Instructor:
Justin Hsia

Teaching Assistants:
Angela Xu
Arjun Narendra
Armin Magness
Assaf Vayner
Carrie Hu
Clare Edmonds
David Dai
Dominick Ta
Effie Zheng
James Froelich
Jenny Peng
Kristina Lansang
Paul Stevans
Renee Ruan
Vincent Xiao

http://xkcd.com/409/
Relevant Course Information

❖ hw7 due Monday, hw8 due Wednesday

❖ Lab 1b due Monday (10/17) at 11:59 pm
  ▪ No major programming restrictions, but should avoid magic numbers by using C macros (#define)
  ▪ For debugging, can use provided utility functions print_binary_short() and print_binary_long()
  ▪ Pay attention to the output of aisle_test and store_test – failed tests will show you actual vs. expected
  ▪ You have late day tokens available
Reading Review

❖ Terminology:

▪ Instruction Set Architecture (ISA): CISC vs. RISC
▪ Instructions: data transfer, arithmetic/logical, control flow
  • Size specifiers: b, w, l, q
▪ Operands: immediates, registers, memory
  • Memory operand: displacement, base register, index register, scale factor

❖ Questions from the Reading?
Review Questions

❖ Assume that the register `%rax` currently holds the value 0x 01 02 03 04 05 06 07 08.

❖ Answer the questions on Ed Lessons about the following instruction (\texttt{<instr> <src> <dst>}): \texttt{xorw $\sim1, \%ax}

- Operation type: logical operation
- Operand types: source: immediate, destination: register
- Operation width: 2 bytes ("word")
- (extra) Result in `%rax`:

\[
0 \times 07 \ 08 \\
\sim 0 \times FFFF \\
\hline
0 \times F8 F7 \Rightarrow %rax: 0x 01 02 03 04 05 06 F8 F7
\]
The Hardware/Software Interface

❖ Topic Group 1: Data
   ▪ Memory, Data, Integers, Floating Point, Arrays, Structs

❖ Topic Group 2: Programs
   ▪ x86-64 Assembly, Procedures, Stacks, Executables

❖ Topic Group 3: Scale & Coherence
   ▪ Caches, Processes, Virtual Memory, Memory Allocation
The Hardware/Software Interface

❖ Topic Group 2: Programs
  ▪ x86-64 Assembly, Procedures, Stacks, Executables

❖ How are programs created and executed on a CPU?
  ▪ How does your source code become something that your computer understands?
  ▪ How does the CPU organize and manipulate local data?
Definitions

❖ **Architecture (ISA):** The parts of a processor design that one needs to understand to write assembly code
  ▪ What is directly visible to software
  ▪ The “contract” or “blueprint” between hardware and software

❖ **Microarchitecture:** Implementation of the architecture
  ▪ CSE/EE 469
Instruction Set Architectures (Review)

- The ISA defines:
  - The system’s **state** (e.g., registers, memory, program counter)
  - The **instructions** the CPU can execute
  - The **effect** that each of these instructions will have on the system state
General ISA Design Decisions

❖ Instructions
- What instructions are available? What do they do?
- How are they encoded?

❖ Registers
- How many registers are there?
- How wide are they?

❖ Memory
- How do you specify a memory location?
Instruction Set Philosophies (Review)

- **Complex Instruction Set Computing (CISC):**
  Add more and more elaborate and specialized instructions as needed
  - Lots of tools for programmers to use, but hardware must be able to handle all instructions
  - x86-64 is CISC, but only a small subset of instructions encountered with Linux programs

- **Reduced Instruction Set Computing (RISC):**
  Keep instruction set small and regular
  - Easier to build fast hardware
  - Let software do the complicated operations by composing simpler ones
## Mainstream ISAs

<table>
<thead>
<tr>
<th>Designer</th>
<th>Intel, AMD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bits</td>
<td>16-bit, 32-bit and 64-bit</td>
</tr>
<tr>
<td>Introduced</td>
<td>1978 (16-bit), 1985 (32-bit), 2003 (64-bit)</td>
</tr>
<tr>
<td>Design</td>
<td>CISC</td>
</tr>
<tr>
<td>Type</td>
<td>Register–memory</td>
</tr>
<tr>
<td>Encoding</td>
<td>Variable (1 to 15 bytes)</td>
</tr>
<tr>
<td>Branching</td>
<td>Condition code</td>
</tr>
<tr>
<td>Endianness</td>
<td>Little</td>
</tr>
</tbody>
</table>

### x86 Instruction Set

- Macbooks & PCs (Core i3, i5, i7, M)
- Smartphone-like devices (iPhone, iPad, Raspberry Pi)
- Mostly research (some traction in embedded)

<table>
<thead>
<tr>
<th>Designer</th>
<th>Arm Holdings</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bits</td>
<td>32-bit, 64-bit</td>
</tr>
<tr>
<td>Introduced</td>
<td>1985</td>
</tr>
<tr>
<td>Design</td>
<td>RISC</td>
</tr>
<tr>
<td>Type</td>
<td>Register-Register</td>
</tr>
<tr>
<td>Encoding</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>Branching</td>
<td>Condition code, compare and branch</td>
</tr>
<tr>
<td>Endianness</td>
<td>Big</td>
</tr>
</tbody>
</table>

### ARM Instruction Set

### RISC-V Instruction Set

- Designer: University of California, Berkeley
- Bits: 32 · 64 · 128
- Introduced: 2010
- Design: RISC
- Type: Load-store
- Encoding: Variable
- Endianness: Little[^1][^2]
Architecture Sits at the Hardware Interface

Source code
Different applications or algorithms

Compiler
Perform optimizations, generate instructions

Architecture
Instruction set

Hardware
Different implementations

C Language

Program A

Program B

Your program

GCC

x86-64

Clang

ARMv8
(AAarch64/A64)

Intel Pentium 4

Intel Core 2

Intel Core i7

AMD Opteron

AMD Athlon

ARM Cortex-A53

Apple A7

we will be using
Writing Assembly Code? In 2022???

❖ Chances are, you’ll never write a program in assembly, but understanding assembly is the key to the machine-level execution model:

▪ Behavior of programs in the presence of bugs
  • When high-level language model breaks down

▪ Tuning program performance
  • Understand optimizations done/not done by the compiler
  • Understanding sources of program inefficiency

▪ Implementing systems software
  • What are the “states” of processes that the OS must manage
  • Using special units (timers, I/O co-processors, etc.) inside processor!

▪ Fighting malicious software
  • Distributed software is in binary form
Assembly Programmer’s View

❖ Programmer-visible state
  ▪ PC: the Program Counter (\%rip in x86-64)
    • Address of next instruction
  ▪ Named registers
    • Together in “register file”
    • Heavily used program data
  ▪ Condition codes
    • Store status information about most recent arithmetic operation
    • Used for conditional branching

❖ Memory
  ▪ Byte-addressable array
  ▪ Code and user data
  ▪ Includes the Stack (for supporting procedures)
x86-64 Assembly “Data Types”

❖ Integral data of 1, 2, 4, or 8 bytes
  ▪ Data values
  ▪ Addresses

❖ Floating point data of 4, 8, 10 or 2x8 or 4x4 or 8x2
  ▪ Different registers for those (e.g., %xmm1, %ymm2)
  ▪ Come from extensions to x86 (SSE, AVX, ...)

❖ No aggregate types such as arrays or structures
  ▪ Just contiguously allocated bytes in memory

❖ Two common syntaxes
  ✔ “AT&T”: used by our course, slides, textbook, gnu tools, ...
  X “Intel”: used by Intel documentation, Intel tools, ...
  ▪ Must know which you’re reading

Not covered In 351
Instruction Types (Review)

1) Transfer data between memory and register
   - *Load* data from memory into register
     - \( \%\text{reg} = \text{Mem}[\text{address}] \)
   - *Store* register data into memory
     - \( \text{Mem}[\text{address}] = \%\text{reg} \)

   **Remember:** Memory is indexed just like an array of bytes!

2) Perform arithmetic operation on register or memory data
   - \( c = a + b; \quad z = x << y; \quad i = h \& g; \)

3) Control flow: what instruction to execute next
   - Unconditional jumps to/from procedures
   - Conditional branches
Instruction Sizes and Operands (Review)

❖ Size specifiers
  ▪ b = 1-byte “byte”, w = 2-byte “word”, l = 4-byte “long word”, q = 8-byte “quad word”
  ▪ Note that due to backwards-compatible support for 8086 programs (16-bit machines!), “word” means 16 bits = 2 bytes in x86 instruction names

❖ Operand types
  ▪ **Immediate:** Constant integer data ($)
  ▪ **Register:** 1 of 16 general-purpose integer registers (%)
  ▪ **Memory:** Consecutive bytes of memory at a computed address (())
What is a Register? (Review)

❖ A location in the CPU that stores a small amount of data, which can be accessed very quickly (once every clock cycle)

❖ Registers have *names*, not *addresses*
  ▪ In assembly, they start with `% (e.g., `%rsi`)

❖ Registers are at the heart of assembly programming
  ▪ They are a precious commodity in all architectures, but *especially* x86 only 16 of them...
x86-64 Integer Registers – 64 bits wide

- Can reference low-order 4 bytes (also low-order 2 & 1 bytes)

<table>
<thead>
<tr>
<th>%rax</th>
<th>%eax</th>
<th>%r8</th>
<th>%r8d</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rbx</td>
<td>%ebx</td>
<td>%r9</td>
<td>%r9d</td>
</tr>
<tr>
<td>%rcx</td>
<td>%ecx</td>
<td>%r10</td>
<td>%r10d</td>
</tr>
<tr>
<td>%rdx</td>
<td>%edx</td>
<td>%r11</td>
<td>%r11d</td>
</tr>
<tr>
<td>%rsi</td>
<td>%esi</td>
<td>%r12</td>
<td>%r12d</td>
</tr>
<tr>
<td>%rdi</td>
<td>%edi</td>
<td>%r13</td>
<td>%r13d</td>
</tr>
<tr>
<td>%rsp</td>
<td>%esp</td>
<td>%r14</td>
<td>%r14d</td>
</tr>
<tr>
<td>%rbp</td>
<td>%ebp</td>
<td>%r15</td>
<td>%r15d</td>
</tr>
</tbody>
</table>

"64-bit names"

"32-bit names"
Some History: IA32 Registers – 32 bits wide

- %eax
- %ecx
- %edx
- %ebx
- %esi
- %edi
- %esp
- %ebp

- %ax
- %cx
- %dx
- %bx
- %si
- %di
- %sp
- %bp

- %ah
- %ch
- %dh
- %bh
- %al
- %cl
- %dl
- %bl

32 bits (same as last slide)

16-bit virtual registers (backwards compatibility)

Name Origin (mostly obsolete)
Memory

❖ Addresses
  ▪ 0x7FFFD024C3DC

❖ Big
  ▪ ~ 8 GiB

❖ Slow
  ▪ ~50-100 ns

❖ Dynamic
  ▪ Can “grow” as needed while program runs

vs. Registers

❖ Addresses
  vs. Names
  ▪ %rdi

❖ Big
  vs. Small
  ▪ (16 x 8 B) = 128 B

❖ Slow
  vs. Fast
  ▪ sub-nanosecond timescale

❖ Dynamic
  vs. Static
  ▪ fixed number in hardware
Moving Data

- General form: `mov_ source, destination`
  - Really more of a “copy” than a “move”
  - Like all instructions, missing letter (\_) is the size specifier
  - Lots of these in typical code
Operand Combinations

Source | Dest | Src, Dest | C Analog
--- | --- | --- | ---

movq

\[
\text{Imm} \quad \begin{cases} \text{Reg} & \text{movq } 0x4, \%rax \\
\text{Mem} & \text{movq } -147, (\%rax) \end{cases} \quad \begin{cases} \text{var}_a = 0x4; \\
\text{*p}_a = -147; \end{cases}
\]

\[
\text{Reg} \quad \begin{cases} \text{Reg} & \text{movq } \%rax, \%rdx \\
\text{Mem} & \text{movq } \%rax, (\%rdx) \end{cases} \quad \begin{cases} \text{var}_d = \text{var}_a; \\
\text{*p}_d = \text{var}_a; \end{cases}
\]

\[
\text{Mem} \quad \text{Reg} \quad \text{movq } (\%rax), \%rdx \quad \text{var}_d = \text{*p}_a;
\]

❖ Cannot do memory-memory transfer with a single instruction

- How would you do it?
  1. Mem → Reg
     movq (%rax), %rdx
  2. Reg → Mem
     movq %rdx, (%rbx)
Some Arithmetic Operations

- Binary (two-operand) Instructions:
  - Maximum of one memory operand
  - Beware argument order!
  - No distinction between signed and unsigned
    - Only arithmetic vs. logical shifts
  
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>Computation</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>addq</td>
<td>src, dst</td>
<td>dst = dst + src</td>
<td>(dst += src)</td>
</tr>
<tr>
<td>subq</td>
<td>src, dst</td>
<td>dst = dst − src</td>
<td></td>
</tr>
<tr>
<td>imulq</td>
<td>src, dst</td>
<td>dst = dst * src</td>
<td>signed mult</td>
</tr>
<tr>
<td>sarq</td>
<td>src, dst</td>
<td>dst = dst &gt;&gt; src</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>shrq</td>
<td>src, dst</td>
<td>dst = dst &gt;&gt; src</td>
<td>Logical</td>
</tr>
<tr>
<td>shlq</td>
<td>src, dst</td>
<td>dst = dst &lt;&lt; src</td>
<td>(same as salq)</td>
</tr>
<tr>
<td>xorq</td>
<td>src, dst</td>
<td>dst = dst ^ src</td>
<td></td>
</tr>
<tr>
<td>andq</td>
<td>src, dst</td>
<td>dst = dst &amp; src</td>
<td></td>
</tr>
<tr>
<td>orq</td>
<td>src, dst</td>
<td>dst = dst</td>
<td></td>
</tr>
</tbody>
</table>
Practice Question

❖ Which of the following are valid implementations of \( rcx = rax + rbx \)?

- \( \text{xorq} \%rax, \%rax \) \( (rax = 0) \)
- \( \text{addq} \%rax, \%rbx, \%rcx \)
- \( \text{addq} \%rbx, \%rcx \)
- \( \text{addq} \%rax, \%rcx \)

- \( \text{movq} \%rax, \%rbx, \%rcx \)
- \( \text{addq} \%rbx, \%rcx \)
- \( rcx = rax + rbx \)

- \( \text{addq} \%rax, \%rcx \)
- \( \text{addq} \%rbx, \%rcx \)
- \( rcx = rax + rbx \)

- \( \text{addq} \%rax, \%rcx \)
- \( \text{addq} \%rbx, \%rcx \)
- \( rcx = rax + rbx \)
Arithmetic Example

```c
long simple_arith(long x, long y)
{
    long t1 = x + y;
    long t2 = t1 * 3;
    return t2;
}
```

<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>

The function `simple_arith`:  
```asm
don't actually need new variables!
addq %rdi, %rsi
imulq $3, %rsi
movq %rsi, %rax
ret  # return
```
Summary

❖ x86-64 is a complex instruction set computing (CISC) architecture

▪ There are 3 types of operands in x86-64
  • Immediate, Register, Memory

▪ There are 3 types of instructions in x86-64
  • Data transfer, Arithmetic, Control Flow