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/
Administrivia

- 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**
  - “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`

**Registers**

- `%rax`
- `%rbx`
- `%rcx` - 0x4
- `%rdx` - 0x100
- `%rdi`
- `%rsi`

**Memory**

<table>
<thead>
<tr>
<th>Word Address</th>
<th>0x100</th>
<th>0x108</th>
<th>0x110</th>
<th>0x118</th>
<th>0x120</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory</td>
<td>0x400</td>
<td>0xF</td>
<td>0x8</td>
<td>0x10</td>
<td>0x1</td>
</tr>
</tbody>
</table>

**Instructions**

- `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;
}
```

### 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:**

- `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>

```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;
}

Control Flow

<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:
    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

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

Current top of the Stack

Program Counter (instruction pointer)

Condition Codes

* CF
* ZF
* SF
* 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
    
    \[(a>0 \&\& b>0 \&\& t<0) || (a<0 \&\& b<0 \&\& t>=0)\]
  - *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
      \[(a>0 \land \land b<0 \land \land (a-b)<0) || (a<0 \land \land b>0 \land \land (a-b)>0)\]
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
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>
<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>%sil</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

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

```plaintext
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;
}
```

```
cmpq %rsi, %rdi  # Compare x:y
setg %al         # Set when >
movzbl %al, %eax # Zero rest of %rax
ret
```
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)

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:

movzbq %al, %rbx  

0x?? 0x?? 0x?? 0x?? 0x?? 0x?? 0x?? 0xFF ← %rax
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0xFF ← %rbx
Aside: movz and movs

\textbf{movz}_{\text{SD}} \quad \text{src, regDest} \quad \text{Move with zero extension}
\textbf{movs}_{\text{SD}} \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 \textbf{zero} (\texttt{movz}) or \textbf{sign bit} (\texttt{movs})

\texttt{movz} \text{SD} / \texttt{movs} \text{SD}:
\begin{itemize}
  \item \textit{S} – size of source (\texttt{b} = 1 byte, \texttt{w} = 2)
  \item \textit{D} – size of dest (\texttt{w} = 2 bytes, \texttt{l} = 4, \texttt{q} = 8)
\end{itemize}

Example:
\texttt{movsbl \%rax), \%ebx}

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

<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><code>a == b</code></td>
<td><code>a&amp;b == 0</code></td>
</tr>
<tr>
<td><code>jne</code></td>
<td>“Not equal”</td>
<td><code>a != b</code></td>
<td><code>a&amp;b != 0</code></td>
</tr>
<tr>
<td><code>js</code></td>
<td>“Sign” (negative)</td>
<td></td>
<td><code>a&amp;b &lt; 0</code></td>
</tr>
<tr>
<td><code>jns</code></td>
<td>(non-negative)</td>
<td></td>
<td><code>a&amp;b &gt;= 0</code></td>
</tr>
<tr>
<td><code>jg</code></td>
<td>“Greater”</td>
<td><code>a &gt; b</code></td>
<td><code>a&amp;b &gt; 0</code></td>
</tr>
<tr>
<td><code>jge</code></td>
<td>“Greater or equal”</td>
<td><code>a &gt;= b</code></td>
<td><code>a&amp;b &gt;= 0</code></td>
</tr>
<tr>
<td><code>jl</code></td>
<td>“Less”</td>
<td><code>a &lt; b</code></td>
<td><code>a&amp;b &lt; 0</code></td>
</tr>
<tr>
<td><code>jle</code></td>
<td>“Less or equal”</td>
<td><code>a &lt;= b</code></td>
<td><code>a&amp;b &lt;= 0</code></td>
</tr>
<tr>
<td><code>ja</code></td>
<td>“Above” (unsigned &gt;)</td>
<td><code>a &gt; b</code></td>
<td></td>
</tr>
<tr>
<td><code>jb</code></td>
<td>“Below” (unsigned &lt;)</td>
<td><code>a &lt; b</code></td>
<td></td>
</tr>
</tbody>
</table>
## Choosing instructions for conditionals

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

### Examples

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

```plaintext
# x < 3:
cmpq $3, %rdi
jge T2
T1: movq $1, %rax
    ret
# !(x < 3):
cmpq $3, %rdi
jne T2
T2: movq $2, %rax
    ret
```

### Register Use(s)

- %rdi: argument \(x\)
- %rsi: argument \(y\)
- %rax: return value
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`
Choosing instructions for conditionals

<table>
<thead>
<tr>
<th>if (x &lt; 3 &amp;&amp; x == y) { return 1; } else { return 2; }</th>
</tr>
</thead>
</table>

### Table: 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>“Equal”</td>
<td>a == b</td>
<td>a&amp;b == 0</td>
</tr>
<tr>
<td>jne</td>
<td>“Not equal”</td>
<td>a != b</td>
<td>a&amp;b != 0</td>
</tr>
<tr>
<td>js</td>
<td>“Sign” (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>“Greater”</td>
<td>a &gt; b</td>
<td>a&amp;b &gt; 0</td>
</tr>
<tr>
<td>jge</td>
<td>“Greater or equal”</td>
<td>a &gt;= b</td>
<td>a&amp;b &gt;= 0</td>
</tr>
<tr>
<td>jl</td>
<td>“Less”</td>
<td>a &lt; b</td>
<td>a&amp;b &lt; 0</td>
</tr>
<tr>
<td>jle</td>
<td>“Less or equal”</td>
<td>a &lt;= b</td>
<td>a&amp;b &lt;= 0</td>
</tr>
<tr>
<td>ja</td>
<td>“Above” (unsigned &gt;)</td>
<td>a &gt; b</td>
<td></td>
</tr>
<tr>
<td>jb</td>
<td>“Below” (unsigned &lt;)</td>
<td>a &lt; b</td>
<td></td>
</tr>
</tbody>
</table>

### Example Assembly Code

```assembly
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

- **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