x86 Programming II
CSE 351 Autumn 2016

Instructor:
Justin Hsia

Teaching Assistants:
Chris Ma
Hunter Zahn
John Kaltenbach
Kevin Bi
Sachin Mehta
Suraj Bhat
Thomas Neuman
Waylon Huang
Xi Liu
Yufang Sun

http://xkcd.com/1652/
Administtrivia

- Lab 2 released tomorrow
  - Learn to use gdb and look at assembly code
- Homework 1 due on Friday (10/21)
Address Computation Instruction

- **leaq** src, dst
  - "leaq" 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
    
    ```
    leaq (%rdx,%rcx,4), %rax
    movq (%rdx,%rcx,4), %rax  \rightarrow  p = x[i]
    ```

- **Uses:**
  - Computing addresses without a memory reference
    - e.g., translation of \( p = \&x[i] \);
  - Computing arithmetic expressions of the form \( x+k\times i \)
    - 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>%rax</td>
<td>0x400</td>
<td>0x120</td>
</tr>
<tr>
<td>%rbx</td>
<td>0xF</td>
<td>0x118</td>
</tr>
<tr>
<td>%rcx</td>
<td>0x8</td>
<td>0x110</td>
</tr>
<tr>
<td>%rdx</td>
<td>0x100</td>
<td>0x110</td>
</tr>
<tr>
<td>%rdi</td>
<td>0x100</td>
<td>0x108</td>
</tr>
<tr>
<td>%rsi</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

rax | ecx | ax | al

0x100 + 0x4 * 4 + 0 = 0x110
Reg[rb] + Reg[rc] * s + d

0x110 (addr) → 0x100
0x8 (data) → 0x1
Example: *lea vs. mov* (solution)

<table>
<thead>
<tr>
<th>Registers</th>
<th>Memory</th>
<th>Word Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>0x400</td>
<td>0x120</td>
</tr>
<tr>
<td>%rbx</td>
<td>0xF</td>
<td>0x118</td>
</tr>
<tr>
<td>%rcx</td>
<td>0x8</td>
<td>0x110</td>
</tr>
<tr>
<td>%rdx</td>
<td>0x10</td>
<td>0x108</td>
</tr>
<tr>
<td>%rdi</td>
<td>0x100</td>
<td>0x100</td>
</tr>
<tr>
<td>%rsi</td>
<td>0x1</td>
<td></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; // leaq + salq
    long t5 = t3 + t4;
    long rval = t2 * t5;
    return rval;
}
```

### Register Use(s)
- `%rdi` 1st argument ($x$)
- `%rsi` 2nd argument ($y$)
- `%rdx` 3rd 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;
}
```

Register Use(s)

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>x</td>
</tr>
<tr>
<td>%rsi</td>
<td>y</td>
</tr>
<tr>
<td>%rdx</td>
<td>z, t4</td>
</tr>
<tr>
<td>%rax</td>
<td>t1, t2, rval</td>
</tr>
<tr>
<td>%rcx</td>
<td>t5</td>
</tr>
</tbody>
</table>

arith:

```assembly
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>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>
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>

Conditional jump

Unconditional jump

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

Long max(long x, long y)
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 {...}
Jumping

- **j* Instructions**
  - Jumps to *target* (argument – actually just an address)
  - Conditional jump relies on special *condition code registers*

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

<table>
<thead>
<tr>
<th>Registers</th>
<th>%rax</th>
<th>%rbx</th>
<th>%rcx</th>
<th>%rdx</th>
<th>%rsi</th>
<th>%rdi</th>
<th>%rsp</th>
<th>%rbp</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>%r8</td>
<td>%r9</td>
<td>%r10</td>
<td>%r11</td>
<td>%r12</td>
<td>%r13</td>
<td>%r14</td>
<td>%r15</td>
</tr>
</tbody>
</table>

- Current top of the Stack
- Program Counter (instruction pointer)
- Condition Codes:
  - Carry (CF)
  - Zero (ZF)
  - Sign (SF)
  - Overflow (OF)
Condition Codes (Implicit Setting)

- *Implicitly* set by **arithmetic** operations
  - (think of it as side effects)
  - **Example**: `addq src, dst ← t = a+b`

- **CF=1** if carry out from MSB (unsigned overflow)
- **ZF=1** if \( t=0 \)
- **SF=1** if \( t<0 \) (assuming signed, actually just if MSB is 1)
- **OF=1** if two’s complement (signed) overflow
  \[ (a>0 \land b>0 \land t<0) \lor (a<0 \land b<0 \land t>=0) \]

- *Not set by lea instruction (beware!)*

- **CF** Carry Flag  **ZF** Zero Flag  **SF** Sign Flag  **OF** Overflow Flag
Condition Codes (Explicit Setting: Compare)

- Explicitly set by Compare instruction
  - `cmpq src2, src1`
  - `cmpq b, a` sets flags based on a-b, but doesn’t store
    - **CF=1** if carry out from MSB (used for unsigned comparison)
    - **ZF=1** if a==b
    - **SF=1** if (a-b)<0 (signed)
    - **OF=1** if two’s complement (signed) overflow
      
      \[(a>0 \land b<0 \land (a-b)<0) \lor (a<0 \land b>0 \land (a-b)>0)\]
Condition Codes (Explicit Setting: Test)

- **Explicitly set by Test instruction**
  - `testq src2, src1`
  - `testq b, a` 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

<table>
<thead>
<tr>
<th>CF</th>
<th>ZF</th>
<th>SF</th>
<th>OF</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>a&gt;0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>a&lt;0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>a==0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td>not possible</td>
</tr>
</tbody>
</table>

**Carry Flag** Zero Flag **Sign Flag** Overflow Flag
# Reading Condition Codes

- **set* Instructions**
  - Set a low-order byte 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><strong>sete</strong> dst</td>
<td>ZF</td>
<td>Equal / Zero</td>
</tr>
<tr>
<td><strong>setne</strong> dst</td>
<td>~ZF</td>
<td>Not Equal / Not Zero</td>
</tr>
<tr>
<td><strong>sets</strong> dst</td>
<td>SF</td>
<td>Negative</td>
</tr>
<tr>
<td><strong>setns</strong> dst</td>
<td>~SF</td>
<td>Nonnegative</td>
</tr>
<tr>
<td><strong>setg</strong> dst</td>
<td>~ (SF^OF) &amp; ~ZF</td>
<td>Greater (Signed)</td>
</tr>
<tr>
<td><strong>setge</strong> dst</td>
<td>~ (SF^OF)</td>
<td>Greater or Equal (Signed)</td>
</tr>
<tr>
<td><strong>setl</strong> dst</td>
<td>(SF^OF)</td>
<td>Less (Signed)</td>
</tr>
<tr>
<td><strong>setle</strong> dst</td>
<td>(SF^OF)</td>
<td>Less or Equal (Signed)</td>
</tr>
<tr>
<td><strong>seta</strong> dst</td>
<td>~CF &amp; ~ZF</td>
<td>Above (unsigned “&gt;”)</td>
</tr>
<tr>
<td><strong>setb</strong> dst</td>
<td>CF</td>
<td>Below (unsigned “&lt;”)</td>
</tr>
</tbody>
</table>
x86-64 Integer Registers

- Accessing the low-order byte:

| %rax     | %al  |
| %rbx     | %bl  |
| %rcx     | %cl  |
| %rdx     | %dl  |
| %rsi     | %sil |
| %rdi     | %sil |
| %rsp     | %spl |
| %rbp     | %bpl |
| %r8      | %r8b |
| %r9      | %r9b |
| %r10     | %r10b|
| %r11     | %r11b|
| %r12     | %r12b|
| %r13     | %r13b|
| %r14     | %r14b|
| %r15     | %r15b|
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       # x-y
setg %al             # %al = (x>y)
movzbl %al, %eax     # %eax = (x>y)
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

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

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

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

Example:

\[ \text{movzbq} \%al, \%rbx \]

\[
\begin{array}{cccccccc}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \Rightarrow \%rax \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \Rightarrow \%rbx
\end{array}
\]
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} (\texttt{movz}) or \textbf{sign bit} (\texttt{movs})

\texttt{movzSD} / \texttt{movsSD}:
\begin{align*}
S & \quad \text{size of source (b = 1 byte, w = 2)} \\
D & \quad \text{size of dest (w = 2 bytes, l = 4, q = 8)}
\end{align*}

Example:
\texttt{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

Replace "j" with "set" to get other instructions.

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

cmp 5, (p)
- je: *p == 5
- jne: *p != 5
- jg: *p > 5
- jl: *p < 5

test a,a
- je: a == 0
- jne: a != 0
- jg: a > 0
- jl: a < 0

test a, 0x1
- je: a_{LSB} == 0
- jne: a_{LSB} == 1

Typically come in pairs:
1. test or compare
2. jump or set
Choosing instructions for conditionals

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

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>

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

cmpq $3, %rdi
jge T2
T1:  # x < 3:
    movq $1, %rax
    ret
T2:  # !(x < 3):  \( x \geq 3 \)
    movq $2, %rax
    ret
Your Turn!

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

Can view in provided `control.s`
- gcc -Og -S -fno-if-conversion control.c

<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>
Your Turn! (solution)

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

<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>

Can view in provided `control.s`

- gcc -Og -S -fno-if-conversion control.c
Choosing instructions for conditionals

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

\[\%al = (x < 3)\]
\[\%bl = (x = y)\]
\[\%al \& \%bl = 0\] when either \%al or \%bl is false

This is the else case!
Summary

- **lea** is address calculation instruction
  - Does NOT actually go to memory
  - Used to compute addresses or some arithmetic expressions

- 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