https://forms.gle/skDnZX9FquXqHuJE8

Code: jmp
CS 107, Lecture 12
Control Flow: When in doubt just JMP!

Reading: B&O 3.1-3.4
Learning Goals

• Learn about how assembly stores comparison and operation results in condition codes
• Understand how assembly implements loops and control flow
• 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
• Live Session Slides
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
- Live Session Slides
What does it mean for a program to execute?
Executing Instructions

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

Today:
• **Who controls the instructions?**
  How do we know what to do now or next?

Answer:
• The **program counter** (PC), %rip.
Register Responsibilities

Some registers take on special responsibilities during program execution.

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

See the x86-64 Guide and Reference Sheet on the Resources webpage for more!
Instructions Are Just Bytes!

- Stack
- Heap
- Data
- Text (code)

Machine code instructions

0x0
000000000004004ed <loop>:
4004ed:  55  push %rbp
4004ee: 48 89 e5  mov  %rsp,%rbp
4004f1: c7 45 fc 00 00 00 00  movl  $0x0,-0x4(%rbp)
4004f8: 83 45 fc 01  addl  $0x1,-0x4(%rbp)
4004fc: eb fa  jmp  4004f8 <loop+0xb>
The program counter (PC), known as %rip in x86-64, stores the address in memory of the next instruction to be executed.
00000000004004ed <loop>:

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

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

%rip += size of bytes of current instruction

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x4004fc</td>
<td>FA</td>
</tr>
<tr>
<td>0x4004fd</td>
<td>FA</td>
</tr>
<tr>
<td>0x4004fe</td>
<td>EB</td>
</tr>
<tr>
<td>0x4004ff</td>
<td>01</td>
</tr>
<tr>
<td>0x400500</td>
<td>FC</td>
</tr>
<tr>
<td>0x400501</td>
<td>45</td>
</tr>
<tr>
<td>0x400502</td>
<td>83</td>
</tr>
<tr>
<td>0x400503</td>
<td>45</td>
</tr>
<tr>
<td>0x400504</td>
<td>01</td>
</tr>
<tr>
<td>0x400505</td>
<td>00</td>
</tr>
<tr>
<td>0x400506</td>
<td>00</td>
</tr>
<tr>
<td>0x400507</td>
<td>00</td>
</tr>
<tr>
<td>0x400508</td>
<td>00</td>
</tr>
<tr>
<td>0x400509</td>
<td>00</td>
</tr>
<tr>
<td>0x40050a</td>
<td>00</td>
</tr>
<tr>
<td>0x40050b</td>
<td>00</td>
</tr>
<tr>
<td>0x40050c</td>
<td>FC</td>
</tr>
<tr>
<td>0x40050d</td>
<td>45</td>
</tr>
<tr>
<td>0x40050e</td>
<td>C7</td>
</tr>
<tr>
<td>0x40050f</td>
<td>E5</td>
</tr>
<tr>
<td>0x400510</td>
<td>89</td>
</tr>
<tr>
<td>0x400511</td>
<td>48</td>
</tr>
<tr>
<td>0x400512</td>
<td>55</td>
</tr>
</tbody>
</table>

%rip

0x4004fc <loop>:

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

• How can we use this representation of execution to represent e.g. a loop?
• **Key Idea:** we can ”interfere” with `%rip` and set it back to an earlier instruction!
The jmp instruction is an unconditional jump that sets the program counter to the jump target (the operand).
The jmp instruction is an unconditional jump that sets the program counter to the jump target (the operand).
The **jmp** instruction is an **unconditional jump** that sets the program counter to the **jump target** (the operand).
The jmp instruction is an **unconditional jump** that sets the program counter to the **jump target** (the operand).

The jmp instruction is an **unconditional jump** that sets the program counter to the **jump target** (the operand).
This assembly represents an infinite loop in C!

```
while (true) {...}
```
The **jmp** instruction jumps to another instruction in the assembly code ("Unconditional Jump").

\[
\text{jmp Label} \quad \text{(Direct Jump)} \\
\text{jmp *Operand} \quad \text{(Indirect Jump)}
\]

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

\[
\text{jmp 404f8 <loop+0xb>} \quad \# \text{ jump to instruction at 0x404f8}
\]

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

\[
\text{jmp *%rax} \quad \# \text{ jump to instruction at address in %rax}
\]
1. How do we repeat instructions in a loop?

jmp [target]
- A 1-step unconditional jump (always jump when we execute this instruction)

What if we want a \textit{conditional} jump?
Lecture Plan

• Assembly Execution and %rip 5
• Control Flow Mechanics 27
  • Condition Codes
  • Assembly Instructions
• If statements 46
• Loops 54
  • While loops 67
  • For loops
• Other Instructions That Depend On Condition Codes 73
• Live Session Slides 81
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?
if (x > y) {
    // a
} else {
    // b
}

In Assembly:
1. Calculate the condition result
2. Based on the result, go to a or b
Control

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

Common Pattern:

1. `cmp S1, S2`  // compare two values
2. `je [target] or jne [target] or jl [target] or ...`  // conditionally jump

“jump if equal”  “jump if not equal”  “jump if less than”
Conditional Jumps

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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Synonym</th>
<th>Set Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>je <code>Label</code></td>
<td>jz</td>
<td>Equal / zero</td>
</tr>
<tr>
<td>jne <code>Label</code></td>
<td>jnz</td>
<td>Not equal / not zero</td>
</tr>
<tr>
<td>js <code>Label</code></td>
<td></td>
<td>Negative</td>
</tr>
<tr>
<td>jns <code>Label</code></td>
<td></td>
<td>Nonnegative</td>
</tr>
<tr>
<td>jg <code>Label</code></td>
<td>jnle</td>
<td>Greater (signed &gt;)</td>
</tr>
<tr>
<td>jge <code>Label</code></td>
<td>jnl</td>
<td>Greater or equal (signed &gt;=)</td>
</tr>
<tr>
<td>jl <code>Label</code></td>
<td>jnge</td>
<td>Less (signed &lt;)</td>
</tr>
<tr>
<td>jle <code>Label</code></td>
<td>jng</td>
<td>Less or equal (signed &lt;=)</td>
</tr>
<tr>
<td>ja <code>Label</code></td>
<td>jnbe</td>
<td>Above (unsigned &gt;)</td>
</tr>
<tr>
<td>jae <code>Label</code></td>
<td>jnb</td>
<td>Above or equal (unsigned &gt;=)</td>
</tr>
<tr>
<td>jb <code>Label</code></td>
<td>jnae</td>
<td>Below (unsigned &lt;)</td>
</tr>
<tr>
<td>jbe <code>Label</code></td>
<td>jna</td>
<td>Below or equal (unsigned &lt;=)</td>
</tr>
</tbody>
</table>
Control

Read `cmp S1,S2` as “compare S2 to S1”:

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

// Jump if %edi <= 1
cmp $1, %edi
jle [target]
Read \texttt{cmp S1,S2} as “\textit{compare S2 to S1}”: 

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

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

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

// Jump if %edi <= 1
```
• The CPU has special registers called *condition codes* that are like “global variables”. They *automatically* keep track of information about the most recent arithmetic or logical operation.
  • *cmp* compares via calculation (subtraction) and info is stored in the condition codes
  • conditional jump instructions look at these condition codes to know whether to jump

• What exactly are the condition codes? How do they store this information?
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.
Alongside normal registers, the CPU also has single-bit *condition code* registers. They store the results of the most recent arithmetic or logical operation.

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

- **CF:** Carry flag (Unsigned Overflow). \((\text{unsigned}) \ t < (\text{unsigned}) \ a\)
- **ZF:** Zero flag (Zero). \((t == 0)\)
- **SF:** Sign flag (Negative). \((t < 0)\)
- **OF:** Overflow flag (Signed Overflow). \((a<0 == b<0) \ &\ & (t<0 != a<0)\)
The `cmp` instruction is like the subtraction instruction, but it does not store the result anywhere. It just sets condition codes. (Note the operand order!)

```
CMP S1, S2  S2 - S1
```

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmpb</td>
<td>Compare byte</td>
</tr>
<tr>
<td>cmpw</td>
<td>Compare word</td>
</tr>
<tr>
<td>cmpd1</td>
<td>Compare double word</td>
</tr>
<tr>
<td>cmpq</td>
<td>Compare quad word</td>
</tr>
</tbody>
</table>
Control

Read \texttt{cmp S1,S2} as "\textit{compare S2 to S1}". It calculates S2 – S1 and updates the condition codes with the result.

\begin{verbatim}
// Jump if %edi > 2
// calculates %edi - 2
cmp $2, %edi
jg [target]

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

// Jump if %edi <= 1
// calculates %edi - 1
cmp $1, %edi
jle [target]
\end{verbatim}
Conditional Jumps

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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Synonym</th>
<th>Set Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>je Label</td>
<td>jz</td>
<td>Equal / zero (ZF = 1)</td>
</tr>
<tr>
<td>jne Label</td>
<td>jnz</td>
<td>Not equal / not zero (ZF = 0)</td>
</tr>
<tr>
<td>js Label</td>
<td></td>
<td>Negative (SF = 1)</td>
</tr>
<tr>
<td>jns Label</td>
<td></td>
<td>Nonnegative (SF = 0)</td>
</tr>
<tr>
<td>jg Label</td>
<td>jn1e</td>
<td>Greater (signed &gt;) (ZF = 0 and SF = OF)</td>
</tr>
<tr>
<td>jge Label</td>
<td>jn1</td>
<td>Greater or equal (signed &gt;=) (SF = OF)</td>
</tr>
<tr>
<td>jl Label</td>
<td>jnge</td>
<td>Less (signed &lt;) (SF != OF)</td>
</tr>
<tr>
<td>jle Label</td>
<td>jng</td>
<td>Less or equal (signed &lt;=) (ZF = 1 or SF! = OF)</td>
</tr>
<tr>
<td>ja Label</td>
<td>jnbe</td>
<td>Above (unsigned &gt;) (CF = 0 and ZF = 0)</td>
</tr>
<tr>
<td>jae Label</td>
<td>jnb</td>
<td>Above or equal (unsigned &gt;=) (CF = 0)</td>
</tr>
<tr>
<td>jb Label</td>
<td>jnae</td>
<td>Below (unsigned &lt;) (CF = 1)</td>
</tr>
<tr>
<td>jbe Label</td>
<td>jna</td>
<td>Below or equal (unsigned &lt;=) (CF = 1 or ZF = 1)</td>
</tr>
</tbody>
</table>
The **test** instruction is like **cmp**, but for AND. It does not store the & result anywhere. It just sets condition codes.

TEST S1, S2  
S2 & S1

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>testb</td>
<td>Test byte</td>
</tr>
<tr>
<td>testw</td>
<td>Test word</td>
</tr>
<tr>
<td>testl</td>
<td>Test double word</td>
</tr>
<tr>
<td>testq</td>
<td>Test quad word</td>
</tr>
</tbody>
</table>

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

• Previously-discussed arithmetic and logical instructions update these flags. `lea` does not (it was intended only for address computations).

• Logical operations (`xor`, etc.) set carry and overflow flags to zero.

• Shift operations set the carry flag to the last bit shifted out and set the overflow flag to zero.

• For more complicated reasons, `inc` and `dec` set the overflow and zero flags, but leave the carry flag unchanged.
Exercise 1: Conditional jump

je target  
jump if ZF is 1

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

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

2. test $0x10,%edi
   je  40056f
   add  $0x1,%edi
Exercise 1: Conditional jump

je target  # jump if ZF is 1

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

1. cmp $0x10,%edi
   je 40056f
   add $0x1,%edi
   S2 - S1 == 0, so jump

2. test $0x10,%edi
   je 40056f
   add $0x1,%edi
   S2 & S1 != 0, so don’t jump
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
- Live Session Slides
If Statements

How can we use instructions like `cmp` and *conditional jumps* to implement if statements in assembly?
int if_then(int param1) {
    if ( __________ ) {
        __________;
    }
    return __________;
}

0000000000401126 <if_then>:
401126:    cmp $0x6,%edi
401129:    je  40112f
40112b:    lea (%rdi,%rdi,1),%eax
40112e:    retq
40112f:    add $0x1,%edi
401132:    jmp 40112b
int if_then(int param1) {
  if (param1 == 6) {
    param1++;
  }
  return param1 * 2;
}

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

If-Else In C

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

If-Else In Assembly pseudocode

```assembly
Test
Jump to else-body if test passes
If-body
Jump to past else-body
Else-body
Past else body
```
Practice: Fill in the Blank

### If-Else In C

```c
long absdiff(long x, long y) {
    long result;
    if (_______) {
        ________________ ;
    } else {
        ________________ ;
    }
    return result;
}
```

### If-Else In Assembly pseudocode

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

**Test**

Jump to else-body if test **passes**

**If-body**

Jump to past else-body

**Else-body**

🤔
Practice: Fill in the Blank

If-Else In C

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

If-Else In Assembly pseudocode

Test
Jump to else-body if test passes

If-body
Jump to past else-body

Else-body
Past else body
C Code
int test(int arg) {
    int ret;
    if (arg > 3) {
        ret = 10;
    } else {
        ret = 0;
    }
}
ret++; return ret;

Assembly
401134 <+0>: cmp $0x3,%edi
401137 <+3>: jle 0x401142 <test+14>
401139 <+5>: mov $0xa,%eax
40113e <+10>: add $0x1,%eax
401141 <+13>: retq
401142 <+14>: mov $0x0,%eax
401147 <+19>: jmp 0x40113e <test+10>
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
• Live Session Slides
void loop() {
    int i = 0;
    while (i < 100) {
        i++;
    }
}
Loops and Control Flow

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

<table>
<thead>
<tr>
<th>Address</th>
<th>Opcode</th>
<th>Instruction</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x000000000040115c &lt;+0&gt;:</td>
<td>mov</td>
<td>$0x0000000000401161 &lt;+5&gt;:</td>
<td>mov $0x0000000000401161 0x0x63,%eax</td>
</tr>
<tr>
<td>0x0000000000401161 &lt;+5&gt;:</td>
<td>cmp</td>
<td>$0x0000000000401164 &lt;+8&gt;:</td>
<td>cmp $0x0000000000401164 0x0x63,%eax</td>
</tr>
<tr>
<td>0x0000000000401164 &lt;+8&gt;:</td>
<td>jg</td>
<td>$0x0000000000401166 &lt;+10&gt;:</td>
<td>jg 0x0000000000401166 0x0x16b,0x16b</td>
</tr>
<tr>
<td>0x0000000000401166 &lt;+10&gt;:</td>
<td>add</td>
<td>$0x0000000000401169 &lt;+13&gt;:</td>
<td>add $0x0000000000401169 0x0x16b,0x16b</td>
</tr>
<tr>
<td>0x0000000000401169 &lt;+13&gt;:</td>
<td>jmp</td>
<td>$0x000000000040116b &lt;+15&gt;:</td>
<td>jmp 0x000000000040116b 0x0x16b,0x16b</td>
</tr>
<tr>
<td>0x000000000040116b &lt;+15&gt;:</td>
<td>retq</td>
<td></td>
<td>retq</td>
</tr>
</tbody>
</table>

Set %eax (i) to 0.
Loops and Control Flow

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

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

\textit{jg} means “jump if greater than”. This jumps if \%eax > 0x63. The flags indicate this is false, so we do not jump.
void loop() {
    int i = 0;
    while (i < 100) {
        i++;
    }
}
Loops and Control Flow

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

Jump to another instruction.
Loops and Control Flow

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

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

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

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++;
    }
}

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++;
    }
}
```

Then, we return from the function.
GCC Common While Loop Construction

C
while (test) {
    body
}

<table>
<thead>
<tr>
<th>Assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test</td>
</tr>
<tr>
<td>Skip loop if test passes</td>
</tr>
<tr>
<td>Body</td>
</tr>
<tr>
<td>Jump back to test</td>
</tr>
</tbody>
</table>

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 5
• Control Flow Mechanics 27
  • Condition Codes
  • Assembly Instructions
• If statements 46

• Loops 54
  • While loops 67
  • For loops
• Other Instructions That Depend On Condition Codes 73
• Live Session Slides 81
Common For Loop Construction

C For loop
for (init; test; update) {
    body
}

C Equivalent While Loop
init
while(test) {
    body
    update
}

Assembly pseudocode
- Init
- Test
- Skip loop if test passes
- Body
- Update
- Jump back to test

For loops and while loops are treated (essentially) the same when compiled down to assembly.
int sum_array(int arr[], int nelems) {
    int sum = 0;
    for (int i = 0; i < nelems; i++) {
        sum += arr[i];
    }
    return sum;
}

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
  5

• Control Flow Mechanics
  27
  • Condition Codes
  • Assembly Instructions

• If statements
  46

• Loops
  54
  • While loops
  67
  • For loops

• Other Instructions That Depend On Condition Codes
  73

• Live Session Slides
  81
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

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

```c
    cmp $0xf,%edi  
    setle %al     
    movzbl %al, %eax 
    retq
```
### set: Read condition codes

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Synonym</th>
<th>Set Condition (1 if true, 0 if false)</th>
</tr>
</thead>
<tbody>
<tr>
<td>sete D</td>
<td>setz</td>
<td>Equal / zero</td>
</tr>
<tr>
<td>setne D</td>
<td>setnz</td>
<td>Not equal / not zero</td>
</tr>
<tr>
<td>sets D</td>
<td></td>
<td>Negative</td>
</tr>
<tr>
<td>setns D</td>
<td></td>
<td>Nonnegative</td>
</tr>
<tr>
<td>setg D</td>
<td>setnle</td>
<td>Greater (signed &gt;)</td>
</tr>
<tr>
<td>setge D</td>
<td>setnl</td>
<td>Greater or equal (signed &gt;=)</td>
</tr>
<tr>
<td>setl D</td>
<td>setnle</td>
<td>Less (signed &lt;)</td>
</tr>
<tr>
<td>setle D</td>
<td>setng</td>
<td>Less or equal (signed &lt;=)</td>
</tr>
<tr>
<td>seta D</td>
<td>setnbe</td>
<td>Above (unsigned &gt;)</td>
</tr>
<tr>
<td>setae D</td>
<td>setnb</td>
<td>Above or equal (unsigned &gt;=)</td>
</tr>
<tr>
<td>setb D</td>
<td>setnae</td>
<td>Below (unsigned &lt;)</td>
</tr>
<tr>
<td>setbe D</td>
<td>setna</td>
<td>Below or equal (unsigned &lt;=)</td>
</tr>
</tbody>
</table>
# 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;**

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

```assembly
cmp   %edi,%esi
mov   %edi, %eax
cmovge %esi, %eax
retq
```
# cmov: Conditional move

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Synonym</th>
<th>Move Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmove S,R</td>
<td>cmovz</td>
<td>Equal / zero (ZF = 1)</td>
</tr>
<tr>
<td>cmovne S,R</td>
<td>cmovnzn</td>
<td>Not equal / not zero (ZF = 0)</td>
</tr>
<tr>
<td>cmovs S,R</td>
<td></td>
<td>Negative (SF = 1)</td>
</tr>
<tr>
<td>cmovns S,R</td>
<td></td>
<td>Nonnegative (SF = 0)</td>
</tr>
<tr>
<td>cmoveg S,R</td>
<td>cmovnle</td>
<td>Greater (signed &gt;) (SF = 0 and SF = OF)</td>
</tr>
<tr>
<td>cmovege S,R</td>
<td>cmovnl</td>
<td>Greater or equal (signed &gt;=) (SF = OF)</td>
</tr>
<tr>
<td>cmovl S,R</td>
<td>cmovng</td>
<td>Less (signed &lt;) (SF ! OF)</td>
</tr>
<tr>
<td>cmovele S,R</td>
<td>cmovnbe</td>
<td>Less or equal (signed &lt;=) (ZF = 1 or SF ! OF)</td>
</tr>
<tr>
<td>cmova S,R</td>
<td>cmovnbe</td>
<td>Above (unsigned &gt;) (CF = 0 and ZF = 0)</td>
</tr>
<tr>
<td>cmovae S,R</td>
<td>cmovnb</td>
<td>Above or equal (unsigned &gt;=) (CF = 0)</td>
</tr>
<tr>
<td>cmovb S,R</td>
<td>cmovnae</td>
<td>Below (unsigned &lt;) (CF = 1)</td>
</tr>
<tr>
<td>cmovbe S,R</td>
<td>cmovna</td>
<td>Below or equal (unsigned &lt;=) (CF = 1 or ZF = 1)</td>
</tr>
</tbody>
</table>
int signed_division(int x) {
    return x / 4;
}

signed_division:
leal 3(%rdi), %eax  \hspace{1cm} \text{Put } x + 3 \text{ into } %eax
	\begin{align*}
    &\text{testl } %edi, %edi \quad \text{Check the sign of } x \\
    &\text{cmovns } %edi, %eax \quad \text{If } x \text{ is positive, put } x \text{ into } %eax \\
    &\text{sarl } $2, %eax \quad \text{Divide } %eax \text{ by } 4 \\
    &\text{ret}
\end{align*}