Instructor: Ruth Anderson

Teaching Assistants: Alex Olshanskyy, Rehaan Bhimani, Callum Walker, Chin Yeoh, Diya Joy, Eric Fan, Edan Sneh, Jonathan Chen, Jeffery Tian, Millicent Li, Melissa Birchfield, Porter Jones, Joseph Schafer, Connie Wang, Eddy (Tianyi) Zhou

http://xkcd.com/409/
Administrivia

- hw8 due Monday – 11am
- Lab 1b due Monday (4/20)
  - Submit `bits.c` and `lab1Breflect.txt`
  - Submissions that fail the autograder get a ZERO
    - No excuses – make full use of tools & Gradescope’s interface
- Lab 2 (x86-64) coming soon
  - Learn to read x86-64 assembly and use GDB

- You must log on with your @uw google account to access!!
  - Google doc for 11:30 Lecture: https://tinyurl.com/351-04-17A
  - Google doc for 2:30 Lecture: https://tinyurl.com/351-04-17B
Address Computation Instruction

- **leaq src, dst**
  - "lea" stands for **load effective address**
  - src is an 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 \( x+k*i+d \)
  - Though \( k \) can only be 1, 2, 4, or 8
Example: `lea` vs. `mov`

### Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>%rax</code></td>
<td>0x110</td>
</tr>
<tr>
<td><code>%rbx</code></td>
<td>0x8</td>
</tr>
<tr>
<td><code>%rcx</code></td>
<td>0x4</td>
</tr>
<tr>
<td><code>%rdx</code></td>
<td>0x100</td>
</tr>
<tr>
<td><code>%rdi</code></td>
<td>0x100</td>
</tr>
<tr>
<td><code>%rsi</code></td>
<td>0x1</td>
</tr>
</tbody>
</table>

### Memory

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

### Instructions

- `leaq (%rdx,%rcx,4), %rax`
- `movq (%rdx,%rcx,4), %rbx`
- `leaq (%rdx), %rdi`
- `movq (%rdx), %rsi`
lea – “It just does math”
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>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>
Polling Question [Asm II – a]

Which of the following x86-64 instructions correctly calculates $rax = 9 \times rdi$?

- Vote at [http://pollev.com/rea](http://pollev.com/rea)

A. `leaq (,%rdi,9), %rax`
B. `movq (,%rdi,9), %rax`
C. `leaq (%rdi,%rdi,8), %rax`  \( S \in \{1,2,4,8\} \)
D. `movq (%rdi,%rdi,8), %rax`  \( \rightarrow rax = 9 \times rdi \)
E. We’re lost...

\( \rightarrow rax = 9 \times rdi \)
## 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:
```
???
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;
}

Register | Use(s)
--- | ---
%rdi | 1st argument (x)
%rsi | 2nd argument (y)
%rax | return value

Conditional jump

Unconditional jump

max:
if TRUE
    if x <= y then jump to else
    movq %rdi, %rax
    jump to done
if FALSE
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 {...}
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 ) “flags”
    - 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

Carry | Zero | Sign | Overflow
---|---|---|---
CF | ZF | SF | OF
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 (if MSB is 1)
  - **OF=1** if signed overflow
    - `(s>0 && d>0 && r<0) || (s<0 && d<0 && r>=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 src1, src2` like `subq a, b → b - a`
  - `cmpq a, b` sets flags based on `b - a`, but *doesn’t store result*

- **CF=1** if carry out from MSB (good for *unsigned* comparison)
- **ZF=1** if `a==b` (`b-a==0`)
- **SF=1** if `(b-a)<0` (if MSB is 1)
- **OF=1** if *signed* overflow
  
  `(a>0 && b<0 && (b-a)>0) || (a<0 && b>0 && (b-a)<0)`
Condition Codes (Explicit Setting: Test)

- *Explicitly set by Test* instruction
  - `testq src2, src1` like `andq a, b`
  - `testq a, b` sets flags based on `a & b`, but doesn’t store result
    - 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)
## 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><code>jmp target</code></td>
<td>1</td>
<td>Unconditional</td>
</tr>
<tr>
<td><code>je target</code></td>
<td>ZF</td>
<td>Equal / Zero</td>
</tr>
<tr>
<td><code>jne target</code></td>
<td>~ZF</td>
<td>Not Equal / Not Zero</td>
</tr>
<tr>
<td><code>js target</code></td>
<td>SF</td>
<td>Negative</td>
</tr>
<tr>
<td><code>jns target</code></td>
<td>~SF</td>
<td>Nonnegative</td>
</tr>
<tr>
<td><code>jg target</code></td>
<td>~SF</td>
<td>Greater (Signed)</td>
</tr>
<tr>
<td><code>jge target</code></td>
<td>~SF^OF</td>
<td>Greater or Equal (Signed)</td>
</tr>
<tr>
<td><code>jl target</code></td>
<td>SF^OF</td>
<td>Less (Signed)</td>
</tr>
<tr>
<td><code>jle target</code></td>
<td>SF^OF</td>
<td>Less or Equal (Signed)</td>
</tr>
<tr>
<td><code>ja target</code></td>
<td>~CF&amp;~ZF</td>
<td>Above (unsigned “&gt;”)</td>
</tr>
<tr>
<td><code>jb target</code></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 $dst$</td>
<td>ZF</td>
<td>Equal / Zero</td>
</tr>
<tr>
<td>setne $dst$</td>
<td>$\sim ZF$</td>
<td>Not Equal / Not Zero</td>
</tr>
<tr>
<td>sets $dst$</td>
<td>SF</td>
<td>Negative</td>
</tr>
<tr>
<td>setns $dst$</td>
<td>$\sim SF$</td>
<td>Nonnegative</td>
</tr>
<tr>
<td>setg $dst$</td>
<td>$(SF^OF) &amp; \sim ZF$</td>
<td>Greater (Signed)</td>
</tr>
<tr>
<td>setge $dst$</td>
<td>$(SF^OF)$</td>
<td>Greater or Equal (Signed)</td>
</tr>
<tr>
<td>setl $dst$</td>
<td>$(SF^OF)$</td>
<td>Less (Signed)</td>
</tr>
<tr>
<td>setle $dst$</td>
<td>$(SF^OF) \mid ZF$</td>
<td>Less or Equal (Signed)</td>
</tr>
<tr>
<td>seta $dst$</td>
<td>$\sim CF &amp; \sim ZF$</td>
<td>Above (unsigned “&gt;”)</td>
</tr>
<tr>
<td>setb $dst$</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>

↑ 8B ↑ 1B
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

```
int gt(long x, long y)
{
    return x > y;  // x-y > 0
}
```

- `cmpq %rsi, %rdi` # set flags based on `x-y`
- `setg %al` # `%al = (x>y)`
- `movzbl %al, %eax` # `%rax = (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;
}
```

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

\begin{itemize}
\item \texttt{movz} \verb|src, regDest| \quad \# Move with zero extension
\item \texttt{movs} \verb|src, regDest| \quad \# Move with sign extension
\end{itemize}

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

\texttt{movzSD} / \texttt{movsSD}:

\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}

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

\begin{align*}
\%al & : 0x?? \quad 0x?? \quad 0x?? \quad 0x?? \quad 0x?? \quad 0x?? \quad 0x?? \quad 0xFF \\
\%rbx & : 0x00 \quad 0x00 \quad 0x00 \quad 0x00 \quad 0x00 \quad 0x00 \quad 0x00 \quad 0xFF
\end{align*}
Aside: movz and movs

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

- 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 (\text{movz}) or sign bit (\text{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:

\[
\begin{align*}
\text{movsbl} & \quad (\%rax), \%ebx \\
\end{align*}
\]

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.
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