Instructor:
Justin Hsia

Teaching Assistants:
Adithi Raghavan
Aman Mohammed
Connie Chen
Eyoel Gebre
Jiawei Huang
Malak Zaki
Naama Amiel
Nathan Khuat
Nikolas McNamee
Pedro Amarante
Will Robertson

http://xkcd.com/571/
Relevant Course Information

❖ Lab 1a regrade requests open on Gradescope
❖ Lab 1b submissions close tonight
❖ Lab 2 due next Friday (2/2)
  ▪ Section tomorrow on to help prep you for Lab 2 – use the midterm reference sheet & bring your laptop!
  ▪ Optional GDB Tutorial in Ed Lessons

❖ Midterm (take home, 2/8–2/10)
  ▪ Make notes and use the midterm reference sheet
  ▪ Form study groups and look at past exams!
Extra Credit

❖ All labs starting with Lab 2 have extra credit portions
   ▪ These are meant to be fun extensions to the labs

❖ Extra credit points *don't* affect your lab grades
   ▪ From the course policies: “they will be accumulated over the course and will be used to bump up borderline grades at the end of the quarter.”
   ▪ Make sure you finish the rest of the lab before attempting any extra credit
Lesson Summary (1/3)

❖ **Labels** (e.g., `main`, `.L0`) refer to an instruction address and used as jump targets in assembly

❖ Control flow in x86 determined by Condition Codes

- **Set instructions** (`set*`)
  - Read out flag values (0/1)

- **Jump instructions** (`j*`)
  - Use flag values to determine next instruction to execute

- Result of 1st instruction gets compared against 0 in a way determined by 2nd instruction:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Test</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>je</code></td>
<td><code>sete</code></td>
<td>“Equal”</td>
</tr>
<tr>
<td><code>jne</code></td>
<td><code>setne</code></td>
<td>“Not equal”</td>
</tr>
<tr>
<td><code>js</code></td>
<td><code>sets</code></td>
<td>“Signed” (negative)</td>
</tr>
<tr>
<td><code>jns</code></td>
<td><code>setns</code></td>
<td>“Not signed” (nonnegative)</td>
</tr>
<tr>
<td><code>jg</code></td>
<td><code>setg</code></td>
<td>“Greater”</td>
</tr>
<tr>
<td><code>jge</code></td>
<td><code>setge</code></td>
<td>“Greater or equal”</td>
</tr>
<tr>
<td><code>jl</code></td>
<td><code>setl</code></td>
<td>“Less”</td>
</tr>
<tr>
<td><code>jle</code></td>
<td><code>setle</code></td>
<td>“Less or equal”</td>
</tr>
<tr>
<td><code>ja</code></td>
<td><code>seta</code></td>
<td>“Above” (unsigned &gt;)</td>
</tr>
<tr>
<td><code>jb</code></td>
<td><code>setb</code></td>
<td>“Below” (unsigned &lt;)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(op) s, d</th>
<th>cmp a, b</th>
<th>test a, b</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>d</code> (op) s == 0</td>
<td>b-a == 0</td>
<td>b&amp;a == 0</td>
</tr>
<tr>
<td><code>d</code> (op) s != 0</td>
<td>b-a != 0</td>
<td>b&amp;a != 0</td>
</tr>
<tr>
<td><code>d</code> (op) s &lt; 0</td>
<td>b-a &lt; 0</td>
<td>b&amp;a &lt; 0</td>
</tr>
<tr>
<td><code>d</code> (op) s &gt;= 0</td>
<td>b-a &gt;= 0</td>
<td>b&amp;a &gt;= 0</td>
</tr>
<tr>
<td><code>d</code> (op) s &gt; 0</td>
<td>b-a &gt; 0</td>
<td>b&amp;a &gt; 0</td>
</tr>
<tr>
<td><code>d</code> (op) s &lt;= 0</td>
<td>b-a &lt;= 0</td>
<td>b&amp;a &lt;= 0</td>
</tr>
<tr>
<td><code>d</code> (op) s &gt; 0U</td>
<td>b &gt;_U a</td>
<td>b&amp;a &gt; 0U</td>
</tr>
<tr>
<td><code>d</code> (op) s &lt; 0U</td>
<td>b &lt;_U a</td>
<td>b&amp;a &lt; 0U</td>
</tr>
</tbody>
</table>
Lesson Summary (2/3)

❖ Most control flow constructs (e.g., if-else, for-loop, while-loop) can be implemented in assembly using combinations of conditional and unconditional jumps

- Differences come from placement of jumps and whether they jump forward or backwards in code

```
if (<test>)
  <then>
else
  <else>

<init>  while (<test>) {
  <loop body>
  <update>
}
```

```
do {
  <body>
} while (<test>)
```

```
while (<test>) {
  <body>
}
```

```
if (<test>)
  <then>
else
  <else>
do {
  <body>
} while (<test>)
```

```
while (<test>) {
  <body>
}
```
Lesson Summary (3/3)

❖ Switch statements can be implemented using *jump tables* and *indirect jump instructions*
  - Jump tables are arrays of pointers to code blocks
  - Indirect jump jumps to address stored somewhere in memory instead of target specified in instruction

```
switch (x) {
  case 1: <code> break;
  case 2: <code>
  case 3: <code> break;
  case 5: 
  case 6: <code> break;
  case 7: <code> break;
  default: <code>
}
```

```
cmpq $7, %rdi
ja .L9    # default
jmp *.L4(%rdi,8)
```

```
.section .rodata
.align 8
.L4:
.quad .L9    # x = 0
.quad .L8    # x = 1
.quad .L7    # x = 2
.quad .L10   # x = 3
.quad .L9    # x = 4
.quad .L5    # x = 5
.quad .L5    # x = 6
.quad .L3    # x = 7
```
Lesson Q&A

❖ Learning Objectives:
  ▪ Without executing, describe the overall purpose of snippets of x86-64 assembly code containing arithmetic, if-else statements, and/or loops.

❖ What lingering questions do you have from the lesson?
  ▪ Chat with your neighbors about the lesson for a few minutes to come up with questions
x86-64 Programming III – Practice
Polling Question (1/2)

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

A. cmpq %rsi, %rdi
   jle .L4
B. cmpq %rsi, %rdi
   jg .L4
C. testq %rsi, %rdi
   jle .L4
D. testq %rsi, %rdi
   jg .L4

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
    ret
```
Polling Question (2/2)

❖ The following is assembly code for a for-loop; identify the corresponding parts (Init, Test, Update)

- $i \to \%eax$, $x \to \%rdi$, $y \to \%esi$

```assembly
for(______; ______; ______) {
    movl $0, %eax
    .L2:
    cmpl %esi, %eax
    jge .L4
    movslq %eax, %rdx
    leaq (%rdi,%rdx,4), %rcx
    movl (%rcx), %edx
    addl $1, %edx
    movl %edx, (%rcx)
    addl $1, %eax
    jmp .L2
    .L4:
}
```

Labels & Jumps in C (goto)

- 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;
}
```
Labels & Jumps in C (goto)

❖ C allows goto as means of transferring control (jump)
  ▪ Closer to assembly programming style
  ▪ Generally considered bad coding style... listen to Kernighan & Ritchie:

3.8 Goto and Labels

C provides the infinitely-abusable goto statement, and labels to branch to. Formally, the goto is never necessary, and in practice it is almost always easy to write code without it. We have not used goto in this book.

Nevertheless, there are a few situations where gotos may find a place. The most common is to abandon processing in some deeply nested structure, such as breaking out of two or more loops at once. The break statement cannot be used directly since it only exits from the innermost loop. Thus:
# Mainstream ISAs, Revisited

## intel

<table>
<thead>
<tr>
<th>x86</th>
<th>ARM</th>
<th>RISC-V</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Designer</strong></td>
<td>Intel, AMD</td>
<td>Arm Holdings</td>
</tr>
<tr>
<td><strong>Bits</strong></td>
<td>16-bit, 32-bit and 64-bit</td>
<td>32-bit, 64-bit</td>
</tr>
<tr>
<td><strong>Introduced</strong></td>
<td>1978 (16-bit), 1985 (32-bit), 2003 (64-bit)</td>
<td>1985</td>
</tr>
<tr>
<td><strong>Design</strong></td>
<td>CISC</td>
<td>RISC</td>
</tr>
<tr>
<td><strong>Type</strong></td>
<td>Register–memory</td>
<td>Register–Register</td>
</tr>
<tr>
<td><strong>Encoding</strong></td>
<td>Variable (1 to 15 bytes)</td>
<td>AArch64/A64 and AArch32/A32 use 32-bit instructions, T32 (Thumb-2) uses mixed 16- and 32-bit instructions; ARMv7 user-space compatibility.[1]</td>
</tr>
<tr>
<td><strong>Branching</strong></td>
<td>Condition code</td>
<td>Condition code, compare and branch</td>
</tr>
<tr>
<td><strong>Endianness</strong></td>
<td>Little</td>
<td>Big (little as default)</td>
</tr>
</tbody>
</table>

## Macbooks & PCs (Core i3, i5, i7, M)

- **x86-64 Instruction Set**

## Smartphone-like devices (iPhone, iPad, Raspberry Pi)

- **ARM Instruction Set**

## Mostly research (some traction in embedded)

- **RISC-V Instruction Set**

[^1]: [1]
[^3]: [3]
Discussion Question

❖ Discuss the following question(s) in groups of 3-4 students
  ▪ I will call on a few groups afterwards so please be prepared to share out
  ▪ Be respectful of others’ opinions and experiences

❖ We taught you assembly using x86-64; you didn’t have a choice
  ▪ What are some of the advantages and drawbacks of this choice?
  ▪ What are some possible assumptions we are making about our students or values we are forcing on our students with this choice?
Group Work Time

❖ During this time, you are encouraged to work on the following:
   1) If desired, continue your discussion
   2) Work on the homework problems
   3) Work on the lab (if applicable)

❖ Resources:
   ▪ You can revisit the lesson material
   ▪ Work together in groups and help each other out
   ▪ Course staff will circle around to provide support