x86 Programming II
CSE 351 Winter 2017

http://xkcd.com/1652/
Administrivia
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*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><code>%rax</code></td>
<td>0x400</td>
<td>0x120</td>
</tr>
<tr>
<td><code>%rbx</code></td>
<td>0xF</td>
<td>0x118</td>
</tr>
<tr>
<td><code>%rcx</code></td>
<td>0x8</td>
<td>0x110</td>
</tr>
<tr>
<td><code>%rdx</code></td>
<td>0x100</td>
<td>0x108</td>
</tr>
<tr>
<td><code>%rdi</code></td>
<td>0x1</td>
<td>0x100</td>
</tr>
<tr>
<td><code>%rsi</code></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```
leaq (%rdx, %rcx, 4), %rax
movq (%rdx, %rcx, 4), %rbx
leaq (%rdx), %rdi
movq (%rdx), %rsi
```
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;
    long t5 = t3 + t4;
    long rval = t2 * t5;
    return rval;
}
```

<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>%rdx</td>
<td>3rd argument (z)</td>
</tr>
</tbody>
</table>

**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>

### Assembly Code

```
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;
}
```

```
max:
    ???
    movq %rdi, %rax
    ???
    ???
    movq %rsi, %rax
    ???
    ret
```
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: if $x \leq y$ then jump to else
Unconditional jump: jump to done

```assembly
max:  
if x <= y then jump to else
    movq  %rdi, %rax
    jump to done
else:
    movq  %rsi, %rax
done:
    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 {...}
Jumping

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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>jmp \texttt{target}</td>
<td>1</td>
<td>Unconditional</td>
</tr>
<tr>
<td>je \texttt{target}</td>
<td>ZF</td>
<td>Equal / Zero</td>
</tr>
<tr>
<td>jne \texttt{target}</td>
<td>\sim ZF</td>
<td>Not Equal / Not Zero</td>
</tr>
<tr>
<td>js \texttt{target}</td>
<td>SF</td>
<td>Negative</td>
</tr>
<tr>
<td>jns \texttt{target}</td>
<td>\sim SF</td>
<td>Nonnegative</td>
</tr>
<tr>
<td>jg \texttt{target}</td>
<td>\sim (SF\land OF) &amp; \sim ZF</td>
<td>Greater (Signed)</td>
</tr>
<tr>
<td>jge \texttt{target}</td>
<td>\sim (SF\land OF)</td>
<td>Greater or Equal (Signed)</td>
</tr>
<tr>
<td>jl \texttt{target}</td>
<td>(SF\land OF)</td>
<td>Less (Signed)</td>
</tr>
<tr>
<td>jle \texttt{target}</td>
<td>(SF\land OF) \lor ZF</td>
<td>Less or Equal (Signed)</td>
</tr>
<tr>
<td>ja \texttt{target}</td>
<td>\sim CF \land \sim ZF</td>
<td>Above (unsigned)</td>
</tr>
<tr>
<td>jb \texttt{target}</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>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
</tr>
<tr>
<td>%rbx</td>
</tr>
<tr>
<td>%rcx</td>
</tr>
<tr>
<td>%rdx</td>
</tr>
<tr>
<td>%r8</td>
</tr>
<tr>
<td>%r9</td>
</tr>
<tr>
<td>%r10</td>
</tr>
<tr>
<td>%r11</td>
</tr>
</tbody>
</table>

- current top of the Stack
- Program Counter (instruction pointer)

<table>
<thead>
<tr>
<th>Condition Codes</th>
</tr>
</thead>
<tbody>
<tr>
<td>CF</td>
</tr>
</tbody>
</table>

- Single bit registers:
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 && b>0 && t<0) || (a<0 && b<0 && t>=0)`

- **Not set by lea instruction** (beware!)
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 \ \&\& \ b<0 \ \&\& \ (a-b)<0) \ \| \ \|
  
  $(a<0 \ \&\& \ b>0 \ \&\& \ (a-b)>0)$
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>ZF</th>
<th>SF</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>&gt;0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>&lt;0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>==0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>not possible</td>
</tr>
</tbody>
</table>

**Carry Flag (CF)** | **Zero Flag (ZF)** | **Sign Flag (SF)** | **Overflow Flag (OF)**

- **ZF** = 1 if `a & b == 0` → `a == 0`
- **SF** = 1 if `a & b < 0` → `a < 0`
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><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>
x86-64 Integer Registers

- Accessing the low-order byte:

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

\( \forall \text{lo} \) for "low-order byte"  
\( \forall \text{b} \) for byte
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

\begin{align*}
\text{movz} & \quad \ell \text{ src, regDest} \quad \text{Move with zero extension} \\
\text{movs} & \quad \ell \text{ src, regDest} \quad \text{Move with sign extension}
\end{align*}

- 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*}
\ell & \quad \text{– size of source (} b = 1 \text{ byte, } w = 2 \text{)} \\
\underline{D} & \quad \text{– size of dest (} w = 2 \text{ bytes, } l = 4, \ q = 8 \text{)}
\end{align*}

Example:
\texttt{movzbq} \%al, \%rbx

\begin{align*}
\text{movzbq} \%al, \%rbx & \quad 0x?? | 0x?? | 0x?? | 0x?? | 0x?? | 0x?? | 0x?? | 0xFF \quad \%rax \\
0x00 | 0x00 | 0x00 | 0x00 | 0x00 | 0x00 | 0x00 | 0xFF \quad \%rbx
\end{align*}

\text{Zero-extend}
Aside: movz and movs

movz ___ src, regDest  Move with zero extension
movs ___ src, regDest  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)

movzSD / movsSD:
S – size of source (b = 1 byte, w = 2)
D – size of dest (w = 2 bytes, l = 4, q = 8)

Example:
movsbl (%rax), %ebx

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

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.
Choosing instructions for conditionals

replace "j" with "set" to get other instructions

<table>
<thead>
<tr>
<th>cmp b,a</th>
<th>test a,b</th>
</tr>
</thead>
<tbody>
<tr>
<td>je</td>
<td>a == b</td>
</tr>
<tr>
<td>jne</td>
<td>a != b</td>
</tr>
<tr>
<td>js</td>
<td>a&amp;b &lt; 0</td>
</tr>
<tr>
<td>jns</td>
<td>a&amp;b &gt;= 0</td>
</tr>
<tr>
<td>jg</td>
<td>a &gt; b</td>
</tr>
<tr>
<td>jge</td>
<td>a &gt;= b</td>
</tr>
<tr>
<td>jl</td>
<td>a &lt; b</td>
</tr>
<tr>
<td>jle</td>
<td>a &lt;= b</td>
</tr>
<tr>
<td>ja</td>
<td>a &gt; b</td>
</tr>
<tr>
<td>jb</td>
<td>a &lt; b</td>
</tr>
</tbody>
</table>

Typically come in pairs:
1. test or compare
2. jump or set

```
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
```
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)
---|---
%rdi | argument x
%rsi | argument y
%rax | return value

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

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

1. cmpq $3, %rdi
   setl %al
   cmpq %rsi, %rdi
   sete %bl
2. testb %al, %bl
   je T2 -> jump to T2 if (%al & %bl) == 0
3. T1: # x < 3 && x == y:
   movq $1, %rax
   ret
T2: # else
   movq $2, %rax
   ret
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