Assembly Programming III
CSE 351 Spring 2017

Guest Lecturer:
Justin Hsia

Instructor:
Ruth Anderson

Teaching Assistants:
Dylan Johnson
Kevin Bi
Linxing Preston Jiang
Cody Ohlsen
Yufang Sun
Joshua Curtis
Administrivia

- Lab 1 due TONIGHT, Friday (4/14)
  - Remember, you have *late days* available if needed.
- Homework 2 due next Wednesday (4/19)
- Lab 2 (x86-64) released soon!
  - Learn to read x86-64 assembly and use GDB
Address Computation Instruction

- leaq src, dst
  - “lea” stands for *load effective address*
  - src is address expression (any of the formats we’ve seen)
  - dst is a register
  - Sets dst to the *address* computed by the src expression (does not go to memory! – it just does math)
  - Example: leaq (%rdx, %rcx, 4), %rax

- Uses:
  - Computing addresses without a memory reference
    - *e.g.* translation of \( p = &x[i] \);
  - Computing arithmetic expressions of the form \( x + k \times i + d \)
    - Though \( k \) can only be 1, 2, 4, or 8
Example: `lea` vs. `mov`

<table>
<thead>
<tr>
<th>Registers</th>
<th>Memory</th>
<th>Word Address</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>%rax</code></td>
<td>0x110</td>
<td>0x120</td>
</tr>
<tr>
<td><code>%rbx</code></td>
<td>0x8</td>
<td>0xF</td>
</tr>
<tr>
<td><code>%rcx</code></td>
<td>0x4</td>
<td>0x118</td>
</tr>
<tr>
<td><code>%rdx</code></td>
<td>0x100</td>
<td>0x110</td>
</tr>
<tr>
<td><code>%rdi</code></td>
<td>0x100</td>
<td>0x108</td>
</tr>
<tr>
<td><code>%rsi</code></td>
<td>0x1</td>
<td>0x100</td>
</tr>
</tbody>
</table>

`leaq (%rdx, %rcx, 4), %rax`

`movq (%rdx, %rcx, 4), %rbx`

`leaq (%rdx), %rdi` ↔ `movq (%rdx), %rsi`
Arithmetic Example

```c
long arith(long x, long y, long z) {
    long t1 = x + y;
    long t2 = z + t1;
    long t3 = x + 4;
    long t4 = y * 48;
    long t5 = t3 + t4;
    long rval = t2 * t5;
    return rval;
}
```

### Register Use(s)
- `%rdi` 1\textsuperscript{st} argument (x)
- `%rsi` 2\textsuperscript{nd} argument (y)
- `%rdx` 3\textsuperscript{rd} argument (z)

### Interesting Instructions
- `leaq`: “address” computation
- `salq`: shift
- `imulq`: multiplication
  - Only used once!
Arithmetic Example

```c
long arith(long x, long y, long z)
{
    long t1 = x + y;
    long t2 = z + t1;
    long t3 = x + 4;
    long t4 = y * 48;
    long t5 = t3 + t4;
    long rval = t2 * t5;
    return rval;
}
```

```
%rdi x
%rsi y
%rdx z, t4
%rax t1, t2, rval
%rcx t5

arith:
  leaq (%rdi,%rsi), %rax  # rax/t1  = x + y
  addq %rdx, %rax         # rax/t2  = t1 + z
  leaq (%rsi,%rsi,2), %rdx # rdx     = 3 * y
  salq $4, %rdx           # rdx/t4  = (3*y) * 16
  leaq 4(%rdi,%rdx), %rcx # rcx/t5  = x + t4 + 4
  imulq %rcx, %rax        # rax/rval = t5 * t2
  ret
```
x86 Control Flow

- Condition codes
- Conditional and unconditional branches
- Loops
- Switches
Control Flow

```c
long max(long x, long y) {
    long max;
    if (x > y) {
        max = x;
    } else {
        max = y;
    }
    return max;
}
```

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>1st argument (x)</td>
</tr>
<tr>
<td>%rsi</td>
<td>2nd argument (y)</td>
</tr>
<tr>
<td>%rax</td>
<td>return value</td>
</tr>
</tbody>
</table>

max:

```assembly
movq %rdi, %rax
movq %rsi, %rax
ret
```
Control Flow

long max(long x, long y) {
    long max;
    if (x > y) {
        max = x;
    } else {
        max = y;
    }
    return max;
}

max:  if x <= y then jump to else
      movq %rdi, %rax
      jump to done

else:  movq %rsi, %rax
done:  ret

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>1st argument (x)</td>
</tr>
<tr>
<td>%rsi</td>
<td>2nd argument (y)</td>
</tr>
<tr>
<td>%rax</td>
<td>return value</td>
</tr>
</tbody>
</table>
Conditionals and Control Flow

- Conditional branch/jump
  - Jump to somewhere else if some condition is true, otherwise execute next instruction

- Unconditional branch/jump
  - Always jump when you get to this instruction

- Together, they can implement most control flow constructs in high-level languages:
  - if (condition) then {...} else {...}
  - while (condition) {...}
  - do {...} while (condition)
  - for (initialization; condition; iterative) {...}
  - switch {...}
x86 Control Flow

- Condition codes
- Conditional and unconditional branches
- Loops
- Switches
Processor State (x86-64, partial)

- Information about currently executing program
  - Temporary data (\%rax, ...)
  - Location of runtime stack (\%rsp)
  - Location of current code control point (\%rip, ...)
  - Status of recent tests (CF, ZF, SF, OF)
    - Single bit registers:

### Registers

| \%rax | \%r8 |
| \%rbx | \%r9 |
| \%rcx | \%r10 |
| \%rdx | \%r11 |
| \%r8 | \%r12 |
| \%r9 | \%r13 |
| \%r10 | \%r14 |
| \%r11 | \%r15 |

#### Current Top of the Stack

- \%rsp
- \%rbp

#### Program Counter

- (instruction pointer)

#### Condition Codes

- CF, ZF, SF, OF

---

**Notes:**

- Single bit registers:
Condition Codes (Implicit Setting)

- **Implicitly** set by **arithmetic** operations
  - (think of it as side effects)
  - **Example**: `addq src, dst ↔ r = d+s; result dst + src`

- **CF=1** if carry out from MSB (unsigned overflow)
- **ZF=1** if `r==0`
- **SF=1** if `r<0` (assuming signed, actually just if MSB is 1)
- **OF=1** if two’s complement (signed) overflow
  
  \[(s>0 \land d>0 \land r<0) \lor (s<0 \land d<0 \land r>=0)\]

- **Not set by lea instruction (beware!)**
Condition Codes (Explicit Setting: Compare)

- *Explicitly* set by **Compare** instruction
  - `cmpq src1, src2`
  - `cmpq a, b` sets flags based on \(b - a\), but doesn’t store

- **CF=1** if carry out from MSB (used for unsigned comparison)
- **ZF=1** if \(a==b\)
- **SF=1** if \((b - a) < 0\) (signed)
- **OF=1** if two’s complement (signed) overflow
  \[ (b>0 \land a<0 \land (b-a)<0) \lor \]
  \[ (b<0 \land a>0 \land (b-a)>0) \]

<table>
<thead>
<tr>
<th>CF</th>
<th>Carry Flag</th>
<th>ZF</th>
<th>Zero Flag</th>
<th>SF</th>
<th>Sign Flag</th>
<th>OF</th>
<th>Overflow Flag</th>
</tr>
</thead>
</table>
Condition Codes (Explicit Setting: Test)

- Explicitly set by Test instruction
  - `testq` src2, src1
  - `testq` a, b sets flags based on b&a, but doesn’t store
    - Useful to have one of the operands be a mask

- Can’t have carry out (CF) or overflow (OF)
  - ZF=1 if a&b==0
  - SF=1 if a&b<0 (signed)

- Example: `testq %rax, %rax`
  - Tells you if (+), 0, or (−) based on ZF and SF

<table>
<thead>
<tr>
<th>CF</th>
<th>Carry Flag</th>
<th>ZF</th>
<th>Zero Flag</th>
<th>SF</th>
<th>Sign Flag</th>
<th>OF</th>
<th>Overflow Flag</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Using Condition Codes: Jumping

- j*Instructions
  - Jumps to target (an address) based on condition codes

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>jmp target</td>
<td>1</td>
<td>Unconditional</td>
</tr>
<tr>
<td>je target</td>
<td>ZF</td>
<td>Equal / Zero</td>
</tr>
<tr>
<td>jne target</td>
<td>~ZF</td>
<td>Not Equal / Not Zero</td>
</tr>
<tr>
<td>js target</td>
<td>SF</td>
<td>Negative</td>
</tr>
<tr>
<td>jns target</td>
<td>~SF</td>
<td>Nonnegative</td>
</tr>
<tr>
<td>jg target</td>
<td>~(SF^OF) &amp; ~ZF</td>
<td>Greater (Signed)</td>
</tr>
<tr>
<td>jge target</td>
<td>~(SF^OF)</td>
<td>Greater or Equal (Signed)</td>
</tr>
<tr>
<td>jl target</td>
<td>(SF^OF)</td>
<td>Less (Signed)</td>
</tr>
<tr>
<td>jle target</td>
<td>(SF^OF)</td>
<td>Less or Equal (Signed)</td>
</tr>
<tr>
<td>ja target</td>
<td>~CF &amp; ~ZF</td>
<td>Above (unsigned “&gt;“)</td>
</tr>
<tr>
<td>jb target</td>
<td>CF</td>
<td>Below (unsigned “&lt;“)</td>
</tr>
</tbody>
</table>
Using Condition Codes: Setting

- **set* Instructions**
  - Set low-order byte of `dst` to 0 or 1 based on condition codes
  - Does not alter remaining 7 bytes

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sete <code>dst</code></td>
<td>ZF</td>
<td>Equal / Zero</td>
</tr>
<tr>
<td>setne <code>dst</code></td>
<td>~ZF</td>
<td>Not Equal / Not Zero</td>
</tr>
<tr>
<td>sets <code>dst</code></td>
<td>SF</td>
<td>Negative</td>
</tr>
<tr>
<td>setns <code>dst</code></td>
<td>~SF</td>
<td>Nonnegative</td>
</tr>
<tr>
<td>setg <code>dst</code></td>
<td>~(SF^OF) &amp;~ZF</td>
<td>Greater (Signed)</td>
</tr>
<tr>
<td>setge <code>dst</code></td>
<td>~(SF^OF)</td>
<td>Greater or Equal (Signed)</td>
</tr>
<tr>
<td>setl <code>dst</code></td>
<td>(SF^OF)</td>
<td>Less (Signed)</td>
</tr>
<tr>
<td>setle <code>dst</code></td>
<td>(SF^OF)</td>
<td>Less or Equal (Signed)</td>
</tr>
<tr>
<td>seta <code>dst</code></td>
<td>~CF&amp;~ZF</td>
<td>Above (unsigned “&gt;”)</td>
</tr>
<tr>
<td>setb <code>dst</code></td>
<td>CF</td>
<td>Below (unsigned “&lt;”)</td>
</tr>
</tbody>
</table>
Reminder: x86-64 Integer Registers

- Accessing the low-order byte:

<table>
<thead>
<tr>
<th>%rax</th>
<th>%al</th>
<th>%r8</th>
<th>%r8b</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rbx</td>
<td>%bl</td>
<td>%r9</td>
<td>%r9b</td>
</tr>
<tr>
<td>%rcx</td>
<td>%cl</td>
<td>%r10</td>
<td>%r10b</td>
</tr>
<tr>
<td>%rdx</td>
<td>%dl</td>
<td>%r11</td>
<td>%r11b</td>
</tr>
<tr>
<td>%rsi</td>
<td>%sil</td>
<td>%r12</td>
<td>%r12b</td>
</tr>
<tr>
<td>%rdi</td>
<td>%dil</td>
<td>%r13</td>
<td>%r13b</td>
</tr>
<tr>
<td>%rsp</td>
<td>%spl</td>
<td>%r14</td>
<td>%r14b</td>
</tr>
<tr>
<td>%rbp</td>
<td>%bpl</td>
<td>%r15</td>
<td>%r15b</td>
</tr>
</tbody>
</table>
Reading Condition Codes

- **set* Instructions**
  - Set a low-order byte to 0 or 1 based on condition codes
  - Operand is byte register (e.g. `al`, `dl`) or a byte in memory
  - Do not alter remaining bytes in register
    - Typically use `movzbl` (zero-extended `mov`) to finish job

```c
int gt(long x, long y) {
    return x > y;
}
```

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>1st argument (x)</td>
</tr>
<tr>
<td>%rsi</td>
<td>2nd argument (y)</td>
</tr>
<tr>
<td>%rax</td>
<td>return value</td>
</tr>
</tbody>
</table>
Reading Condition Codes

- **set* Instructions**
  - Set a low-order byte to 0 or 1 based on condition codes
  - Operand is byte register (e.g. `al`, `dl`) or a byte in memory
  - Do not alter remaining bytes in register
    - Typically use `movzbl` (zero-extended `mov`) to finish job

```c
int gt(long x, long y) {
    return x > y;
}
```

```assembly
cmpq %rsi, %rdi  # Compare x:y
setg %al         # Set when >
movzbl %al, %eax # Zero rest of %rax
ret
```
Aside: movz and movs

- Copy from a smaller source value to a larger destination
- Source can be memory or register; Destination must be a register
- Fill remaining bits of dest with zero (movz) or sign bit (movs)

movz SD / movs SD:
- S – size of source (b = 1 byte, w = 2)
- D – size of dest (w = 2 bytes, l = 4, q = 8)

Example:
- movz bq %al, %rbx

\[
\begin{array}{c}
\text{movz bq } \%al, \%rbx \\
\text{0x?? 0x?? 0x?? 0x?? 0x?? 0x?? 0x?? 0xFF} \leftarrow \%rax \\
\text{0x00 0x00 0x00 0x00 0x00 0x00 0x00 0xFF} \leftarrow \%rbx
\end{array}
\]
Aside: movz and movs

\[
\text{movz} \_ \_ \ src, \ regDest \quad \text{Move with zero extension}
\]
\[
\text{movs} \_ \_ \ src, \ regDest \quad \text{Move with sign extension}
\]

- Copy from a \textit{smaller} source value to a \textit{larger} destination
- Source can be memory or register; Destination \textit{must} be a register
- Fill remaining bits of dest with \textbf{zero} (\textit{movz}) or \textbf{sign bit} (\textit{movs})

\[
\text{movz} SD / \text{movs} SD:
\]
\[
S – \text{size of source (} b = 1 \text{ byte, } w = 2) \\
D – \text{size of dest (} w = 2 \text{ bytes, } l = 4, \ q = 8)
\]

Example:
\[
\text{movsbl} (\%rax), \%ebx
\]

Copy 1 byte from memory into 8-byte register & sign extend it

Note: In x86-64, \textit{any instruction} that generates a 32-bit (long word) value for a register also sets the high-order portion of the register to 0. Good example on p. 184 in the textbook.
Choosing instructions for conditionals

- All arithmetic instructions set condition flags based on result of operation (op)
  - Conditionals are comparisons against 0
- Come in instruction pairs

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Condition</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>addq 5, (p)</code></td>
<td><code>je: *p+5 == 0</code></td>
<td><code>je: b|a == 0</code></td>
</tr>
<tr>
<td></td>
<td><code>jne: *p+5 != 0</code></td>
<td><code>jne: b|a != 0</code></td>
</tr>
<tr>
<td></td>
<td><code>jg: *p+5 &gt; 0</code></td>
<td><code>jg: b|a &gt; 0</code></td>
</tr>
<tr>
<td></td>
<td><code>jl: *p+5 &lt; 0</code></td>
<td><code>jl: b|a &lt; 0</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Condition</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>orq a, b</code></td>
<td><code>je: b|a == 0</code></td>
<td><code>je: b|a == 0</code></td>
</tr>
<tr>
<td></td>
<td><code>jne: b|a != 0</code></td>
<td><code>jne: b|a != 0</code></td>
</tr>
<tr>
<td></td>
<td><code>jg: b|a &gt; 0</code></td>
<td><code>jg: b|a &gt; 0</code></td>
</tr>
<tr>
<td></td>
<td><code>jl: b|a &lt; 0</code></td>
<td><code>jl: b|a &lt; 0</code></td>
</tr>
</tbody>
</table>

- `addq 5, (p)`
- `orq a, b`

<table>
<thead>
<tr>
<th>Condition</th>
<th>Instruction</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>je</code></td>
<td>“Equal”</td>
<td><code>d (op) s == 0</code></td>
</tr>
<tr>
<td><code>jne</code></td>
<td>“Not equal”</td>
<td><code>d (op) s != 0</code></td>
</tr>
<tr>
<td><code>js</code></td>
<td>“Sign” (negative)</td>
<td><code>d (op) s &lt; 0</code></td>
</tr>
<tr>
<td><code>jns</code></td>
<td>(non-negative)</td>
<td><code>d (op) s &gt;= 0</code></td>
</tr>
<tr>
<td><code>jg</code></td>
<td>“Greater”</td>
<td><code>d (op) s &gt; 0</code></td>
</tr>
<tr>
<td><code>jge</code></td>
<td>“Greater or equal”</td>
<td><code>d (op) s &gt;= 0</code></td>
</tr>
<tr>
<td><code>jl</code></td>
<td>“Less”</td>
<td><code>d (op) s &lt; 0</code></td>
</tr>
<tr>
<td><code>jle</code></td>
<td>“Less or equal”</td>
<td><code>d (op) s &lt;= 0</code></td>
</tr>
<tr>
<td><code>ja</code></td>
<td>“Above” (unsigned &gt;)</td>
<td><code>d (op) s &gt; 0U</code></td>
</tr>
<tr>
<td><code>jb</code></td>
<td>“Below” (unsigned &lt;)</td>
<td><code>d (op) s &lt; 0U</code></td>
</tr>
</tbody>
</table>
Choosing instructions for conditionals

- Reminder: `cmp` is like `sub`, `test` is like `and`
  - Result is not stored anywhere

<table>
<thead>
<tr>
<th></th>
<th>Sub $\text{cmp a,b}$</th>
<th>And $\text{test a,b}$</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>je</code></td>
<td>“Equal”</td>
<td>$b == a$</td>
</tr>
<tr>
<td><code>jne</code></td>
<td>“Not equal”</td>
<td>$b != a$</td>
</tr>
<tr>
<td><code>js</code></td>
<td>“Sign” (negative)</td>
<td>$b - a &lt; 0$</td>
</tr>
<tr>
<td><code>jns</code></td>
<td>(non-negative)</td>
<td>$b - a \geq 0$</td>
</tr>
<tr>
<td><code>jg</code></td>
<td>“Greater”</td>
<td>$b &gt; a$</td>
</tr>
<tr>
<td><code>jge</code></td>
<td>“Greater or equal”</td>
<td>$b \geq a$</td>
</tr>
<tr>
<td><code>jl</code></td>
<td>“Less”</td>
<td>$b &lt; a$</td>
</tr>
<tr>
<td><code>jle</code></td>
<td>“Less or equal”</td>
<td>$b \leq a$</td>
</tr>
<tr>
<td><code>ja</code></td>
<td>“Above” (unsigned $&gt;$)</td>
<td>$b &gt; a$</td>
</tr>
<tr>
<td><code>jb</code></td>
<td>“Below” (unsigned $&lt;$)</td>
<td>$b &lt; a$</td>
</tr>
</tbody>
</table>

```plaintext
\text{cmpq 5, (p)}
je: *p == 5
jne: *p != 5
jg: *p > 5
jl: *p < 5

\text{testq a, a}
je: a == 0
jne: a != 0
jg: a > 0
jl: a < 0

\text{testb a, 0x1}
je: a_{\text{LSB}} == 0
jne: a_{\text{LSB}} == 1
```
Choosing instructions for conditionals

<table>
<thead>
<tr>
<th>Register Use(s)</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>argument x</td>
</tr>
<tr>
<td>%rsi</td>
<td>argument y</td>
</tr>
<tr>
<td>%rax</td>
<td>return value</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction</th>
<th>cmp a,b</th>
<th>test a,b</th>
</tr>
</thead>
<tbody>
<tr>
<td>je &quot;Equal&quot;</td>
<td>b == a</td>
<td>b&amp;a == 0</td>
</tr>
<tr>
<td>jne &quot;Not equal&quot;</td>
<td>b != a</td>
<td>b&amp;a != 0</td>
</tr>
<tr>
<td>js &quot;Sign&quot; (negative)</td>
<td>b-a &lt; 0</td>
<td>b&amp;a &lt; 0</td>
</tr>
<tr>
<td>jns (non-negative)</td>
<td>b-a &gt;=0</td>
<td>b&amp;a &gt;= 0</td>
</tr>
<tr>
<td>jg &quot;Greater&quot;</td>
<td>b &gt; a</td>
<td>b&amp;a &gt; 0</td>
</tr>
<tr>
<td>jge &quot;Greater or equal&quot;</td>
<td>b &gt;= a</td>
<td>b&amp;a &gt;= 0</td>
</tr>
<tr>
<td>jl &quot;Less&quot;</td>
<td>b &lt; a</td>
<td>b&amp;a &lt; 0</td>
</tr>
<tr>
<td>jle &quot;Less or equal&quot;</td>
<td>b &lt;= a</td>
<td>b&amp;a &lt;= 0</td>
</tr>
<tr>
<td>ja &quot;Above&quot; (unsigned &gt;)</td>
<td>b &gt; a</td>
<td>b&amp;a &gt; 0U</td>
</tr>
<tr>
<td>jb &quot;Below&quot; (unsigned &lt;)</td>
<td>b &lt; a</td>
<td>b&amp;a &lt; 0U</td>
</tr>
</tbody>
</table>

if (x < 3) {
    return 1;
} else {
    return 2;
}

cmpq $3, %rdi
jge T2
T1: # x < 3: (if)
    movq $1, %rax
    ret
T2: # !(x < 3): (else)
    movq $2, %rax
    ret
Question

A. `cmpq %rsi, %rdi`  
   `jle .L4`

B. `cmpq %rsi, %rdi`  
   `jg .L4`

C. `testq %rsi, %rdi`  
   `jle .L4`

D. `testq %rsi, %rdi`  
   `jg .L4`

E. We’re lost…

---

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

### Register Use(s)

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>1st argument (x)</td>
</tr>
<tr>
<td>%rsi</td>
<td>2nd argument (y)</td>
</tr>
<tr>
<td>%rax</td>
<td>return value</td>
</tr>
</tbody>
</table>

---

```
absdiff:

___________________________
# x > y:
    movq %rdi, %rax
    subq %rsi, %rax
    ret
___________________________
# x <= y:
    movq %rsi, %rax
    subq %rdi, %rax
    ret
```
Choosing instructions for conditionals

<table>
<thead>
<tr>
<th></th>
<th>cmp a,b</th>
<th>test a,b</th>
</tr>
</thead>
<tbody>
<tr>
<td>je</td>
<td>&quot;Equal&quot;</td>
<td>b ( \leq 3 )</td>
</tr>
<tr>
<td>jne</td>
<td>&quot;Not equal&quot;</td>
<td>b ( \neq a )</td>
</tr>
<tr>
<td>js</td>
<td>&quot;Sign&quot; (negative)</td>
<td>b ( \leq 0 )</td>
</tr>
<tr>
<td>jns</td>
<td>(non-negative)</td>
<td>b ( \geq 0 )</td>
</tr>
<tr>
<td>jg</td>
<td>&quot;Greater&quot;</td>
<td>b ( &gt; a )</td>
</tr>
<tr>
<td>jge</td>
<td>&quot;Greater or equal&quot;</td>
<td>b ( \geq a )</td>
</tr>
<tr>
<td>jl</td>
<td>&quot;Less&quot;</td>
<td>b ( \leq 3 )</td>
</tr>
<tr>
<td>jle</td>
<td>&quot;Less or equal&quot;</td>
<td>b ( \leq a )</td>
</tr>
<tr>
<td>ja</td>
<td>&quot;Above&quot; (unsigned &gt;)</td>
<td>b ( &gt; a )</td>
</tr>
<tr>
<td>jb</td>
<td>&quot;Below&quot; (unsigned &lt;)</td>
<td>b ( &lt; a )</td>
</tr>
</tbody>
</table>

if \( (x < 3 \&\& x == y) \) {
    return 1;
} else {
    do this if either \%al or \%bl are False
    return 2;
}

1. \texttt{cmpq} $3$, \%rdi \{\%al = (x<3)\}
   \texttt{setl} \%al
2. \texttt{cmpq} \%rsi, \%rdi \{\%bl=(x==y)\}
   \texttt{sete} \%bl
3. \texttt{testb} \%al, \%bl
   \texttt{je} T2 \iff (\%al \&\& \%bl) = 0

T1: \# x < 3 \&\& x == y:
    \texttt{movq} $1$, \%rax
    \texttt{ret}
T2: \# else
    \texttt{movq} $2$, \%rax
    \texttt{ret}
Choosing instructions for conditionals

<table>
<thead>
<tr>
<th></th>
<th>cmp a,b</th>
<th>test a,b</th>
</tr>
</thead>
<tbody>
<tr>
<td>je “Equal”</td>
<td>b == a</td>
<td>b&amp;a == 0</td>
</tr>
<tr>
<td>jne “Not equal”</td>
<td>b != a</td>
<td>b&amp;a != 0</td>
</tr>
<tr>
<td>js “Sign” (negative)</td>
<td>b-a &lt; 0</td>
<td>b&amp;a &lt; 0</td>
</tr>
<tr>
<td>jns (non-negative)</td>
<td>b-a &gt;=0</td>
<td>b&amp;a &gt;= 0</td>
</tr>
<tr>
<td>jg “Greater”</td>
<td>b &gt; a</td>
<td>b&amp;a &gt; 0</td>
</tr>
<tr>
<td>jge “Greater or equal”</td>
<td>b &gt;= a</td>
<td>b&amp;a &gt;= 0</td>
</tr>
<tr>
<td>jl “Less”</td>
<td>b &lt; a</td>
<td>b&amp;a &lt; 0</td>
</tr>
<tr>
<td>jle “Less or equal”</td>
<td>b &lt;= a</td>
<td>b&amp;a &lt;= 0</td>
</tr>
<tr>
<td>ja “Above” (unsigned &gt;)</td>
<td>b &gt; a</td>
<td>b&amp;a &gt; 0U</td>
</tr>
<tr>
<td>jb “Below” (unsigned &lt;)</td>
<td>b &lt; a</td>
<td>b&amp;a &lt; 0U</td>
</tr>
</tbody>
</table>

if (x < 3 && x == y) {
    return 1;
} else {
    return 2;
}

cmpq $3, %rdi
setl %al
cmpq %rsi, %rdi
sete %bl
testb %al, %bl
je T2
T1: # x < 3 && x == y:
    movq $1, %rax
    ret
T2: # else
    movq $2, %rax
    ret

https://godbolt.org/g/Ovh3jN
Summary

- Control flow in x86 determined by status of Condition Codes
  - Showed Carry, Zero, Sign, and Overflow, though others exist
  - Set flags with arithmetic instructions (implicit) or Compare and Test (explicit)
  - Set instructions read out flag values
  - Jump instructions use flag values to determine next instruction to execute
Bonus content (nonessential). Does contain examples.

- Conditional Operator with Jumps
- Conditional Move
Conditional Operator with Jumps

**C Code**

\[
\text{val} = \text{Test} \; ? \; \text{Then-Expr} \; : \; \text{Else-Expr};
\]

**Example:**

\[
\text{result} = x > y \; ? \; x - y \; : \; y - x;
\]

**Goto Version**

\[

test = \neg \text{Test};
\]

\[
\text{if (ntest) goto Else;}
\]

\[
\text{val} = \text{Then-Expr};
\]

\[
\text{goto Done;}
\]

**Else:**

\[
\text{val} = \text{Else-Expr};
\]

**Done:**

\[
\ldots
\]

- Ternary operator \( ? : \)
- \( \text{Test} \) is expression returning integer
  - \( = 0 \) interpreted as false
  - \( \neq 0 \) interpreted as true
- Create separate code regions for then & else expressions
- Execute appropriate one

**Bonus Content (nonessential)**
Conditional Move

- Conditional Move Instructions: `cmovC src, dst`
  - Move value from src to dst if condition C holds
  - if (Test) Dest ← Src
  - GCC tries to use them (but only when known to be safe)

- Why is this useful?
  - Branches are very disruptive to instruction flow through pipelines
  - Conditional moves do not require control transfer

```asm
long absdiff(long x, long y)
{
    return x>y ? x-y : y-x;
}
```

```c
long absdiff(long x, long y)
{
    return x>y ? x-y : y-x;
}
```
Using Conditional Moves

- Conditional Move Instructions
  - \texttt{cmovC src, dest}
  - Move value from src to dest if condition \texttt{C} holds
  - Instruction supports:
    - \texttt{if (Test) Dest \leftarrow Src}
  - Supported in post-1995 x86 processors
  - GCC tries to use them
    - But, only when known to be safe

- Why is this useful?
  - Branches are very disruptive to instruction flow through pipelines
  - Conditional moves do not require control transfer

\textbf{C Code}

```
val = Test
? Then_Expr
: Else_Expr;
```

\textbf{“Goto” Version}

```
result = Then_Expr;
else_val = Else_Expr;
nt = ! Test;
if (nt) result = else_val;
return result;
```
Conditional Move Example

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

### Register Use(s)

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>Argument (x)</td>
</tr>
<tr>
<td>%rsi</td>
<td>Argument (y)</td>
</tr>
<tr>
<td>%rax</td>
<td>Return value</td>
</tr>
</tbody>
</table>

**absdiff:**

```assembly
movq %rdi, %rax  # x
subq %rsi, %rax  # result = x-y
movq %rsi, %rdx
subq %rdi, %rdx  # else_val = y-x
cmpq %rsi, %rdi  # x:y
cmovle %rdx, %rax # if <=, result = else_val
ret
```
Bad Cases for Conditional Move

Expensive Computations

\[
\text{val} = \text{Test}(x) \ ? \ \text{Hard1}(x) : \text{Hard2}(x);
\]
- Both values get computed
- Only makes sense when computations are very simple

Risky Computations

\[
\text{val} = p \ ? \ *p : 0;
\]
- Both values get computed
- May have undesirable effects

Computations with side effects

\[
\text{val} = x > 0 \ ? \ x*=7 : x+=3;
\]
- Both values get computed
- Must be side-effect free