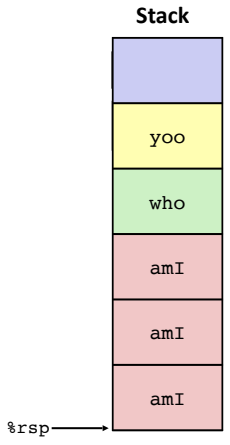
Call stack tracks context



13

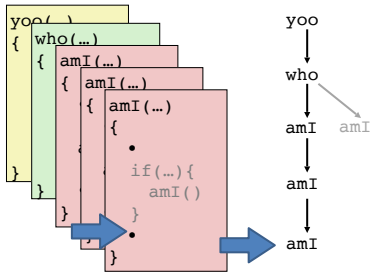
Call stack tracks context

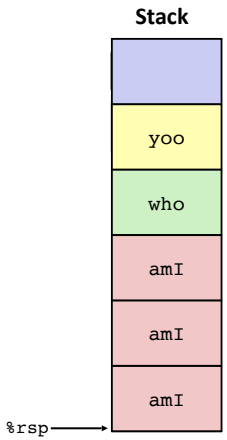




14

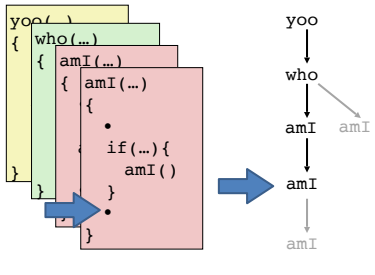
Call stack tracks context

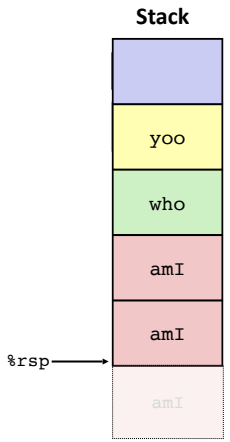




15

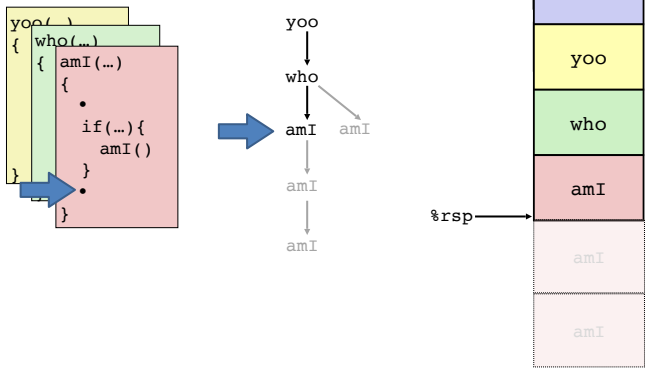
Call stack tracks context





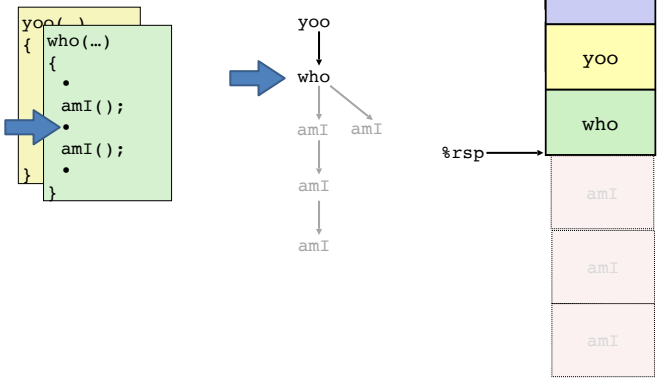
16

Call stack tracks context



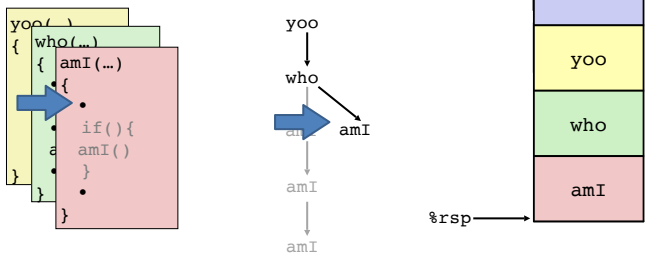
17

Call stack tracks context



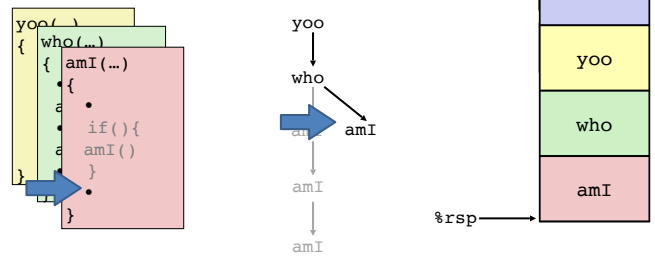
18

Call stack tracks context



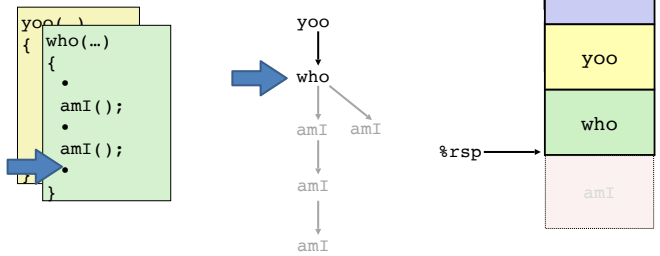
19

Call stack tracks context



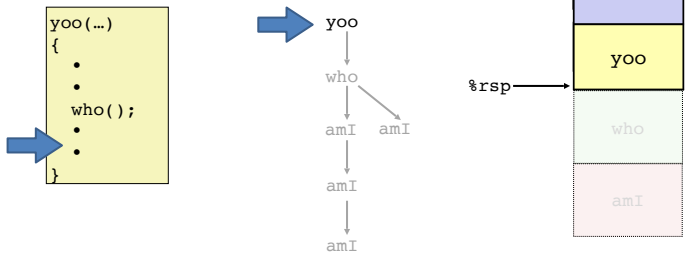
20

Call stack tracks context



21

Call stack tracks context

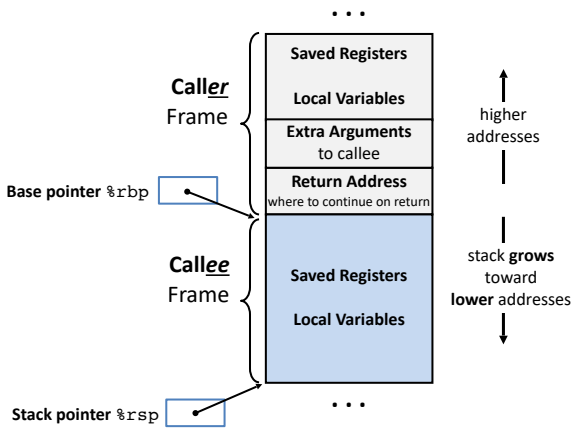


22

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.



23

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

24

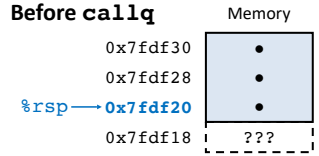
Call example

```
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`



%rsp	%rip
0x7fdf20	0x400544

25

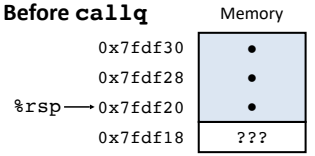
Call example

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


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

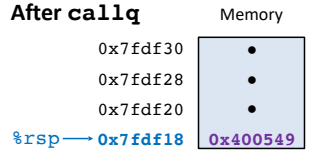
`callq` *target*
1. Push return address on stack
2. Jump to *target*

Before `callq`



%rsp	%rip
0x7fdf20	0x400544

After `callq`



%rsp	%rip
0x7fdf18	0x400550

26

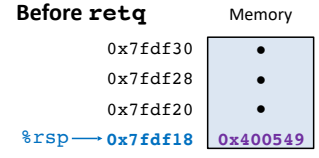
Return example

```
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`



%rsp	%rip
0x7fdf18	0x400557

27

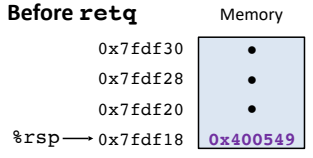
Return example

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

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

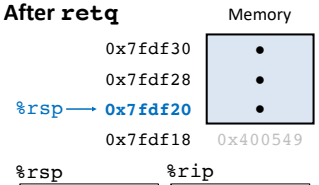
`retq`
1. Pop return address from stack
2. Jump to return address

Before `retq`



%rsp	%rip
0x7fdf18	0x400557

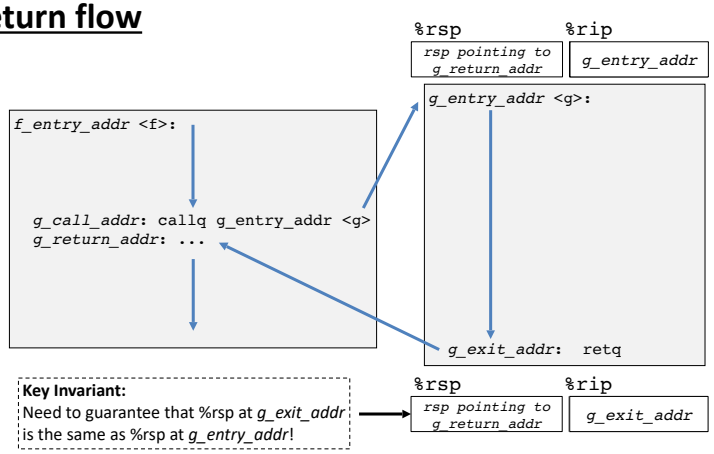
After `retq`



%rsp	%rip
0x7fdf20	400549

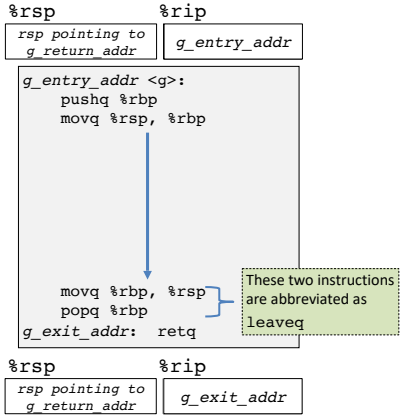
28

Call/Return flow



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

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



Procedure data flow conventions

Recall:

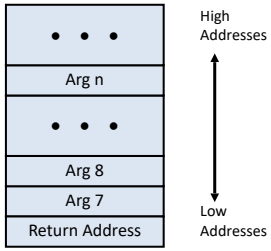
First 6 arguments: passed in **registers**

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

Return value: passed in **%rax**

%rax

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



Procedure call / stack frame example

```
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
```

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

Passes address of local variable (in stack).

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)

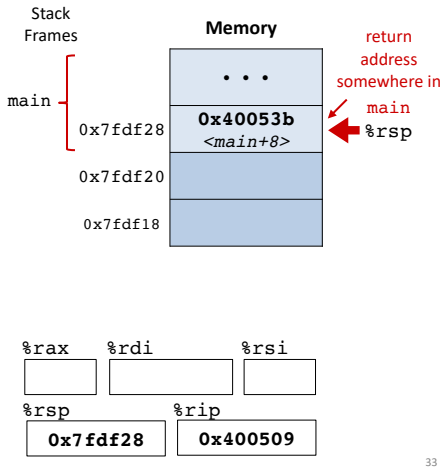
```
long step_up() {
    long v1 = 240;
    long v2 = increment(&v1, 61);
    return v1+v2;
}
```

red line shows %rip

```
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
```

main called step_up



33

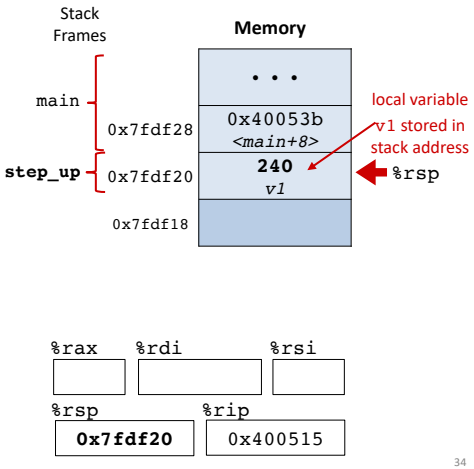
Procedure call example (step 1)

```
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
```

Allocate space for local vars



34

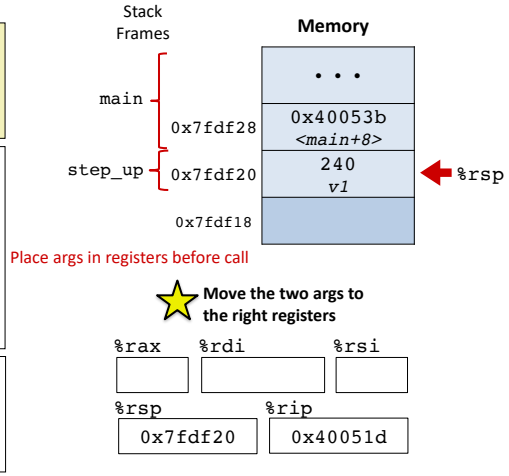
Procedure call example (step 2)

```
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
```

Set up args for call to increment



35

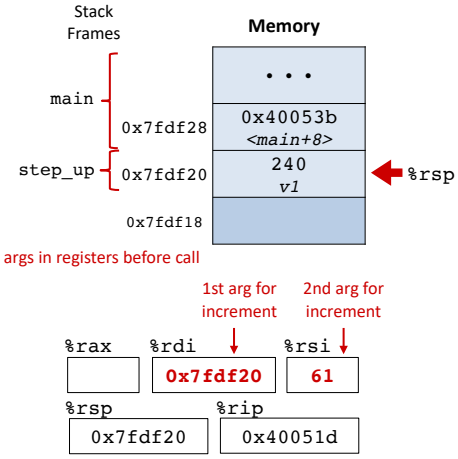
Procedure call example (step 2)

```
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
```

Set up args for call to increment



36

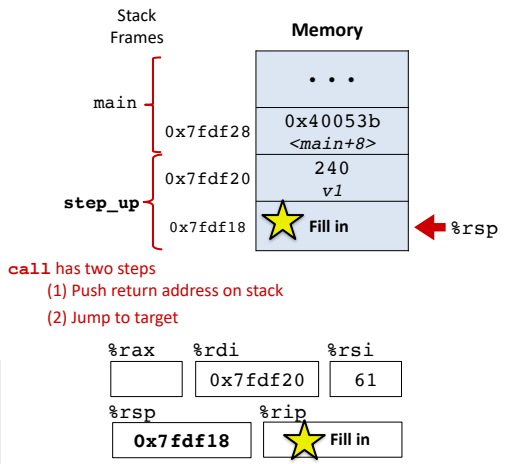
Procedure call example (step 3)

```
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 increment



`call` has two steps

(1) Push return address on stack

(2) Jump to target

37

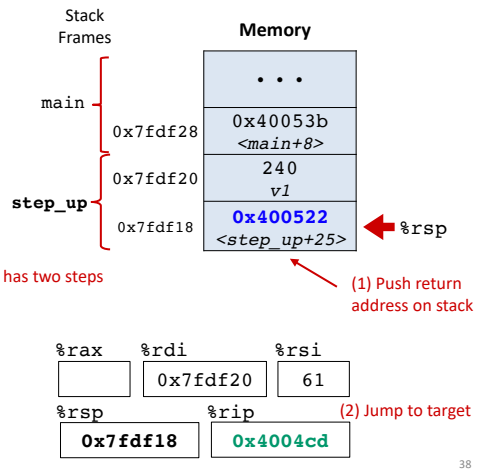
Procedure call example (step 3)

```
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 increment



`call` has two steps

(1) Push return address on stack

(2) Jump to target

38

Procedure call example (step 4)

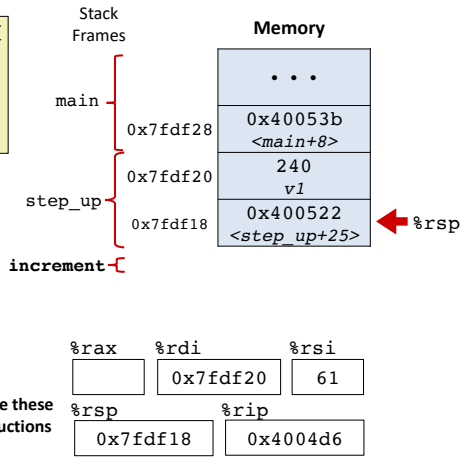
```
long increment(long* p, long val) {
    long x = *p;
    long y = x + val;
    *p = y;
    return x;
}
```

```
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

Run increment



39

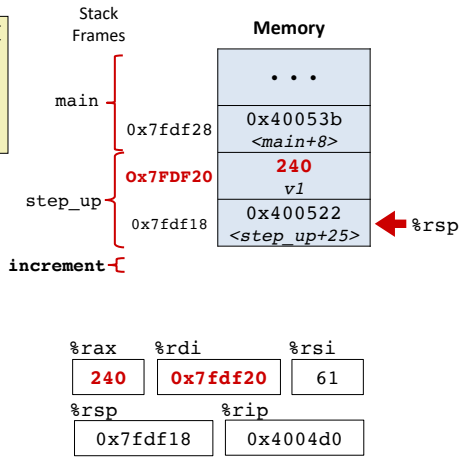
Procedure call example (step 4)

```
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
```

Run increment



40

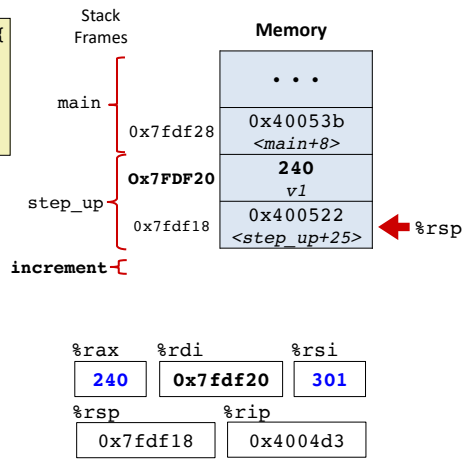
Procedure call example (step 4)

```
lo  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
```

Run increment



41

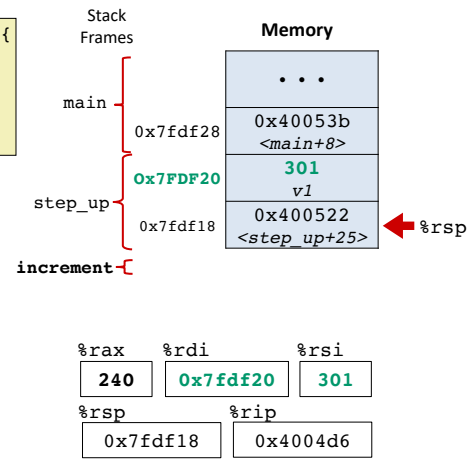
Procedure call example (step 4)

```
lo  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
```

Run increment



42

Procedure call example (step 5a)

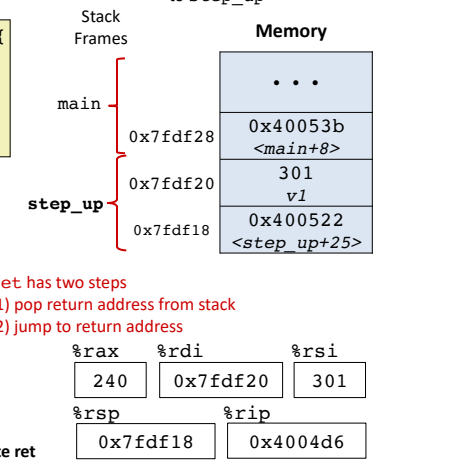
```
lo  long increment(long* p, long val) {
        long x = *p;
        long y = x + val;
        *p = y;
        return x;
    }
```

```
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 ret

Return from increment to step_up



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

43

Procedure call example (step 5b)

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

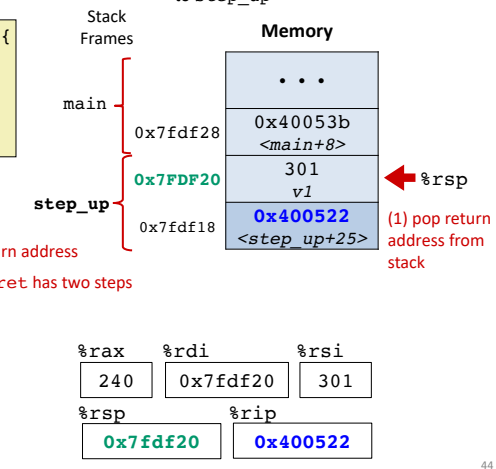
```
step_up:
400509:  subq  $8, %rsp
40050d:  movq  $240, (%rsp)
400515:  movq  %rsp, %rdi
400518:  movl  $61, %esi
40051d:  callq 4004cd <increment>
400522:  addq  (%rsp), %rax
400526:  addq  $8, %rsp
40052a:  retq
```

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

(2) jump to return address

ret has two steps

Return from increment to step_up



44

Procedure call example (step 6)

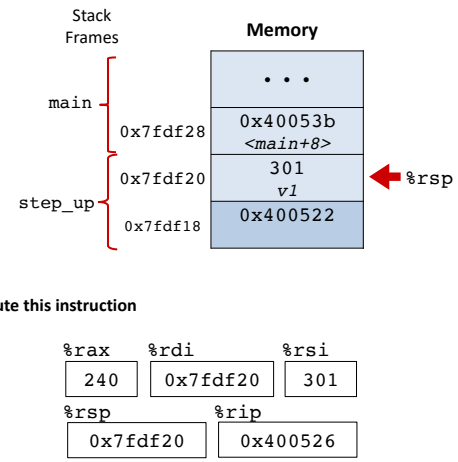
```
long step_up() {
    long v1 = 240;
    long v2 = increment(&v1, 61);
    return v1+v2;
}
```

```
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
```

Execute this instruction

Prepare step_up result



45

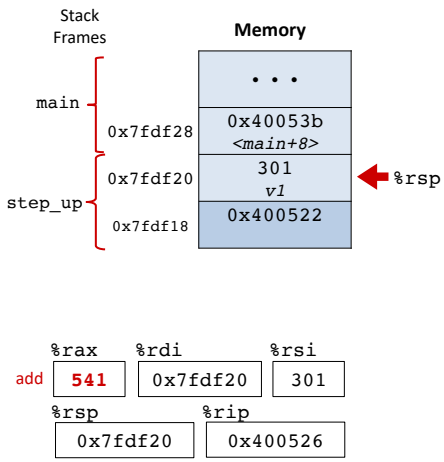
Procedure call example (step 6)


```
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
```

Prepare step_up result



46

Procedure call example (step 7)

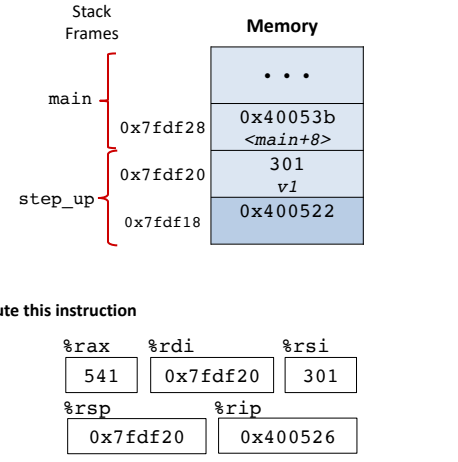
```
long step_up() {
    long v1 = 240;
    long v2 = increment(&v1, 61);
    return v1+v2;
}
```

```
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
```

Execute this instruction

Deallocate space for local vars



47

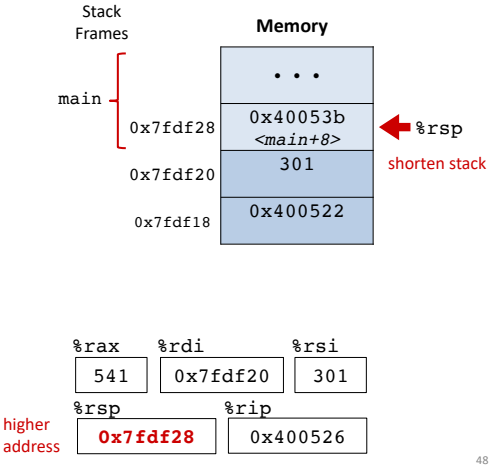
Procedure call example (step 7)


```
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
```

Deallocate space for local vars



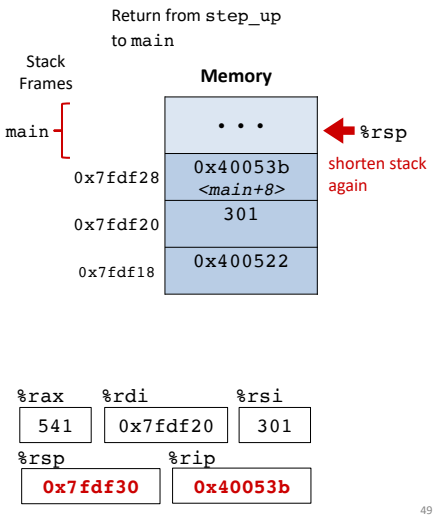
48

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
```



49

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? ??

50

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

51

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

52

Callee-save example (step 0)

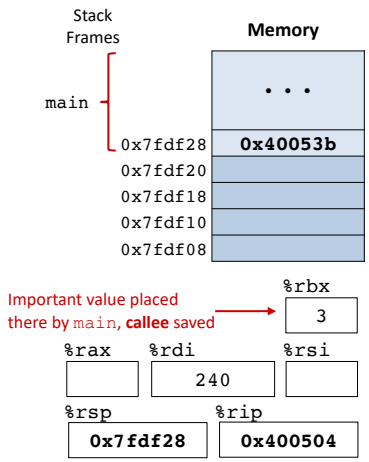
Similar function, but now takes an arg for the local variable

```
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)

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

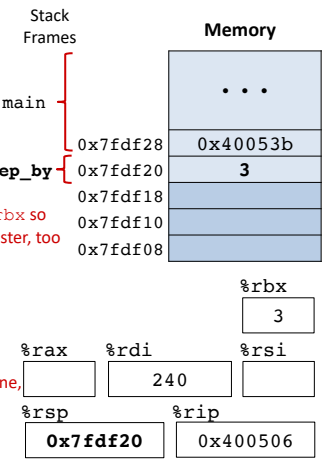
```
step_by:
400504:  pushq %rbx
400506:  movq  %rdi, %rbx
400509:  subq  $16, %rsp
40050d:  movq  %rdi, (%rsp)
400515:  movq  %rsp, %rdi
400518:  movl  $61, %esi
40051d:  callq 4004cd <increment>
400522:  addq  %rbx, %rax
400525:  addq  $16, %rsp
400529:  popq  %rbx
40052b:  retq
```

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

Save register %rbx



Callee-save example (step 2)

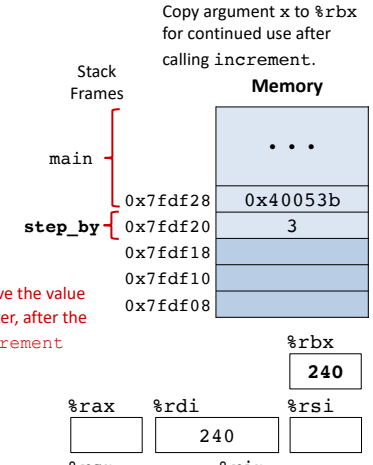
```
long step_by(long x) {
    long v1 = x;
    long v2 = increment(&v1, 61);
    return x + v2;
}
```

```
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

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



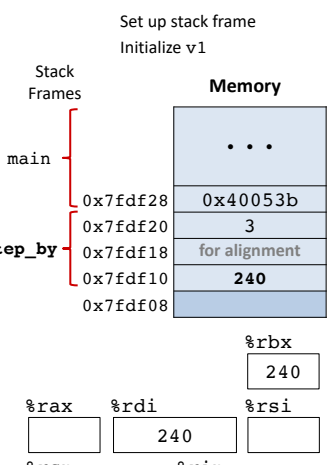
Callee-save example (step 3)

```
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

Set up stack frame
Initialize v1



Convention: at call, %rsp must be a multiple of 16

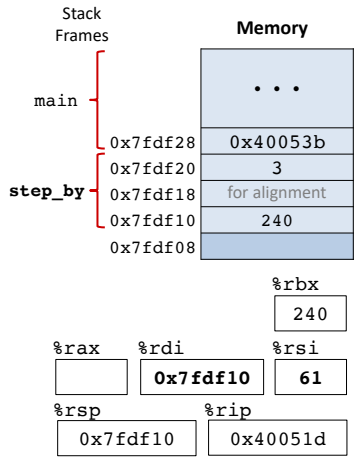
Callee-save example (step 4)

```
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

Set up arguments



57

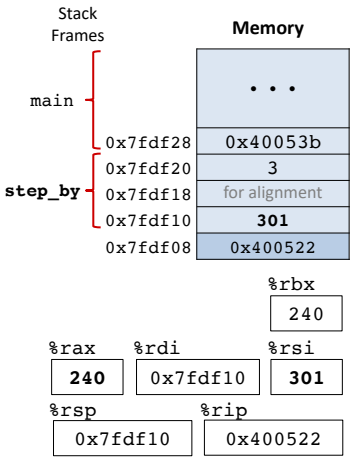
Callee-save example (step 5)

```
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

Call, execute, and return from `increment`



58

Callee-save example (step 6)

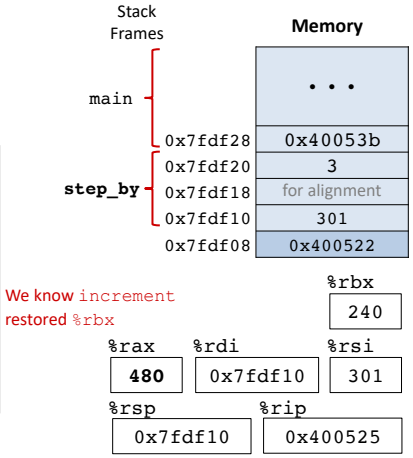
```
long step_by(long x) {
    long v1 = x;
    long v2 = increment(&v1, 61);
    return x + v2;
}
```

```
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

Prepare return value

We know `increment` restored %rbx



59

Callee-save example (step 7)

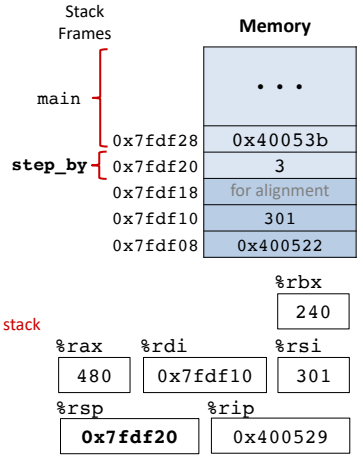
```
long step_by(long x) {
    long v1 = x;
    long v2 = increment(&v1, 61);
    return x + v2;
}
```

```
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

Clean up stack frame

Shrink stack



60

Callee-save example (step 8)

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

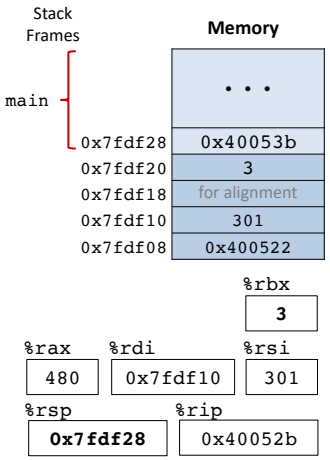
```
step_by:
400504:  pushq %rbx
400506:  movq  %rdi, %rbx
400509:  subq  $16, %rsp
40050d:  movq  %rdi, (%rsp)
400515:  movq  %rsp, %rdi
400518:  movl  $61, %esi
40051d:  callq 4004cd <increment>
400522:  addq  %rbx, %rax
400525:  addq  $16, %rsp
400529:  popq  %rbx
40052b:  retq
```

Restore %rbx for main

caller saved: %rax, %rdi, %rsi

callee saved: %rbx

Restore register %rbx
Ready to return



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

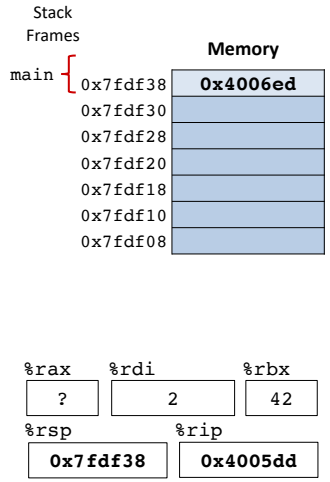
x&1 in %rbx across call

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);
  }
}
```

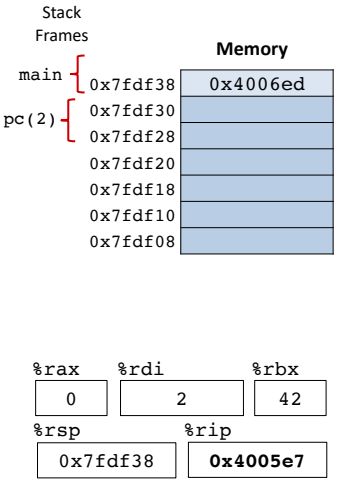
```
pcount:
4005dd:  movl  $0, %eax
4005e2:  testq %rdi, %rdi
4005e5:  je    4005fa <.L6>
4005e7:  pushq %rbx
4005e8:  movq  %rdi, %rbx
4005eb:  andl  $1, %ebx
4005ee:  shrq  %rdi
4005f1:  callq pcount
4005f6:  addq  %rbx, %rax
4005f9:  popq  %rbx
.L6:
4005fa:  rep
4005fb:  retq
```



Recursion Example: `pcount(2)`

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

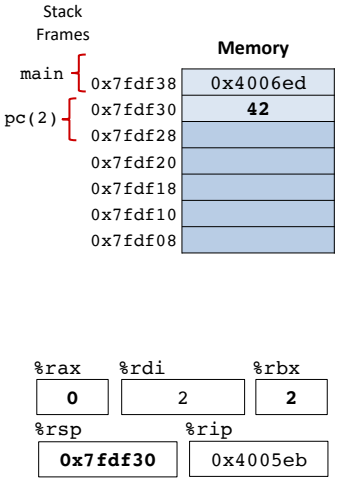
```
pcount:
4005dd:  movl  $0, %eax
4005e2:  testq %rdi, %rdi
4005e5:  je    4005fa <.L6>
4005e7:  pushq %rbx
4005e8:  movq  %rdi, %rbx
4005eb:  andl  $1, %ebx
4005ee:  shrq  %rdi
4005f1:  callq pcount
4005f6:  addq  %rbx, %rax
4005f9:  popq  %rbx
.L6:
4005fa:  rep
4005fb:  retq
```



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
```



%rax	%rdi	%rbx
0	2	2

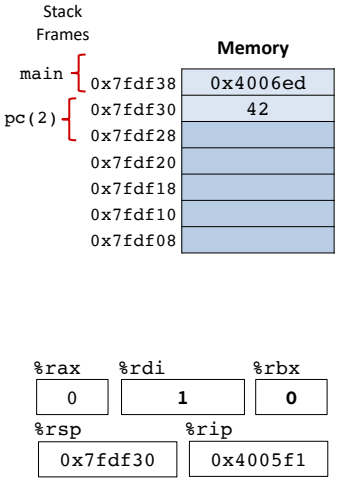
%rsp	%rip
0x7fdf30	0x4005eb

65

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
```



%rax	%rdi	%rbx
0	1	0

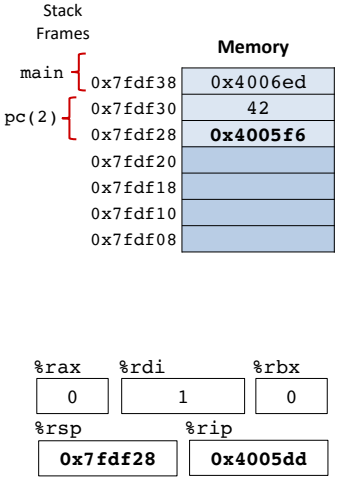
%rsp	%rip
0x7fdf30	0x4005f1

66

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
```



%rax	%rdi	%rbx
0	1	0

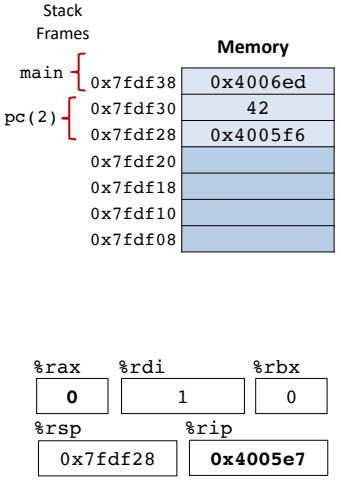
%rsp	%rip
0x7fdf28	0x4005dd

67

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
```



%rax	%rdi	%rbx
0	1	0

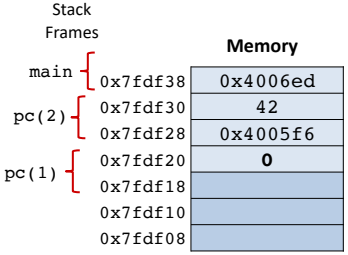
%rsp	%rip
0x7fdf28	0x4005e7

68

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
```



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

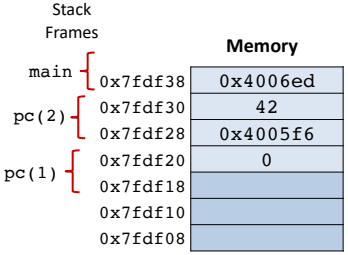
%rsp	%rip
0x7fdf20	0x4005eb

69

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
```



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

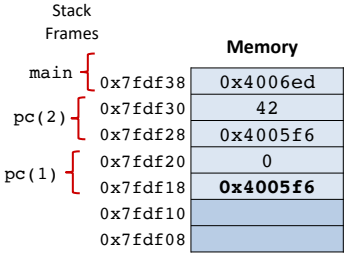
%rsp	%rip
0x7fdf20	0x4005f1

70

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
```



%rax	%rdi	%rbx
0	0	1

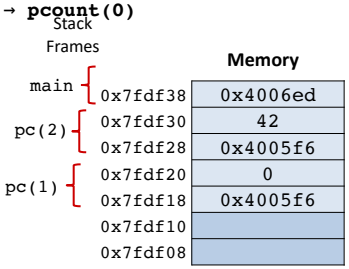
%rsp	%rip
0x7fdf18	**0x4005dd**

71

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
```



%rax	%rdi	%rbx
0	0	1

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

72

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 / 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
0x7fdf20	0x4005f6

73

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
0x7fdf20	0x4005f6

74

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
1	0	1

%rsp	%rip
0x7fdf20	0x4005f9

75

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
	0x7fdf20	0
	0x7fdf18	0x4005f6
	0x7fdf10	
	0x7fdf08	

%rax	%rdi	%rbx
1	0	0

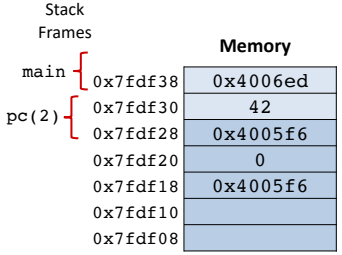
%rsp	%rip
0x7fdf28	0x4005fa

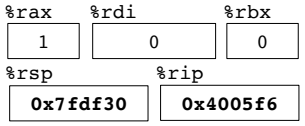
76

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
```



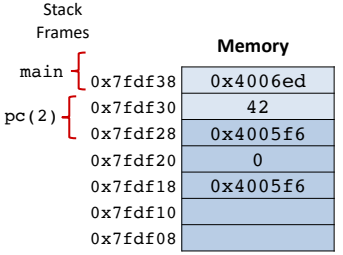


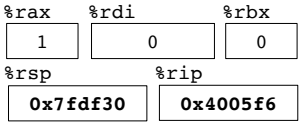
77

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
```



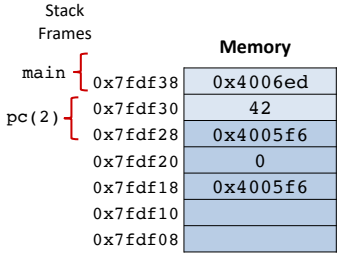


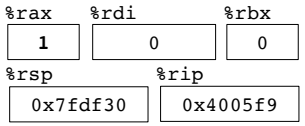
78

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
```



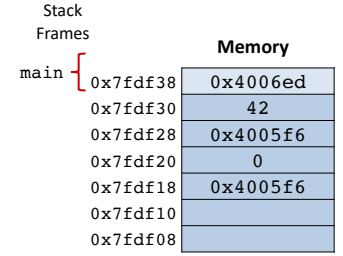


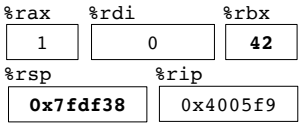
79

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
```



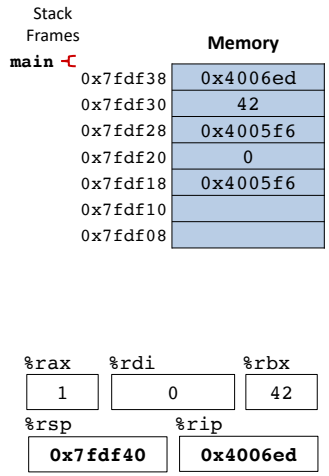


80

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
```



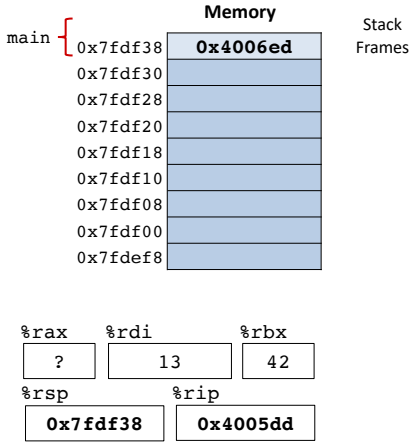
Recursion Exercise: `pcount(13)`

(1) What do the stack & registers look like at the base case?
(2) What is the final value returned by pcount(13)?

ex

```
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
```



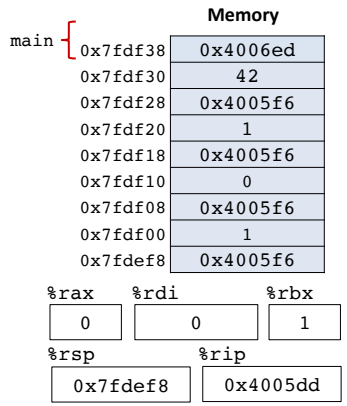
Recursion Exercise: `pcount(13)` Solutions

(1) Q: What do the stack & registers look like at the base case?
A: See below

ex

```
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
```



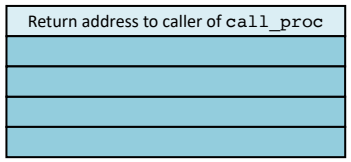
(2) Q: What is the final value returned by pcount(13)? A: 3

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
  • • •
```



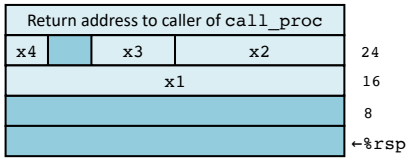
Stack storage example

(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
  • • •
```



85

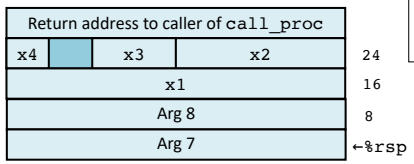
Stack storage example

(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`

86

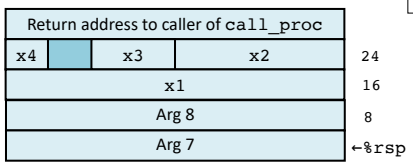
Stack storage example

(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
```



87

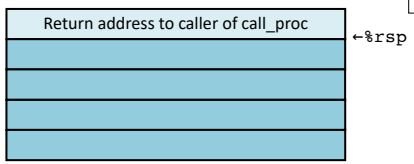
Stack storage example

(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
```



88

Procedure Summary

`call, ret, push, pop`

Stack discipline fits procedure call / return.*

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

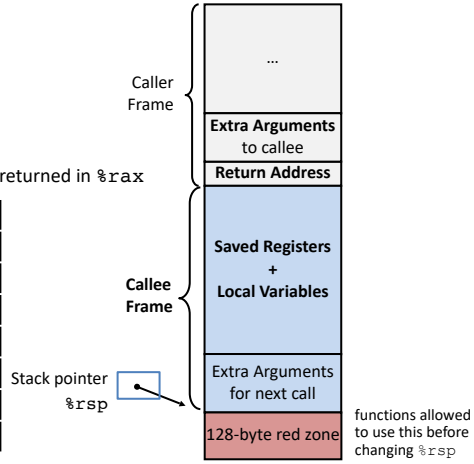
Conventions support arbitrary function calls.

Register-save conventions.

Stack frame saves extra args or local variables. Result returned in `%rax`

`%rax`	Return value – Caller saved
`%rbx`	Callee saved
`%rcx`	Argument #4 – Caller saved
`%rdx`	Argument #3 – Caller saved
`%rsi`	Argument #2 – Caller saved
`%rdi`	Argument #1 – Caller saved
`%rsp`	Stack pointer
`%rbp`	Callee saved

`%r8`	Argument #5 – Caller saved
`%r9`	Argument #6 – Caller saved
`%r10`	Caller saved
`%r11`	Caller Saved
`%r12`	Callee saved
`%r13`	Callee saved
`%r14`	Callee saved
`%r15`	Callee saved



89