Administrivia

- Lab 2 (x86-64) released today
  - Learn to read x86-64 assembly and use GDB
- Homework 2 due next week
- Midterm is in two weeks
  - More information soon
Control Flow

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

### 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>

```assembly
max:  
    ??? 
    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;
}

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

Conditional jump: \( \text{if } x \leq y \text{ then jump to else} \)

Unconditional jump: \( \text{jump to done} \)

\[
\text{max:}
\]

\[
\text{if } x \leq y \text{ then jump to else}
\]

\[
\text{movq } %rdi, %rax
\]

\[
\text{else:}
\]

\[
\text{movq } %rsi, %rax
\]

\[
\text{done:}
\]

\[
\text{ret}
\]
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 |
| %rsi | %r12 |
| %rdi | %r13 |
| %rsp | %r14 |
| %rbp | %r15 |

### Condition Codes

- CF
- ZF
- SF
- OF

### Program Counter

(Instruction Pointer)

- **current top of the Stack**

\%rip
Condition Codes (Implicit Setting)

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

- **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
  \[(s>0 \&\& d>0 \&\& r<0) || (s<0 \&\& d<0 \&\& r>=0)\]

- *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
    - $\textbf{CF}=1$ if carry out from MSB (used for unsigned comparison)
    - $\textbf{ZF}=1$ if $a==b$
    - $\textbf{SF}=1$ if $(b-a)<0$ (signed)
    - $\textbf{OF}=1$ if two’s complement (signed) overflow
      - $(a>0 \&\& b<0 \&\& (b-a)>0) ||$
      - $(a<0 \&\& b>0 \&\& (b-a)<0)$

![Carry Flag Zero Flag Sign Flag Overflow Flag](image)
Condition Codes (Explicit Setting: Test)

- **Explicitly** set by **Test** instruction
  - `testq src2, src1`
  - `testq a, b` sets flags based on `a & b`, 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**
Using Condition Codes: Jumping

- \( \text{j} * \) Instructions
  - Jumps to \textit{target} (an address) based on condition codes

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{jmp target}</td>
<td>1</td>
<td>Unconditional</td>
</tr>
<tr>
<td>\texttt{je target}</td>
<td>ZF</td>
<td>Equal / Zero</td>
</tr>
<tr>
<td>\texttt{jne target}</td>
<td>(~ZF)</td>
<td>Not Equal / Not Zero</td>
</tr>
<tr>
<td>\texttt{js target}</td>
<td>SF</td>
<td>Negative</td>
</tr>
<tr>
<td>\texttt{jns target}</td>
<td>(~SF)</td>
<td>Nonnegative</td>
</tr>
<tr>
<td>\texttt{jg target}</td>
<td>(<del>(SF\lor OF)\lor</del>ZF)</td>
<td>Greater (Signed)</td>
</tr>
<tr>
<td>\texttt{jge target}</td>
<td>(~(SF\lor OF))</td>
<td>Greater or Equal (Signed)</td>
</tr>
<tr>
<td>\texttt{jl target}</td>
<td>((SF\lor OF))</td>
<td>Less (Signed)</td>
</tr>
<tr>
<td>\texttt{jle target}</td>
<td>((SF\lor OF)\lor ZF)</td>
<td>Less or Equal (Signed)</td>
</tr>
<tr>
<td>\texttt{ja target}</td>
<td>(<del>CF\lor</del>ZF)</td>
<td>Above (unsigned “&gt;“)</td>
</tr>
<tr>
<td>\texttt{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><code>sete dst</code></td>
<td><code>ZF</code></td>
<td>Equal / Zero</td>
</tr>
<tr>
<td><code>setne dst</code></td>
<td><code>~ZF</code></td>
<td>Not Equal / Not Zero</td>
</tr>
<tr>
<td><code>sets dst</code></td>
<td><code>SF</code></td>
<td>Negative</td>
</tr>
<tr>
<td><code>setns dst</code></td>
<td><code>~SF</code></td>
<td>Nonnegative</td>
</tr>
<tr>
<td><code>setg dst</code></td>
<td><code>~(SF^OF) &amp; ~ZF</code></td>
<td>Greater (Signed)</td>
</tr>
<tr>
<td><code>setge dst</code></td>
<td><code>~(SF^OF)</code></td>
<td>Greater or Equal (Signed)</td>
</tr>
<tr>
<td><code>setl(dst</code></td>
<td><code>(SF^OF)</code></td>
<td>Less (Signed)</td>
</tr>
<tr>
<td><code>setle dst</code></td>
<td>`(SF^OF)</td>
<td>ZF`</td>
</tr>
<tr>
<td><code>seta dst</code></td>
<td><code>~CF &amp; ~ZF</code></td>
<td>Above (unsigned “&gt;”)</td>
</tr>
<tr>
<td><code>setb dst</code></td>
<td><code>CF</code></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;
}
```

```
cmpq %rsi, %rdi   #
setg %al          #
movzbl %al, %eax  #
ret
```
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

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

- 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 \textit{zero} (\textit{movz}) or \textit{sign bit} (\textit{movs})

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

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

```
movz __ src, regDest
movs __ src, regDest
```

- **movz**: Move with zero extension
- **movs**: Move with sign extension

- 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)

\[
\text{movz}_{SD} / \text{movs}_{SD}:
\]

- \( S \) – size of source (\( b = 1 \) byte, \( w = 2 \))
- \( D \) – size of dest (\( w = 2 \) bytes, \( l = 4 \), \( q = 8 \))

**Example:**
```
movsbl (%rax), %ebx
```

Note: In x86-64, *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.

Copy 1 byte from memory into 8-byte register & sign extend it
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>Operation</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>addq 5, (p)</td>
<td>*p+5 == 0</td>
<td><strong>je</strong></td>
</tr>
<tr>
<td>addq 5, (p)</td>
<td>*p+5 != 0</td>
<td><strong>jne</strong></td>
</tr>
<tr>
<td>addq 5, (p)</td>
<td>*p+5 &gt; 0</td>
<td><strong>jg</strong></td>
</tr>
<tr>
<td>addq 5, (p)</td>
<td>*p+5 &lt; 0</td>
<td><strong>jl</strong></td>
</tr>
<tr>
<td>orq a, b</td>
<td>b</td>
<td>a == 0</td>
</tr>
<tr>
<td>orq a, b</td>
<td>b</td>
<td>a != 0</td>
</tr>
<tr>
<td>orq a, b</td>
<td>b</td>
<td>a &gt; 0</td>
</tr>
<tr>
<td>orq a, b</td>
<td>b</td>
<td>a &lt; 0</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>Instruction</th>
<th><code>cmp a,b</code></th>
<th><code>test a,b</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>je</code> &quot;Equal&quot;</td>
<td><code>b == a</code></td>
<td><code>b&amp;a == 0</code></td>
</tr>
<tr>
<td><code>jne&quot;Not equal&quot;</code></td>
<td><code>b != a</code></td>
<td><code>b&amp;a != 0</code></td>
</tr>
<tr>
<td><code>js&quot;Sign&quot; (negative)</code></td>
<td><code>b-a &lt; 0</code></td>
<td><code>b&amp;a &lt; 0</code></td>
</tr>
<tr>
<td><code>jns (non-negative)</code></td>
<td><code>b-a &gt;=0</code></td>
<td><code>b&amp;a &gt;= 0</code></td>
</tr>
<tr>
<td><code>jg&quot;Greater&quot;</code></td>
<td><code>b &gt; a</code></td>
<td><code>b&amp;a &gt; 0</code></td>
</tr>
<tr>
<td><code>jge&quot;Greater or equal&quot;</code></td>
<td><code>b &gt;= a</code></td>
<td><code>b&amp;a &gt;= 0</code></td>
</tr>
<tr>
<td><code>jl&quot;Less&quot;</code></td>
<td><code>b &lt; a</code></td>
<td><code>b&amp;a &lt; 0</code></td>
</tr>
<tr>
<td><code>jle&quot;Less or equal&quot;</code></td>
<td><code>b &lt;= a</code></td>
<td><code>b&amp;a &lt;= 0</code></td>
</tr>
<tr>
<td><code>ja&quot;Above&quot; (unsigned &gt;)</code></td>
<td><code>b &gt; a</code></td>
<td><code>b&amp;a &gt; 0U</code></td>
</tr>
<tr>
<td><code>jb&quot;Below&quot; (unsigned &lt;)</code></td>
<td><code>b &lt; a</code></td>
<td><code>b&amp;a &lt; 0U</code></td>
</tr>
</tbody>
</table>
Choosing instructions for conditionals

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

CSE351, Spring 2018

L09: x86-64 Programming II

Raw code:

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

```assembly
cmpq $3, %rdi
jge T2
T1:  # x < 3:
movq $1, %rax
ret
jne T2
T2:  # !(x < 3):
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…"

---

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

---

```
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>“Equal”</td>
<td>b == a</td>
</tr>
<tr>
<td>jne</td>
<td>“Not equal”</td>
<td>b != a</td>
</tr>
<tr>
<td>js</td>
<td>“Sign” (negative)</td>
<td>b-a &lt; 0</td>
</tr>
<tr>
<td>jns</td>
<td>(non-negative)</td>
<td>b-a &gt;= 0</td>
</tr>
<tr>
<td>jg</td>
<td>“Greater”</td>
<td>b &gt; a</td>
</tr>
<tr>
<td>jge</td>
<td>“Greater or equal”</td>
<td>b &gt;= a</td>
</tr>
<tr>
<td>jl</td>
<td>“Less”</td>
<td>b &lt; a</td>
</tr>
<tr>
<td>jle</td>
<td>“Less or equal”</td>
<td>b &lt;= a</td>
</tr>
<tr>
<td>ja</td>
<td>“Above” (unsigned &gt;)</td>
<td>b &gt; a</td>
</tr>
<tr>
<td>jb</td>
<td>“Below” (unsigned &lt;)</td>
<td>b &lt; a</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
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