COMP201

Computer Systems & Programming

Lecture #17 – x86-64 Procedures

Aykut Erdem // Koç University // Spring 2021

KOÇ UNIVERSITY

Good news, everyone!

- No labs this week!
- Midterm exam will be held on April 25, 2021 (Sunday) at 18:15am
  - More info on Blackboard

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)?

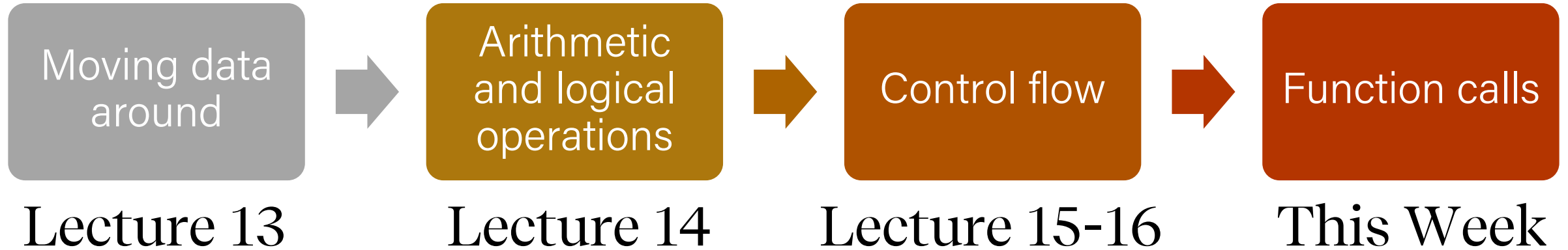# Practice: "Escape Room"

```
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

Moving data around

Lecture 13

Arithmetic and logical operations

Lecture 14

Control flow

Lecture 15-16

Function calls

This Week

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

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++;
    }
}
```

```
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++;
    }
}
```

```
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++;
    }
}
```

```
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++;
    }
}
```

```
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

```
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
```
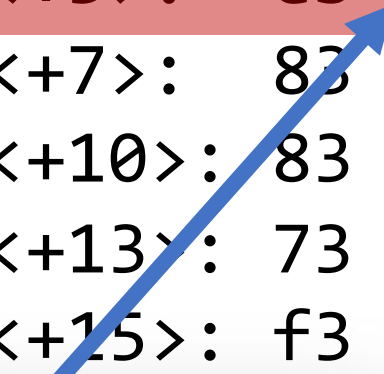
%rip

```
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

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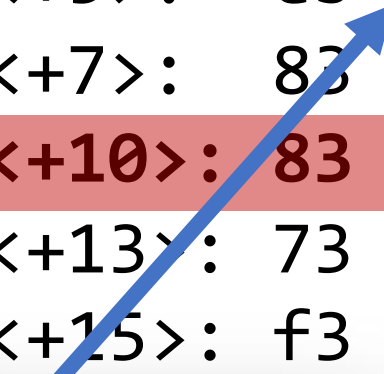
0x03 is the number of instruction bytes to jump relative to **%rip**.

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

%rip

```
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 <u>then</u> go **3** bytes further!

%rip

```
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

```
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 <u>then</u> go **8** bytes back!

20

%rip

```
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 <u>then</u> 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.

Question Break

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.

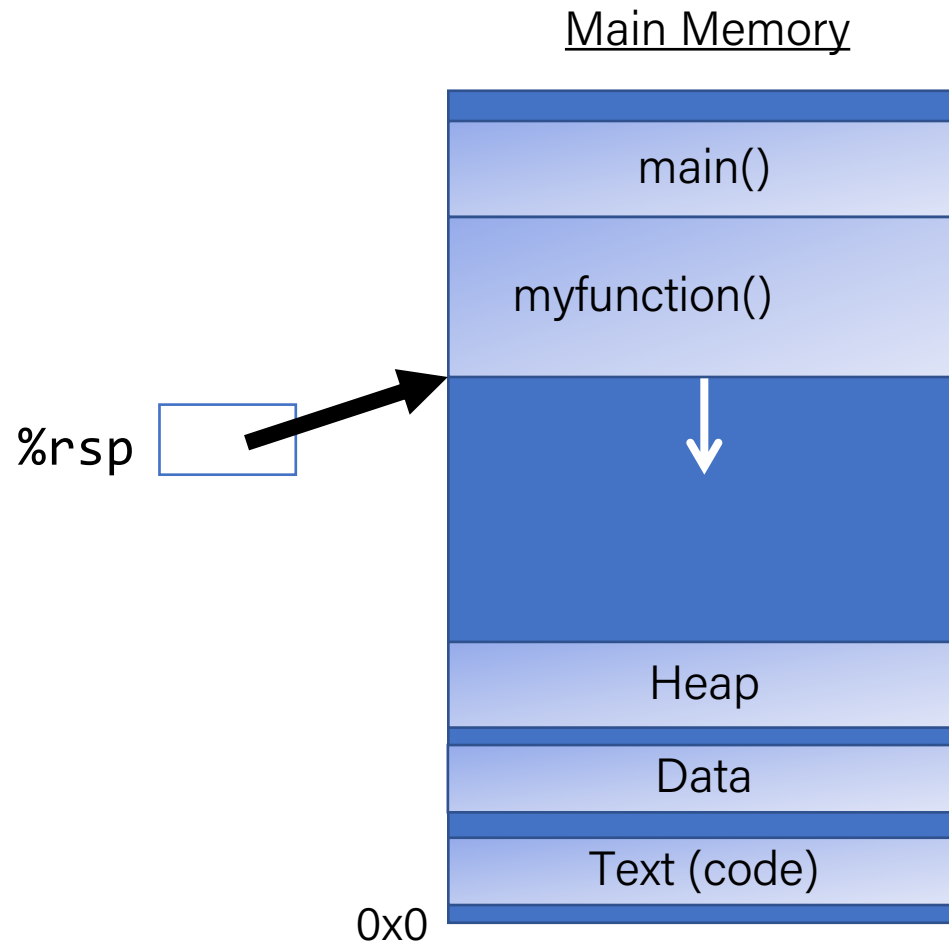How does assembly interact with the stack?

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

Lecture Plan

- Revisiting %rip

- Calling Functions
  - The Stack
  - Passing Control
  - Passing Data
  - Local Storage

- Register Restrictions

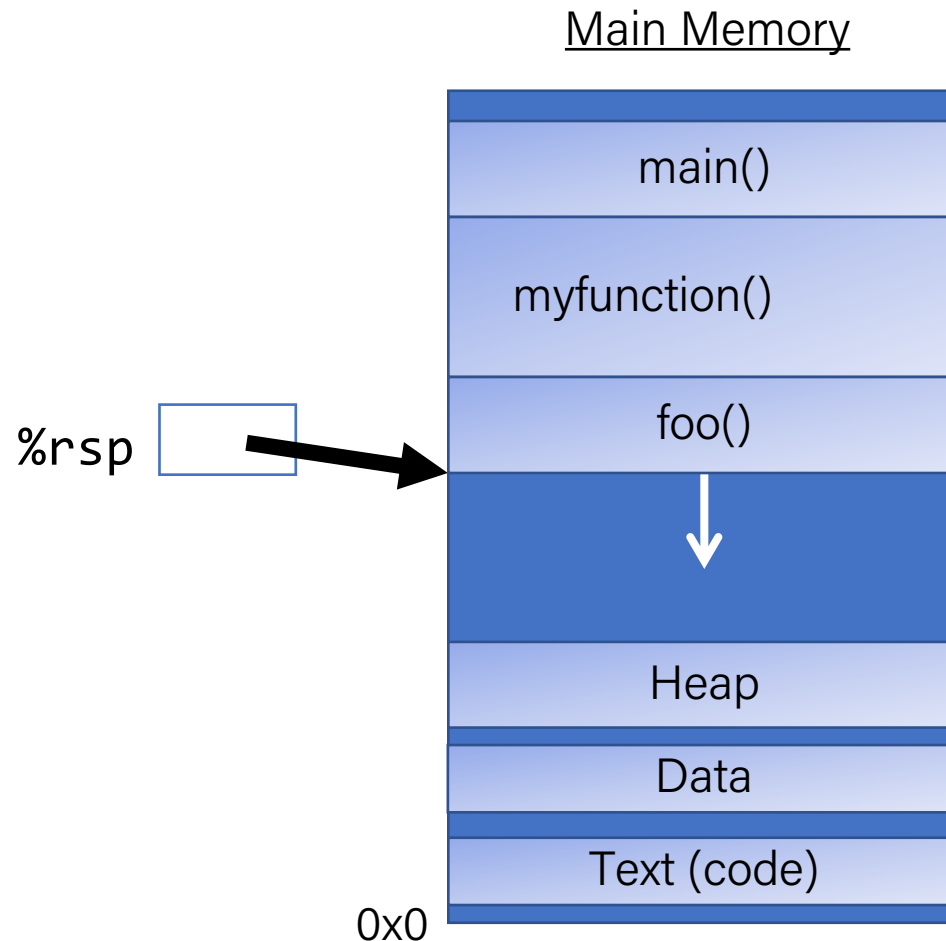- Pulling it all together: recursion example

%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).

Main Memory

main()

myfunction()

%rsp

Heap

Data

Text (code)

0x0

%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).

Main Memory

%rsp

main()

myfunction()
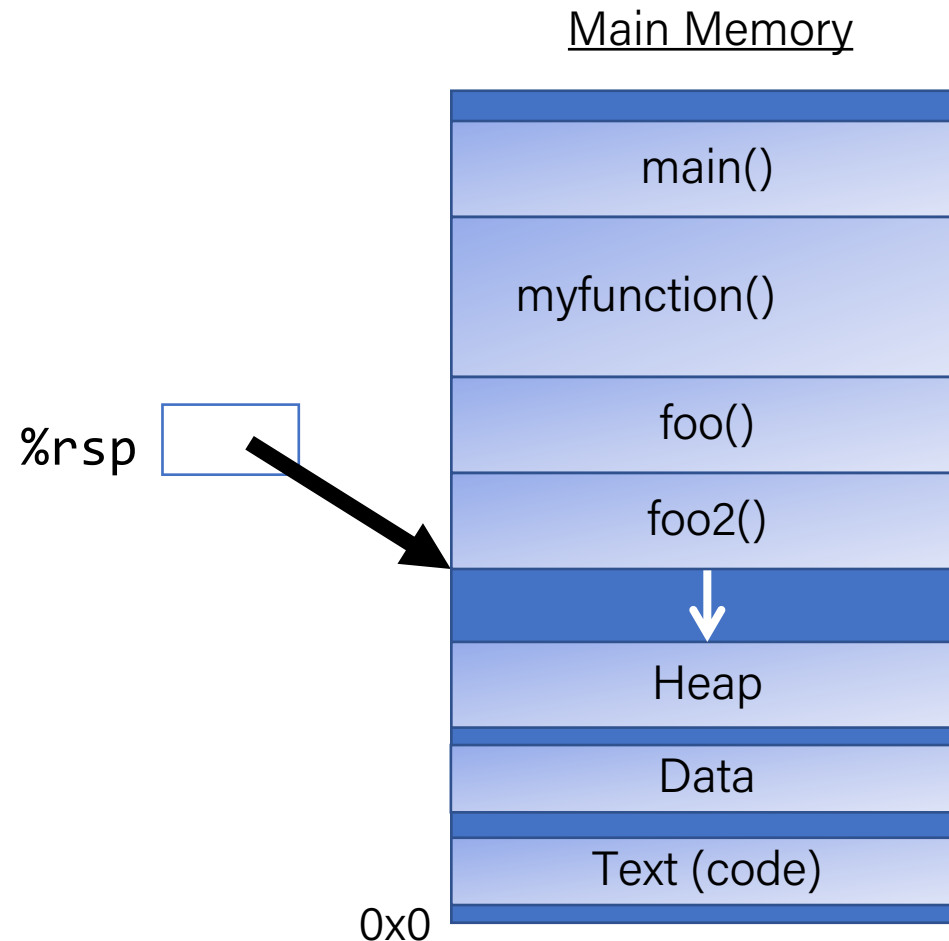
foo()

Heap

Data

Text (code)

0x0

%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).

Main Memory

main()
myfunction()
foo()
foo2()
Heap
Data
Text (code)

%rsp →

0x0

%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).

Main Memory

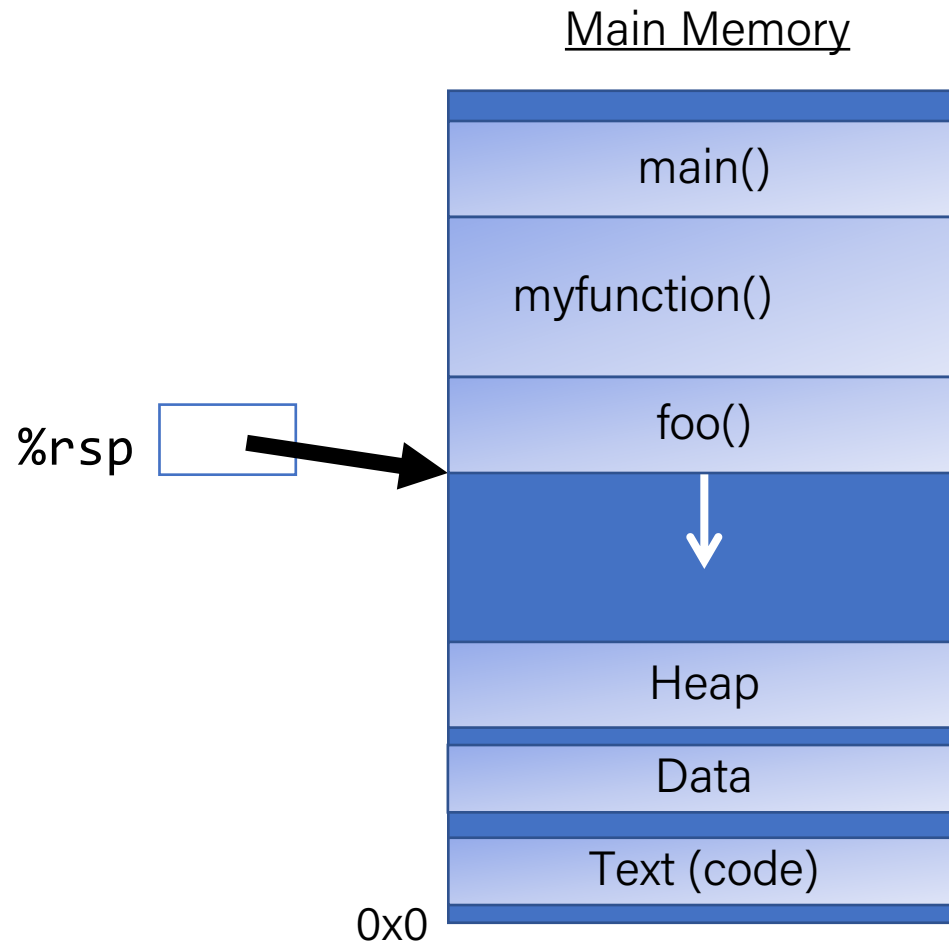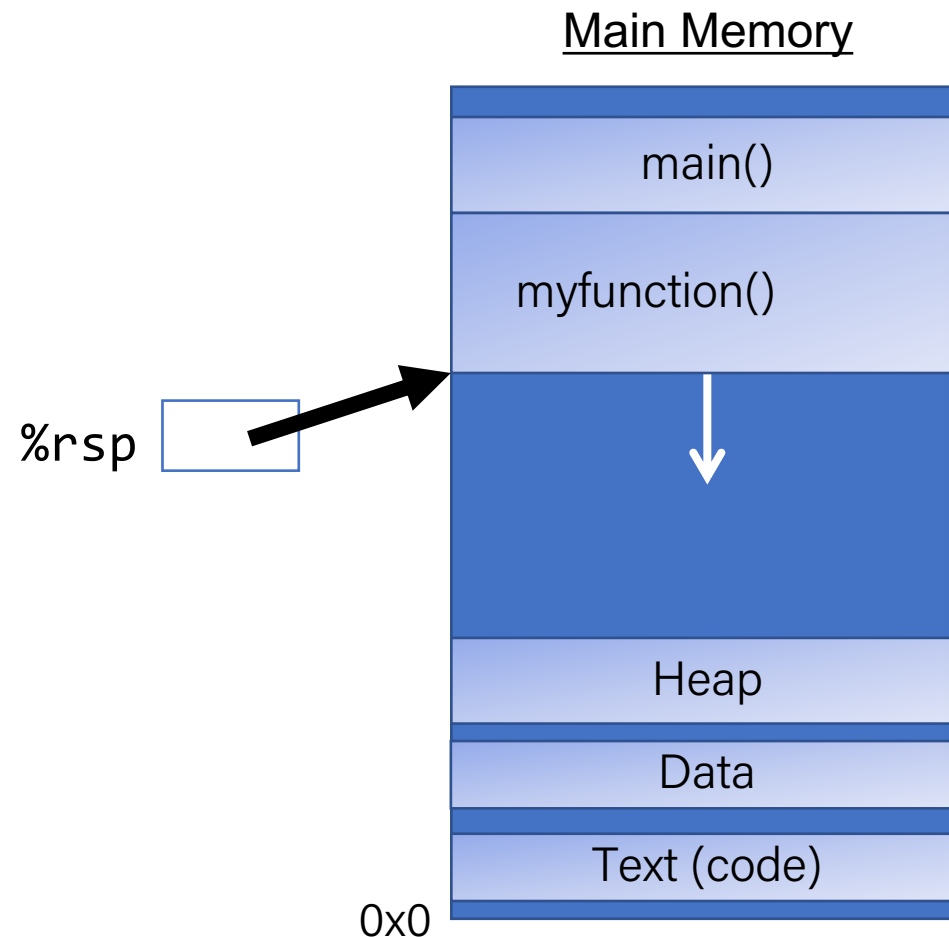| |
|---|
| main() |
| myfunction() |
| foo() |
| ↓ |
| Heap |
| Data |
| Text (code) |

%rsp

0x0

%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).

Main Memory

main()
myfunction()
↓
Heap
Data
Text (code)

%rsp →

0x0

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.

32

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] ← R[%rsp] – 8; M[R[%rsp]] ← 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] ← R[%rsp] – 8; M[R[%rsp]] ← 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] ← R[%rsp] – 8; M[R[%rsp]] ← 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] ← R[%rsp] – 8; M[R[%rsp]] ← 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 ⟵ M[R[%rsp]] R[%rsp] ⟵ 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.

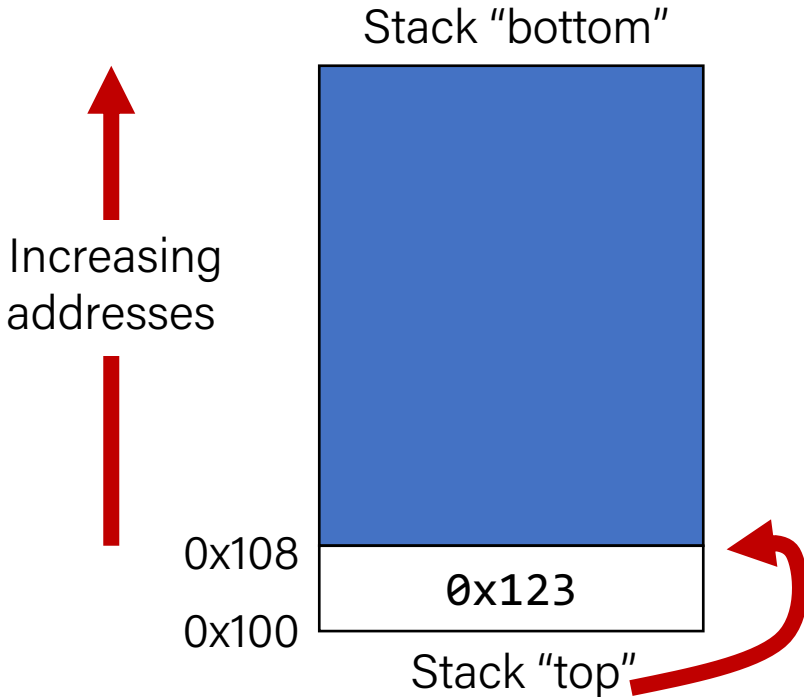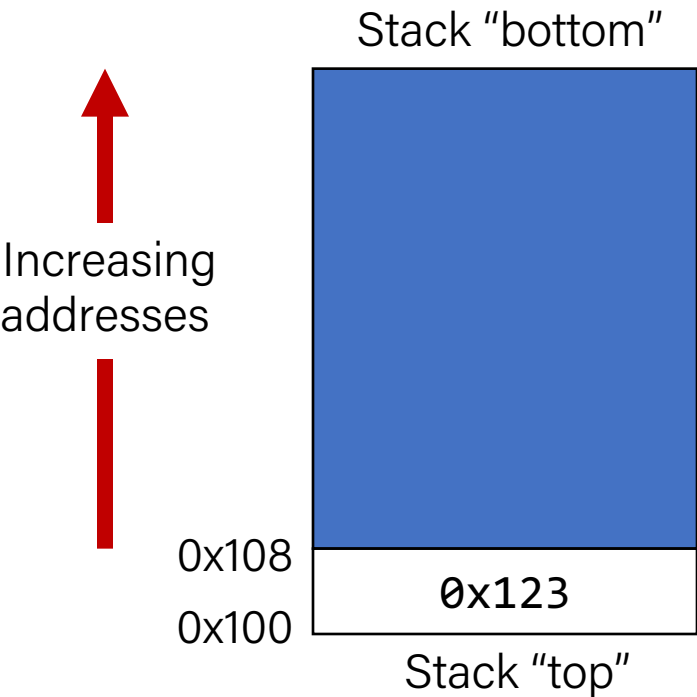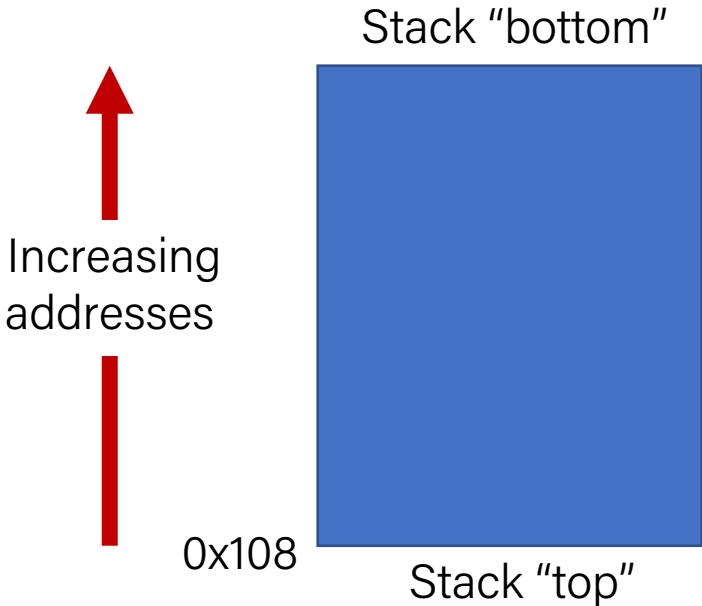| Instruction	Effect
popq D	D ← M[R[%rsp]] R[%rsp] ← R[%rsp] + 8;

- 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

Stack "bottom"

Increasing addresses

0x108

Stack "top"

Stack "bottom"

Increasing addresses

0x108

0x100

0x123

Stack "top"

Stack "bottom"

Increasing addresses

0x108

0x100

0x123

Stack "top"

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.
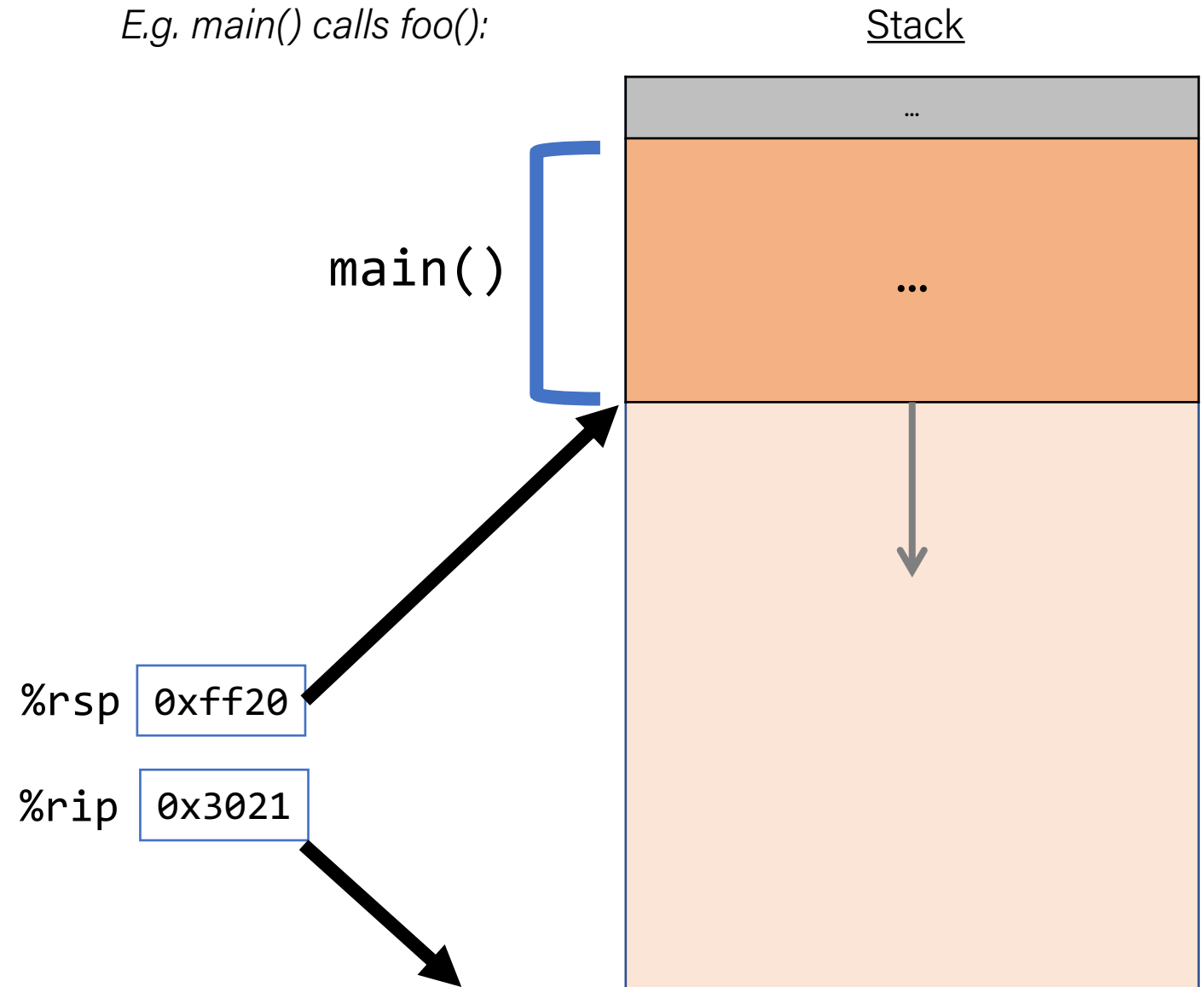
Question Break

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.

E.g. main() calls foo():

Stack

main()

…

…

%rsp 0xff20

%rip 0x3021

Remembering Where We Left Off

Problem: `%rip` points to the next instruction to execute. To call a function, we must <u>remember</u> the *next* caller instruction to resume at after.
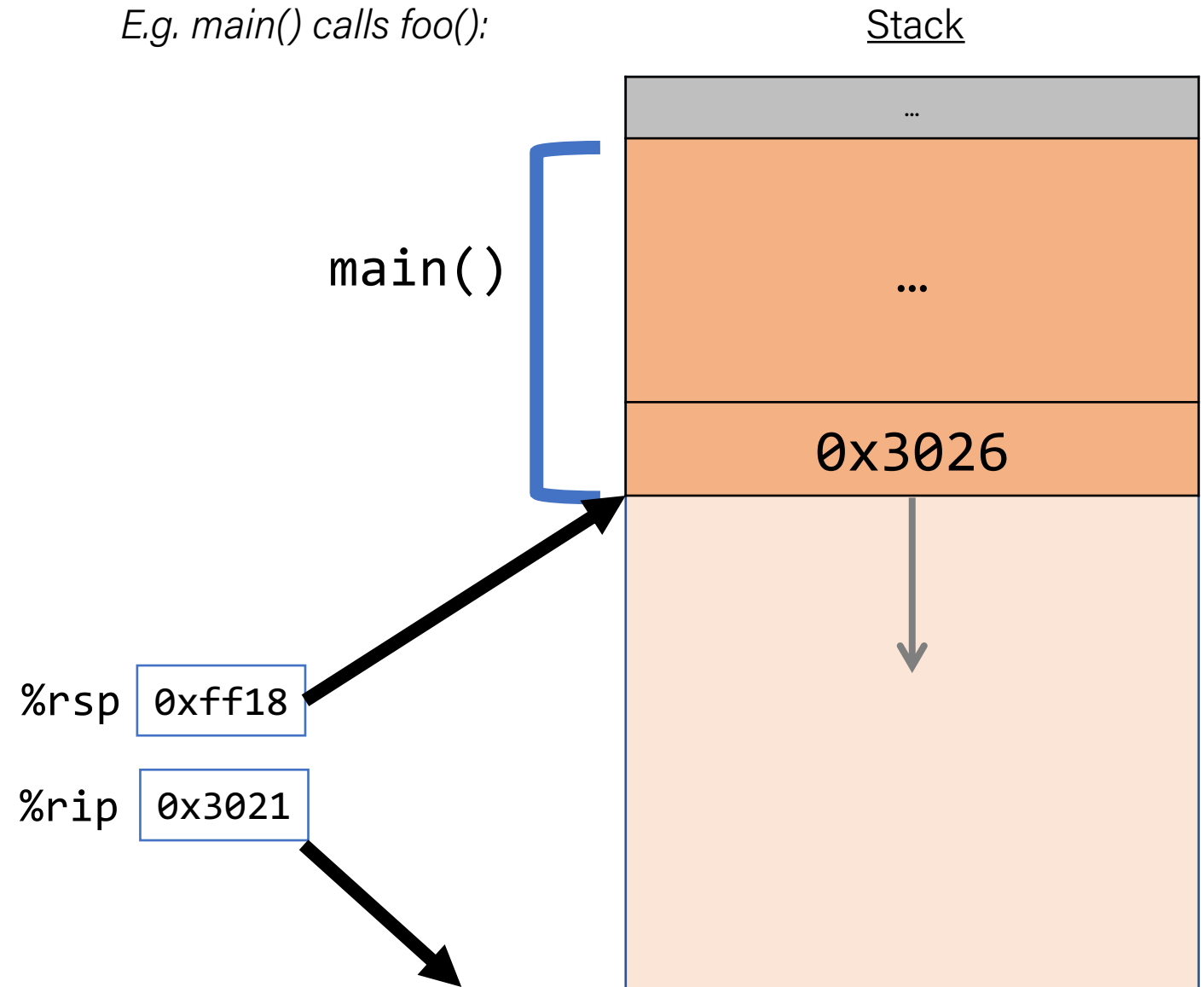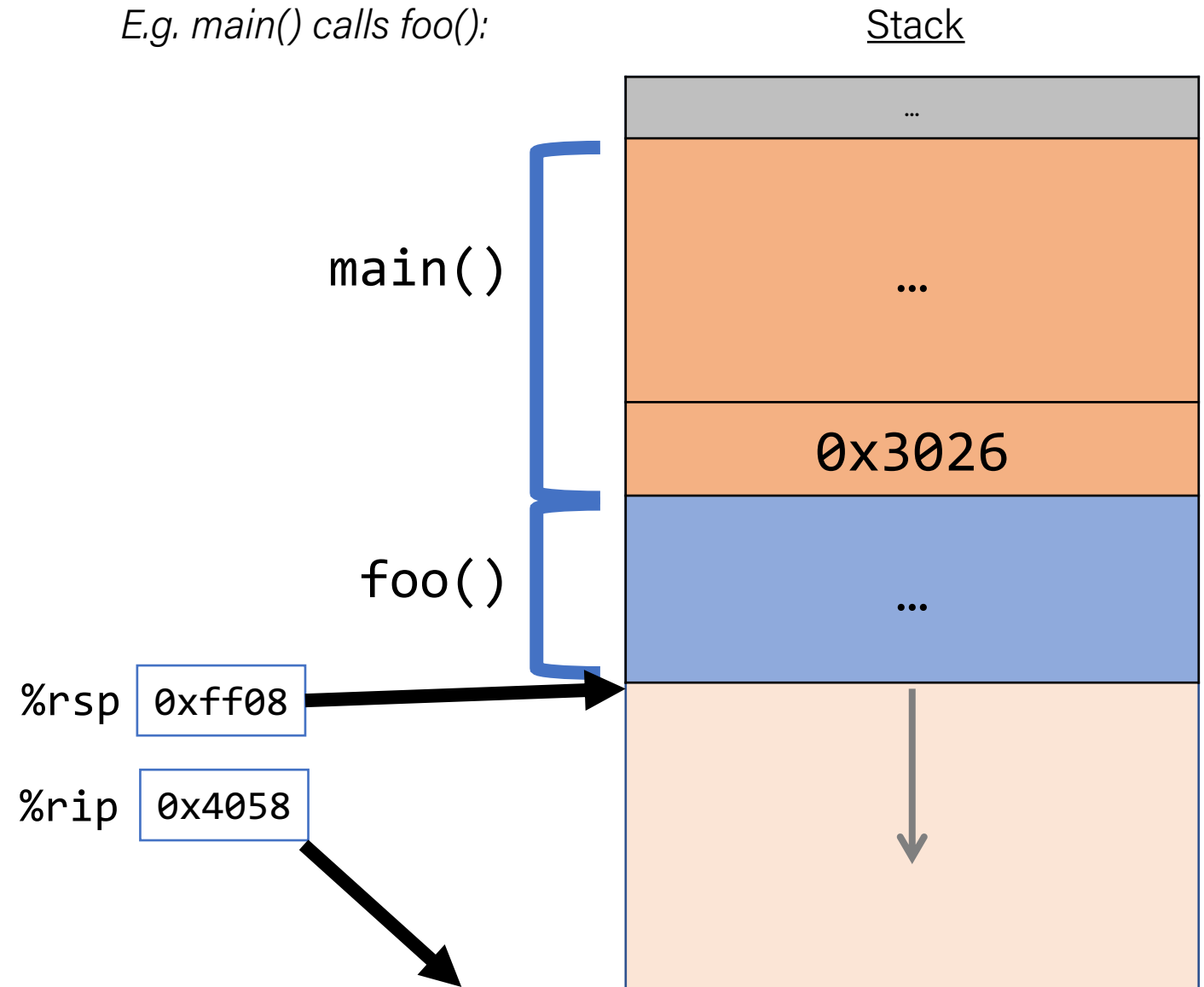
Solution: push the next value of `%rip` onto the stack. Then call the function. When it is finished, put this value back into `%rip` and continue executing.

E.g. main() calls foo():

Stack

main()

…

…

0x3026

%rsp 0xff18

%rip 0x3021

Remembering Where We Left Off

Problem: `%rip` points to the next instruction to execute. To call a function, we must <u>remember</u> 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.

E.g. main() calls foo():

Stack

…
…
0x3026
…

main()

foo()

%rsp | 0xff08 |

%rip | 0x4058 |

Remembering Where We Left Off

Problem: %rip points to the next instruction to execute. To call a function, we must <u>remember</u> 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.

E.g. main() calls foo():

Stack

main()

…

…

0x3026

%rsp | 0xff18

%rip | 0x4058

Remembering Where We Left Off

Problem: %rip points to the next instruction to execute. To call a function, we must <u>remember</u> the *next* caller instruction to resume at after.
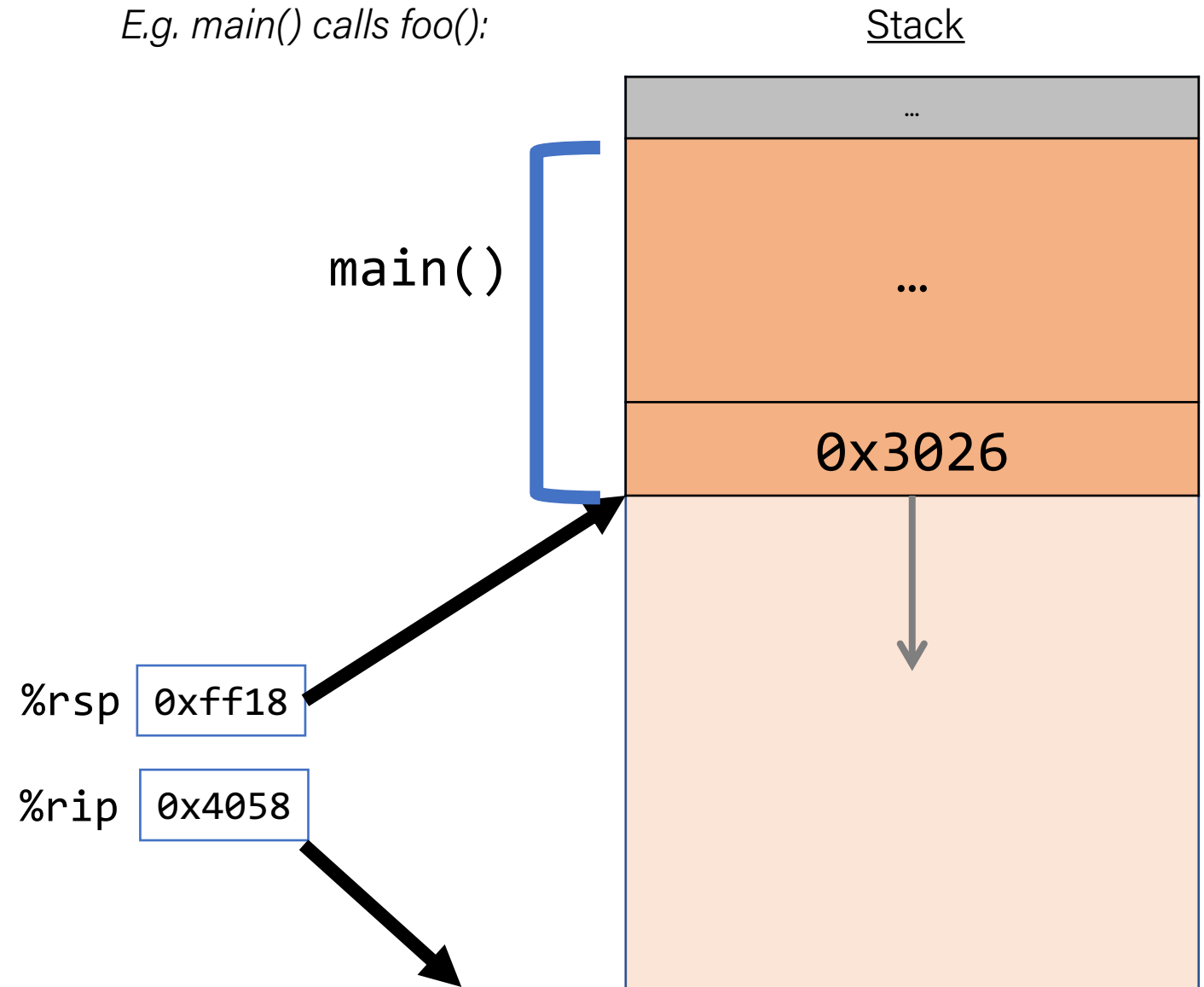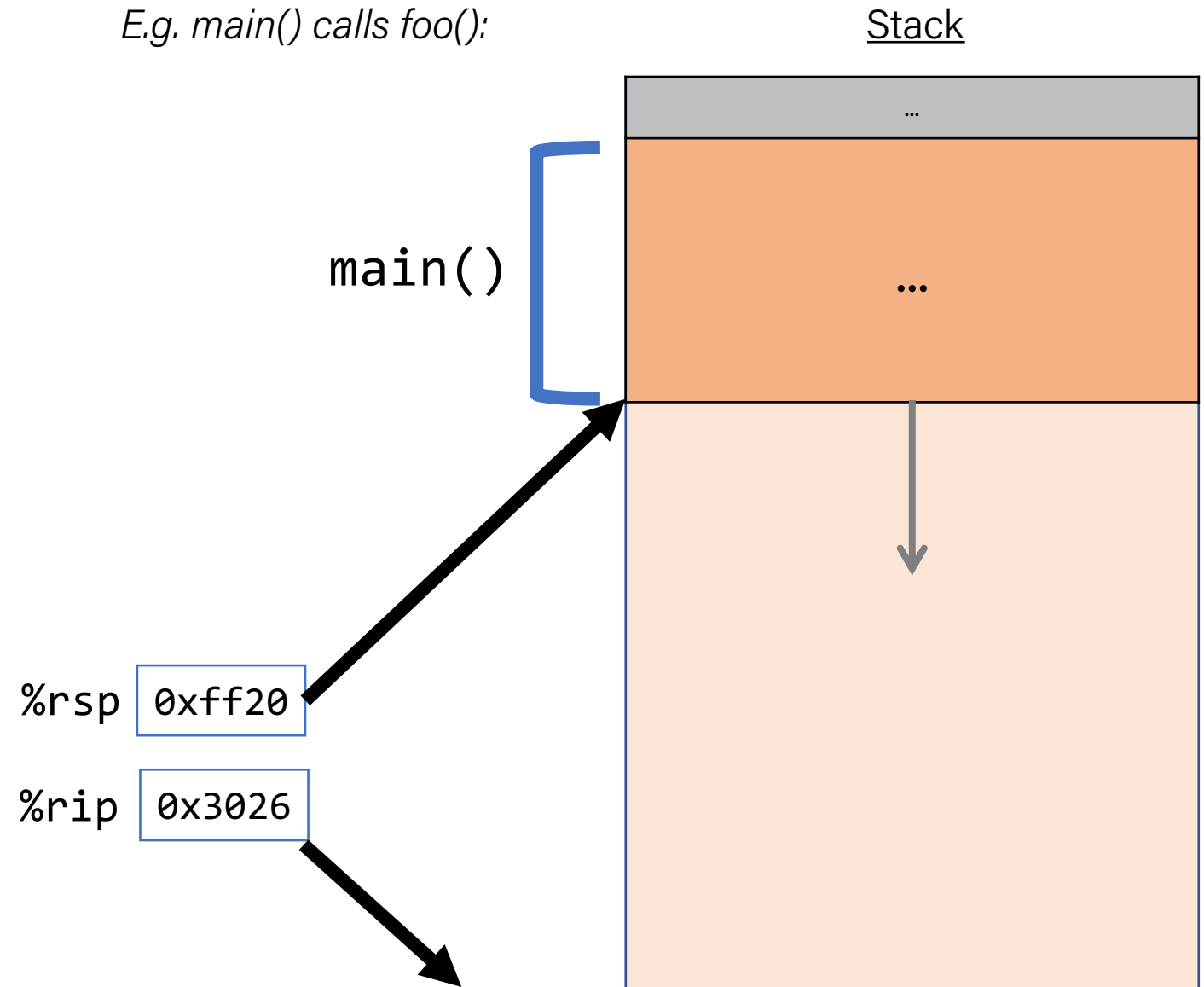
Solution: push the next value of **%rip** onto the stack. Then call the function. When it is finished, put this value back into **%rip** and continue executing.

E.g. main() calls foo():

Stack

main()

...

%rsp `0xff20`

%rip `0x3026`

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
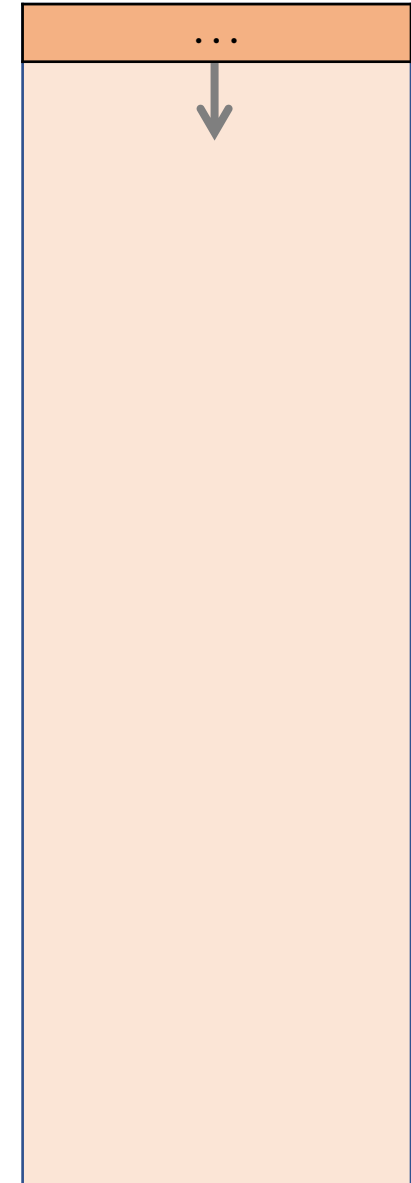
Parameters and Return

- 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.

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()

…

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,(%rsp)
```

main()

…

%rsp
0xffea08

%rip
0x40054f

Parameters and Return

```c
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()

0xffe9f0

%rsp

0xffe9f0

%rip

0x400553

```
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,(%rsp)
```
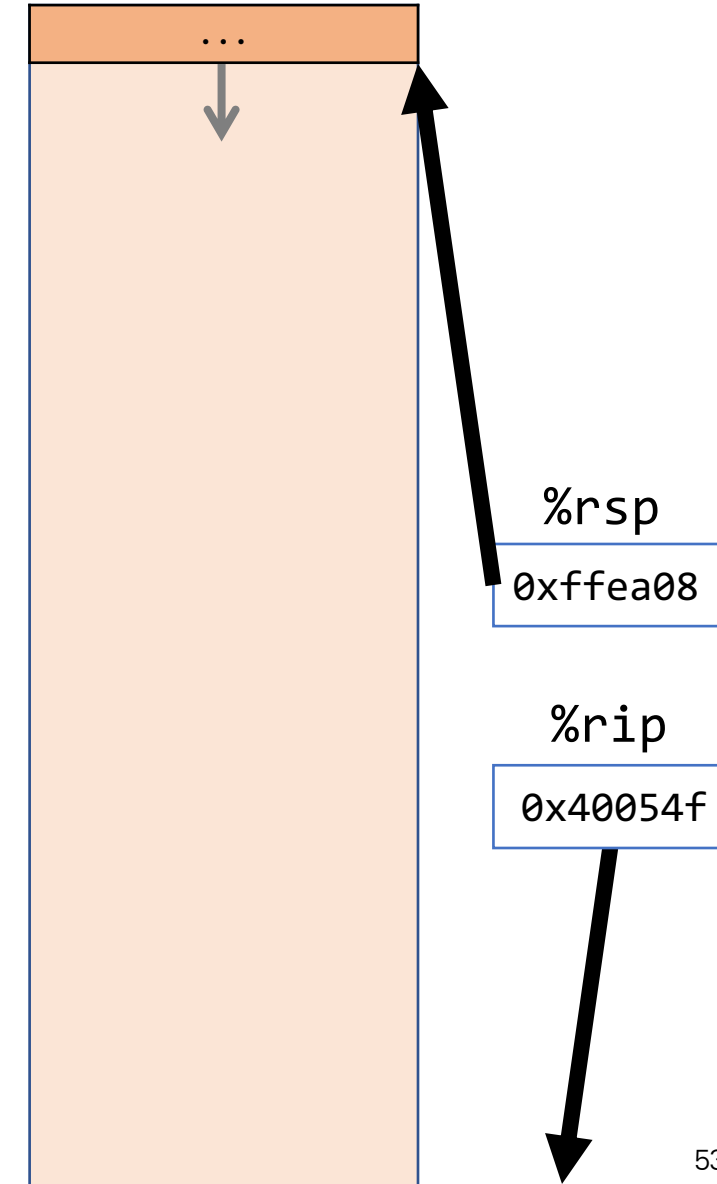
Parameters and Return

```c
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,(%rsp)
```
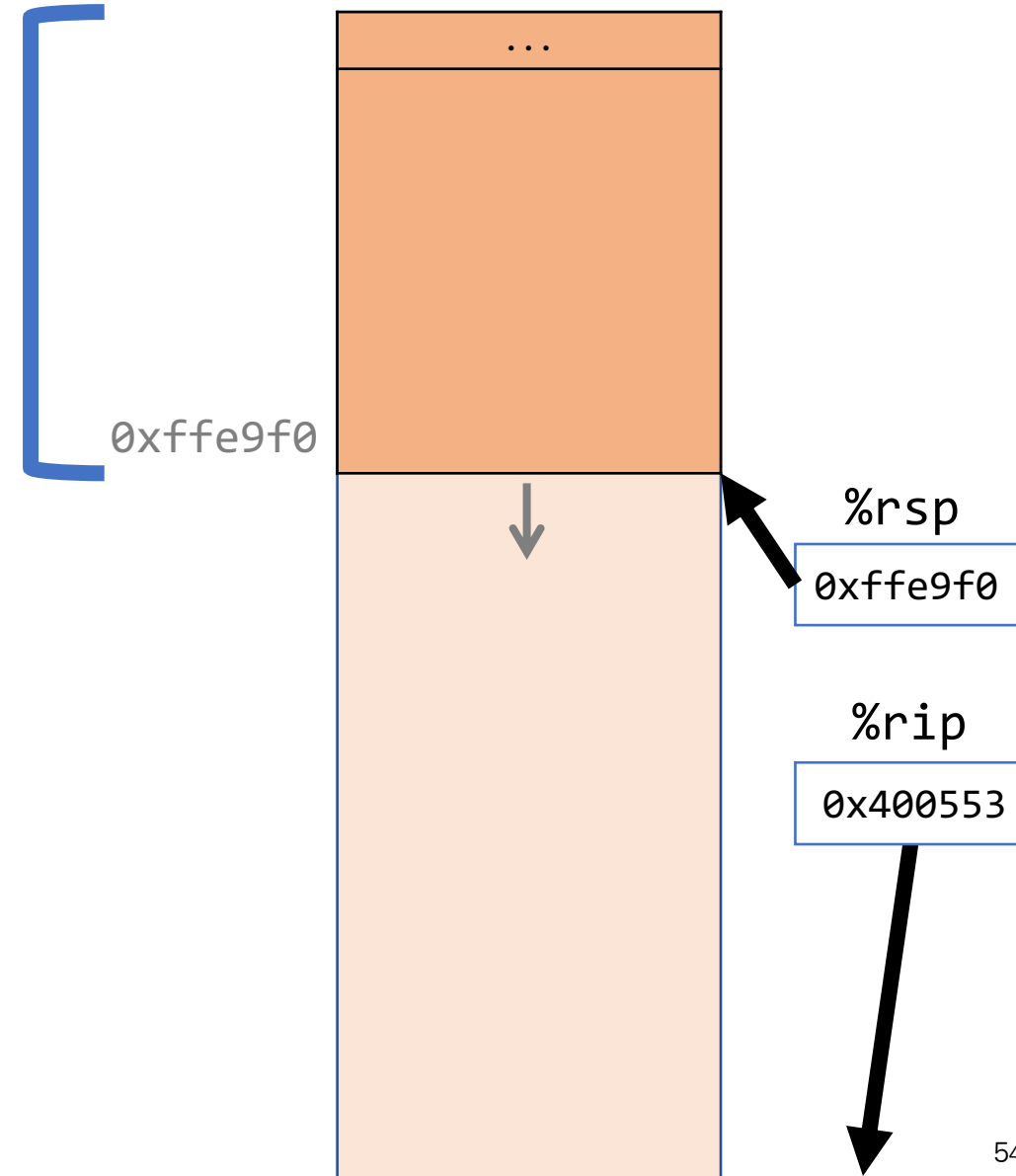
main()

...

1

0xffe9fc

0xffe9f0

%rsp
0xffe9f0

%rip
0x40055b

Parameters and Return

```c
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()

	···
0xffe9fc	1
0xffe9f8	2
0xffe9f0	

%rsp
0xffe9f0

%rip
0x400563

```
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,(%rsp)
```
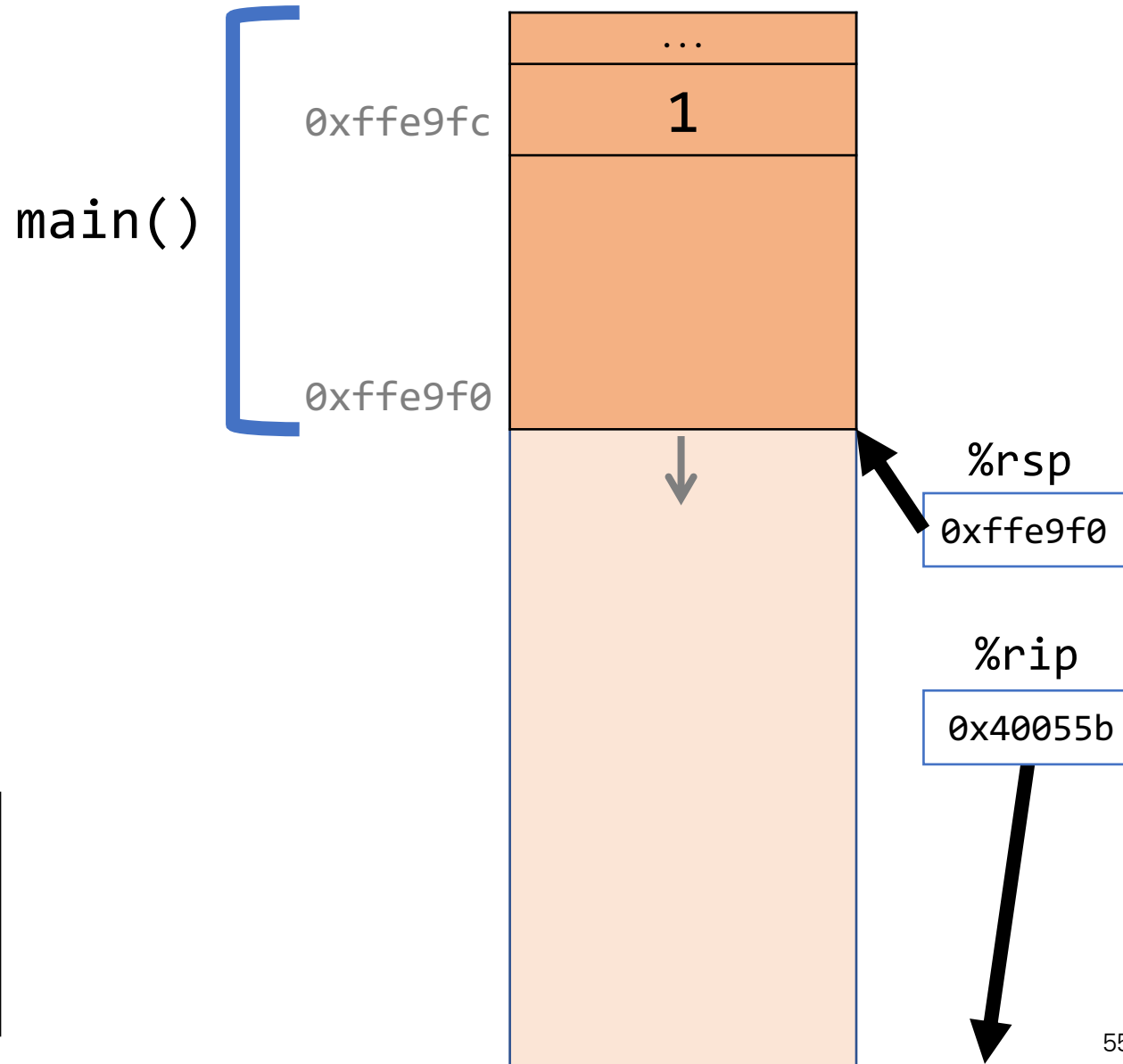
Parameters and Return

```c
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>:    pushq   $0x4
```

main()

	…
0xffe9fc	1
0xffe9f8	2
0xffe9f4	3
0xffe9f0	

%rsp
0xffe9f0

%rip
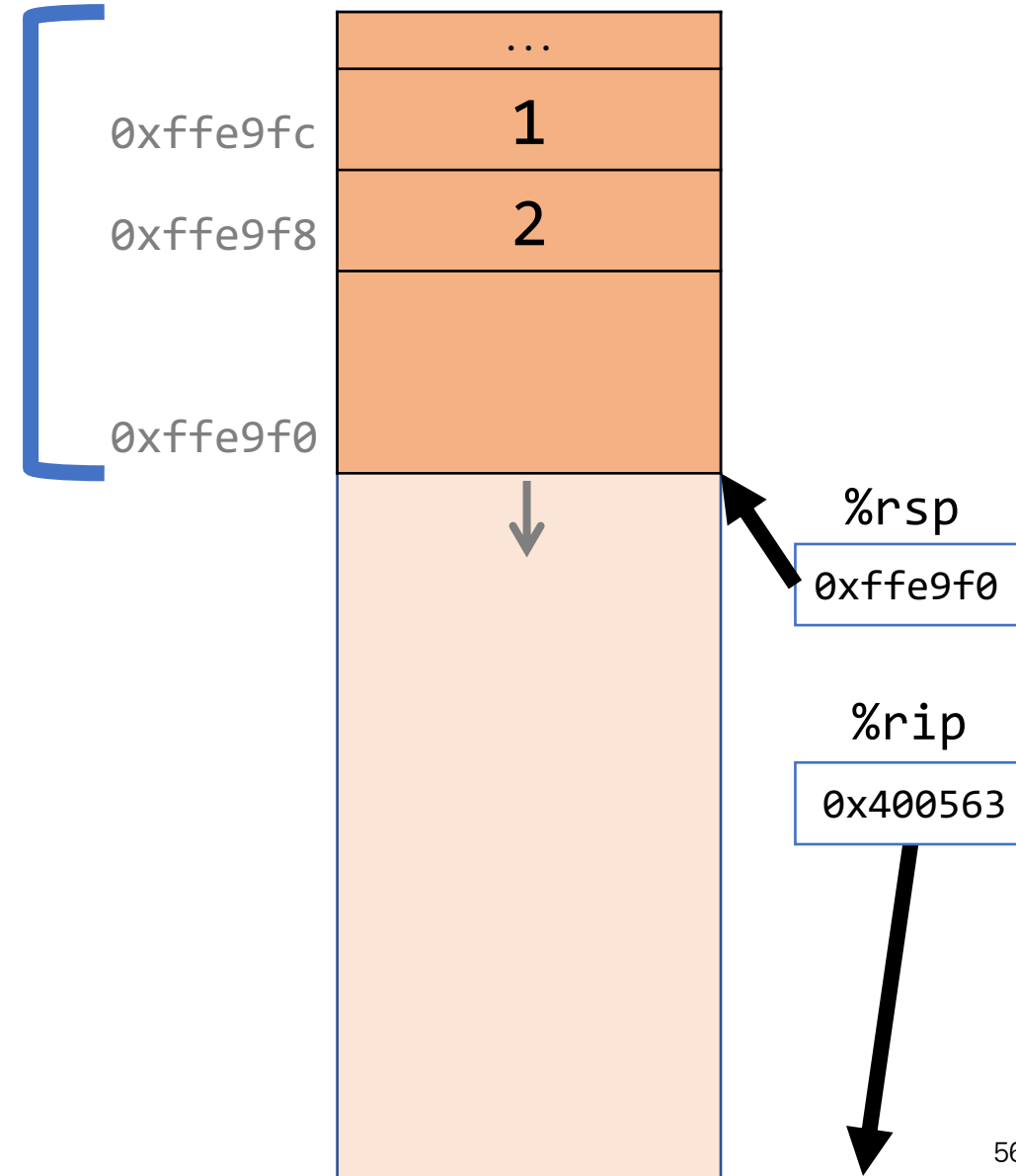0x40056b

Parameters and Return

```c
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   $0x3
```

main()

0xffe9fc	…
	1
0xffe9f8	2
0xffe9f4	3
0xffe9f0	4

%rsp
0xffe9f0

%rip
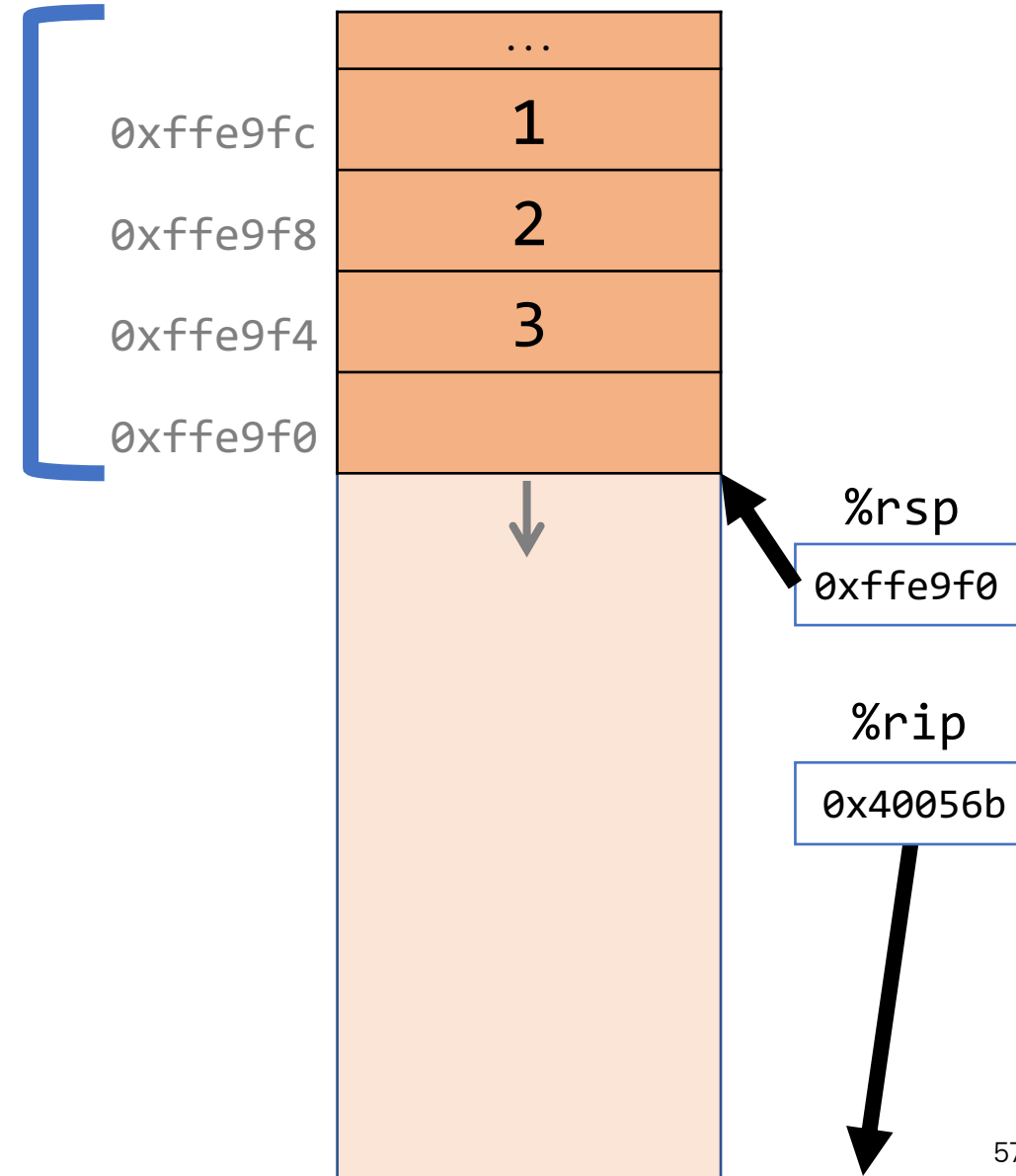0x400572

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()

…
1
2
3
4
4

0xffe9fc
0xffe9f8
0xffe9f4
0xffe9f0
0xffe9e8

%rsp
0xffe9e8

%rip
0x400574

```
0x400563 <+20>:    movl    $0x3,0x4(%rsp)
0x40056b <+28>:    movl    $0x4,(%rsp)
0x400572 <+35>:    pushq   $0x4
0x400574 <+37>:    pushq   $0x3
0x400576 <+39>:    mov     $0x2,%r9d
```
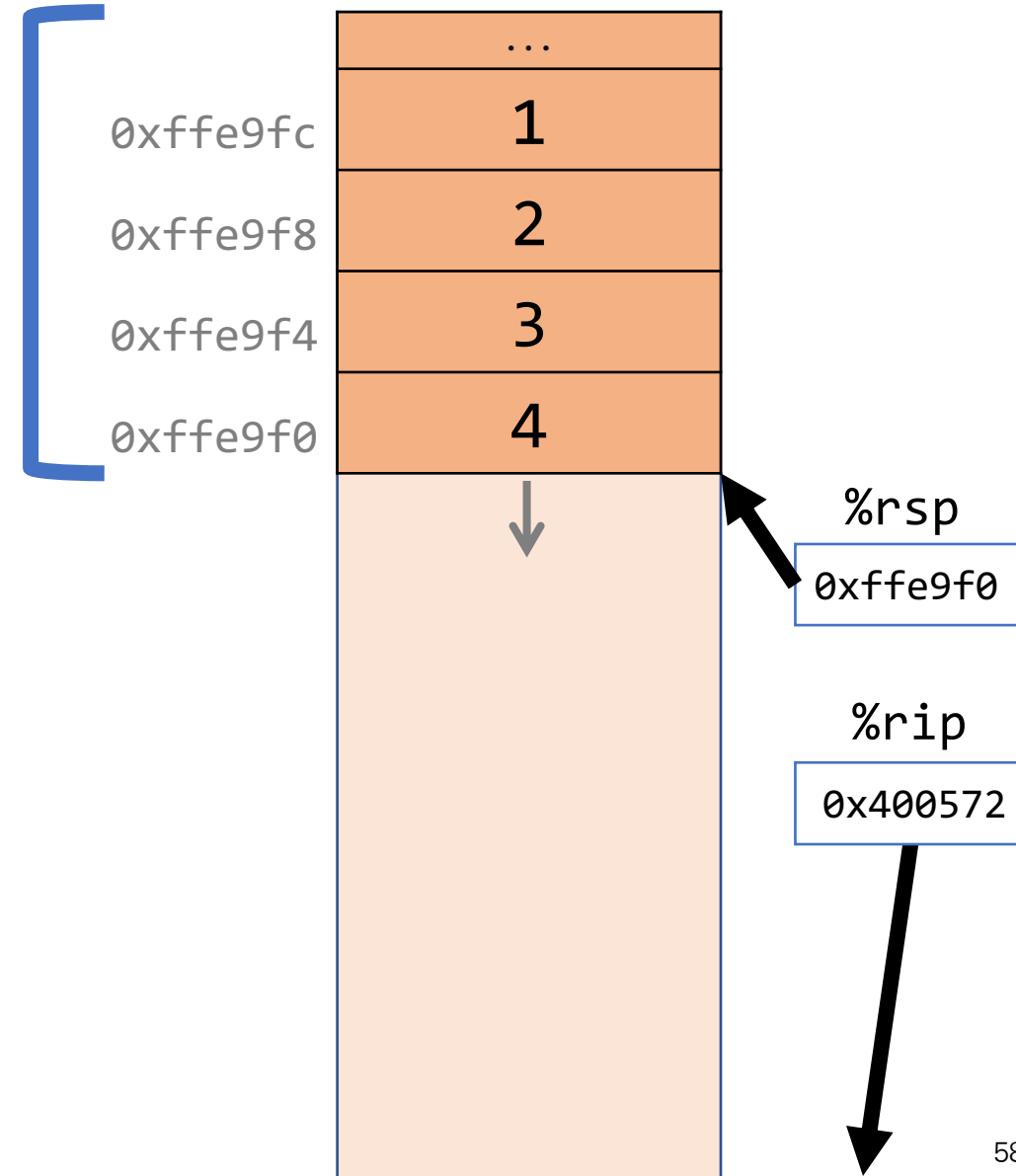
Parameters and Return

```
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()

	…
0xffe9fc	1
0xffe9f8	2
0xffe9f4	3
0xffe9f0	4
0xffe9e8	4
0xffe9e0	3

%rsp

0xffe9e0

%rip

0x400576

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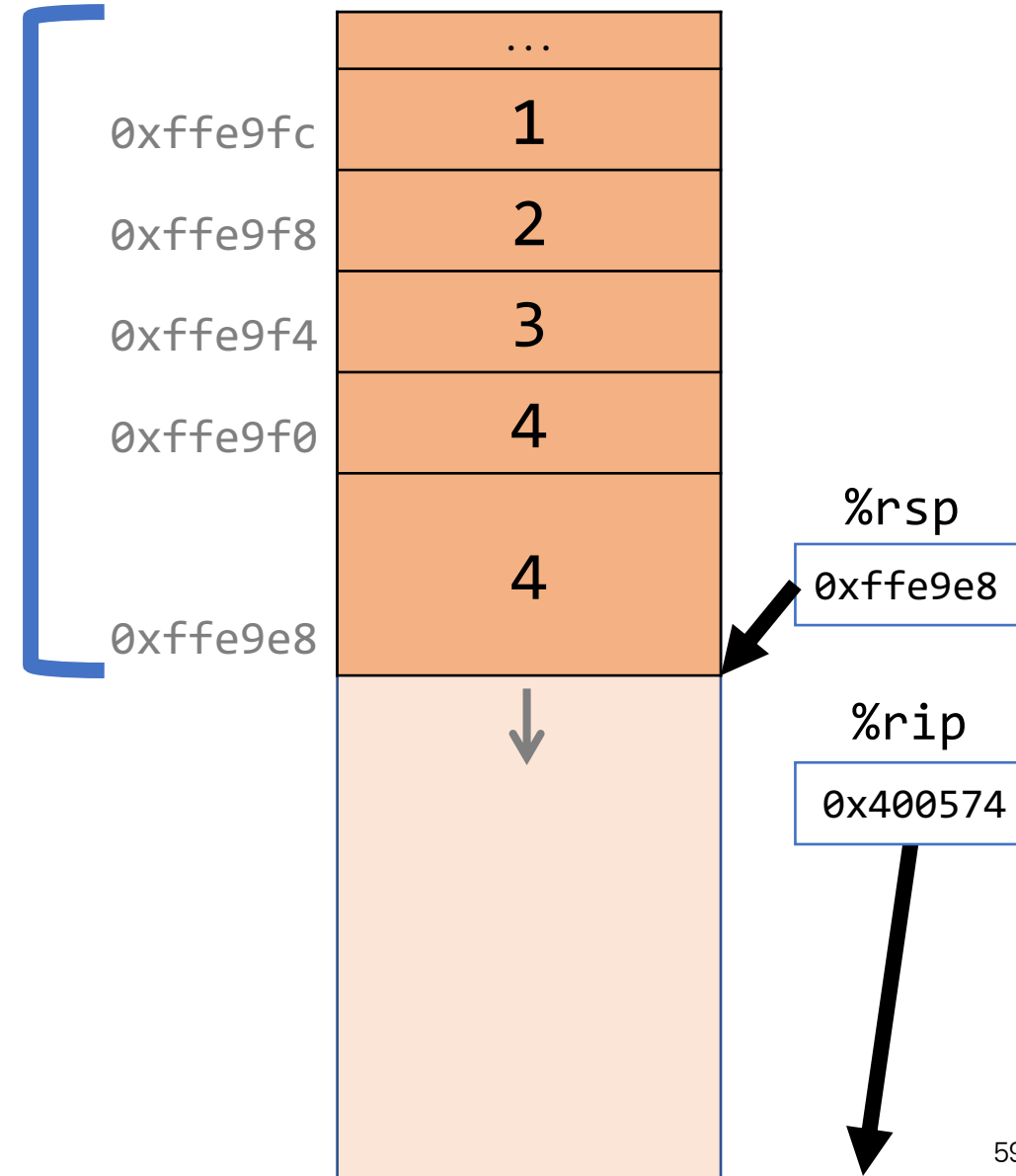
Parameters and Return

```
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()

address	value
	…
0xffe9fc	1
0xffe9f8	2
0xffe9f4	3
0xffe9f0	4
0xffe9e8	4
0xffe9e0	3

%rsp
0xffe9e0

%rip
0x40057c

```
0x400572 <+35>:    pushq   $0x4
0x400574 <+37>:    pushq   $0x3
0x400576 <+39>:    mov     $0x2,%r9d
0x40057c <+45>:    mov     $0x1,%r8d
0x400582 <+51>:    lea     0x10(%rsp),%rcx
```
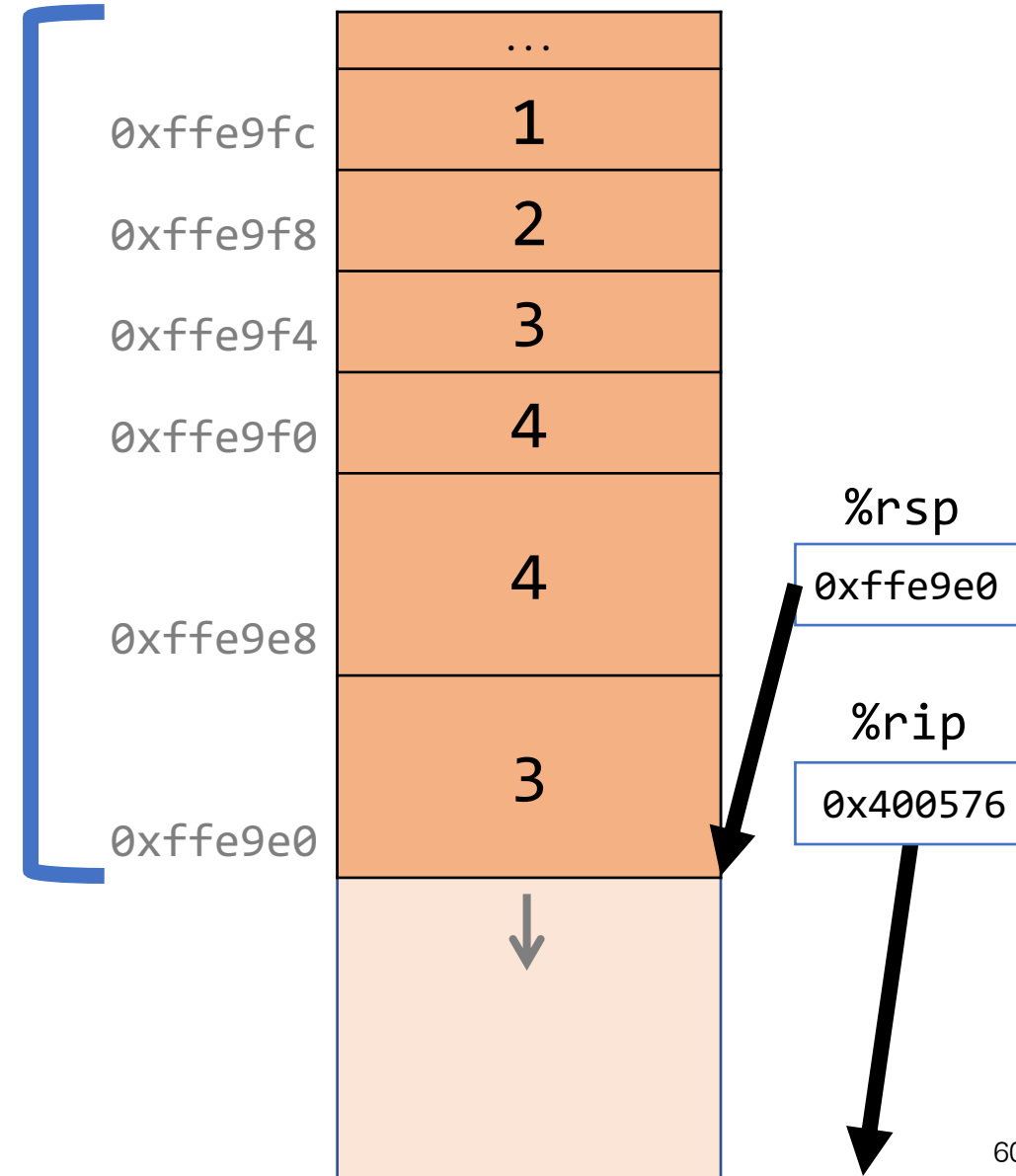
61

Parameters and Return

```c
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()

address	value
	…
0xffe9fc	1
0xffe9f8	2
0xffe9f4	3
0xffe9f0	4
	4
0xffe9e8	
	3
0xffe9e0	

%rsp
0xffe9e0

%rip
0x40057c

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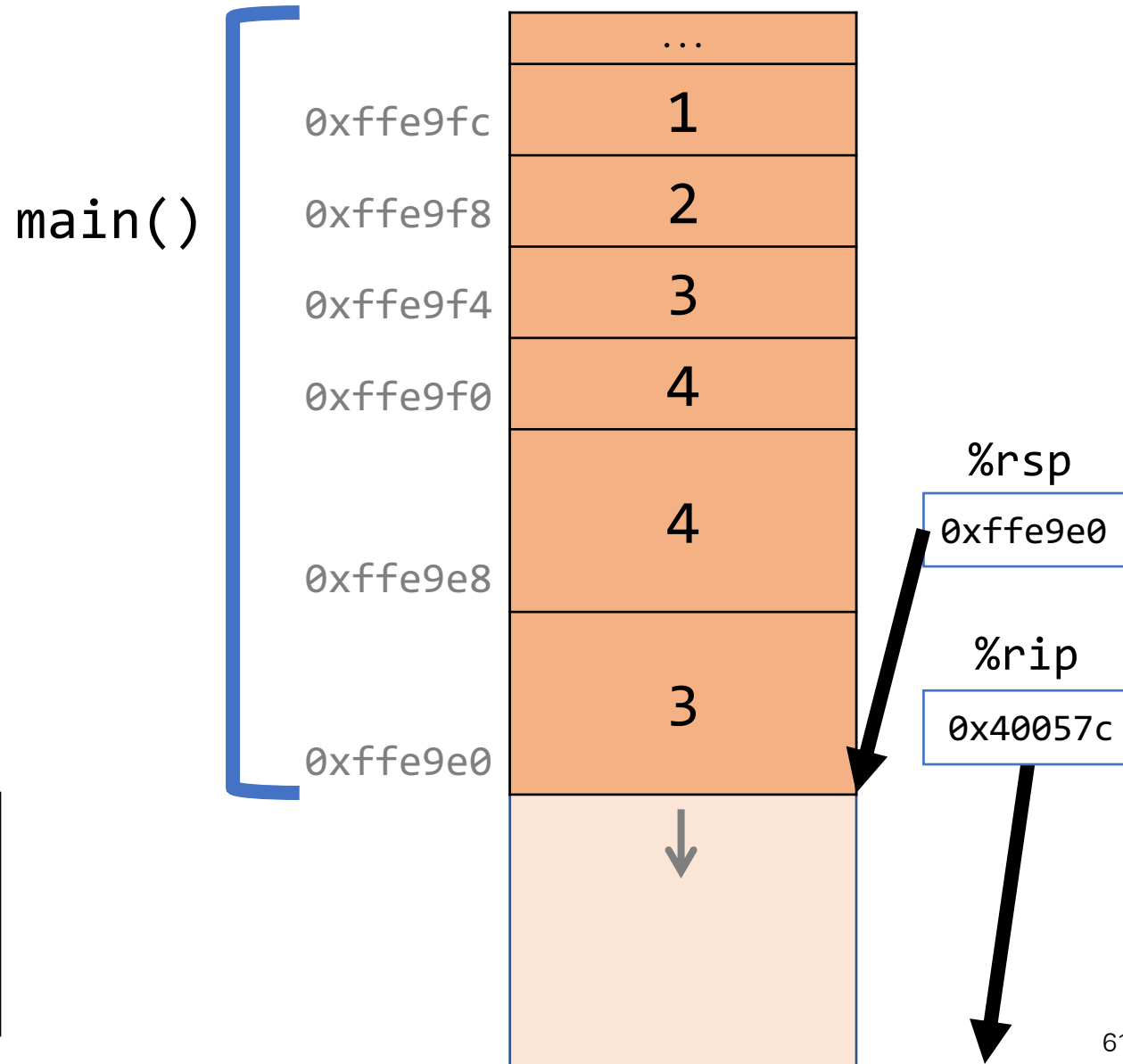
Parameters and Return

```
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>:    lea      0x10(%rsp),%rcx
```

main()

	…
0xffe9fc	1
0xffe9f8	2
0xffe9f4	3
0xffe9f0	4
0xffe9e8	4
0xffe9e0	3

%r9d

2

%rsp

0xffe9e0

%rip

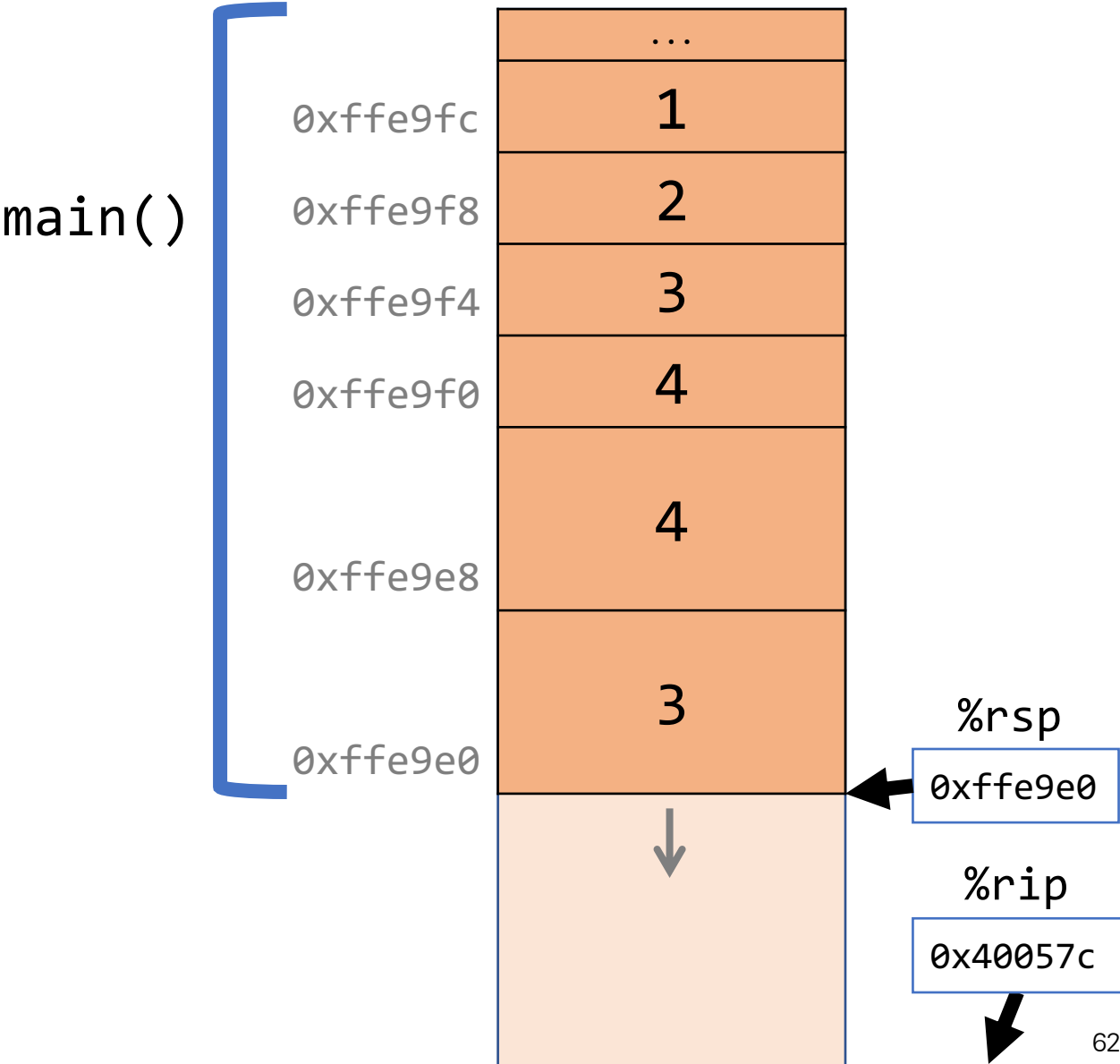0x40057c

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()

	…
0xffe9fc	1
0xffe9f8	2
0xffe9f4	3
0xffe9f0	4
0xffe9e8	4
0xffe9e0	3

%r8d

1

%r9d

2

%rsp

0xffe9e0

%rip

0x400582

```
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
```
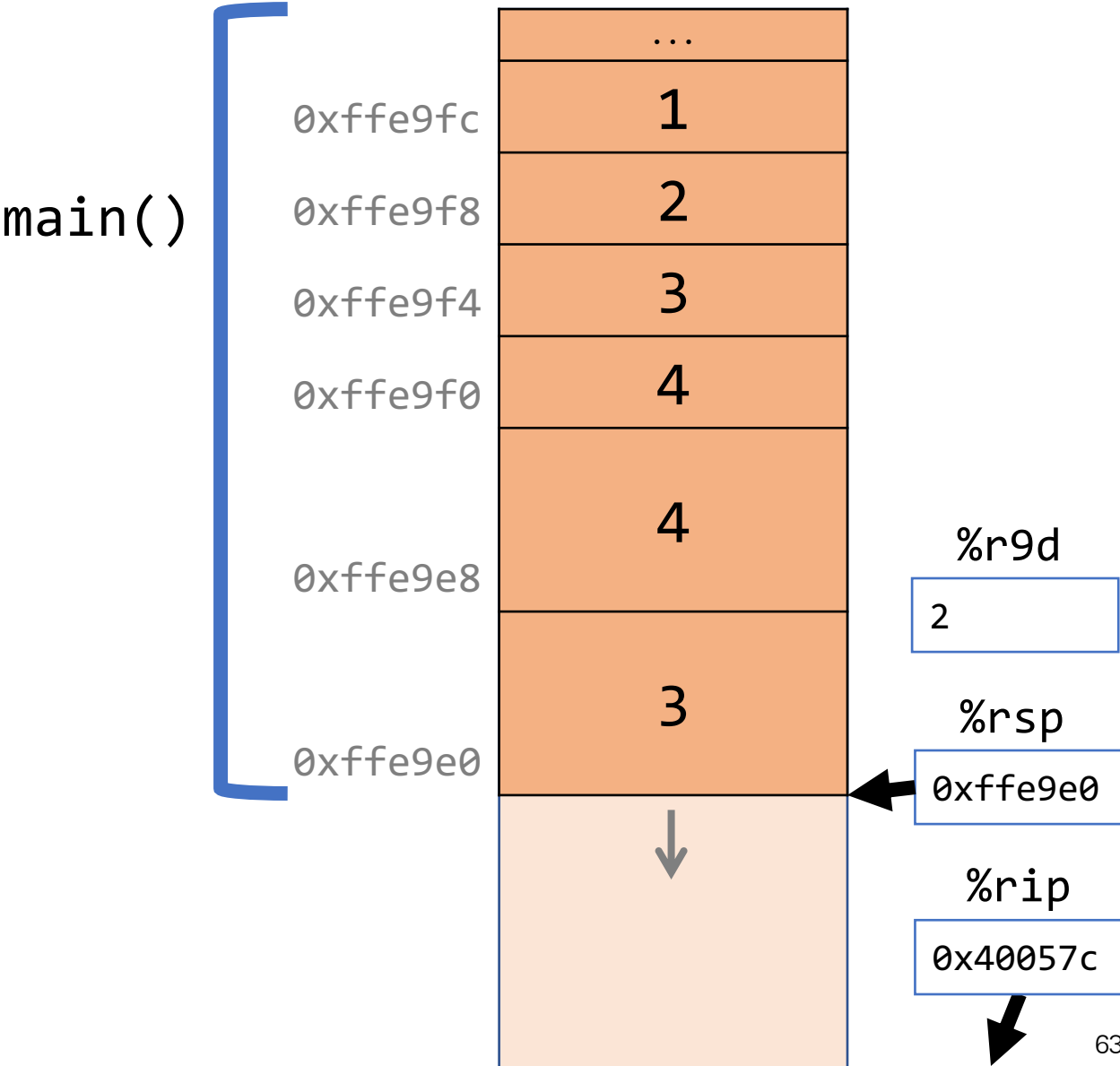
64

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()

address	value
	…
0xffe9fc	1
0xffe9f8	2
0xffe9f4	3
0xffe9f0	4
0xffe9e8	4
0xffe9e0	3

%rcx
0xffe9f0

%r8d
1

%r9d
2

%rsp
0xffe9e0

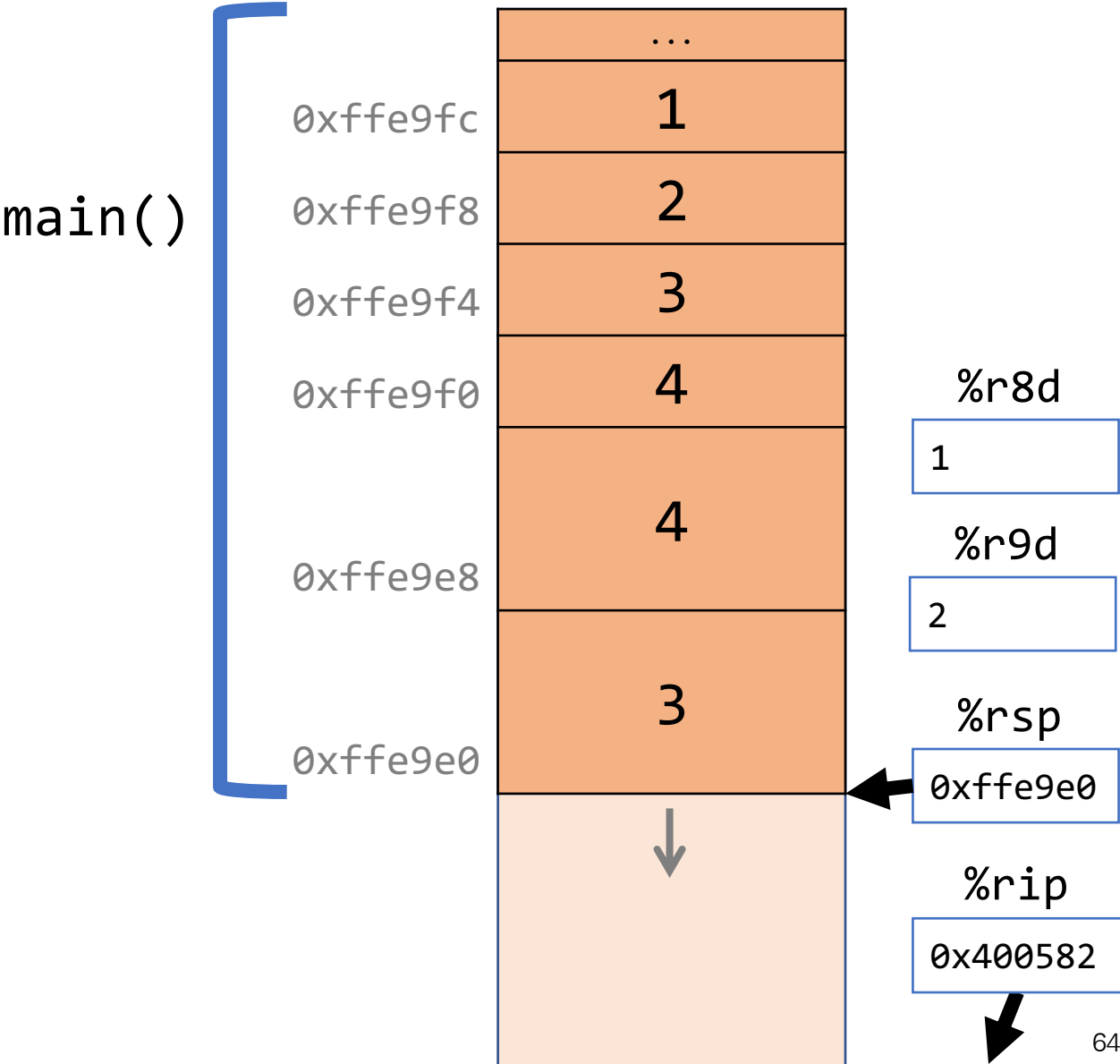%rip
0x400587

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()

0xffe9fc	···
	1
0xffe9f8	2
0xffe9f4	3
0xffe9f0	4
	4
0xffe9e8	
	3
0xffe9e0	

%rdx
0xffe9f4

%rcx
0xffe9f0

%r8d
1

%r9d
2

%rsp
0xffe9e0

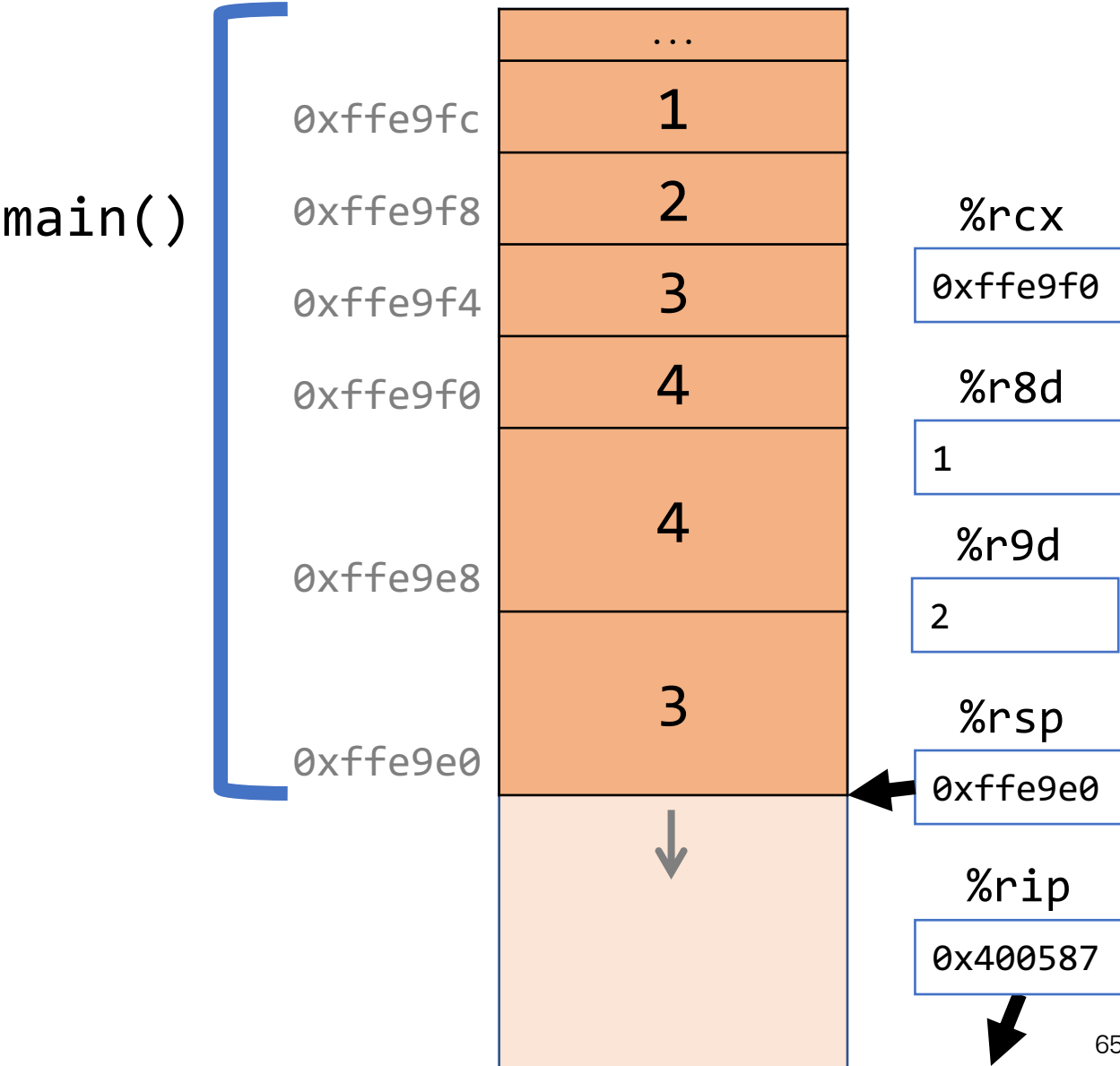%rip
0x40058c

Parameters and Return

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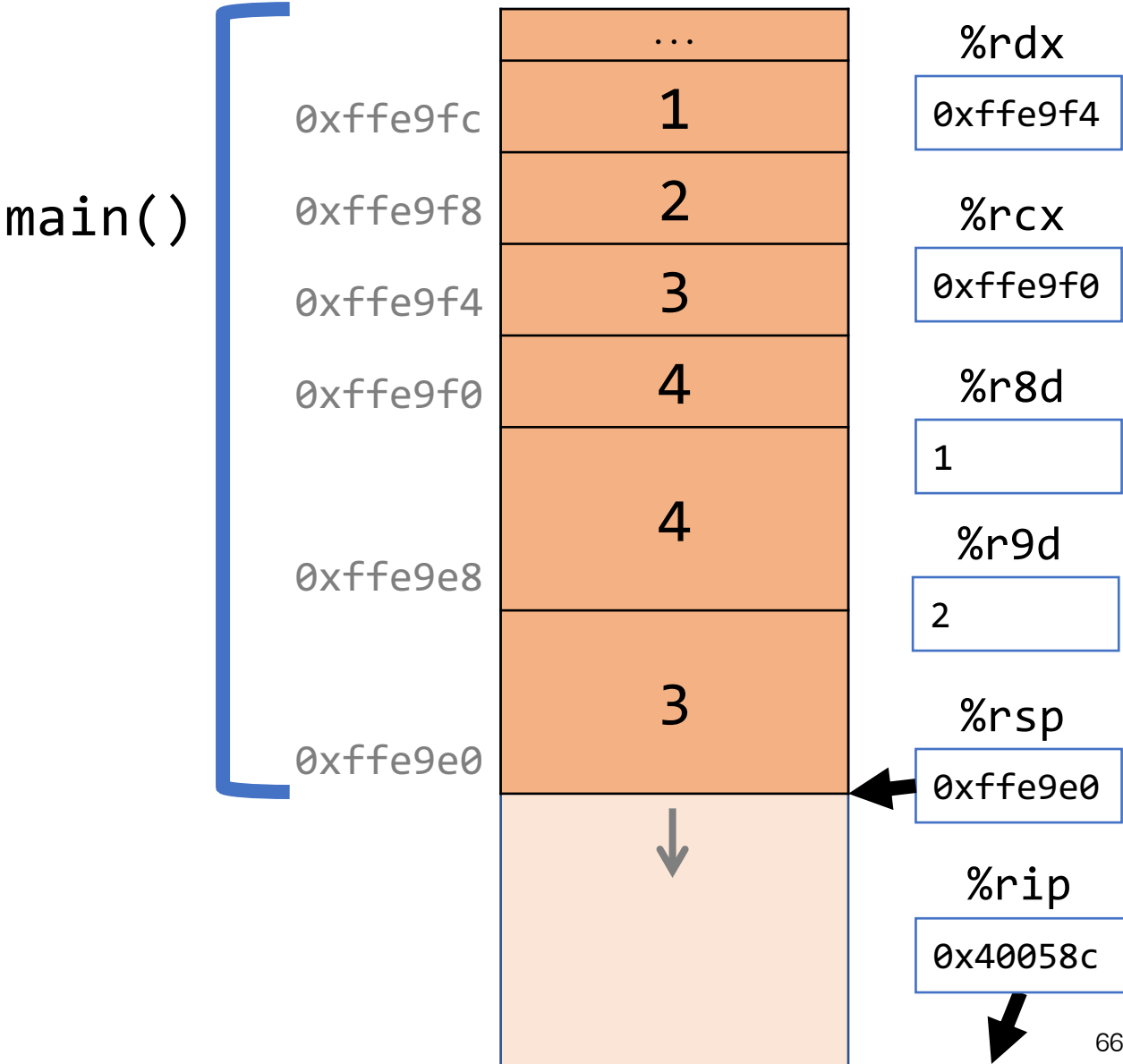
```
0x400582 <+51>:    lea    0x10(%rsp),%rcx
0x400587 <+56>:    lea    0x14(%rsp),%rdx
0x40058c <+61>:    lea    0x18(%rsp),%rsi
0x400591 <+66>:    lea    0x1c(%rsp),%rdi
0x400596 <+71>:    callq  0x400546 <func>
```

main()

0xffe9fc	…
	1
0xffe9f8	2
0xffe9f4	3
0xffe9f0	4
	4
0xffe9e8	
	3
0xffe9e0	

%rdx
0xffe9f4

%rcx
0xffe9f0

%r8d
1

%r9d
2

%rsp
0xffe9e0

%rip
0x400591

%rsi
0xffe9f8

Parameters and Return

```c
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
```

main()

address	value
	…
0xffe9fc	1
0xffe9f8	2
0xffe9f4	3
0xffe9f0	4
	4
0xffe9e8	
	3
0xffe9e0	

%rdx
0xffe9f4

%rcx
0xffe9f0

%r8d
1

%r9d
2

%rsp
0xffe9e0

%rip
0x400596

%rsi
0xffe9f8

%rdi
0xffe9fc

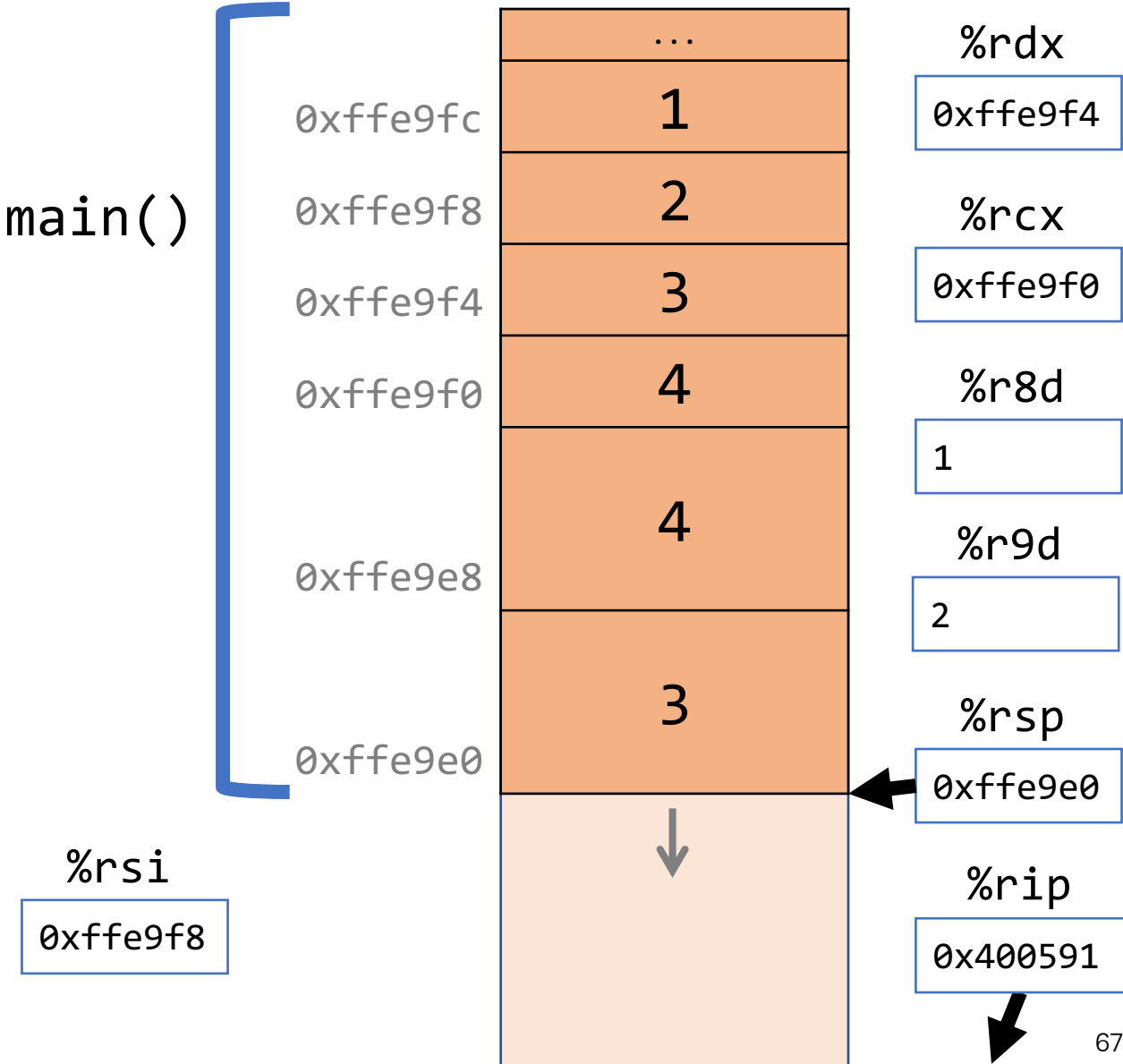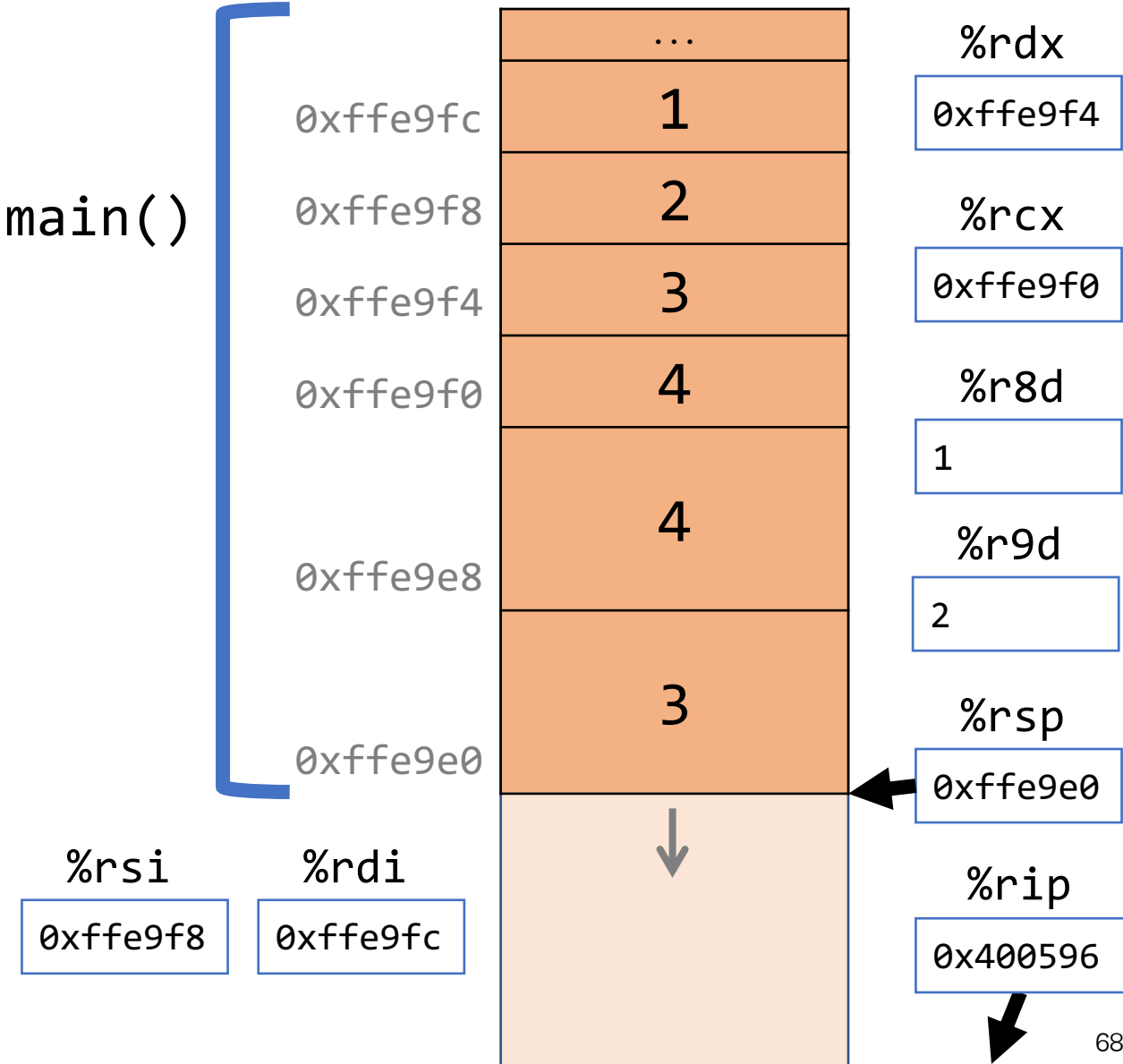# Parameters and Return

```
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
…
```

main()

address	value
	…
0xffe9fc	1
0xffe9f8	2
0xffe9f4	3
0xffe9f0	4
	4
0xffe9e8	
	3
0xffe9e0	

%rdx

0xffe9f4

%rcx

0xffe9f0

%r8d

1

%r9d

2

%rsp

0xffe9e0

%rip

0x400596

%rsi

0xffe9f8

%rdi

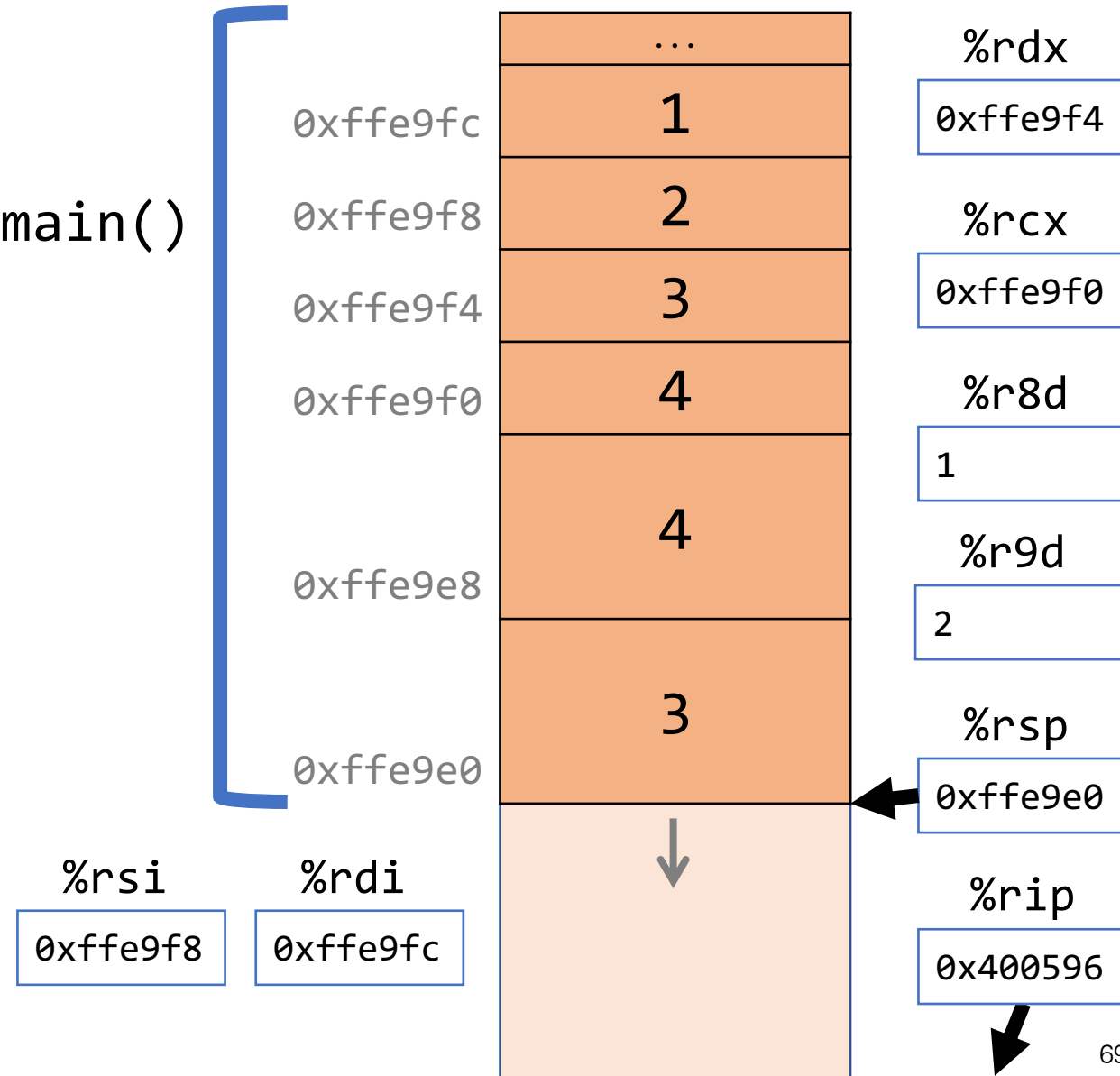0xffe9fc

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
…
```


main()

...	%rdx
1	0xffe9fc 0xffe9f4
2	0xffe9f8 %rcx
3	0xffe9f4 0xffe9f0
4	0xffe9f0 %r8d
	1
4	%r9d
	0xffe9e8 2
3	%rsp
	0xffe9e0 0xffe9d8
0x40059b	%rip

%rsi   %rdi
0xffe9f8   0xffe9fc

0x400596

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

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).

```
main
```

calls

```
function1
```

calls

```
function2
```

`main` is the caller, and `function1` is the callee.

`function1` is the caller, and `function2` is the callee.
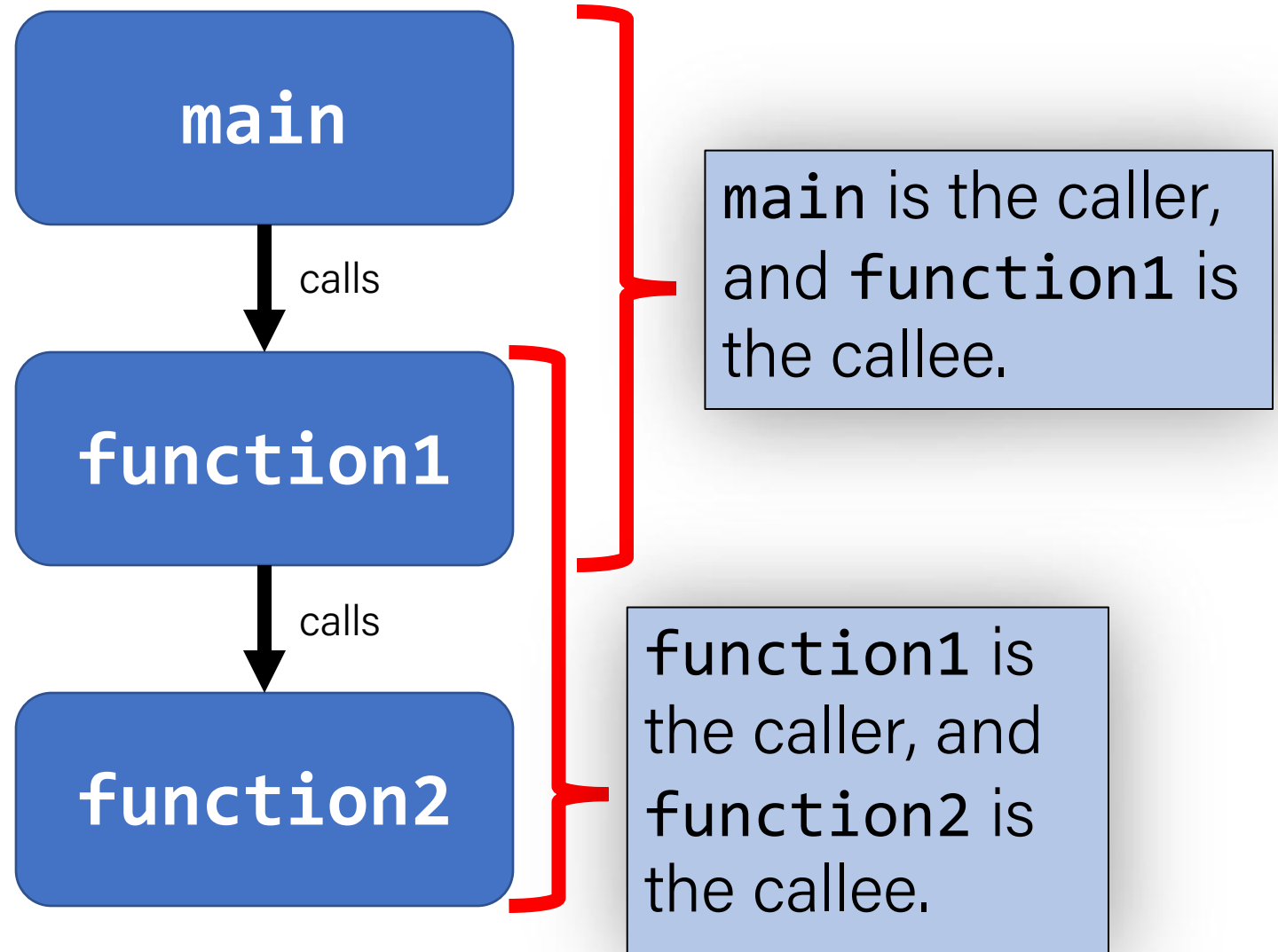
Register Restrictions

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.

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.
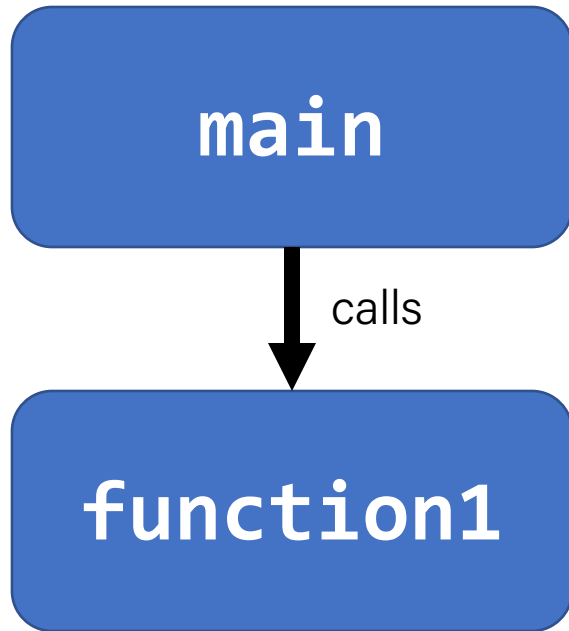
Caller-Owned Registers

main

↓ calls
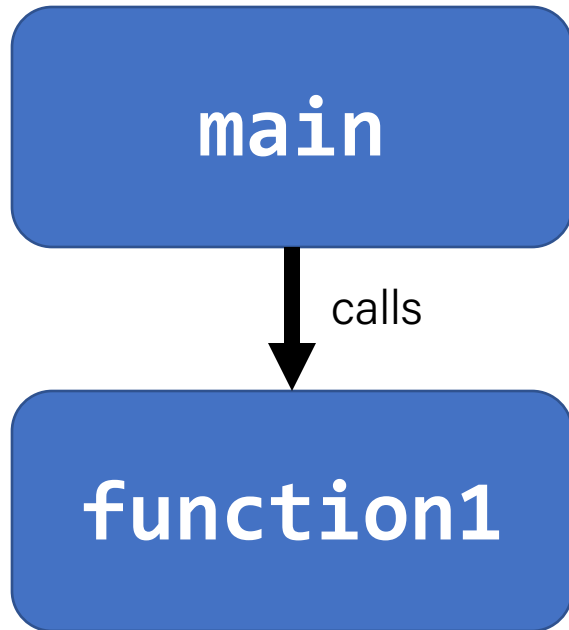
function1

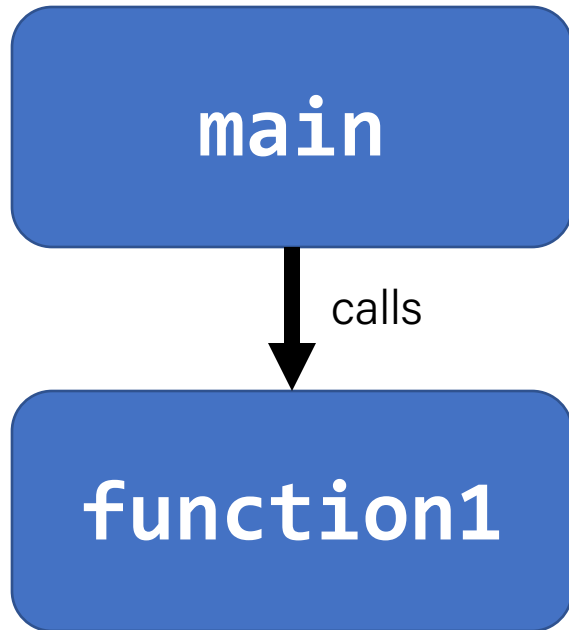`main` can use caller-owned registers and know that `function1` will not permanently modify their values.

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

Caller-Owned Registers

main

calls

function1

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

Callee-Owned Registers

main

↓ calls

function1
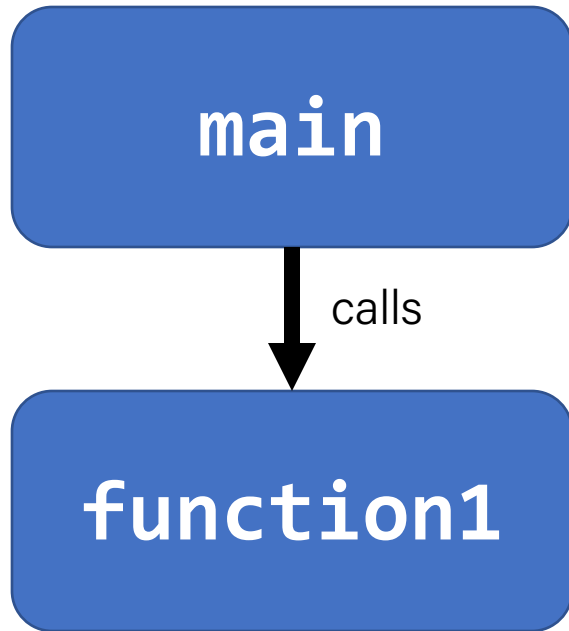
`main` can use callee-owned registers but calling `function1` may permanently modify their values.
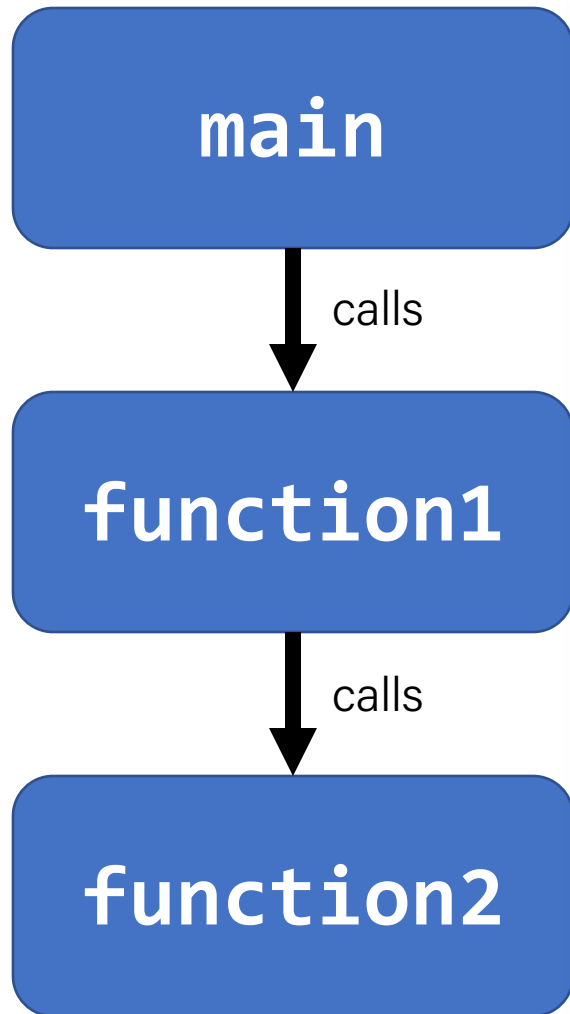
If `function1` wants to use any callee-owned registers, it can do so without saving the existing values.

Callee-Owned Registers


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

main → calls → function1

A Day In the Life of `function1`

main

↓ calls

function1

↓ calls

function2

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

Extra Practice – Escape Room 2

https://godbolt.org/z/jch_D2

`escape_room`

Our First Assembly

```c
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                    jl      4005c2 <sum_array+0xc>
    4005cf:    f3 c3                    repz retq
```

Recap

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

Next time: register restrictions, data stack frames