**x86-64 Programming II**
CSE 351 Autumn 2017

**Instructor:**
Justin Hsia

**Teaching Assistants:**
Lucas Wotton
Michael Zhang
Parker DeWilde
Ryan Wong
Sam Gehman
Sam Wolfson
Savanna Yee
Vinny Palaniappan

http://xkcd.com/409/
Administrivia

- Lab 2 (x86-64) released tomorrow (10/17)
  - Learn to read x86-64 assembly and use GDB
- Homework 2 due Friday (10/20)
- Midterm is in two Mondays (10/30, 5pm in KNE 120)
  - No lecture that day
  - You will be provided a fresh reference sheet
    - Study and use this NOW so you are comfortable with it when the exam comes around
  - You get 1 *handwritten*, double-sided cheat sheet (letter)
  - Find a study group! Look at past exams!
Control Flow

```c
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

**Assembly Code**
- `movq %rdi, %rax` (for `x`)
- `movq %rsi, %rax` (for `y`)
- `ret`
Control Flow

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

### Conditional jump

if \( x \leq y \) then jump to else

### Unconditional jump

jump to done

---

<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 TRUE

movq %rdi, %rax

if FALSE

jmp done

done:

movq %rsi, %rax

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></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>

- Program Counter (instruction pointer)
  - Carry
  - Zero
  - Sign
  - Overflow

Condition Codes
Condition Codes (Implicit Setting)

- *Implicitly* set by arithmetic operations
  - (think of it as side effects)
  - **Example:** `addq src, dst ⇔ r = d+s
    \[
    \text{result} = \text{dst} + \text{src}
    \]

- **CF=1** if carry out from MSB (unsigned overflow)
- **ZF=1** if \( r==0 \)
- **SF=1** if \( r<0 \) (assuming signed, actually just if MSB is 1)
- **OF=1** if two’s complement (signed) overflow:
  \[
  (s>0 \land d>0 \land r<0) \lor (s<0 \land d<0 \land r\geq0)
  \]
- *Not set by lea instruction (beware!)*

<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>
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

- **CF=1** if carry out from MSB (used for unsigned comparison)
- **ZF=1** if $a==b$
- **SF=1** if $(b-a)<0$ (signed)
- **OF=1** if two’s complement (signed) overflow
  $$(a>0 \land b<0 \land (b-a)>0) \lor (a<0 \land b>0 \land (b-a)<0)$$
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
    - 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></th>
<th>ZF</th>
<th>SF</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>&gt;0</td>
<td>a &gt; 0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>&lt;0</td>
<td>a &lt; 0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>==0</td>
<td>a == 0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>a = 0</td>
</tr>
</tbody>
</table>

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

\[ 8B \quad 18 \]
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;
}
```

```
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;
}
```

<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}\hspace{1cm} \textit{Move with zero extension}
- \texttt{movs\_src, regDest}\hspace{1cm} \textit{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 \textit{zero} (\texttt{movz}) or \textit{sign bit} (\texttt{movs})

\texttt{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:

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

\begin{verbatim}
0x?? 0x?? 0x?? 0x?? 0x?? 0x?? 0x?? 0xFF \rarrow \%rax
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0xFF \rarrow \%rbx
\end{verbatim}
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 (\texttt{movz}) or sign bit (\texttt{movs})

\texttt{movz SD / movs 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}
  \begin{itemize}
    \item Copy 1 byte from memory into 8-byte register & sign extend it
  \end{itemize}

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

- All arithmetic instructions set condition flags based on result of operation (op)
  - Conditionals are comparisons against 0

- Come in instruction pairs

```
1 addq 5, (p)
   je:  *p+5 == 0
2 jne: *p+5 != 0
   jg:  *p+5 > 0
   jl:  *p+5 < 0
```

```
1 orq a, b
   je:  b|a == 0
2 jne: b|a != 0
   jg:  b|a > 0
   jl:  b|a < 0
```

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>je</td>
<td>“Equal”</td>
<td>d (op) s == 0</td>
</tr>
<tr>
<td>jne</td>
<td>“Not equal”</td>
<td>d (op) s != 0</td>
</tr>
<tr>
<td>js</td>
<td>“Sign” (negative)</td>
<td>d (op) s &lt; 0</td>
</tr>
<tr>
<td>jns</td>
<td>(non-negative)</td>
<td>d (op) s &gt;= 0</td>
</tr>
<tr>
<td>jg</td>
<td>“Greater”</td>
<td>d (op) s &gt; 0</td>
</tr>
<tr>
<td>jge</td>
<td>“Greater or equal”</td>
<td>d (op) s &gt;= 0</td>
</tr>
<tr>
<td>jl</td>
<td>“Less”</td>
<td>d (op) s &lt; 0</td>
</tr>
<tr>
<td>jle</td>
<td>“Less or equal”</td>
<td>d (op) s &lt;= 0</td>
</tr>
<tr>
<td>ja</td>
<td>“Above” (unsigned &gt;)</td>
<td>d (op) s &gt; 0U</td>
</tr>
<tr>
<td>jb</td>
<td>“Below” (unsigned &lt;)</td>
<td>d (op) s &lt; 0U</td>
</tr>
</tbody>
</table>
Choosing instructions for conditionals

- **Reminder:** `cmp` is like `sub`, `test` is like `and`
  - Result is not stored anywhere

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

- **Example Instructions**

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

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

  - `testb a, 0x1`
    - `je: a_{LSB} == 0`
    - `jne: a_{LSB} == 1`
Choosing instructions for conditionals

if \( (x < 3) \) {
    return 1;
} else {
    return 2;
}

cmpq $3, %rdi
jge T2

T1: \# x < 3: (if)
    movq $1, %rax
    ret

T2: \# !(x < 3): (else)
    movq $2, %rax
    ret
Question

<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>

A. \(\text{cmpq} \ %\text{rsi}, \ %\text{rdi}\) \(\text{jle} \ .L4\)

B. \(\text{cmpq} \ %\text{rsi}, \ %\text{rdi}\) \(\text{jg} \ .L4\)

C. \(\text{testq} \ %\text{rsi}, \ %\text{rdi}\) \(\text{jle} \ .L4\)

D. \(\text{testq} \ %\text{rsi}, \ %\text{rdi}\) \(\text{jg} \ .L4\)

E. We’re lost...

---

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

absdiff:

```assembly
.absdiff:
    __________________________
    ________________ # x > y:
    movq %rdi, %rax
    subq %rsi, %rax
    ret
    .L4:                  # x <= y:
    movq %rsi, %rax
    subq %rdi, %rax
    ret
```
Choosing instructions for conditionals

### Conditionals

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Condition</th>
<th>X86-64 Code</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>je</strong></td>
<td>“Equal”</td>
<td>$x &lt; 3$ and $x == y$</td>
<td>$b \ x == \ y$ and $b&amp;a == 0$</td>
</tr>
</tbody>
</table>
| **jne** | “Not equal” | $x 
eq y$ | $b \ ! = \ a$ and $b\&a \ != \ 0$ |
| **js** | “Sign” (negative) | $b - a < 0$ | $b\&a < 0$ |
| **jns** | (non-negative) | $b - a \geq 0$ | $b\&a \geq 0$ |
| **jg** | “Greater” | $b > a$ | $b\&a > 0$ |
| **jge** | “Greater or equal” | $b \geq a$ | $b\&a \geq 0$ |
| **jl** | “Less” | $b < 3$ | $b\&a < 0$ |
| **jle** | “Less or equal” | $b \leq a$ | $b\&a \leq 0$ |
| **ja** | “Above” (unsigned $>$) | $b > a$ | $b\&a > 0_U$ |
| **jb** | “Below” (unsigned $<$) | $b < a$ | $b\&a < 0_U$ |

### Examples

```c
if (x < 3 && x == y) {
    return 1;
} else {
    do this if either %al or %bl are False
    return 2;
}
```

```x86-64
# x < 3 && x == y:
cmpq $3, %rdi
setl %al
cmpq %rsi, %rdi
sete %bl
testb %al, %bl
je T2 \rightarrow jump to T2 if (%al \& %bl) == 0

T1: \# x < 3 && x == y:
    movq $1, %rax
    ret

T2: \# else
    movq $2, %rax
    ret
```
Choosing instructions for conditionals

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

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

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

https://godbolt.org/g/KntpyG
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