CS107, Lecture 13

Control Flow: When in doubt just JMP!

Reading: B&O 3.1-3.4

Learning Goals

- Learn about how assembly stores comparison and operation results in condition codes
- Understand how assembly implements loops and control flow

Executing Instructions

What does it mean for a program to execute?

Executing Instructions

So far:

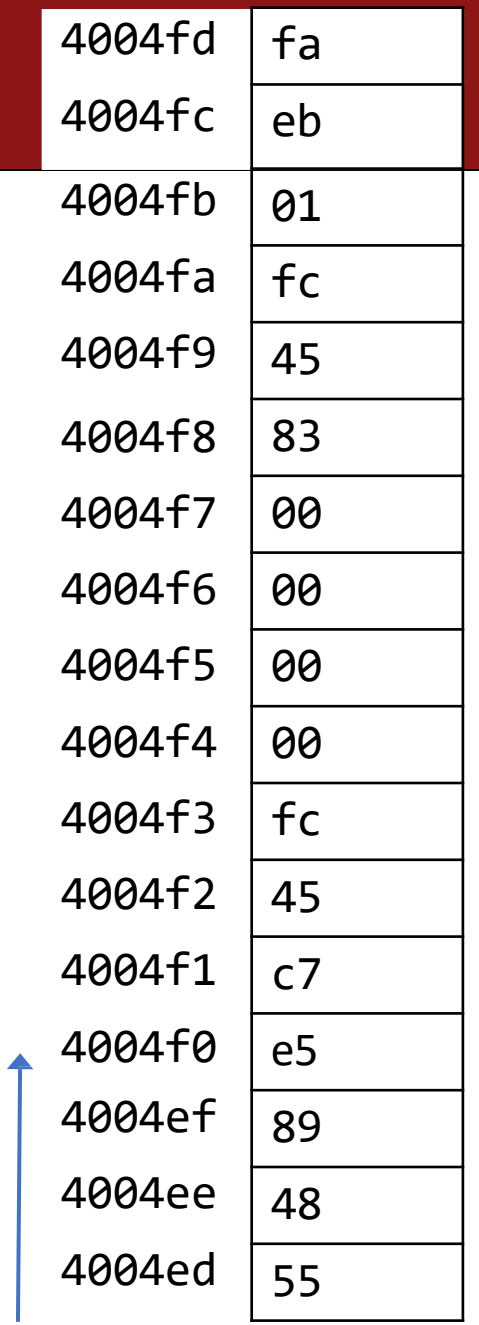
- Program values can be stored in memory or registers.
- Assembly instructions read/write values back and forth between registers (on the CPU) and memory.
- Assembly instructions are also stored in memory.

Today:

- **Who controls the instructions**?
  How do we know what to do now or next?

Answer:

- The **program counter** (PC), %rip.



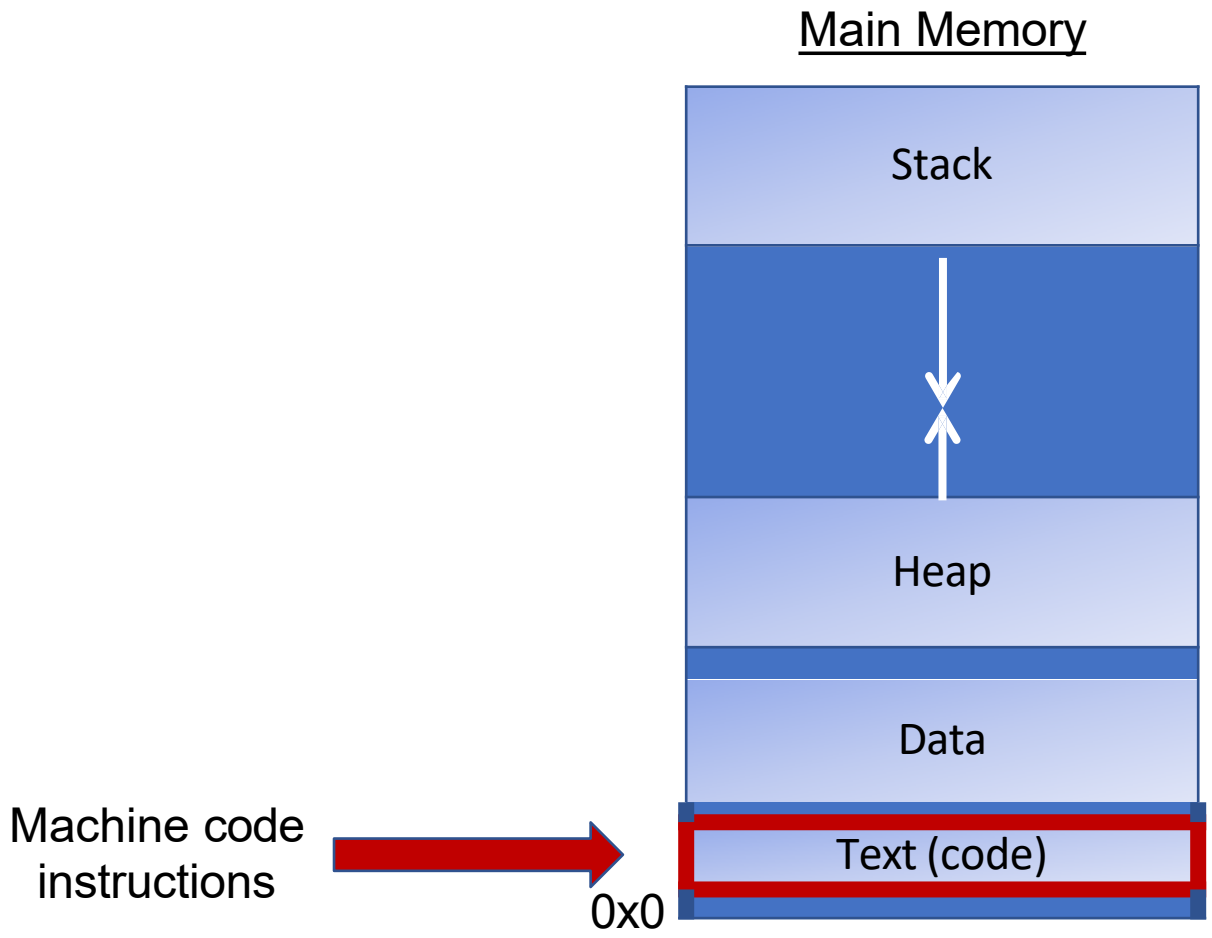
Register Responsibilities

Some registers take on special responsibilities during program execution.

- **%rax** stores the return value
- **%rdi** stores the first parameter to a function
- **%rsi** stores the second parameter to a function
- **%rdx** stores the third parameter to a function
- **%rip** stores the address of the next instruction to execute
- **%rsp** stores the address of the current top of the stack

See the x86-64 Guide and Reference Sheet on the Resources webpage for more!

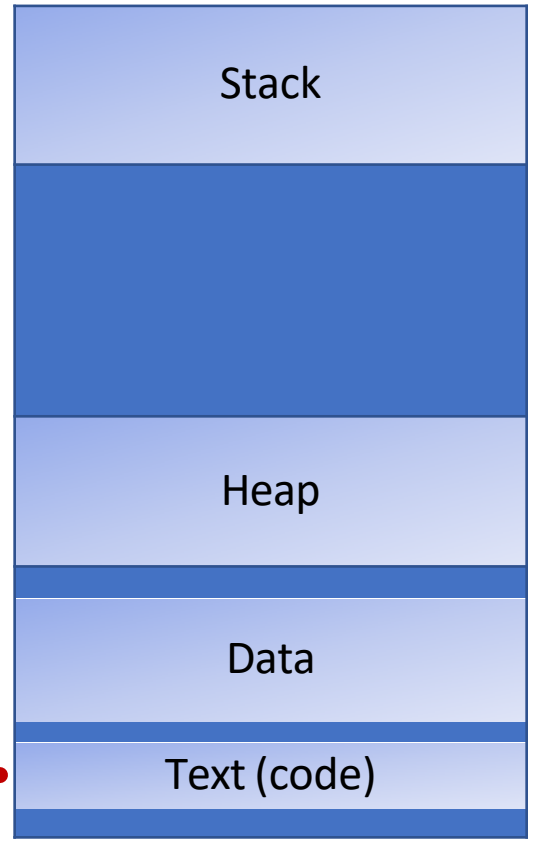
Instructions Are Just Bytes!



%rip

```
00000000004004ed <loop>:
4004ed: 55                              push   %rbp
4004ee: 48 89 e5                        mov    %rsp,%rbp
4004f1: c7 45 fc 00 00 00 00            movl   $0x0,-0x4(%rbp)
4004f8: 83 45 fc 01                     addl   $0x1,-0x4(%rbp)
4004fc: eb fa                           jmp    4004f8 <loop+0xb>
```

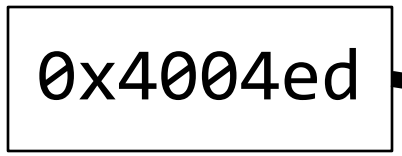
4004fd	fa
4004fc	eb
4004fb	01
4004fa	fc
4004f9	45
4004f8	83
4004f7	00
4004f6	00
4004f5	00
4004f4	00
4004f3	fc
4004f2	45
4004f1	c7
4004f0	e5
4004ef	89
4004ee	48
4004ed	55



%rip

```
0000000004004ed <loop>:
4004ed:  55                    push   %rbp
4004ee:  48 89 e5              mov    %rsp,%rbp
4004f1:  c7 45 fc 00 00 00 00  movl   $0x0,-0x4(%rbp)
4004f8:  83 45 fc 01           addl   $0x1,-0x4(%rbp)
4004fc:  eb fa                 jmp    4004f8 <loop+0xb>
```

The **program counter** (PC), known as %rip in x86-64, stores the address in memory of the ***next instruction*** to be executed.

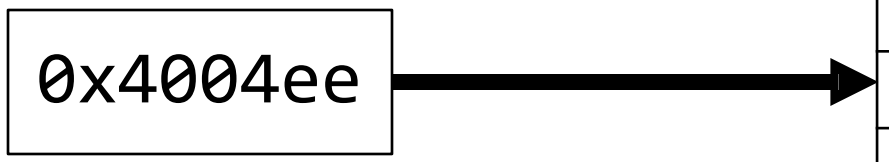


4004fd	fa
4004fc	eb
4004fb	01
4004fa	fc
4004f9	45
4004f8	83
4004f7	00
4004f6	00
4004f5	00
4004f4	00
4004f3	fc
4004f2	45
4004f1	c7
4004f0	e5
4004ef	89
4004ee	48
4004ed	55

%rip

```
0000000004004ed <loop>:
4004ed:  55                              push   %rbp
4004ee:  48 89 e5                        mov    %rsp,%rbp
4004f1:  c7 45 fc 00 00 00 00            movl   $0x0,-0x4(%rbp)
4004f8:  83 45 fc 01                     addl   $0x1,-0x4(%rbp)
4004fc:  eb fa                           jmp    4004f8 <loop+0xb>
```

The **program counter** (PC), known as %rip in x86-64, stores the address in memory of the ***next instruction*** to be executed.

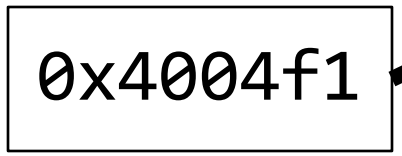


4004fd	fa
4004fc	eb
4004fb	01
4004fa	fc
4004f9	45
4004f8	83
4004f7	00
4004f6	00
4004f5	00
4004f4	00
4004f3	fc
4004f2	45
4004f1	c7
4004f0	e5
4004ef	89
4004ee	48
4004ed	55

%rip

```
0000000004004ed <loop>:
4004ed:  55                          push   %rbp
4004ee:  48 89 e5                    mov    %rsp,%rbp
4004f1:  c7 45 fc 00 00 00 00        movl   $0x0,-0x4(%rbp)
4004f8:  83 45 fc 01                 addl   $0x1,-0x4(%rbp)
4004fc:  eb fa                       jmp    4004f8 <loop+0xb>
```

The **program counter** (PC), known as %rip in x86-64, stores the address in memory of the ***next instruction*** to be executed.

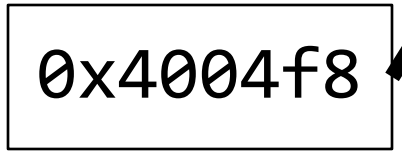


4004fd	fa
4004fc	eb
4004fb	01
4004fa	fc
4004f9	45
4004f8	83
4004f7	00
4004f6	00
4004f5	00
4004f4	00
4004f3	fc
4004f2	45
4004f1	c7
4004f0	e5
4004ef	89
4004ee	48
4004ed	55

%rip

```
0000000004004ed <loop>:
4004ed:  55                              push   %rbp
4004ee:  48 89 e5                        mov    %rsp,%rbp
4004f1:  c7 45 fc 00 00 00 00            movl   $0x0,-0x4(%rbp)
4004f8:  83 45 fc 01                     addl   $0x1,-0x4(%rbp)
4004fc:  eb fa                           jmp    4004f8 <loop+0xb>
```

The **program counter** (PC), known as %rip in x86-64, stores the address in memory of the ***next instruction*** to be executed.

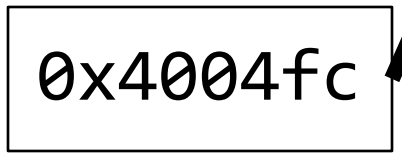


4004fd	fa
4004fc	eb
4004fb	01
4004fa	fc
4004f9	45
4004f8	83
4004f7	00
4004f6	00
4004f5	00
4004f4	00
4004f3	fc
4004f2	45
4004f1	c7
4004f0	e5
4004ef	89
4004ee	48
4004ed	55

%rip

```
0000000004004ed <loop>:
4004ed: 55                              push   %rbp
4004ee: 48 89 e5                        mov    %rsp,%rbp
4004f1: c7 45 fc 00 00 00 00            movl   $0x0,-0x4(%rbp)
4004f8: 83 45 fc 01                     addl   $0x1,-0x4(%rbp)
4004fc: eb fa                           jmp    4004f8 <loop+0xb>
```

The **program counter** (PC), known as %rip in x86-64, stores the address in memory of the ***next instruction*** to be executed.



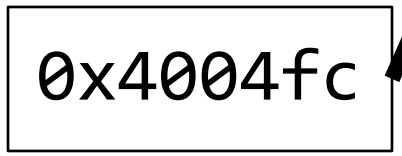
4004fd	fa
4004fc	eb
4004fb	01
4004fa	fc
4004f9	45
4004f8	83
4004f7	00
4004f6	00
4004f5	00
4004f4	00
4004f3	fc
4004f2	45
4004f1	c7
4004f0	e5
4004ef	89
4004ee	48
4004ed	55

%rip

```
0000000004004ed <loop>:
4004ed: 55                          push   %rbp
4004ee: 48 89 e5                    mov    %rsp,%rbp
4004f1: c7 45 fc 00 00 00 00        movl   $0x0,-0x4(%rbp)
4004f8: 83 45 fc 01                 addl   $0x1,-0x4(%rbp)
4004fc: eb fa                       jmp    4004f8 <loop+0xb>
```

Special hardware sets the program counter to the next instruction:

%rip += size of bytes of current instruction



%rip

4004fd	fa
4004fc	eb
4004fb	01
4004fa	fc
4004f9	45
4004f8	83
4004f7	00
4004f6	00
4004f5	00
4004f4	00
4004f3	fc
4004f2	45
4004f1	c7
4004f0	e5
4004ef	89
4004ee	48
4004ed	55

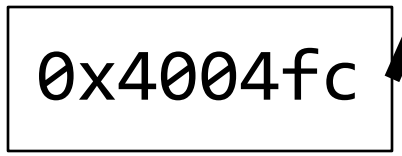
Going In Circles

- How can we use this representation of execution to represent e.g. a **loop**?
- **Key Idea:** we can ”interfere” with **%rip** and set it back to an earlier instruction!

Jump!

```
0000000004004ed <loop>:
4004ed: 55                              push   %rbp
4004ee: 48 89 e5                        mov    %rsp,%rbp
4004f1: c7 45 fc 00 00 00 00            movl   $0x0,-0x4(%rbp)
4004f8: 83 45 fc 01                     addl   $0x1,-0x4(%rbp)
4004fc: eb fa                           jmp    4004f8 <loop+0xb>
```

The **jmp** instruction is an **unconditional jump** that sets the program counter to the **jump target** (the operand).



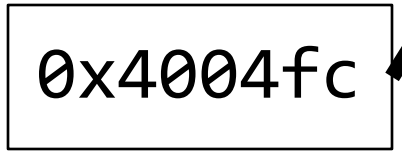
%rip

4004fd	fa
4004fc	eb
4004fb	01
4004fa	fc
4004f9	45
4004f8	83
4004f7	00
4004f6	00
4004f5	00
4004f4	00
4004f3	fc
4004f2	45
4004f1	c7
4004f0	e5
4004ef	89
4004ee	48
4004ed	55

Jump!

```
0000000004004ed <loop>:
4004ed:  55                              push   %rbp
4004ee:  48 89 e5                        mov    %rsp,%rbp
4004f1:  c7 45 fc 00 00 00 00            movl   $0x0,-0x4(%rbp)
4004f8:  83 45 fc 01                     addl   $0x1,-0x4(%rbp)
4004fc:  eb fa                           jmp    4004f8 <loop+0xb>
```

The **jmp** instruction is an **unconditional jump** that sets the program counter to the **jump target** (the operand).

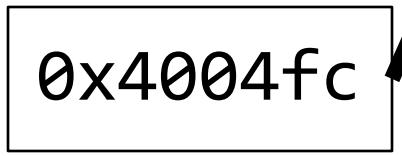


4004fd	fa
4004fc	eb
4004fb	01
4004fa	fc
4004f9	45
4004f8	83
4004f7	00
4004f6	00
4004f5	00
4004f4	00
4004f3	fc
4004f2	45
4004f1	c7
4004f0	e5
4004ef	89
4004ee	48
4004ed	55

Jump!

```
0000000004004ed <loop>:
4004ed: 55                              push   %rbp
4004ee: 48 89 e5                        mov    %rsp,%rbp
4004f1: c7 45 fc 00 00 00 00            movl   $0x0,-0x4(%rbp)
4004f8: 83 45 fc 01                     addl   $0x1,-0x4(%rbp)
4004fc: eb fa                           jmp    4004f8 <loop+0xb>
```

The **jmp** instruction is an **unconditional jump** that sets the program counter to the **jump target** (the operand).



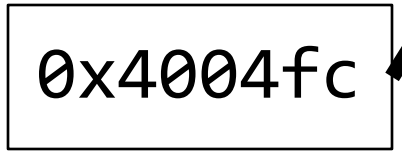
%rip

4004fd	fa
4004fc	eb
4004fb	01
4004fa	fc
4004f9	45
4004f8	83
4004f7	00
4004f6	00
4004f5	00
4004f4	00
4004f3	fc
4004f2	45
4004f1	c7
4004f0	e5
4004ef	89
4004ee	48
4004ed	55

Jump!

```
00000000004004ed <loop>:
4004ed: 55                              push   %rbp
4004ee: 48 89 e5                        mov    %rsp,%rbp
4004f1: c7 45 fc 00 00 00 00            movl   $0x0,-0x4(%rbp)
4004f8: 83 45 fc 01                     addl   $0x1,-0x4(%rbp)
4004fc: eb fa                           jmp    4004f8 <loop+0xb>
```

The **jmp** instruction is an **unconditional jump** that sets the program counter to the **jump target** (the operand).



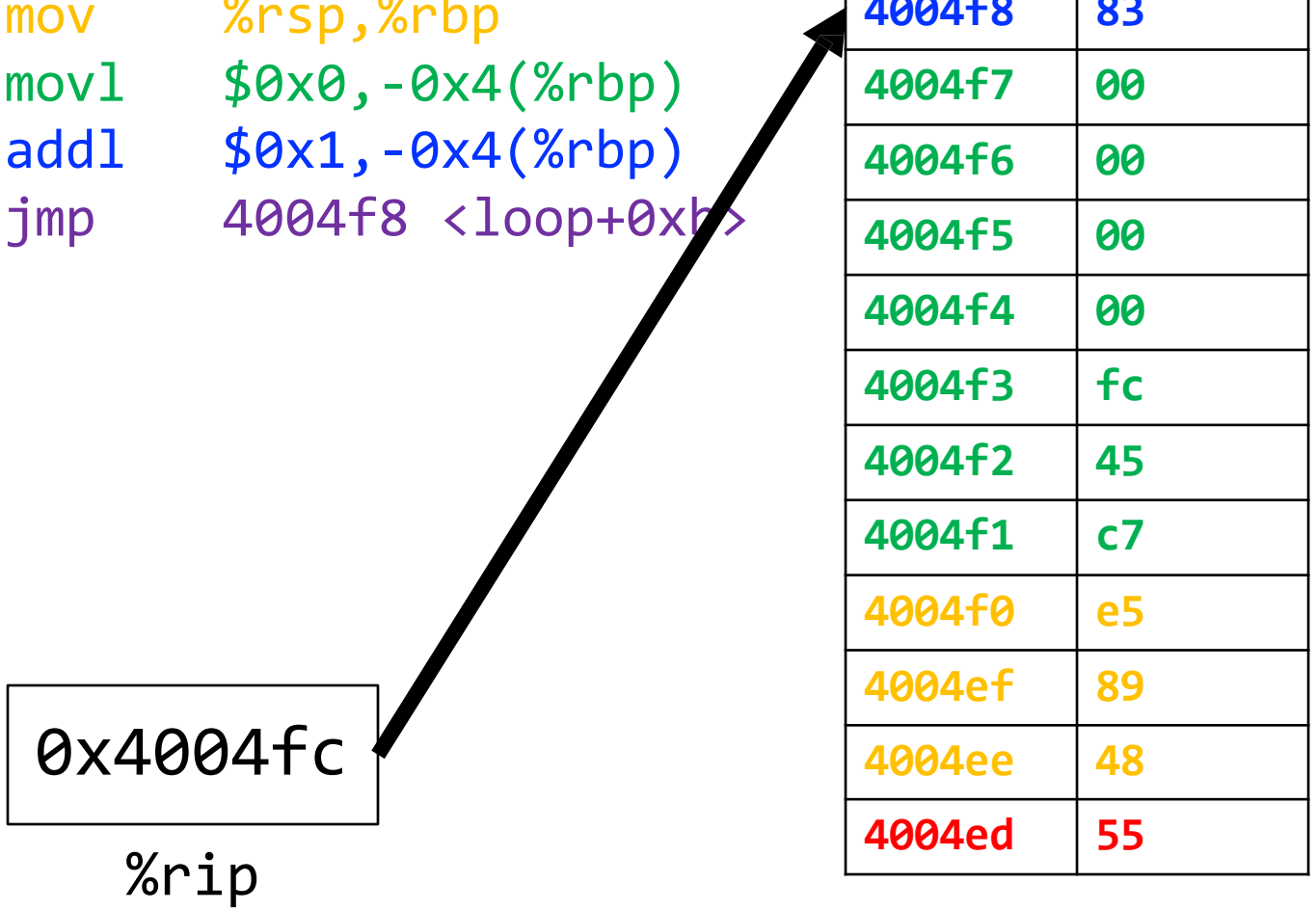
4004fd	fa
4004fc	eb
4004fb	01
4004fa	fc
4004f9	45
4004f8	83
4004f7	00
4004f6	00
4004f5	00
4004f4	00
4004f3	fc
4004f2	45
4004f1	c7
4004f0	e5
4004ef	89
4004ee	48
4004ed	55

Jump!

```
0000000004004ed <loop>:
4004ed:  55                              push   %rbp
4004ee:  48 89 e5                        mov    %rsp,%rbp
4004f1:  c7 45 fc 00 00 00 00            movl   $0x0,-0x4(%rbp)
4004f8:  83 45 fc 01                     addl   $0x1,-0x4(%rbp)
4004fc:  eb fa                           jmp    4004f8 <loop+0xb>
```

This assembly represents an infinite loop in C!

```
while (true) {…}
```



jmp

The **jmp** instruction jumps to another instruction in the assembly code ("Unconditional Jump").

```
jmp Label      (Direct Jump)
jmp *Operand   (Indirect Jump)
```

The destination can be hardcoded into the instruction (direct jump):

```
jmp 404f8 <loop+0xb>  # jump to instruction at 0x404f8
```

The destination can also be one of the usual operand forms (indirect jump):

```
jmp *%rax     # jump to instruction at address in %rax
```

"Interfering" with %rip

1. How do we repeat instructions in a loop?

`jmp [target]`

- A 1-step unconditional jump (always jump when we execute this instruction)

What if we want a **conditional jump**?

Control

- In C, we have control flow statements like **if**, **else**, **while**, **for**, etc. to write programs that are more expressive than just one instruction following another.
- This is *conditional execution of statements*: executing statements if one condition is true, executing other statements if one condition is false, etc.
- How is this represented in assembly?

Control

```
if (x > y){
        // a
} else {
        // b
}
```

In Assembly:

1. Calculate the condition result
2. Based on the result, go to a or b

Control

- In assembly, it takes more than one instruction to do these two steps.
- Most often: 1 instruction to calculate the condition, 1 to conditionally jump

Common Pattern:

1. `cmp S1, S2` // compare two values

2. je [target] *or* **jne [target]** *or* **jl [target]** *or* ... // conditionally jump

"jump if equal"

"jump if not equal"

"jump if less than"

Conditional Jumps

There are also variants of **jmp** that jump only if certain conditions are true ("Conditional Jump"). The jump location for these must be hardcoded into the instruction.

Instruction	Synonym	Set Condition
`je` *Label*	`jz`	Equal / zero
`jne` *Label*	`jnz`	Not equal / not zero
`js` *Label*		Negative
`jns` *Label*		Nonnegative
`jg` *Label*	`jnle`	Greater (signed >)
`jge` *Label*	`jnl`	Greater or equal (signed >=)
`jl` *Label*	`jnge`	Less (signed <)
`jle` *Label*	`jng`	Less or equal (signed <=)
`ja` *Label*	`jnbe`	Above (unsigned >)
`jae` *Label*	`jnb`	Above or equal (unsigned >=)
`jb` *Label*	`jnae`	Below (unsigned <)
`jbe` *Label*	`jna`	Below or equal (unsigned <=)

Control

Read **cmp S1,S2** as *"compare S2 to S1"*:

```
// Jump if %edi > 2
cmp $2, %edi
jg [target]
```

```
// Jump if %edi == 4
cmp $4, %edi
je [target]
```

```
// Jump if %edi != 3
cmp $3, %edi
jne [target]
```

```
// Jump if %edi <= 1
cmp $1, %edi
jle [target]
```

Control

Read **cmp S1,S2** as *"compare S2 to S1"*:

```
// Jump if %edi > 2
cmp $2, %edi
jg [target]
```

```
// Jump if %edi == 4
cmp $4, %edi
je [target]
```

```
// Jump if %edi != 3
cmp $3, %edi
jne [target]
```

```
// Jump if %edi <= 1
cmp $1, %edi
jle [target]
```

Wait a minute – how does the jump instruction know anything about the compared values in the earlier instruction?

Control

- The CPU has special registers called *condition codes* that are like “global variables”. They *automatically* keep track of information about the most recent arithmetic or logical operation.
  - **cmp** compares via calculation (subtraction) and info is stored in the condition codes
  - conditional jump instructions look at these condition codes to know whether to jump
- What exactly are the condition codes? How do they store this information?

Condition Codes

Alongside normal registers, the CPU also has single-bit *condition code* registers. They store the results of the most recent arithmetic or logical operation.

Most common condition codes:

- **CF:** Carry flag. The most recent operation generated a carry out of the most significant bit. Used to detect overflow for unsigned operations.
- **ZF:** Zero flag. The most recent operation yielded zero.
- **SF:** Sign flag. The most recent operation yielded a negative value.
- **OF:** Overflow flag. The most recent operation caused a two's-complement overflow-either negative or positive.

Condition Codes

Alongside normal registers, the CPU also has single-bit *condition code* registers. They store the results of the most recent arithmetic or logical operation.

Example: if we calculate t = a + b, condition codes are set according to:

- **CF:** Carry flag (Unsigned Overflow). `(unsigned) t < (unsigned) a`
- **ZF:** Zero flag (Zero). `(t == 0)`
- **SF:** Sign flag (Negative). `(t < 0)`
- **OF:** Overflow flag (Signed Overflow). `(a<0 == b<0) && (t<0 != a<0)`

Setting Condition Codes

The **cmp** instruction is like the subtraction instruction, but it does not store the result anywhere. It just sets condition codes. (**Note** the operand order!)

```
CMP S1, S2          S2 – S1
```

Instruction	Description
cmpb	Compare byte
cmpw	Compare word
cmpl	Compare double word
cmpq	Compare quad word

Control

Read **cmp S1,S2** as *"compare S2 to S1".* It calculates S2 – S1 and updates the condition codes with the result.

```
// Jump if %edi > 2
// calculates %edi - 2
cmp $2, %edi
jg [target]
```

```
// Jump if %edi == 4
// calculates %edi - 4
cmp $4, %edi
je [target]
```

```
// Jump if %edi != 3
// calculates %edi - 3
cmp $3, %edi
jne [target]
```

```
// Jump if %edi <= 1
// calculates %edi - 1
cmp $1, %edi
jle [target]
```

Conditional Jumps

Conditional jumps can look at subsets of the condition codes in order to check their condition of interest.

Instruction	Synonym	Set Condition
`je` *Label*	`jz`	Equal / zero (ZF = 1)
`jne` *Label*	`jnz`	Not equal / not zero (ZF = 0)
`js` *Label*		Negative (SF = 1)
`jns` *Label*		Nonnegative (SF = 0)
`jg` *Label*	`jnle`	Greater (signed >) (ZF = 0 and SF = OF)
`jge` *Label*	`jnl`	Greater or equal (signed >=) (SF = OF)
`jl` *Label*	`jnge`	Less (signed <) (SF != OF)
`jle` *Label*	`jng`	Less or equal (signed <=) (ZF = 1 or SF! = OF)
`ja` *Label*	`jnbe`	Above (unsigned >) (CF = 0 and ZF = 0)
`jae` *Label*	`jnb`	Above or equal (unsigned >=) (CF = 0)
`jb` *Label*	`jnae`	Below (unsigned <) (CF = 1)
`jbe` *Label*	`jna`	Below or equal (unsigned <=) (CF = 1 or ZF = 1)

Setting Condition Codes

The **test** instruction is like **cmp**, but for AND. It does not store the & result anywhere. It just sets condition codes.

```
TEST S1, S2          S2 & S1
```

Instruction	Description
`testb`	Test byte
`testw`	Test word
`testl`	Test double word
`testq`	Test quad word

Cool trick: if we pass the same value for both operands, we can check the sign of that value using the **Sign Flag** and **Zero Flag** condition codes!

Condition Codes

- Previously-discussed arithmetic and logical instructions update these flags. **lea** does not (it was intended only for address computations).
- Logical operations (**xor**, etc.) set carry and overflow flags to zero.
- Shift operations set the carry flag to the last bit shifted out and set the overflow flag to zero.
- For more complicated reasons, **inc** and **dec** set the overflow and zero flags, but leave the carry flag unchanged.

Exercise 1: Conditional jump

```
je target              jump if ZF is 1
```

Let %edi store 0x10. Will we jump in the following cases?

%edi | 0x10

1.
```
cmp $0x10,%edi
je  40056f
add  $0x1,%edi
```

2.
```
test $0x10,%edi
je   40056f
add  $0x1,%edi
```

Exercise 1: Conditional jump

```
je target          jump if ZF is 1
```

Let %edi store 0x10. Will we jump in the following cases? Assume they are run in order.

%edi | 0x10

1.
```
cmp $0x10,%edi
je  40056f
add  $0x1,%edi
```

S2 - S1 == 0, so jump

2.
```
test $0x10,%edi
je   40056f
add  $0x1,%edi
```

S2 & S1 != 0, so don't jump

Exercise 2: Conditional jump

```
0000000004004d6 <if_then>:
  4004d6:   83 ff 06    cmp     $0x6,%edi
  4004d9:   75 03       jne     4004de <if_then+0x8>
  400rdb:   83 c7 01    add     $0x1,%edi
  4004de:   8d 04 3f    lea     (%rdi,%rdi,1),%eax
  4004e1:   c3          retq
```

%edi | 0x5

1. What is the value of `%rip` after executing the `jne` instruction?
   A. `4004d9`
   B. `4004db`
   C. `4004de`
   D. Other

2. What is the value of `%eax` when we hit the `retq` instruction?
   A. `4004e1`
   B. `0x2`
   C. `0xa`
   D. `0xc`
   E. Other

Exercise 2: Conditional jump

```
0000000004004d6 <if_then>:
  4004d6:   83 ff 06    cmp     $0x6,%edi
  4004d9:   75 03       jne     4004de <if_then+0x8>
  400rdb:   83 c7 01    add     $0x1,%edi
  4004de:   8d 04 3f    lea     (%rdi,%rdi,1),%eax
  4004e1:   c3          retq
```

%edi **0x5**

1. What is the value of `%rip` after executing the `jne` instruction?
   - A. 4004d9
   - B. 4004db
   - C. 4004de
   - D. Other

2. What is the value of `%eax` when we hit the `retq` instruction?
   - A. 4004e1
   - B. 0x2
   - C. 0xa
   - D. 0xc
   - E. Other

If Statements

How can we use instructions like **cmp** and *conditional jumps* to implement if statements in assembly?

Practice: Fill In The Blank

```
int if_then(int param1) {
   if ( ___________ ) {
        ___________;
   }

   return ___________;
}
```

```
0000000000401126 <if_then>:
  401126:    cmp     $0x6,%edi
  401129:    je      40112f
  40112b:    lea     (%rdi,%rdi,1),%eax
  40112e:    retq
  40112f:    add     $0x1,%edi
  401132:    jmp     40112b
```

Practice: Fill In The Blank

```
int if_then(int param1) {
   if ( param1 == 6 ) {
         param1++;
   }

   return param1 * 2;
}
```

```
0000000000401126 <if_then>:
  401126:    cmp     $0x6,%edi
  401129:    je      40112f
  40112b:    lea     (%rdi,%rdi,1),%eax
  40112e:    retq
  40112f:    add     $0x1,%edi
  401132:    jmp     40112b
```

Common If-Else Construction

If-Else In C

```
long absdiff(long x, long y) {
    long result;
    if (x < y) {
        result = y - x;
    } else {
        result = x - y;
    }

    return result;
}
```

If-Else In Assembly pseudocode

```
Test
Jump to else-body if test passes
If-body
Jump to past else-body
Else-body
Past else body
```

Practice: Fill in the Blank

If-Else In C

```
long absdiff(long x, long y) {
    long result;
    if (________) {

        ____________________ ;
    } else {

        ____________________ ;
    }

    return result;
}
```

```
401134 <+0>:   mov     %rsi,%rax
401137 <+3>:   cmp     %rsi,%rdi
40113a <+6>:   jge     0x401140 <absdiff+12>
40113c <+8>:   sub     %rdi,%rax
40113f <+11>:  retq
401140 <+12>:  sub     %rsi,%rdi
401143 <+15>:  mov     %rdi,%rax
401146 <+18>:  retq
```

If-Else In Assembly pseudocode

```
Test
Jump to else-body if test passes
If-body
Jump to past else-body
Else-body
Past else body
```

Practice: Fill in the Blank

If-Else In C

```
long absdiff(long x, long y) {
    long result;
    if ( x < y ) {
        result = y - x ;
    } else {
        result = x - y ;
    }
    return result;
}
```

```
401134 <+0>:   mov    %rsi,%rax
401137 <+3>:   cmp    %rsi,%rdi
40113a <+6>:   jge    0x401140 <absdiff+12>
40113c <+8>:   sub    %rdi,%rax
40113f <+11>:  retq
401140 <+12>:  sub    %rsi,%rdi
401143 <+15>:  mov    %rdi,%rax
401146 <+18>:  retq
```

If-Else In Assembly pseudocode

```
Test
Jump to else-body if test passes
If-body
Jump to past else-body
Else-body
Past else body
```

If-Else Construction Variations

C Code

```
int test(int arg) {
    int ret;
    if (arg > 3) {
        ret = 10;
    } else {
        ret = 0;
    }

    ret++;
    return ret;
}
```

Assembly

```
401134 <+0>:   cmp    $0x3,%edi
401137 <+3>:   jle    0x401142 <test+14>
401139 <+5>:   mov    $0xa,%eax
40113e <+10>:  add    $0x1,%eax
401141 <+13>:  retq
401142 <+14>:  mov    $0x0,%eax
401147 <+19>:  jmp    0x40113e <test+10>
```

Loops and Control Flow

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

```
0x000000000040115c <+0>:    mov    $0x0,%eax
0x0000000000401161 <+5>:    cmp    $0x63,%eax
0x0000000000401164 <+8>:    jg     0x40116b <loop+15>
0x0000000000401166 <+10>:   add    $0x1,%eax
0x0000000000401169 <+13>:   jmp    0x401161 <loop+5>
0x000000000040116b <+15>:   retq
```

Loops and Control Flow

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

```
0x000000000040115c <+0>:    mov    $0x0,%eax
0x0000000000401161 <+5>:    cmp    $0x63,%eax
0x0000000000401164 <+8>:    jg     0x40116b <loop+15>
0x0000000000401166 <+10>:   add    $0x1,%eax
0x0000000000401169 <+13>:   jmp    0x401161 <loop+5>
0x000000000040116b <+15>:   retq
```

Set %eax (i) to 0.

Loops and Control Flow

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

```
0x000000000040115c <+0>:    mov    $0x0,%eax
0x0000000000401161 <+5>:    cmp    $0x63,%eax
0x0000000000401164 <+8>:    jg     0x40116b <loop+15>
0x0000000000401166 <+10>:   add    $0x1,%eax
0x0000000000401169 <+13>:   jmp    0x401161 <loop+5>
0x000000000040116b <+15>:   retq
```

Compare %eax (i) to 0x63 (99) by calculating %eax – 0x63. This is 0 – 99 = -99, so it sets the Sign Flag to 1.

Loops and Control Flow

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

```
0x000000000040115c <+0>:    mov    $0x0,%eax
0x0000000000401161 <+5>:    cmp    $0x63,%eax
0x0000000000401164 <+8>:    jg     0x40116b <loop+15>
0x0000000000401166 <+10>:   add    $0x1,%eax
0x0000000000401169 <+13>:   jmp    0x401161 <loop+5>
0x000000000040116b <+15>:   retq
```

jg means “jump if greater than”. This jumps if %eax > 0x63. The flags indicate this is false, so we do not jump.

Loops and Control Flow

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

```
0x000000000040115c <+0>:    mov    $0x0,%eax
0x0000000000401161 <+5>:    cmp    $0x63,%eax
0x0000000000401164 <+8>:    jg     0x40116b <loop+15>
0x0000000000401166 <+10>:   add    $0x1,%eax
0x0000000000401169 <+13>:   jmp    0x401161 <loop+5>
0x000000000040116b <+15>:   retq
```

Add 1 to %eax (i).

Loops and Control Flow

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

```
0x000000000040115c <+0>:    mov    $0x0,%eax
0x0000000000401161 <+5>:    cmp    $0x63,%eax
0x0000000000401164 <+8>:    jg     0x40116b <loop+15>
0x0000000000401166 <+10>:   add    $0x1,%eax
0x0000000000401169 <+13>:   jmp    0x401161 <loop+5>
0x000000000040116b <+15>:   retq
```

Jump to another instruction.

Loops and Control Flow

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

```
0x000000000040115c <+0>:    mov    $0x0,%eax
0x0000000000401161 <+5>:    cmp    $0x63,%eax
0x0000000000401164 <+8>:    jg     0x40116b <loop+15>
0x0000000000401166 <+10>:   add    $0x1,%eax
0x0000000000401169 <+13>:   jmp    0x401161 <loop+5>
0x000000000040116b <+15>:   retq
```

Compare %eax (i) to 0x63 (99) by calculating %eax – 0x63. This is 1 – 99 = -98, so it sets the Sign Flag to 1.

Loops and Control Flow

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

```
0x000000000040115c <+0>:    mov    $0x0,%eax
0x0000000000401161 <+5>:    cmp    $0x63,%eax
0x0000000000401164 <+8>:    jg     0x40116b <loop+15>
0x0000000000401166 <+10>:   add    $0x1,%eax
0x0000000000401169 <+13>:   jmp    0x401161 <loop+5>
0x000000000040116b <+15>:   retq
```

We continue in this pattern until we make this conditional jump. When will that be?

Loops and Control Flow

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

```
0x000000000040115c <+0>:    mov    $0x0,%eax
0x0000000000401161 <+5>:    cmp    $0x63,%eax
0x0000000000401164 <+8>:    jg     0x40116b <loop+15>
0x0000000000401166 <+10>:   add    $0x1,%eax
0x0000000000401169 <+13>:   jmp    0x401161 <loop+5>
0x000000000040116b <+15>:   retq
```

We will stop looping when this comparison says that %eax – 0x63 > 0!

Loops and Control Flow

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

```
0x000000000040115c <+0>:    mov    $0x0,%eax
0x0000000000401161 <+5>:    cmp    $0x63,%eax
0x0000000000401164 <+8>:    jg     0x40116b <loop+15>
0x0000000000401166 <+10>:   add    $0x1,%eax
0x0000000000401169 <+13>:   jmp    0x401161 <loop+5>
0x000000000040116b <+15>:   retq
```

Then, we return from the function.

GCC Common While Loop Construction

C

```
while (test) {
      body
}
```

Assembly

```
Test
Skip loop if test passes
Body
Jump back to test
```

From Previous Slide:

```
0x000000000040115c <+0>:    mov    $0x0,%eax
0x0000000000401161 <+5>:    cmp    $0x63,%eax
0x0000000000401164 <+8>:    jg     0x40116b <loop+15>
0x0000000000401166 <+10>:   add    $0x1,%eax
0x0000000000401169 <+13>:   jmp    0x401161 <loop+5>
0x000000000040116b <+15>:   retq
```

GCC Other While Loop Construction

C

```
while (test) {
      body
}
```

Assembly

```
Jump to test
Body
Test
Jump to body if test passes
```

From Previous Slide:

```
0x0000000000400570 <+0>:     mov    $0x0,%eax
0x0000000000400575 <+5>:     jmp    0x40057a <loop+10>
0x0000000000400577 <+7>:     add    $0x1,%eax
0x000000000040057a <+10>:    cmp    $0x63,%eax
0x000000000040057d <+13>:    jle    0x400577 <loop+7>
0x000000000040057f <+15>:    repz retq
```

Common For Loop Construction

C For loop

```
for (init; test; update) {
      body
}
```

C Equivalent While Loop

```
init
while(test) {
     body
     update
}
```

Assembly pseudocode

```
Init
Test
Skip loop if test passes
Body
Update
Jump back to test
```

For loops and while loops are treated (essentially) the same when compiled down to assembly.

Back to 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;
}
```

```
0000000000401136 <sum_array>:
  401136 <+0>:   mov    $0x0,%eax
  40113b <+5>:   mov    $0x0,%edx
  401140 <+10>:  cmp    %esi,%eax
  401142 <+12>:  jge    0x40114f <sum_array+25>
  401144 <+14>:  movslq %eax,%rcx
  401147 <+17>:  add    (%rdi,%rcx,4),%edx
  40114a <+20>:  add    $0x1,%eax
  40114d <+23>:  jmp    0x401140 <sum_array+10>
  40114f <+25>:  mov    %edx,%eax
  401151 <+27>:  retq
```

1. Which register is C code's `sum`?
2. Which register is C code's `i`?
3. Which assembly instruction is C code's `sum += arr[i]`?
4. What are the **`cmp`** and **`jge`** instructions doing?
   (**`jge`**: signed jump greater than/equal)

Condition Code-Dependent Instructions

There are three common instruction types that use condition codes:

- **jmp** instructions conditionally jump to a different next instruction
- **set** instructions conditionally set a byte to 0 or 1
- new versions of **mov** instructions conditionally move data

`set`: Read condition codes

`set` instructions condinonally set a byte to 0 or 1.

- Reads current state of flags
- operand is a single-byte register (e.g., %al) or single-byte memory locanon
- Does not perturb other bytes of register
- Typically followed by movzbl to zero those bytes

```
int small(int x) {
    return x < 16;
}
```

```
cmp $0xf,%edi
setle %al
movzbl %al, %eax
retq
```

`set`: Read condition codes

Instruction	Synonym	Set Condition (1 if true, 0 if false)
`sete D`	`setz`	Equal / zero
`setne D`	`setnz`	Not equal / not zero
`sets D`		Negative
`setns D`		Nonnegative
`setg D`	`setnle`	Greater (signed >)
`setge D`	`setnl`	Greater or equal (signed >=)
`setl D`	`setnge`	Less (signed <)
`setle D`	`setng`	Less or equal (signed <=)
`seta D`	`setnbe`	Above (unsigned >)
`setae D`	`setnb`	Above or equal (unsigned >=)
`setb D`	`setnae`	Below (unsigned <)
`setbe D`	`setna`	Below or equal (unsigned <=)

cmov: Conditional move

`cmovx src,dst` conditionally moves data in `src` to data in `dst`.

- Mov `src` to `dst` if condition `x` holds; no change otherwise
- `src` is memory address/register, `dst` is register
- May be more efficient than branch (i.e., jump)
- Often seen with C ternary operator: `result = test ? then: else;`

```
int max(int x, int y) {
    return x > y ? x : y;
}
```

```
cmp     %edi,%esi
mov     %edi, %eax
cmovge  %esi, %eax
retq
```

cmov: Conditional move

Instruction	Synonym	Move Condition
`cmove S,R`	`cmovz`	Equal / zero (ZF = 1)
`cmovne S,R`	`cmovnz`	Not equal / not zero (ZF = 0)
`cmovs S,R`		Negative (SF = 1)
`cmovns S,R`		NonnegaOve (SF = 0)
`cmovg S,R`	`cmovnle`	Greater (signed >) (SF = 0 and SF = OF)
`cmovge S,R`	`cmovnl`	Greater or equal (signed >=) (SF = OF)
`cmovl S,R`	`cmovnge`	Less (signed <) (SF != OF)
`cmovle S,R`	`cmovng`	Less or equal (signed <=) (ZF = 1 or SF! = OF)
`cmova S,R`	`cmovnbe`	Above (unsigned >) (CF = 0 and ZF = 0)
`cmovae S,R`	`cmovnb`	Above or equal (unsigned >=) (CF = 0)
`cmovb S,R`	`cmovnae`	Below (unsigned <) (CF = 1)
`cmovbe S,R`	`cmovna`	Below or equal (unsigned <=) (CF = 1 or ZF = 1)

How to remember cmp/jmp

- `CMP S1, S2` is `S2 – S1` (just sets condition codes). **But generally**:

```
cmp S1, S2
jg  …
```

⟺ **`S2 > S1`** ⟺ **`S2 - S1 > 0`**

- Much less important to remember exact condition codes
  - Yes, they fully explain conditional jmp…
  - …but more important to know how to translate assembly back into C
  - If you're interested, B&O p. 206 has details

Instruction	Synonym	Jump condition	Description
`jmp` *Label*		1	Direct jump
`jmp` **Operand*		1	Indirect jump
`je` *Label*	`jz`	ZF	Equal / zero
`jne` *Label*	`jnz`	~ZF	Not equal / not zero
`js` *Label*		SF	Negative
`jns` *Label*		~SF	Nonnegative
`jg` *Label*	`jnle`	~(SF ^ OF) & ~ZF	Greater (signed >)
`jge` *Label*	`jnl`	~(SF ^ OF)	Greater or equal (signed >=)
`jl` *Label*	`jnge`	SF ^ OF	Less (signed <)
`jle` *Label*	`jng`	(SF ^ OF) | ZF	Less or equal (signed <=)
`ja` *Label*	`jnbe`	~CF & ~ZF	Above (unsigned >)
`jae` *Label*	`jnb`	~CF	Above or equal (unsigned >=)
`jb` *Label*	`jnae`	CF	Below (unsigned <)
`jbe` *Label*	`jna`	CF | ZF	Below or equal (unsigned <=)

Figure 3.15 The jump instructions. These instructions jump to a labeled destination when the jump condition holds. Some instructions have "synonyms," alternate names for the same machine instruction.

🌟 Remember `test` exists

- `TEST S1, S2` is `S2 & S1`

```
test %edi, %edi
jns  …
```


`%edi & %edi is nonnegative`

`%edi is nonnegative`

Instruction		Synonym	Jump condition	Description
jmp	*Label*		1	Direct jump
jmp	**Operand*		1	Indirect jump
je	*Label*	jz	ZF	Equal / zero
jne	*Label*	jnz	~ZF	Not equal / not zero
js	*Label*		SF	Negative
jns	*Label*		~SF	Nonnegative
jg	*Label*	jnle	~(SF ^ OF) & ~ZF	Greater (signed >)
jge	*Label*	jnl	~(SF ^ OF)	Greater or equal (signed >=)
jl	*Label*	jnge	SF ^ OF	Less (signed <)
jle	*Label*	jng	(SF ^ OF) | ZF	Less or equal (signed <=)
ja	*Label*	jnbe	~CF & ~ZF	Above (unsigned >)
jae	*Label*	jnb	~CF	Above or equal (unsigned >=)
jb	*Label*	jnae	CF	Below (unsigned <)
jbe	*Label*	jna	CF | ZF	Below or equal (unsigned <=)

Figure 3.15 The jump instructions. These instructions jump to a labeled destination when the jump condition holds. Some instructions have "synonyms," alternate names for the same machine instruction.

Practice: Fill in the blanks

```
long loop(long a, long b) {
    long result = ___(1)___;
    while (___(2)___) {
      result = ____(3)_____;
      a = ___(4)___;
    }
    return result;
}
```

```
<+0>:   mov    $0x1,%eax
<+5>:   cmp    %rsi,%rdi
<+8>:   jge    0x1151 <loop+24>
<+10>:  lea    (%rdi,%rsi,1),%rdx
<+14>:  imul   %rdx,%rax
<+18>:  add    $0x1,%rdi
<+22>:  jmp    0x113e <loop+5>
<+24>:  retq
```

GCC common while loop construction:

Test

Jump past loop if fails

Body

Jump to test

https://godbolt.org/z/zrW6c5MGa

Practice: Fill in the blanks

```
long loop(long a, long b) {
    long result = ___(1)___;
    while (___(2)___) {
      result = ____(3)_____;
      a = ___(4)___;
    }
    return result;
}
```

```
<+0>:   mov    $0x1,%eax
<+5>:   cmp    %rsi,%rdi
<+8>:   jge    0x1151 <loop+24>
<+10>:  lea    (%rdi,%rsi,1),%rdx
<+14>:  imul   %rdx,%rax
<+18>:  add    $0x1,%rdi
<+22>:  jmp    0x113e <loop+5>
<+24>:  retq
```

GCC common while loop construction:

Test
Jump past loop if fails
Body
Jump to test

Practice: Fill in the blanks

```
long loop(long a, long b) {
    long result = _________;
    while (_________) {
      result = _____________;
      a = _________;
    }
    return result;
}
```

```
<+0>:    mov     $0x1,%eax

<+5>:    cmp     %rsi,%rdi
<+8>:    jge     0x1151 <loop+24>

<+10>:   lea     (%rdi,%rsi,1),%rdx
<+14>:   imul    %rdx,%rax
<+18>:   add     $0x1,%rdi

<+22>:   jmp     0x113e <loop+5>

<+24>:   retq
```

Practice: Fill in the blanks

```
long loop(long a, long b) {
    long result = ____1____;
    while (__a < b___) {
      result = result*(a+b);
      a = __a + 1___;
    }
    return result;
}
```

```
<+0>:    mov     $0x1,%eax

<+5>:    cmp     %rsi,%rdi
<+8>:    jge     0x1151 <loop+24>

<+10>:   lea     (%rdi,%rsi,1),%rdx
<+14>:   imul    %rdx,%rax
<+18>:   add     $0x1,%rdi

<+22>:   jmp     0x113e <loop+5>

<+24>:   retq
```

Warm-up: Reverse Engineering

```
int elem_arithmetic(int nums[], int y) {
    int z = nums[________] * ________;

    z -= ________;

    return ________;
}
----------
// nums in %rdi, y in %esi
elem_arithmetic:
  movl %esi, %eax
  imull 4(%rdi), %eax
  movslq %esi, %rsi
  subl (%rdi,%rsi,4), %eax
  lea 2(%rax, %rax), %eax
  ret
```

Warm-up: Reverse Engineering

```
int elem_arithmetic(int nums[], int y) {
    int z = nums[1] * y;

    z -= ________;

    return ________;
}
----------
// nums in %rdi, y in %esi
elem_arithmetic:
  movl %esi, %eax           // copy y into %eax
  imull 4(%rdi), %eax       // multiply %eax by nums[1]
  movslq %esi, %rsi         // sign-extend %esi to %rsi
  subl (%rdi,%rsi,4), %eax
  lea 2(%rax, %rax), %eax
  ret
```

Work through the last two blanks in groups and input your answer for the first blank on PollEv:
pollev.com/cs107 or text CS107 to 22333 once to join.

Warm-up: Reverse Engineering

```
int elem_arithmetic(int nums[], int y) {
    int z = nums[1] * y;

    z -= nums[y];

    return 2 * z + 2;
}
----------
// nums in %rdi, y in %esi
elem_arithmetic:
  movl %esi, %eax           // copy y into %eax
  imull 4(%rdi), %eax       // multiply %eax by nums[1]
  movslq %esi, %rsi         // sign-extend %esi to %rsi
  subl (%rdi,%rsi,4), %eax  // subtract nums[y] from %eax
  lea 2(%rax, %rax), %eax   // multiply %rax by 2, and add 2
  ret
```

`test` practice: What's the C code?

```
0x400546 <test_func>     test   %edi,%edi
0x400548 <test_func+2>   jns    0x400550 <test_func+10>
0x40054a <test_func+4>   mov    $0xfeed,%eax
0x40054f <test_func+9>   retq
0x400550 <test_func+10>  mov    $0xaabbccdd,%eax
0x400555 <test_func+15>  retq
```

`test` practice: What's the C code?

```
0x400546 <test_func>     test   %edi,%edi
0x400548 <test_func+2>   jns    0x400550 <test_func+10>
0x40054a <test_func+4>   mov    $0xfeed,%eax
0x40054f <test_func+9>   retq
0x400550 <test_func+10>  mov    $0xaabbccdd,%eax
0x400555 <test_func+15>  retq
```

```
int test_func(int x) {
    if (x < 0) {
        return 0xfeed;
    }
    return 0xaabbccdd;
}
```

(or anything like this)

Practice: "Escape Room"

```
<escape_room+0>     lea    (%rdi,%rdi,1),%eax
<escape_room+3>     cmp    $0x5,%eax
<escape_room+6>     jg     0x114c <escape_room+19>
<escape_room+8>     cmp    $0x1,%edi
<escape_room+11>    je     0x1152 <escape_room+25>
<escape_room+13>    mov    $0x0,%eax
<escape_room+18>    retq
<escape_room+19>    mov    $0x1,%eax
<escape_room+24>    retq
<escape_room+25>    mov    $0x1,%eax
<escape_room+30>    retq
```

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

You don't have to reverse-engineer C code exactly!

Practice: "Escape Room"

```
<escape_room+0>     lea    (%rdi,%rdi,1),%eax
<escape_room+3>     cmp    $0x5,%eax
<escape_room+6>     jg     0x114c <escape_room+19>
<escape_room+8>     cmp    $0x1,%edi
<escape_room+11>    je     0x1152 <escape_room+25>
<escape_room+13>    mov    $0x0,%eax
<escape_room+18>    retq
<escape_room+19>    mov    $0x1,%eax
<escape_room+24>    retq
<escape_room+25>    mov    $0x1,%eax
<escape_room+30>    retq
```

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

First param > 2 or == 1.

%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++;
    }
}
```

```
0x40113f <+0>:  b8 00 00 00 00  mov  $0x0,%eax
0x401144 <+5>:  83 f8 63        cmp  $0x63,%eax
0x401147 <+8>:  7f 05           jg   40114e <loop2+15>
0x401149 <+10>: 83 c0 01        add  $0x1,%eax
0x40114c <+13>: eb f6           jmp  401144 <loop2+5>
0x40114e <+15>: c3              retq
```

%rip

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

```
0x40113f <+0>:  b8 00 00 00 00   mov   $0x0,%eax
0x401144 <+5>:  83 f8 63         cmp   $0x63,%eax
0x401147 <+8>:  7f 05            jg    40114e <loop2+15>
0x401149 <+10>: 83 c0 01         add   $0x1,%eax
0x40114c <+13>: eb f6            jmp   401144 <loop2+5>
0x40114e <+15>: c3               retq
```

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

%rip

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

```
0x40113f <+0>:  b8 00 00 00 00   mov   $0x0,%eax
0x401144 <+5>:  83 f8 63         cmp   $0x63,%eax
0x401147 <+8>:  7f 05            jg    40114e <loop2+15>
0x401149 <+10>: 83 c0 01         add   $0x1,%eax
0x40114c <+13>: eb f6            jmp   401144 <loop2+5>
0x40114e <+15>: c3               retq
```

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

%rip

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

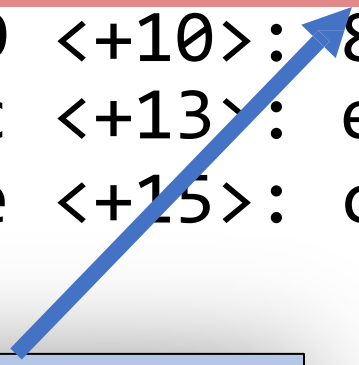
```
0x40113f <+0>:   b8 00 00 00 00   mov   $0x0,%eax
0x401144 <+5>:   83 f8 63         cmp   $0x63,%eax
0x401147 <+8>:   7f 05            jg    40114e <loop2+15>
0x401149 <+10>:  83 c0 01         add   $0x1,%eax
0x40114c <+13>:  eb f6            jmp   401144 <loop2+5>
0x40114e <+15>:  c3               retq
```

%rip

```
0x40113f <+0>:   b8 00 00 00 00   mov   $0x0,%eax
0x401144 <+5>:   83 f8 63         cmp   $0x63,%eax
0x401147 <+8>:   7f 05            jg    40114e <loop2+15>
0x401149 <+10>: 83 c0 01          add   $0x1,%eax
0x40114c <+13>: eb f6             jmp   401144 <loop2+5>
0x40114e <+15>: c3                retq
```

%rip

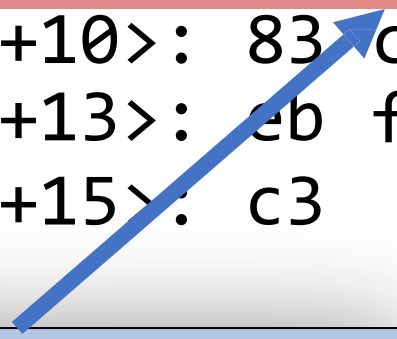
```
0x40113f <+0>:   b8 00 00 00 00   mov   $0x0,%eax
0x401144 <+5>:   83 f8 63         cmp   $0x63,%eax
0x401147 <+8>:   7f 05            jg    40114e <loop2+15>
0x401149 <+10>:  83 c0 01         add   $0x1,%eax
0x40114c <+13>:  eb f6            jmp   401144 <loop2+5>
0x40114e <+15>:  c3               retq
```



0x7f means **jg**.

%rip

```
0x40113f <+0>:   b8 00 00 00 00   mov   $0x0,%eax
0x401144 <+5>:   83 f8 63         cmp   $0x63,%eax
0x401147 <+8>:   7f 05            jg    40114e <loop2+15>
0x401149 <+10>:  83 c0 01         add   $0x1,%eax
0x40114c <+13>:  eb f6            jmp   401144 <loop2+5>
0x40114e <+15>:  c3               retq
```

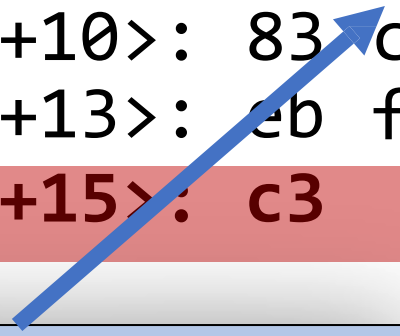


0x05 is the number of instruction bytes to jump relative to %rip.

With no jump, %rip would advance to the next line. This **jg** says to <u>then</u> go **5** bytes further!

%rip

```
0x40113f <+0>:   b8 00 00 00 00   mov   $0x0,%eax
0x401144 <+5>:   83 f8 63         cmp   $0x63,%eax
0x401147 <+8>:   7f 05            jg    40114e <loop2+15>
0x401149 <+10>:  83 c0 01         add   $0x1,%eax
0x40114c <+13>:  eb f6            jmp   401144 <loop2+5>
0x40114e <+15>:  c3               retq
```

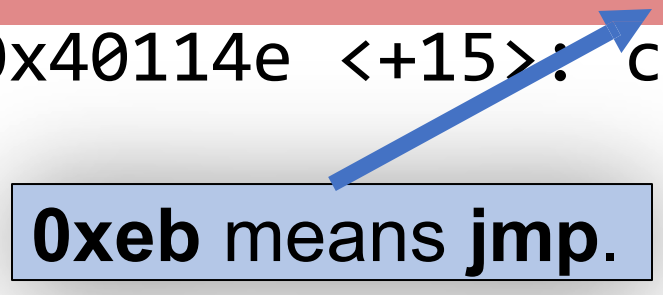


0x05 is the number of instruction bytes to jump relative to %rip.

With no jump, %rip would advance to the next line. This **jg** says to <u>then</u> go **5** bytes further!

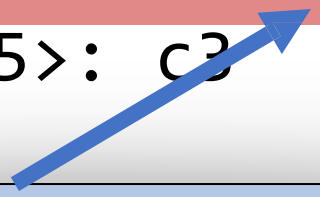
%rip

```
0x40113f <+0>:   b8 00 00 00 00   mov   $0x0,%eax
0x401144 <+5>:   83 f8 63         cmp   $0x63,%eax
0x401147 <+8>:   7f 05            jg    40114e <loop2+15>
0x401149 <+10>:  83 c0 01         add   $0x1,%eax
0x40114c <+13>:  eb f6            jmp   401144 <loop2+5>
0x40114e <+15>:  c3               retq
```



%rip

```
0x40113f <+0>:   b8 00 00 00 00   mov   $0x0,%eax
0x401144 <+5>:   83 f8 63         cmp   $0x63,%eax
0x401147 <+8>:   7f 05            jg    40114e <loop2+15>
0x401149 <+10>:  83 c0 01         add   $0x1,%eax
0x40114c <+13>:  eb f6            jmp   401144 <loop2+5>
0x40114e <+15>:  c3               retq
```

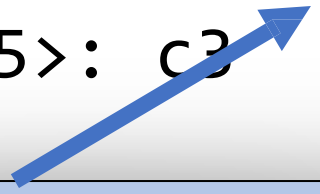


0xf6 is the number of instruction bytes to jump relative to %rip. This is -10 (in two's complement!).

With no jump, %rip would advance to the next line. This **jmp** says to then go **10** bytes back!

%rip

```
0x40113f <+0>:   b8 00 00 00 00   mov   $0x0,%eax
0x401144 <+5>:   83 f8 63         cmp   $0x63,%eax
0x401147 <+8>:   7f 05            jg    40114e <loop2+15>
0x401149 <+10>:  83 c0 01         add   $0x1,%eax
0x40114c <+13>:  eb f6            jmp   401144 <loop2+5>
0x40114e <+15>:  c3               retq
```



0xf6 is the number of instruction bytes to jump relative to %rip. This is -10 (in two's complement!).

With no jump, %rip would advance to the next line. This **jmp** says to then go **10** 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.

How do we call functions in assembly?

Calling Functions In Assembly

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

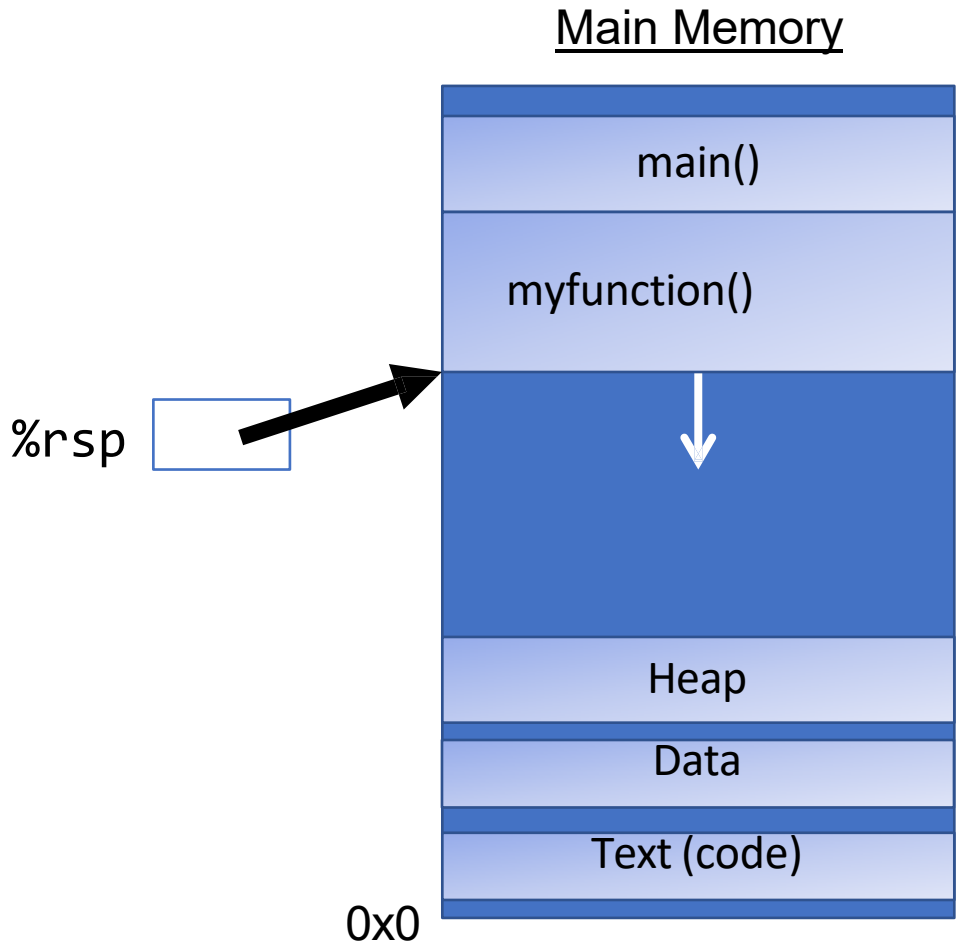
- **Pass Control** – %rip must be adjusted to execute the callee's instructions, and then resume the caller's instructions afterwards.
- **Pass Data** – we must pass any parameters and receive any return value.
- **Manage Memory** – we must handle any space needs of the callee on the stack.

How does assembly interact with the stack?

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

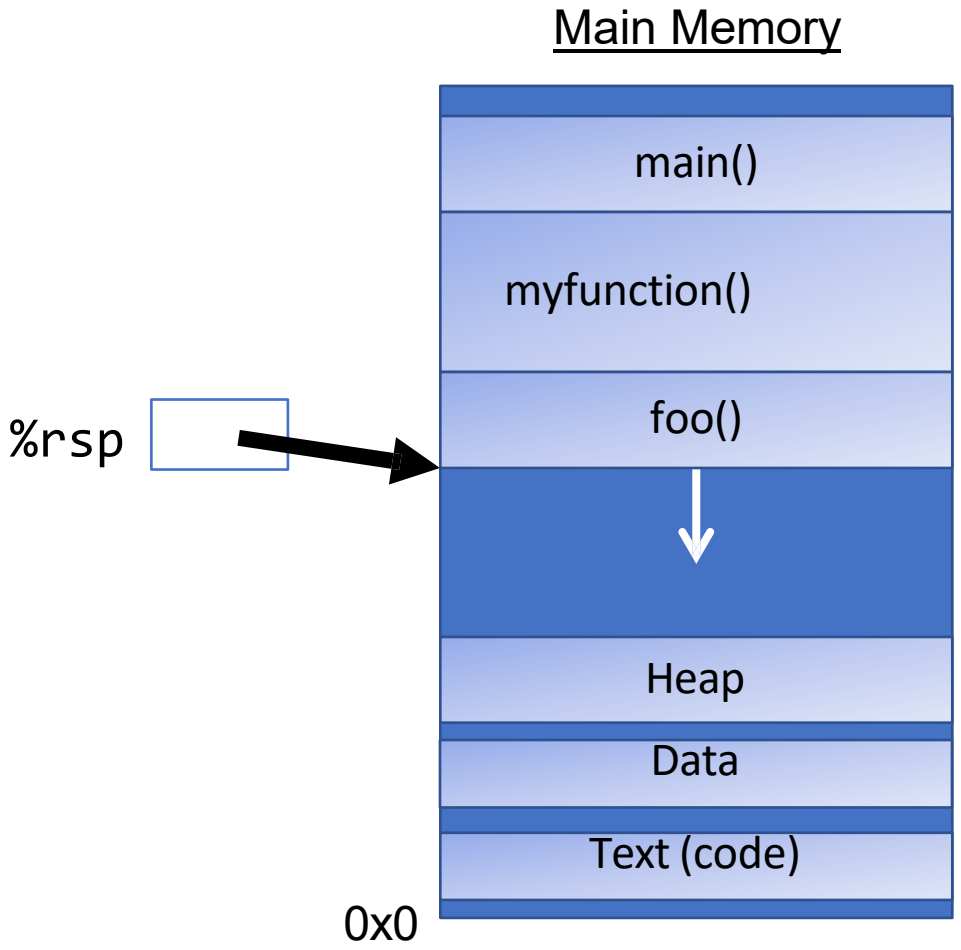
%rsp

- **%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).



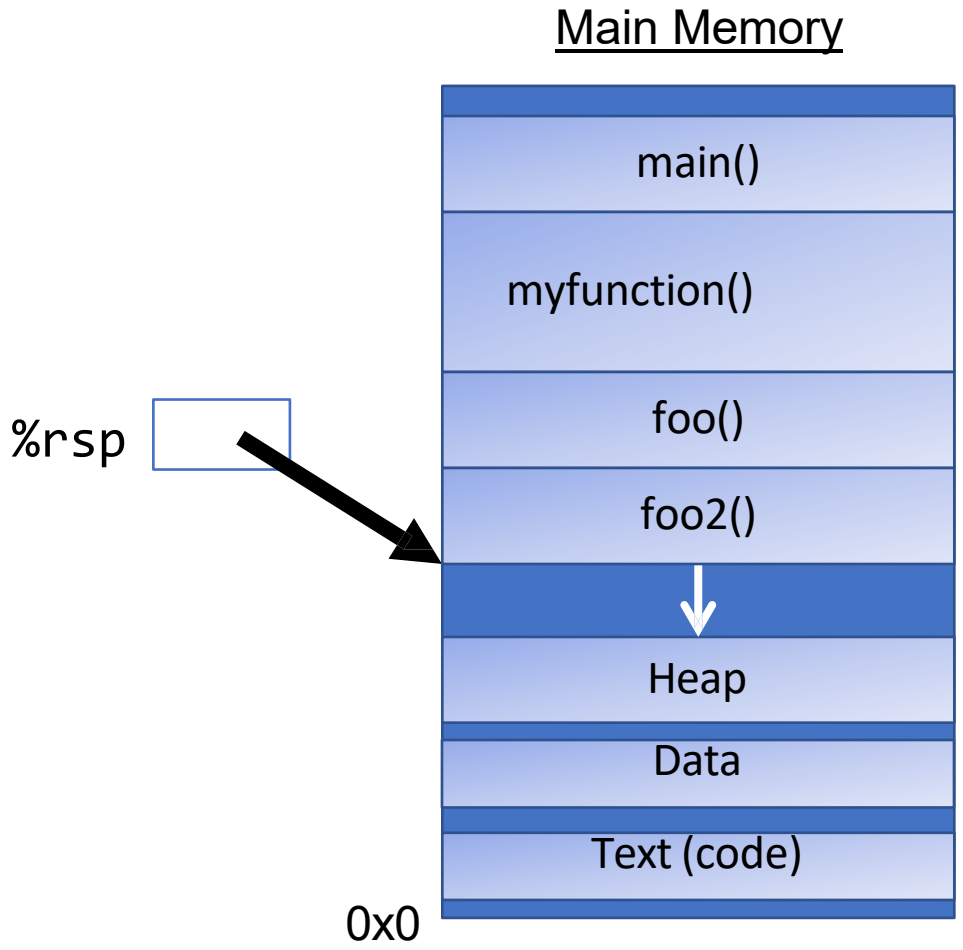
%rsp

- **%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).



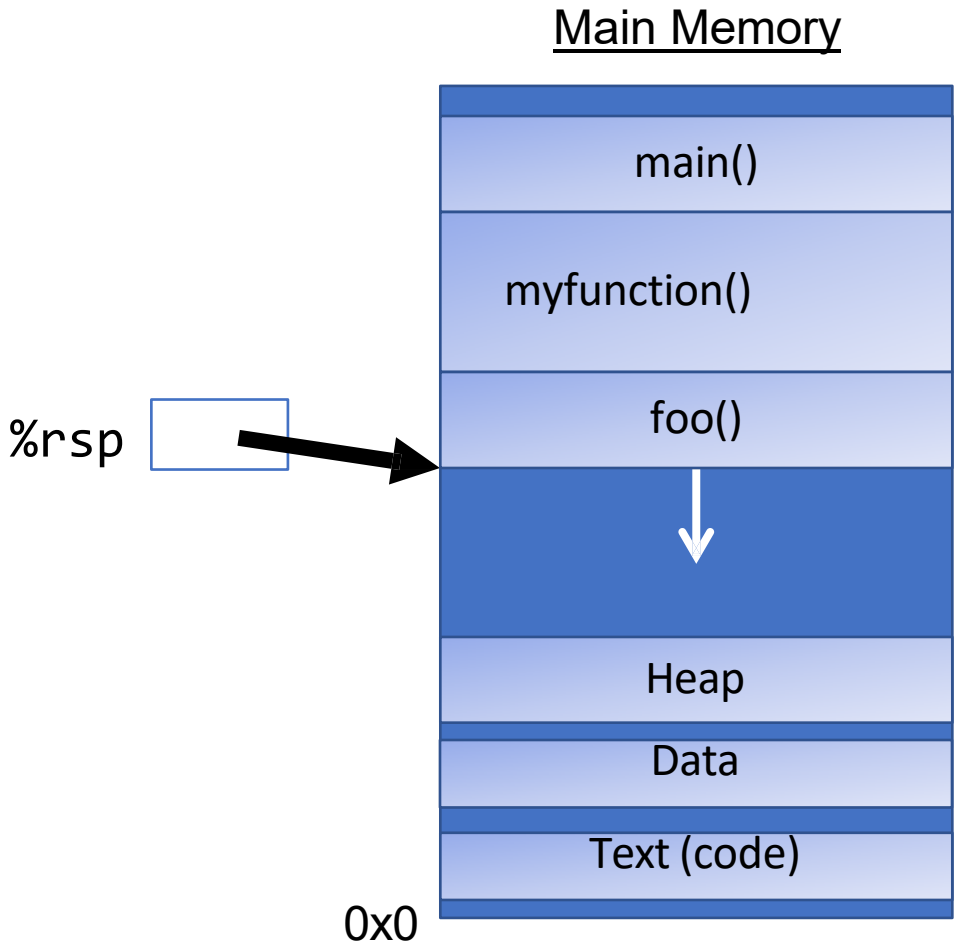
%rsp

- **%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).



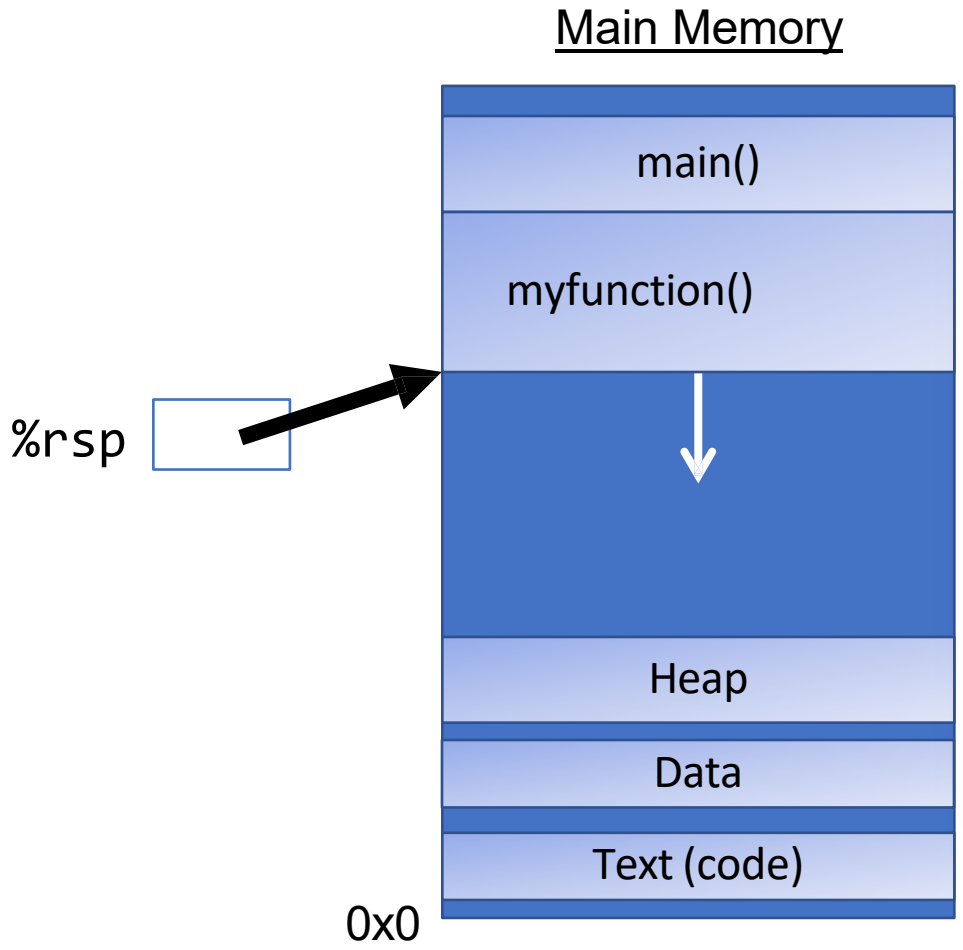
%rsp

- **%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] ⟵ 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.

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
```

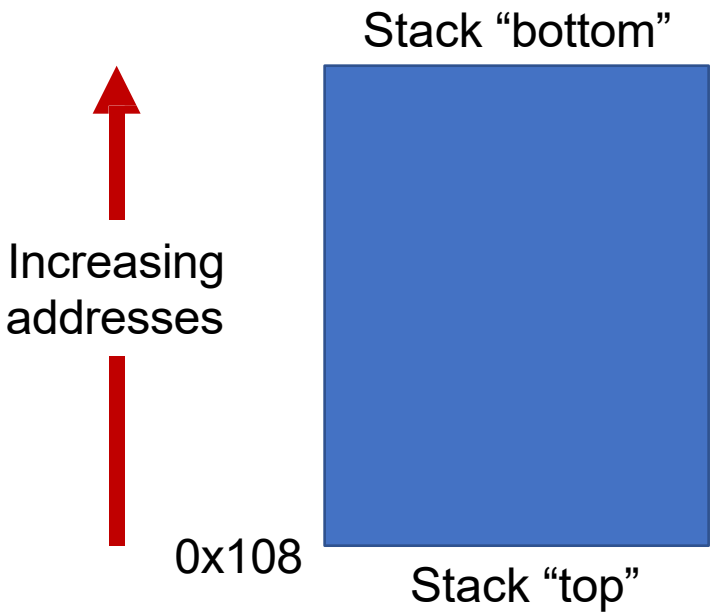
- Sometimes, you'll see instructions just explicitly increment the stack pointer to pop data.

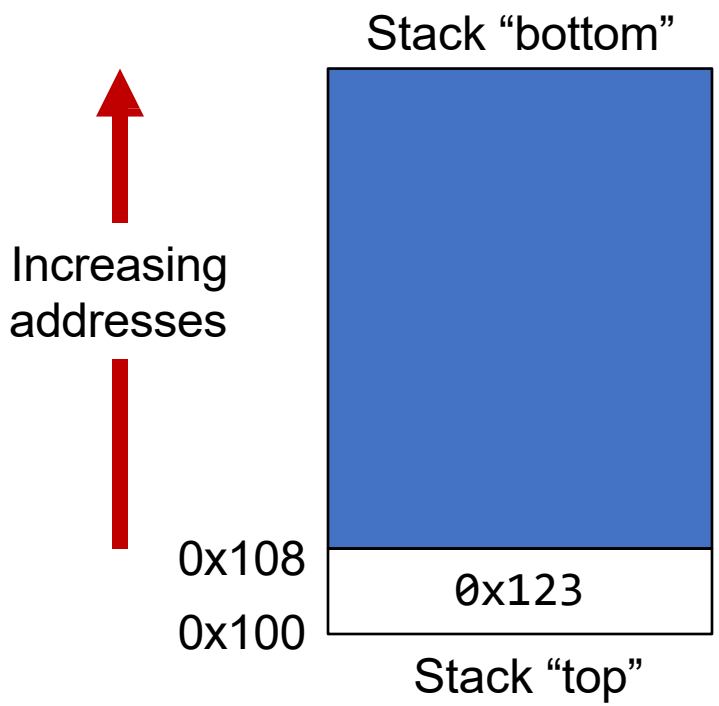
Stack Example

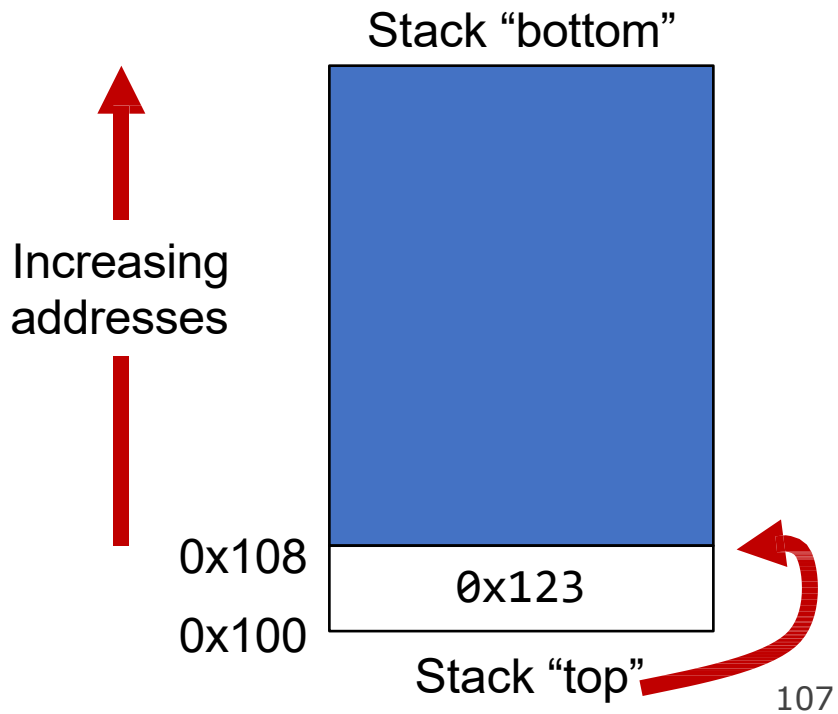
Initially	
%rax	0x123
%rdx	0
%rsp	0x108

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

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







Calling Functions In Assembly

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

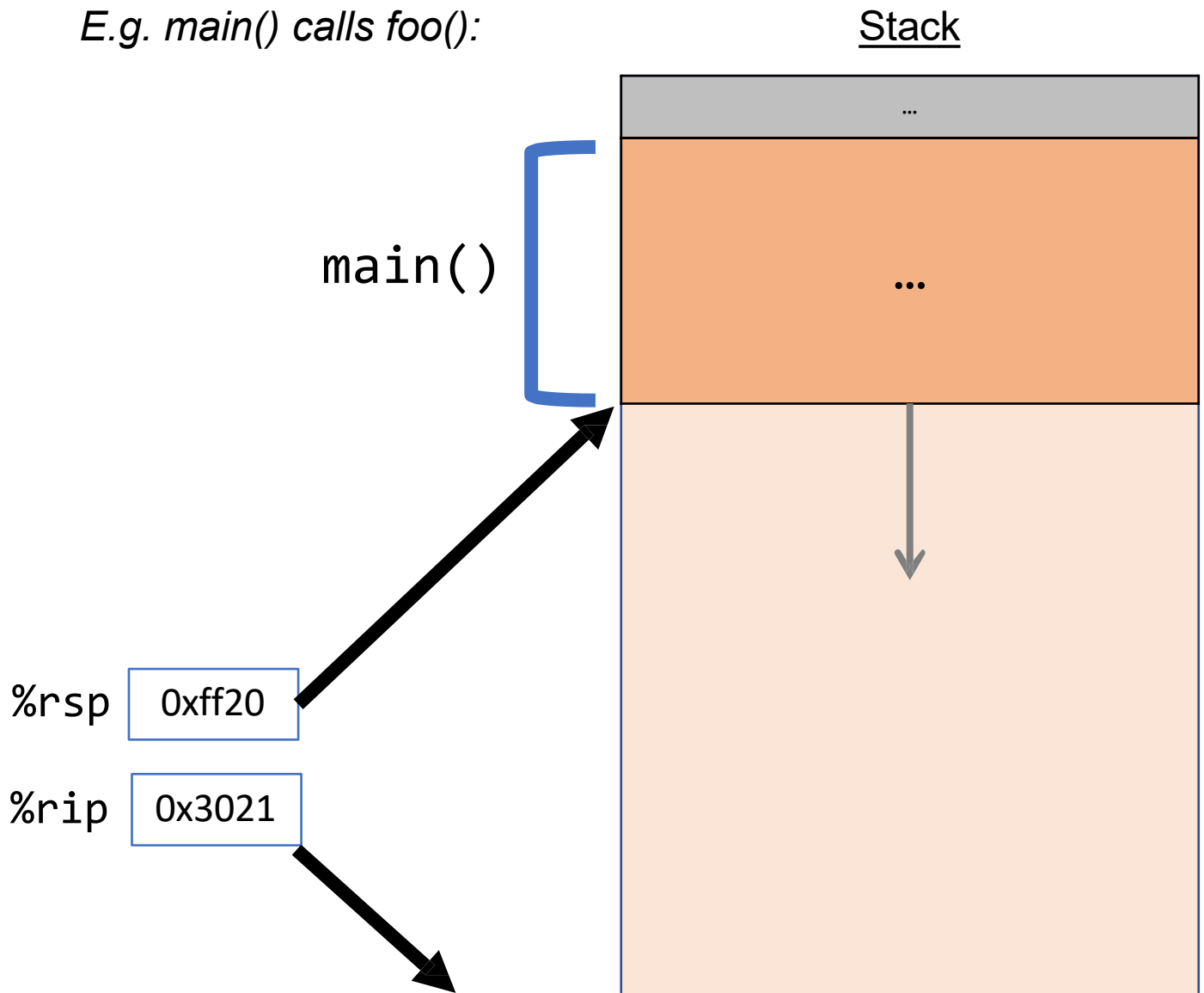
- **Pass Control** – %rip must be adjusted to execute the callee's instructions, and then resume the caller's instructions afterwards.
- **Pass Data** – we must pass any parameters and receive any return value.
- **Manage Memory** – we must handle any space needs of the callee on the stack.

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

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.

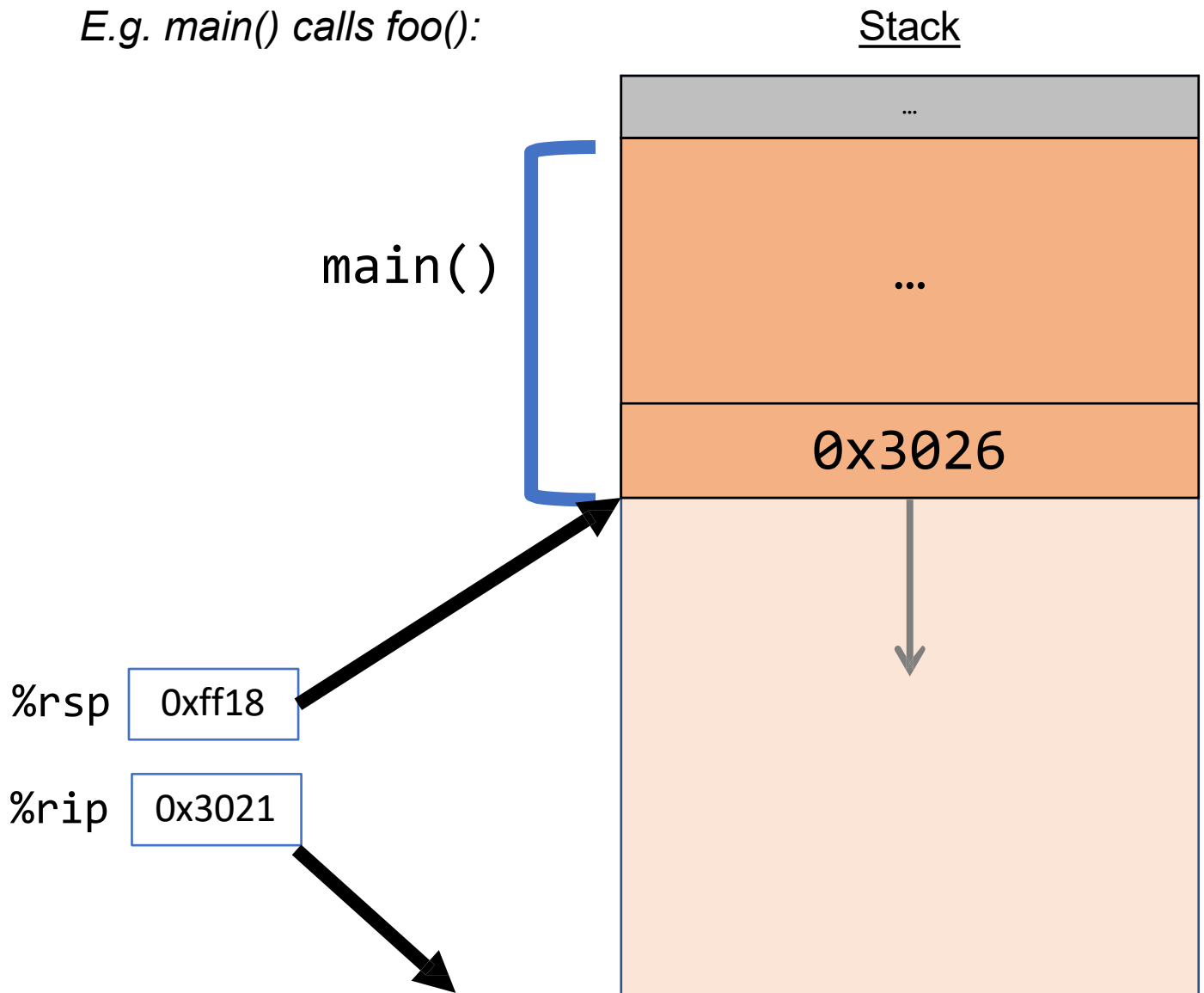
Solution: push the next value of %rip onto the stack. Then call the function. When it is finished, put this value back into %rip and continue executing.



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.

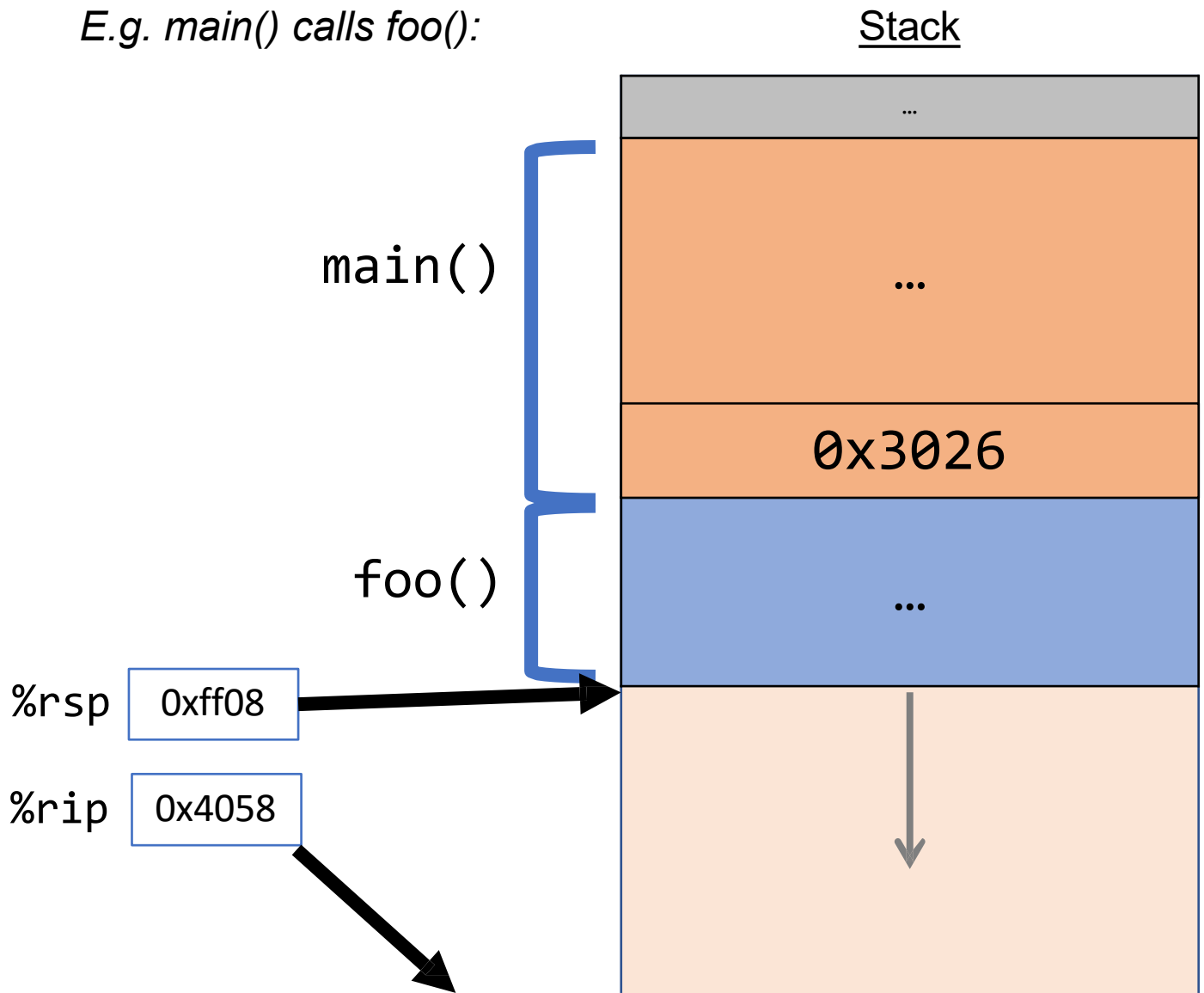
Solution: push the next value of %rip onto the stack. Then call the function. When it is finished, put this value back into %rip and continue executing.



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.

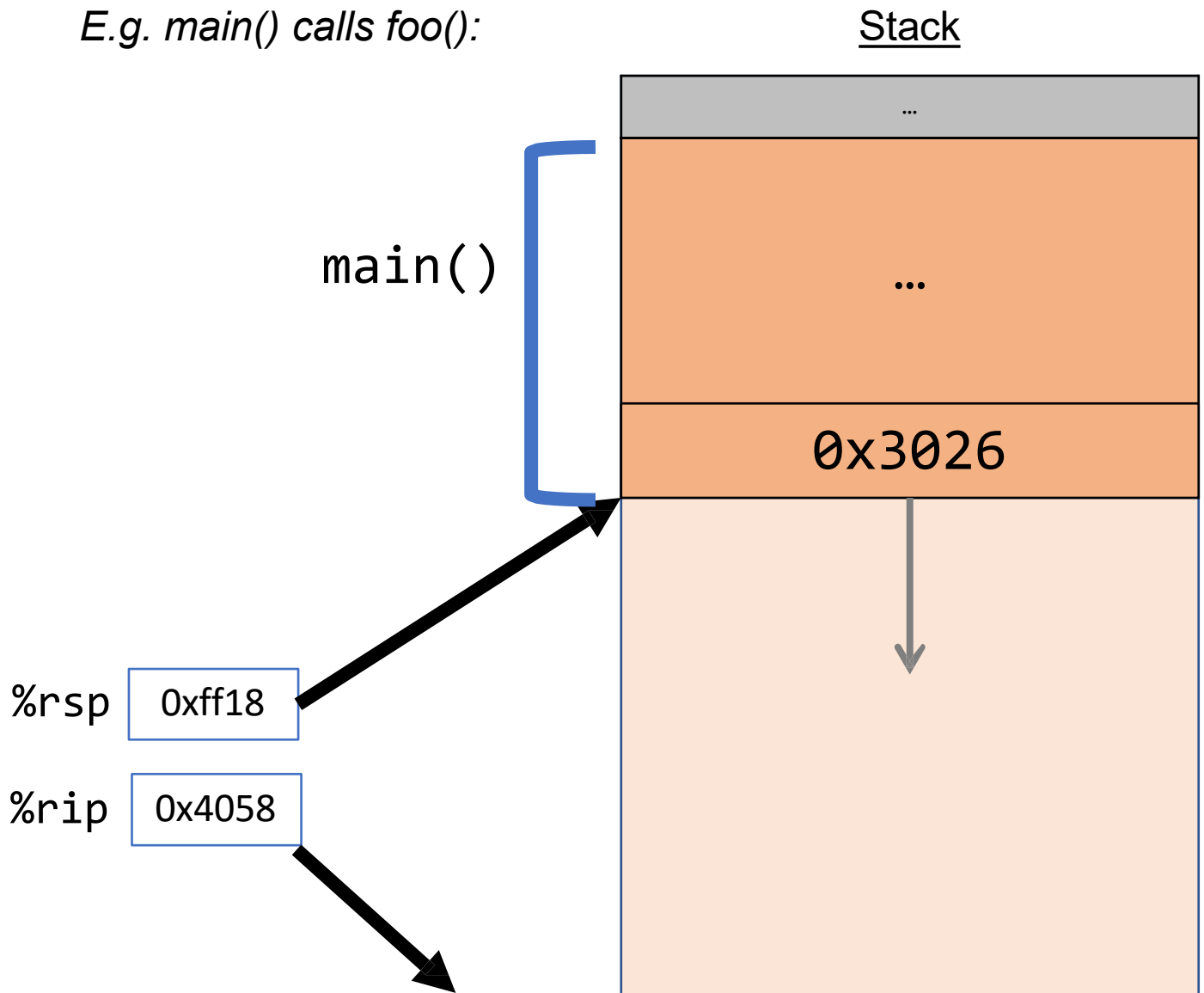
Solution: push the next value of %rip onto the stack. Then call the function. When it is finished, put this value back into %rip and continue executing.



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.

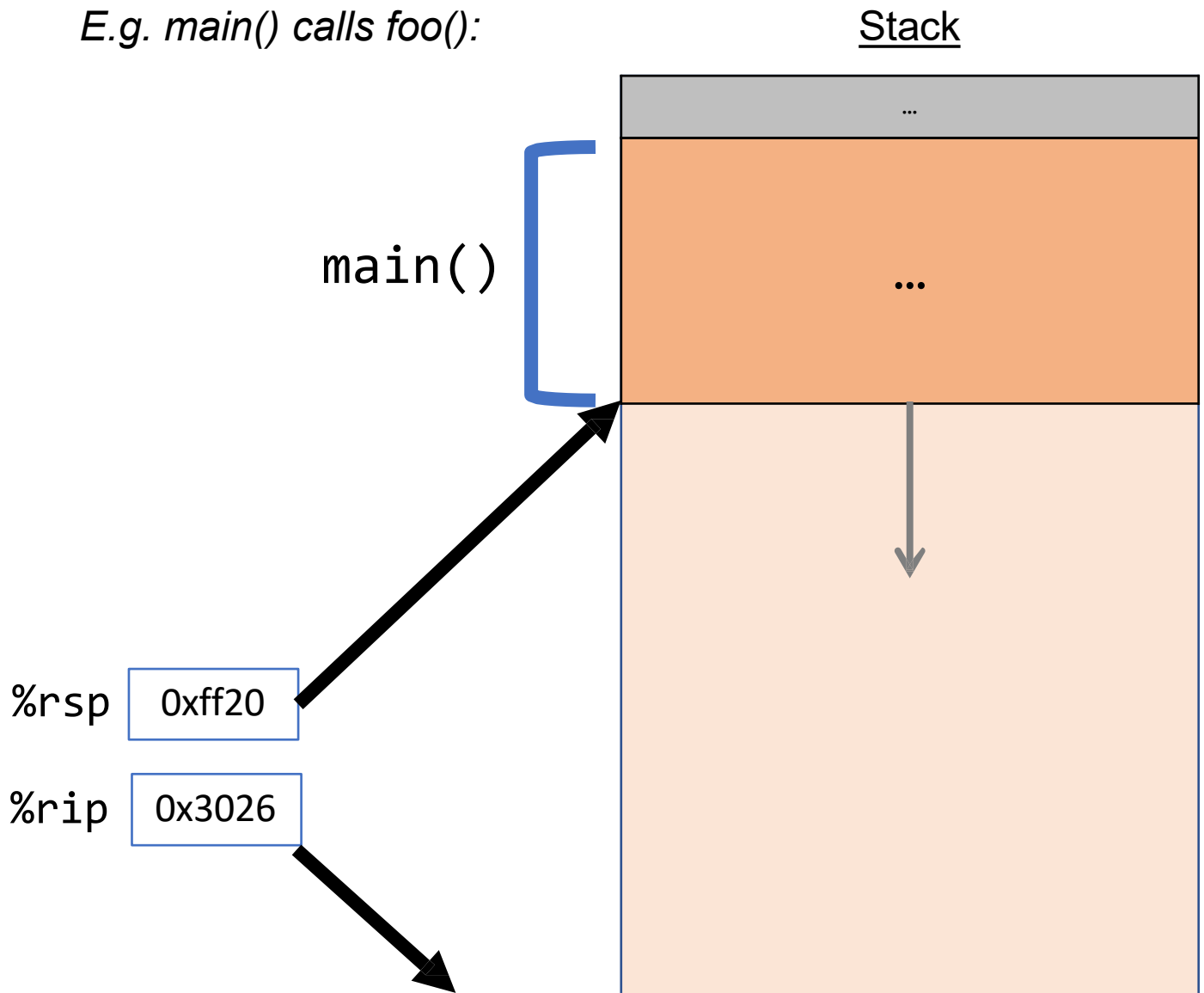
Solution: push the next value of %rip onto the stack. Then call the function. When it is finished, put this value back into %rip and continue executing.



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.



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).

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.

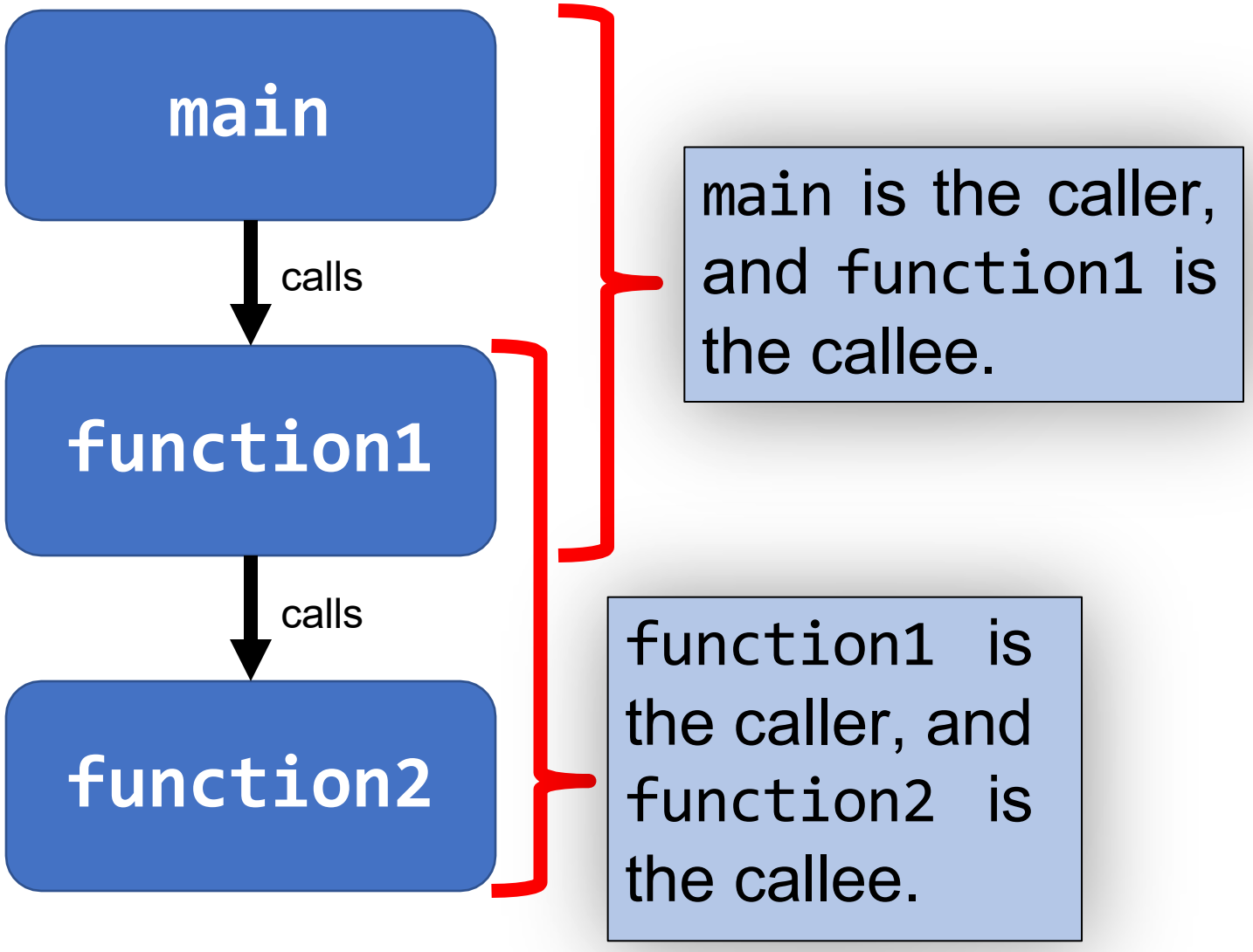
Register Restrictions

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

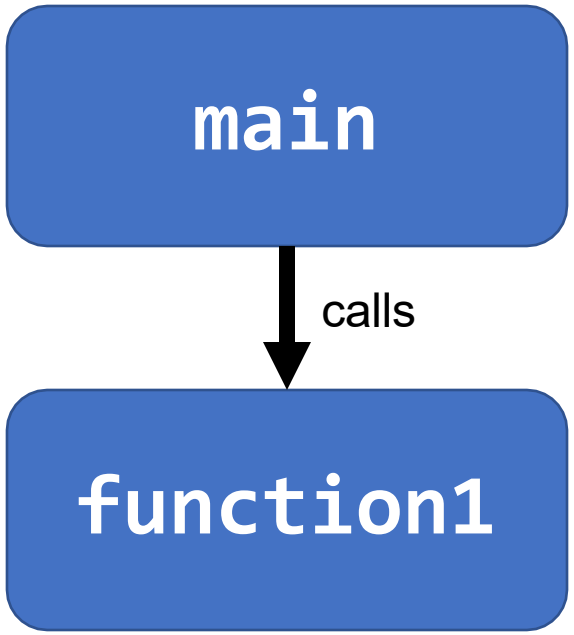
Caller-Owned

- Callee 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

- 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.

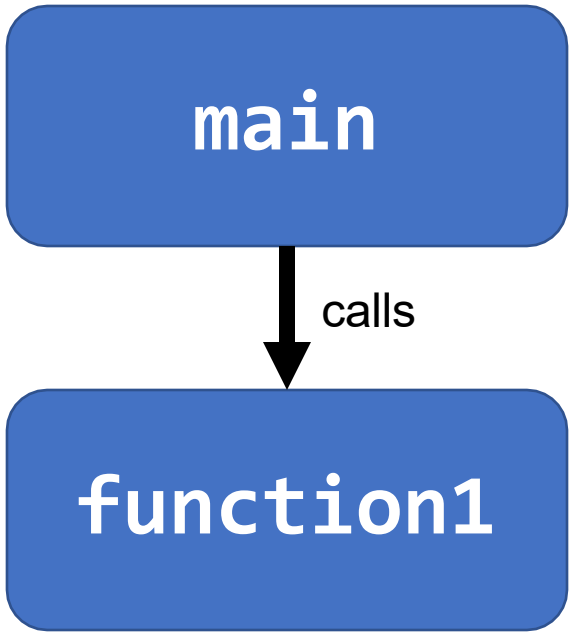
Caller-Owned Registers



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

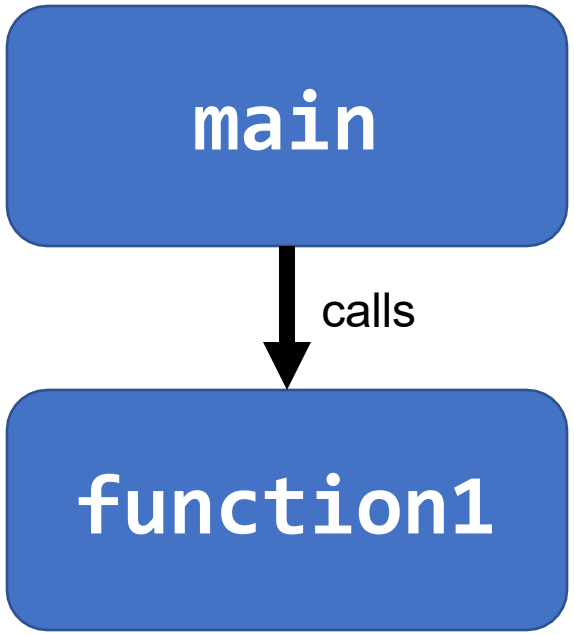
If function1 wants to use any caller-owned registers, it must save the existing values and restore them before returning.

Caller-Owned Registers




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

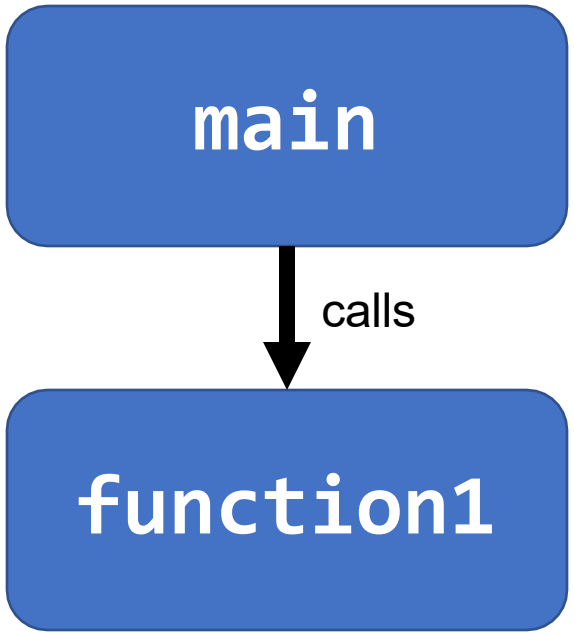
Callee-Owned Registers



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

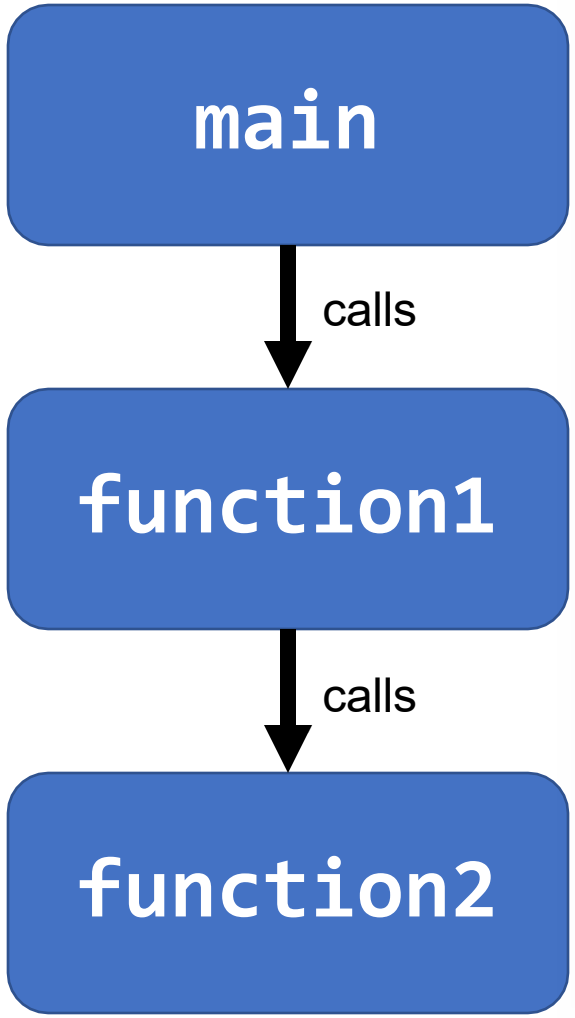
If function1 wants to use any callee-owned registers, it can do so without saving the existing values.

Callee-Owned Registers




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

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.

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.

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.

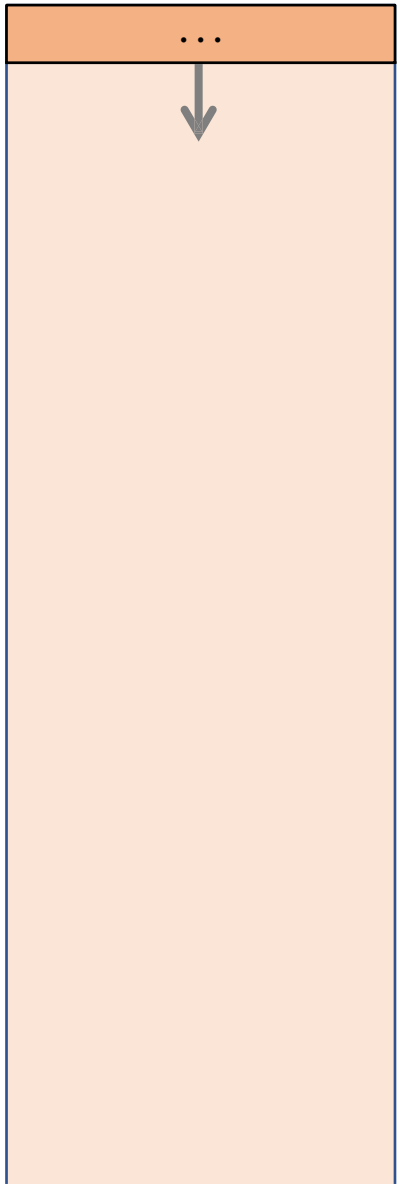
Terminology: **caller** function calls the **callee** function.

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()



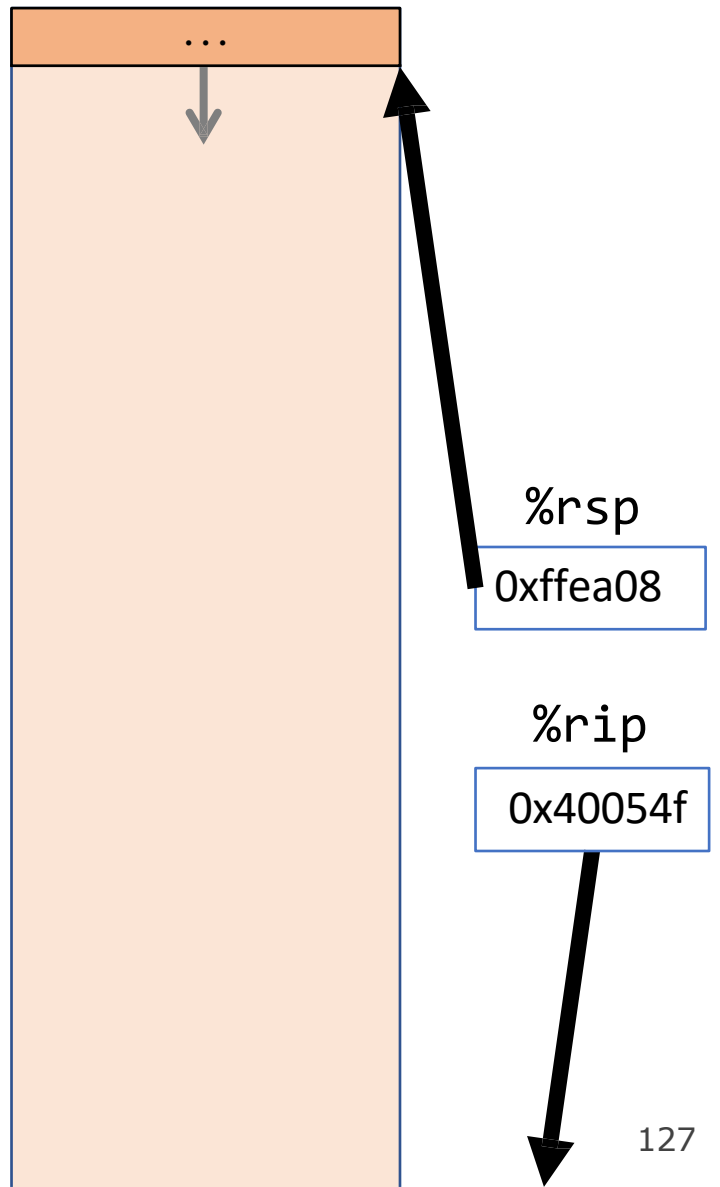
Parameters and Return

```
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()

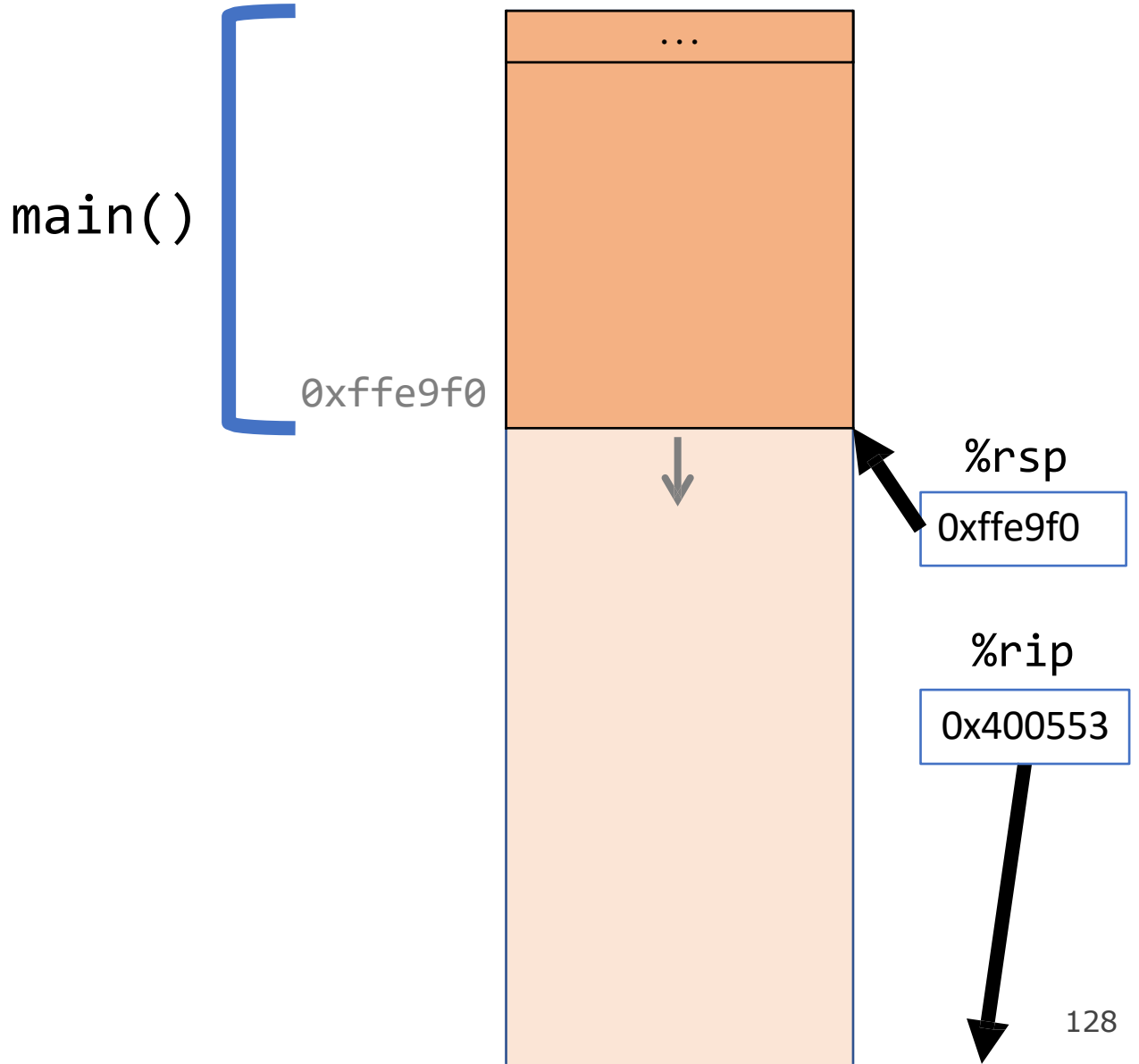


Parameters and Return

```
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)
```

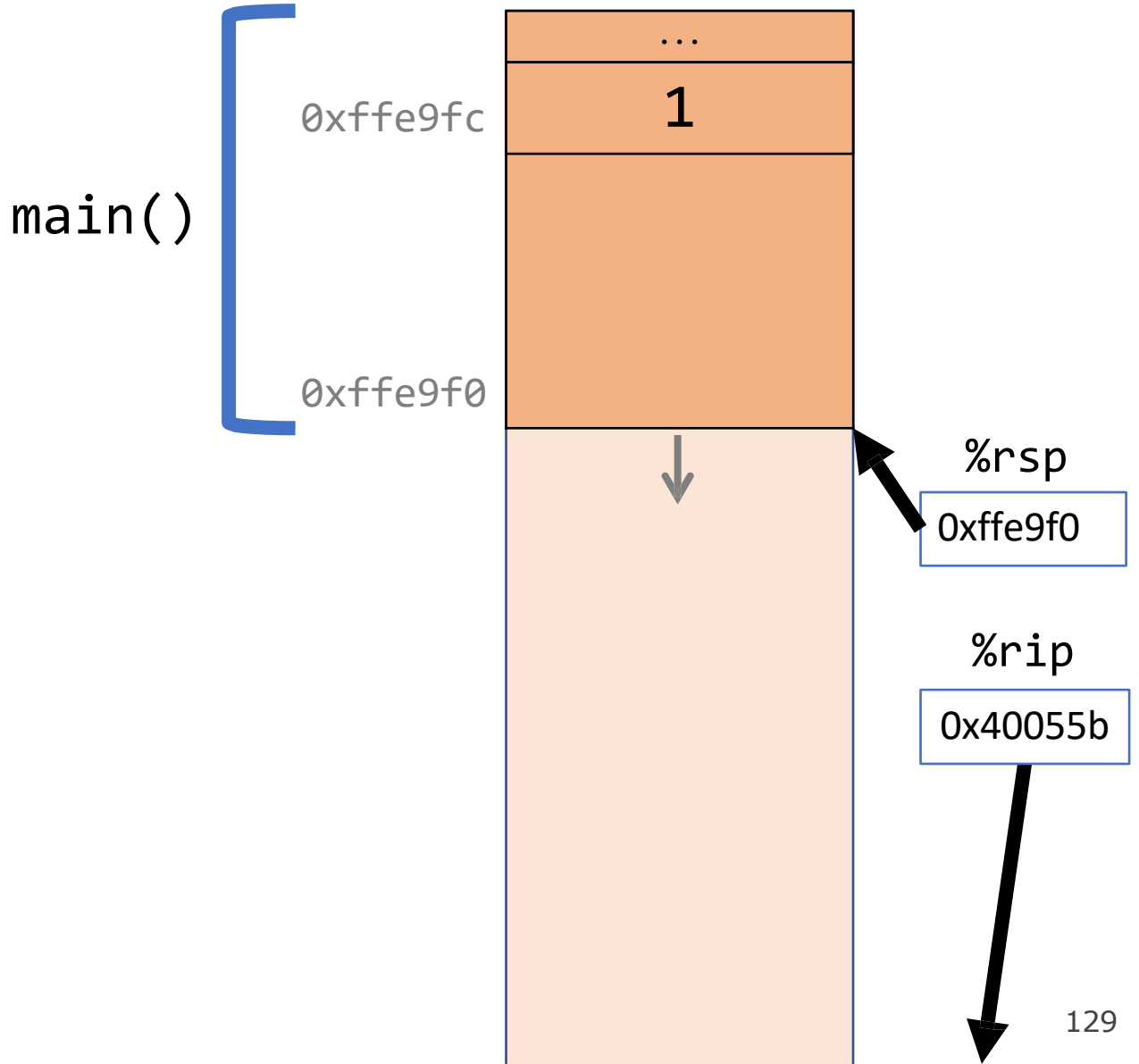


Parameters and Return

```
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)
```

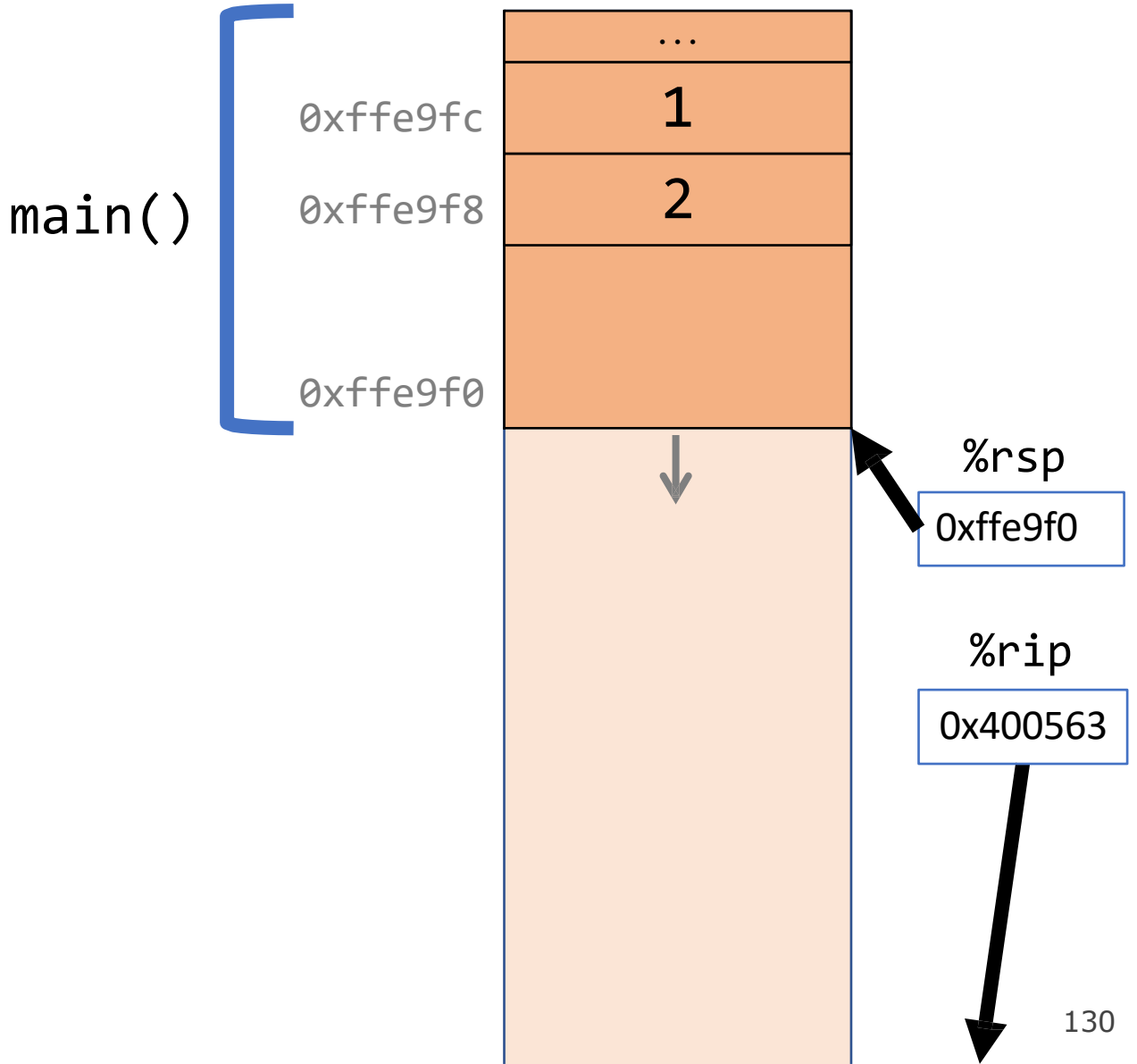


Parameters and Return

```
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)
```

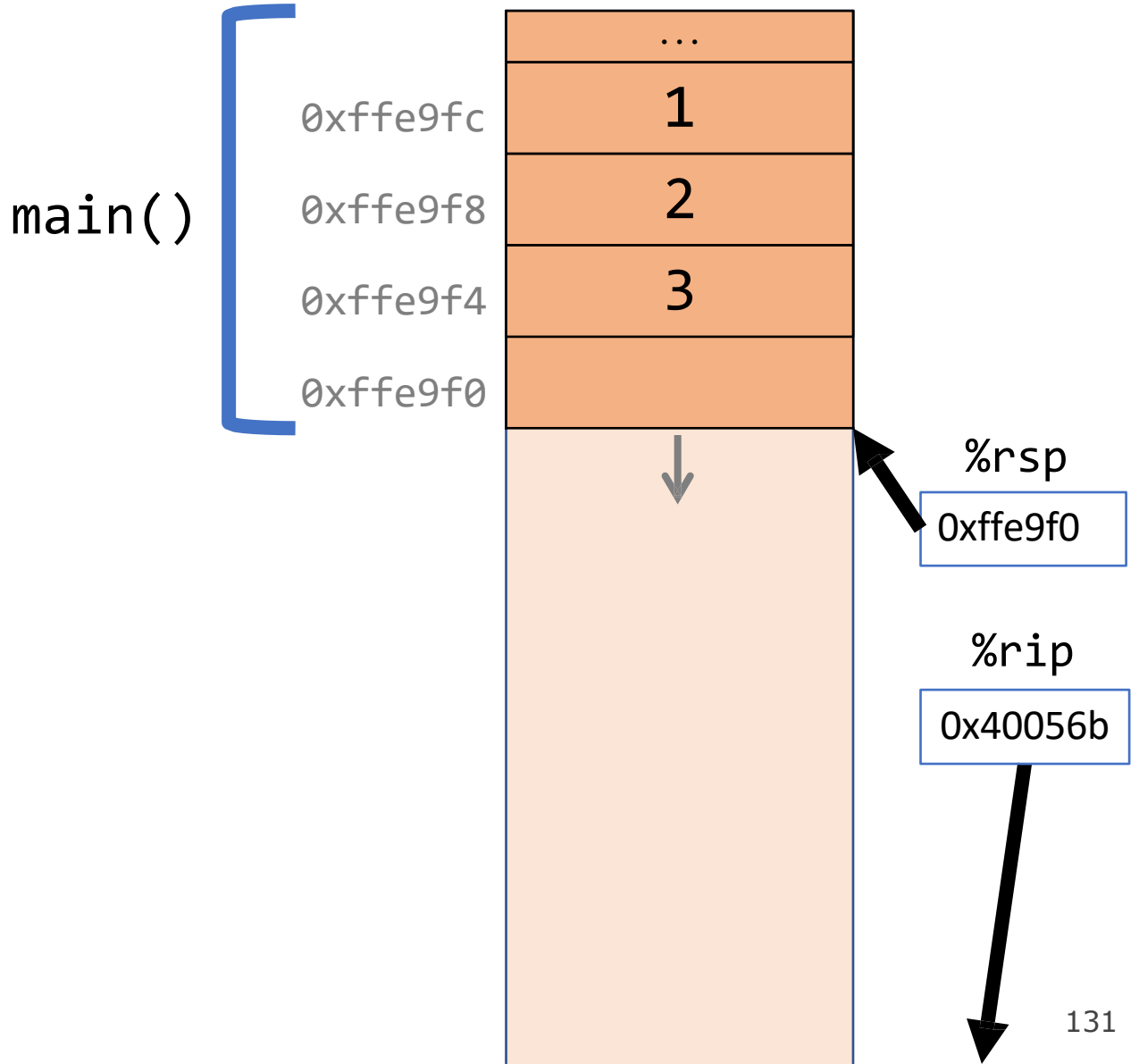


Parameters and Return

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

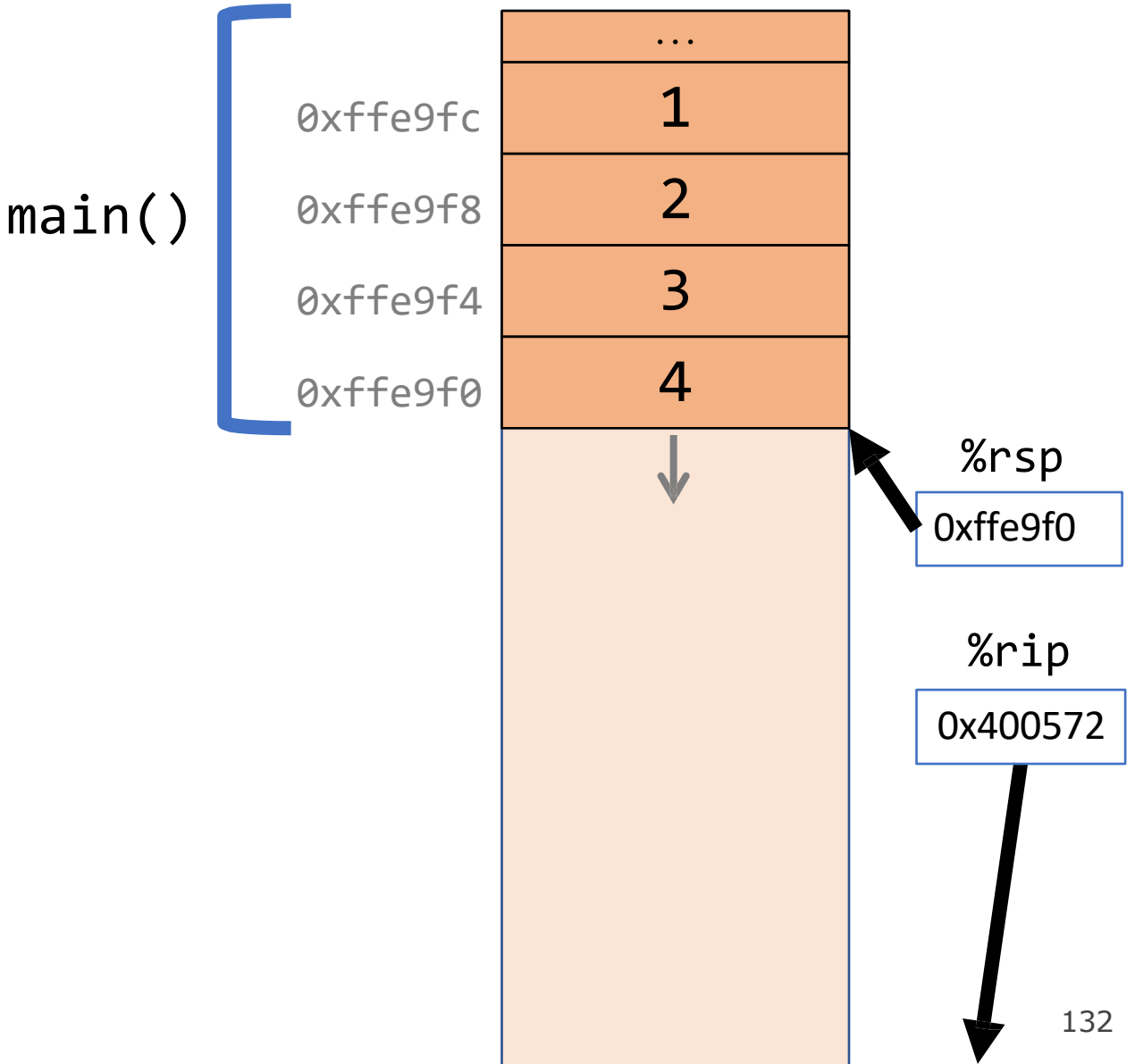


Parameters and Return

```
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  $0x3
```

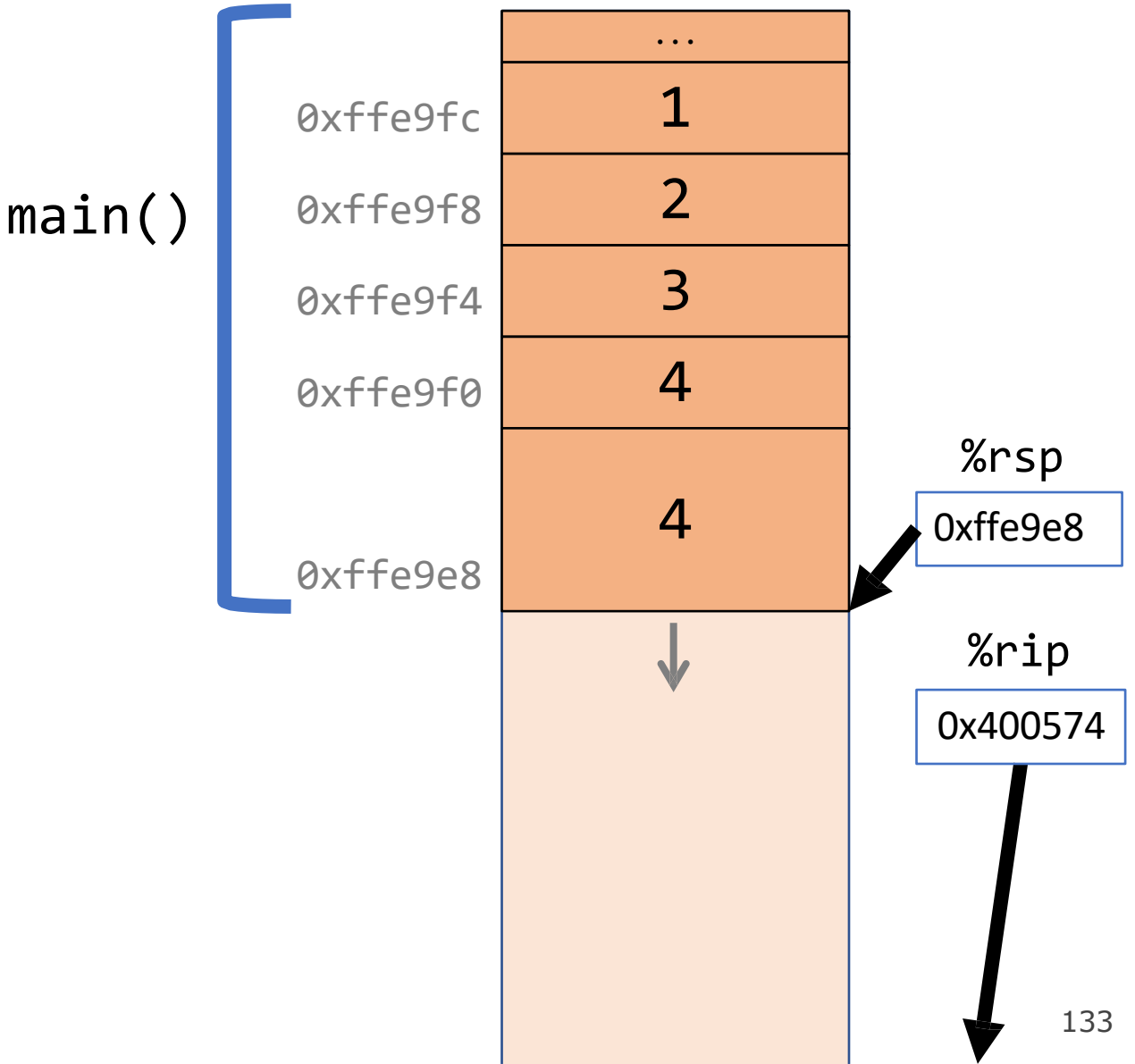


Parameters and Return

```
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,%r9d
```

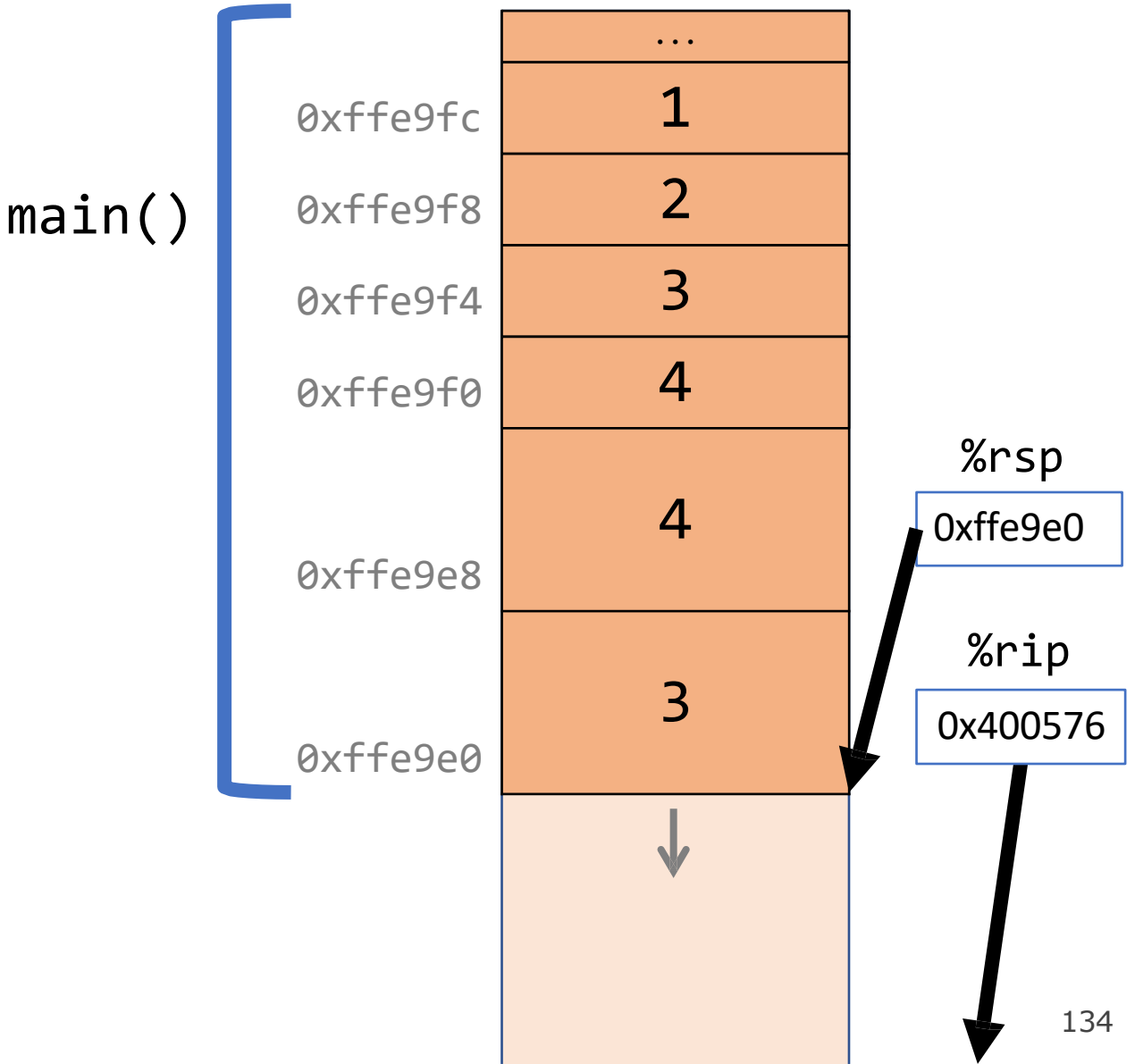


Parameters and Return

```
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) {
    …
}
```

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

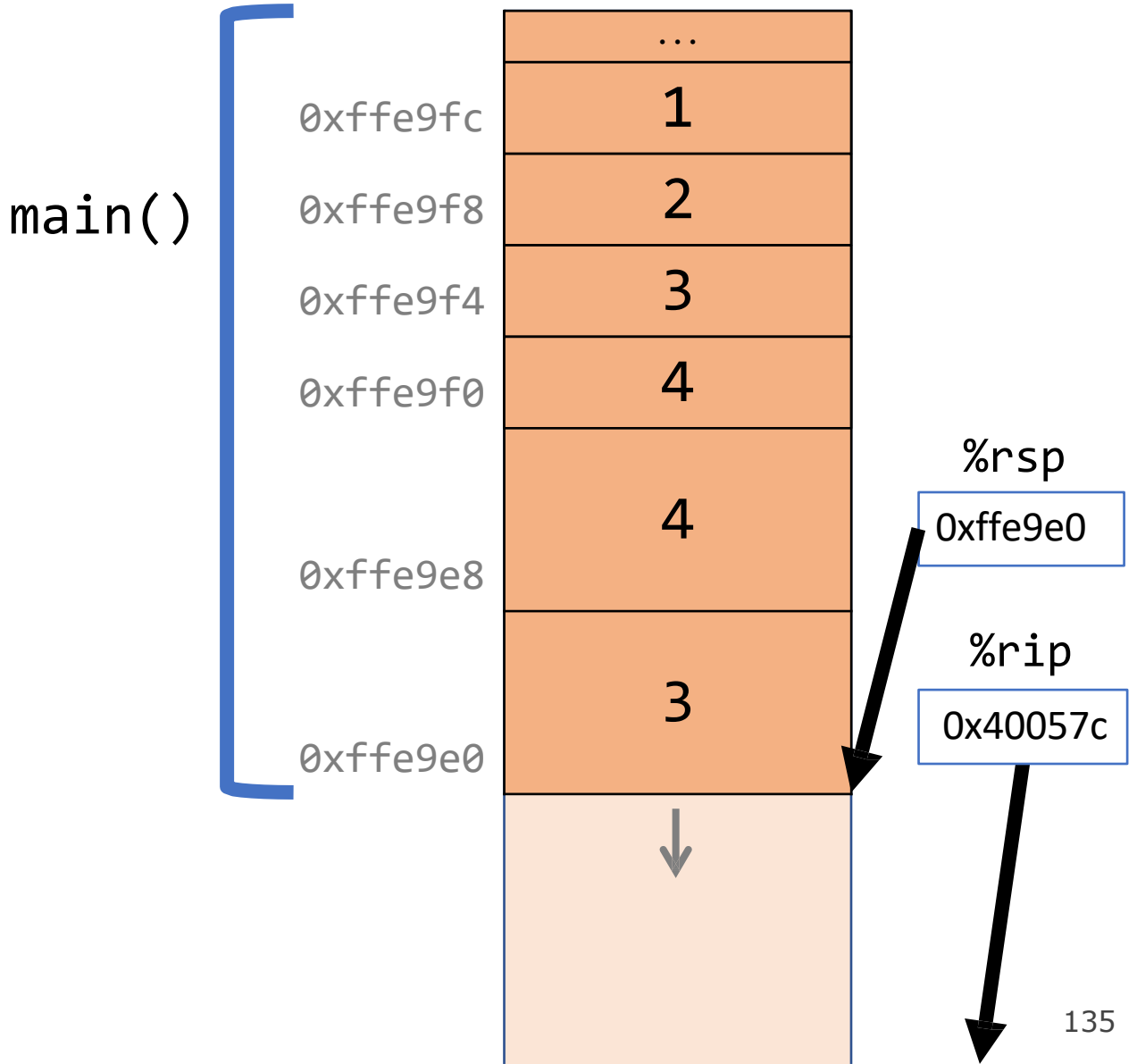


Parameters and Return

```
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) {
    …
}
```

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

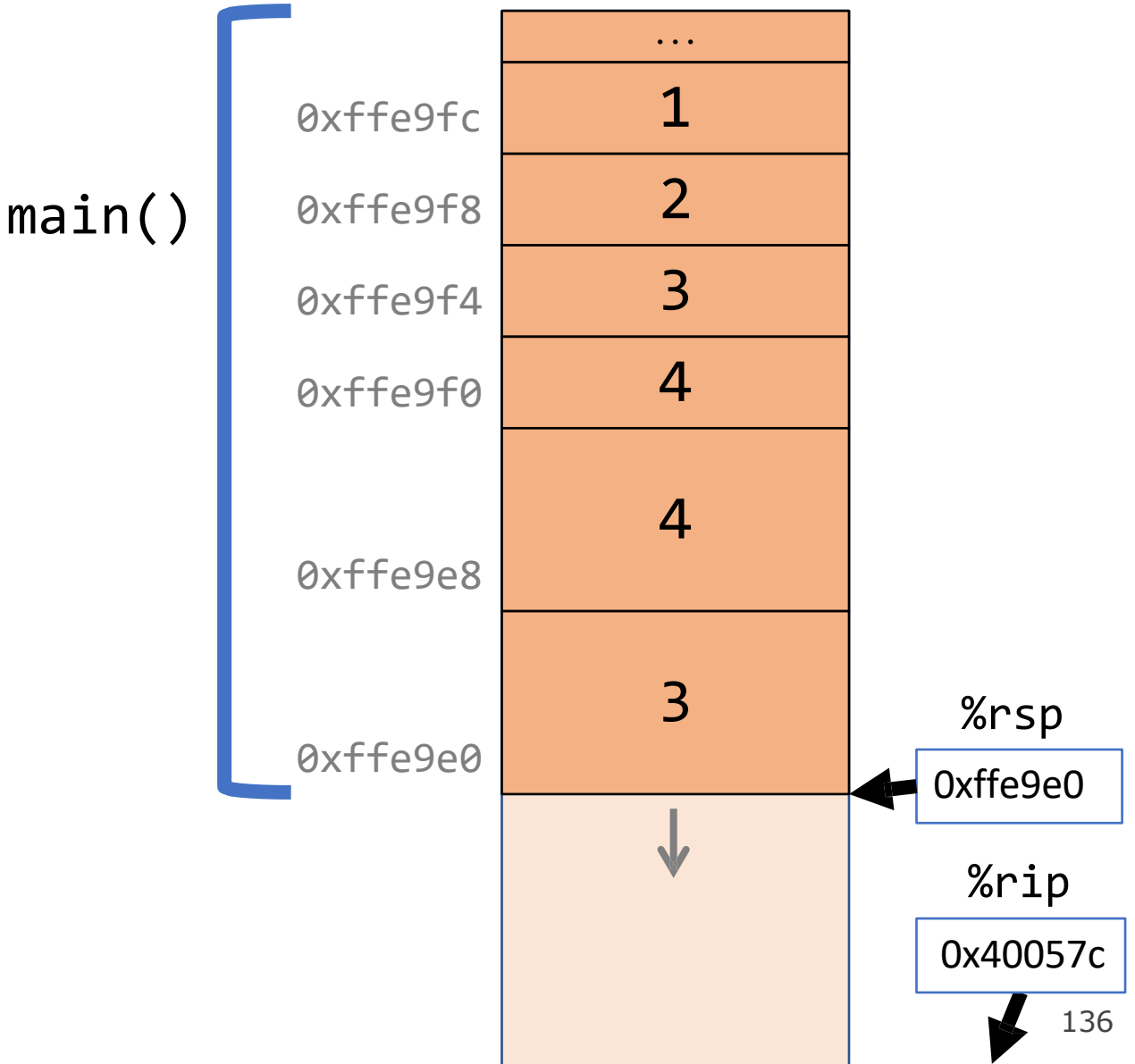


Parameters and Return

```
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
```

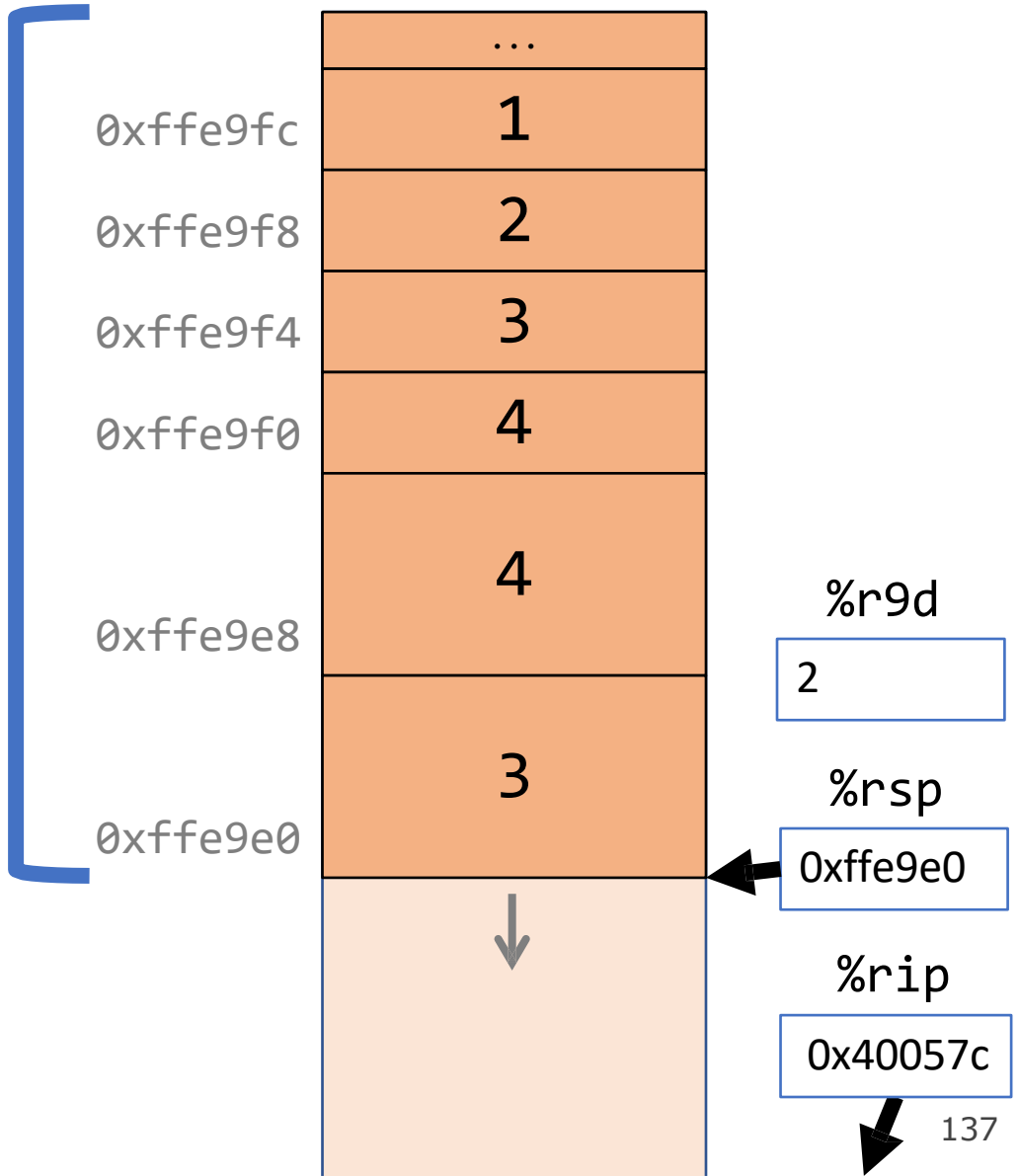


Parameters and Return

```
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
```

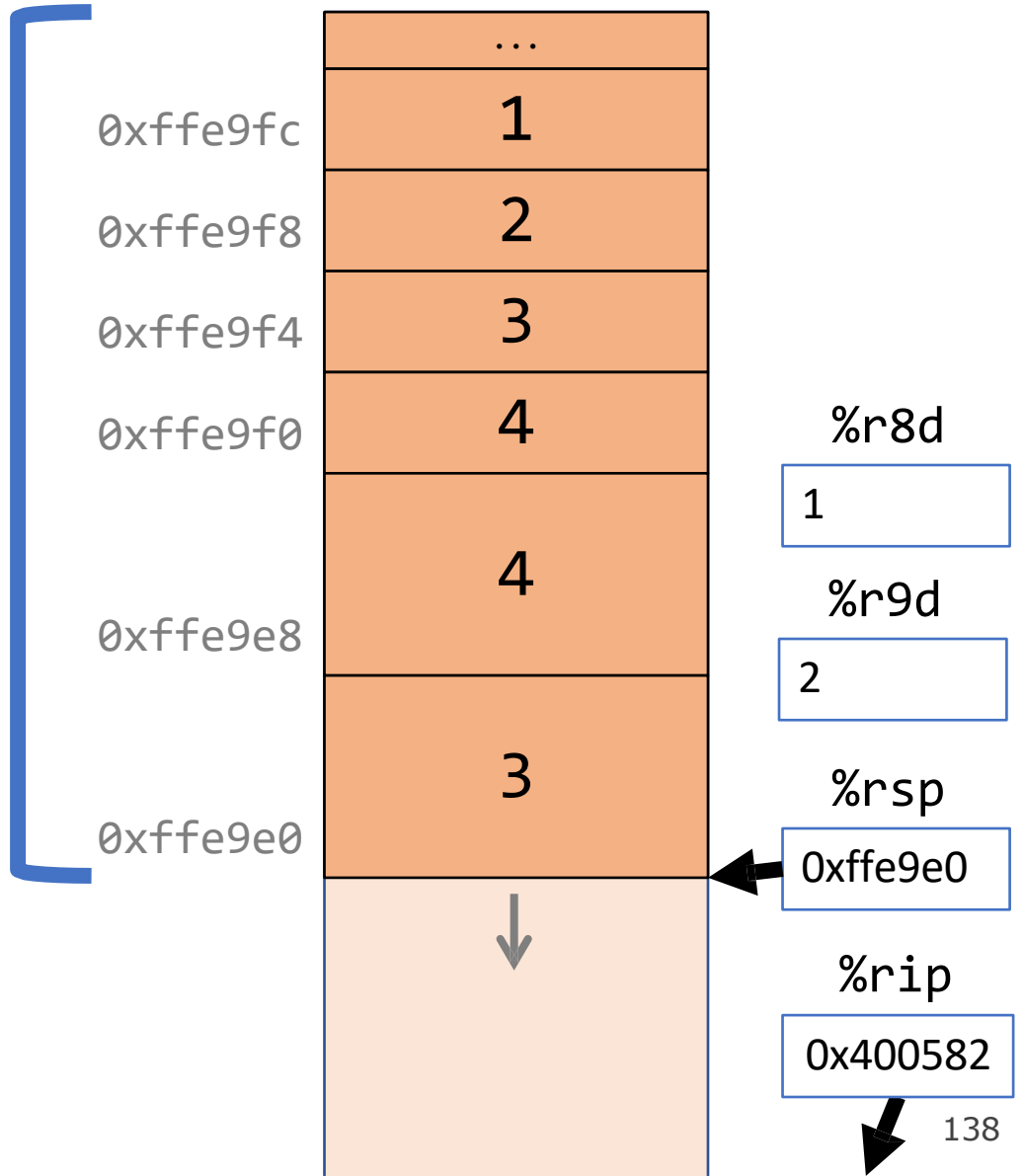


Parameters and Return

```
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
```

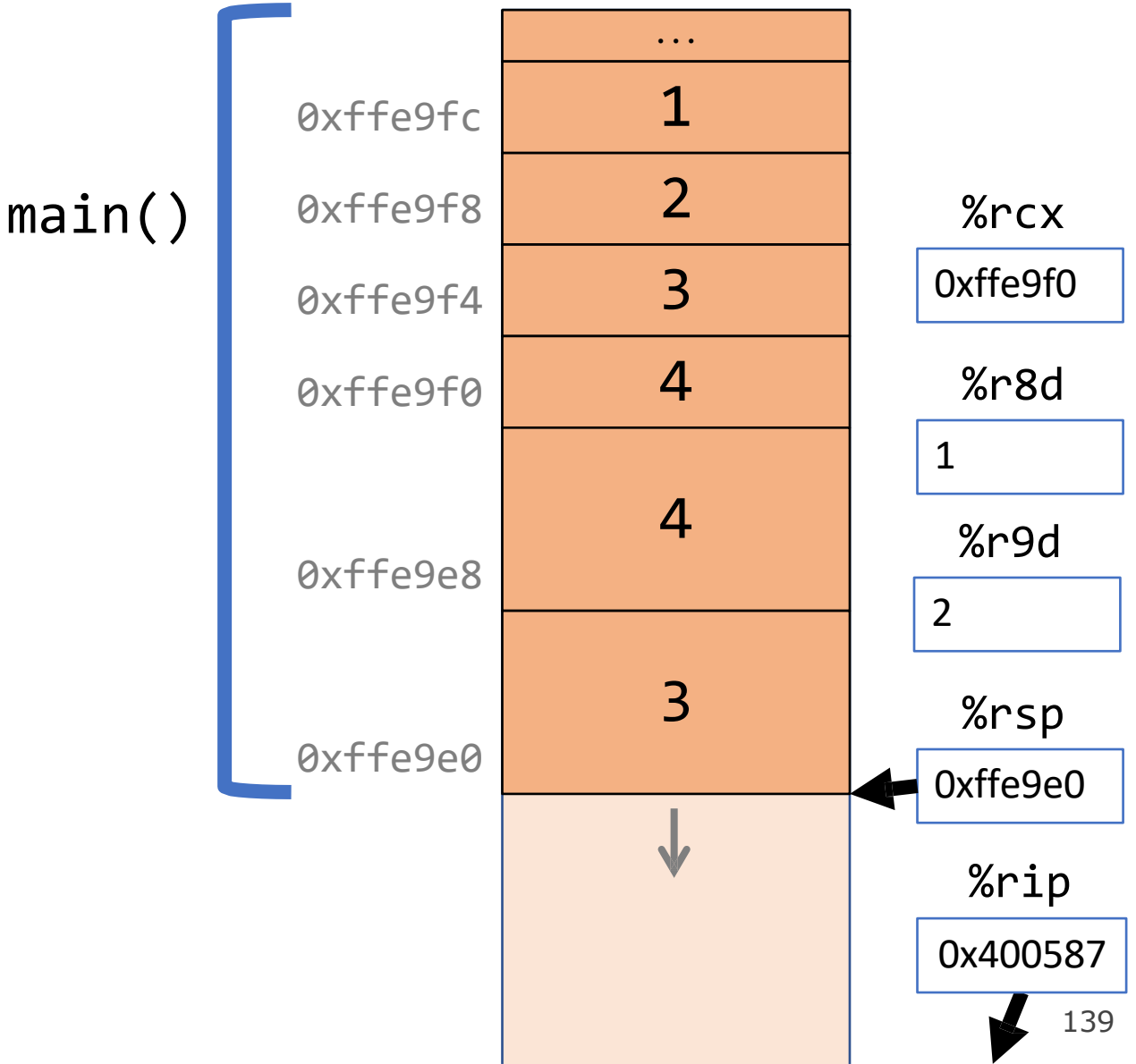


Parameters and Return

```
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
```

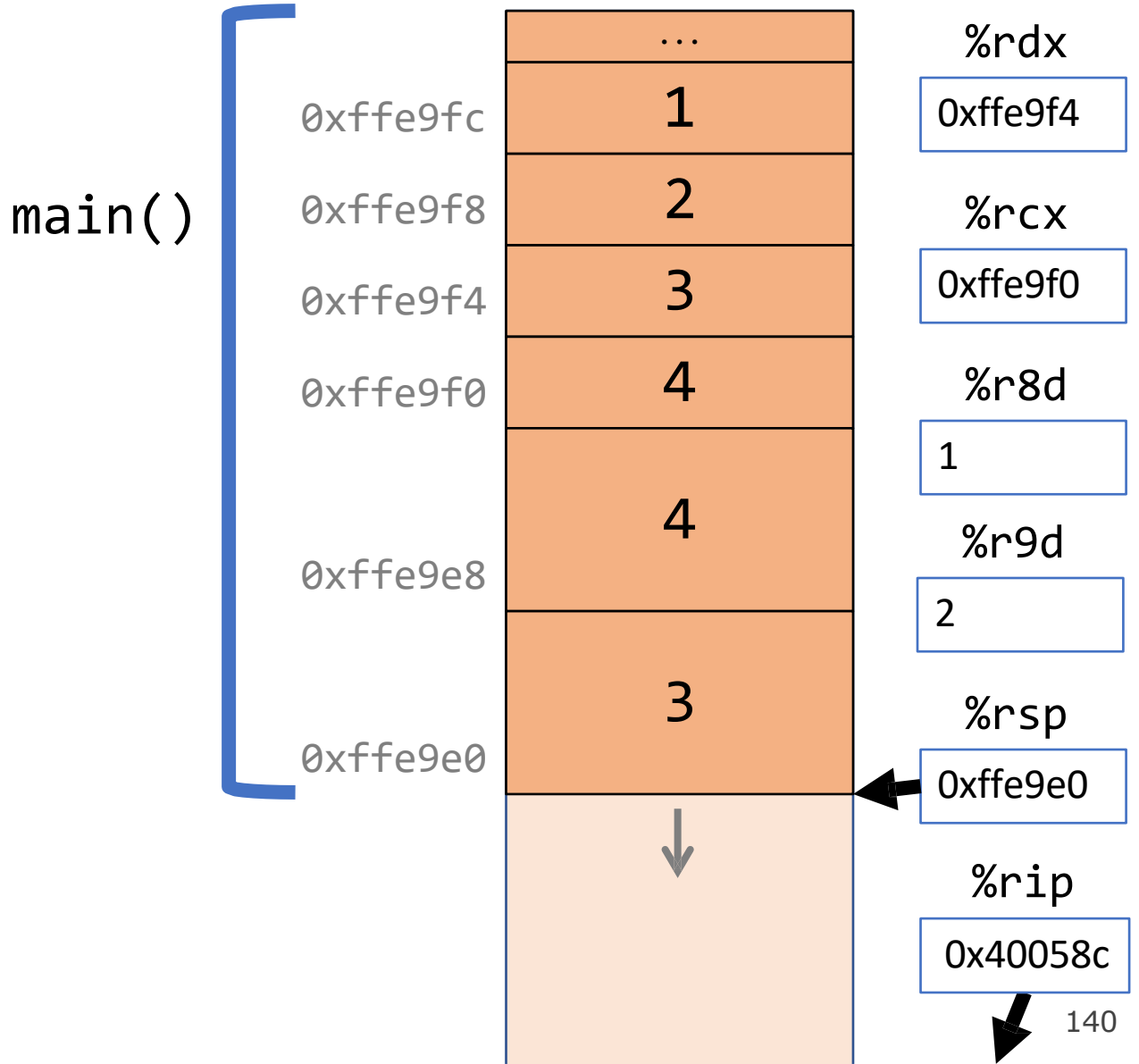


Parameters and Return

```
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
```

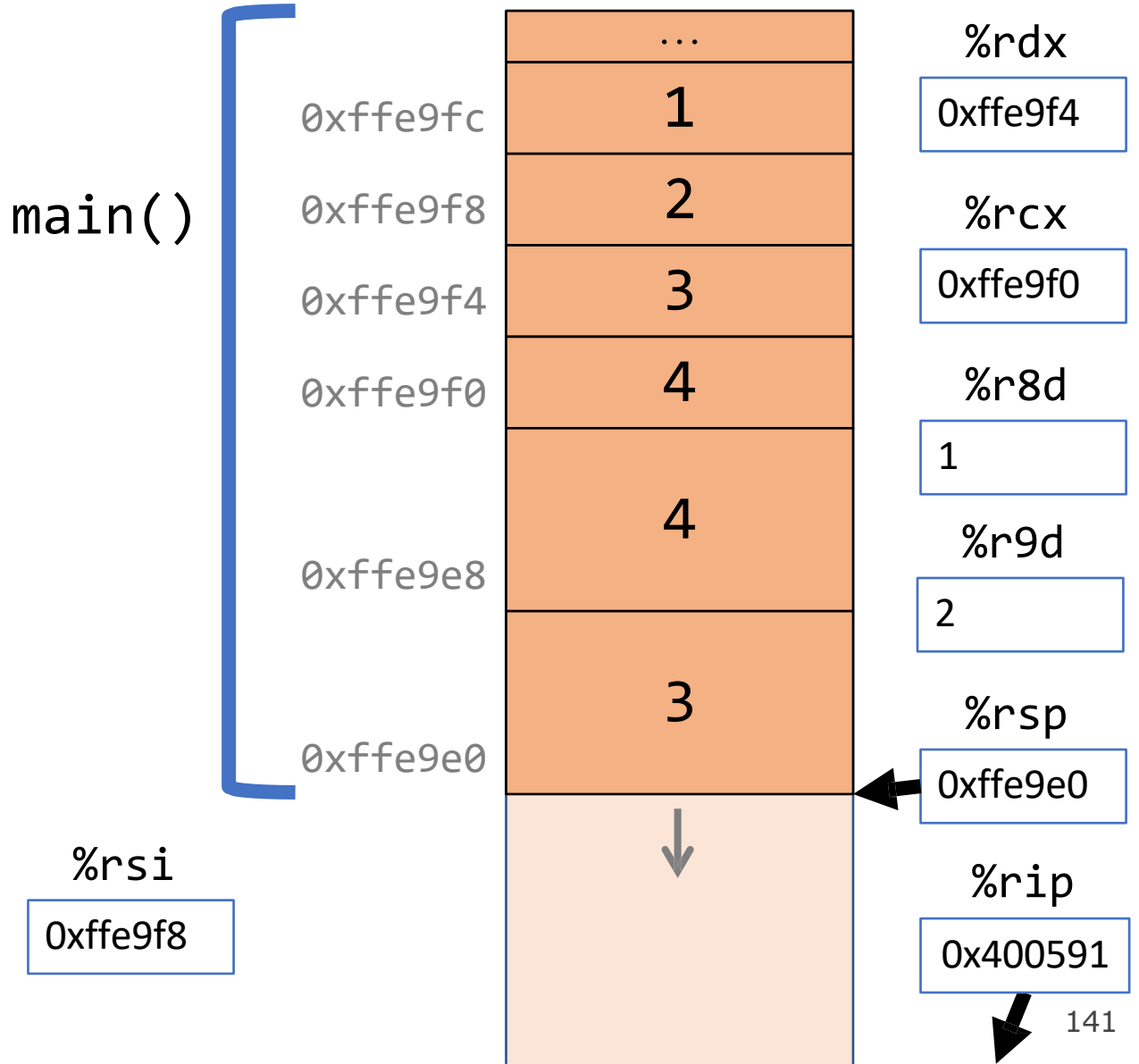


Parameters and Return

```
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>
```

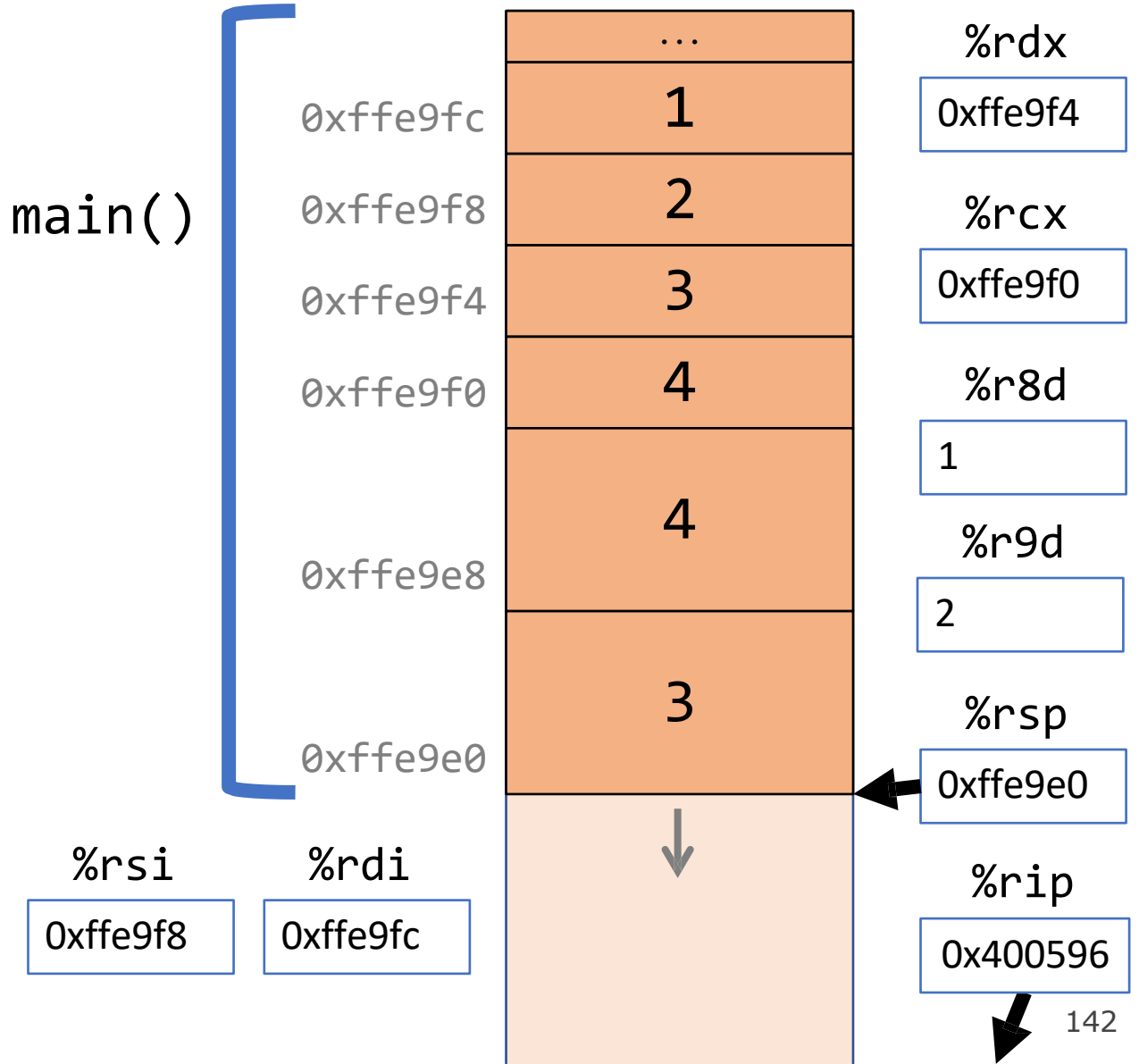


Parameters and Return

```
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
```

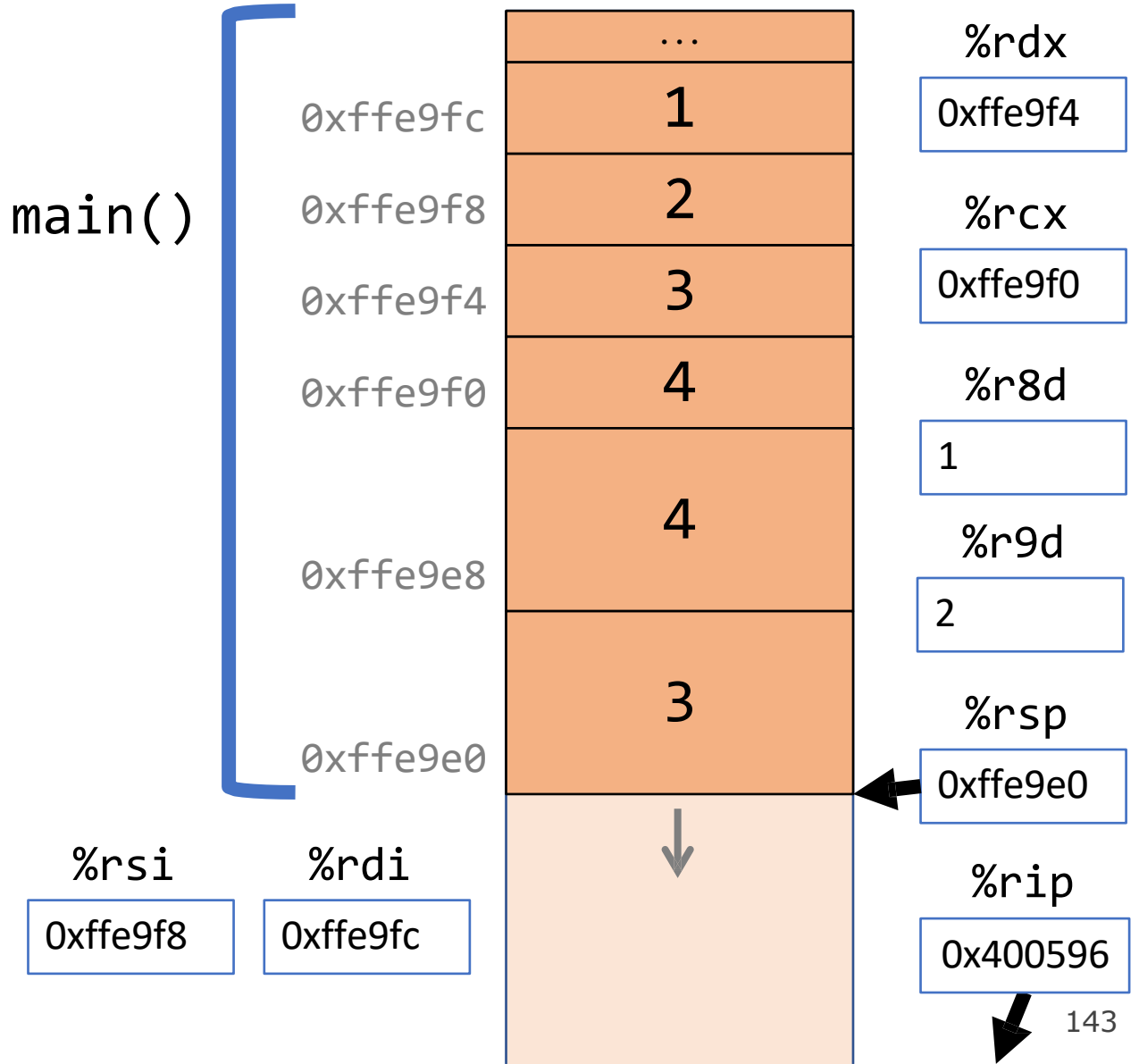


Parameters and Return

```
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
…
```

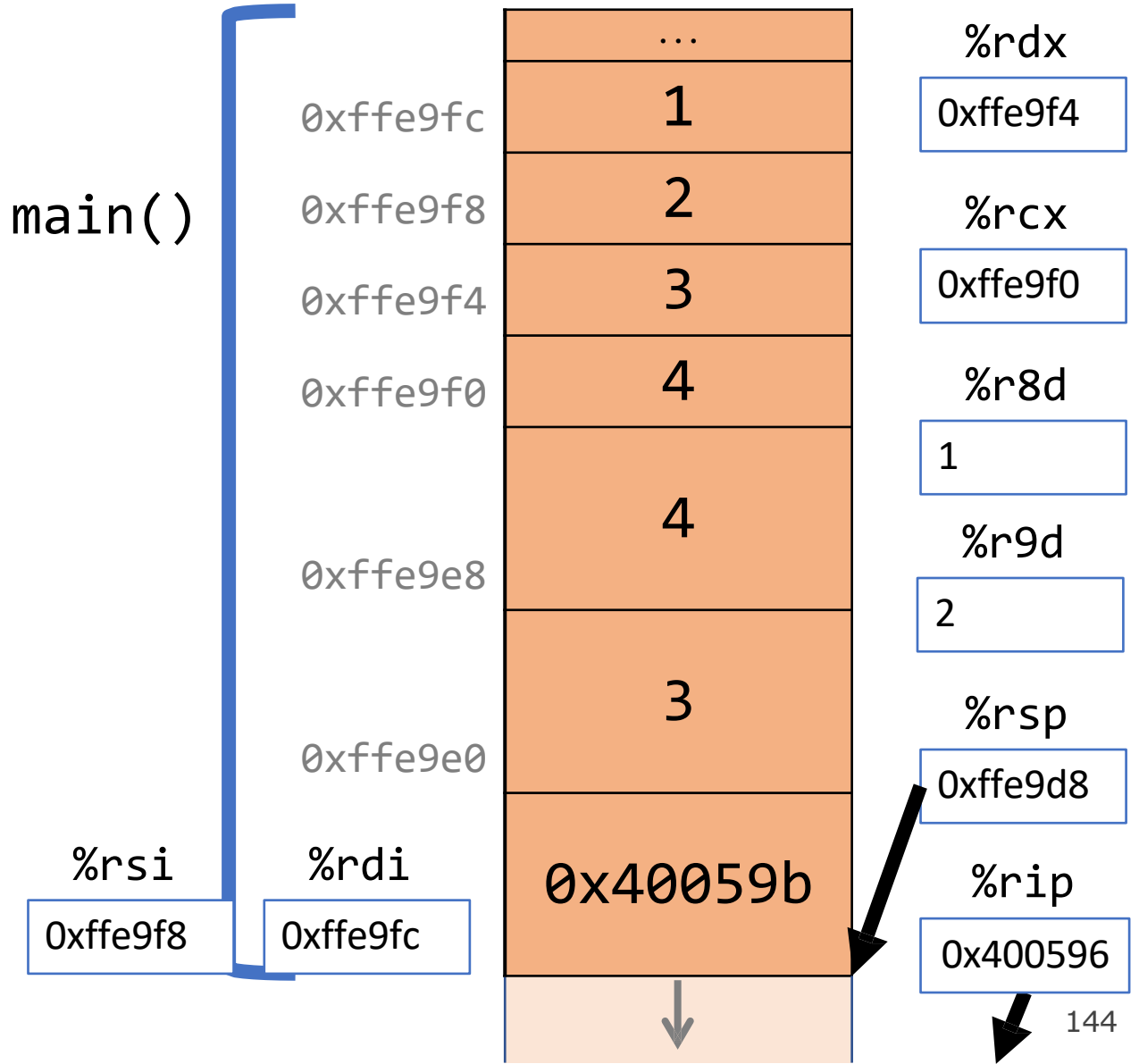


Parameters and Return

```
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
…
```



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)

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.

```
0000000000401136 <sum_array>:
  401136 <+0>:  mov    $0x0,%eax
  40113b <+5>:  mov    $0x0,%edx
  401140 <+10>: cmp    %esi,%eax
  401142 <+12>: jge    0x40114f <sum_array+25>
  401144 <+14>: movslq %eax,%rcx
  401147 <+17>: add    (%rdi,%rcx,4),%edx
  40114a <+20>: add    $0x1,%eax
  40114d <+23>: jmp    0x401140 <sum_array+10>
  40114f <+25>: mov    %edx,%eax
  401151 <+27>: retq
```

Optimizations you'll see

`nop`

- **`nop/nopl`** are "no-op" instructions – they do nothing!
- Intent: Make functions align on address boundaries that are nice multiples of 8.
- "Sometimes, doing nothing is how to be most productive" – Philosopher Nick

`mov %ebx,%ebx`

- Zeros out the top 32 register bits (because a `mov` on an e-register zeros out rest of 64 bits).

`xor %ebx,%ebx`

- Optimizes for performance as well as code size (read more here):

```
b8 00 00 00 00          mov $0x0,%eax
31 c0                   xor %eax,%eax
```

GCC For Loop Output

GCC Common For Loop Output

```
Initialization
Test
Jump past loop if success
Body
Update
Jump to test
```

Possible Alternative

```
Initialization
Jump to test
Body
Update
Test
Jump to body if success
```

GCC For Loop Output

GCC Common For Loop Output

```
Initialization
Test
Jump past loop if success
Body
Update
Jump to test
```

```
for (int i = 0; i < n; i++)          // n = 100
```

GCC For Loop Output

GCC Common For Loop Output

```
Initialization
Test
Jump past loop if success
Body
Update
Jump to test
```

```
for (int i = 0; i < n; i++)          // n = 100

Initialization
Test
No jump
Body
Update
Jump to test
Test
No jump
Body
Update
Jump to test
...
```

GCC For Loop Output

GCC Common For Loop Output

```
Initialization
Test
Jump past loop if success
Body
Update
Jump to test
```

```
for (int i = 0; i < n; i++)          // n = 100

Initialization
Test
No jump
Body
Update
Jump to test
Test
No jump
Body
Update
Jump to test
...
```

GCC For Loop Output

```
for (int i = 0; i < n; i++)          // n = 100

Initialization
Jump to test
Test
Jump to body
Body
Update
Test
Jump to body
Body
Update
Test
Jump to body
...
```

Possible Alternative

```
Initialization
Jump to test
Body
Update
Test
Jump to body if success
```

GCC For Loop Output

```
for (int i = 0; i < n; i++)          // n = 100

Initialization
Jump to test
Test
Jump to body
Body
Update
Test
Jump to body
Body
Update
Test
Jump to body
...
```

Possible Alternative

```
Initialization
Jump to test
Body
Update
Test
Jump to body if success
```

GCC For Loop Output

GCC Common For Loop Output

```
Initialization
Test
Jump past loop if passes
Body
Update
Jump to test
```

Possible Alternative

```
Initialization
Jump to test
Body
Update
Test
Jump to body if success
```

Which instructions are better when n = 0? n = 1000?

for (int i = 0; i < n; i++)

Optimizing Instruction Counts

- Both versions have the same **static** **instruction count** (# of written instructions).
- But they have different **dynamic** **instruction counts** (# of executed instructions when program is run).
  - If n = 0, left (GCC common output) is best b/c fewer instructions
  - If n is large, right (alternative) is best b/c fewer instructions
- The compiler may emit a static instruction count that is several times longer than an alternative, but it may be more efficient if loop executes many times.
- Does the compiler *know* that a loop will execute many times? (in general, no)
- So what if our code had loops that always execute a small number of times? How do we know when gcc makes a bad decision?
- (take EE108, EE180, CS316 for more!)

Optimizations

- **Conditional Moves** can sometimes eliminate “branches” (jumps), which are particularly inefficient on modern computer hardware.
- Processors try to *predict* the future execution of instructions for maximum performance. This is difficult to do with jumps.

Data Alignment

- Computer systems often put restrictions on the allowable addresses for primitive data types, requiring that the address for some objects must be a multiple of some value *K* (normally 2, 4, or 8).
- These ***alignment restrictions*** simplify the design of the hardware.
- For example, suppose that a processor always fetches 8 bytes from the memory system, and an address must be a multiple of 8. If we can guarantee that any `double` will be aligned to have its address as a multiple of 8, then we can read or write the values with a single memory access.
- For x86-64, Intel recommends the following alignments for best performance:

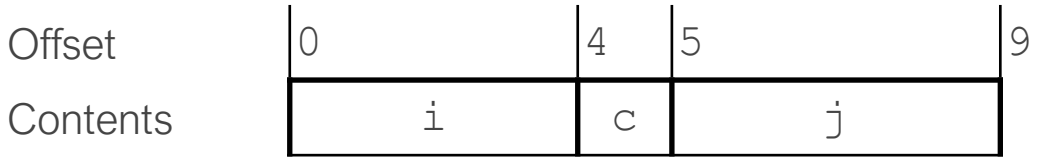
K	Types
1	`char`
2	`short`
4	`int, float`
8	`long, double, char *`

Data Alignment

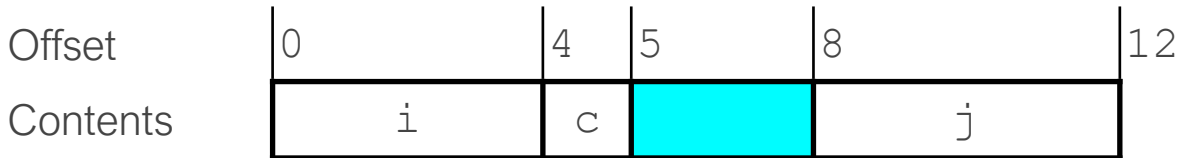
- The compiler enforces alignment by making sure that every data type is organized in such a way that every field within the struct satisfies the alignment restrictions.
- For example, let's look at the following struct:

```
struct S1 {
    int i;
    char c;
    int j;
};
```

- If the compiler used a minimal allocation:



- This would make it impossible to align fields `i` (offset 0) and `j` (offset 5). Instead, the compiler inserts a 3-byte gap between fields `c` and `j`:



- So, don't be surprised if your structs have a `sizeof()` that is larger than you expect!

Some Extra Reading

Key GDB Tips For Assembly

- Examine 4 giant words (8 bytes) on the stack:

```
(gdb) x/4g $rsp
0x7fffffffe870: 0x0000000000000005      0x0000000000400559
0x7fffffffe880: 0x0000000000000000      0x0000000000400575
```

- display/undisplay (prints out things every time you step/next)

```
(gdb) display/4w $rsp
1: x/4xw $rsp
0x7fffffffe8a8:
0xf7a2d830      0x00007fff      0x00000000      0x00000000
```

Key GDB Tips For Assembly

- `stepi`/`finish`: step into current function call/return to caller:

```
(gdb) finish
```

- Set register values during the run

```
(gdb) p $rdi = $rdi + 1
```

(Might be useful to write down the original value of $rdi somewhere)

- Tui things
  - `refresh`
  - `focus cmd` – use up/down arrows on gdb command line (vs `focus asm`, `focus regs`)
  - `layout regs`, `layout asm`

gdb tips

`layout split` (ctrl-x a: exit, ctrl-l: resize)	View C, assembly, and gdb (lab5)
`info reg`	Print all registers
`p $eax`	Print register value
`p $eflags`	Print all condition codes currently set
`b *0x400546`	Set breakpoint at assembly instruction
`b *0x400550 if $eax > 98`	Set **conditional breakpoint**
`ni`	Next assembly instruction
`si`	Step into assembly instruction (will step into function calls)

gdb tips

`p/x $rdi`	Print register value in hex
`p/t $rsi`	Print register value in binary
`x $rdi`	Examine the byte stored at this address
`x/4bx $rdi`	Examine 4 bytes starting at this address
`x/4wx $rdi`	Examine 4 ints starting at this address

Array Allocation and Access

- Arrays in C map in a fairly straightforward way to X86 assembly code, thanks to the addressing modes available in instructions.
- When we perform pointer arithmetic, the assembly code that is produced will have address computations built into them.
- Optimizing compilers are ***very*** good at simplifying the address computations (in lab you will see another optimizing compiler benefit in the form of division — if the compiler can avoid dividing, it will!). Because of the transformations, compiler-generated assembly for arrays often doesn't look like what you are expecting.
- Consider the following form of a data type *T* and integer constant *N:*

  T `A[`*N*`]`

- The starting location is designated as x_A
- The declaration allocates *N* `* sizeof(`*T*`)` bytes, and gives us an identifier that we can use as a pointer (but it isn't a pointer!), with a value of x_A.

Array Allocation and Access

- Example:

```
char      A[12];
char     *B[8];
int       C[6];
double   *D[5]
```

Array	Element Size	Total Size	Start address	Element i
A	1	12	x_A	$x_A + i$
B	8	64	x_B	$x_B + 8i$
C	4	24	x_C	$x_C + 4i$
D	8	40	x_D	$x_D + 8i$

- The memory referencing operations in x86-64 are designed to simplify array access. Suppose we wanted to access `C[3]` above. If the address of C is in register `%rdx`, and `3` is in register `%rcx`
- The following copies `C[3]` into `%eax`,

```
movl  (%rdx,%rcx,4),  %eax
```


Pointer Arithmetic

- C allows arithmetic on pointers, where the computed value is calculated according to the size of the data type referenced by the pointer.
- The array reference `A[i]` is identical to `*(A+i)`
- Example: if the address of array E is in `%rdx`, and the integer index, `i`, is in `%rcx`, the following are some expressions involving E:

Expression	Type	Value	Assembly Code
`E`	int *	x_E	`movq %rdx, %rax`
`E[0]`	int	M[x_E]	`movl (%rdx), %eax`
`E[i]`	int	M[x_E+4i]	`movl (%rdx,%rcx,4) %eax`
`&E[2]`	int *	x_E+8	`leaq 8(%rdx), %rax`
`E+i-1`	int *	x_E+4i-4	`leaq -4(%rdx,%rcx,4), %rax`
`*(E+i-3)`	int	M[x_E+4i-12]	`movl -12(%rdx,%rcx,4) %eax`
`&E[i]-E`	long	i	`movq %rcx,%rax`

Pointer Arithmetic

- Practice: x_S is the address of a `short` integer array, S, stored in `%rdx`, and a long integer index, `i`, is stored in register `%rcx`.
- For each of the following expressions, give its type, a formula for its value, and an assembly-code implementation. The result should be stored in `%rax` if it is a pointer, and the result should be in register `%ax` if it has a data type `short`.

Expression	Type	Value	Assembly Code
`S+1`	`short *`	$x_S + 2$	`leaq 2(%rdx),%rax`
`S[3]`	`short`	$M[x_S + 6]$	`movw 6(%rdx),%ax`
`&S[i]`	`short *`	$x_S + 2i$	`leaq (%rdx,%rcx,2),%rax`
`S[4*i+1]`	`short`	$M[x_S + 8i + 2]$	`movw 2(%rdx,%rcx,8),%ax`
`S+i-5`	`short *`	$x_S + 2i - 10$	`leaq -10(%rdx,%rcx,2),%rax`

References and Advanced

- References:
  - Stanford guide to x86-64: https://web.stanford.edu/class/cs107/guide/x86-64.html
  - CS107 one-page of x86-64: https://web.stanford.edu/class/cs107/resources/onepage_x86-64.pdf
  - gdbtui: https://beej.us/guide/bggdb/
  - More gdbtui: https://sourceware.org/gdb/onlinedocs/gdb/TUI.html
  - Compiler explorer: https://gcc.godbolt.org
- Advanced Reading:
  - Stack frame layout on x86-64: https://eli.thegreenplace.net/2011/09/06/stack-frame-layout-on-x86-64
  - x86-64 Intel Software Developer manual: https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf
  - history of x86 instructions: https://en.wikipedia.org/wiki/X86_instruction_listings
  - x86-64 Wikipedia: https://en.wikipedia.org/wiki/X86-64