CS107, Lecture 13
Assembly: Control Flow and The Runtime Stack

Reading: B&O 3.6

Learning Goals

- Learn how assembly implements loops and control flow
- Learn how assembly calls functions.

Plan For Today

- Control Flow
  - Condition Codes
  - Assembly Instructions
- **Break:** Announcements
- Function Calls and the Stack

mov Variants

- **mov** only updates the specific register bytes or memory locations indicated.
- **Exception: movl** writing to a register will also set high order 4 bytes to 0.

C declaration	Intel data type	Assembly-code suffix	Size (bytes)
`char`	Byte	`b`	1
`short`	Word	`w`	2
`int`	Double word	`l`	4
`long`	Quad word	`q`	8
`char *`	Quad word	`q`	8
`float`	Single precision	`s`	4
`double`	Double precision	`l`	8

- Suffix sometimes optional if size can be inferred.

No-Op

- The **nop/nopl** instructions are "no-op" instructions – they do nothing!
- No-op instructions do nothing except increment %rip
- Why? To make functions align on nice multiple-of-8 address boundaries.

> "Sometimes, doing nothing is the way to be most productive." – Philosopher Nick

Mov

- Sometimes, you'll see the following: **mov %ebx, %ebx**
- What does this do? It zeros out the top 32 register bits, because when mov is performed on an e- register, the rest of the 64 bits are zeroed out.

xor

- Sometimes, you'll see the following: **xor %ebx, %ebx**
- What does this do? It sets %ebx to zero! May be more efficient than using **mov**.

Plan For Today

- Control Flow
  - Condition Codes
  - Assembly Instructions
- **Break:** Announcements
- Function Calls and the Stack

Control

- In C, we have control flow statements like **if**, **else**, **while**, **for**, etc. that let us write programs that are more expressive than just "straight-line code" (one instruction following another). We can "control" the "flow" of our programs.
- This boils down to *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?
  - A way to store conditions that we will check later
  - Assembly instructions whose behavior is dependent on these conditions

Control

- In C, we have control flow statements like **if**, **else**, **while**, **for**, etc. that let us write programs that are more expressive than just "straight-line code" (one instruction following another). We can "control" the "flow" of our programs.
- This boils down to *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?
  - A way to store conditions that we will check later
  - Assembly instructions whose behavior is dependent on these conditions

Plan For Today

- Control Flow
  - Condition Codes
  - Assembly Instructions
- **Break:** Announcements
- Function Calls and the Stack

Condition Codes

Alongside normal registers, the CPU also has single-bit *condition code* registers. These can be updated and read to influence what to do next. They are automatically updated by 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

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.

Which flag would be set after this code?

```
int a = 5;
int b = -5;
int t = a + b;
```

Condition Codes

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.

Which flag would be set after this code?

```
int a = 5;
int b = -5;
int t = a + b;
```

Condition Codes

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.

Which flag would be set after this code?

```
int a = 5;
int b = -20;
int t = a + b;
```

Condition Codes

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.

Which flag would be set after this code?

```
int a = 5;
int b = -20;
int t = a + b;
```

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.

Setting Condition Codes

- In addition to being set automatically from logical and arithmetic operations, we can also update condition codes ourselves.
- The **cmp** instruction is like the subtraction instruction, but it does not store the result anywhere. It just sets condition codes.

Instruction		Based on	Description
`CMP`	S_1, S_2	$S_2 - S_1$	Compare
`cmpb`			Compare byte
`cmpw`			Compare word
`cmpl`			Compare double word
`cmpq`			Compare quad word

- **NOTE:** the operand order can be confusing!

Setting Condition Codes

- In addition to being set automatically from logical and arithmetic operations, we can also update condition codes ourselves.
- The **test** instruction is like the AND instruction, but it does not store the result anywhere. It just sets condition codes.

Instruction		Based on	Description
`TEST`	S_1, S_2	S_2 & S_1	Test
`testb`			Test byte
`testw`			Test word
`testl`			Test double word
`testq`			Test quad word

Setting Condition Codes

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

Instruction		Based on	Description
`TEST`	S_1, S_2	S_2 & S_1	Test
`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!

Control

- In C, we have control flow statements like **if**, **else**, **while**, **for**, etc. that let us write programs that are more expressive than just "straight-line code" (one instruction following another). We can "control" the "flow" of our programs.
- This boils down to *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?
  - A way to store conditions that we will check later
  - Assembly instructions whose behavior is dependent on these conditions

Plan For Today

- Control Flow
  - Condition Codes
  - Assembly Instructions
- **Break:** Announcements
- Function Calls and the Stack

Condition Code-Dependent Instructions

There are three common instruction types that use condition codes:

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

Conditionally Setting Bytes

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 <=)

Conditionally Moving Data

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`		Nonnegative (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)

Condition Codes

Different combinations of condition codes can indicate different things.

- E.g. To check equality, we can look at the ZERO flag (a = b means a – b = 0)

How can we check whether signed a < b?

- **Overflow may occur!** We have to take that into account.
- **If no overflow:** a < b if a – b < 0, and a >= b if a – b >= 0
- **If overflow:** a < b if a – b > 0 (negative overflow), a > b if a – b < 0 (positive overflow)

Idea: a < b when overflow flag is 1 and sign flag is 0, or when overflow flag is 0 and sign flag is 1. **OF ^ SF**!

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 read from a memory location (indirect jump):

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

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 (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 >) (SF=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)

Loops and Control Flow

Jump instructions are critical to implementing control flow in assembly. Let's see why!

Practice: Fill In The Blank

C Code

```
void if_then(int param1) {
    if (____________) {
        ____________;
    }

    param1 *= ______;
}
```

What does this assembly code translate to?

```
00000000004004fe <if_then>:
  4004fe:    push %rbp
  4004ff:    mov  %rsp,%rbp
  400502:    mov  %edi,-0x4(%rbp)
  400505:    cmpl $0x6,-0x4(%rbp)
  400509:    jne  40050f
  40050b:    addl $0x1,-0x4(%rbp)
  40050f:    shll -0x4(%rbp)
  400512:    pop  %rbp
  400513:    retq
```

Practice: Fill In The Blank

C Code

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

    param1 *= ______;
}
```

What does this assembly code translate to?

```
00000000004004fe <if_then>:
  4004fe:    push %rbp
  4004ff:    mov  %rsp,%rbp
  400502:    mov  %edi,-0x4(%rbp)
  400505:    cmpl $0x6,-0x4(%rbp)
  400509:    jne  40050f
  40050b:    addl $0x1,-0x4(%rbp)
  40050f:    shll -0x4(%rbp)
  400512:    pop  %rbp
  400513:    retq
```

Practice: Fill In The Blank

C Code

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

    param1 *= ______;
}
```

What does this assembly code translate to?

```
00000000004004fe <if_then>:
  4004fe:     push %rbp
  4004ff:     mov  %rsp,%rbp
  400502:     mov  %edi,-0x4(%rbp)
  400505:     cmpl $0x6,-0x4(%rbp)
  400509:     jne  40050f
  40050b:     addl $0x1,-0x4(%rbp)
  40050f:     shll -0x4(%rbp)
  400512:     pop  %rbp
  400513:     retq
```

Practice: Fill In The Blank

C Code

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

    param1 *= 2;
}
```

What does this assembly code translate to?

```
00000000004004fe <if_then>:
  4004fe:    push %rbp
  4004ff:    mov  %rsp,%rbp
  400502:    mov  %edi,-0x4(%rbp)
  400505:    cmpl $0x6,-0x4(%rbp)
  400509:    jne  40050f
  40050b:    addl $0x1,-0x4(%rbp)
  40050f:    shll -0x4(%rbp)
  400512:    pop  %rbp
  400513:    retq
```

Common If-Else Construction

If-Else In C

```
if (num > 3) {
    x = 10;
} else {
    x = 7;
}

num++;
```

If-Else In Assembly

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

Loops and Control Flow

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

```
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
```

Loops and Control Flow

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

```
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
```

Set %eax (i) to 0.

Loops and Control Flow

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

```
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
```

Jump to another instruction.

Loops and Control Flow

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

```
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
```

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++;
    }
}
```

```
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
```

jle means “jump if less than or equal”. The sign flag indicates the result was negative, so we jump.

Loops and Control Flow

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

```
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
```

Add 1 to %eax (i).

Loops and Control Flow

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

```
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
```

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++;
    }
}
```

```
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
```

jle means "jump if less than or equal". The sign flag indicates the result was negative, so we jump.

Loops and Control Flow

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

```
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
```

We continue in this pattern until we do not make this conditional jump. When will that be?

Loops and Control Flow

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

```
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
```

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++;
    }
}
```

```
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
```

Then, we return from the function.

Common Loop Construction

For Loop In C

```
for (int i = 0; i < n; i++) {
     // body
}
```

```
/* equivalent while loop */
int i = 0;
while (i < n) {
    // body
    i++;
}
```

For Loop In Assembly

```
Initialization
Test
Jump past loop if fails
Body
Increment
Jump to test
```

GCC For Loop Output

For Loop In Assembly

```
Initialization
Test
Jump past loop if fails
Body
Increment
Jump to test
```

GCC For Loop Output

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

GCC For Loop Output

For Loop In Assembly

```
Initialization
Test
Jump past loop if fails
Body
Increment
Jump to test
```

GCC For Loop Output

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

GCC For Loop Output

For Loop In Assembly

```
Initialization
Test
Jump past loop if fails
Body
Increment
Jump to test
```

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

GCC For Loop Output

For Loop In Assembly

```
Initialization
Test
Jump past loop if fails
Body
Increment
Jump to test
```

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

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

GCC For Loop Output

For Loop In Assembly

```
Initialization
Test
Jump past loop if fails
Body
Increment
Jump to test
```

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

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

GCC For Loop Output

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

GCC For Loop Output

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

GCC For Loop Output

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

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

GCC For Loop Output

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

GCC For Loop Output

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

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

GCC For Loop Output

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

GCC For Loop Output

For Loop In Assembly

```
Initialization
Test
Jump past loop if fails
Body
Increment
Jump to test
```

GCC For Loop Output

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

Which instructions are better when n = 0?

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

Optimizing Instruction Counts

- Both of these loop forms have the same **static instruction count** – same number of written instructions.
- But they have different **dynamic** instruction counts – the number of times these instructions are executed when the program is run.
  - If n = 0, left is best
  - If n is large, right is best
- The compiler may emit static instruction counts many times longer than alternatives, but which is more efficient if loop executes many times.
- Problem: the compiler may not know whether the loop will execute many times! Hard problem….. (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.

Practice: Fill In The Blank

C Code

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

What does this assembly code translate to?

```
// a in %rdi, b in %rsi
loop:
    movl $1, %eax
    jmp .L2
.L3
    leaq (%rdi,%rsi), %rdx
    imulq %rdx, %rax
    addq $1, %rdi
.L2
    cmpq %rsi, %rdi
    jl .L3
rep; ret
```

Practice: Fill In The Blank

C Code

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

What does this assembly code translate to?

```
// a in %rdi, b in %rsi
loop:
    movl $1, %eax
    jmp .L2
.L3
    leaq (%rdi,%rsi), %rdx
    imulq %rdx, %rax
    addq $1, %rdi
.L2
    cmpq %rsi, %rdi
    jl .L3
rep; ret
```

Plan For Today

- Control Flow
  - Condition Codes
  - Assembly Instructions
- **Break:** Announcements
- Function Calls and the Stack

Announcements

- Midterms have been graded, and will be returned after class
- Visit gradescope.com (you should receive an email) to view your exam and score
- Regrades accepted until next Friday at 2PM

Plan For Today

- Control Flow
  - Condition Codes
  - Assembly Instructions
- Instruction Pointer
- **Break:** Announcements
- Function Calls and the Stack

How do we call functions in assembly?

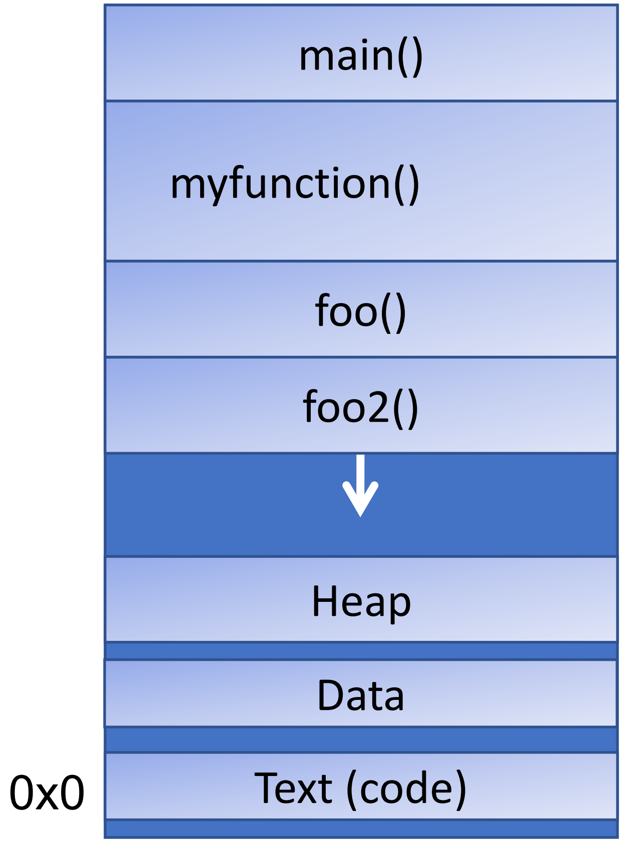
Calling Functions In Assembly

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

- **Pass Control** – %rip must be adjusted to execute the 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 caller on the stack.

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

Stack Frame



Register Responsibilities

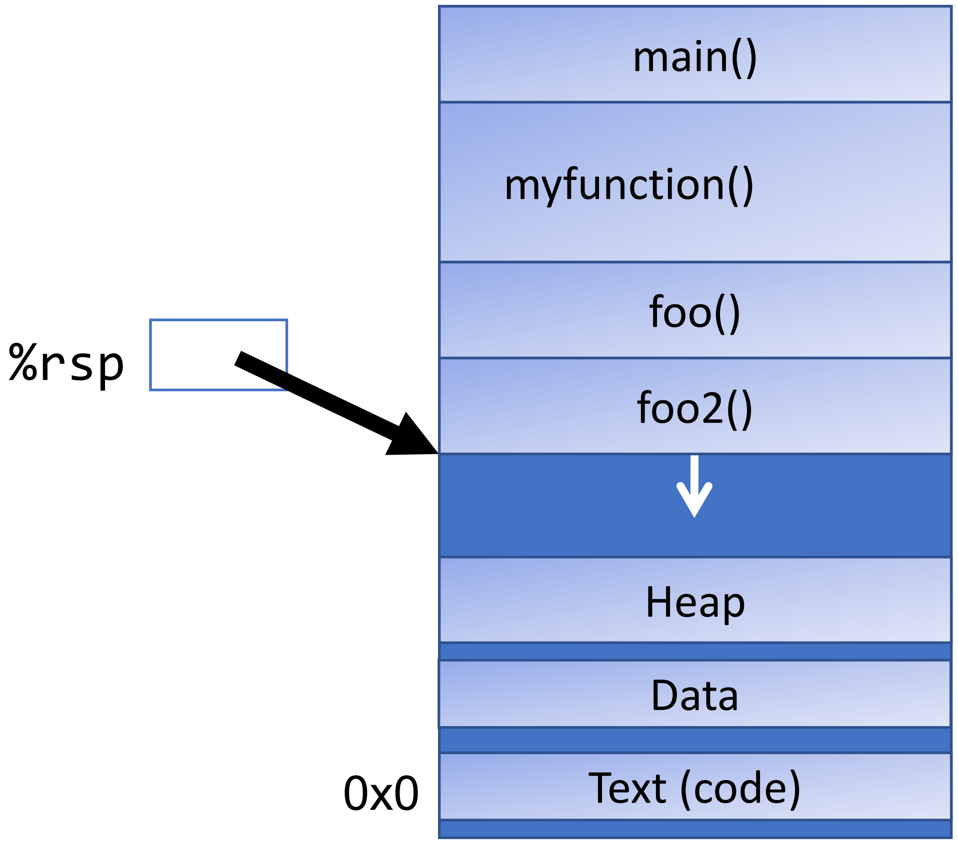
Some registers take on special responsibilities during program execution.

- **%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 element on the stack

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

%rsp

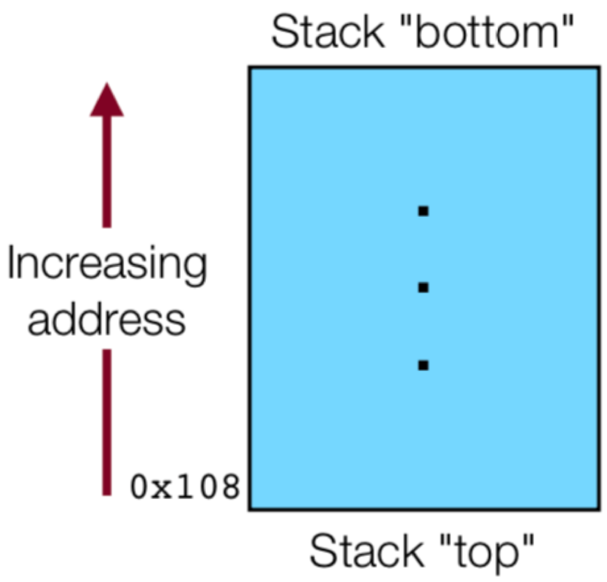
Memory



Pushing/Popping with the Stack

- The push and pop operations write and read from the stack, and they also modify the stack pointer, `%rsp`:

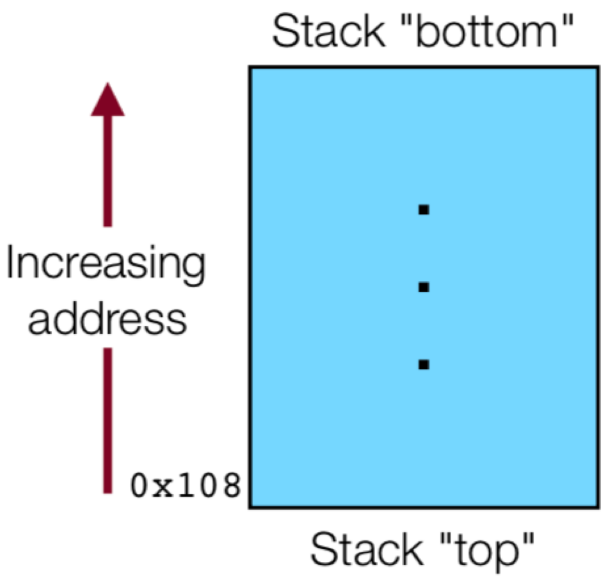
Instruct	Effect	Description
`pushq` *S*	R[`%rsp`] ← R[`%rsp`]-8; M[R[%rsp]] ← *S*	Push quad word
`popq` *D*	D ← M[R[`%rsp`]]; R[`%rsp`] ← R[`%rsp`]+8	Push quad word



Pushing onto the Stack

- Example:

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

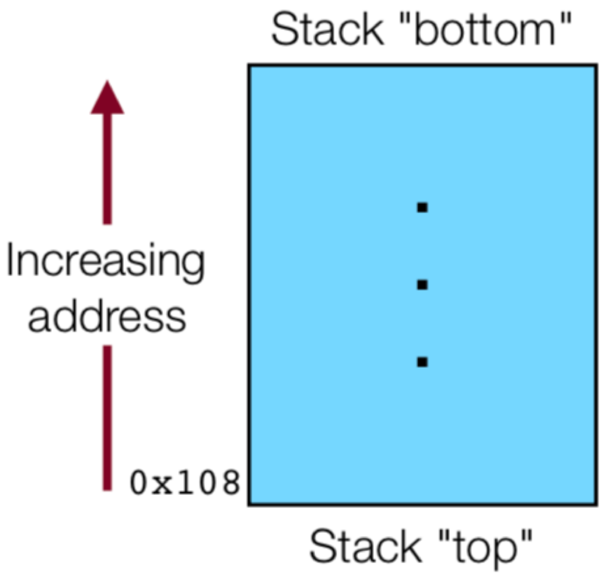


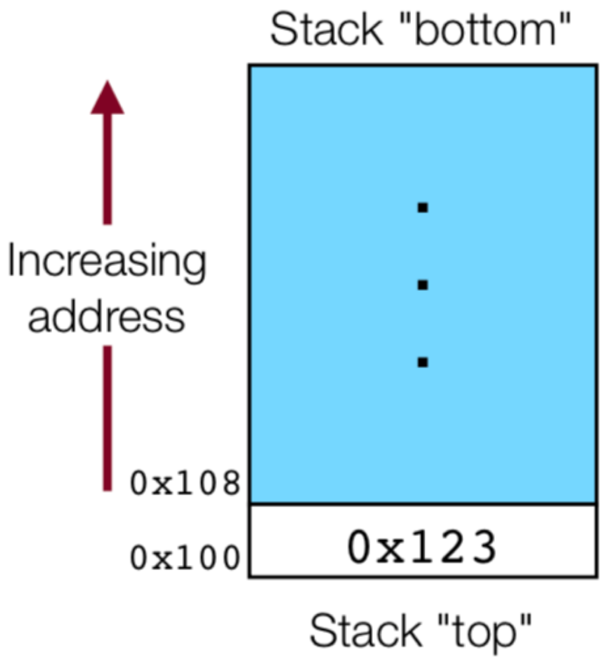
Pushing onto the Stack

- Example:

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

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





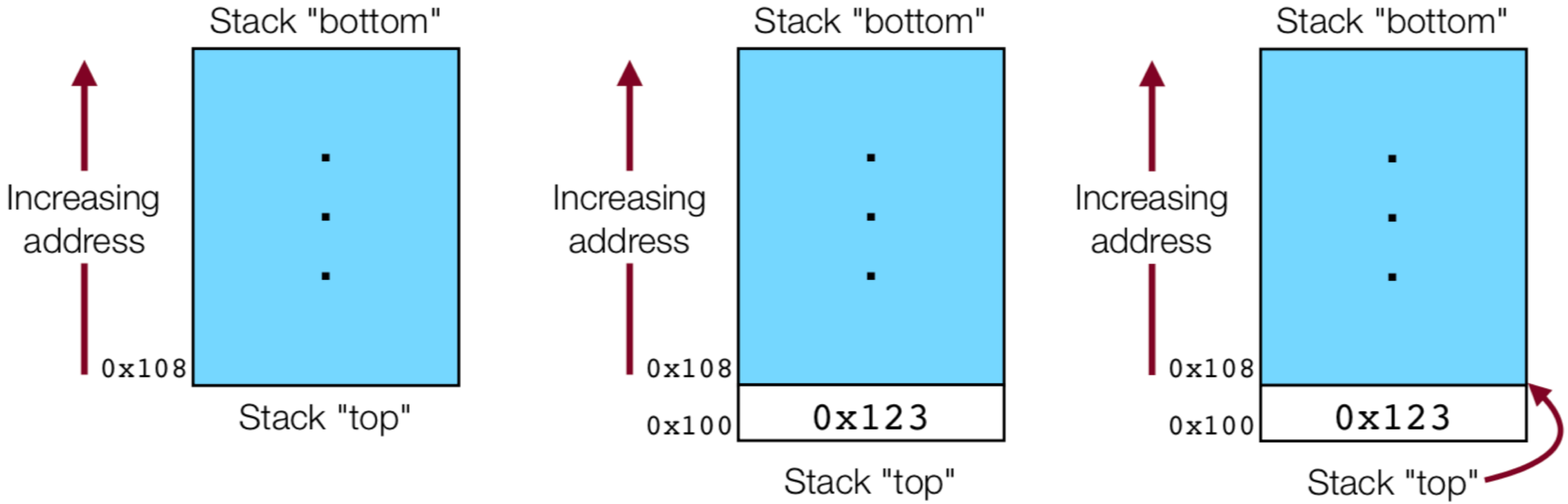
Pushing onto the Stack

- Example:

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

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

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



Pushing/Popping with the Stack

- Pushing a quad word onto the stack means decrementing the stack pointer by 8, and writing the value to the new top-of-stack address. Equivalent (e.g. with %rax):

```
subq   $8, %rsp
movq %rax, (%rsp)
```

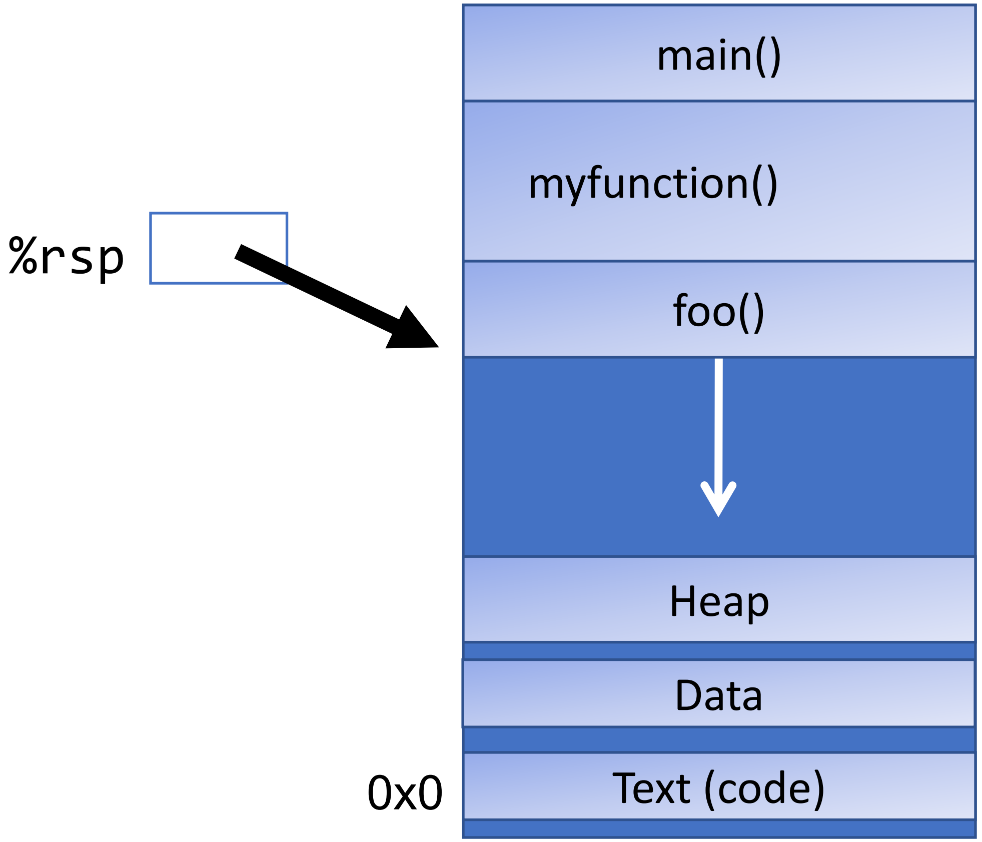
- Popping a quad word off the stack means reading the value at %rsp, and then incrementing the stack pointer by 8. Equivalent (e.g. with %rdx):

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

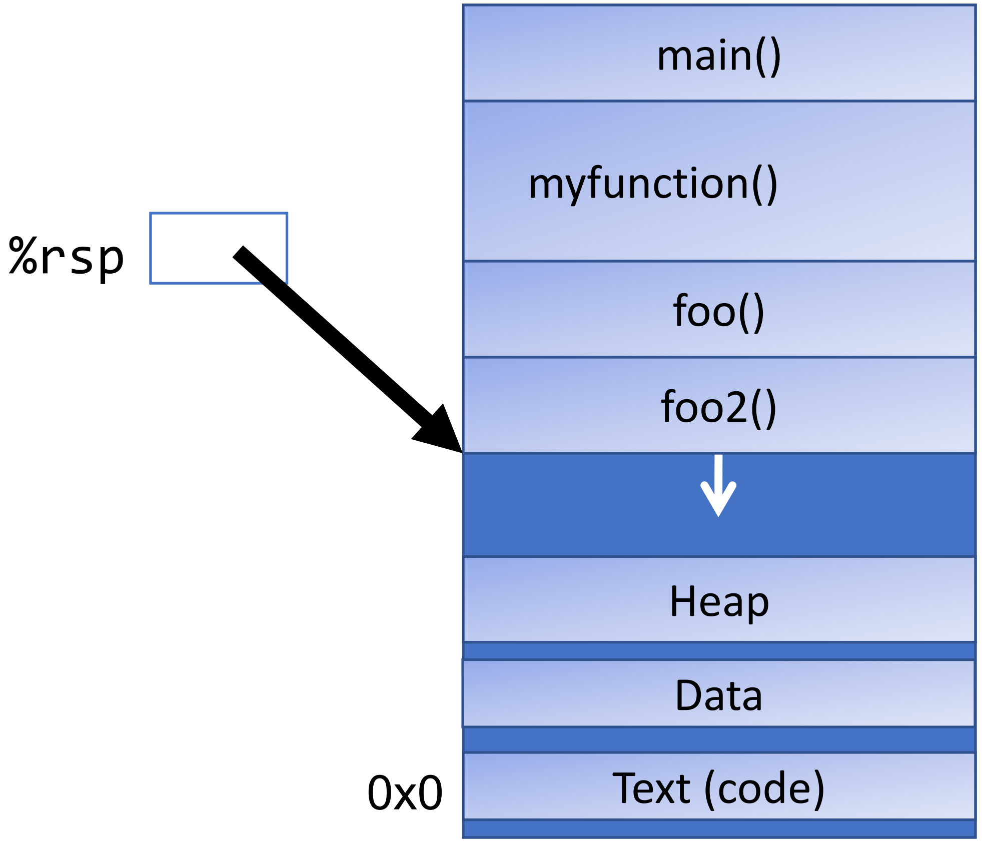
Key Idea: %rsp

%rsp must point to the same place before and after a function is called.

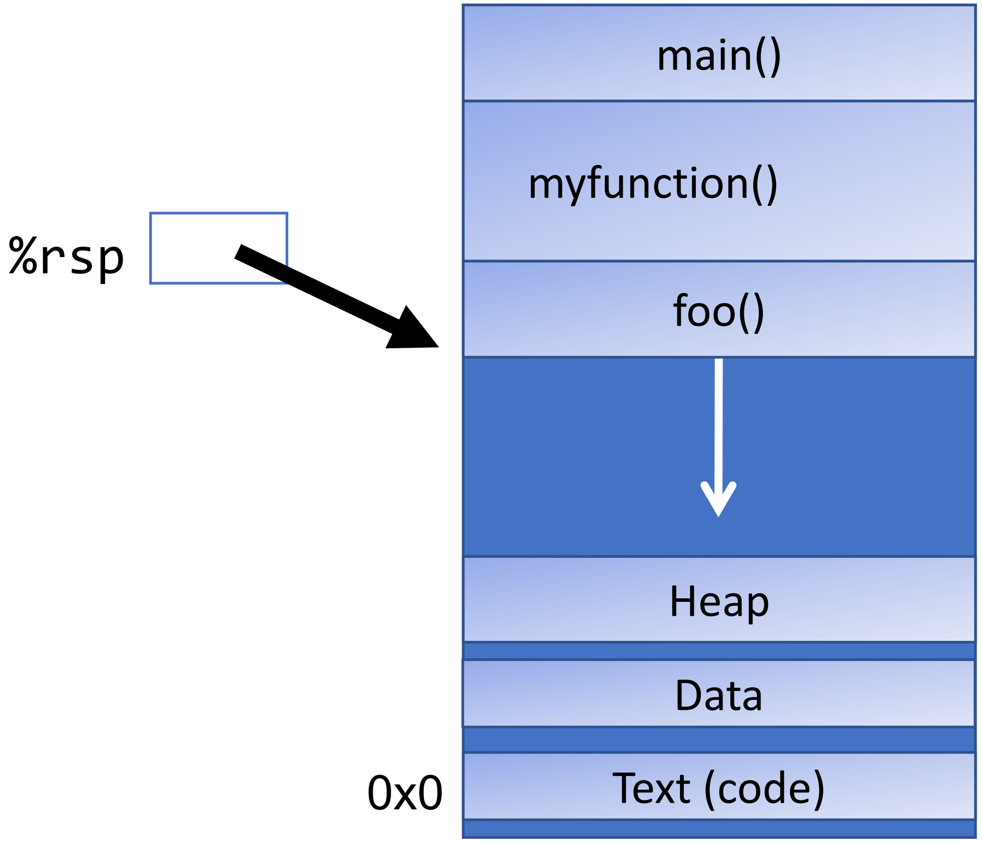
Stack Frame



Stack Frame



Stack Frame



Calling Functions In Assembly

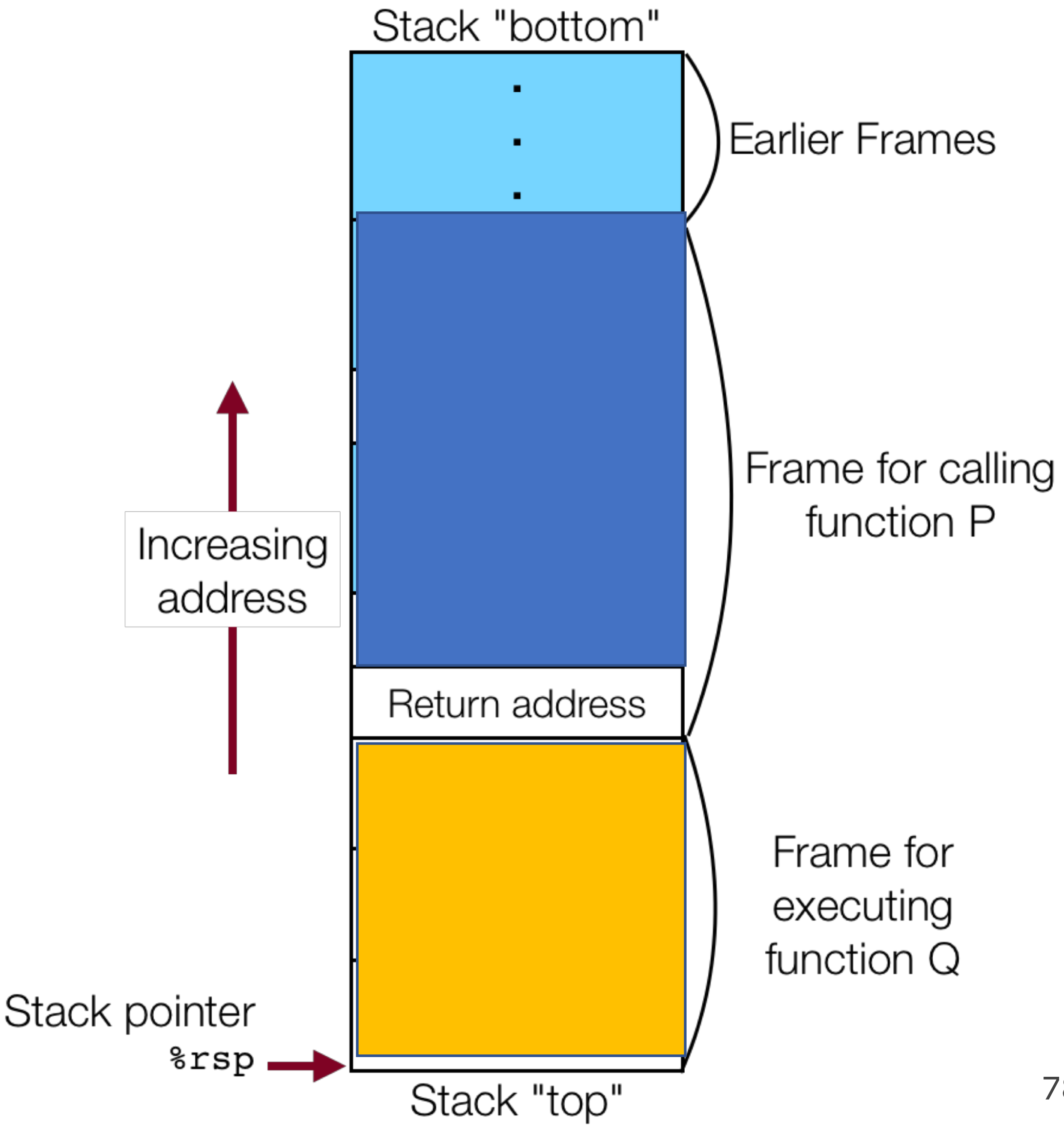
To call a function in assembly, we must do a few things:

- **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 caller on the stack.

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

Passing Control

- **Problem:** %rip stores the current instruction being executed. If we execute the callee's instructions, we must **remember** what instruction to resume at in the caller after!
- **Solution:** use the **callq** command to push the current value of %rip at the bottom of the caller's stack frame before calling the function. Then after the callee is finished, use the **ret** instruction to put this value back into %rip and continue executing.



Call And Return

The **call** instruction pushes the value of %rip onto the stack and sets %rip to point to the beginning of the specified function.

```
call Label
call *Operand
```

The **ret** instruction pops the value of %rip from the stack and sets %rip to store this value.

```
ret
```

Recap

- Control Flow
  - Condition Codes
  - Assembly Instructions
- Instruction Pointer
- **Break:** Announcements
- Function Calls and the Stack

Next time: more function calls and optimizations