

COMP201

Computer Systems & Programming

Lecture #18 – x86-64 Procedures



KOÇ
UNIVERSITY

Aykut Erdem // Koç University // Fall 2022

Recap

- Assembly Execution and `%rip`
- Control Flow Mechanics
 - Condition Codes
 - Assembly Instructions
- If statements
- Loops
 - While loops
 - For loops
- Other Instructions That Depend On Condition Codes

Practice: “Escape Room”

```
escapeRoom:
    leal (%rdi,%rdi), %eax
    cmpl $5, %eax
    jg .L3
    cmpl $1, %edi
    jne .L4
    movl $1, %eax
    ret
.L3:
    movl $1, %eax
    ret
.L4:
    movl $0, %eax
    ret
```

What must be passed to the escapeRoom function such that it returns true (1) and not false (0)?

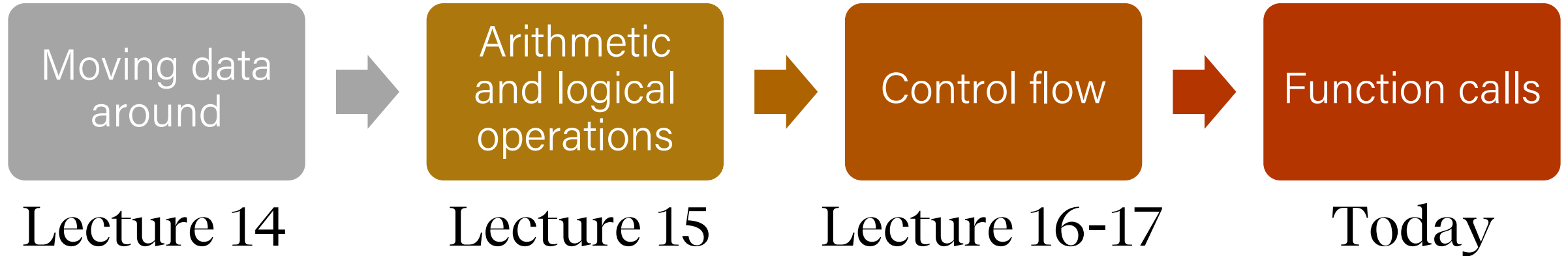
Practice: “Escape Room”

```
escapeRoom:
    leal (%rdi,%rdi), %eax
    cmpl $5, %eax
    jg .L3
    cmpl $1, %edi
    jne .L4
    movl $1, %eax
    ret
.L3:
    movl $1, %eax
    ret
.L4:
    movl $0, %eax
    ret
```

What must be passed to the escapeRoom function such that it returns true (1) and not false (0)?

First param > 2 or == 1.

Learning Assembly



Learning Goals

- Learn how assembly calls functions and manages stack frames.
- Learn the rules of register use when calling functions.

Plan for Today

- Revisiting `%rip`
- Calling Functions
 - The Stack
 - Passing Control
 - Passing Data
 - Local Storage
- Register Restrictions
- Pulling it all together: recursion example

Disclaimer: Slides for this lecture were borrowed from

—Nick Troccoli's Stanford CS107 class

—Randal E. Bryant and David R. O'Hallaron's CMU 15-213 class

Lecture Plan

- Revisiting `%rip`
- Calling Functions
 - The Stack
 - Passing Control
 - Passing Data
 - Local Storage
- Register Restrictions
- Pulling it all together: recursion example

%rip

- **%rip** is a special register that points to the next instruction to execute.
- **Let's dive deeper into how %rip works, and how jumps modify it.**

%rip

```
void loop() {  
    int i = 0;  
    while (i < 100) {  
        i++;  
    }  
}
```

```
0000000000400570 <loop>:  
0x400570 <+0>:  b8 00 00 00 00 mov $0x0,%eax  
0x400575 <+5>:  eb 03          jmp 0x40057a <loop+10>  
0x400577 <+7>:  83 c0 01          add $0x1,%eax  
0x40057a <+10>: 83 f8 63          cmp $0x63,%eax  
0x40057d <+13>: 73 f8          jle 0x400577 <loop+7>  
0x40057f <+15>: f3 c3          repz retq
```

%rip

```
void loop() {  
    int i = 0;  
    while (i < 100) {  
        i++;  
    }  
}
```

```
0000000000400570 <loop>:  
0x400570 <+0>:  b8 00 00 00 00 mov $0x0,%eax  
0x400575 <+5>:  eb 03                jmp 0x40057a <loop+10>  
0x400577 <+7>:  83 c0 01                add $0x1,%eax  
0x40057a <+10>: 83 f8 63                cmp $0x63,%eax  
0x40057d <+13>: 73 f8                jle 0x400577 <loop+7>  
0x40057f <+15>: f3 c3                repz retq
```

These are 0-based offsets in bytes for each instruction relative to the start of this function.

%rip

```
void loop() {  
    int i = 0;  
    while (i < 100) {  
        i++;  
    }  
}
```

```
0000000000400570 <loop>:  
0x400570 <+0>:  b8 00 00 00 00 mov $0x0,%eax  
0x400575 <+5>:  eb 03 jmp 0x40057a <loop+10>  
0x400577 <+7>:  83 c0 01 add $0x1,%eax  
0x40057a <+10>: 83 f8 63 cmp $0x63,%eax  
0x40057d <+13>: 73 f8 jle 0x400577 <loop+7>  
0x40057f <+15>: f3 c3 repz retq
```

These are bytes for the machine code instructions. Instructions are variable length.

%rip

```
void loop() {  
    int i = 0;  
    while (i < 100) {  
        i++;  
    }  
}
```

```
0000000000400570 <loop>:  
0x400570 <+0>:  b8 00 00 00 00 mov $0x0,%eax  
0x400575 <+5>:  eb 03      jmp 0x40057a <loop+10>  
0x400577 <+7>:  83 c0 01      add $0x1,%eax  
0x40057a <+10>: 83 f8 63      cmp $0x63,%eax  
0x40057d <+13>: 73 f8      jle 0x400577 <loop+7>  
0x40057f <+15>: f3 c3      repz retq
```

%rip

0000000000400570 <loop>:

0x400570 <+0>: b8 00 00 00 00 mov \$0x0,%eax

0x400575 <+5>: eb 03 jmp 0x40057a <loop+10>

0x400577 <+7>: 83 c0 01 add \$0x1,%eax

0x40057a <+10>: 83 f8 63 cmp \$0x63,%eax

0x40057d <+13>: 73 f8 jle 0x400577 <loop+7>

0x40057f <+15>: f3 c3 repz retq

%rip

0000000000400570 <loop>:

0x400570 <+0>: b8 00 00 00 00 mov \$0x0,%eax

0x400575 <+5>: eb 03 jmp 0x40057a <loop+10>

0x400577 <+7>: 83 c0 01 add \$0x1,%eax

0x40057a <+10>: 83 f8 63 cmp \$0x63,%eax

0x40057d <+13>: 73 f8 jle 0x400577 <loop+7>

0x40057f <+15>: f3 c3 repz retq

0xeb means jmp.

%rip

0000000000400570 <loop>:

0x400570 <+0>: b8 00 00 00 00 mov \$0x0,%eax

0x400575 <+5>: eb 03 jmp 0x40057a <loop+10>

0x400577 <+7>: 83 c0 01 add \$0x1,%eax

0x40057a <+10>: 83 f8 63 cmp \$0x63,%eax

0x40057d <+13>: 73 f8 jle 0x400577 <loop+7>

0x40057f <+15>: f3 c3 repz retq

0x03 is the number of instruction bytes to jump relative to %rip.

With no jump, %rip would advance to the next line. This **jmp** says to then go **3** bytes further!

%rip

0000000000400570 <loop>:

0x400570 <+0>: b8 00 00 00 00 mov \$0x0,%eax

0x400575 <+5>: eb 03 jmp 0x40057a <loop+10>

0x400577 <+7>: 83 c0 01 add \$0x1,%eax

0x40057a <+10>: 83 f8 63 cmp \$0x63,%eax

0x40057d <+13>: 73 f8 jle 0x400577 <loop+7>

0x40057f <+15>: f3 c3 repz retq

0x03 is the number of instruction bytes to jump relative to %rip.

With no jump, %rip would advance to the next line. This **jmp** says to then go **3** bytes further!

%rip


```
0000000000400570 <loop>:  
0x400570 <+0>:  b8 00 00 00 00  mov $0x0,%eax  
0x400575 <+5>:  eb 03                jmp 0x40057a <loop+10>  
0x400577 <+7>:  83 c0 01            add $0x1,%eax  
0x40057a <+10>: 83 f8 63            cmp $0x63,%eax  
0x40057d <+13>: 73 f8                jle 0x400577 <loop+7>  
0x40057f <+15>: f3 c3                repz retq
```



0x73 means **jle**.

%rip

```
0000000000400570 <loop>:  
0x400570 <+0>:  b8 00 00 00 00  mov $0x0,%eax  
0x400575 <+5>:  eb 03                jmp 0x40057a <loop+10>  
0x400577 <+7>:  83 c0 01            add $0x1,%eax  
0x40057a <+10>: 83 f8 63            cmp $0x63,%eax  
0x40057d <+13>: 73 f8                jle 0x400577 <loop+7>  
0x40057f <+15>: f3 c3            repz retq
```



0xf8 is the number of instruction bytes to jump relative to %rip. This is -8 (in two's complement!).

With no jump, %rip would advance to the next line. This **jmp** says to then go **8** bytes back!

%rip

0000000000400570 <loop>:

0x400570 <+0>: b8 00 00 00 00 mov \$0x0,%eax


0x400575 <+5>: eb 03 jmp 0x40057a <loop+10>

0x400577 <+7>: 83 c0 01 add \$0x1,%eax

0x40057a <+10>: 83 f8 63 cmp \$0x63,%eax

0x40057d <+13>: 73 f8 jle 0x400577 <loop+7>

0x40057f <+15>: f3 c3 repz retq



0xf8 is the number of instruction bytes to jump relative to %rip. This is -8 (in two's complement!).

With no jump, %rip would advance to the next line. This **jmp** says to then go **8** bytes back!

Summary: Instruction Pointer

- Machine code instructions live in main memory, just like stack and heap data.
- `%rip` is a register that stores a number (an address) of the next instruction to execute. It marks our place in the program's instructions.
- To advance to the next instruction, special hardware adds the size of the current instruction in bytes.
- **`jmp`** instructions work by adjusting `%rip` by a specified amount.

Lecture Plan

- Revisiting `%rip`
- Calling Functions
 - The Stack
 - Passing Control
 - Passing Data
 - Local Storage
- Register Restrictions
- Pulling it all together: recursion example

How do we call functions in
assembly?

Calling Functions In Assembly

To call a function in assembly, we must do a few things:

- **Pass Control** – `%rip` must be adjusted to execute the callee's instructions, and then resume the caller's instructions afterwards.
- **Pass Data** – we must pass any parameters and receive any return value.
- **Manage Memory** – we must handle any space needs of the callee on the stack.

How does assembly
interact with the stack?

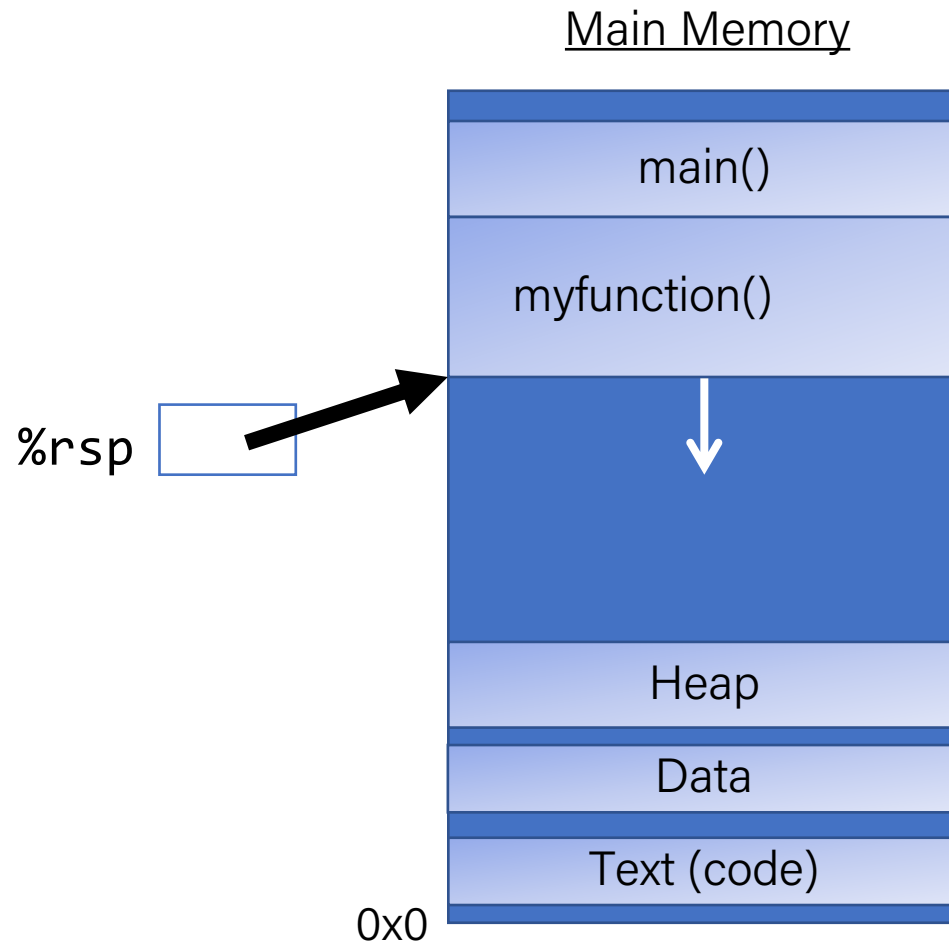
Terminology: **caller** function calls the **callee** function.

Lecture Plan

- Revisiting %rip
- Calling Functions
 - The Stack
 - Passing Control
 - Passing Data
 - Local Storage
- Register Restrictions
- Pulling it all together: recursion example

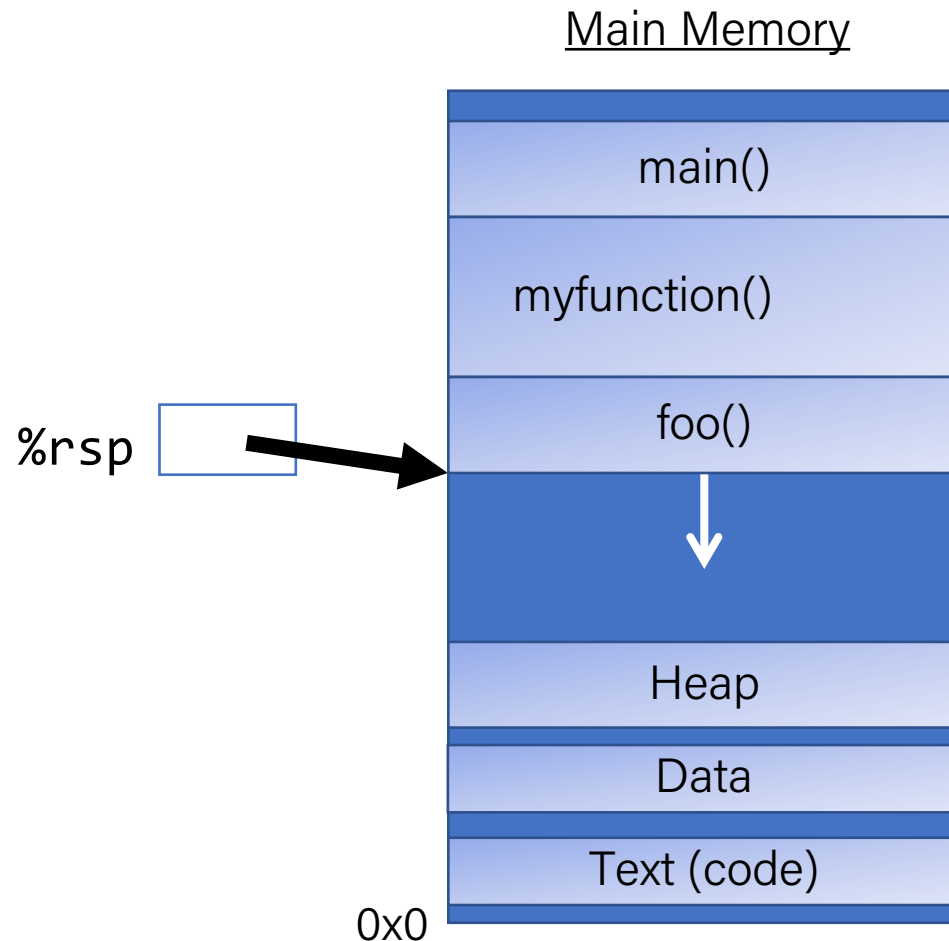
%rsp

- **%rsp** is a special register that stores the address of the current “top” of the stack (the bottom in our diagrams, since the stack grows downwards).



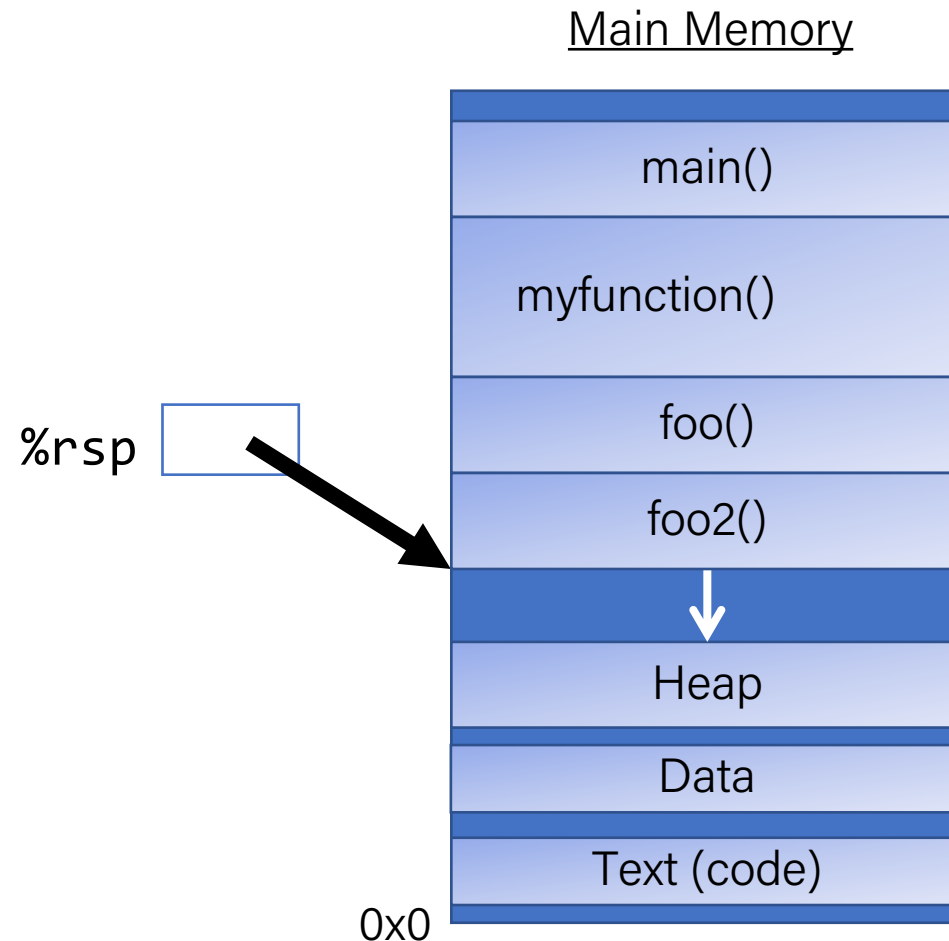
%rsp

- **%rsp** is a special register that stores the address of the current “top” of the stack (the bottom in our diagrams, since the stack grows downwards).



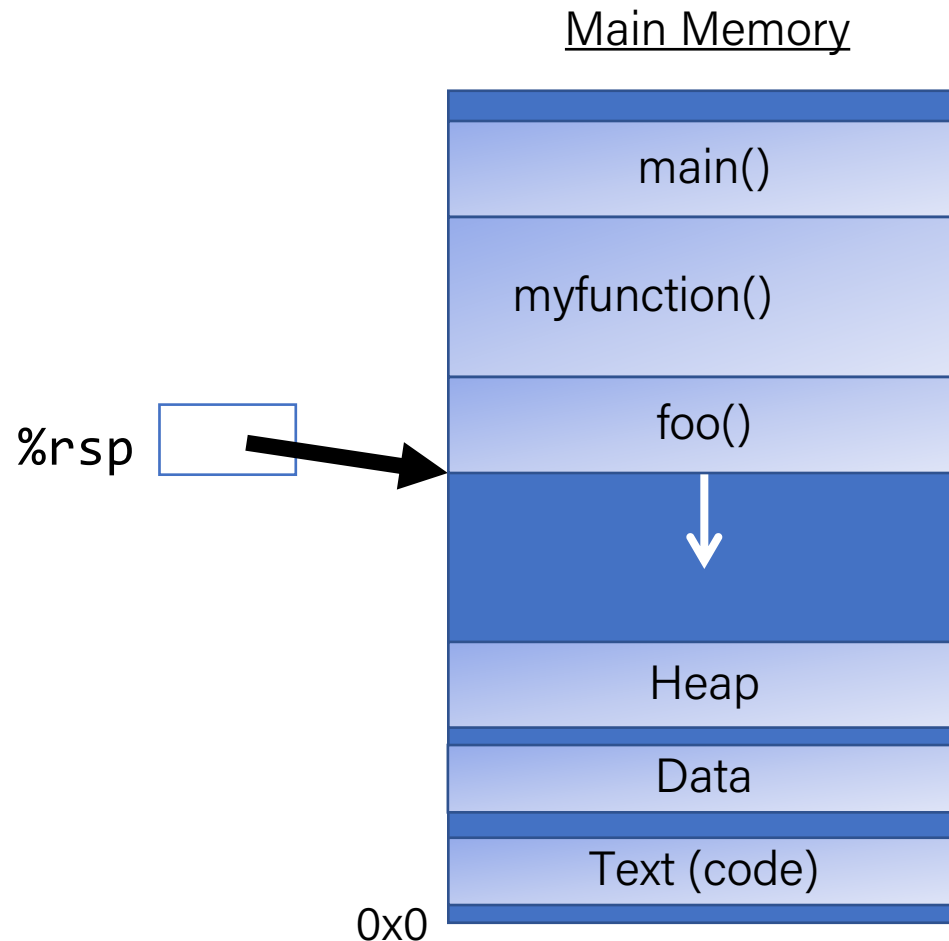
%rsp

- **%rsp** is a special register that stores the address of the current “top” of the stack (the bottom in our diagrams, since the stack grows downwards).



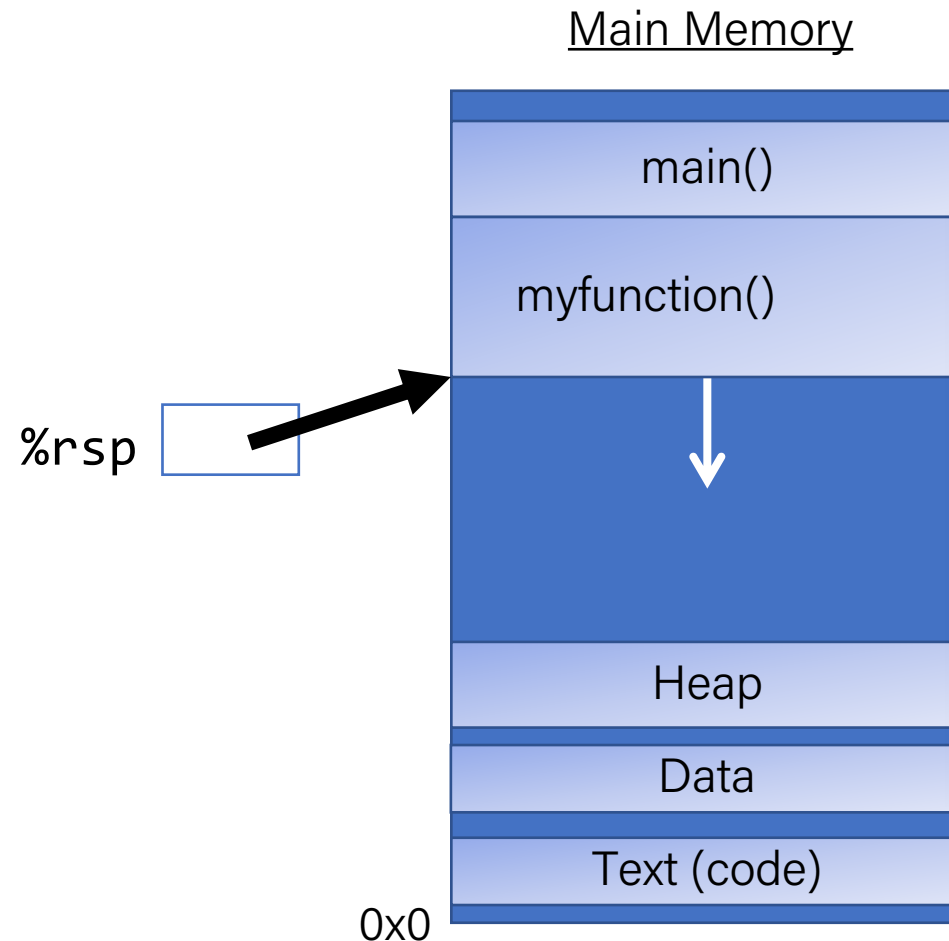
%rsp

- **%rsp** is a special register that stores the address of the current “top” of the stack (the bottom in our diagrams, since the stack grows downwards).



%rsp

- **%rsp** is a special register that stores the address of the current “top” of the stack (the bottom in our diagrams, since the stack grows downwards).



Key idea: %rsp must point to the same place before a function is called and after that function returns, since stack frames go away when a function finishes.

push

- The **push** instruction pushes the data at the specified source onto the top of the stack, adjusting **%rsp** accordingly.

Instruction	Effect
pushq S	$R[\%rsp] \leftarrow R[\%rsp] - 8;$ $M[R[\%rsp]] \leftarrow S$

push

- The **push** instruction pushes the data at the specified source onto the top of the stack, adjusting **%rsp** accordingly.

Instruction	Effect
pushq S	$R[\%rsp] \leftarrow R[\%rsp] - 8;$ $M[R[\%rsp]] \leftarrow S$

push

- The **push** instruction pushes the data at the specified source onto the top of the stack, adjusting **%rsp** accordingly.

Instruction	Effect
pushq S	$R[\%rsp] \leftarrow R[\%rsp] - 8;$ $M[R[\%rsp]] \leftarrow S$

push

- The **push** instruction pushes the data at the specified source onto the top of the stack, adjusting **%rsp** accordingly.

Instruction	Effect
pushq S	$R[\%rsp] \leftarrow R[\%rsp] - 8;$ $M[R[\%rsp]] \leftarrow S$

- This behavior is equivalent to the following, but **pushq** is a shorter instruction:
subq \$8, %rsp
movq S, (%rsp)
- Sometimes, you'll see instructions just explicitly decrement the stack pointer to make room for future data. [More on this later!](#)

pop

- The **pop** instruction pops the topmost data from the stack and stores it in the specified destination, adjusting **%rsp** accordingly.

Instruction	Effect
popq D	$D \leftarrow M[R[\%rsp]]$ $R[\%rsp] \leftarrow R[\%rsp] + 8;$

- Note:** this does not remove/clear out the data! It just increments **%rsp** to indicate the next push can overwrite that location.

pop

- The **pop** instruction pops the topmost data from the stack and stores it in the specified destination, adjusting **%rsp** accordingly.

Instruction	Effect
popq D	$D \leftarrow M[R[\%rsp]]$ $R[\%rsp] \leftarrow R[\%rsp] + 8;$

- This behavior is equivalent to the following, but **popq** is a shorter instruction:

```
movq (%rsp), D
addq $8, %rsp
```

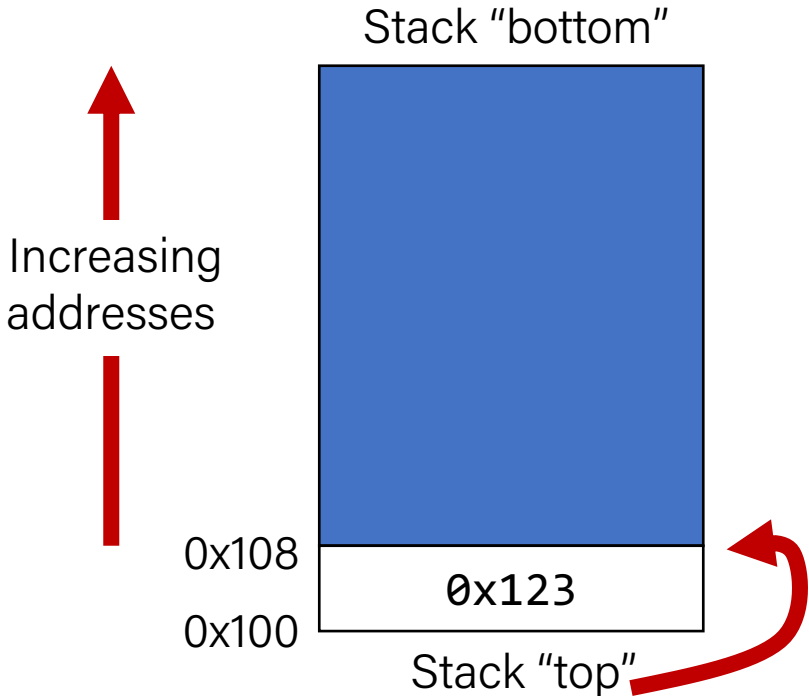
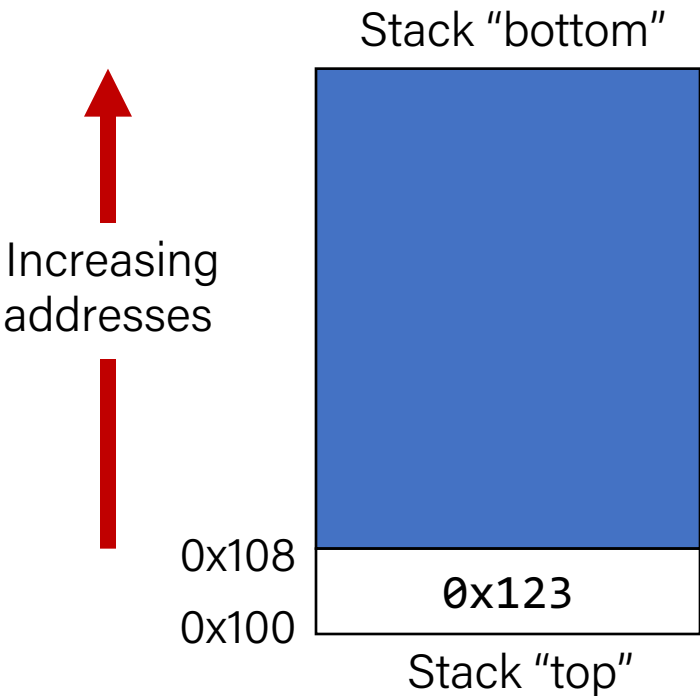
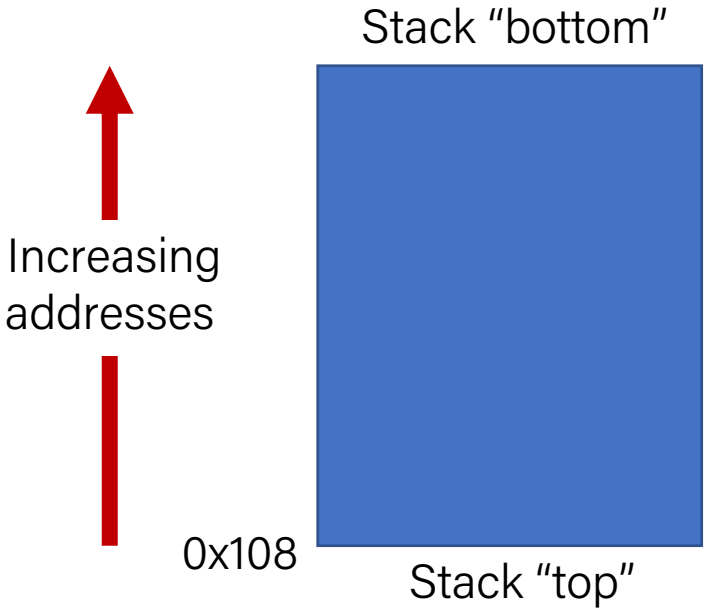
- Sometimes, you'll see instructions just explicitly increment the stack pointer to pop data.

Stack Example

Initially	
%rax	0x123
%rdx	0
%rsp	0x108

pushq %rax	
%rax	0x123
%rdx	0
%rsp	0x100

popq %rdx	
%rax	0x123
%rdx	0x123
%rsp	0x108



Calling Functions In Assembly

To call a function in assembly, we must do a few things:

- **Pass Control** – `%rip` must be adjusted to execute the callee's instructions, and then resume the caller's instructions afterwards.
- **Pass Data** – we must pass any parameters and receive any return value.
- **Manage Memory** – we must handle any space needs of the callee on the stack.

Terminology: **caller** function calls the **callee** function.

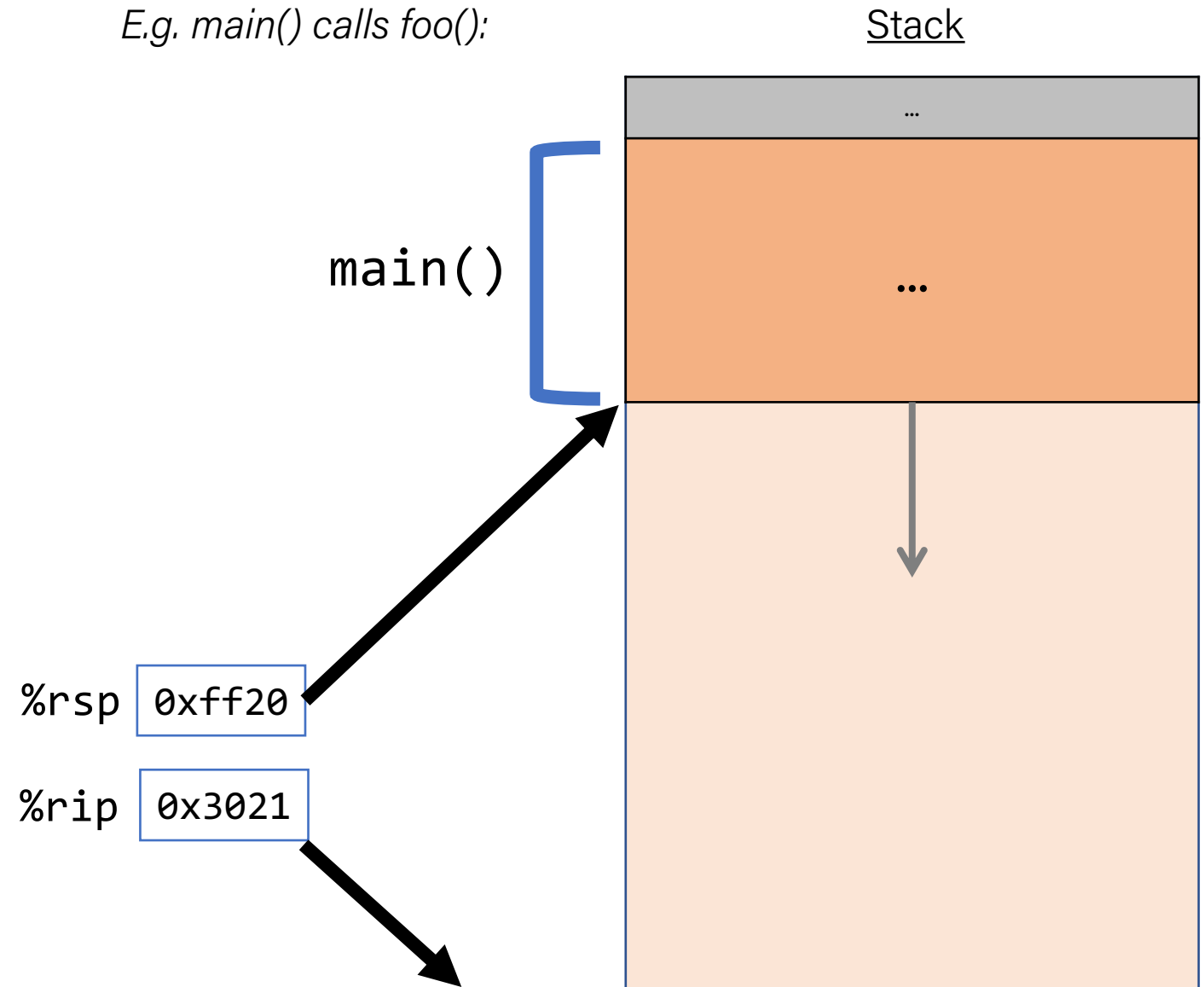
Lecture Plan

- Revisiting `%rip`
- Calling Functions
 - The Stack
 - Passing Control
 - Passing Data
 - Local Storage
- Register Restrictions
- Pulling it all together: recursion example

Remembering Where We Left Off

Problem: `%rip` points to the next instruction to execute. To call a function, we must remember the *next* caller instruction to resume at after.

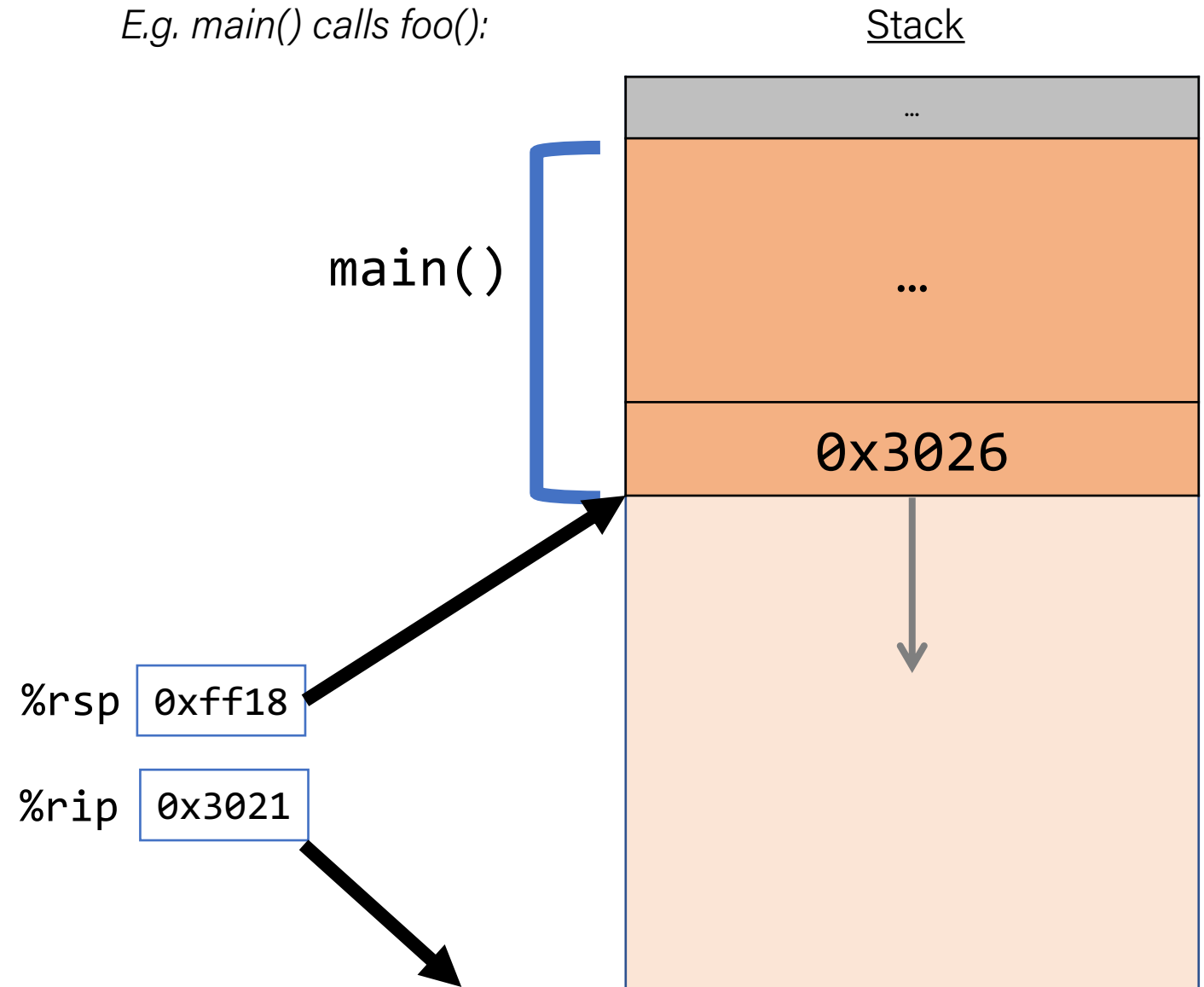
Solution: push the next value of `%rip` onto the stack. Then call the function. When it is finished, put this value back into `%rip` and continue executing.



Remembering Where We Left Off

Problem: `%rip` points to the next instruction to execute. To call a function, we must remember the *next* caller instruction to resume at after.

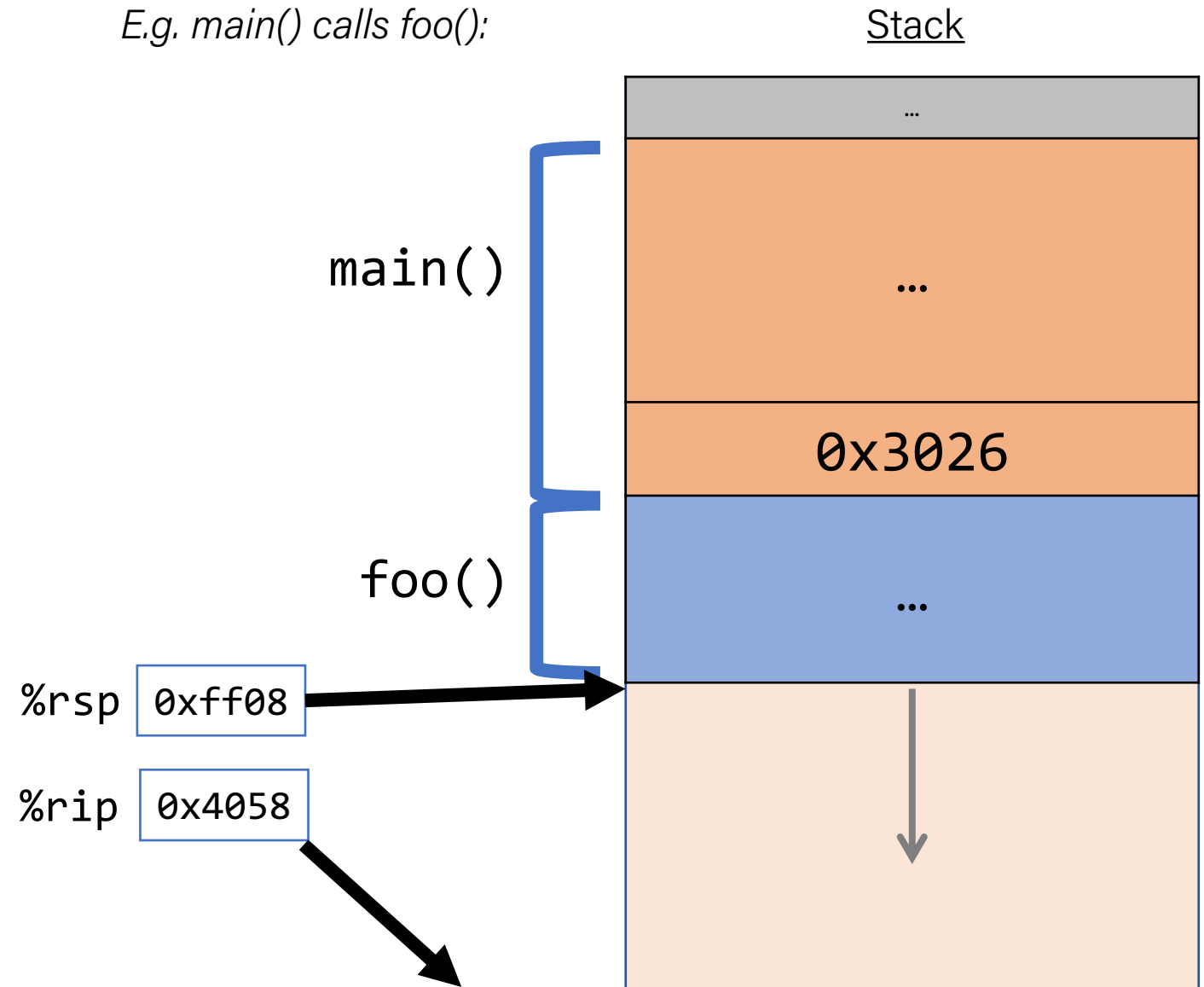
Solution: push the next value of `%rip` onto the stack. Then call the function. When it is finished, put this value back into `%rip` and continue executing.



Remembering Where We Left Off

Problem: `%rip` points to the next instruction to execute. To call a function, we must remember the *next* caller instruction to resume at after.

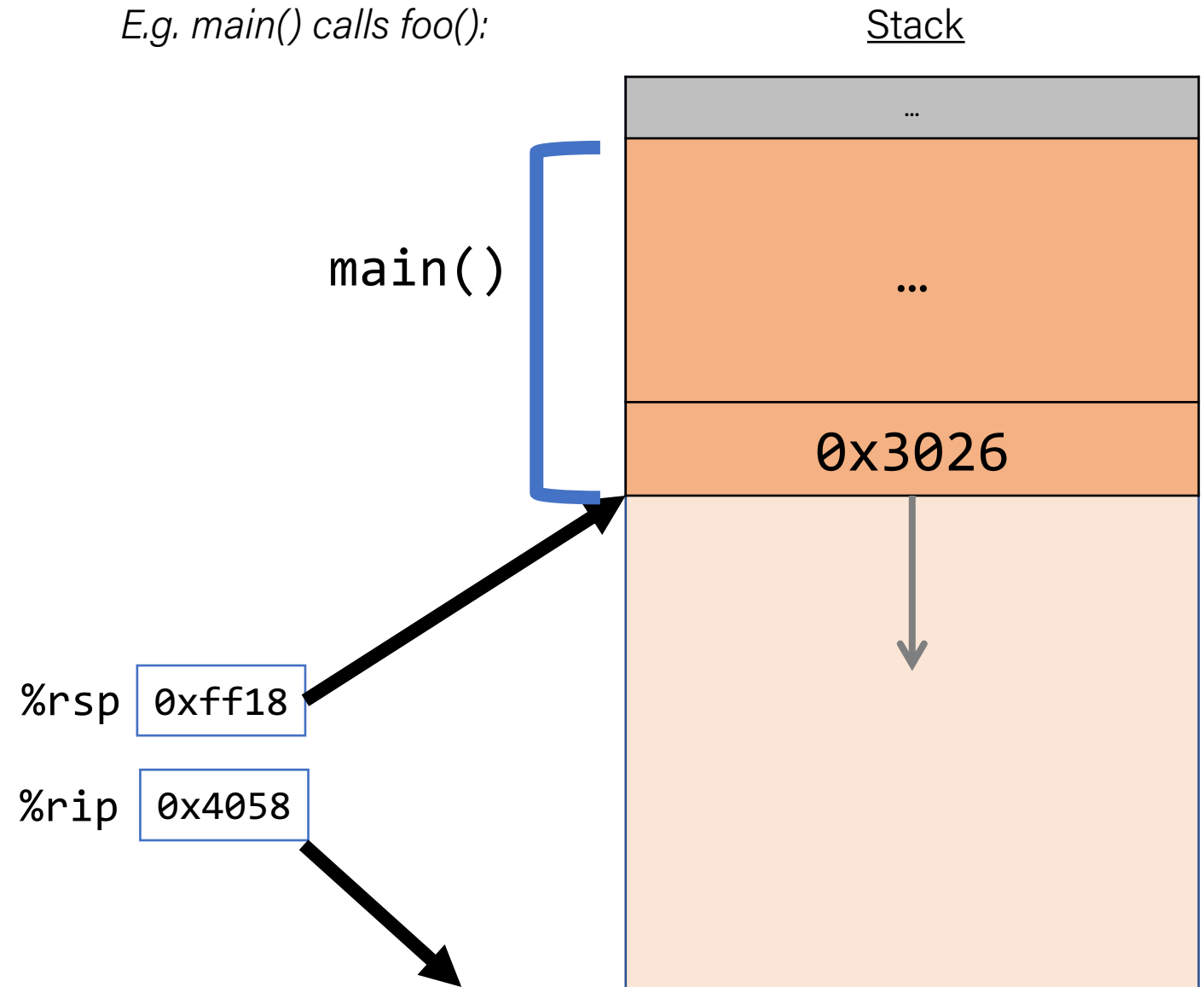
Solution: push the next value of `%rip` onto the stack. Then call the function. When it is finished, put this value back into `%rip` and continue executing.



Remembering Where We Left Off

Problem: `%rip` points to the next instruction to execute. To call a function, we must remember the *next* caller instruction to resume at after.

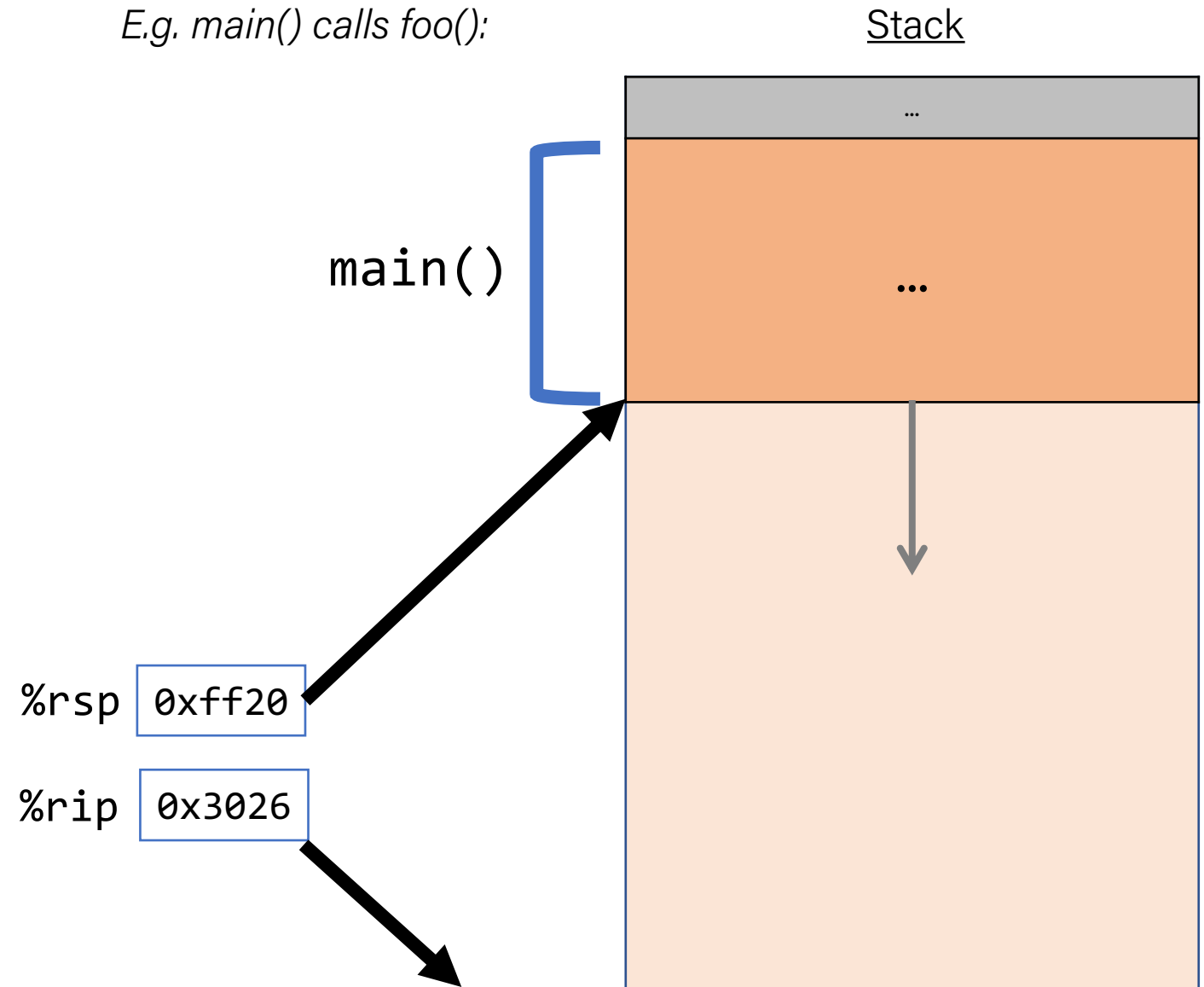
Solution: push the next value of `%rip` onto the stack. Then call the function. When it is finished, put this value back into `%rip` and continue executing.



Remembering Where We Left Off

Problem: `%rip` points to the next instruction to execute. To call a function, we must remember the *next* caller instruction to resume at after.

Solution: push the next value of `%rip` onto the stack. Then call the function. When it is finished, put this value back into `%rip` and continue executing.



Example: Remembering Where We Left Off

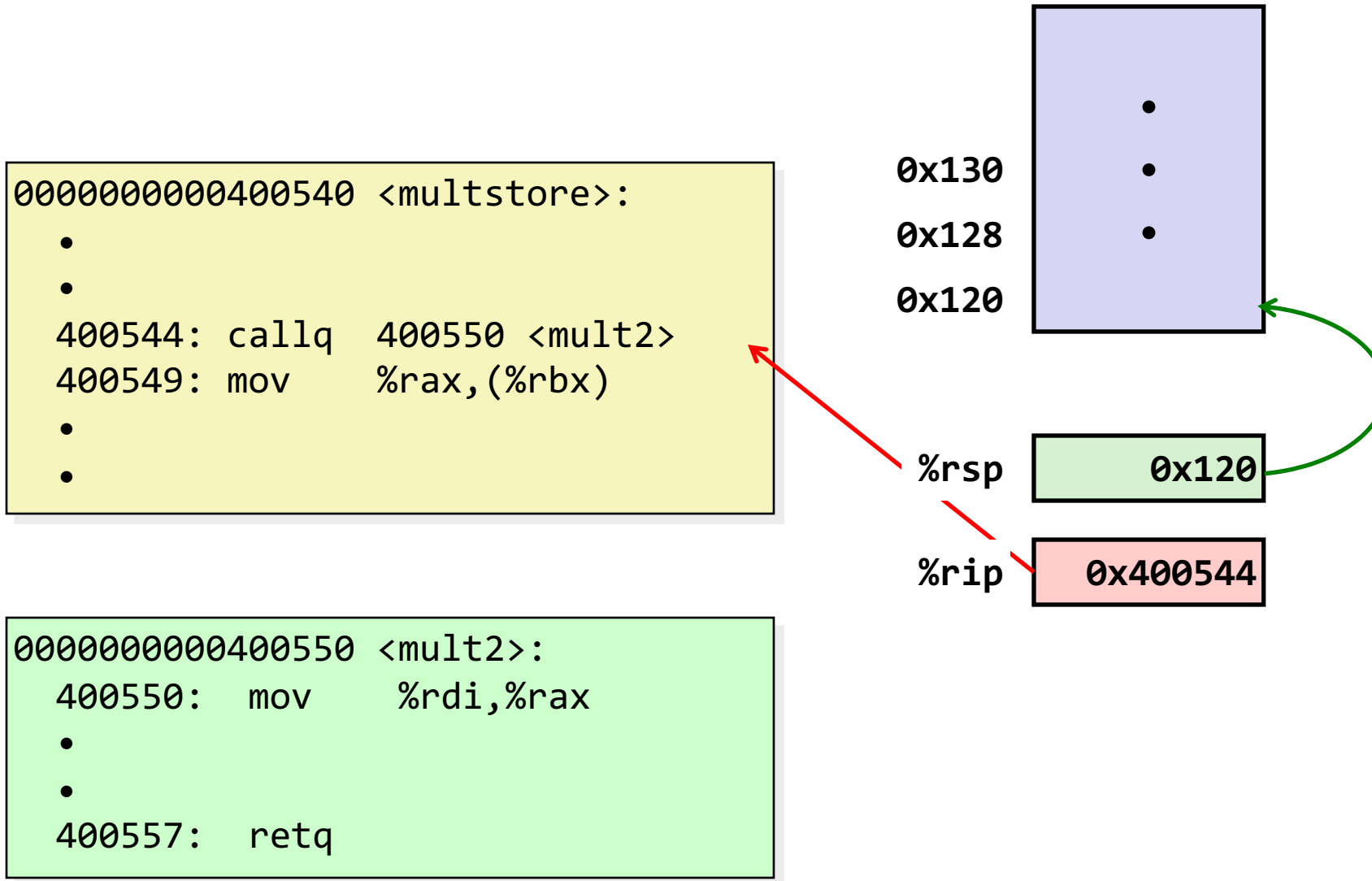
```
void multstore
(long x, long y, long *dest) {
    long t = mult2(x, y);
    *dest = t;
}
```

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

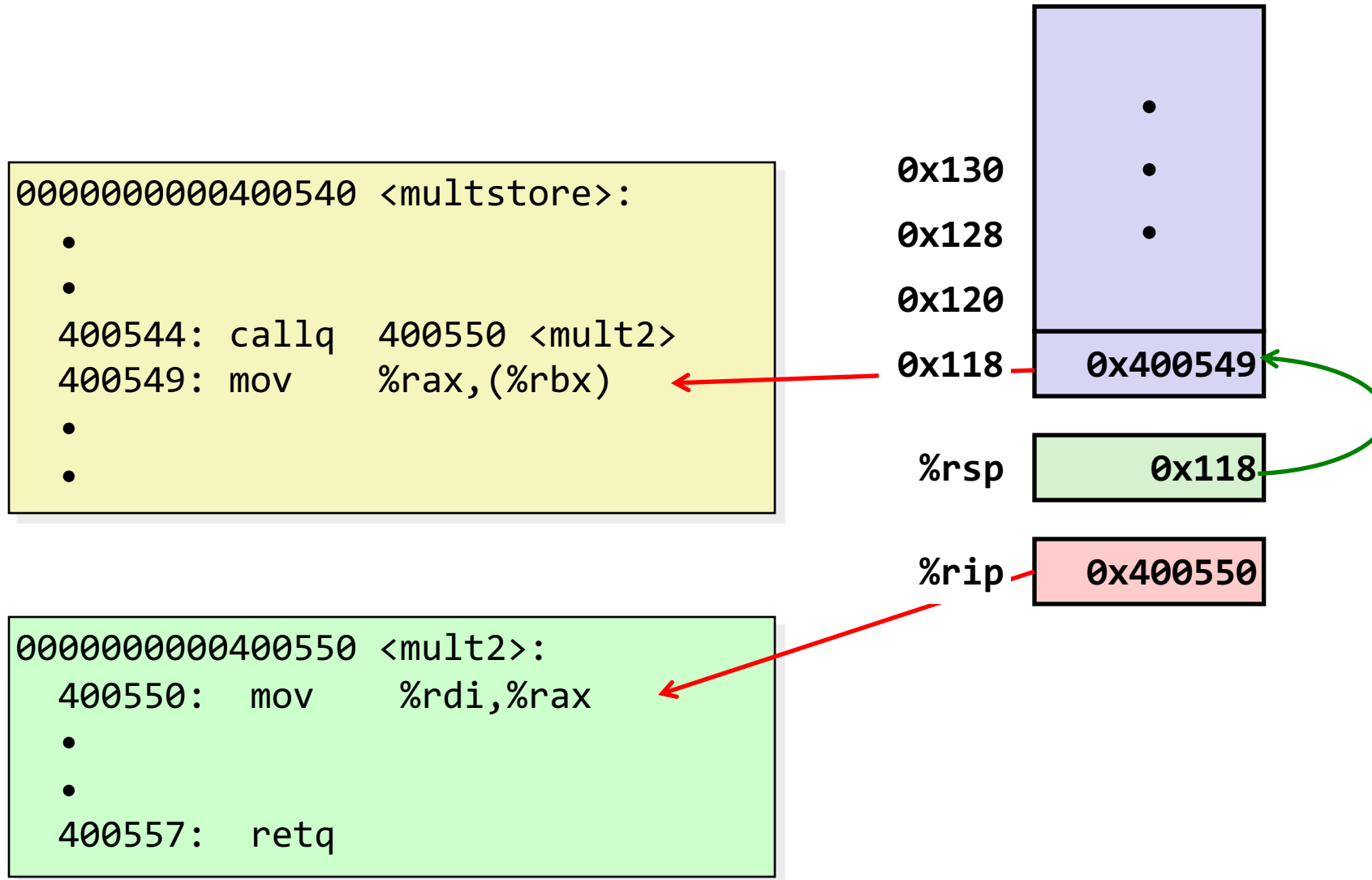
```
long mult2
(long a, long b)
{
    long s = a * b;
    return s;
}
```

```
0000000000400550 <mult2>:
400550: mov     %rdi,%rax            # a
400553: imul    %rsi,%rax            # a * b
400557: retq                          # Return
```

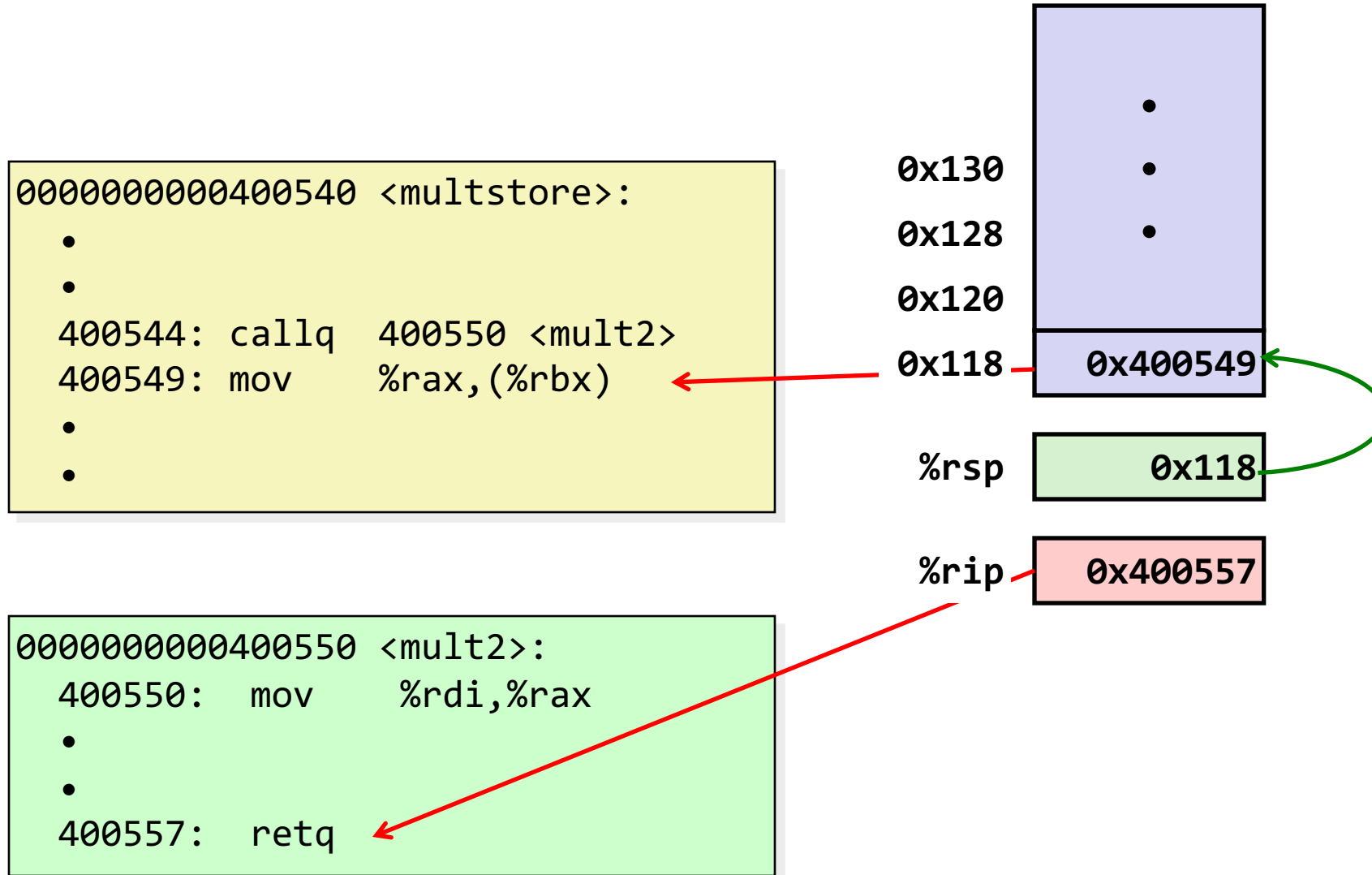
Example: Remembering Where We Left Off



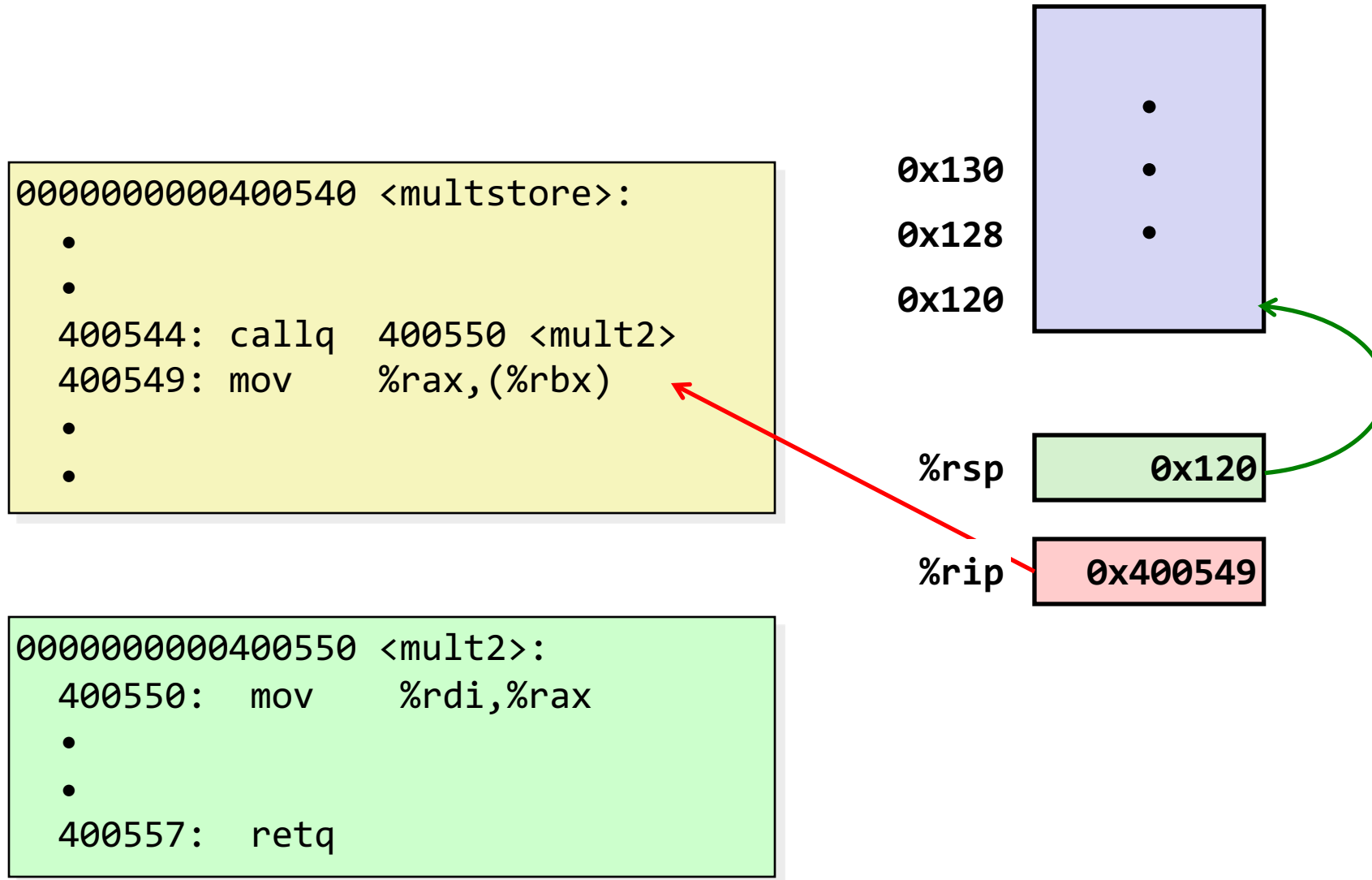
Example: Remembering Where We Left Off



Example: Remembering Where We Left Off



Example: Remembering Where We Left Off



Call And Return

The **call** instruction pushes the address of the instruction immediately following the **call** instruction onto the stack and sets `%rip` to point to the beginning of the specified function's instructions.

call Label

call *Operand

The **ret** instruction pops this instruction address from the stack and stores it in `%rip`.

ret

The stored `%rip` value for a function is called its **return address**. It is the address of the instruction at which to resume the function's execution. (not to be confused with **return value**, which is the value returned from a function).

What's left? Calling Functions In Assembly

To call a function in assembly, we must do a few things:

- **Pass Control** – %rip must be adjusted to execute the function being called and then resume the caller function afterwards.
- **Pass Data** – we must pass any parameters and receive any return value.
- **Manage Memory** – we must handle any space needs of the callee on the stack.

Terminology: **caller** function calls the **callee** function.

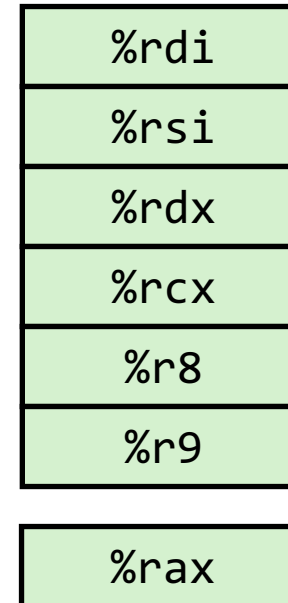
Lecture Plan

- Revisiting `%rip`
- Calling Functions
 - The Stack
 - Passing Control
 - Passing Data
 - Local Storage
- Register Restrictions
- Pulling it all together: recursion example

Parameters and Return

- There are special registers that store parameters and the return value.
- To call a function, we must put any parameters we are passing into the correct registers. (`%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, `%r9`, in that order)
- Parameters beyond the first 6 are put on the stack.
- If the caller expects a return value, it looks in `%rax` after the callee completes.

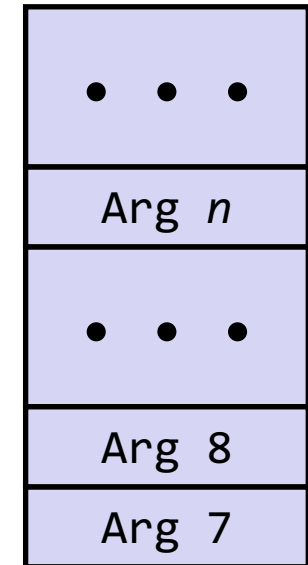
Registers



First 6 arguments

Return value

Stack



Only allocate stack space when needed

Example 1: Parameters and Return

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


```
0000000000400540 <multstore>:
    # x in %rdi, y in %rsi, dest in %rdx
    . . .
400541: mov     %rdx,%rbx          # Save dest
400544: callq   400550 <mult2>      # mult2(x,y)
    # t in %rax
400549: mov     %rax,(%rbx)         # Save at dest
    . . .
```

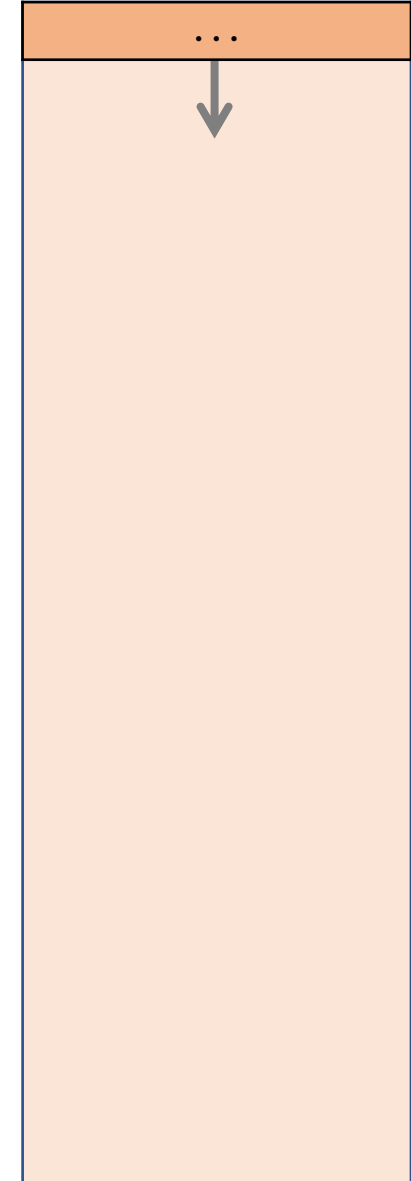
```
long mult2
(long a, long b)
{
    long s = a * b;
    return s;
}
```

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

Example 2: Parameters and Return

```
int main(int argc, char *argv[]) {  
    int i1 = 1;  
    int i2 = 2;  
    int i3 = 3;  
    int i4 = 4;  
    int result = func(&i1, &i2, &i3, &i4,  
                     i1, i2, i3, i4);  
    ...  
}  
  
int func(int *p1, int *p2, int *p3, int *p4,  
         int v1, int v2, int v3, int v4) {  
    ...  
}
```


main() 

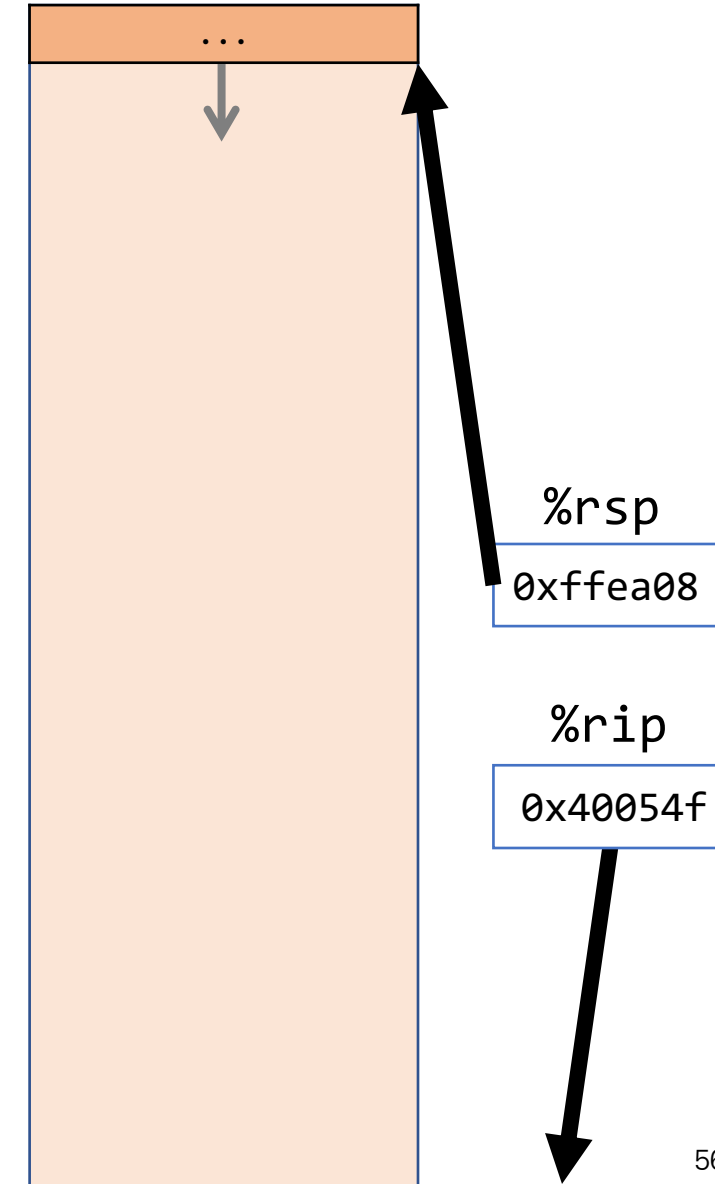


Example 2: Parameters and Return

```
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4,
                     i1, i2, i3, i4);
    ...
}

int func(int *p1, int *p2, int *p3, int *p4,
        int v1, int v2, int v3, int v4) {
    ...
}
```

main() 



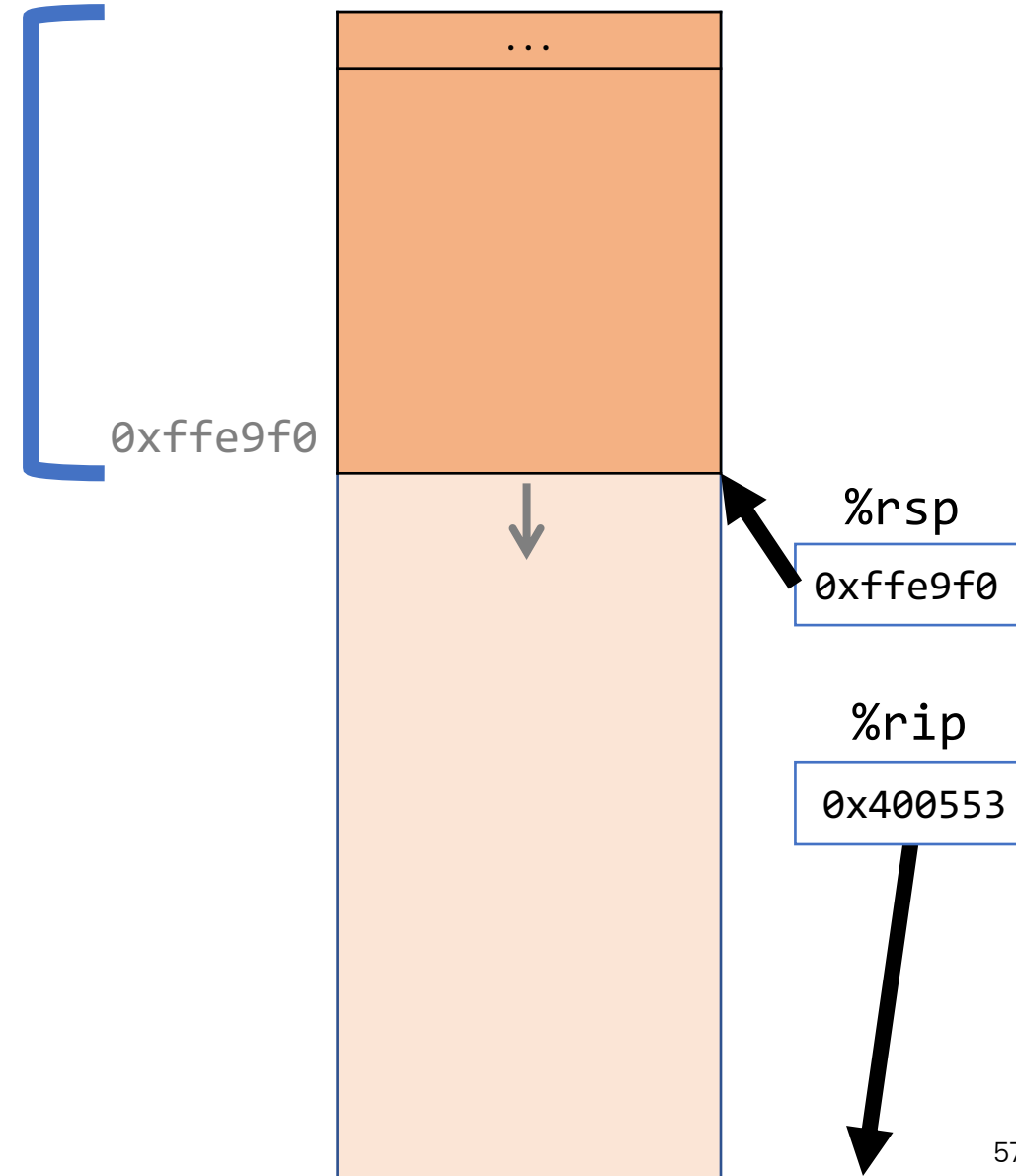
```
0x40054f <+0>:    sub    $0x18,%rsp
0x400553 <+4>:    movl   $0x1,0xc(%rsp)
0x40055b <+12>:   movl   $0x2,0x8(%rsp)
0x400563 <+20>:   movl   $0x3,0x4(%rsp)
0x40056b <+28>:   movl   $0x4,0x0(%rsp)
```

Example 2: Parameters and Return

```
int main(int argc, char *argv[]) {  
    int i1 = 1;  
    int i2 = 2;  
    int i3 = 3;  
    int i4 = 4;  
    int result = func(&i1, &i2, &i3, &i4,  
                     i1, i2, i3, i4);  
    ...  
}  
  
int func(int *p1, int *p2, int *p3, int *p4,  
         int v1, int v2, int v3, int v4) {  
    ...  
}
```

```
0x40054f <+0>:    sub    $0x18,%rsp  
0x400553 <+4>:    movl   $0x1,0xc(%rsp)  
0x40055b <+12>:   movl   $0x2,0x8(%rsp)  
0x400563 <+20>:   movl   $0x3,0x4(%rsp)  
0x40056b <+28>:   movl   $0x4,0x0(%rsp)
```

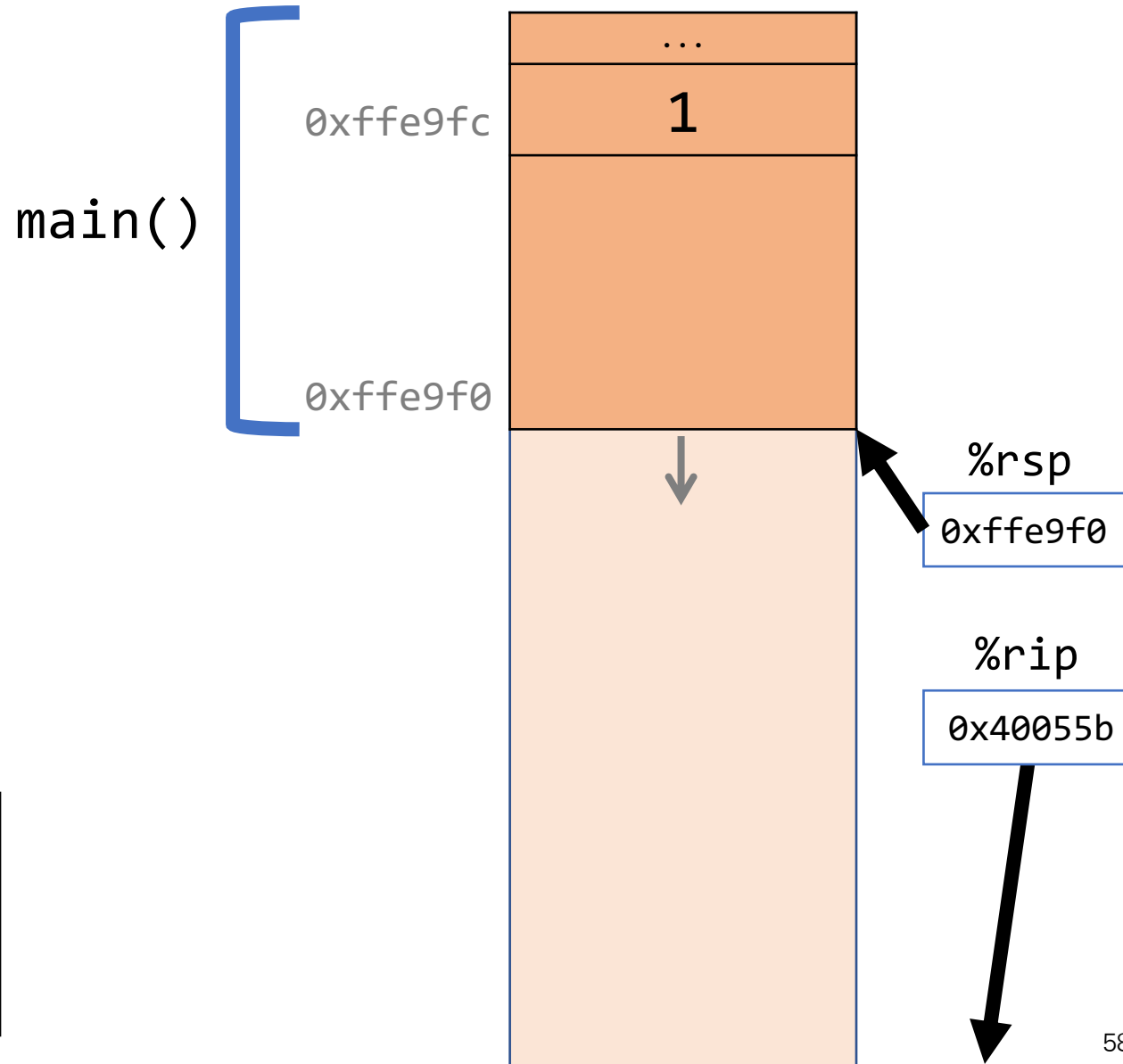
main()



Example 2: Parameters and Return

```
int main(int argc, char *argv[]) {  
    int i1 = 1;  
    int i2 = 2;  
    int i3 = 3;  
    int i4 = 4;  
    int result = func(&i1, &i2, &i3, &i4,  
                     i1, i2, i3, i4);  
    ...  
}  
  
int func(int *p1, int *p2, int *p3, int *p4,  
         int v1, int v2, int v3, int v4) {  
    ...  
}
```

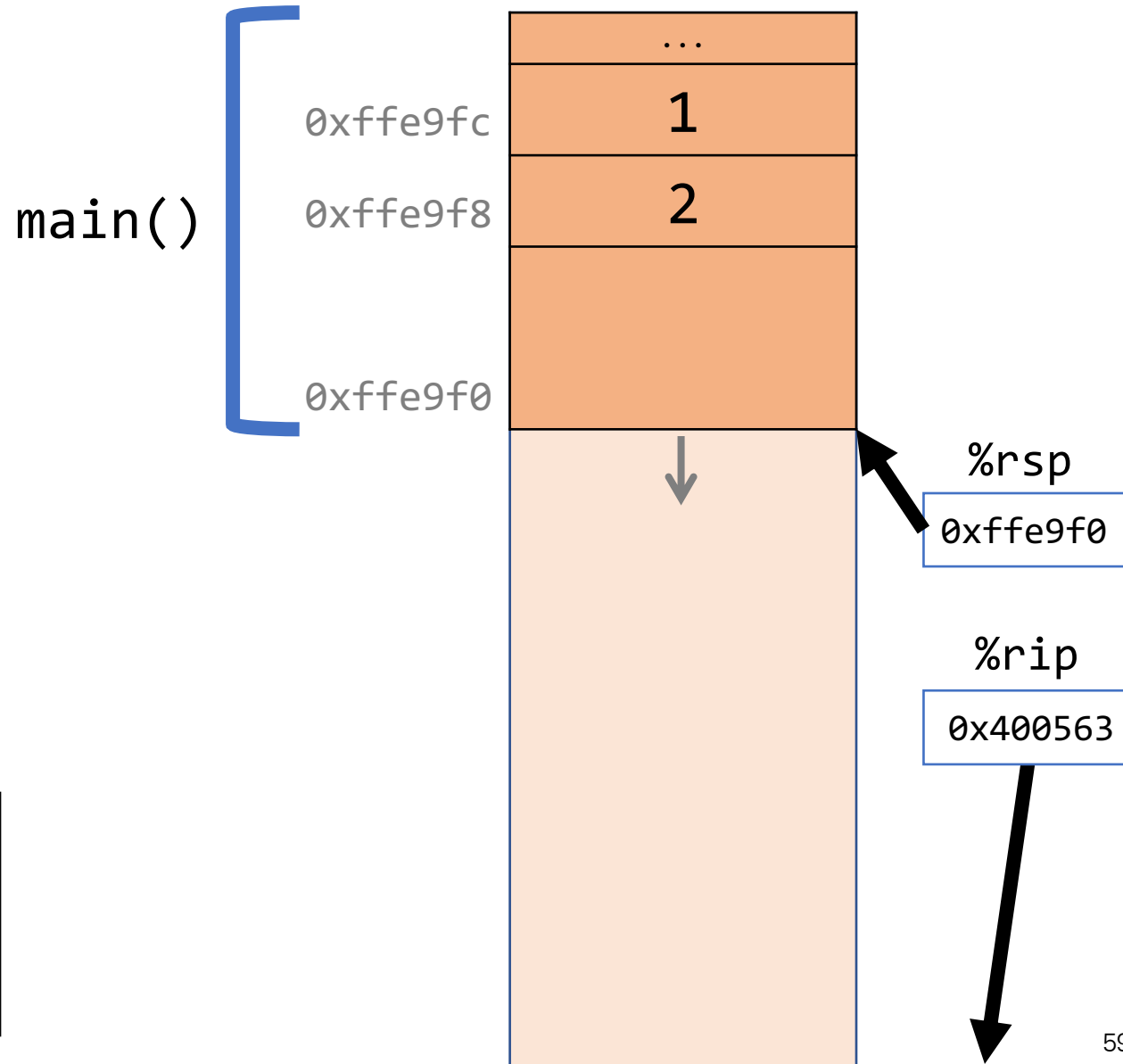
```
0x40054f <+0>:    sub    $0x18,%rsp  
0x400553 <+4>:    movl    $0x1,0xc(%rsp)  
0x40055b <+12>:   movl    $0x2,0x8(%rsp)  
0x400563 <+20>:   movl    $0x3,0x4(%rsp)  
0x40056b <+28>:   movl    $0x4,0x0(%rsp)
```



Example 2: Parameters and Return

```
int main(int argc, char *argv[]) {  
    int i1 = 1;  
    int i2 = 2;  
    int i3 = 3;  
    int i4 = 4;  
    int result = func(&i1, &i2, &i3, &i4,  
                     i1, i2, i3, i4);  
    ...  
}  
  
int func(int *p1, int *p2, int *p3, int *p4,  
        int v1, int v2, int v3, int v4) {  
    ...  
}
```

```
0x40054f <+0>:    sub    $0x18,%rsp  
0x400553 <+4>:    movl   $0x1,0xc(%rsp)  
0x40055b <+12>:    movl   $0x2,0x8(%rsp)  
0x400563 <+20>:    movl   $0x3,0x4(%rsp)  
0x40056b <+28>:    movl   $0x4,0x0(%rsp)
```

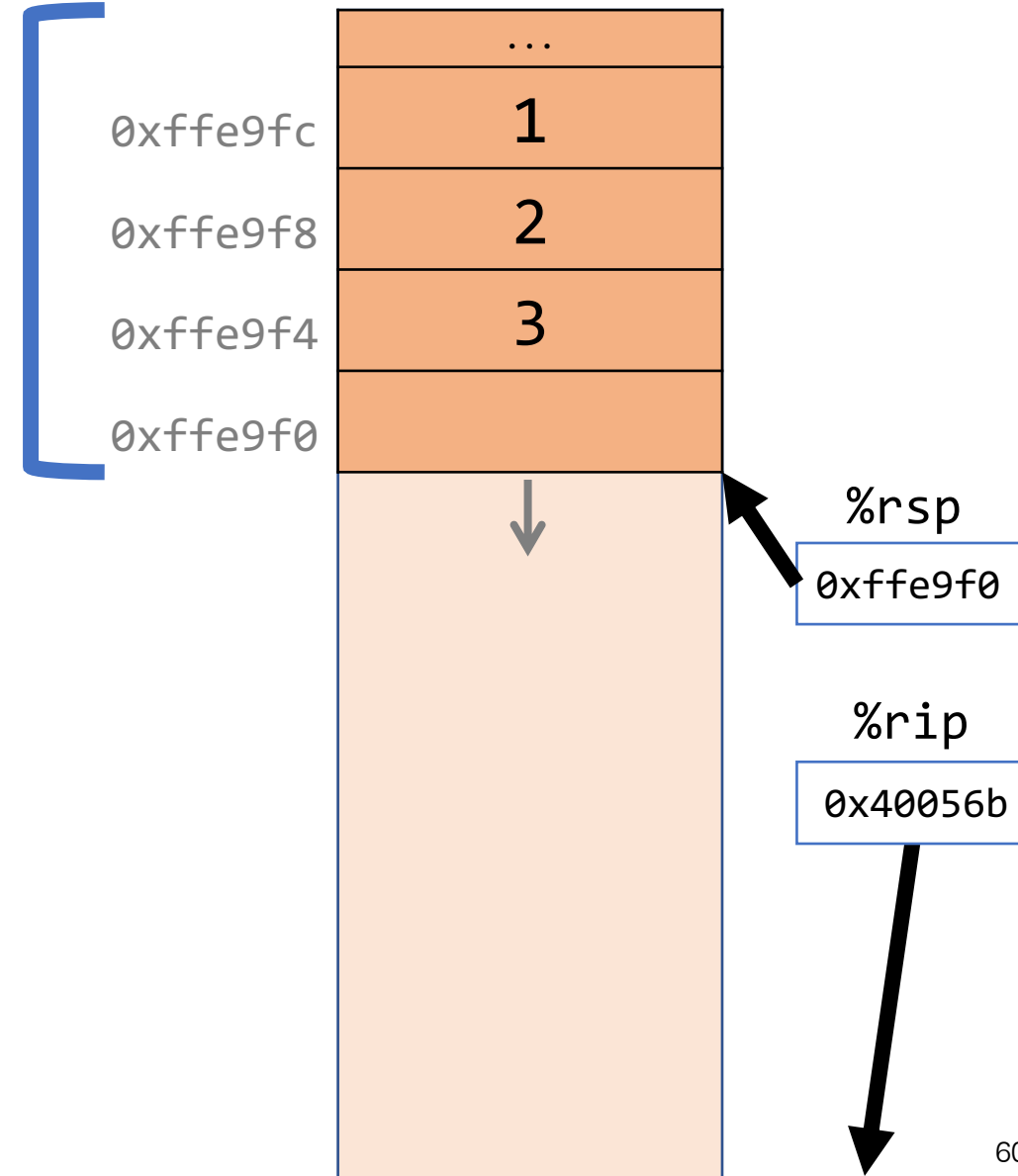


Example 2: Parameters and Return

```
int main(int argc, char *argv[]) {  
    int i1 = 1;  
    int i2 = 2;  
    int i3 = 3;  
    int i4 = 4;  
    int result = func(&i1, &i2, &i3, &i4,  
                     i1, i2, i3, i4);  
    ...  
}  
  
int func(int *p1, int *p2, int *p3, int *p4,  
        int v1, int v2, int v3, int v4) {  
    ...  
}
```

```
0x400553 <+4>:    movl    $0x1,0xc(%rsp)  
0x40055b <+12>:   movl    $0x2,0x8(%rsp)  
0x400563 <+20>:   movl    $0x3,0x4(%rsp)  
0x40056b <+28>:   movl    $0x4,(%rsp)  
0x400572 <+35>:   pusha   $0x4
```

main()

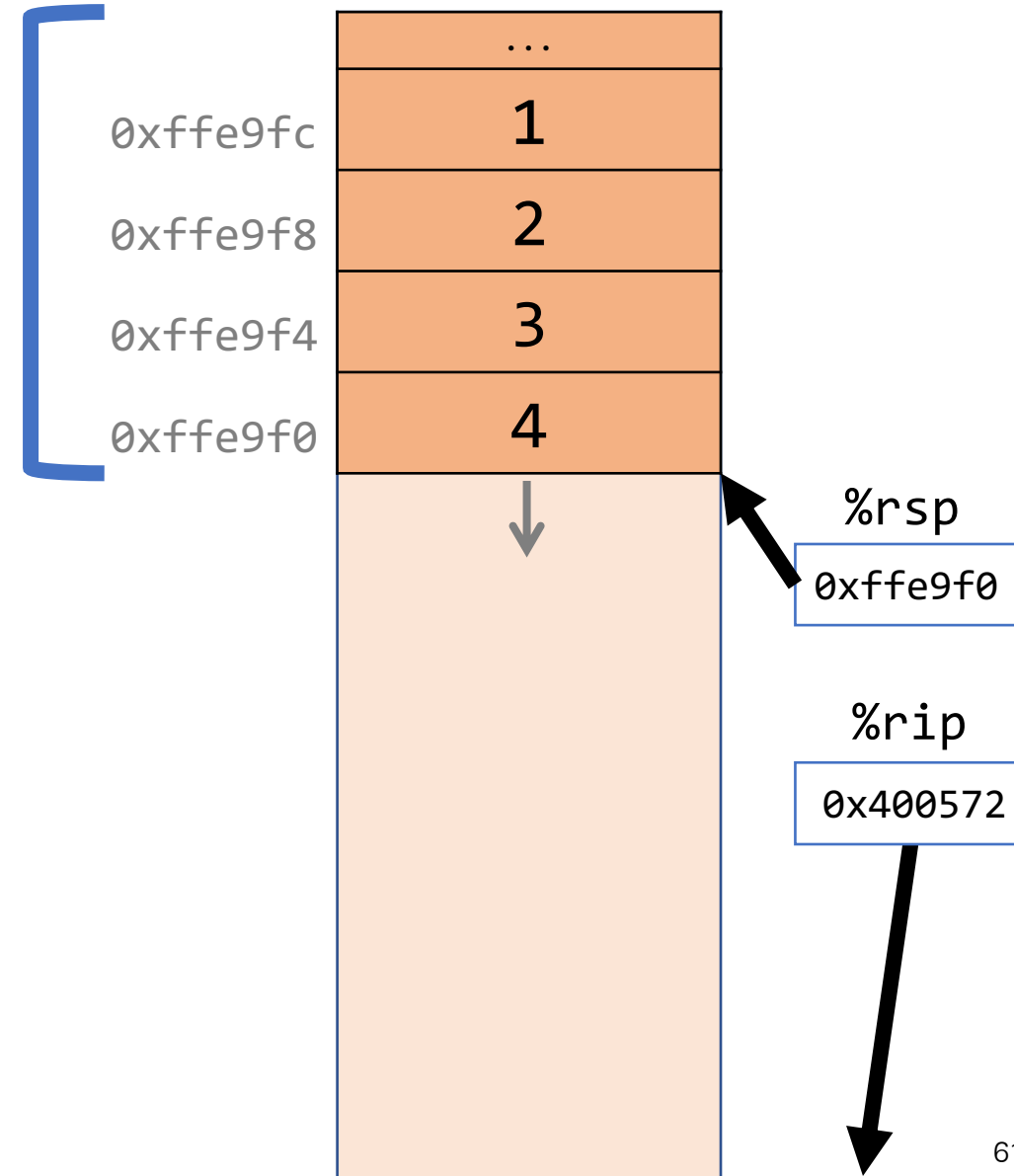


Example 2: Parameters and Return

```
int main(int argc, char *argv[]) {  
    int i1 = 1;  
    int i2 = 2;  
    int i3 = 3;  
    int i4 = 4;  
    int result = func(&i1, &i2, &i3, &i4,  
                     i1, i2, i3, i4);  
    ...  
}  
  
int func(int *p1, int *p2, int *p3, int *p4,  
        int v1, int v2, int v3, int v4) {  
    ...  
}
```

```
0x40055b <+12>:    movl    $0x2,0x8(%rsp)  
0x400563 <+20>:    movl    $0x3,0x4(%rsp)  
0x40056b <+28>:    movl    $0x4, (%rsp)  
0x400572 <+35>:    pushq   $0x4  
0x400574 <+37>:    pushq   $0x2
```

main()

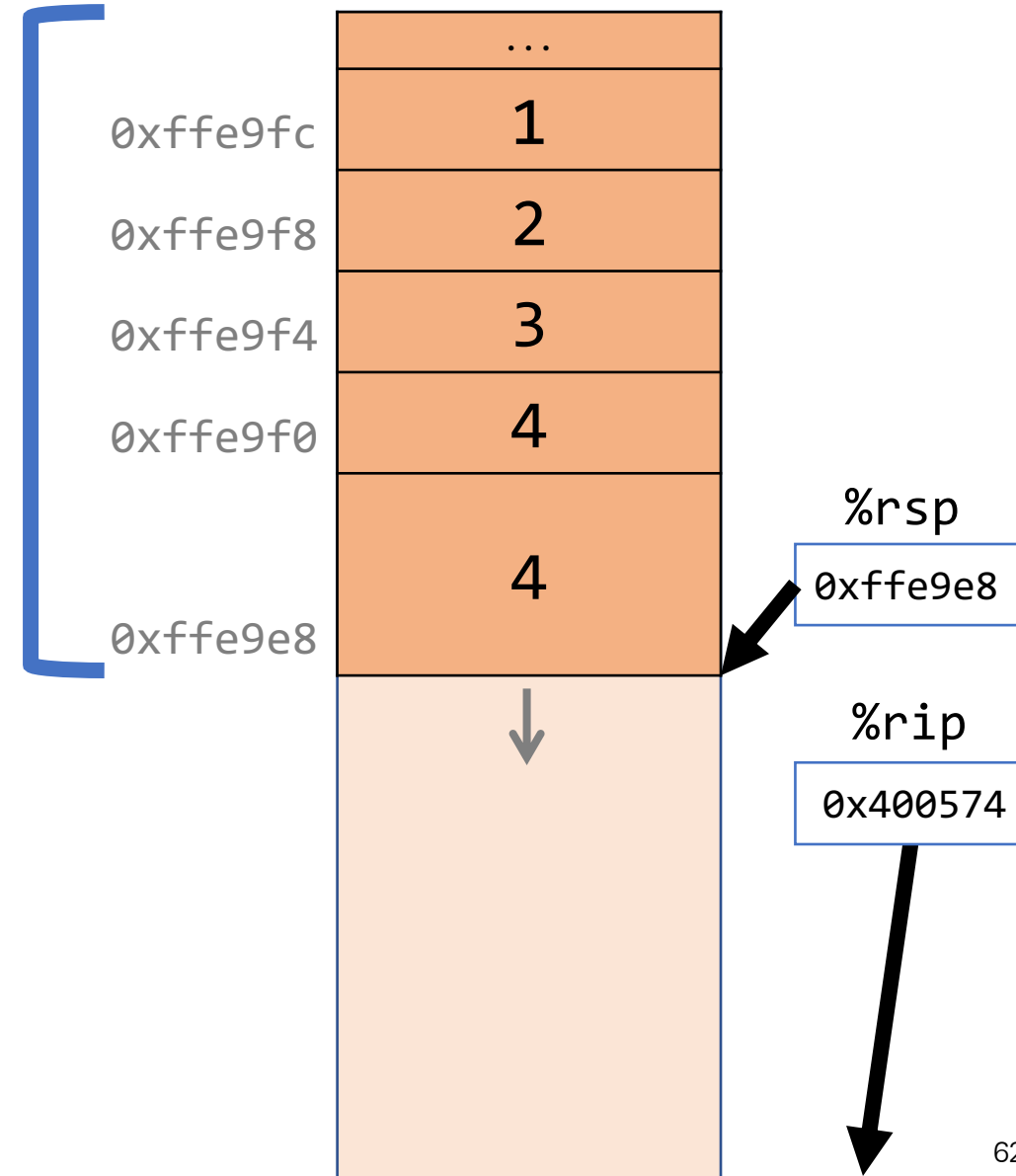


Example 2: Parameters and Return

```
int main(int argc, char *argv[]) {  
    int i1 = 1;  
    int i2 = 2;  
    int i3 = 3;  
    int i4 = 4;  
    int result = func(&i1, &i2, &i3, &i4,  
                    i1, i2, i3, i4);  
    ...  
}  
  
int func(int *p1, int *p2, int *p3, int *p4,  
        int v1, int v2, int v3, int v4) {  
    ...  
}
```

```
0x400563 <+20>:    movl    $0x3,0x4(%rsp)  
0x40056b <+28>:    movl    $0x4,(%rsp)  
0x400572 <+35>:    pushq   $0x4  
0x400574 <+37>:    pushq   $0x3  
0x400576 <+39>:    mov     $0x2,%rax
```

main()

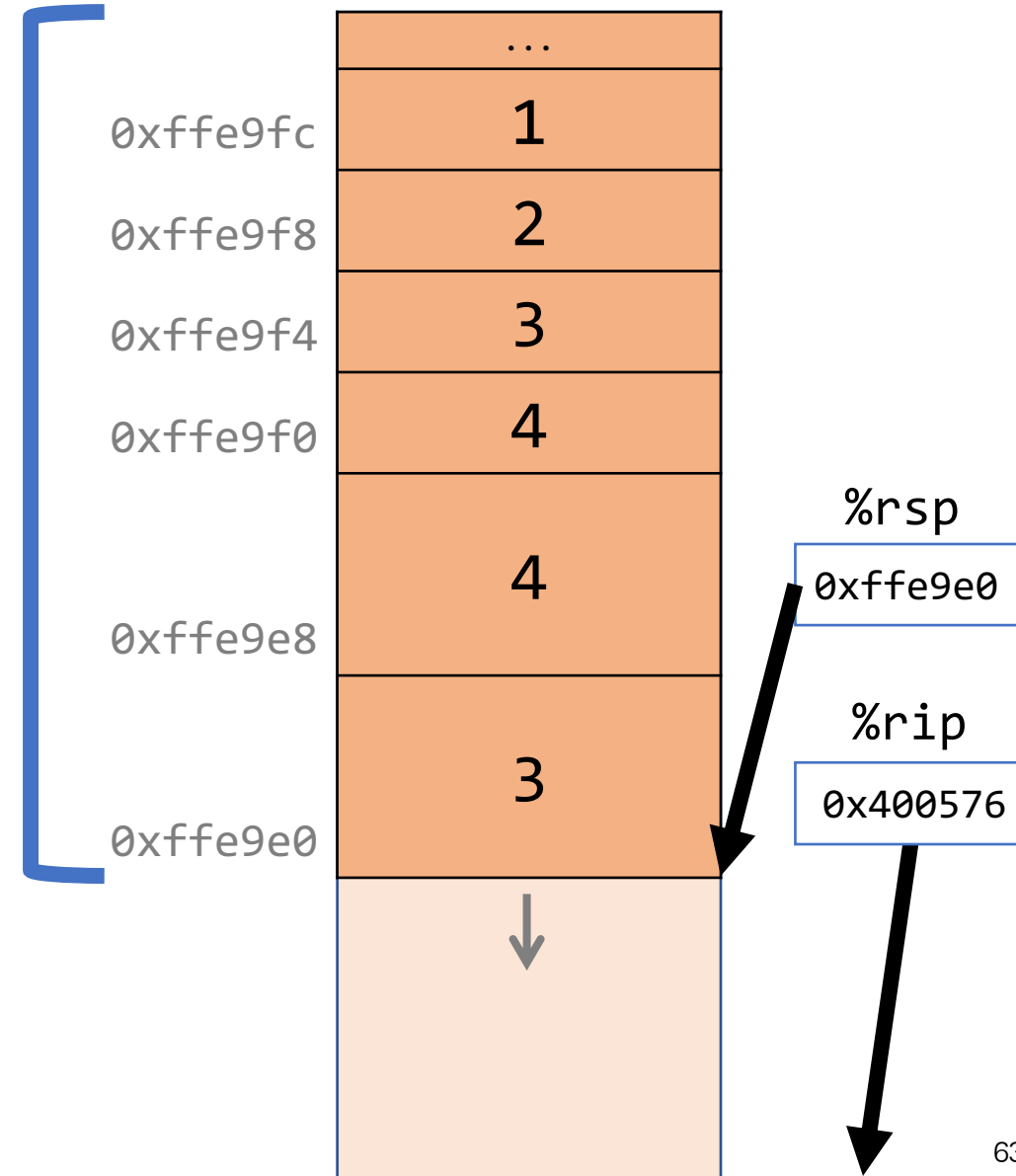


Example 2: Parameters and Return

```
int main(int argc, char *argv[]) {  
    int i1 = 1;  
    int i2 = 2;  
    int i3 = 3;  
    int i4 = 4;  
    int result = func(&i1, &i2, &i3, &i4,  
                    i1, i2, i3, i4);  
    ...  
}  
  
int func(int *p1, int *p2, int *p3, int *p4,  
        int v1, int v2, int v3, int v4) {  
    ...  
}
```

```
0x40056b <+28>: movl    $0x4, (%rsp)  
0x400572 <+35>: pushq   $0x4  
0x400574 <+37>: pushq   $0x3  
0x400576 <+39>: mov     $0x2, %r9d  
0x40057c <+45>: mov     $0x1, %r8d
```

main()

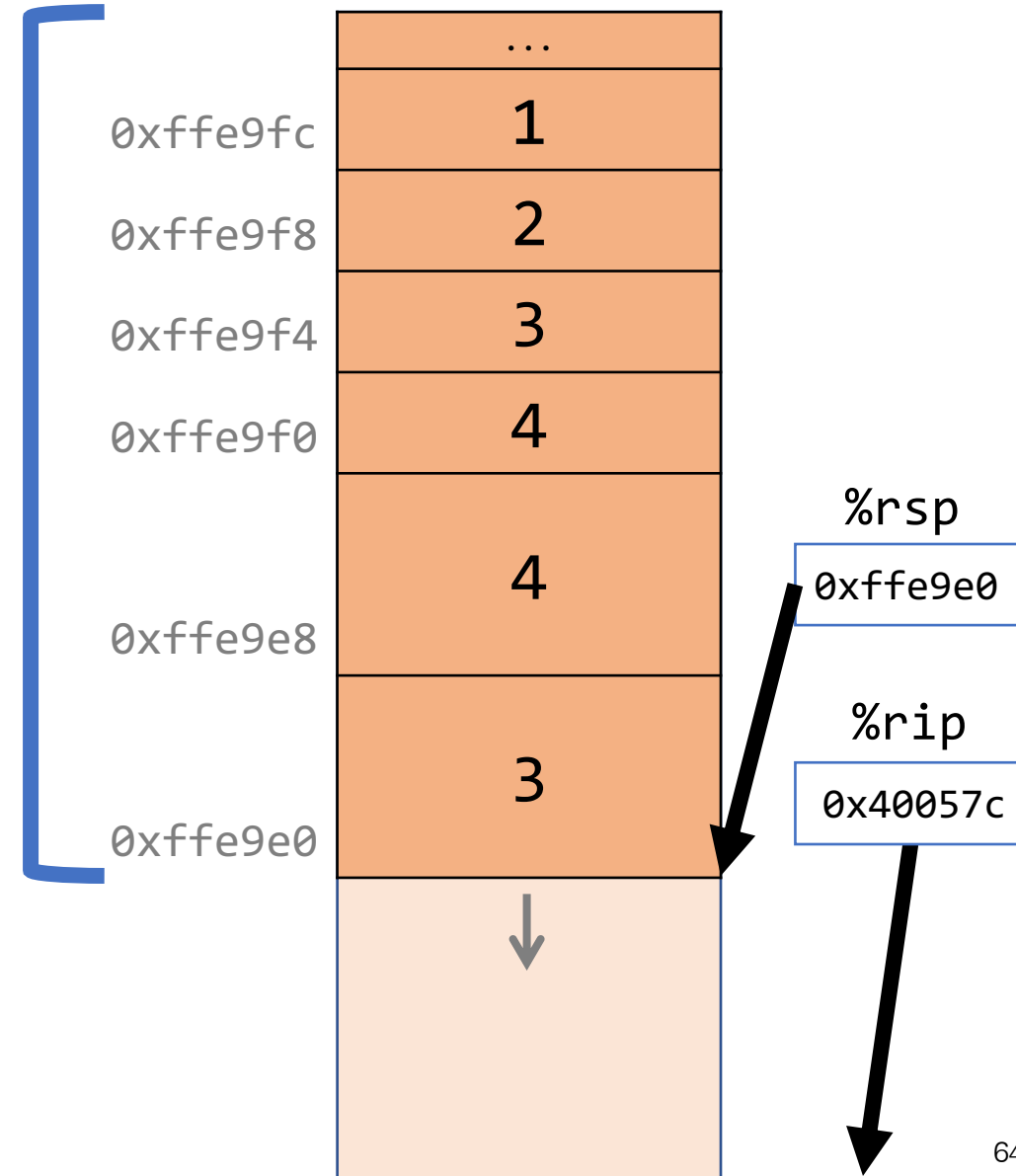


Example 2: Parameters and Return

```
int main(int argc, char *argv[]) {  
    int i1 = 1;  
    int i2 = 2;  
    int i3 = 3;  
    int i4 = 4;  
    int result = func(&i1, &i2, &i3, &i4,  
                    i1, i2, i3, i4);  
    ...  
}  
  
int func(int *p1, int *p2, int *p3, int *p4,  
        int v1, int v2, int v3, int v4) {  
    ...  
}
```

```
0x400572 <+35>:    pushq    $0x4  
0x400574 <+37>:    pushq    $0x3  
0x400576 <+39>:    mov     $0x2,%r9d  
0x40057c <+45>:    mov     $0x1,%r8d  
0x400582 <+51>:    leaq    0x10(%rsp),%rcx
```

main()



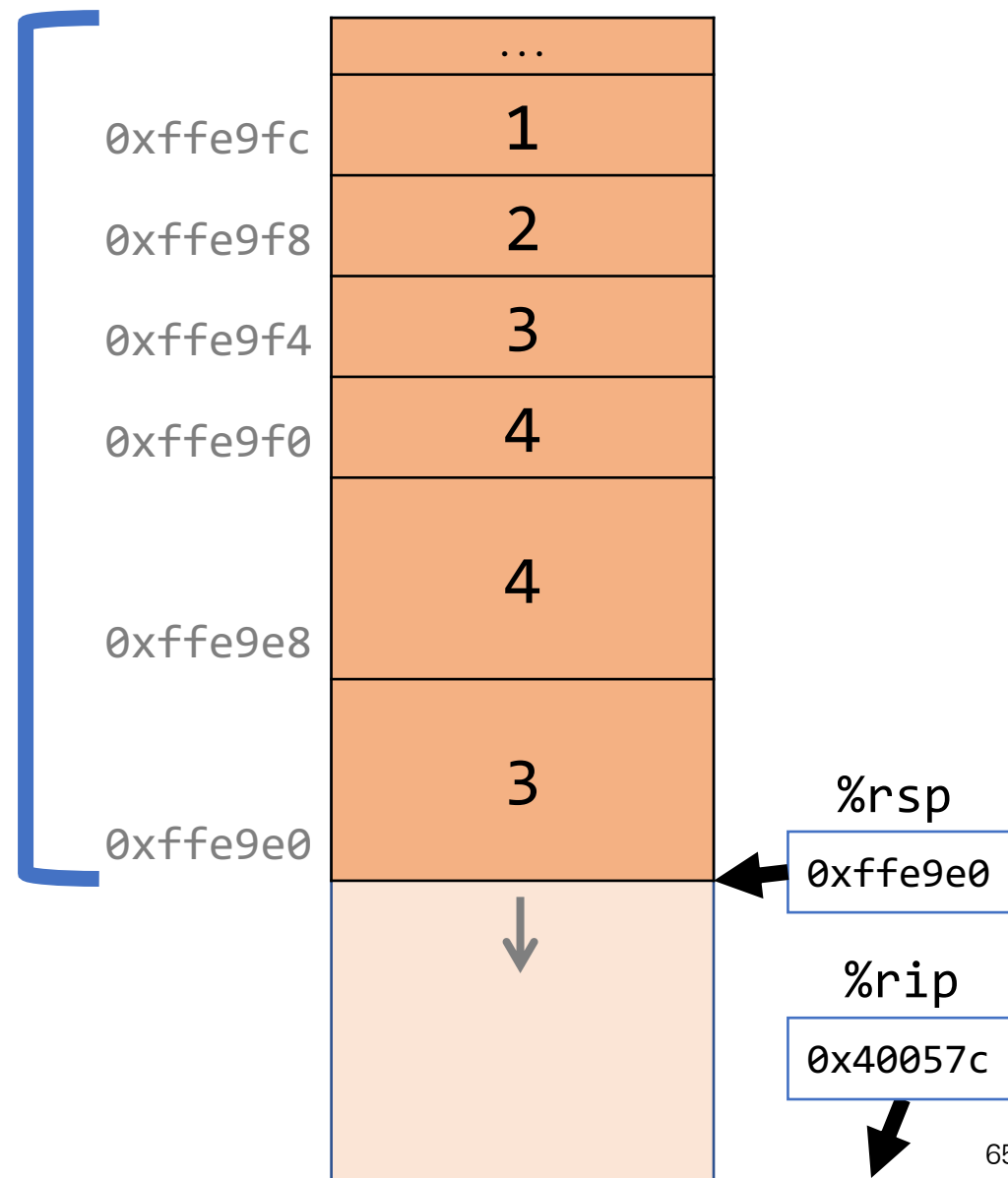
Example 2: Parameters and Return

```
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4,
                     i1, i2, i3, i4);
    ...
}

int func(int *p1, int *p2, int *p3, int *p4,
        int v1, int v2, int v3, int v4) {
    ...
}
```

```
0x400572 <+35>:    pushq    $0x4
0x400574 <+37>:    pushq    $0x3
0x400576 <+39>:    mov     $0x2,%r9d
0x40057c <+45>:    mov     $0x1,%r8d
0x400582 <+51>:    leaq    0x10(%rsp),%rcx
```

main()



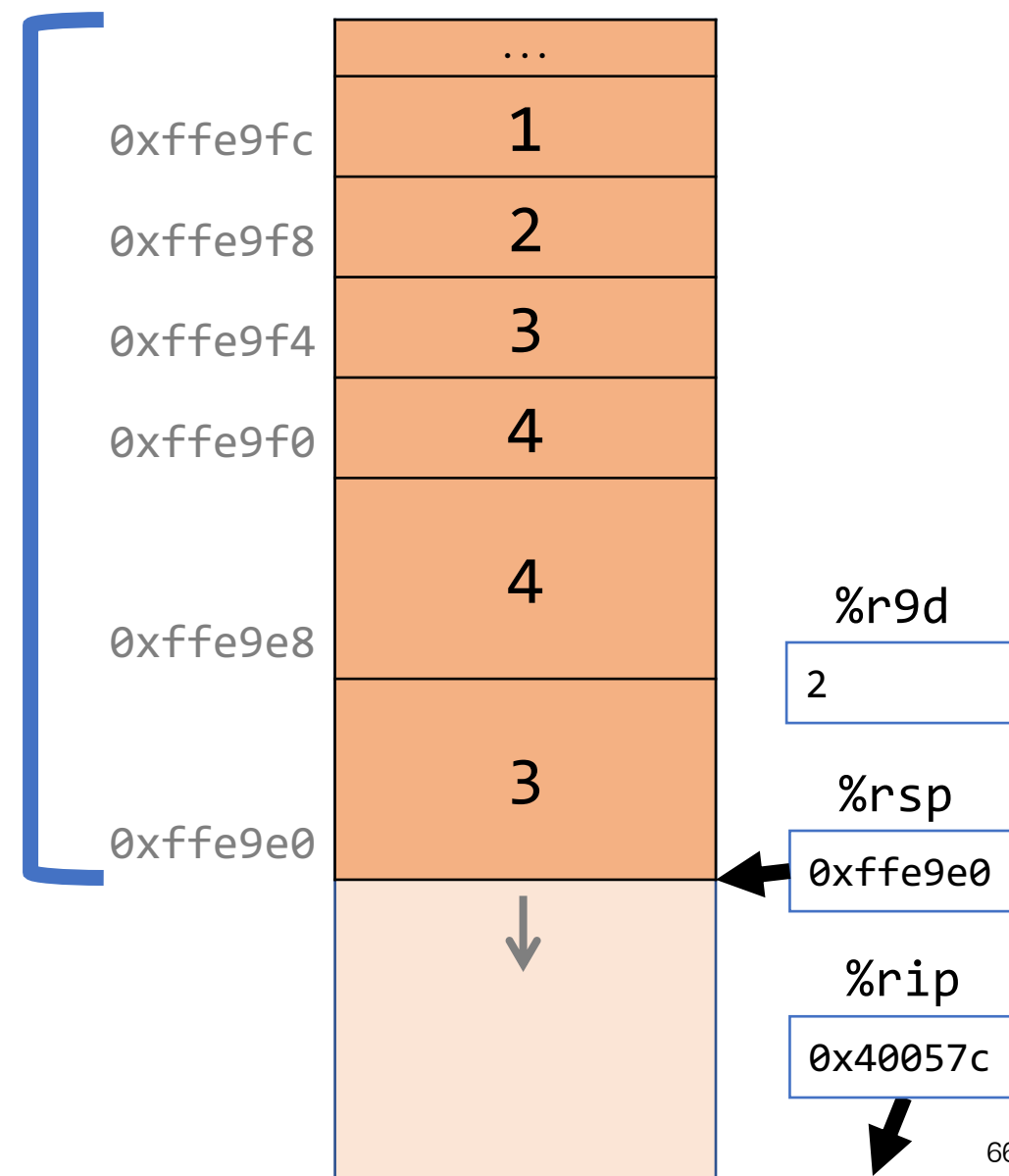
Example 2: Parameters and Return

```
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4,
                     i1, i2, i3, i4);
    ...
}

int func(int *p1, int *p2, int *p3, int *p4,
        int v1, int v2, int v3, int v4) {
    ...
}
```

```
0x400572 <+35>:    pushq    $0x4
0x400574 <+37>:    pushq    $0x3
0x400576 <+39>:    mov     $0x2,%r9d
0x40057c <+45>:    mov     $0x1,%r8d
0x400582 <+51>:    leaq    0x10(%rsp),%rcx
```

main()



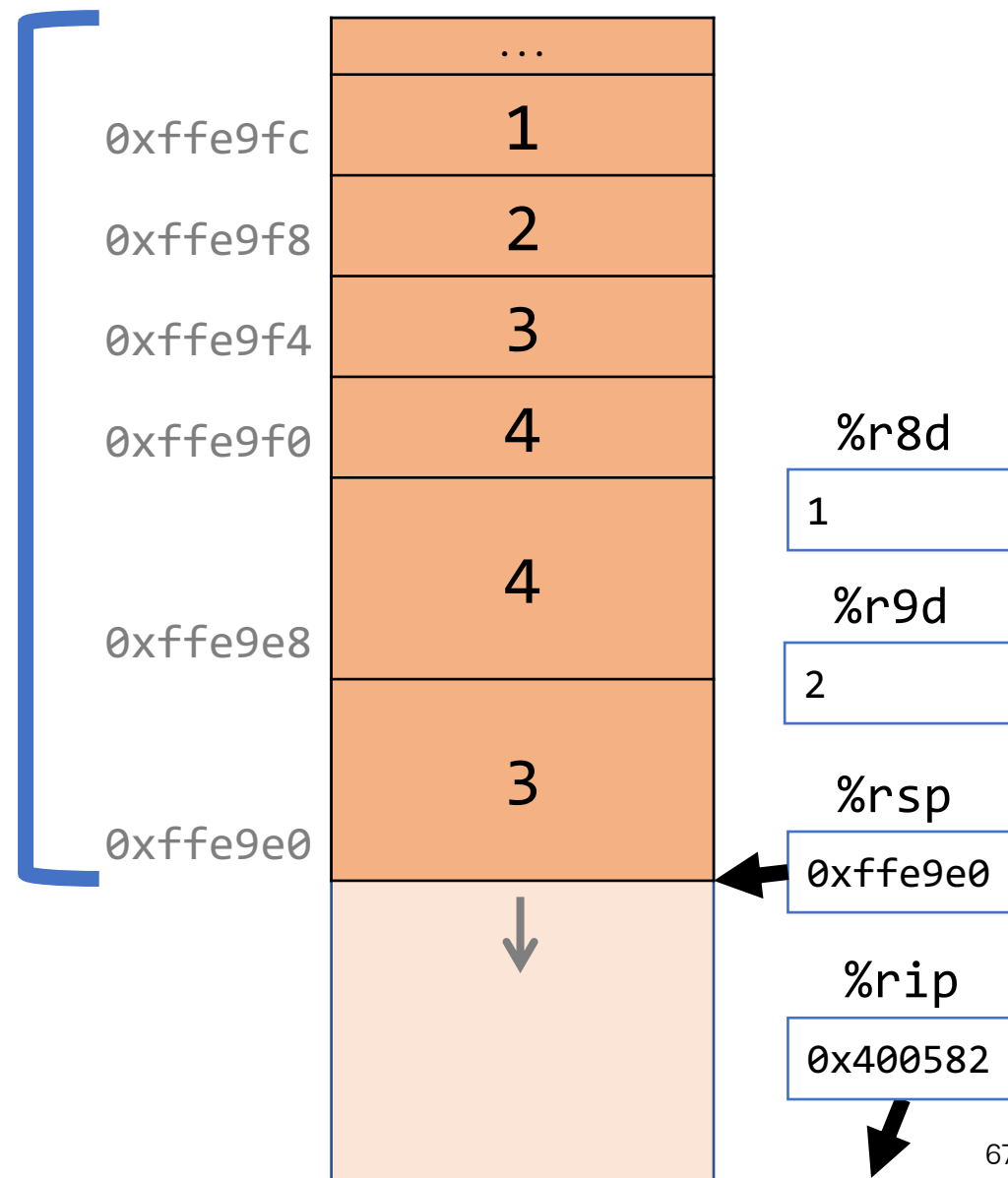
Example 2: Parameters and Return

```
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4,
                     i1, i2, i3, i4);
    ...
}

int func(int *p1, int *p2, int *p3, int *p4,
        int v1, int v2, int v3, int v4) {
    ...
}
```

```
0x400574 <+37>: pushq $0x3
0x400576 <+39>: mov    $0x2,%r9d
0x40057c <+45>: mov    $0x1,%r8d
0x400582 <+51>: lea    0x10(%rsp),%rcx
0x400587 <+56>: lea    0x14(%rsp),%rdx
```

main()



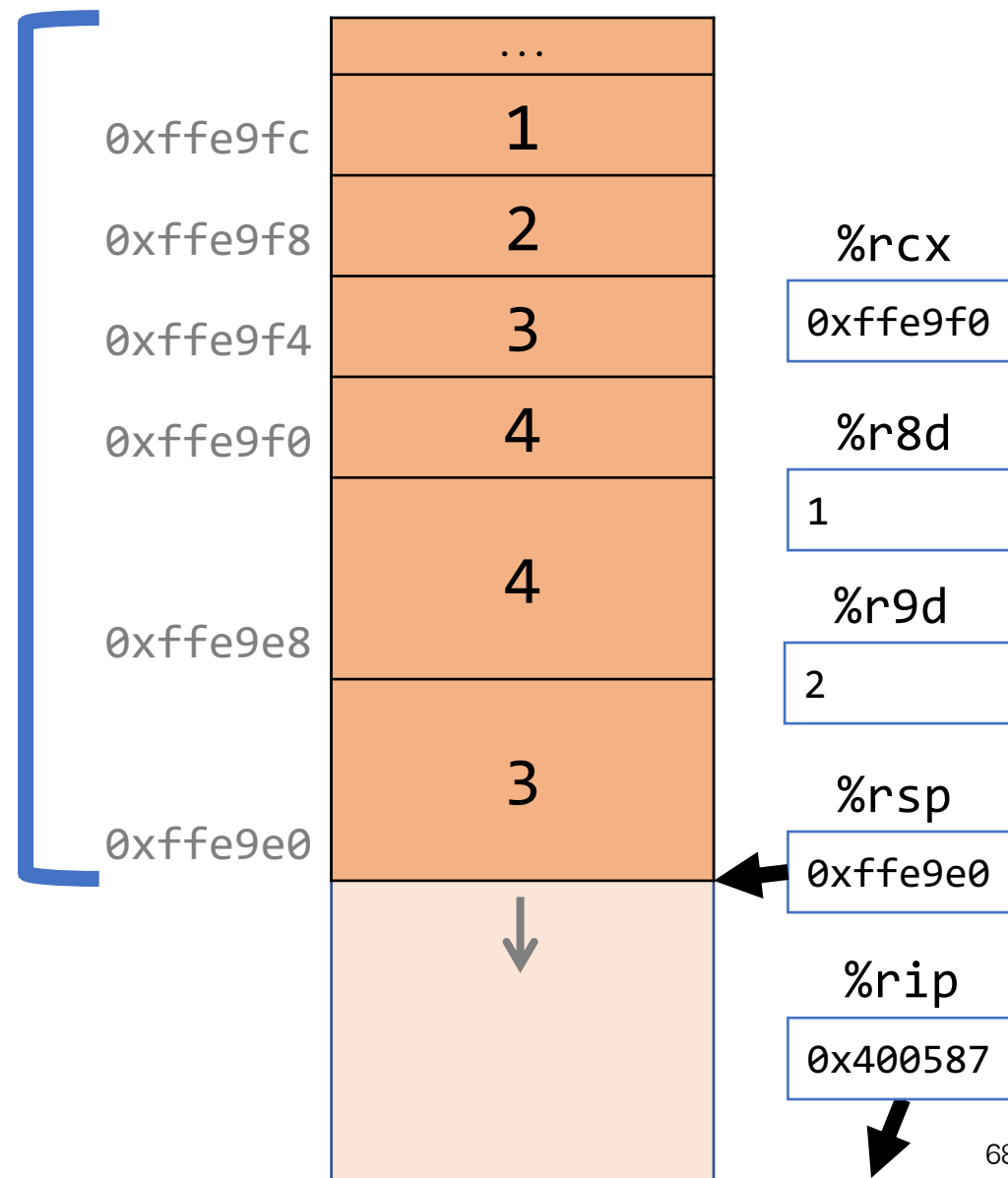
Example 2: Parameters and Return

```
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4,
                     i1, i2, i3, i4);
    ...
}

int func(int *p1, int *p2, int *p3, int *p4,
        int v1, int v2, int v3, int v4) {
    ...
}
```

```
0x400576 <+39>:  mov    $0x2,%r9d
0x40057c <+45>:  mov    $0x1,%r8d
0x400582 <+51>:  lea    0x10(%rsp),%rcx
0x400587 <+56>:  lea    0x14(%rsp),%rdx
0x40058c <+61>:  lea    0x18(%rsp),%rsi
```

main()



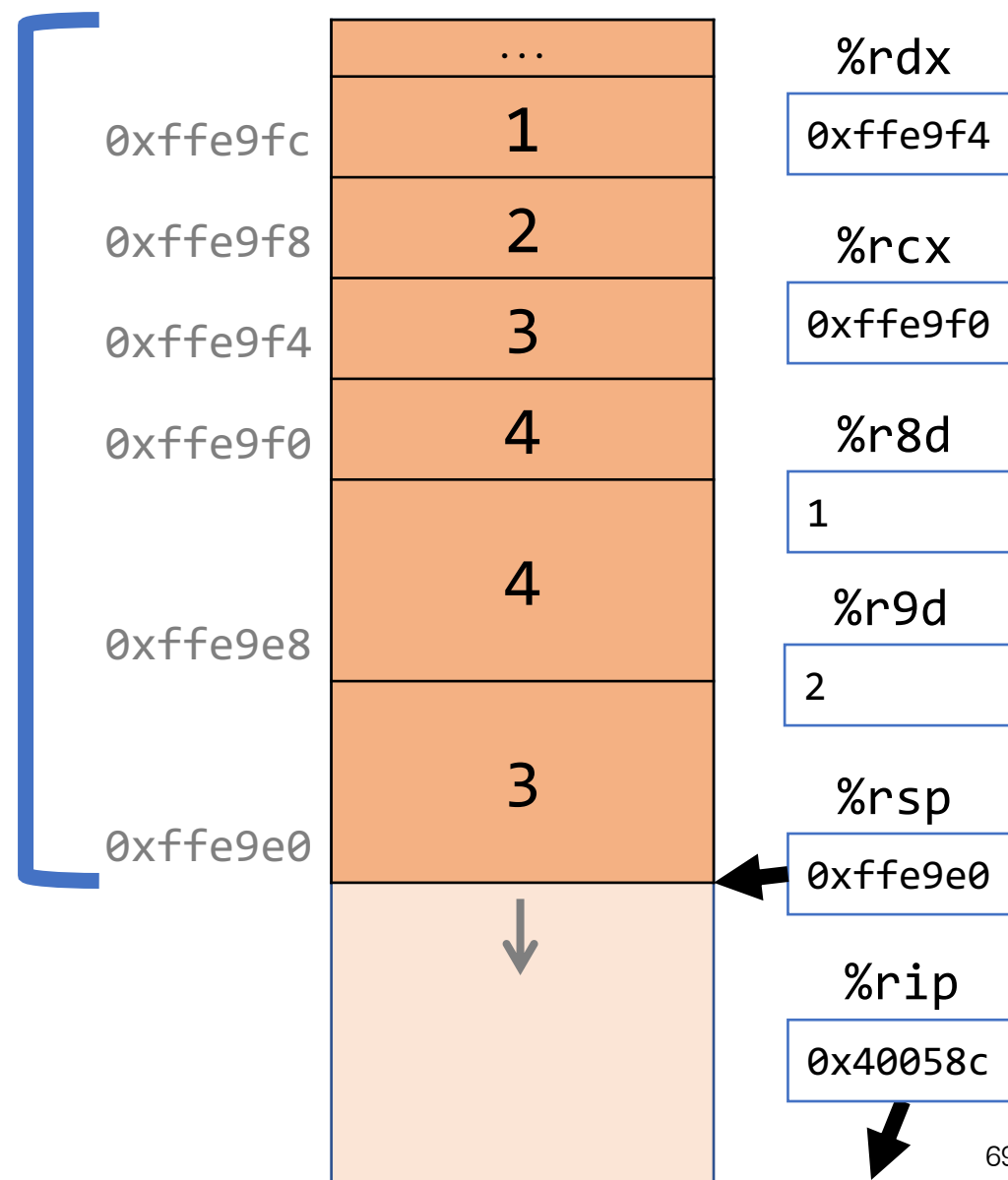
Example 2: Parameters and Return

```
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4,
                     i1, i2, i3, i4);
    ...
}

int func(int *p1, int *p2, int *p3, int *p4,
        int v1, int v2, int v3, int v4) {
    ...
}
```

```
0x40057c <+45>:  mov    $0x1,%r8d
0x400582 <+51>:  lea    0x10(%rsp),%rcx
0x400587 <+56>:  lea    0x14(%rsp),%rdx
0x40058c <+61>:  lea    0x18(%rsp),%rsi
0x400591 <+66>:  lea    0x1c(%rsp),%rdi
```

main()



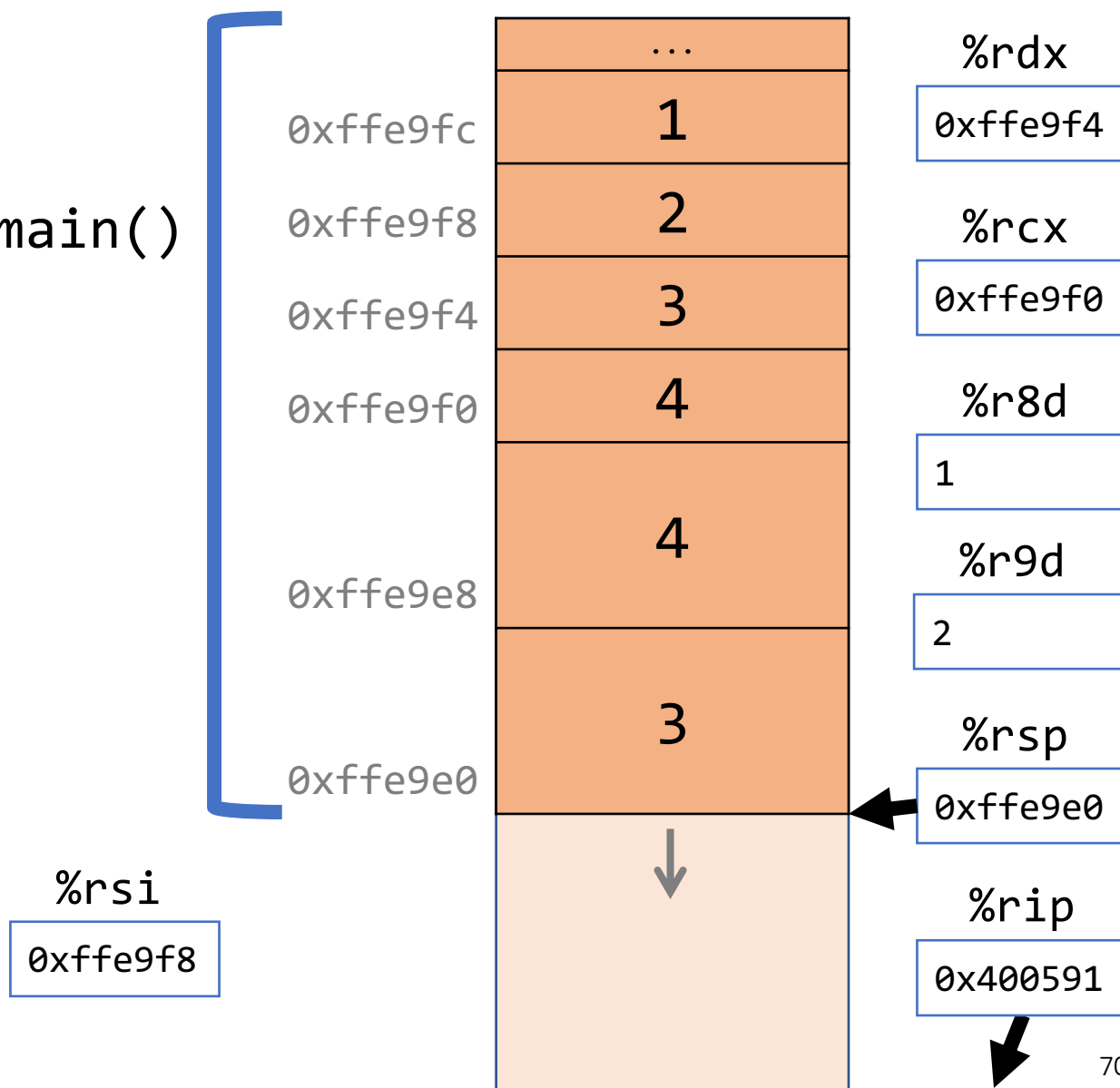
Example 2: Parameters and Return

```
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4,
                     i1, i2, i3, i4);
    ...
}

int func(int *p1, int *p2, int *p3, int *p4,
        int v1, int v2, int v3, int v4) {
    ...
}
```

```
0x400582 <+51>: lea    0x10(%rsp),%rcx
0x400587 <+56>: lea    0x14(%rsp),%rdx
0x40058c <+61>: lea    0x18(%rsp),%rsi
0x400591 <+66>: lea    0x1c(%rsp),%rdi
0x400596 <+71>: callq  0x400546 <func>
```

main()

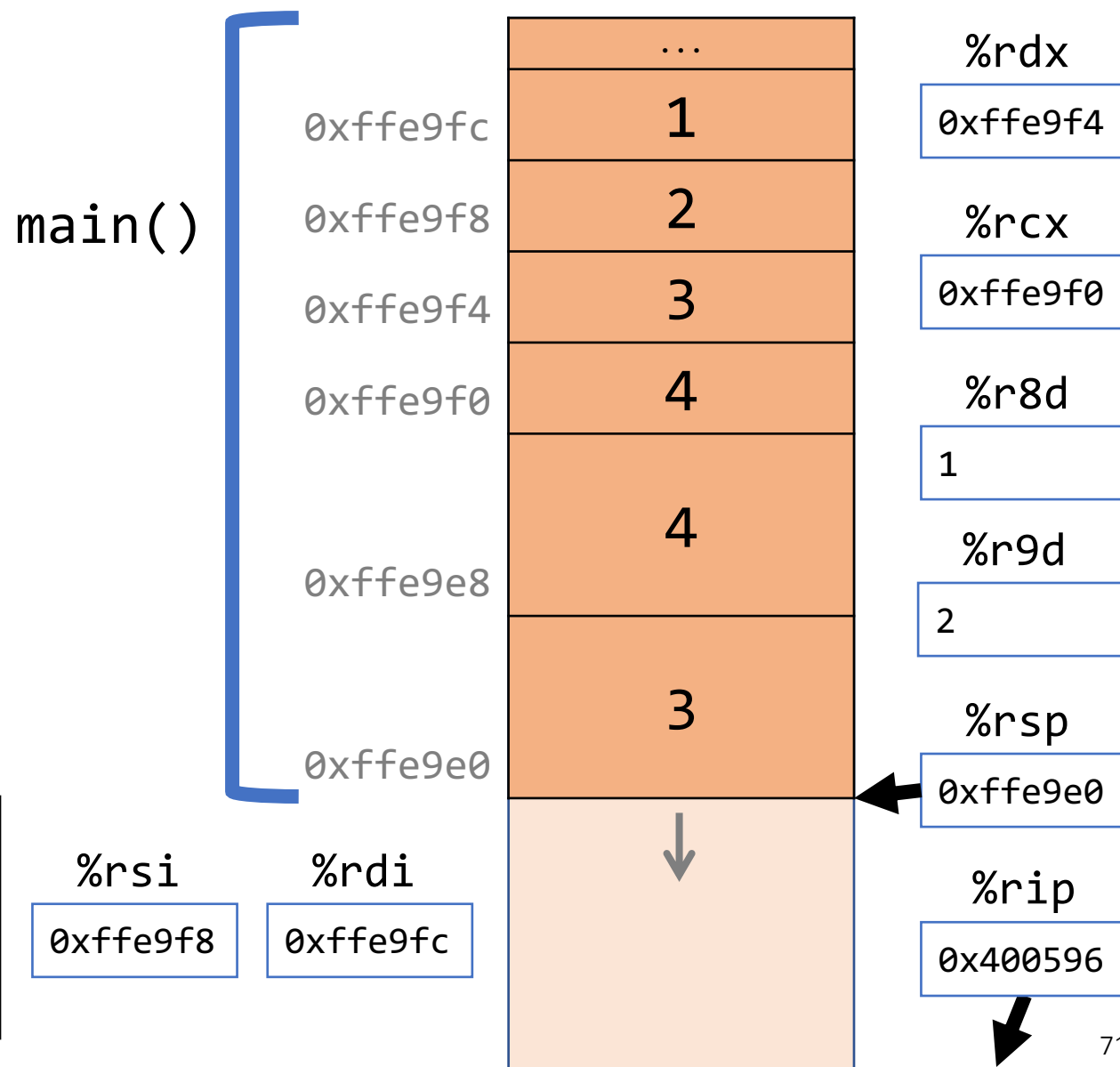


Example 2: Parameters and Return

```
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4,
                     i1, i2, i3, i4);
    ...
}

int func(int *p1, int *p2, int *p3, int *p4,
         int v1, int v2, int v3, int v4) {
    ...
}
```

```
0x400587 <+56>: lea    0x14(%rsp),%rdx
0x40058c <+61>: lea    0x18(%rsp),%rsi
0x400591 <+66>: lea    0x1c(%rsp),%rdi
0x400596 <+71>: callq  0x400546 <func>
0x40059b <+76>: add     $0x10,%rsp
```

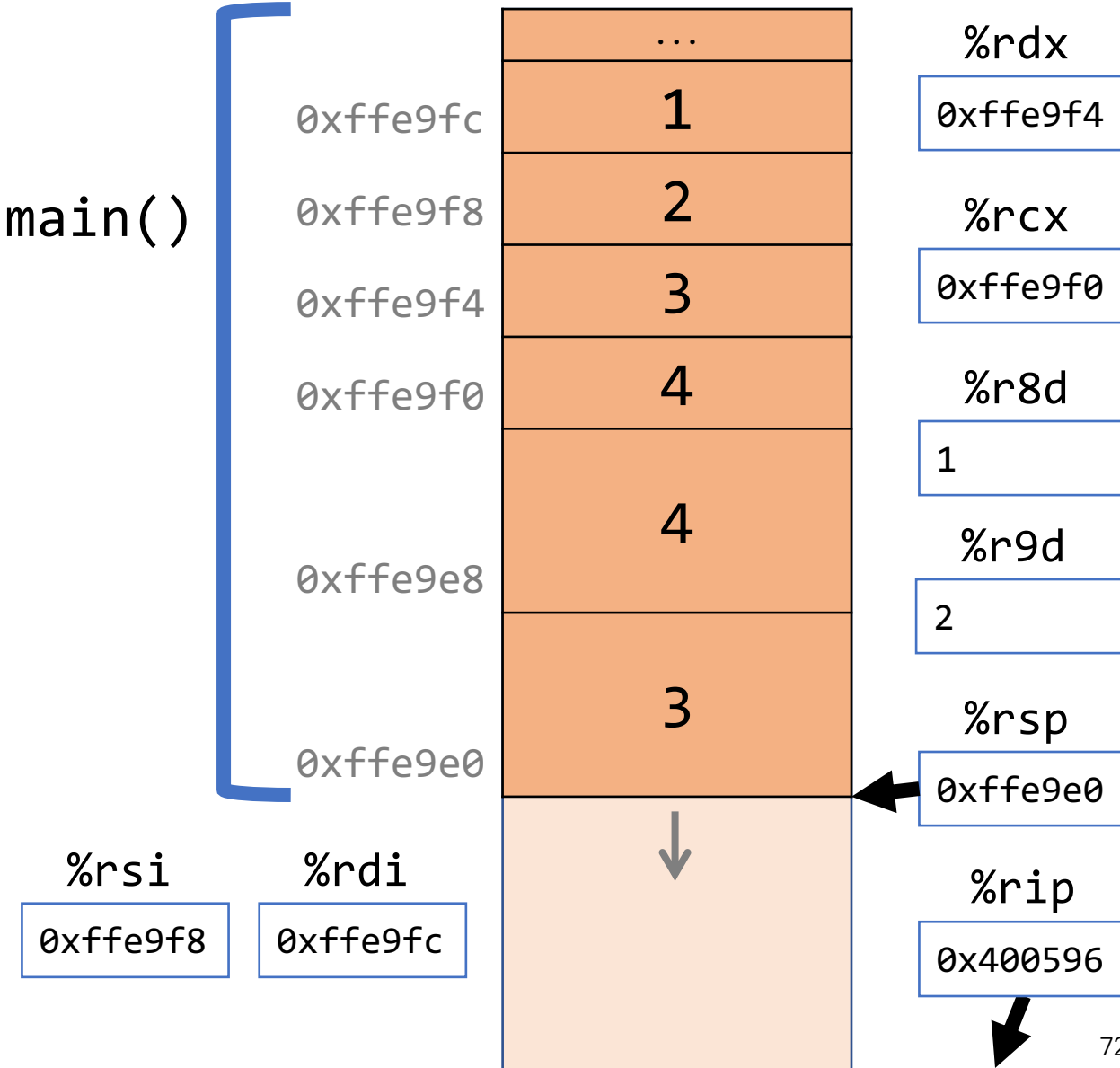


Example 2: Parameters and Return

```
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4,
                     i1, i2, i3, i4);
    ...
}

int func(int *p1, int *p2, int *p3, int *p4,
        int v1, int v2, int v3, int v4) {
    ...
}
```

```
0x40058c <+61>: lea    0x18(%rsp),%rsi
0x400591 <+66>: lea    0x1c(%rsp),%rdi
0x400596 <+71>: callq  0x400546 <func>
0x40059b <+76>: add    $0x10,%rsp
...
```

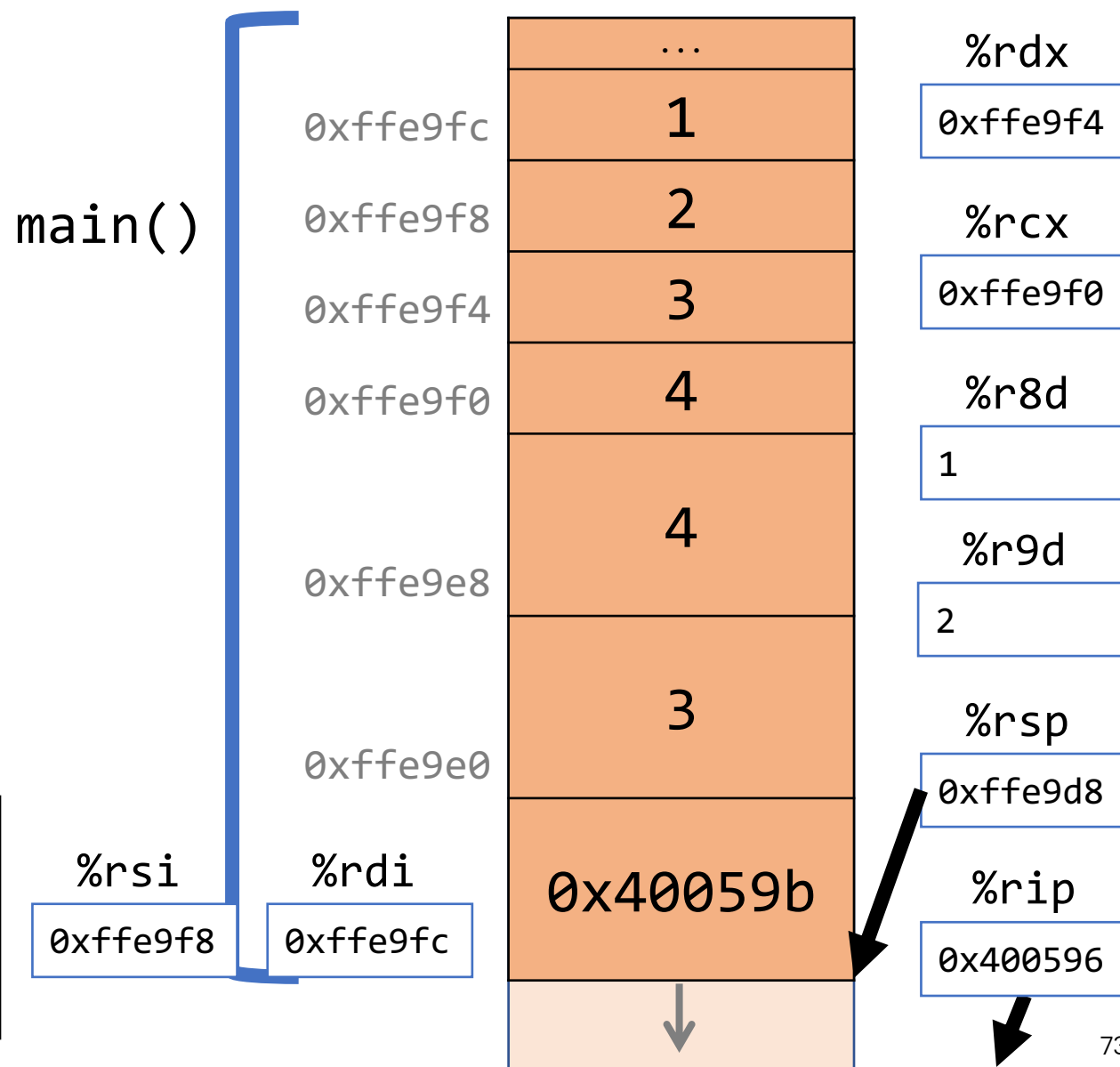


Example 2: Parameters and Return

```
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4,
                     i1, i2, i3, i4);
    ...
}

int func(int *p1, int *p2, int *p3, int *p4,
        int v1, int v2, int v3, int v4) {
    ...
}
```

```
0x40058c <+61>: lea    0x18(%rsp),%rsi
0x400591 <+66>: lea    0x1c(%rsp),%rdi
0x400596 <+71>: callq  0x400546 <func>
0x40059b <+76>: add    $0x10,%rsp
...
```



Lecture Plan

- Revisiting `%rip`
- Calling Functions
 - The Stack
 - Passing Control
 - Passing Data
 - Local Storage
- Register Restrictions
- Pulling it all together: recursion example

Calling Functions In Assembly

To call a function in assembly, we must do a few things:

- **Pass Control** – %rip must be adjusted to execute the function being called and then resume the caller function afterwards.
- **Pass Data** – we must pass any parameters and receive any return value.
- **Manage Memory** – we must handle any space needs of the callee on the stack.

Terminology: **caller** function calls the **callee** function.

Local Storage

- So far, we've often seen local variables stored directly in registers, rather than on the stack as we'd expect. This is for optimization reasons.
- There are **three** common reasons that local data must be in memory:
 - We've run out of registers
 - The '&' operator is used on it, so we must generate an address for it
 - They are arrays or structs (need to use address arithmetic)

Local Storage

```
long caller() {  
    long arg1 = 534;  
    long arg2 = 1057;  
    long sum = swap_add(&arg1, &arg2);  
    ...  
}
```

```
caller:  
    subq $0x10, %rsp          // 16 bytes for stack frame  
    movq $0x216, (%rsp)      // store 534 in arg1  
    movq $0x421, 8(%rsp)     // store 1057 in arg2  
    leaq 8(%rsp), %rsi       // compute &arg2 as second arg  
    movq %rsp, %rdi          // compute &arg1 as first arg  
    call swap_add            // call swap_add(&arg1, &arg2)
```

Lecture Plan

- Revisiting `%rip`
- Calling Functions
 - The Stack
 - Passing Control
 - Passing Data
 - Local Storage
- Register Restrictions
- Pulling it all together: recursion example

Register Restrictions

- When procedure `yoo` calls `who`:
 - `yoo` is the **caller**
 - `who` is the **callee**
- Can register be used for temporary storage?

`yoo:`

```
• • •  
movq $15213, %rdx  
call who  
addq %rdx, %rax  
• • •  
ret
```

`who:`

```
• • •  
subq $18213, %rdx  
• • •  
ret
```

- Contents of register `%rdx` overwritten by `who`
- This could be trouble → something should be done!
 - Need some coordination

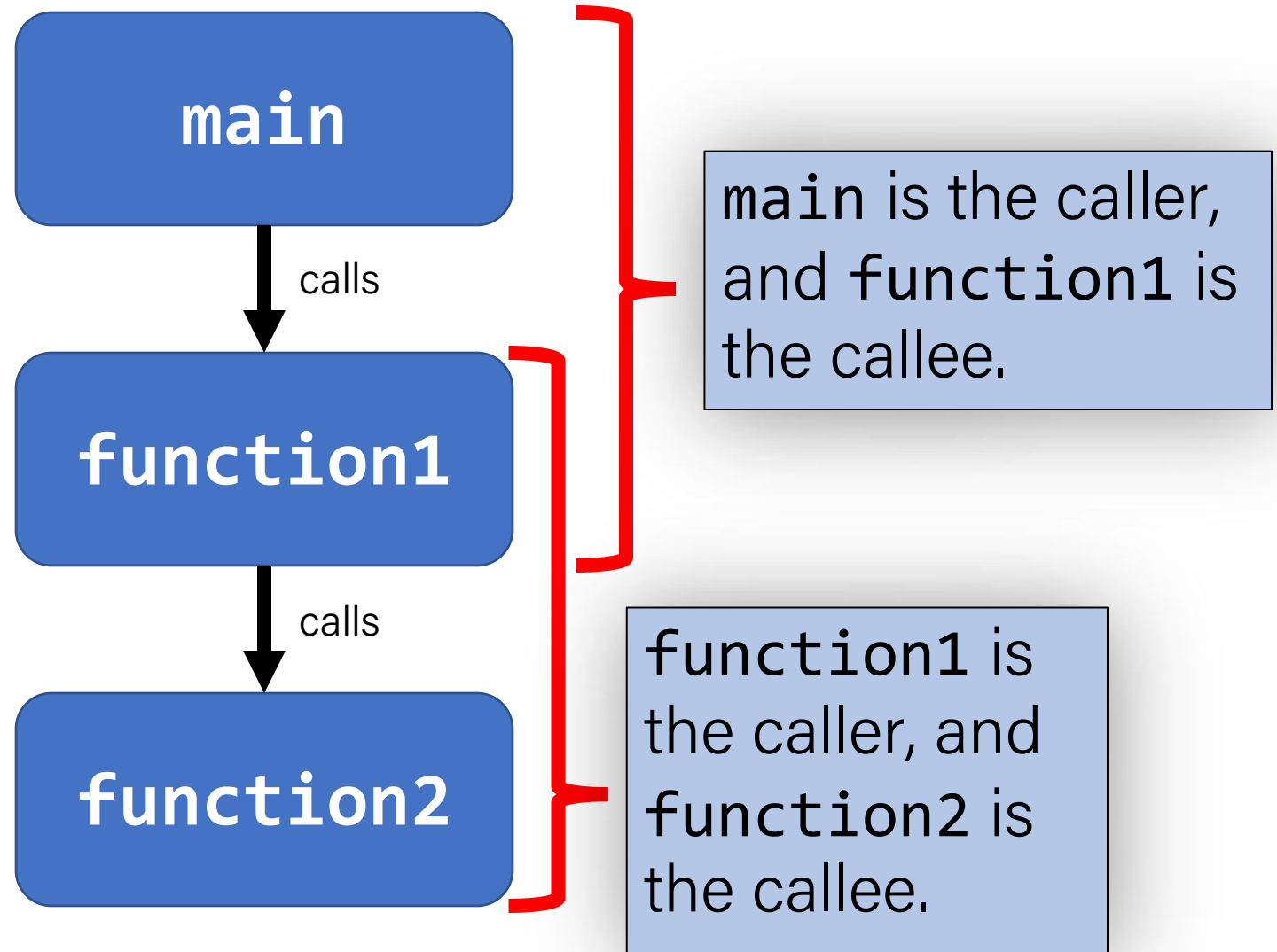
Register Restrictions

There is only one copy of registers for all programs and functions.

- **Problem:** what if *funcA* is building up a value in register %r10, and calls *funcB* in the middle, which also has instructions that modify %r10? *funcA*'s value will be overwritten!
- **Solution:** make some “rules of the road” that callers and callees must follow when using registers so they do not interfere with one another.
- These rules define two types of registers: **caller-owned** and **callee-owned**

Caller/Callee

Caller/callee is terminology that refers to a pair of functions. A single function may be both a caller and callee simultaneously (e.g. `function1` at right).



Register Restrictions

Caller-Owned (Callee Saved)

- Callee must save the existing value and *restore* it when done.
- Caller can store values and assume they will be preserved across function calls.

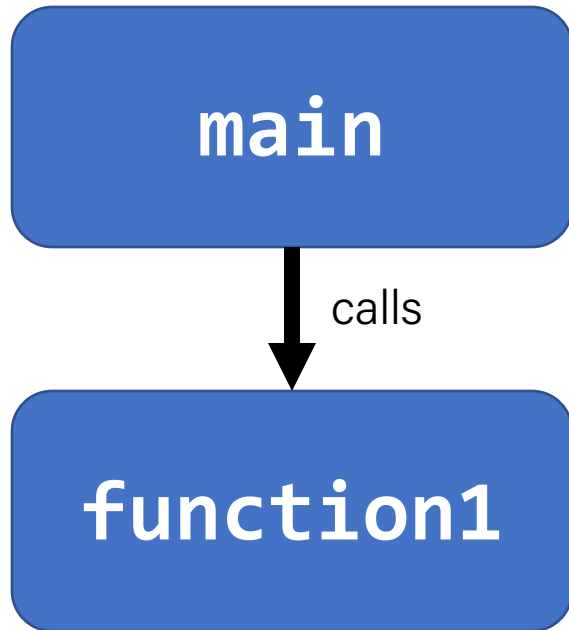
Callee-Owned (Caller Saved)

- Callee does not need to save the existing value.
- Caller's values could be overwritten by a callee! The caller may consider saving values elsewhere before calling functions.



Figure 3.2 Integer registers. The low-order portions of all 16 registers can be accessed as byte, word (16-bit), double word (32-bit), and quad word (64-bit) quantities.

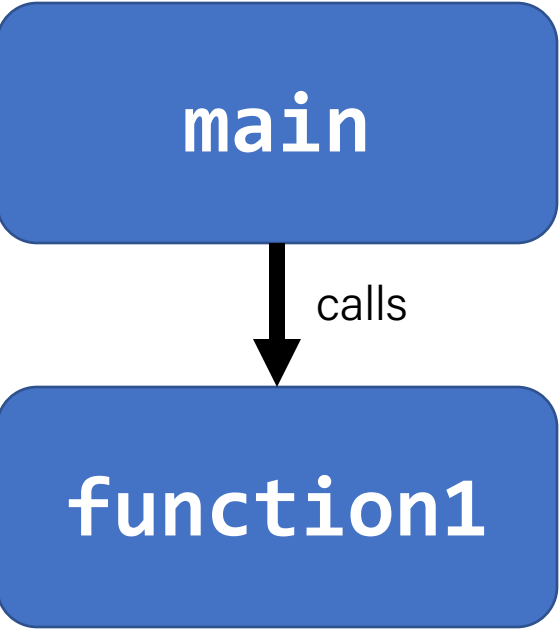
Caller-Owned Registers



`main` can use caller-owned registers and know that `function1` will not permanently modify their values.

If `function1` wants to use any caller-owned registers, it must save the existing values and restore them before returning.

Caller-Owned Registers

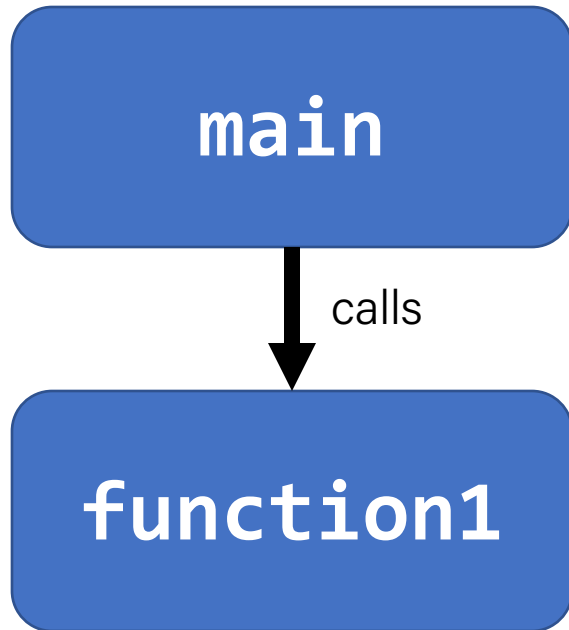


```
function1:  
    push %rbp  
    push %rbx  
    ...  
    pop %rbx  
    pop %rbp  
    retq
```

63	31	15	7	0	
%rax	%eax	%ax	%al		Return value
%rbx	%ebx	%bx	%bl		Callee saved
%rcx	%ecx	%cx	%cl		4th argument
%rdx	%edx	%dx	%dl		3rd argument
%rsi	%esi	%si	%sil		2nd argument
%rdi	%edi	%di	%dil		1st argument
%rbp	%ebp	%bp	%bpl		Callee saved
%rsp	%esp	%sp	%spl		Stack pointer
%r8	%r8d	%r8w	%r8b		5th argument
%r9	%r9d	%r9w	%r9b		6th argument
%r10	%r10d	%r10w	%r10b		Caller saved
%r11	%r11d	%r11w	%r11b		Caller saved
%r12	%r12d	%r12w	%r12b		Callee saved
%r13	%r13d	%r13w	%r13b		Callee saved
%r14	%r14d	%r14w	%r14b		Callee saved
%r15	%r15d	%r15w	%r15b		Callee saved

Figure 3.2 Integer registers. The low-order portions of all 16 registers can be accessed as byte, word (16-bit), double word (32-bit), and quad word (64-bit) quantities.

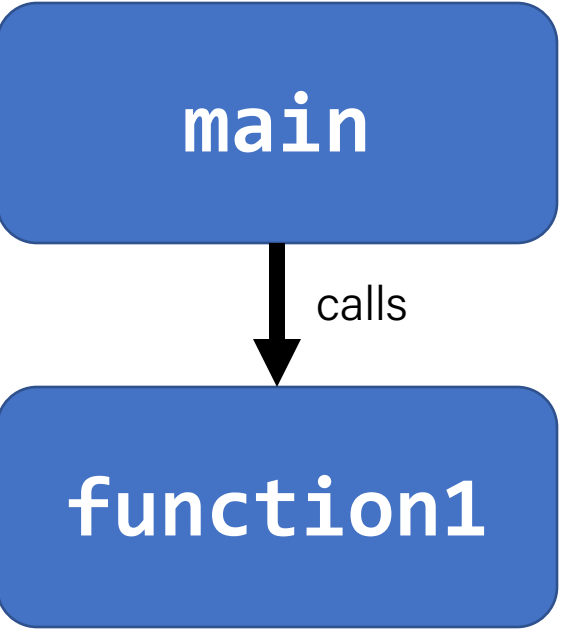
Callee-Owned Registers



`main` can use callee-owned registers but calling `function1` may permanently modify their values.

If `function1` wants to use any callee-owned registers, it can do so without saving the existing values.

Callee-Owned Registers



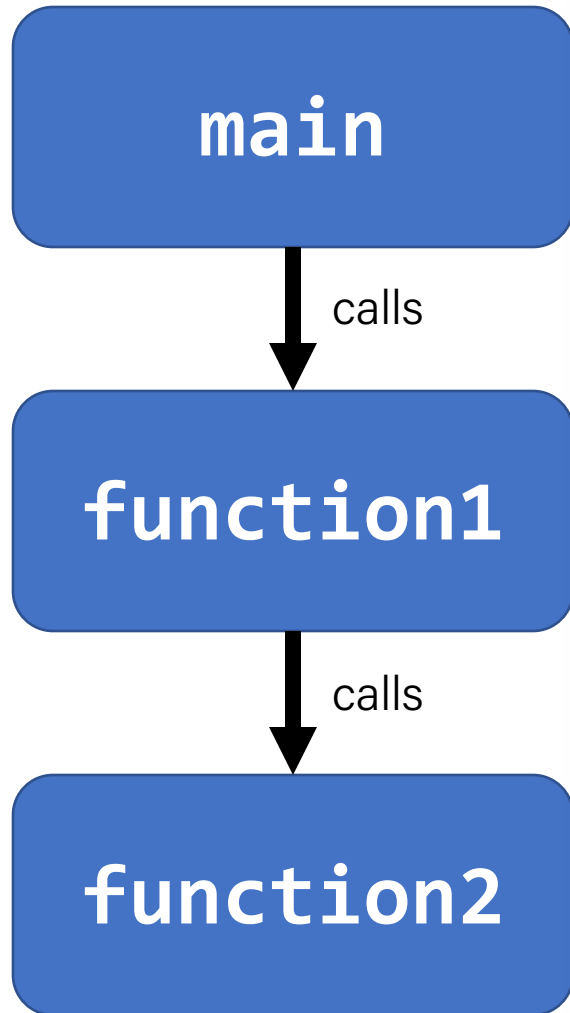
```

main:
    ...
    push %r10
    push %r11
    callq function1
    pop %r11
    pop %r10
    ...
  
```

63	31	15	7	0	
%rax	%eax	%ax	%al		Return value
%rbx	%ebx	%bx	%bl		Callee saved
%rcx	%ecx	%cx	%cl		4th argument
%rdx	%edx	%dx	%dl		3rd argument
%rsi	%esi	%si	%sil		2nd argument
%rdi	%edi	%di	%dil		1st argument
%rbp	%ebp	%bp	%bpl		Callee saved
%rsp	%esp	%sp	%spl		Stack pointer
%r8	%r8d	%r8w	%r8b		5th argument
%r9	%r9d	%r9w	%r9b		6th argument
%r10	%r10d	%r10w	%r10b		Caller saved
%r11	%r11d	%r11w	%r11b		Caller saved
%r12	%r12d	%r12w	%r12b		Callee saved
%r13	%r13d	%r13w	%r13b		Callee saved
%r14	%r14d	%r14w	%r14b		Callee saved
%r15	%r15d	%r15w	%r15b		Callee saved

Figure 3.2 Integer registers. The low-order portions of all 16 registers can be accessed as byte, word (16-bit), double word (32-bit), and quad word (64-bit) quantities.

A Day In the Life of `function1`



Caller-owned registers:

- **`function1`** must save/restore existing values of any it wants to use.
- **`function1`** can assume that calling **`function2`** will not permanently change their values.

Callee-owned registers:

- **`function1`** does not need to save/restore existing values of any it wants to use.
- calling **`function2`** may permanently change their values.

Lecture Plan

- Revisiting `%rip`
- Calling Functions
 - The Stack
 - Passing Control
 - Passing Data
 - Local Storage
- Register Restrictions
- Pulling it all together: recursion example

Example: Recursion

- Let's look at an example of recursion at the assembly level.
- We'll use everything we've learned about registers, the stack, function calls, parameters, and assembly instructions!

<https://godbolt.org/z/f43dz1>



`factorial.c` and `factorial`

Our First Assembly

```
int sum_array(int arr[], int nelems) {  
    int sum = 0;  
    for (int i = 0; i < nelems; i++) {  
        sum += arr[i];  
    }  
    return sum;  
}
```

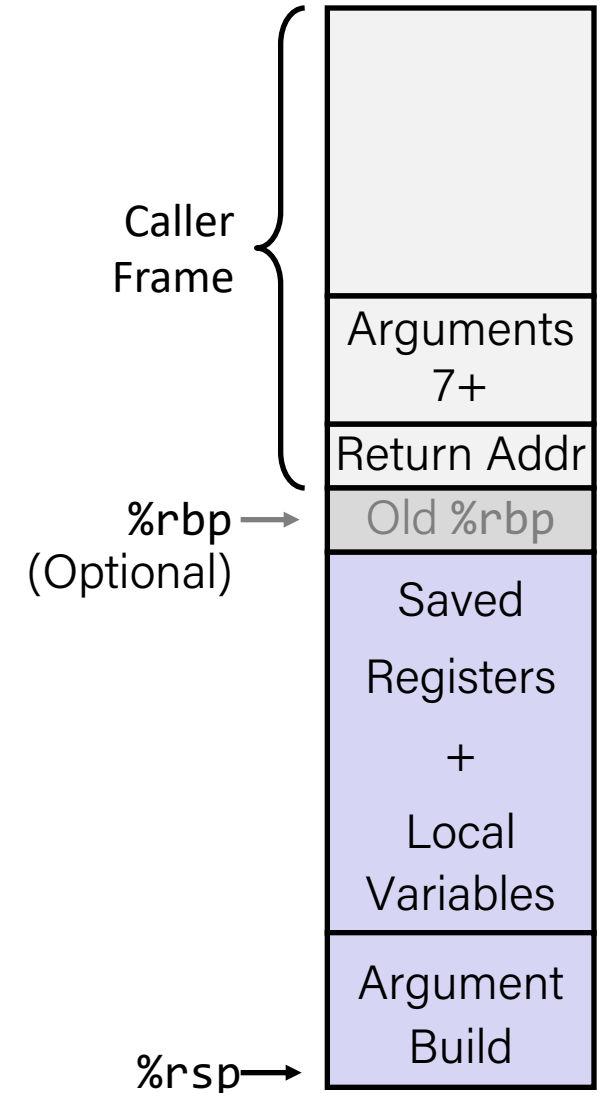
We're done with all our assembly lectures! Now we can fully understand what's going on in the assembly below, including how someone would call `sum_array` in assembly and what the **ret** instruction does.

00000000004005b6 <sum_array>:

4005b6:	ba 00 00 00 00	mov	\$0x0,%edx
4005bb:	b8 00 00 00 00	mov	\$0x0,%eax
4005c0:	eb 09	jmp	4005cb <sum_array+0x15>
4005c2:	48 63 ca	movslq	%edx,%rcx
4005c5:	03 04 8f	add	(%rdi,%rcx,4),%eax
4005c8:	83 c2 01	add	\$0x1,%edx
4005cb:	39 f2	cmp	%esi,%edx
4005cd:	7c f3	j1	4005c2 <sum_array+0xc>
4005cf:	f3 c3	repz	retq

x86-64 Procedure Summary

- Important Points
 - Stack is the right data structure for procedure call/return
 - If P calls Q, then Q returns before P
- Recursion (& mutual recursion) handled by normal calling conventions
 - Can safely store values in local stack frame and in callee-saved registers
 - Put function arguments at top of stack
 - Result return in **%rax**
- Pointers are addresses of values
 - On stack or global



Recap

- Revisiting `%rip`
- Calling Functions
 - The Stack
 - Passing Control
 - Passing Data
 - Local Storage
- Register Restrictions
- Pulling it all together: recursion example

Next time: *data stack frames*