Instructor: Porter Jones

Teaching Assistants: Amy Xu
Callum Walker
Sam Wolfson
Tim Mandzyuk

http://xkcd.com/409/
Administrivia

- Questions doc: [https://tinyurl.com/CSE351-7-10](https://tinyurl.com/CSE351-7-10)
- See my email about accommodations!
- Lab 1b now due Monday at 11:59pm (7/13)
  - Submit `aisle_manager.c`, `store_client.c`, and `lab1Breflect.txt`
  - Can still use late days until 7/15
- hw6, hw7 now due Monday (7/13) – 10:30am
- Unit Summary 1 now due Friday (7/17) – 11:59pm
  - Can still use late days until 7/20
- Mid-quarter Survey still due Friday (7/17) – 11:59pm
- hw8, hw9, hw10 now due Monday (7/20) – 10:30am
Administrivia

- Lab1a grades released later today
  - Talk to us about any questions you have!
  - Regrades open 24 hours after an assignment is due, stay open usually for about a week

- Lab 2 released later today!
  - Debugging x86-64 assembly using gdb

- I will now drop your lowest homework score (see Syllabus for more details).
  - Essentially will bump your homework total up by 11.5 points (the largest single homework total).
x86-64 Introduction

❖ Data transfer instruction \texttt{(mov)}
❖ Arithmetic operations
❖ Memory addressing modes
❖ Address computation instruction \texttt{(lea)}
Memory Addressing Modes: Basic

❖ **Indirect:** \((R)\) \(\text{Mem}[\text{Reg}[R]]\)
  - Data in register \(R\) specifies the memory address
  - Like pointer dereference in C
  - **Example:** \texttt{movq }\(\%rcx\), \%rax

❖ **Displacement:** \(D(R)\) \(\text{Mem}[\text{Reg}[R] + D]\)
  - Data in register \(R\) specifies the *start* of some memory region
  - Constant displacement \(D\) specifies the offset from that address
  - **Example:** \texttt{movq }\(8(\%rbp)\), \%rdx
Complete Memory Addressing Modes

❖ General:

- \( D(Rb, Ri, S) \) \( \text{Mem}[\text{Reg}[Rb]+\text{Reg}[Ri] \times S+D] \)
  - **Rb**: Base register (any register)
  - **Ri**: Index register (any register except \( %rsp \))
  - **S**: Scale factor (1, 2, 4, 8) – *why these numbers?*
  - **D**: Constant displacement value (a.k.a. immediate)

❖ Special cases (see CSPP Figure 3.3 on p.181)

- \( D(Rb, Ri) \) \( \text{Mem}[\text{Reg}[Rb]+\text{Reg}[Ri]+D] \) (S=1)
- \( (Rb, Ri, S) \) \( \text{Mem}[\text{Reg}[Rb]+\text{Reg}[Ri] \times S] \) (D=0)
- \( (Rb, Ri) \) \( \text{Mem}[\text{Reg}[Rb]+\text{Reg}[Ri]] \) (S=1, D=0)
- \( (, Ri, S) \) \( \text{Mem}[\text{Reg}[Ri] \times S] \) (Rb=0, D=0)
## Address Computation Examples

<table>
<thead>
<tr>
<th>Expression</th>
<th>Address Computation</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x8(%%rdx)</td>
<td>Mem[Reg[Rb]+Reg[Ri]*S+D]</td>
<td></td>
</tr>
<tr>
<td>(%%rdx,%%rcx)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(%%rdx,%%rcx,4)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x80(,%%rdx,2)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Table:**

<table>
<thead>
<tr>
<th>Expression</th>
<th>Address Computation</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x8(%%rdx)</td>
<td>Mem[Reg[Rb]+Reg[Ri]*S+D]</td>
<td></td>
</tr>
<tr>
<td>(%%rdx,%%rcx)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(%%rdx,%%rcx,4)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x80(,%%rdx,2)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Examples:**

- `%%rdx`: 0xf000
- `%%rcx`: 0x0100

D(Rb,Ri,S) → Mem[Reg[Rb]+Reg[Ri]*S+D]
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+d`
    - 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 0x4</td>
<td>0x8</td>
<td>0x110</td>
</tr>
<tr>
<td>%rdx 0x100</td>
<td>0x10</td>
<td>0x108</td>
</tr>
<tr>
<td>%rdi</td>
<td>0x1</td>
<td>0x100</td>
</tr>
<tr>
<td>%rsi</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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;
}
```

<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;
}
```

<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
```
Polling Question [Asm II – a]

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

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

A. `leaq (,%rdi,9), %rax`
B. `movq (,%rdi,9), %rax`
C. `leaq (%rdi,%rdi,8), %rax`
D. `movq (%rdi,%rdi,8), %rax`
E. We’re lost...
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:

```plaintext
???
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;
}

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

<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>
Conditionals and Control Flow

❖ Conditional branch/\textit{jump}
  ▪ Jump to somewhere else if some \textit{condition} is true, otherwise execute next instruction

❖ Unconditional branch/\textit{jump}
  ▪ \textit{Always} jump when you get to this instruction

❖ Together, they can implement most control flow constructs in high-level languages:
  ▪ \textbf{if} (\textit{condition}) \textbf{then} {...} \textbf{else} {...}
  ▪ \textbf{while} (\textit{condition}) {...}
  ▪ \textbf{do} {...} \textbf{while} (\textit{condition})
  ▪ \textbf{for} (\textit{initialization}; \textit{condition}; \textit{iterative}) {...}
  ▪ \textbf{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>%r8</th>
<th>%r9</th>
<th>%r10</th>
<th>%r11</th>
<th>%r12</th>
<th>%r13</th>
<th>%r14</th>
<th>%r15</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%rbx</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%rcx</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%rdx</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%rsi</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%rdi</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%rsp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%rbp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- 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 ↔ 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
  
  \[(s>0 \land d>0 \land r<0) \lor (s<0 \land d<0 \land r>=0)\]

- **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 `b-a`
  - **CF=1** if carry out from MSB (good for unsigned comparison)
  - **ZF=1** if `a==b`
  - **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`
  - `testq a, b` sets flags based on `a&b`, but doesn’t store `a&b`
    - 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)

<table>
<thead>
<tr>
<th>CF</th>
<th>Carry Flag</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZF</td>
<td>Zero Flag</td>
</tr>
<tr>
<td>SF</td>
<td>Sign Flag</td>
</tr>
<tr>
<td>OF</td>
<td>Overflow Flag</td>
</tr>
</tbody>
</table>
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>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 “&gt;”)</td>
</tr>
<tr>
<td>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>sete <code>dst</code></td>
<td><code>ZF</code></td>
<td>Equal / Zero</td>
</tr>
<tr>
<td>setne <code>dst</code></td>
<td><code>~ZF</code></td>
<td>Not Equal / Not Zero</td>
</tr>
<tr>
<td>sets <code>dst</code></td>
<td><code>SF</code></td>
<td>Negative</td>
</tr>
<tr>
<td>setns <code>dst</code></td>
<td><code>~SF</code></td>
<td>Nonnegative</td>
</tr>
<tr>
<td>setg <code>dst</code></td>
<td><code>~(SF^OF) &amp; ~ZF</code></td>
<td>Greater (Signed)</td>
</tr>
<tr>
<td>setge <code>dst</code></td>
<td><code>~(SF^OF)</code></td>
<td>Greater or Equal (Signed)</td>
</tr>
<tr>
<td>setl <code>dst</code></td>
<td><code>(SF^OF)</code></td>
<td>Less (Signed)</td>
</tr>
<tr>
<td>setle <code>dst</code></td>
<td>`(SF^OF)</td>
<td>ZF`</td>
</tr>
<tr>
<td>seta <code>dst</code></td>
<td><code>~CF &amp; ~ZF</code></td>
<td>Above (unsigned “&gt;”)</td>
</tr>
<tr>
<td>setb <code>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>Register</th>
<th>Low-Order Byte</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>%al</td>
</tr>
<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>%dil</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>
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;
}
```

<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>
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;
}
```

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

\texttt{movz\_\_ src, regDest} \quad \# \text{Move with \underline{zero} extension}
\texttt{movs\_\_ src, regDest} \quad \# \text{Move with \underline{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 SD / movs SD:}
\begin{itemize}
  \item \texttt{S} – size of source (\texttt{b} = 1 byte, \texttt{w} = 2)
  \item \texttt{D} – size of dest (\texttt{w} = 2 bytes, \texttt{l} = 4, \texttt{q} = 8)
\end{itemize}

Example:
\texttt{movzbq \%al, \%rbx} \quad \begin{array}{c}
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 \# 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:
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.
Summary

❖ **Memory Addressing Modes:** The addresses used for accessing memory in `mov` (and other) instructions can be computed in several different ways
  - *Base register, index register, scale factor, and displacement* map well to pointer arithmetic operations

❖ **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