

CS107, Lecture 13

Control Flow: When in doubt just JMP!

Reading: B&O 3.1-3.4

Lecture Plan

- 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

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 fails

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	

Lecture Plan

- 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

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 fails
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)



`gdb tips`



`layout split` (ctrl-x a: exit,
ctrl-l: resize)

`info reg`

`p $eax`

`p $eflags`

`b *0x400546`

`b *0x400550 if $eax > 98`

`ni`

`si`

View C, assembly, and gdb (lab5)

Print all registers

Print register value

Print all condition codes currently set

Set breakpoint at assembly instruction

Set **conditional breakpoint**

Next assembly instruction

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

Lecture Plan

- 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**

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 conditionally set a byte to 0 or 1.

- Reads current state of flags
- Destination is a single-byte register (e.g., %al) or single-byte memory location
- 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
cmovz 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		Nonnegative (SF = 0)
cmovg S,R	cmovnl	Greater (signed >) (ZF = 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)

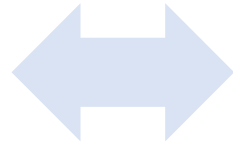
Recap

- 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

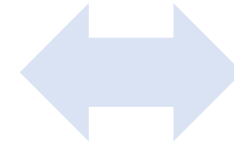
★ How to remember cmp/jmp

- `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
<code>jmp Label</code>		1	Direct jump
<code>jmp *Operand</code>		1	Indirect jump
<code>je Label</code>	<code>jz</code>	ZF	Equal / zero
<code>jne Label</code>	<code>jnz</code>	$\sim$ ZF	Not equal / not zero
<code>js Label</code>		SF	Negative
<code>jns Label</code>		$\sim$ SF	Nonnegative
<code>jg Label</code>	<code>jnle</code>	$\sim$ (SF $\wedge$ OF) & $\sim$ ZF	Greater (signed >)
<code>jge Label</code>	<code>jnl</code>	$\sim$ (SF $\wedge$ OF)	Greater or equal (signed $\geq$)
<code>jl Label</code>	<code>jnge</code>	SF $\wedge$ OF	Less (signed <)
<code>jle Label</code>	<code>jng</code>	(SF $\wedge$ OF) ZF	Less or equal (signed $\leq$)
<code>ja Label</code>	<code>jnbe</code>	$\sim$ CF & $\sim$ ZF	Above (unsigned >)
<code>jae Label</code>	<code>jnb</code>	$\sim$ CF	Above or equal (unsigned $\geq$)
<code>jb Label</code>	<code>jnae</code>	CF	Below (unsigned <)
<code>jbe Label</code>	<code>jna</code>	CF ZF	Below or equal (unsigned $\leq$)

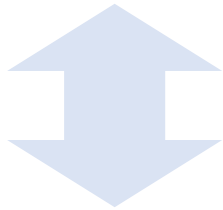
Figure 3.15 The jump instructions. These instructions jump to a labeled destination when the jump condition holds. Some instructions have "synonyms," alternate names for the same machine instruction.

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



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
```

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	

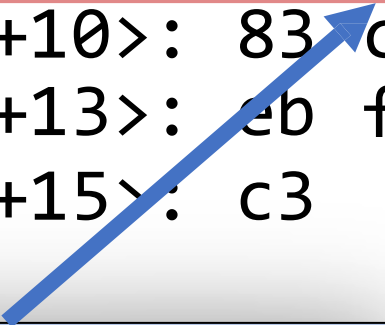
%rip

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	

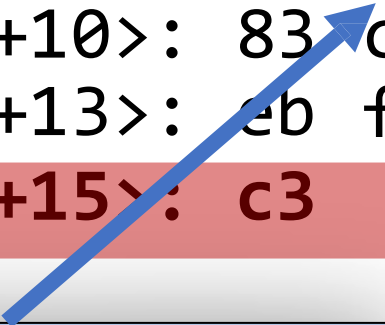


0x05 is the number of instruction bytes to jump relative to %rip.

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

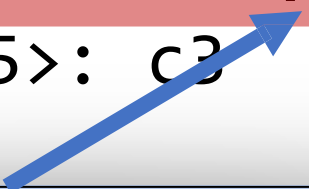
%rip

```
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
```

0xeb means **jmp**.

%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
```

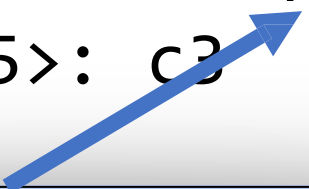


0xf6 is the number of instruction bytes to jump relative to %rip. This is -10 (in two's complement!).

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:

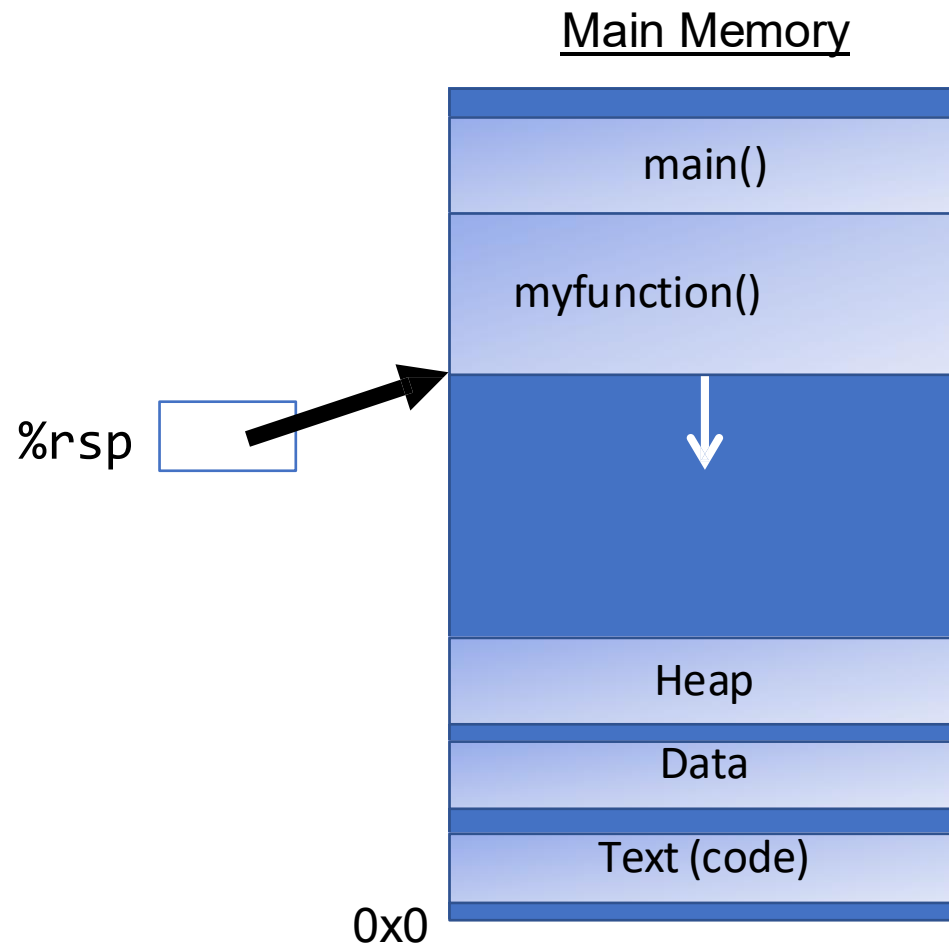
- **Pass Control** – %rip must be adjusted to execute the callee's instructions, and then resume the caller's instructions afterwards.
- **Pass Data** – we must pass any parameters and receive any return value.
- **Manage Memory** – we must handle any space needs of the callee on the stack.

How does assembly
interact with the stack?

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

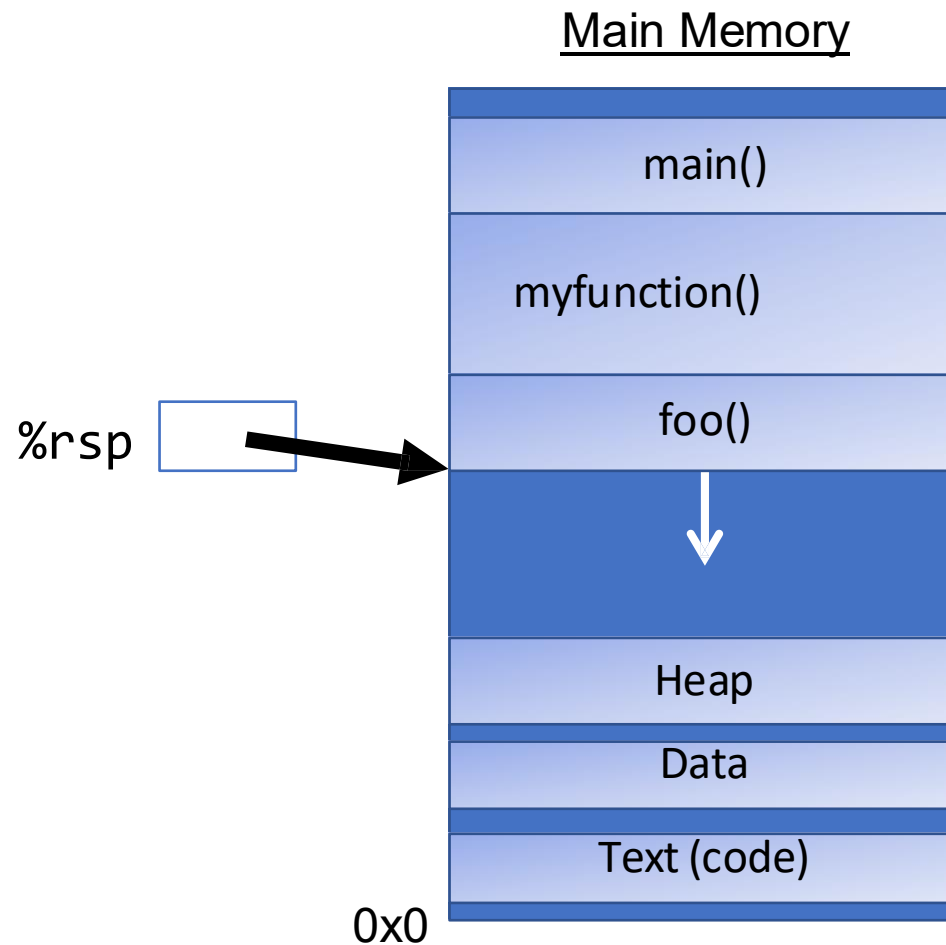
%rsp

- **%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).



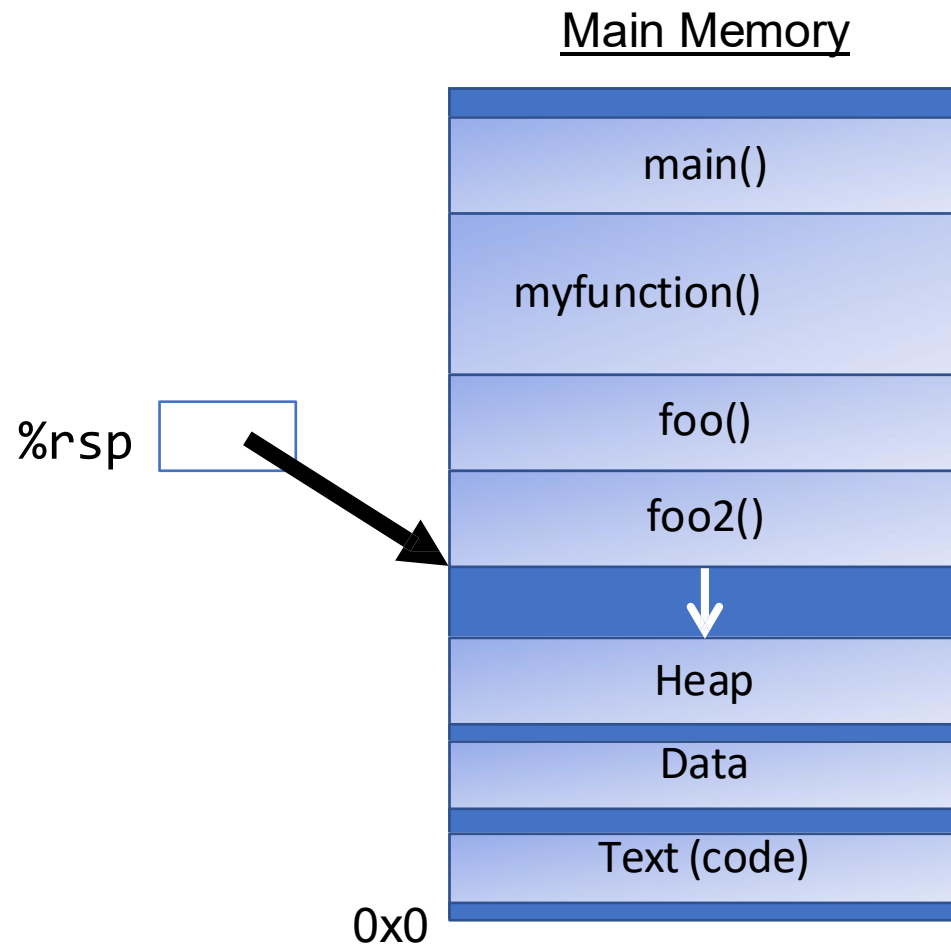
%rsp

- **%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).



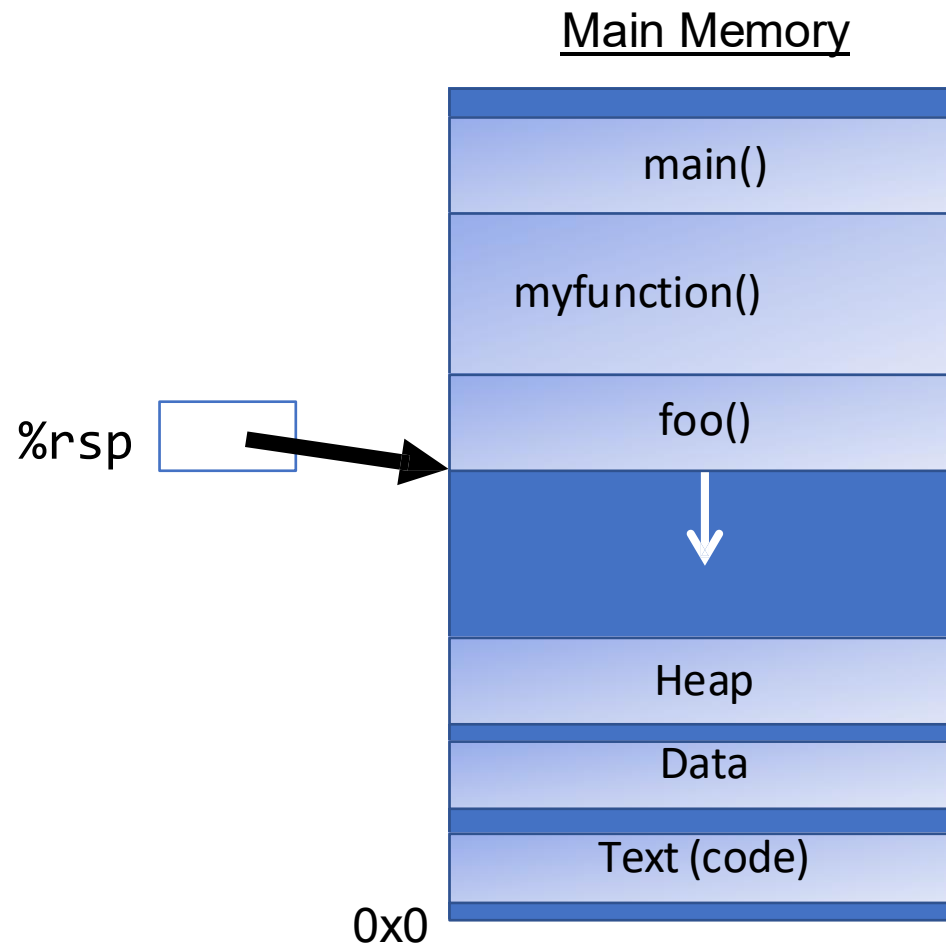
%rsp

- **%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).



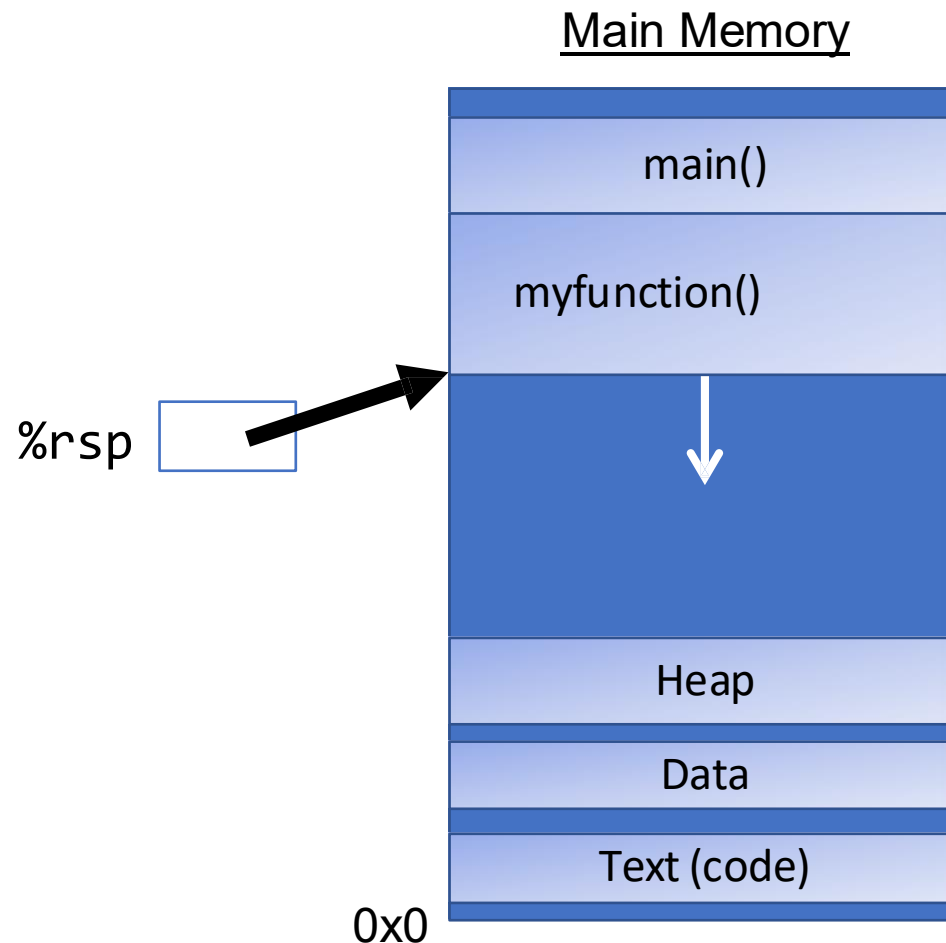
%rsp

- **%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	$R[\%rsp] \leftarrow R[\%rsp] - 8;$ $M[R[\%rsp]] \leftarrow S$

push

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

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

push

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

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

push

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

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

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

pop

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

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

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

pop

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

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

- This behavior is equivalent to the following, but popq is a shorter instruction:
movq (%rsp), D
addq \$8, %rsp
- Sometimes, you'll see instructions just explicitly increment the stack pointer to pop data.

Stack Example

Initially

%rax	0x123
%rdx	0
%rsp	0x108

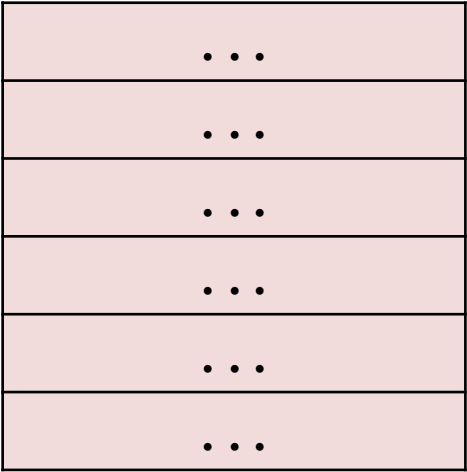
pushq %rax

%rax	0x123
%rdx	0
%rsp	0x100

popq %rdx

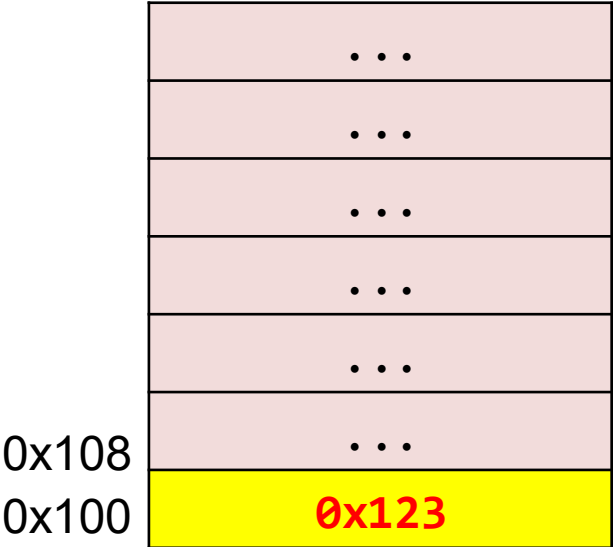
%rax	0x12
%rdx	0x123
%rsp	0x108

Stack “bottom”



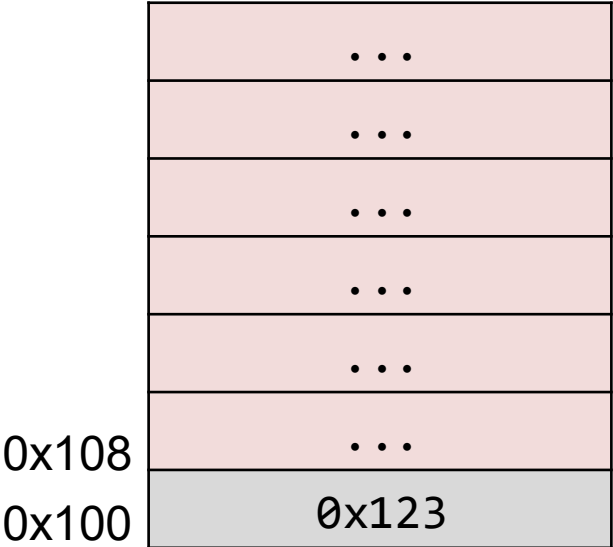
Stack “top”

Stack “bottom”



Stack “top”

Stack “bottom”



Calling Functions In Assembly

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

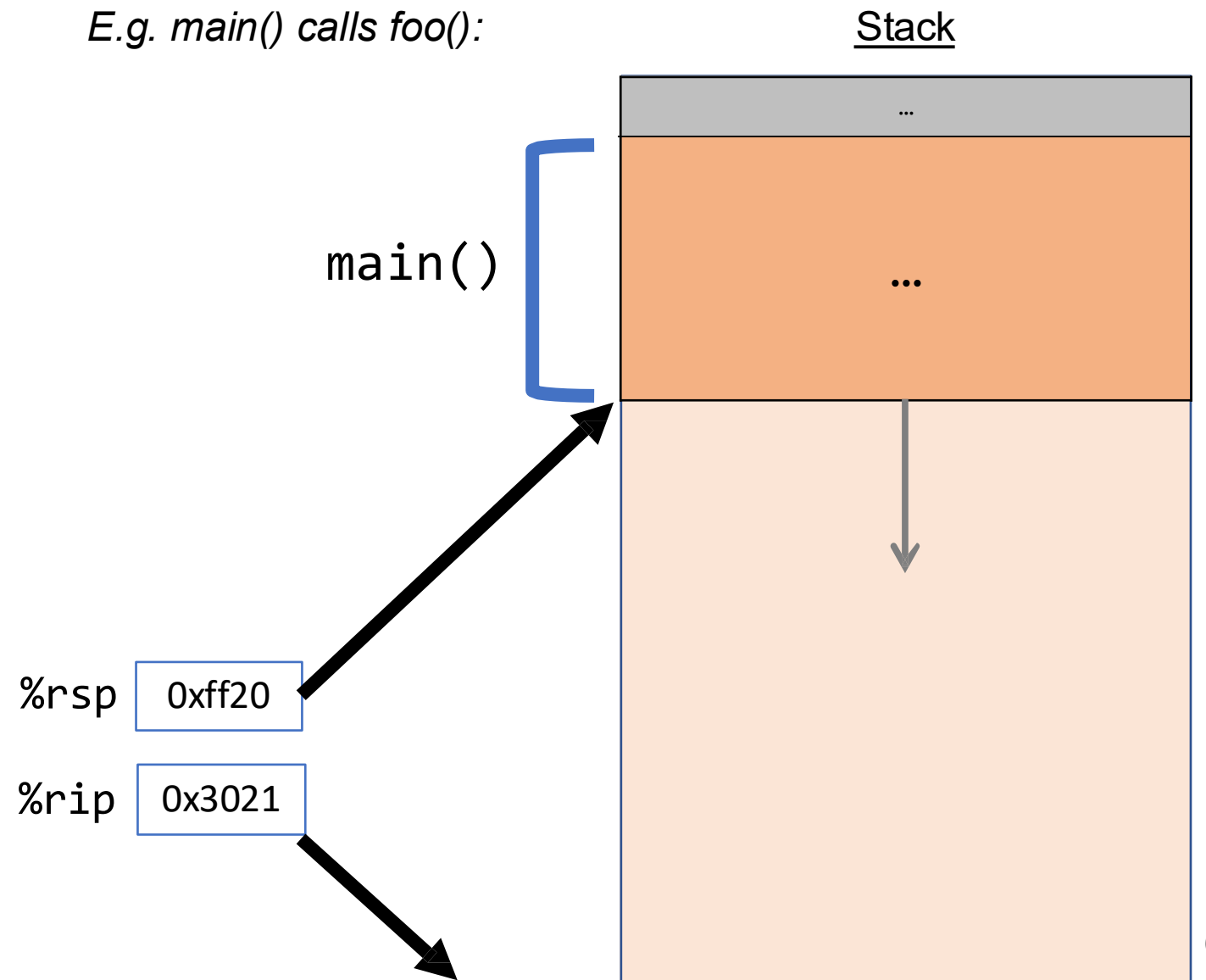
- **Pass Control** – %rip must be adjusted to execute the callee's instructions, and then resume the caller's instructions afterwards.
- **Pass Data** – we must pass any parameters and receive any return value.
- **Manage Memory** – we must handle any space needs of the callee on the stack.

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.

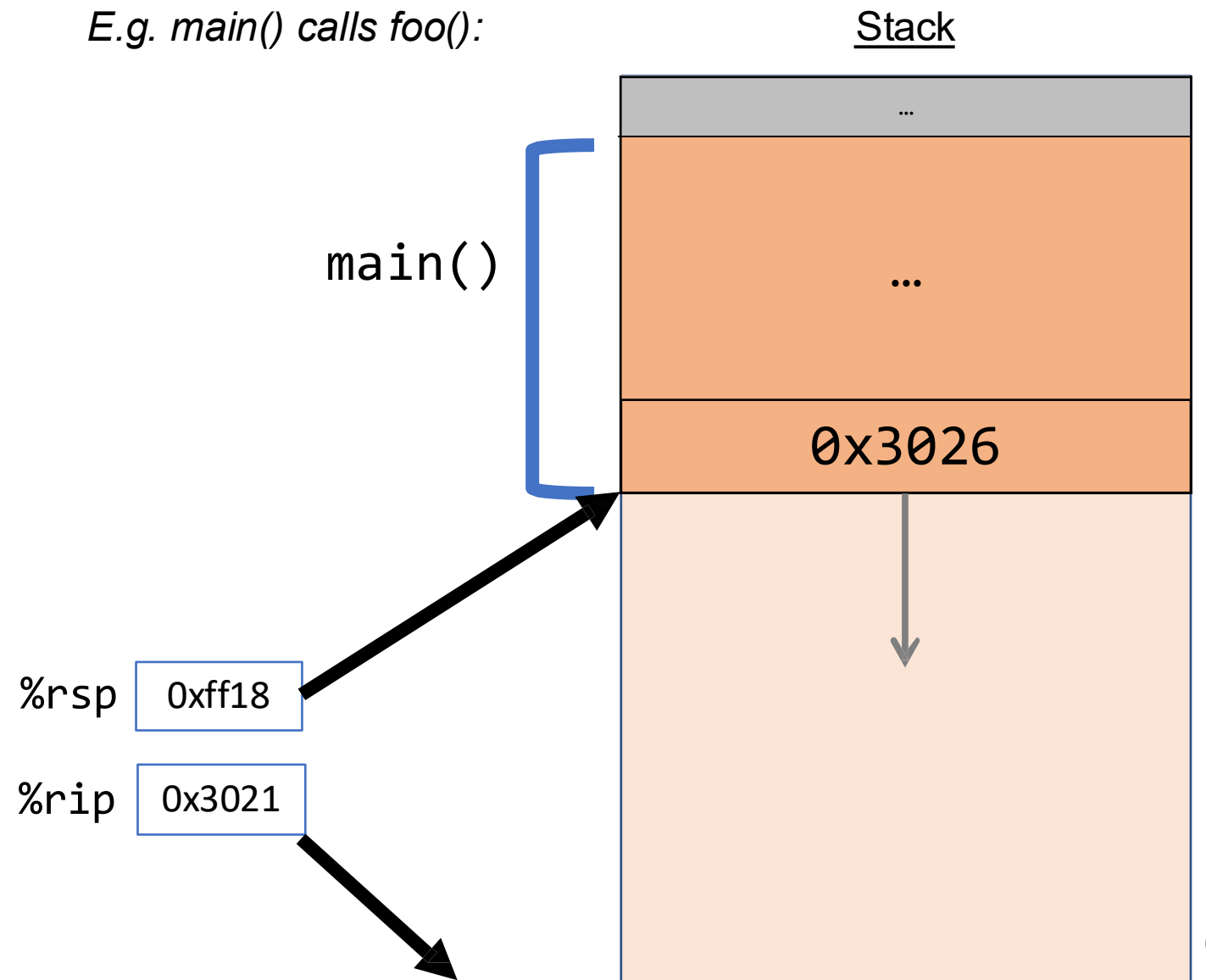
Solution: push the next value of %rip onto the stack. Then call the function. When it is finished, put this value back into %rip and continue executing.



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.

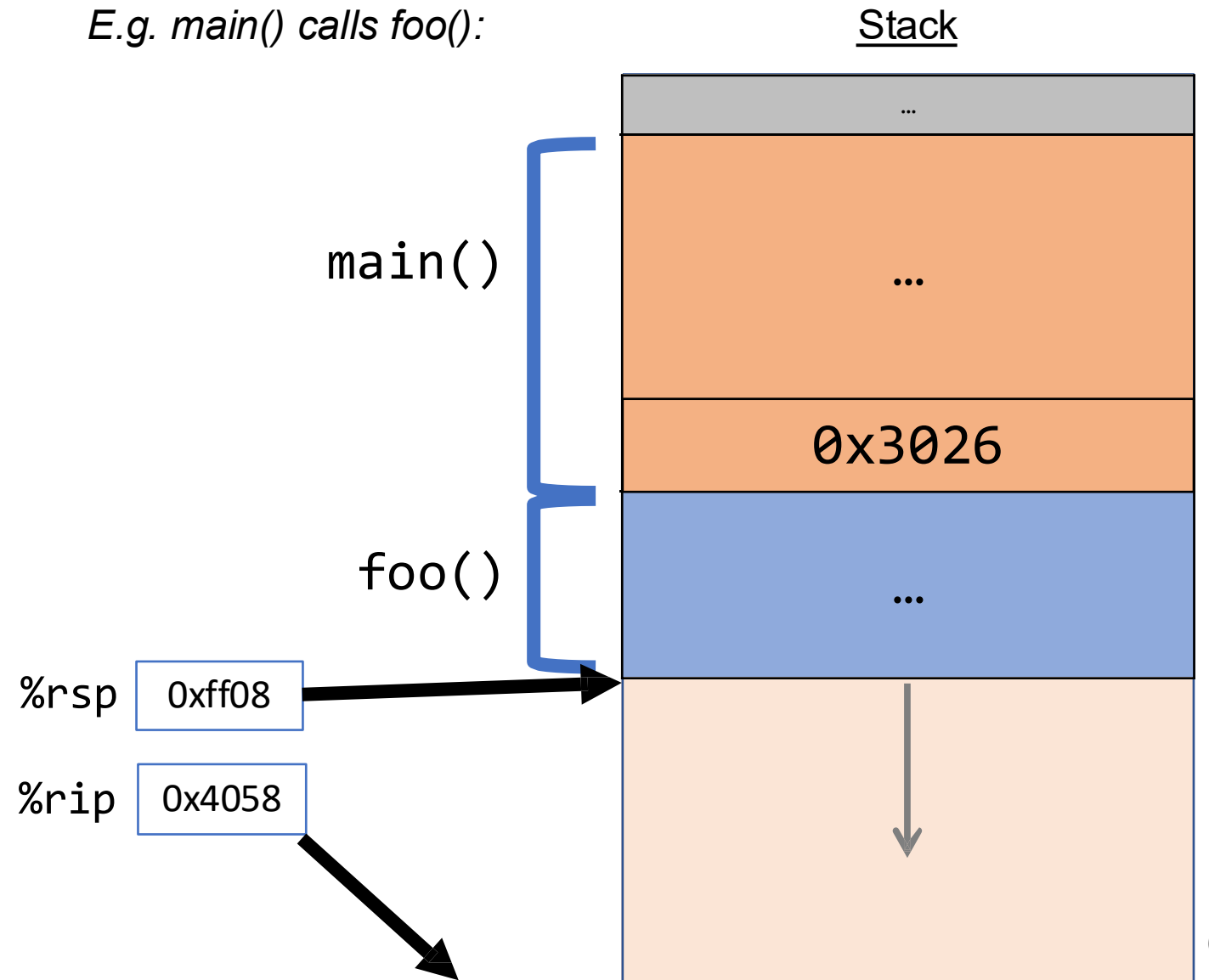
Solution: push the next value of `%rip` onto the stack. Then call the function. When it is finished, put this value back into `%rip` and continue executing.



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.

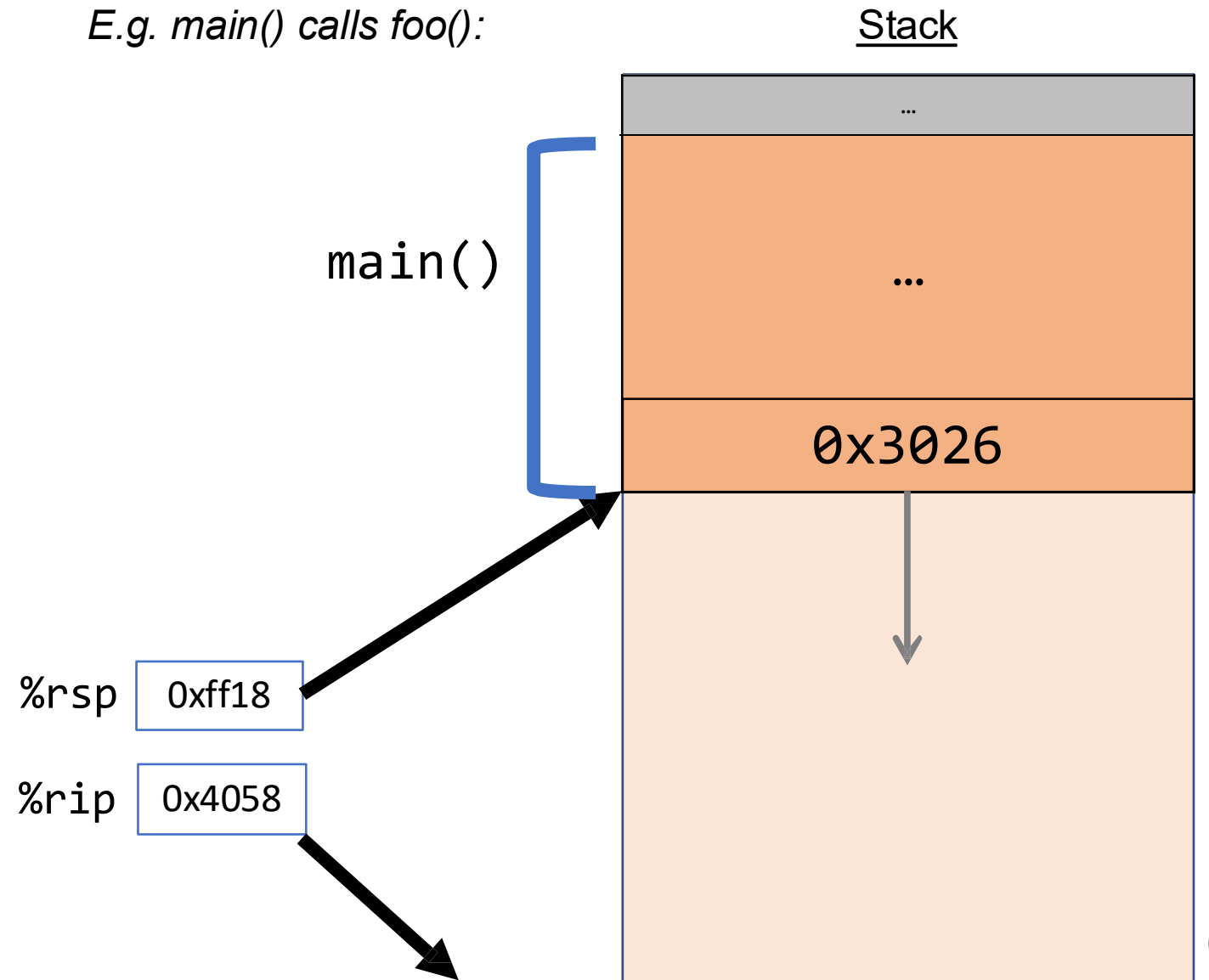
Solution: push the next value of `%rip` onto the stack. Then call the function. When it is finished, put this value back into `%rip` and continue executing.



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.

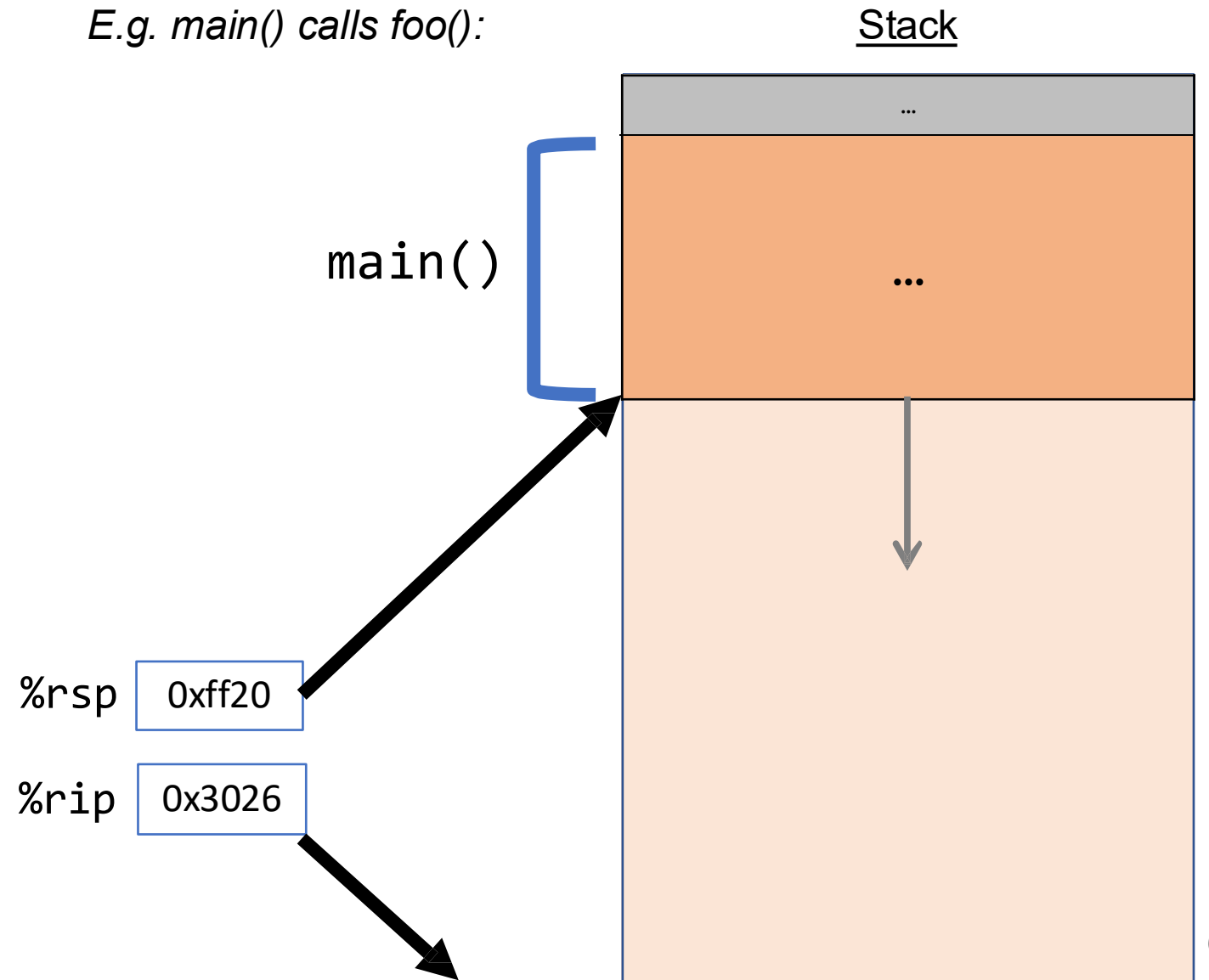
Solution: push the next value of %rip onto the stack. Then call the function. When it is finished, put this value back into %rip and continue executing.



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.

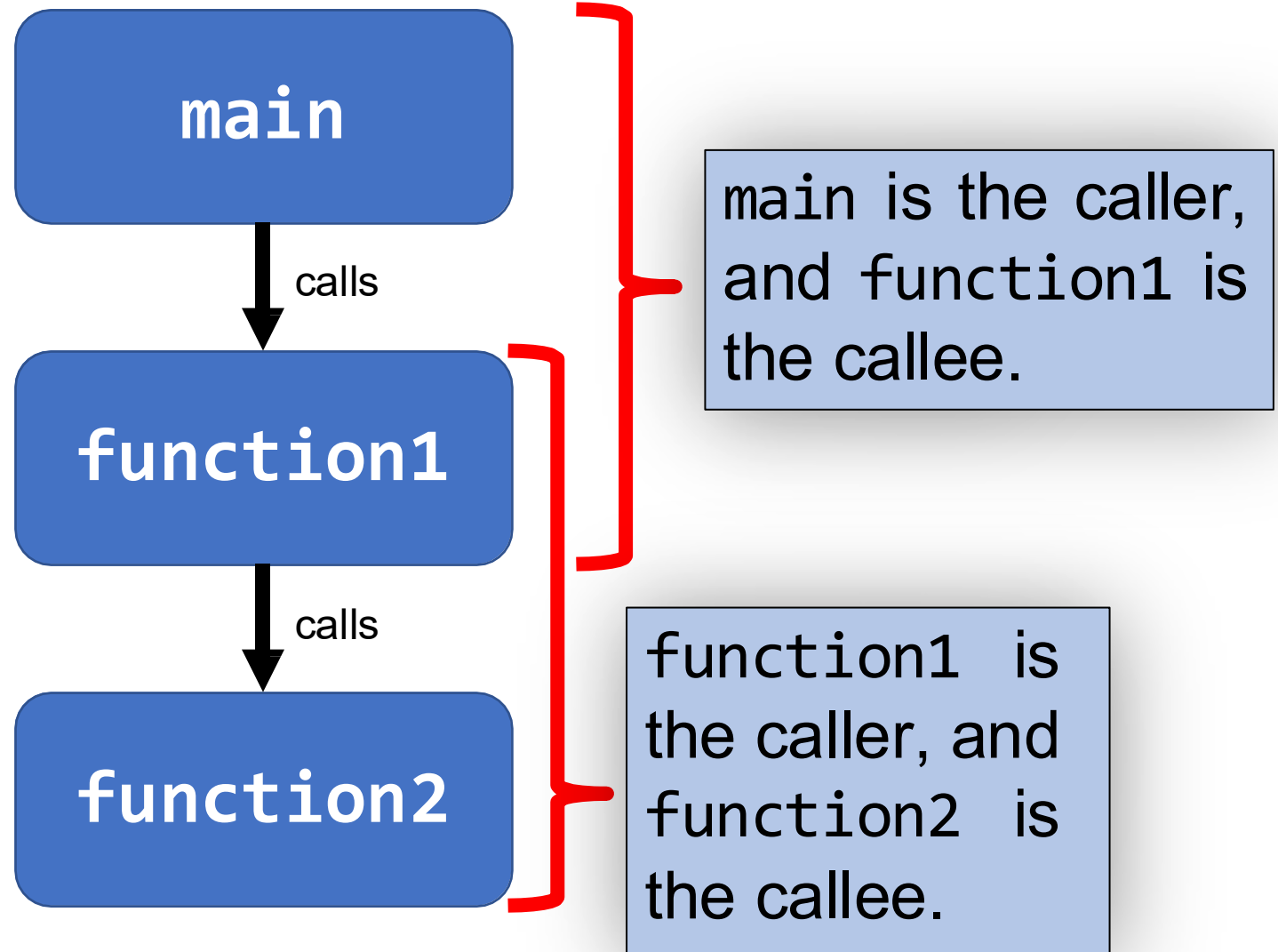
Register Restrictions

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

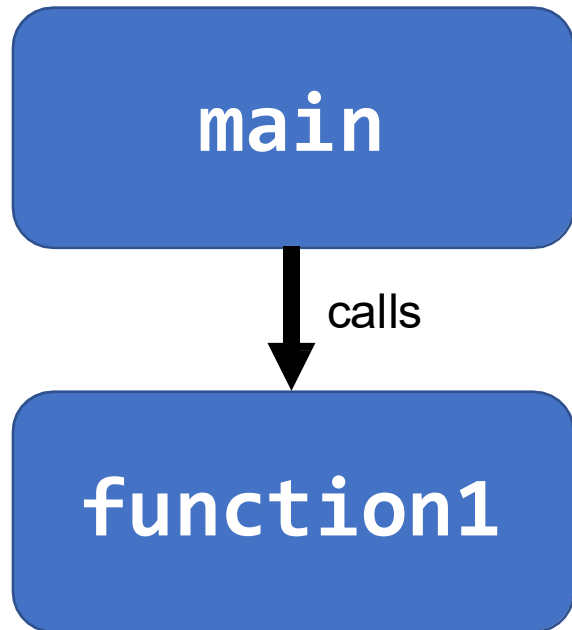
Caller-Owned

- 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.

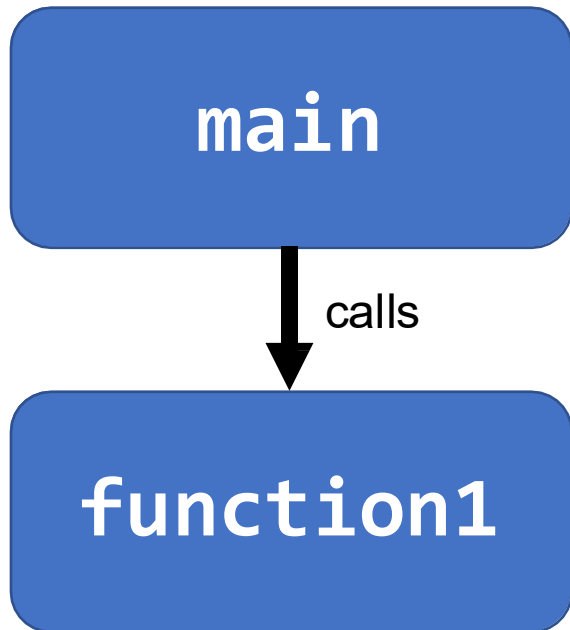
Caller-Owned Registers



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

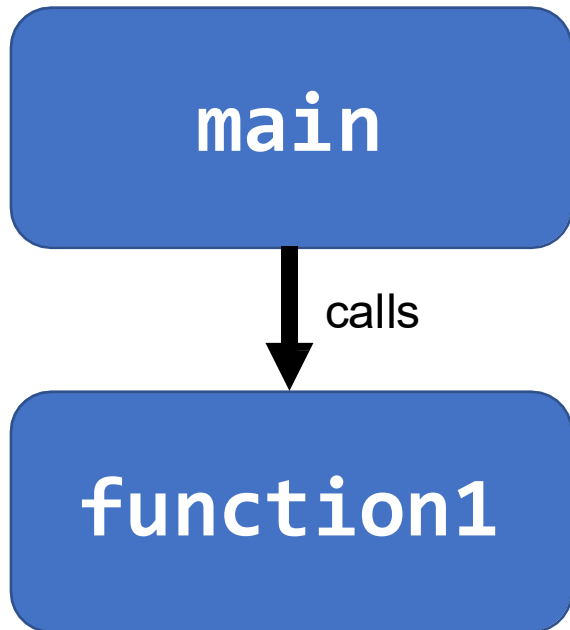
If function1 wants to use any caller-owned registers, it must save the existing values and restore them before returning.

Caller-Owned Registers



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

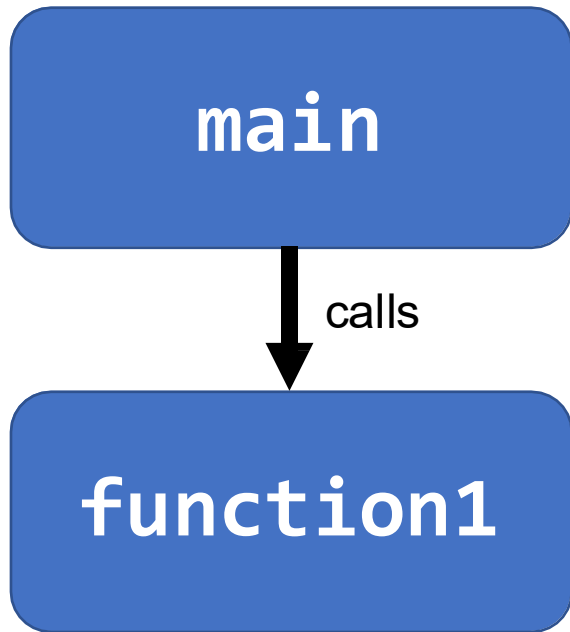
Callee-Owned Registers



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

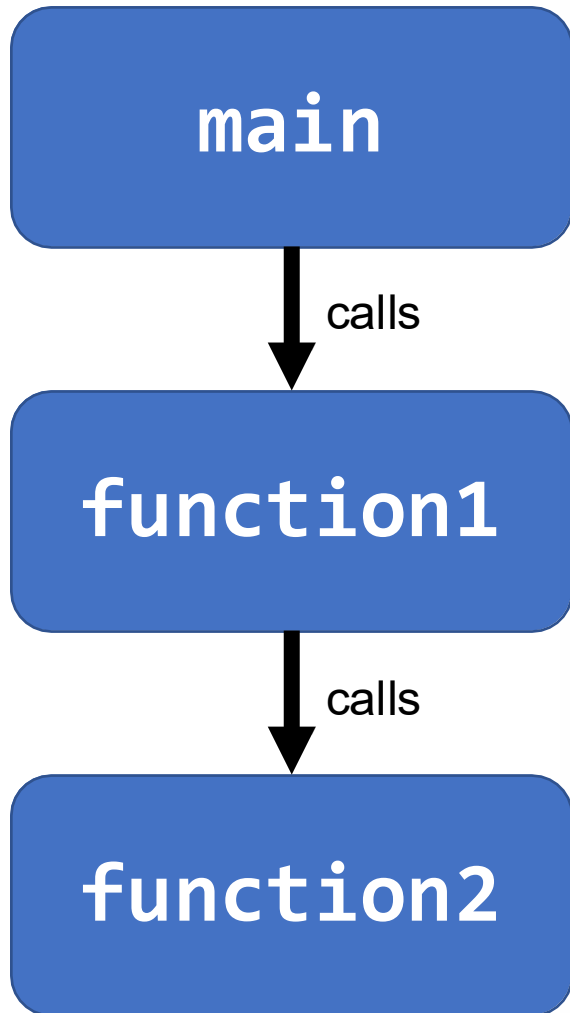
If function1 wants to use any callee-owned registers, it can do so without saving the existing values.

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:

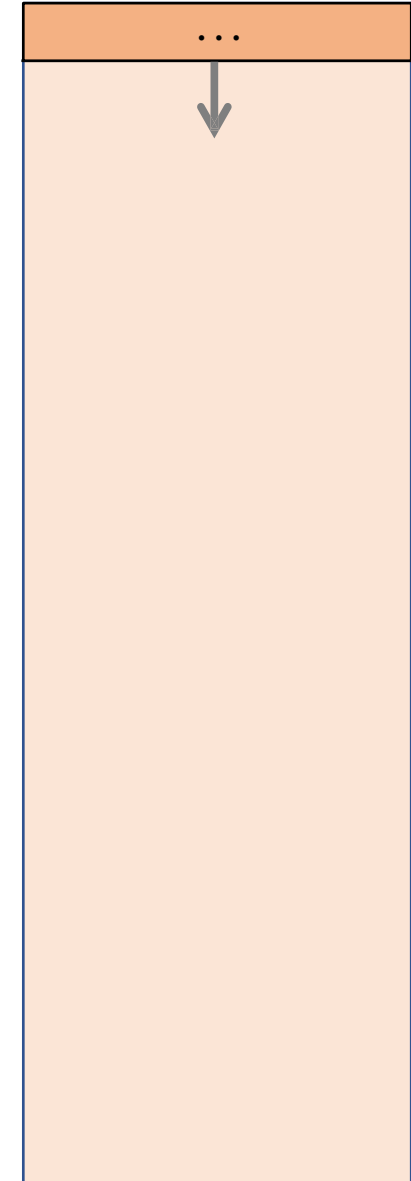
- **Pass Control** – %rip must be adjusted to execute the function being called and then resume the caller function afterwards.
- **Pass Data** – we must pass any parameters and receive any return value.
- **Manage Memory** – we must handle any space needs of the callee on the stack.

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

Stack Management

```
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() 

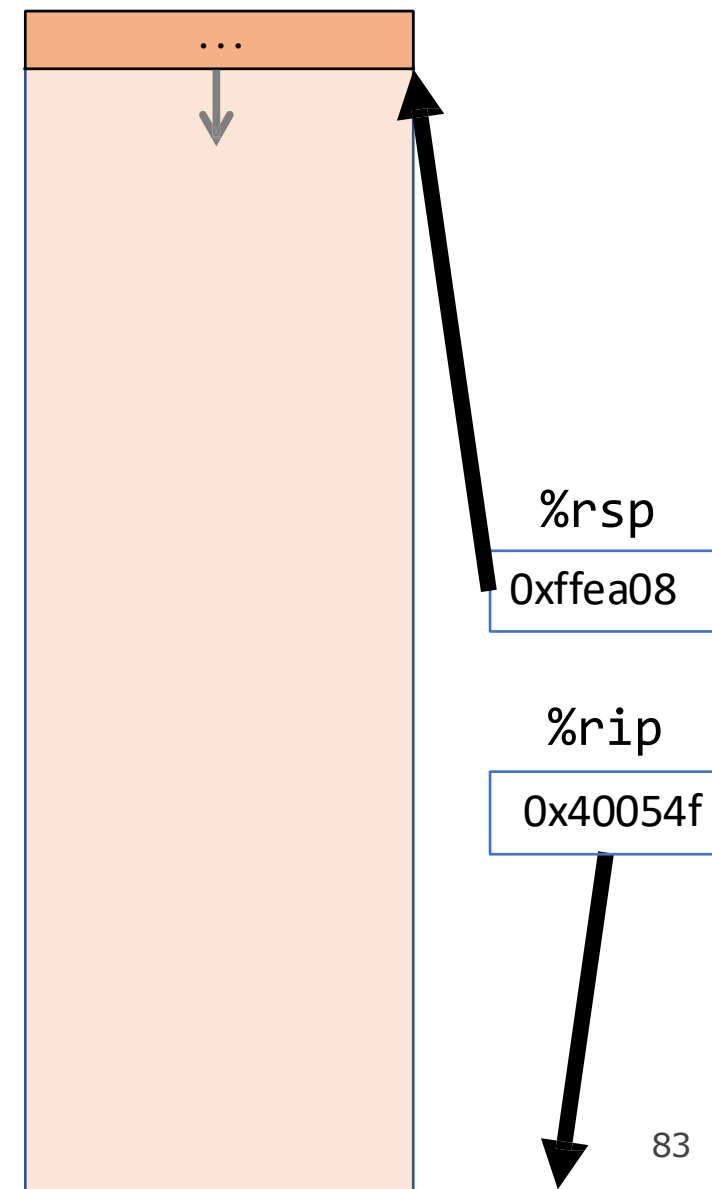


Stack Management

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

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

main() 



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

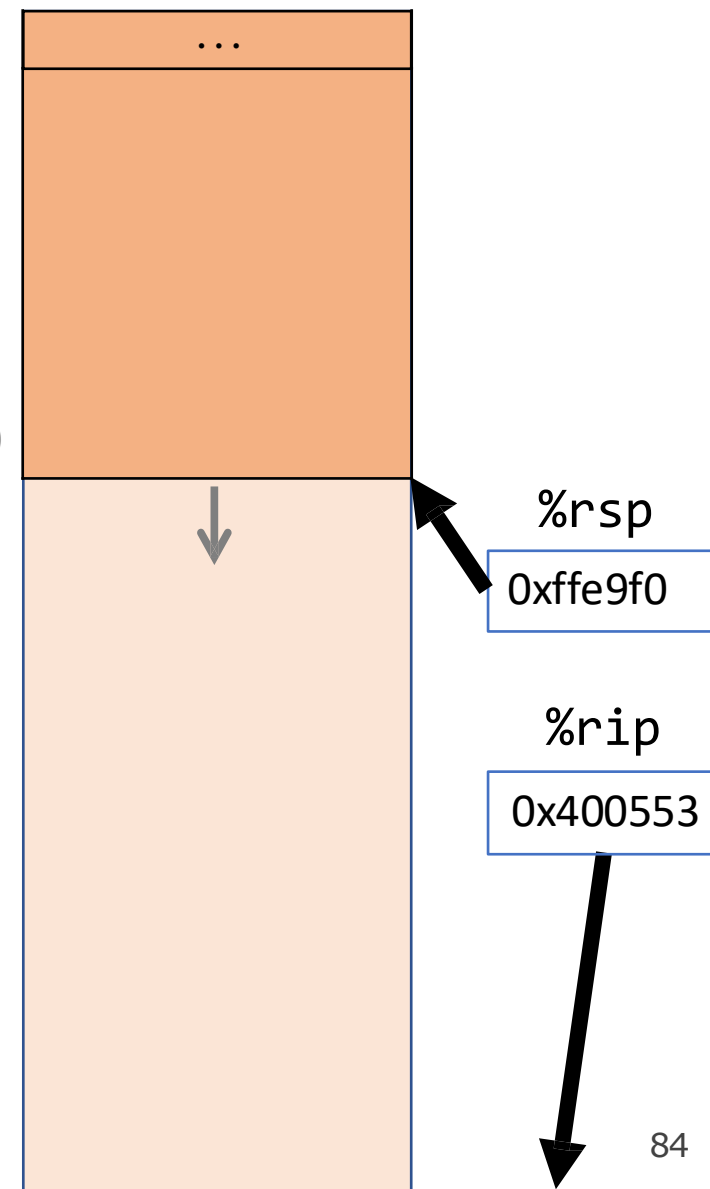
Stack Management

```
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()

0xffe9f0

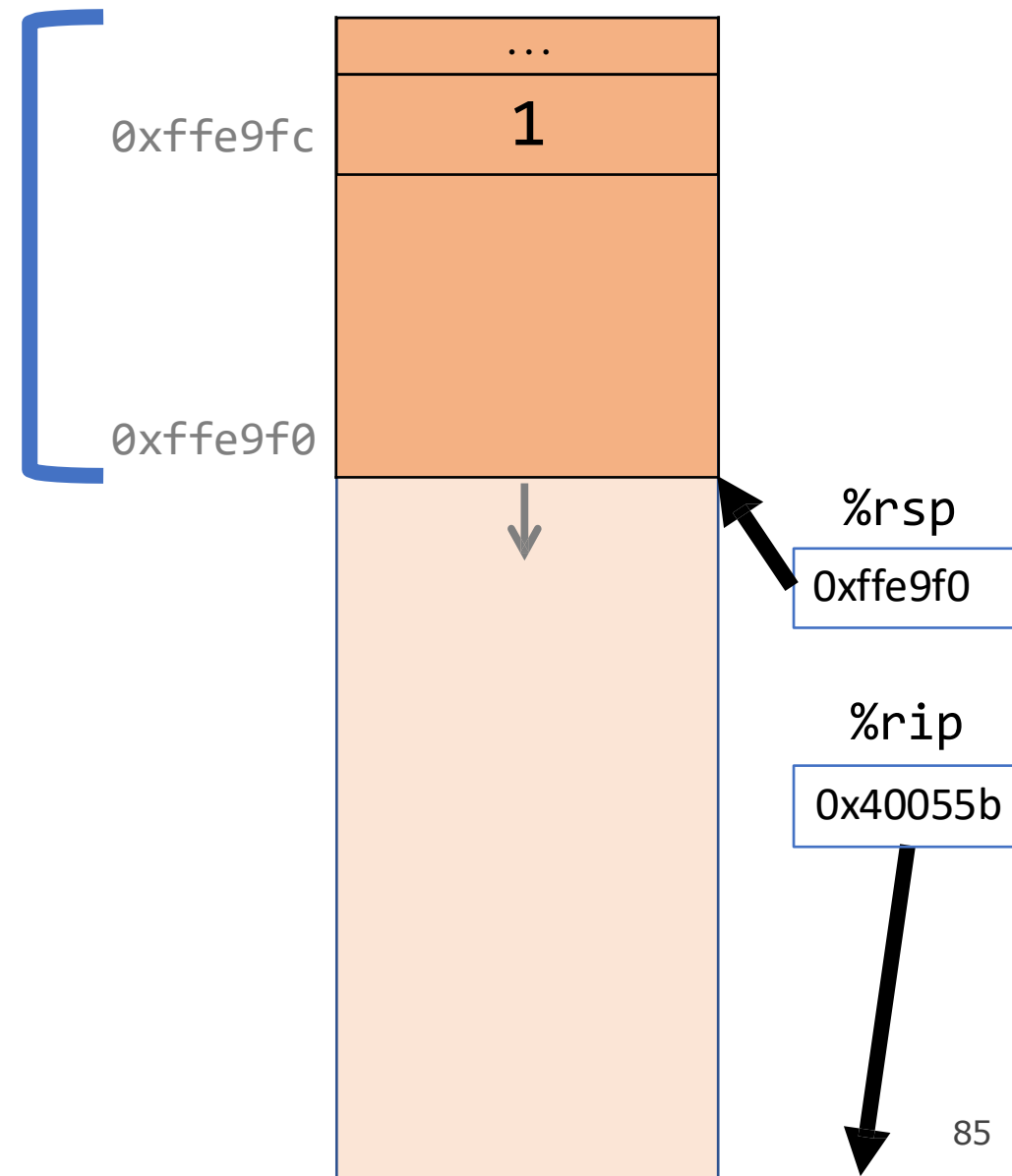


Stack Management

```
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()

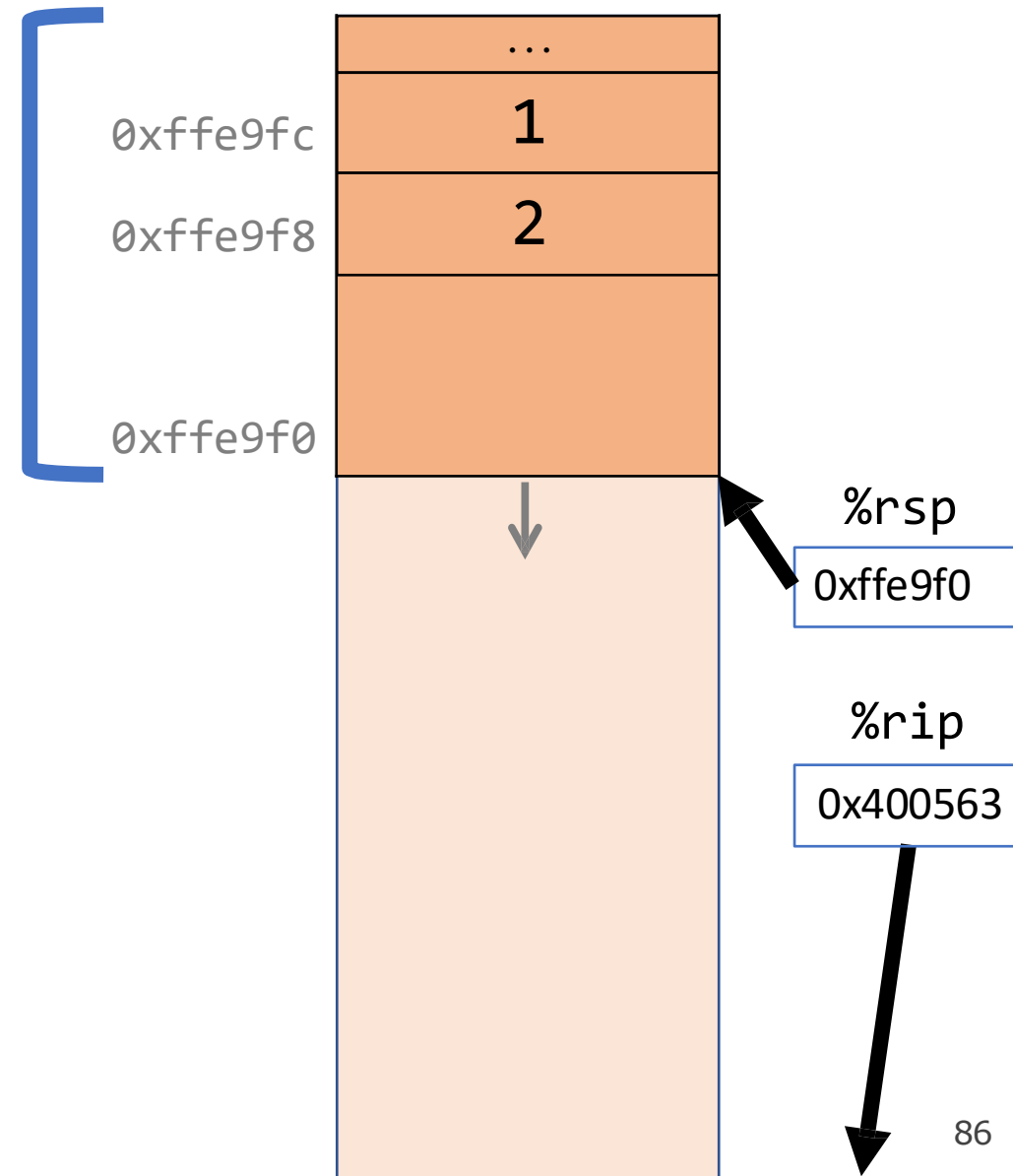


Stack Management

```
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()

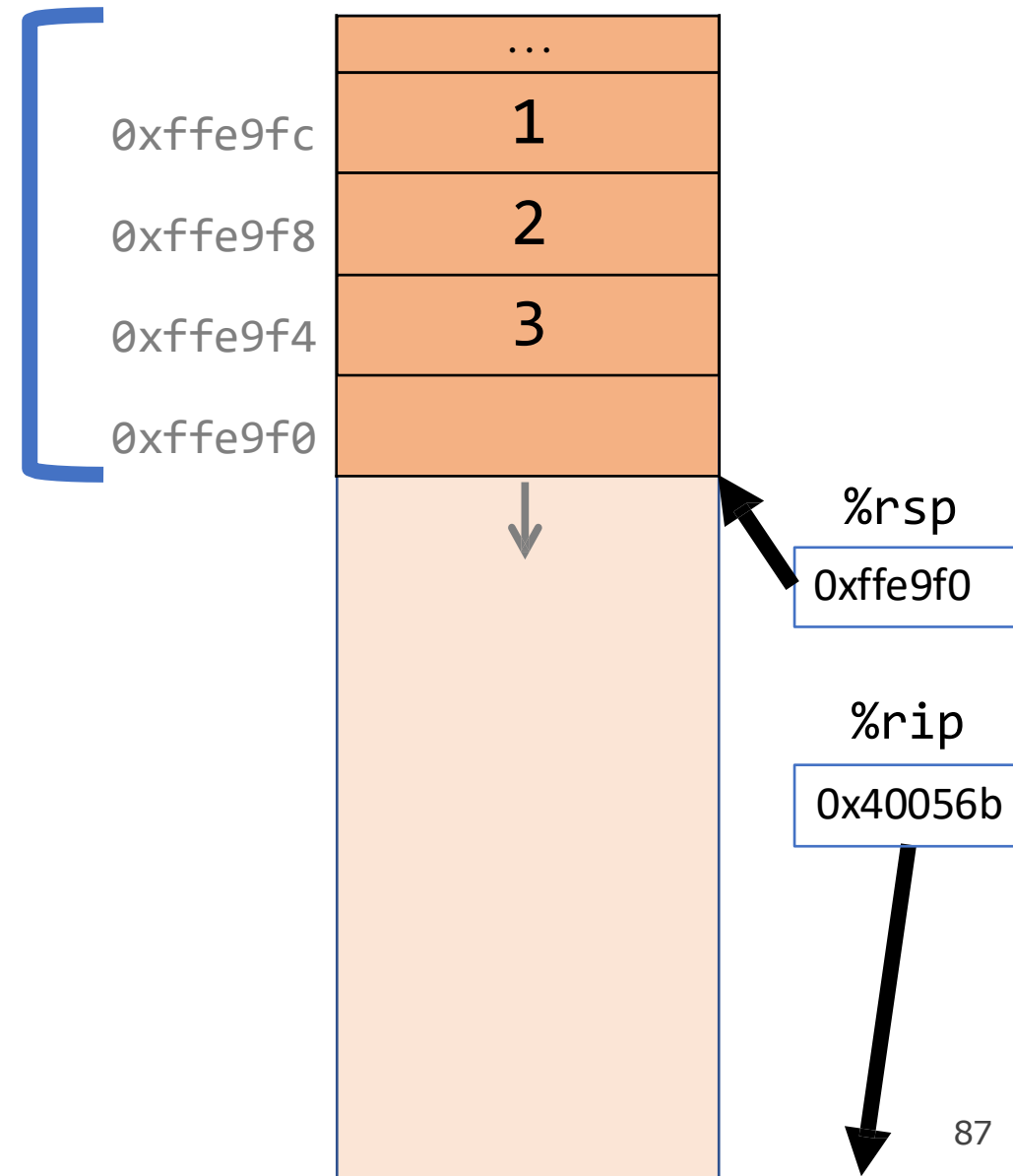


Stack Management

```
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()

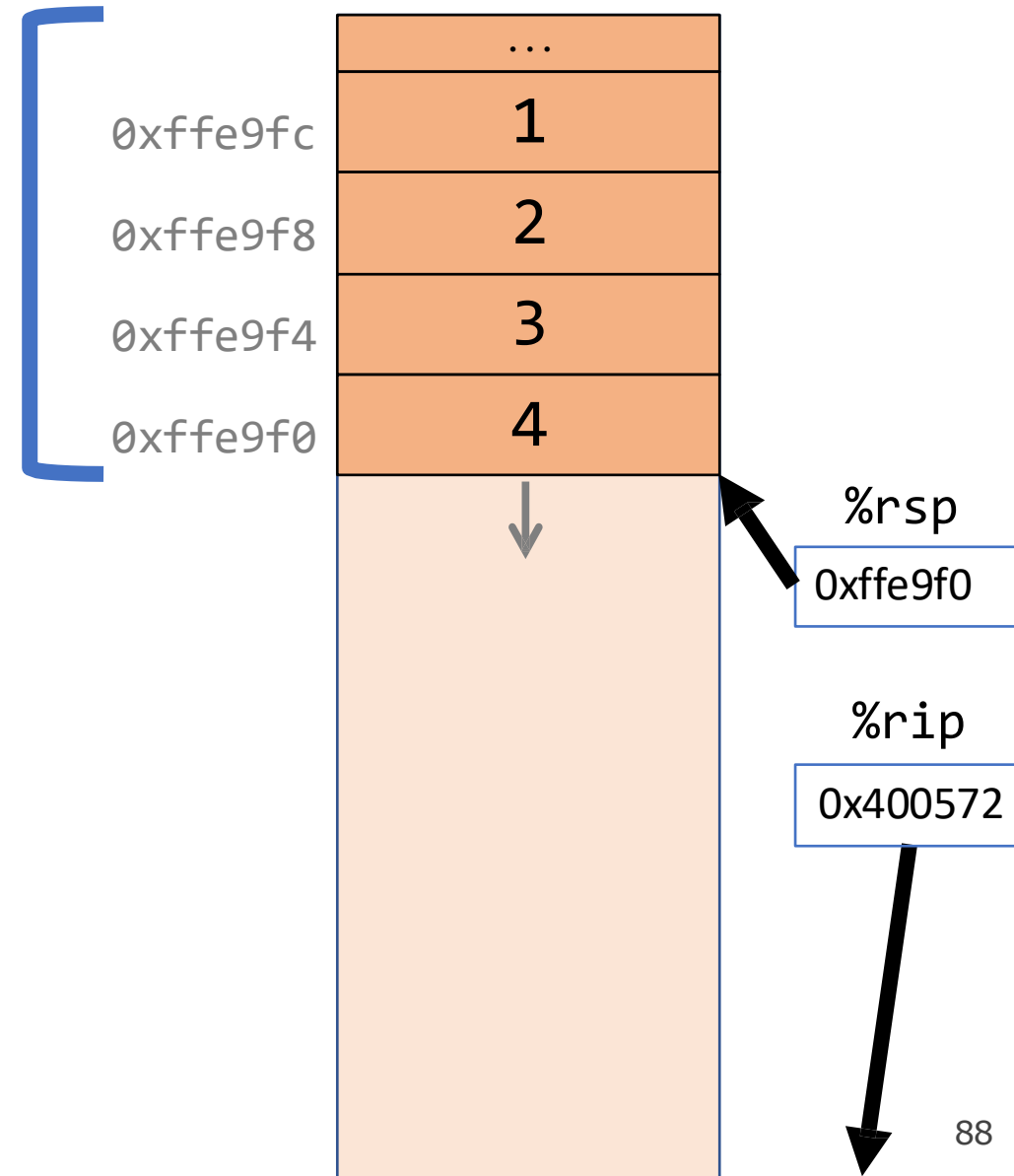


Stack Management

```
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()

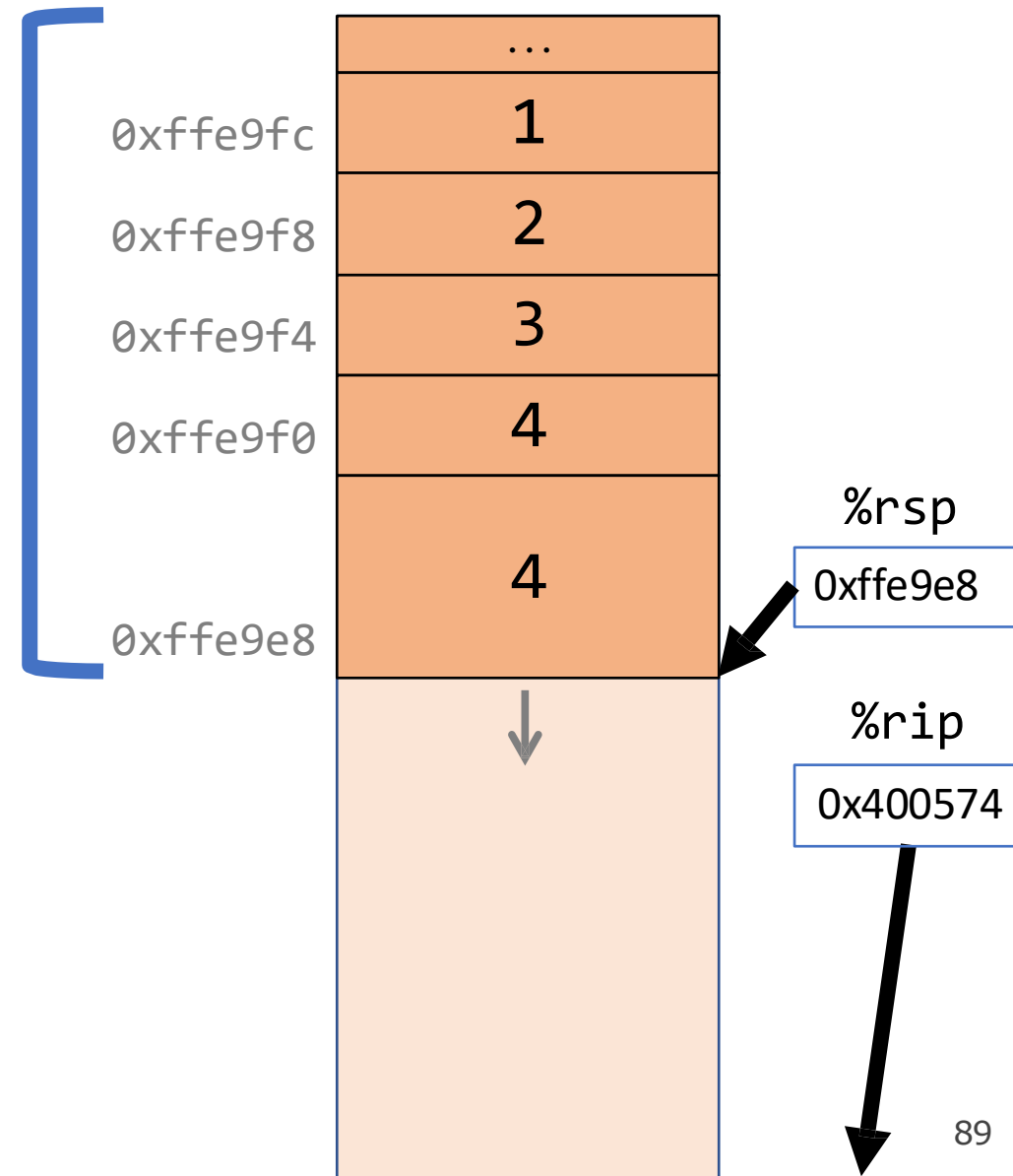


Stack Management

```
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
```

main()

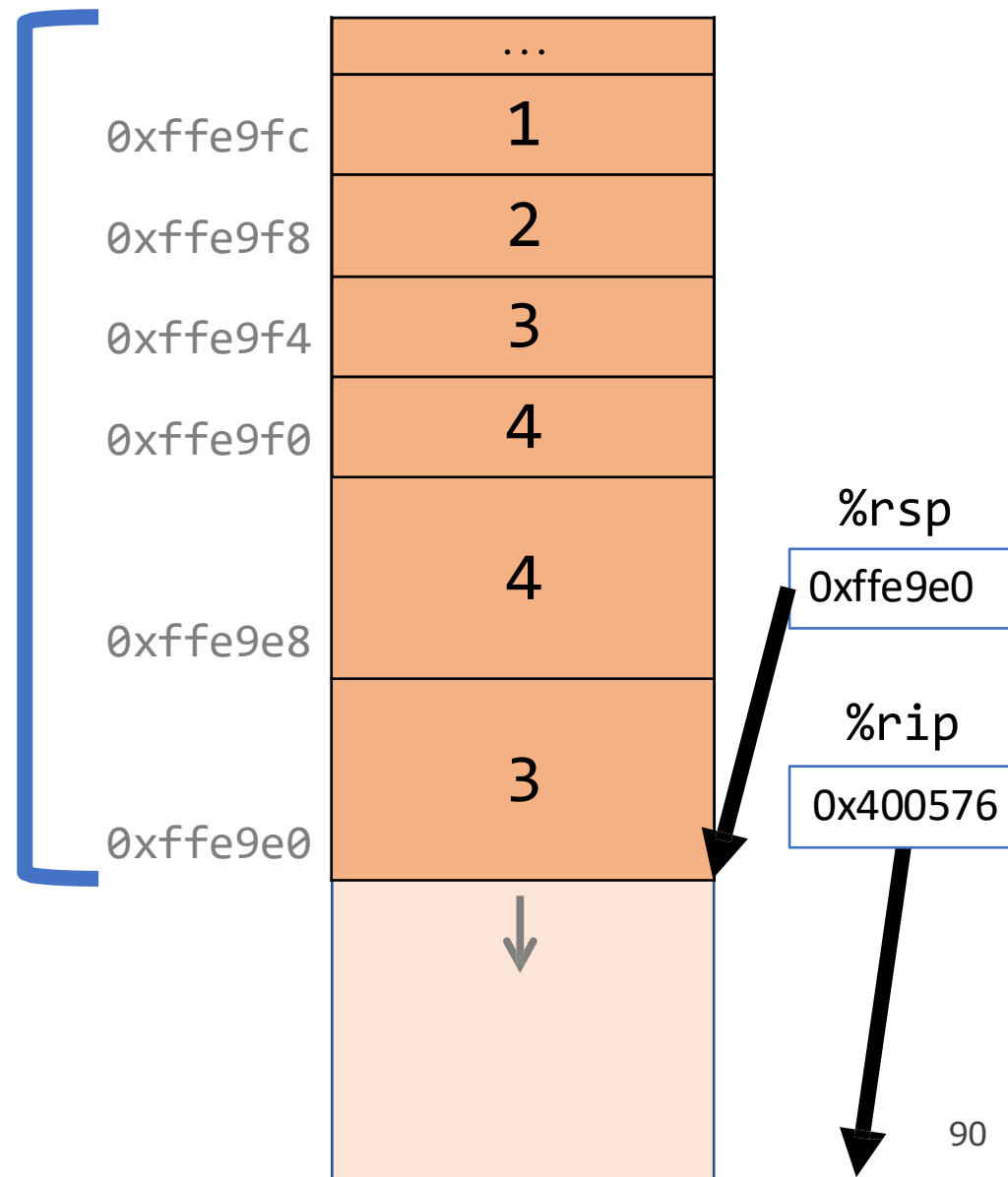


Stack Management

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

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

main()

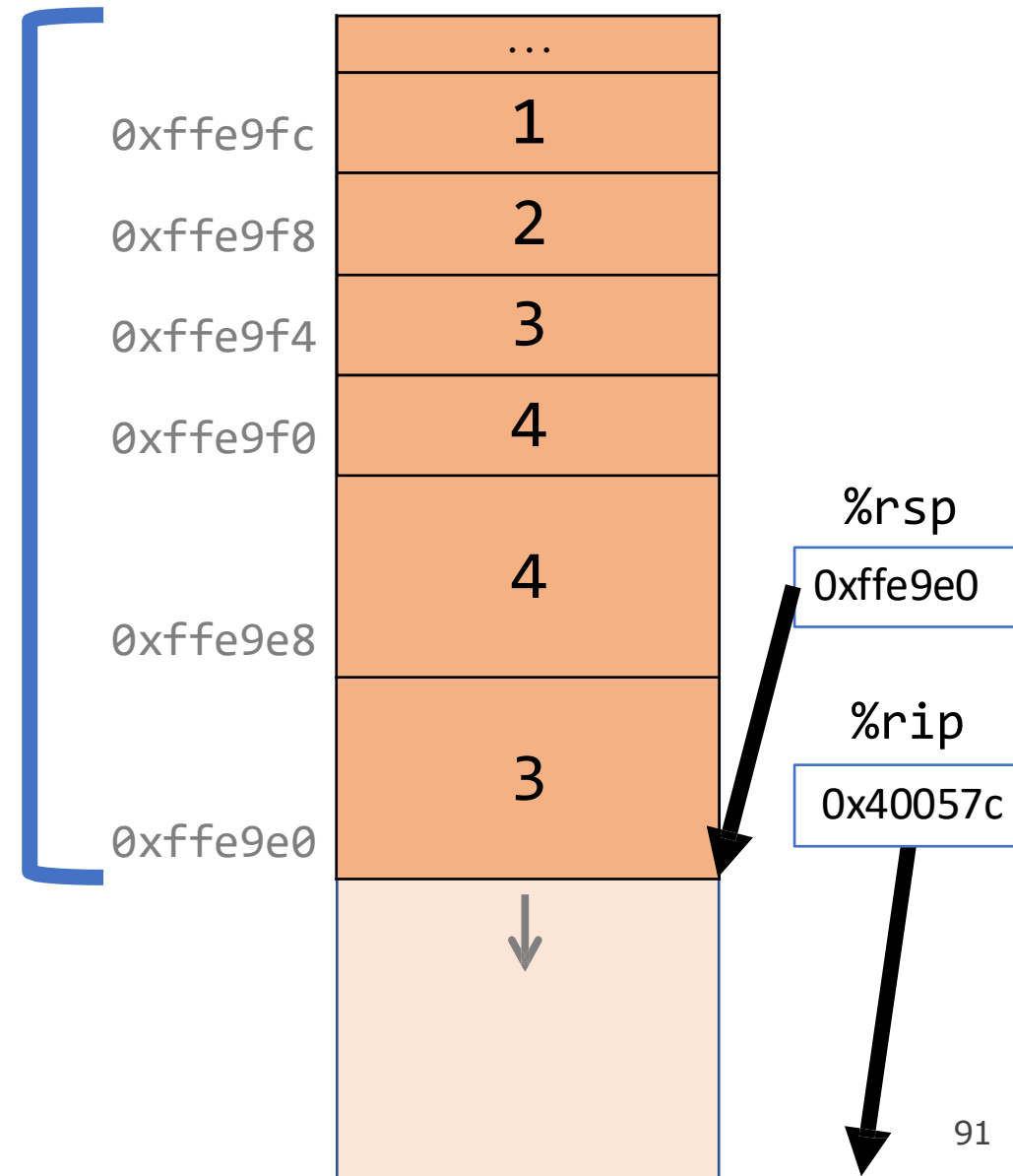


Stack Management

```
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()

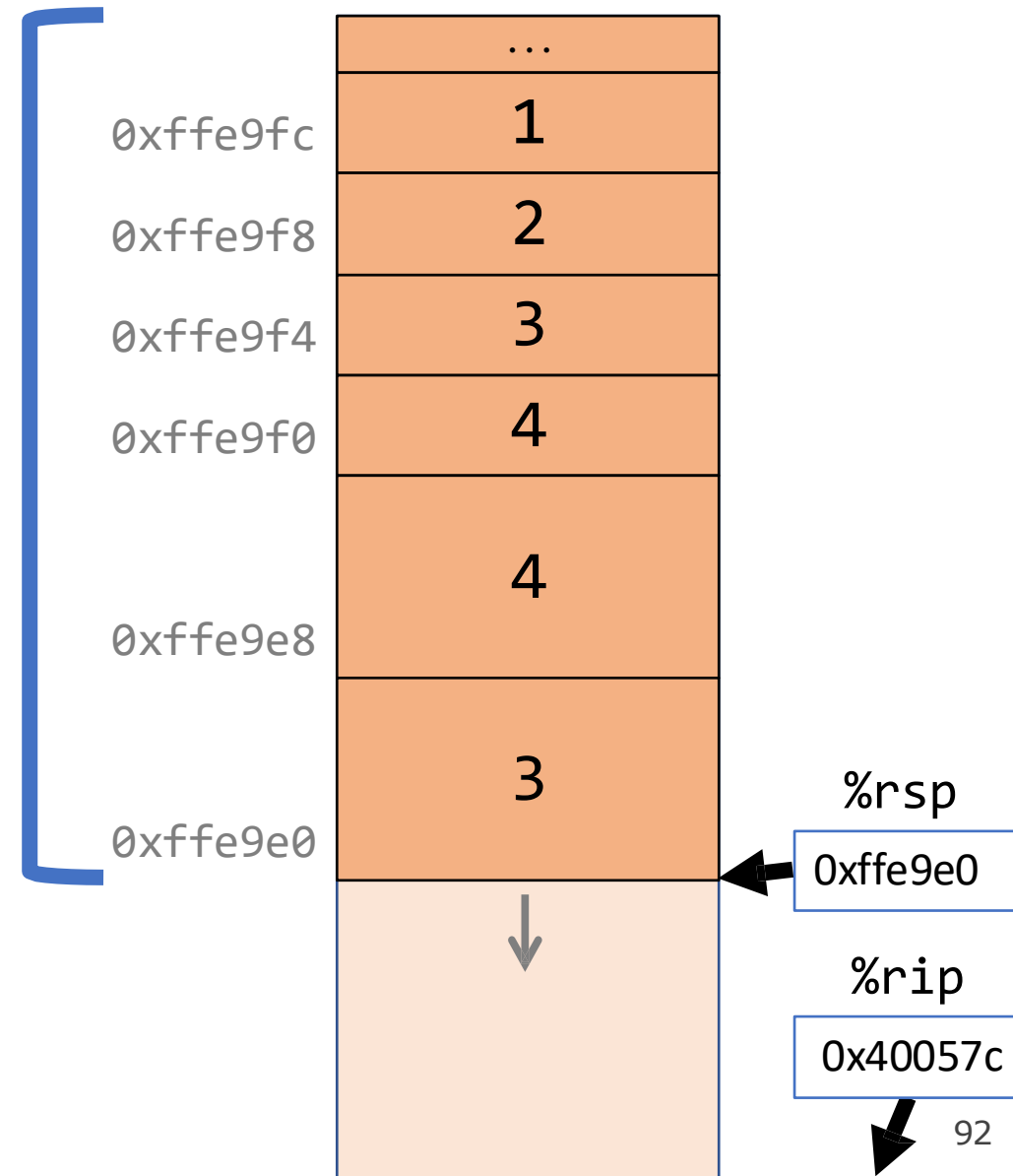


Stack Management

```
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()

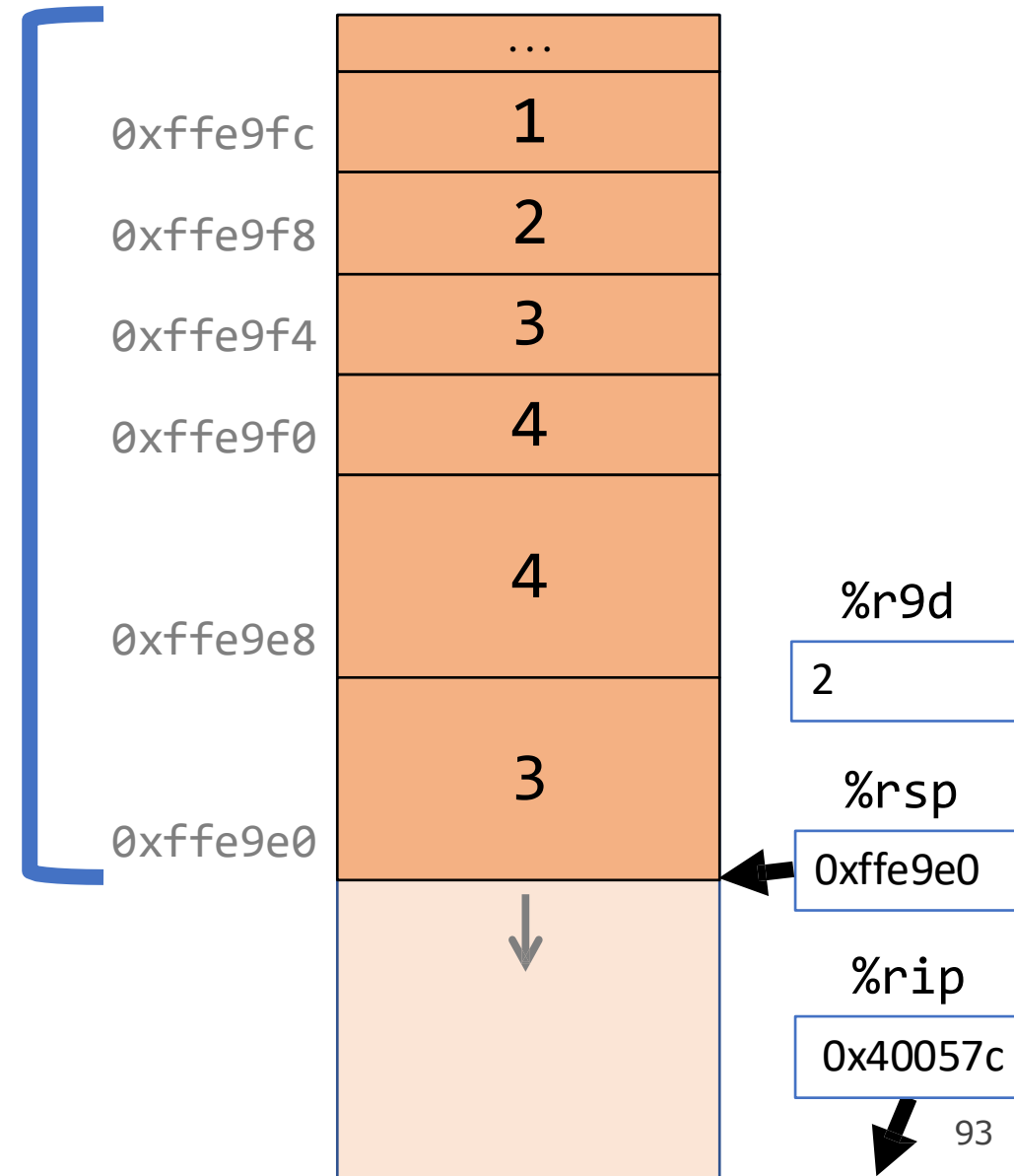


Stack Management

```
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()

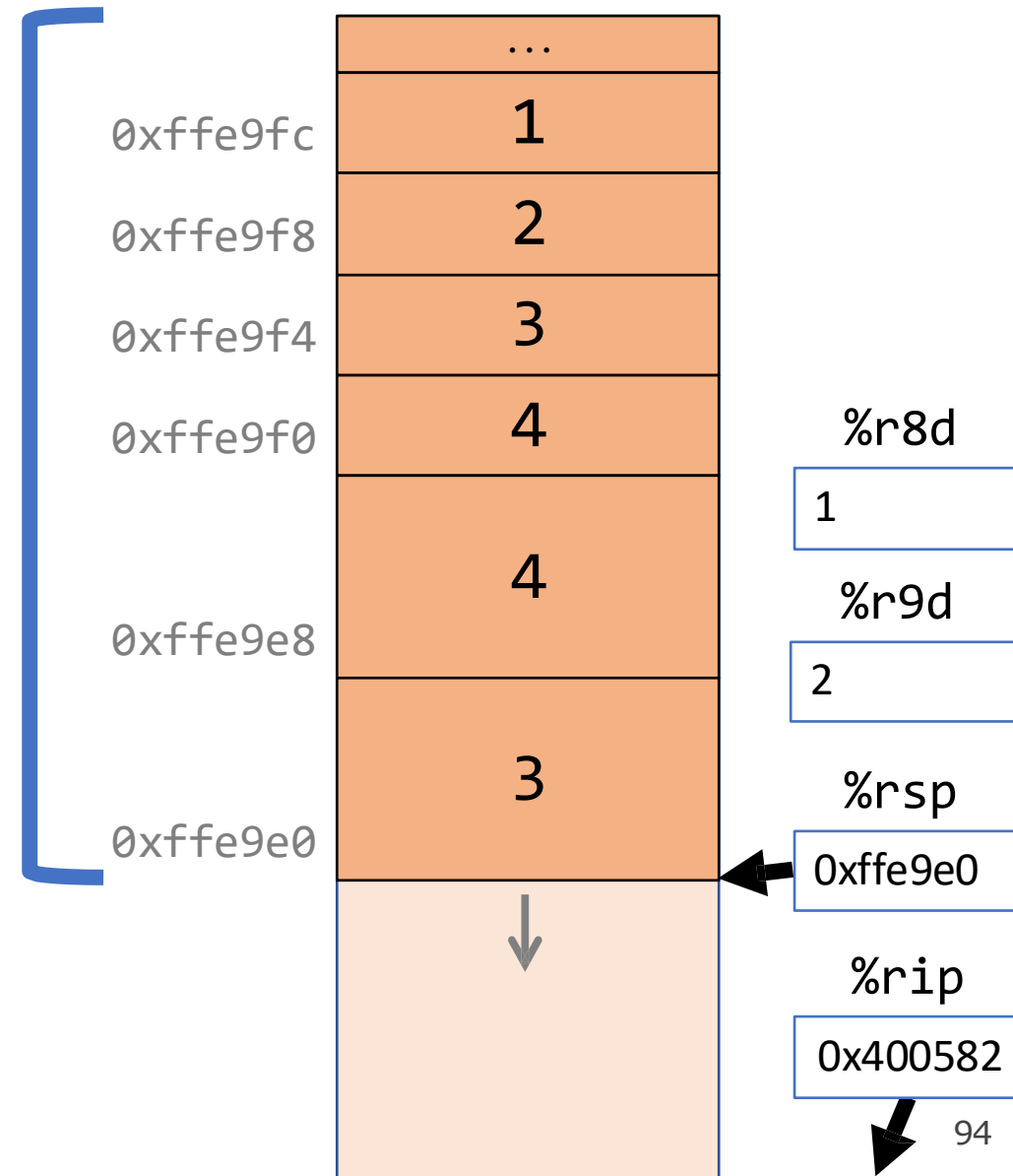


Stack Management

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

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

main()



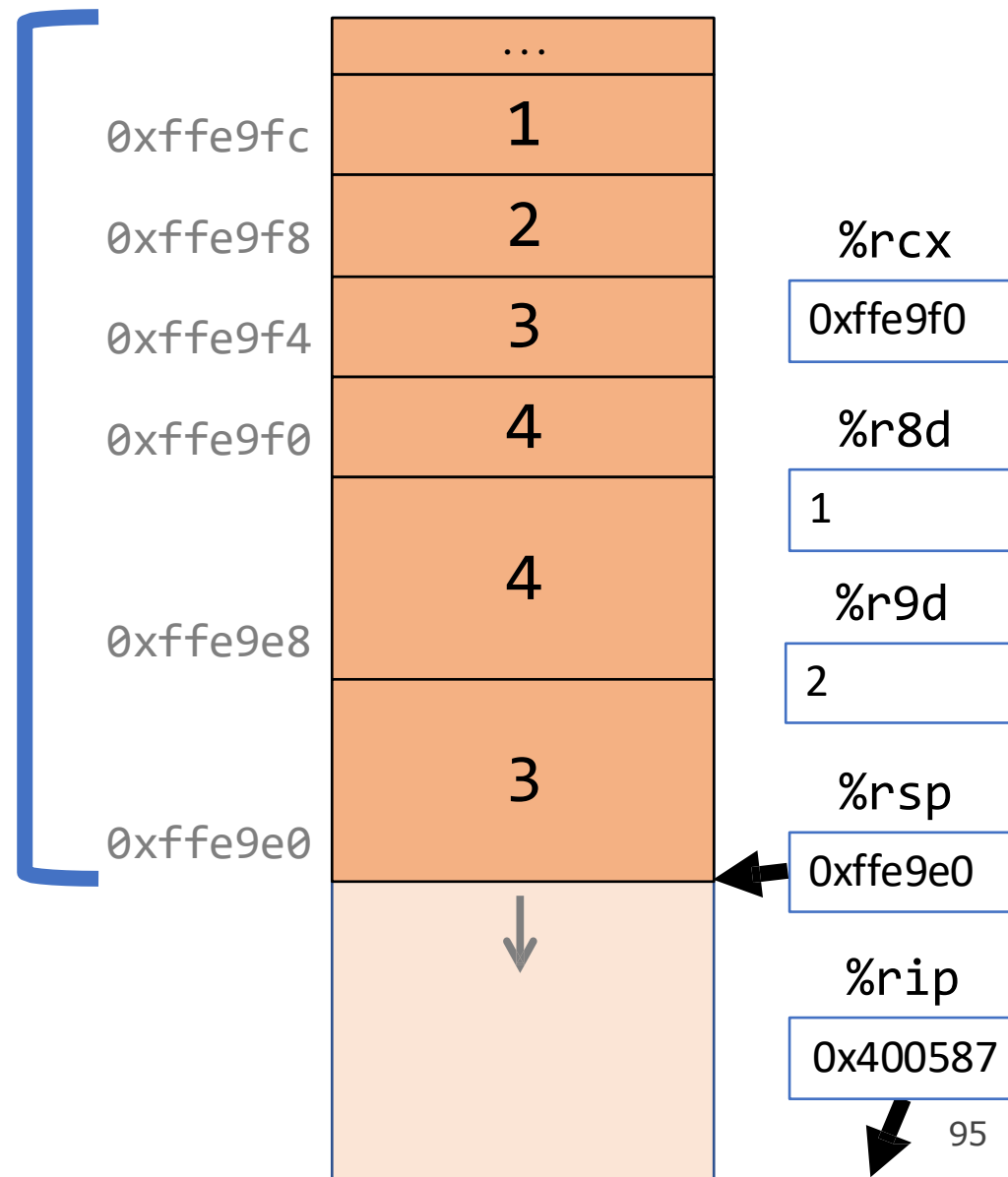
Stack Management

```
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()



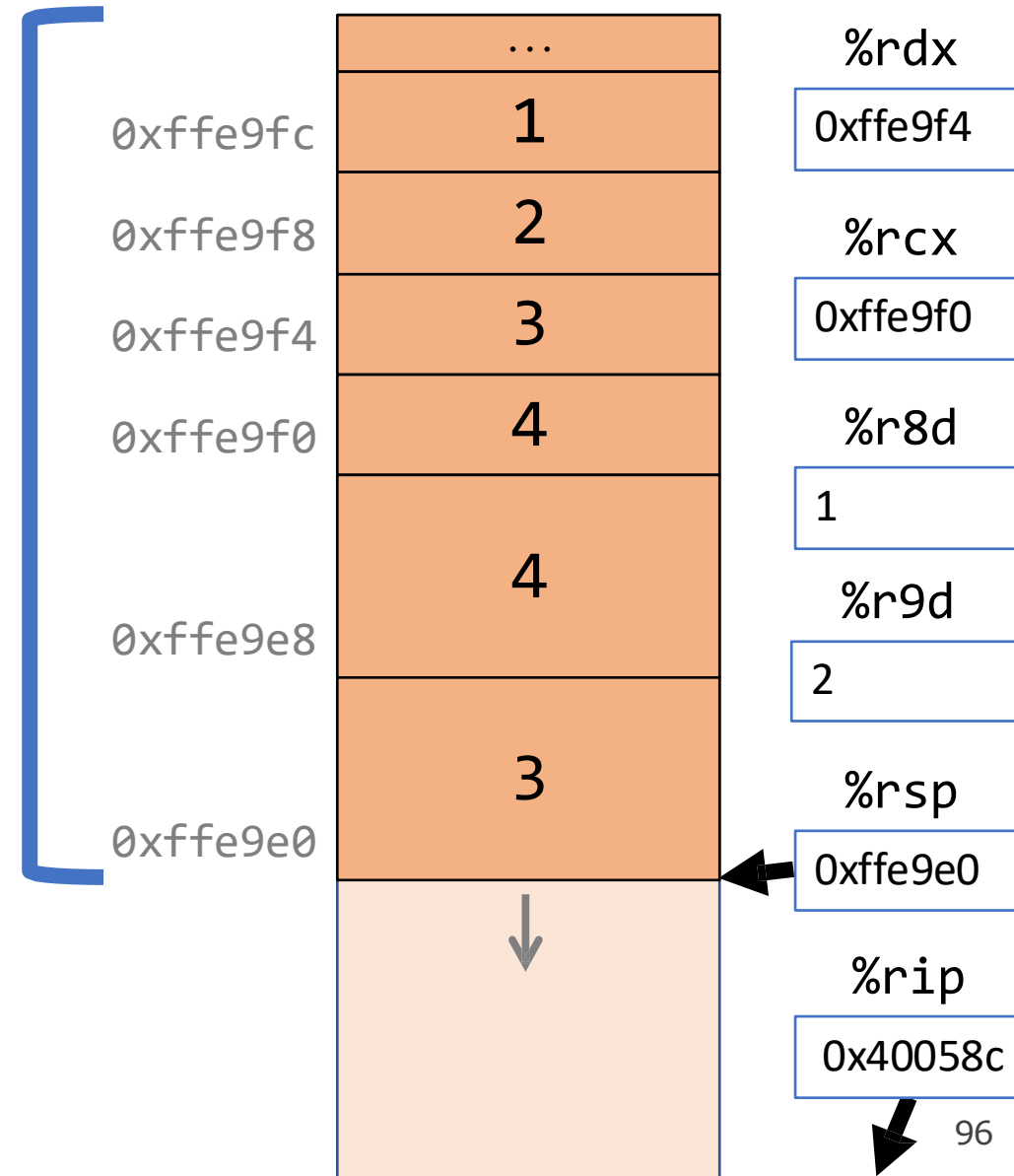
Stack Management

```
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()



Stack Management

```
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()

%rsi

0xffe9f8

0xffe9fc

0xffe9f8

0xffe9f4

0xffe9f0

0xffe9e8

0xffe9e0

...

1

2

3

4

4

3

↓

%rdx

0xffe9f4

%rcx

0xffe9f0

%r8d

1

%r9d

2

%rsp

0xffe9e0

%rip

0x400591

97

Stack Management

```
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()

%rsi
0xffe9f8

%rdi
0xffe9fc

0xffe9fc

0xffe9f8

0xffe9f4

0xffe9f0

0xffe9e8

0xffe9e0

...

1

2

3

4

4

3



%rdx

0xffe9f4

%rcx

0xffe9f0

%r8d

1

%r9d

2

%rsp

0xffe9e0

%rip

0x400596

98

Stack Management

```
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()

%rsi
0xffe9f8

%rdi
0xffe9fc

0xffe9fc

0xffe9f8

0xffe9f4

0xffe9f0

0xffe9e8

0xffe9e0

...

1

2

3

4

4

3



%rdx

0xffe9f4

%rcx

0xffe9f0

%r8d

1

%r9d

2

%rsp

0xffe9e0

%rip

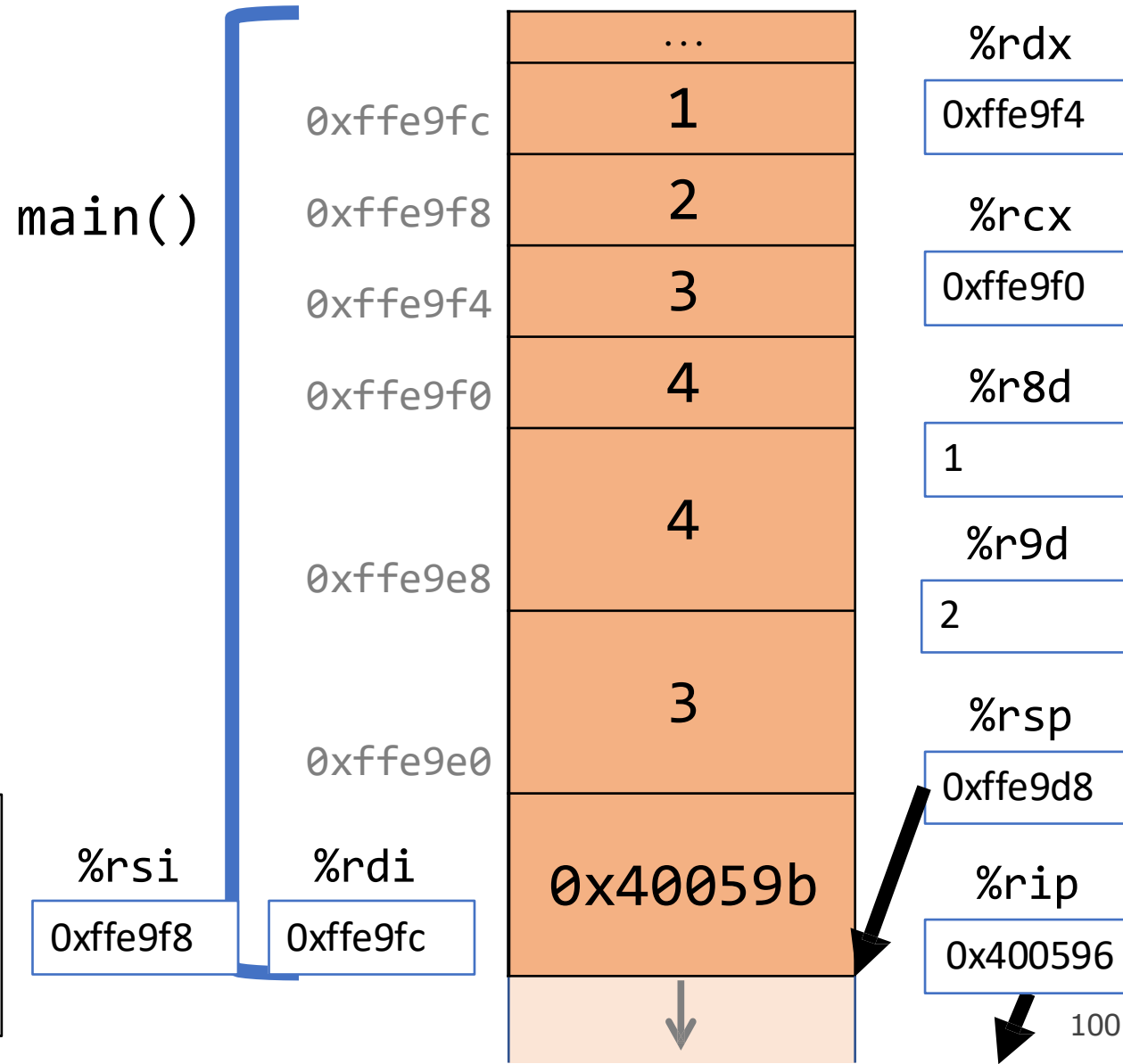
0x400596

Stack Management

```
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

31 c0

mov \$0x0,%eax

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.

- 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	<code>char</code>
2	<code>short</code>
4	<code>int</code> , <code>float</code>
8	<code>long</code> , <code>double</code> , <code>char *</code>

- 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`:

Offset	0	4	5	9
Contents	i		c	j

Offset	0	4	5	8	12
Contents	i		c		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
```

```
0x7fffffffefe870: 0x000000000000000005          0x000000000000400559
```

```
0x7fffffffefe880: 0x000000000000000000          0x000000000000400575
```

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

```
(gdb) display/4w $rsp
```

```
1: x/4xw $rsp
```

```
0x7fffffffefe8a8:
```

```
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)

`info reg`

`p $eax`

`p $eflags`

`b *0x400546`

`b *0x400550 if $eax > 98`

`ni`

`si`

View C, assembly, and gdb (lab5)

Print all registers

Print register value

Print all condition codes currently set

Set breakpoint at assembly instruction

Set **conditional breakpoint**

Next assembly instruction

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

- 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 :

$T \text{ } A[N]$

- The starting location is designated as x_A
- The declaration allocates $N * \text{sizeof}(T)$ 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 .

- Example:

		Array	Element Size	Total Size	Start address	Element i
char	A[12];	A	1	12	x_A	$x_A + i$
char	*B[8];	B	8	64	x_B	$x_B + 8i$
int	C[6];	C	4	24	x_C	$x_C + 4i$
double	*D[5]	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
```

- 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	<code>movq %rdx, %rax</code>
E[0]	int	$M[x_E]$	<code>movl (%rdx), %eax</code>
E[i]	int	$M[x_E+4i]$	<code>movl (%rdx,%rcx,4) %eax</code>
&E[2]	int *	x_E+8	<code>leaq 8(%rdx), %rax</code>
E+i-1	int *	x_E+4i-4	<code>leaq -4(%rdx,%rcx,4), %rax</code>
*(E+i-3)	int	$M[x_E+4i-12]$	<code>movl -12(%rdx,%rcx,4) %eax</code>
&E[i]-E	long	i	<code>movq %rcx,%rax</code>

- Practice: `xs` 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
<code>S+1</code>	<code>short *</code>	<code>xs + 2</code>	<code>leaq 2(%rdx), %rax</code>
<code>S[3]</code>	<code>short</code>	<code>M[xs + 6]</code>	<code>movw 6(%rdx), %ax</code>
<code>&S[i]</code>	<code>short *</code>	<code>xs + 2i</code>	<code>leaq (%rdx,%rcx,2), %rax</code>
<code>S[4*i+1]</code>	<code>short</code>	<code>M[xs + 8i + 2]</code>	<code>movw 2(%rdx,%rcx,8), %ax</code>
<code>S+i-5</code>	<code>short *</code>	<code>xs + 2i - 10</code>	<code>leaq -10(%rdx,%rcx,2), %rax</code>

- 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>