x86-64 Programming III
CSE 351 Autumn 2018

Instructor:
Justin Hsia

Teaching Assistants:
Akshat Aggarwal
An Wang
Andrew Hu
Brian Dai
Britt Henderson
James Shin
Kevin Bi
Kory Watson
Riley Germundson
Sophie Tian
Teagan Horkan

http://xkcd.com/1652/
Administrivia

- Homework 2 due Friday (10/19)
- Lab 2 due next Friday (10/26)

- Section tomorrow on Assembly and GDB
  - Bring your laptops!
  - GDB Tutorial Session: Thu 10/18, 5:30 pm, GUG 220

- Midterm: 10/29, 5 pm in KNE 210 & 220
  - You will be provided a fresh reference sheet
  - You get 1 handwritten, double-sided cheat sheet (letter)
  - Midterm Clobber Policy: replace midterm score with score on midterm portion of the final if you “do better”
Aside: movz and movs

movz
dest src, regDest  # Move with zero extension
movs
dest 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

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.
GDB Demo

- The `movz` and `movs` examples on a real machine!
  - `movzbq %al, %rbx`
  - `movsbl (%rax), %ebx`

- You will need to use GDB to get through Lab 2
  - Useful debugger in this class and beyond!

- Pay attention to:
  - Setting breakpoints (`break`)
  - Stepping through code (`step/next` and `steppi/nexti`)
  - Printing out expressions (`print` – works with regs & vars)
  - Examining memory (`x`)
Choosing instructions for conditionals

- All arithmetic instructions set condition flags based on result of operation \( \text{op} \)
  - Conditionals are comparisons against 0
- Come in instruction pairs

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>addq 5, (p)</td>
<td>je:</td>
<td>*p+5 == 0</td>
</tr>
<tr>
<td></td>
<td>jne:</td>
<td>*p+5 != 0</td>
</tr>
<tr>
<td></td>
<td>jg:</td>
<td>*p+5 &gt; 0</td>
</tr>
<tr>
<td></td>
<td>jl:</td>
<td>*p+5 &lt; 0</td>
</tr>
<tr>
<td>orq a, b</td>
<td>je:</td>
<td>b</td>
</tr>
<tr>
<td></td>
<td>jne:</td>
<td>b</td>
</tr>
<tr>
<td></td>
<td>jg:</td>
<td>b</td>
</tr>
<tr>
<td></td>
<td>jl:</td>
<td>b</td>
</tr>
<tr>
<td></td>
<td>jg:</td>
<td>d (op) s &gt; 0</td>
</tr>
<tr>
<td></td>
<td>jns:</td>
<td>(non-negative) d (op) s &gt;= 0</td>
</tr>
<tr>
<td></td>
<td>js:</td>
<td>“Sign” (negative) d (op) s &lt; 0</td>
</tr>
<tr>
<td></td>
<td>jge:</td>
<td>“Greater or equal” d (op) s &gt;= 0</td>
</tr>
<tr>
<td></td>
<td>jle:</td>
<td>“Less or equal” d (op) s &lt;= 0</td>
</tr>
<tr>
<td></td>
<td>ja:</td>
<td>“Above” (unsigned &gt;) d (op) s &gt; 0U</td>
</tr>
<tr>
<td></td>
<td>jb:</td>
<td>“Below” (unsigned &lt;) 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>Instruction</th>
<th>cmp a,b</th>
<th>test a,b</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>je</code> “Equal”</td>
<td>b == a</td>
<td>b&amp;a == 0</td>
</tr>
<tr>
<td><code>jne</code> “Not equal”</td>
<td>b != a</td>
<td>b&amp;a != 0</td>
</tr>
<tr>
<td><code>js</code> “Sign” (negative)</td>
<td>b-a &lt; 0</td>
<td>b&amp;a &lt; 0</td>
</tr>
<tr>
<td><code>jns</code> (non-negative)</td>
<td>b-a &gt;= 0</td>
<td>b&amp;a &gt;= 0</td>
</tr>
<tr>
<td><code>jg</code> “Greater”</td>
<td>b &gt; a</td>
<td>b&amp;a &gt; 0</td>
</tr>
<tr>
<td><code>jge</code> “Greater or equal”</td>
<td>b &gt;= a</td>
<td>b&amp;a &gt;= 0</td>
</tr>
<tr>
<td><code>jl</code> “Less”</td>
<td>b &lt; a</td>
<td>b&amp;a &lt; 0</td>
</tr>
<tr>
<td><code>jle</code> “Less or equal”</td>
<td>b &lt;= a</td>
<td>b&amp;a &lt;= 0</td>
</tr>
<tr>
<td><code>ja</code> “Above” (unsigned &gt;)</td>
<td>b &gt; a</td>
<td>b&amp;a &gt; 0U</td>
</tr>
<tr>
<td><code>jb</code> “Below” (unsigned &lt;)</td>
<td>b &lt; a</td>
<td>b&amp;a &lt; 0U</td>
</tr>
</tbody>
</table>

- `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

<table>
<thead>
<tr>
<th>Condition</th>
<th>Description</th>
<th>Comparison</th>
<th>Truth Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{je} )</td>
<td>“Equal”</td>
<td>( b == a )</td>
<td>( b&amp;a == 0 )</td>
</tr>
<tr>
<td>( \text{jne} )</td>
<td>“Not equal”</td>
<td>( b \neq a )</td>
<td>( b&amp;a \neq 0 )</td>
</tr>
<tr>
<td>( \text{js} )</td>
<td>“Sign” (negative)</td>
<td>( b-a &lt; 0 )</td>
<td>( b&amp;a &lt; 0 )</td>
</tr>
<tr>
<td>( \text{jns} )</td>
<td>(non-negative)</td>
<td>( b-a &gt;= 0 )</td>
<td>( b&amp;a &gt;= 0 )</td>
</tr>
<tr>
<td>( \text{jg} )</td>
<td>“Greater”</td>
<td>( b &gt; a )</td>
<td>( b&amp;a &gt; 0 )</td>
</tr>
<tr>
<td>( \text{jge} )</td>
<td>“Greater or equal”</td>
<td>( b &gt;= a )</td>
<td>( b&amp;a &gt;= 0 )</td>
</tr>
<tr>
<td>( \text{jl} )</td>
<td>“Less”</td>
<td>( b &lt; a )</td>
<td>( b&amp;a &lt; 0 )</td>
</tr>
<tr>
<td>( \text{jle} )</td>
<td>“Less or equal”</td>
<td>( b &lt;= a )</td>
<td>( b&amp;a &lt;= 0 )</td>
</tr>
<tr>
<td>( \text{ja} )</td>
<td>“Above” (unsigned &gt;)</td>
<td>( b &gt; a )</td>
<td>( b&amp;a &gt; 0U )</td>
</tr>
<tr>
<td>( \text{jb} )</td>
<td>“Below” (unsigned &lt;)</td>
<td>( b &lt; a )</td>
<td>( b&amp;a &lt; 0U )</td>
</tr>
</tbody>
</table>

Register Use(s):

- \( %\text{rdi} \): argument \( x \)
- \( %\text{rsi} \): argument \( y \)
- \( %\text{rax} \): return value

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

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

A. `cmpq %rsi, %rdi`  
   `jle .L4`  
   `x-y`

B. `cmpq %rsi, %rdi`  
   `jg .L4`  
   `x-y`

C. `testq %rsi, %rdi`  
   `jle .L4`  
   `x & y`

D. `testq %rsi, %rdi`  
   `jg .L4`  
   `x & y`

E. We’re lost…

---

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

```
absdiff:  

        # x > y:
        movq %rdi, %rax
        subq %rsi, %rax
        ret

.L4:  

        # x <= y:
        movq %rsi, %rax
        subq %rdi, %rax  
        x-y <= 0  
        less than or equal to (le)
        ret
```
Choosing instructions for conditionals

<table>
<thead>
<tr>
<th>Condition</th>
<th>cmp a,b</th>
<th>test a,b</th>
</tr>
</thead>
<tbody>
<tr>
<td>je “Equal”</td>
<td>$1 = x == y$</td>
<td>$0 = b &amp; a$</td>
</tr>
<tr>
<td>jne “Not equal”</td>
<td>$a = b ! = 0$</td>
<td></td>
</tr>
<tr>
<td>js “Sign” (negative)</td>
<td>$a = b - a$</td>
<td>$0 = b &amp; a$</td>
</tr>
<tr>
<td>jns (non-negative)</td>
<td>$a = b - a &gt;= 0$</td>
<td>$0 = b &amp; a$</td>
</tr>
<tr>
<td>jg “Greater”</td>
<td>$a = b &gt; a$</td>
<td>$0 = b &amp; a$</td>
</tr>
<tr>
<td>jge “Greater or equal”</td>
<td>$a = b &gt;= a$</td>
<td>$0 = b &amp; a$</td>
</tr>
<tr>
<td>jl “Less”</td>
<td>$0 = b - a$</td>
<td>$0 = b &amp; a$</td>
</tr>
<tr>
<td>jle “Less or equal”</td>
<td>$a = b &lt;= a$</td>
<td></td>
</tr>
<tr>
<td>ja “Above” (unsigned &gt;)</td>
<td>$0 = b &gt; a$</td>
<td>$0 = b &amp; a$</td>
</tr>
<tr>
<td>jb “Below” (unsigned &lt;)</td>
<td>$a = b &lt; a$</td>
<td>$0 = b &amp; a$</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 ← jump to T2 if (%al %bl) = 0

T1: # x < 3 && x == y:
    movq $1, %rax
    ret
T2: # else
    movq $2, %rax
    ret

https://godbolt.org/z/j72AEn
Labels

A jump changes the program counter (%rip)
- %rip tells the CPU the address of the next instr to execute

Labels give us a way to refer to a specific instruction in our assembly/machine code
- Associated with the next instruction found in the assembly code (ignores whitespace)
- Each use of the label will eventually be replaced with something that indicates the final address of the instruction that it is associated with

```
swap:
    movq (%rdi), %rax
    movq (%rsi), %rdx
    movq %rdx, (%rdi)
    movq %rax, (%rsi)
    ret

max:
    movq %rdi, %rax
    cmpq %rsi, %rdi
    jg done
    movq %rsi, %rax

done:
    ret
```
x86 Control Flow

- Condition codes
- Conditional and unconditional branches
- Loops
- Switches
Expressing with Goto Code

- C allows `goto` as means of transferring control (jump)
  - Closer to assembly programming style
  - Generally considered bad coding style

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

```c
long absdiff_j(long x, long y) {
    long result;
    int ntest = (x <= y);
    if (ntest) goto Else;
    result = x-y;
    goto Done;
    Else:
        result = y-x;
    Done:
    return result;
}
```
Compiling Loops

- Other loops compiled similarly
  - Will show variations and complications in coming slides, but may skip a few examples in the interest of time

- Most important to consider:
  - When should conditionals be evaluated? (while vs. do-while)
  - How much jumping is involved?

C/Java code:

```java
while ( sum != 0 ) {
    <loop body>
}
```

Assembly code:

```assembly
loopTop:    testq  %rax, %rax
            je     loopDone
            <loop body code>
            jmp   loopTop
loopDone:
```

```
Compiling Loops

C/Java code:

```c
while ( Test ) {
    Body
}
```

Goto version

```c
Loop: if (!Test) goto Exit;
Body
goto Loop;
Exit:
```

What are the Goto versions of the following?

- Do...while: Test and Body
- For loop: Init, Test, Update, and Body
Compiling Loops

While Loop:
C:
while ( sum != 0 ) {
   <loop body>
}

Do-while Loop:
C:
do {
   <loop body>
} while ( sum != 0 )

While Loop (ver. 2):
C:
while ( sum != 0 ) {
   <loop body>
}

x86-64:

loopTop:  testq %rax, %rax
       je   loopDone
       jmp  loopTop

loopDone:

all jump instructions
update the program counter (rip)

Do-while Loop (ver. 2):
C:

while ( sum != 0 ) {
   <loop body code>
}

x86-64:

loopTop:  testq %rax, %rax
         jne  loopTop
         jmp  loopTop

loopDone:

x86-64:

loopTop:  testq %rax, %rax
         je  loopDone

loopDone:

x86-64:

loopTop:  testq %rax, %rax
         jne  loopTop

loopDone:
For-Loop → While-Loop

For-Loop:

```java
for (Init; Test; Update) {
    Body
}
```

While-Loop Version:

```java
Init;
while (Test) {
    Body
    Update;
}
```

Caveat: C and Java have `break` and `continue`

- Conversion works fine for `break`
  - Jump to same label as loop exit condition
- But not `continue`: would skip doing `Update`, which it should do with for-loops
  - Introduce new label at `Update`
x86 Control Flow

- Condition codes
- Conditional and unconditional branches
- Loops
- **Switches**
Switch Statement
Example

- Multiple case labels
  - Here: 5 & 6

- Fall through cases
  - Here: 2

- Missing cases
  - Here: 4

- Implemented with:
  - Jump table
  - Indirect jump instruction
Jump Table Structure

Switch Form

```java
switch (x) {
    case val_0:
        Block 0
    case val_1:
        Block 1
        ...
    case val_n-1:
        Block n-1
}
```

Approximate Translation

```java
target = JTab[x];
goto target;
```

Jump Table

- `JTab`: address of jump table
- `Targ0`
- `Targ1`
- `Targ2`
- `...`
- `Targn-1`

Jump Targets

- `Targ0`: Code Block 0
- `Targ1`: Code Block 1
- `Targ2`: Code Block 2
- `...`
- `Targn-1`: Code Block n–1

Notes:

- Addresses are 8 bytes wide.
- The jump table is like an array of pointers.
Jump Table Structure

C code:

```
switch (x) {
    case 1: <some code>
        break;
    case 2: <some code>
    case 3: <some code>
        break;
    case 5:
    case 6: <some code>
        break;
    default: <some code>
}
```

Use the jump table when $x \leq 6$:

```
if (x <= 6)
    target = JTab[x];
    goto target;
else
    goto default;
```
Switch Statement Example

```c
long switch_ex(long x, long y, long z) {
    long w = 1;
    switch (x) {
        . . .
    }
    return w;
}
```

Register Use(s)
- `%rdi` 1st argument (x)
- `%rsi` 2nd argument (y)
- `%rdx` 3rd argument (z)
- `%rax` return value

Note compiler chose to not initialize `w`

Switch_eg:
- `movq %rdx, %rcx`
- `cmpq $6, %rdi`  # x:6
- `ja .L8`  # default
- `jmp *.L4(,%rdi,8)`  # jump table

Jump to default case if x > 6 (unsigned)

Jump above – unsigned > catches negative default cases

Take a look!
https://godbolt.org/z/dOWSFR

-1 > 6U -> jump to default case
Switch Statement Example

```c
long switch_ex(long x, long y, long z) {
    long w = 1;
    switch (x) {
    ...
    }
    return w;
}
```

Jump table

```
.section .rodata
.align 8
.L4:
    .quad .L8  # x = 0
    .quad .L3  # x = 1
    .quad .L5  # x = 2
    .quad .L9  # x = 3
    .quad .L8  # x = 4
    .quad .L7  # x = 5
    .quad .L7  # x = 6
```

Indirect jump

```
switch_eg:
    movq %rdx, %rcx
    cmpq $6, %rdi  # x:6
    ja .L8  # default
    jmp *.L4(,%rdi,8)  # jump table
```
Assembly Setup Explanation

- **Table Structure**
  - Each target requires 8 bytes (address)
  - Base address at `.L4`

- **Direct jump**: `jmp .L8`
  - Jump target is denoted by label `.L8`

- **Indirect jump**: `jmp *.L4(,%rdi,8)`
  - Start of jump table: `.L4`
  - Must scale by factor of 8 (addresses are 8 bytes)
  - Fetch target from effective address `.L4 + x*8`
    - Only for $0 \leq x \leq 6$

Jump table

```
.section .rodata
.align 8
.L4:
.quad .L8  # x = 0
.quad .L3  # x = 1
.quad .L5  # x = 2
.quad .L9  # x = 3
.quad .L8  # x = 4
.quad .L7  # x = 5
.quad .L7  # x = 6
```
Jump Table

declaring data, not instructions

Jump table

this data is 64-bits wide

8-byte memory alignment

```
.switch(x) {
    case 1:      // .L3
        w = y*z;
        break;
    case 2:      // .L5
        w = y/z;
        /* Fall Through */
    case 3:      // .L9
        w += z;
        break;
    case 5:      
    case 6:      // .L7
        w -= z;
        break;
    default:     // .L8
        w = 2;
}
```
Code Blocks (x == 1)

```
switch(x) {
    case 1:    // .L3
        w = y*z;
        break;
    ... 
}
```

<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>
<tr>
<td>%rax</td>
<td>Return value</td>
</tr>
</tbody>
</table>

```
.L3:
    movq   %rsi, %rax  # y
    imulq  %rdx, %rax  # y*z
    ret
```
Handling Fall-Through

```c
long w = 1;
   . . .
switch (x) {
   . . .
case 2: // .L5
   w = y/z;
   /* Fall Through */
case 3: // .L9
   w += z;
   break;
   . . .
}
```

More complicated choice than “just fall-through” forced by “migration” of `w = 1;`

- Example compilation trade-off

```c
case 2:
   w = y/z;
   goto merge;
merge:
   w += z;
case 3:
   w = 1;
```
Code Blocks (x == 2, x == 3)

```c
long w = 1;
  ...
switch (x) {
  ...
  case 2:  // .L5
    w = y/z;
    /* Fall Through */
  case 3:  // .L9
    w += z;
    break;
  ...
}
```

<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>%rdx</td>
<td>3\textsuperscript{rd} argument (z)</td>
</tr>
<tr>
<td>%rax</td>
<td>Return value</td>
</tr>
</tbody>
</table>

```
.L5:
  # Case 2:
  movq %rsi, %rax  # y in rax
  cqto            # Div prep
  idivq %rcx      # y/z
  jmp .L6         # goto merge
.L9:
  # Case 3:
  movl $1, %eax   # w = 1
.L6:
  # merge:
  addq %rcx, %rax # w += z
ret
```
### Code Blocks (rest)

```c
switch (x) {
    ... 
    case 5:  // .L7
        w -= z;
        break;
    case 6:  // .L7
        w -= z;
        break;
    default: // .L8
        w = 2;
}
```

<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>
<tr>
<td>%rax</td>
<td>Return value</td>
</tr>
</tbody>
</table>

```asm
.L7:       # Case 5,6:  
movl $1, %eax  # w = 1
subq %rdx, %rax # w -= z
ret
.L8:       # Default: 
movl $2, %eax  # 2
ret
```