# CS107, Lecture 13
## Assembly: Control Flow

Reading: B&O 3.6

# Learning Assembly

Moving data around → Arithmetic and logical operations → Control flow → Function calls

**5/6** | **5/13** | **Today** | **5/20**

# Learning Goals

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

# Plan For Today

- Recap: Arithmetic and Logic
- Control Flow
  - Condition Codes
  - Assembly Instructions
  - If statements
  - While loops
  - For loops

# Plan For Today

- **Recap: Arithmetic and Logic**
- Control Flow
  - Condition Codes
  - Assembly Instructions
  - If statements
  - While loops
  - For loops

# 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!

# mov Variants

- **mov** can take an optional suffix (b,w,l,q) that specifies the size of data to move: `movb, movw, movl, movq`
- **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.

# lea

The **lea** instruction <u>copies</u> an “effective address” from one place to another.

```
lea        src,dst
```

Unlike **mov**, which copies data <u>at</u> the address src to the destination, **lea** copies the value of src *itself* to the destination.

# No-Op

- The **nop/nopl** instructions are "no-op" instructions – they do nothing!
- 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**.

# Assembly Exercise 1

```
00000000004005ac <sum_example1>:
    4005bd:   8b 45 e8        mov  %esi,%eax
    4005c3:   01 d0           add  %edi,%eax
    4005cc:   c3              retq
```

Which of the following is most likely to have generated the above assembly?

```
// A)
void sum_example1() {
    int x;
    int y;
    int sum = x + y;
}
```

```
// B)
int sum_example1(int x, int y) {
    return x + y;
}
```

```
// C)
void sum_example1(int x, int y) {
    int sum = x + y;
}
```

# Assembly Exercise 2

```
0000000000400578 <sum_example2>:
     400578:   8b 47 0c         mov  0xc(%rdi),%eax
     40057b:   03 07            add  (%rdi),%eax
     40057d:   2b 47 18         sub  0x18(%rdi),%eax
     400580:   c3               retq
```

```c
int sum_example2(int arr[]) {
    int sum = 0;
    sum += arr[0];
    sum += arr[3];
    sum -= arr[6];
    return sum;
}
```

What location or value in the assembly above represents the C code's **sum** variable?

# Assembly Exercise 2

```
0000000000400578 <sum_example2>:
    400578:   8b 47 0c        mov  0xc(%rdi),%eax
    40057b:   03 07           add  (%rdi),%eax
    40057d:   2b 47 18        sub  0x18(%rdi),%eax
    400580:   c3              retq
```

```
int sum_example2(int arr[]) {
    int sum = 0;
    sum += arr[0];
    sum += arr[3];
    sum -= arr[6];
    return sum;
}
```

What location or value in the assembly above represents the C code's **sum** variable?

**%eax**

# Assembly Exercise 3

```
0000000000400578 <sum_example2>:
    400578:   8b 47 0c        mov  0xc(%rdi),%eax
    40057b:   03 07           add  (%rdi),%eax
    40057d:   2b 47 18        sub  0x18(%rdi),%eax
    400580:   c3              retq
```

```c
int sum_example2(int arr[]) {
    int sum = 0;
    sum += arr[0];
    sum += arr[3];
    sum -= arr[6];
    return sum;
}
```

What location or value in the assembly code above represents the C code's **6** (as in **arr[6]**)?

# Assembly Exercise 3

```
0000000000400578 <sum_example2>:
     400578:   8b 47 0c          mov  0xc(%rdi),%eax
     40057b:   03 07             add  (%rdi),%eax
     40057d:   2b 47 18          sub  0x18(%rdi),%eax
     400580:   c3                retq
```

```c
int sum_example2(int arr[]) {
    int sum = 0;
    sum += arr[0];
    sum += arr[3];
    sum -= arr[6];
    return sum;
}
```

What location or value in the assembly code above represents the C code's **6** (as in **arr[6]**)?

**0x18**

# Plan For Today

- Recap: Arithmetic and Logic
- **Control Flow**
  - Condition Codes
  - Assembly Instructions
  - If statements
  - While loops
  - For loops

# 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?
    - A way to store conditions that we will check later
    - Assembly instructions whose behavior is dependent on these conditions

# Control

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

In Assembly:

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

There are special “condition code” registers that automatically store the results of the most recent arithmetic or logical operation.

# 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

**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

- 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)
- 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. (**Note** the operand order!)

| 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 |

# 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 |

**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. 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?
  - A way to store conditions that we will check later
  - <u>**Assembly instructions whose behavior is dependent on these conditions**</u>

# Plan For Today

- Recap: Arithmetic and Logic
- **Control Flow**
  - Condition Codes
  - **Assembly Instructions**
  - If statements
  - While loops
  - For loops

# 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) |

# 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!

# Plan For Today

- Recap: Arithmetic and Logic
- **Control Flow**
  - Condition Codes
  - Assembly Instructions
  - **If statements**
  - While loops
  - For loops

# Practice: Fill In The Blank

**C Code**

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

    return ______;
}
```

**What does this assembly code translate to?**

```
00000000004004d6 <if_then>:
  4004d6:  cmp    $0x6,%edi
  4004d9:  jne    4004de
  4004db:  add    $0x1,%edi
  4004de:  lea    (%rdi,%rdi,1),%eax
  4004e1:  retq
```

# Practice: Fill In The Blank

**C Code**

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

    return ______;
}
```

**What does this assembly code translate to?**

```
00000000004004d6 <if_then>:
  4004d6:  cmp    $0x6,%edi
  4004d9:  jne    4004de
  4004db:  add    $0x1,%edi
  4004de:  lea    (%rdi,%rdi,1),%eax
  4004e1:  retq
```

# Practice: Fill In The Blank

**C Code**

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

    return ______;
}
```

**What does this assembly code translate to?**

```
00000000004004d6 <if_then>:
  4004d6:  cmp    $0x6,%edi
  4004d9:  jne    4004de
  4004db:  add    $0x1,%edi
  4004de:  lea    (%rdi,%rdi,1),%eax
  4004e1:  retq
```

# Practice: Fill In The Blank

**C Code**

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

    return param1 * 2;
}
```

**What does this assembly code translate to?**

```
00000000004004d6 <if_then>:
  4004d6:  cmp    $0x6,%edi
  4004d9:  jne    4004de
  4004db:  add    $0x1,%edi
  4004de:  lea    (%rdi,%rdi,1),%eax
  4004e1:  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
```

# Announcements

- Midterm statistics updated on course website
- Assignment 6 released tomorrow – Assembly exercises
  - Security
  - Reverse engineering

# Plan For Today

- Recap: Arithmetic and Logic
- **Control Flow**
  - Condition Codes
  - Assembly Instructions
  - If statements
  - **While loops**
  - For loops

# 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

```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
```

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

```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
```

**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

```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
```

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

```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
```

Then, we return from the function.

# Common While Loop Construction

**C**

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

**Assembly**

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

**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
```

# Plan For Today

- Recap: Arithmetic and Logic
- **Control Flow**
  - Condition Codes
  - Assembly Instructions
  - If statements
  - While loops
  - **For loops**

# Common For Loop Construction

**C**

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

# Common For Loop Construction

**C**

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

**C Equivalent While Loop**

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

# Common For Loop Construction

**C**

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

**C Equivalent While Loop**

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

**While Loop Assembly**

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

# Common For Loop Construction

**C**

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

**C Equivalent While Loop**

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

**For Loop Assembly**

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

# GCC For Loop Output

**GCC For Loop Output**

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

**Possible Alternative?**

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

# GCC For Loop Output

**GCC For Loop Output**

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

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

# GCC For Loop Output

**GCC For Loop Output**

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

**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
...
```

# GCC For Loop Output

**GCC For Loop Output**

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

```
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
...
```

# GCC For Loop Output

```
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
...
```

**Possible Alternative?**

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

# GCC For Loop Output

```
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
...
```

**Possible Alternative?**

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

# GCC For Loop Output

**GCC For Loop Output**

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

**Possible Alternative?**

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

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, right is best
  - If n is large, left 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

- Recap: Arithmetic and Logic
- Control Flow
  - Condition Codes
  - Assembly Instructions
  - If statements
  - While loops
  - For loops

**Next time:** function calls in assembly